

Transport Area Working Group
Internet-Draft
Updates: 3819 (if approved)
Intended status: Best Current Practice
Expires: September 4, 2014

B. Briscoe
BT
J. Kaippallimalil
Huawei
P. Thaler
Broadcom Corporation
March 03, 2014

Guidelines for Adding Congestion Notification to Protocols that
Encapsulate IP
draft-briscoe-tsvwg-ecn-encap-guidelines-04

Abstract

The purpose of this document is to guide the design of congestion notification in any lower layer or tunnelling protocol that encapsulates IP. The aim is for explicit congestion signals to propagate consistently from lower layer protocols into IP. Then the IP internetwork layer can act as a portability layer to carry congestion notification from non-IP-aware congested nodes up to the transport layer (L4). Following these guidelines should assure interworking between new lower layer congestion notification mechanisms, whether specified by the IETF or other standards bodies.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 4, 2014.

Copyright Notice

Copyright (c) 2014 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Scope	5
2. Terminology	6
3. Modes of Operation	7
3.1. Feed-Forward-and-Up Mode	8
3.2. Feed-Up-and-Forward Mode	9
3.3. Feed-Backward Mode	10
3.4. Null Mode	12
4. Feed-Forward-and-Up Mode: Guidelines for Adding Congestion Notification	12
4.1. IP-in-IP Tunnels with Tightly Coupled Shim Headers	13
4.2. Wire Protocol Design: Indication of ECN Support	13
4.3. Encapsulation Guidelines	15
4.4. Decapsulation Guidelines	17
4.5. Sequences of Similar Tunnels or Subnets	18
4.6. Reframing and Congestion Markings	19
5. Feed-Up-and-Forward Mode: Guidelines for Adding Congestion Notification	19
6. Feed-Backward Mode: Guidelines for Adding Congestion Notification	21
7. IANA Considerations (to be removed by RFC Editor)	22
8. Security Considerations	22
9. Conclusions	22
10. Acknowledgements	23
11. Comments Solicited	23
12. References	23
12.1. Normative References	23
12.2. Informative References	24
Appendix A. Outstanding Document Issues	27
Appendix B. Changes in This Version (to be removed by RFC Editor)	27

1. Introduction

The benefits of Explicit Congestion Notification (ECN) described below can only be fully realised if support for ECN is added to the relevant subnetwork technology, as well as to IP. When a lower layer buffer drops a packet obviously it does not just drop at that layer; the packet disappears from all layers. In contrast, when a lower layer marks a packet with ECN, the marking needs to be explicitly propagated up the layers. The same is true if a buffer marks the outer header of a packet that encapsulates inner tunnelled headers. Forwarding ECN is not as straightforward as other headers because it has to be assumed ECN may be only partially deployed. If an egress at any layer is not ECN-aware, or if the ultimate receiver or sender is not ECN-aware, congestion needs to be indicated by dropping a packet, not marking it.

The purpose of this document is to guide the addition of congestion notification to any subnet technology or tunnelling protocol, so that lower layer equipment can signal congestion explicitly and it will propagate consistently into encapsulated (higher layer) headers, otherwise the signals will not reach their ultimate destination.

ECN is defined in the IP header (v4 & v6) [RFC3168] to allow a resource to notify the onset of queue build-up without having to drop packets, by explicitly marking a proportion of packets with the congestion experienced (CE) codepoint.

Given a suitable marking scheme, ECN removes nearly all congestion loss and it cuts delays for two main reasons:

- o It avoids the delay when recovering from congestion losses, which particularly benefits small flows or real-time flows, making their delivery time predictably short [RFC2884];
- o As ECN is used more widely by end-systems, it will gradually remove the need to configure a degree of delay into buffers before they start to notify congestion (the cause of bufferbloat). This is because drop involves a trade-off between sending a timely signal and trying to avoid impairment, whereas ECN is solely a signal not an impairment, so there is no harm triggering it earlier.

Some lower layer technologies (e.g. MPLS, Ethernet) are used to form subnetworks with IP-aware nodes only at the edges. These networks are often sized so that it is rare for interior queues to overflow. However, this has often been more due to the inability of the original TCP protocol to saturate the links. For many years, fixes such as window scaling [RFC1323] proved hard to deploy. But now that modern

operating systems are finally capable of saturating interior links, even the buffers of well-provisioned interior switches will need to signal episodes of queuing.

Propagation of ECN is defined for MPLS [RFC5129], and is being defined for TRILL [trill-rbridge-options], but it remains to be defined for a number of other subnetwork technologies.

Similarly, ECN propagation is yet to be defined for many tunnelling protocols. [RFC6040] defines how ECN should be propagated for IP-in-IP [RFC2003] and IPsec [RFC4301] tunnels. However, as Section 9.3 of RFC3168 pointed out, ECN support will need to be defined for other tunnelling protocols, e.g. L2TP [RFC2661], GRE [RFC1701], [RFC2784], PPTP [RFC2637] and GTP [GTPv1], [GTPv1-U], [GTPv2-C].

Incremental deployment is the most tricky aspect when adding support for ECN. The original ECN protocol in IP [RFC3168] was carefully designed so that a congested buffer would not mark a packet (rather than drop it) unless both source and destination hosts were ECN-capable. Otherwise its congestion markings would never be detected and congestion would just deteriorate further. However, to support congestion marking below the IP layer, it is not sufficient to only check that the two end-points support ECN; correct operation also depends on the decapsulator at each subnet egress faithfully propagating congestion notifications to the higher layer. Otherwise, a legacy decapsulator might silently fail to propagate any ECN signals from the outer to the forwarded header. Then the lost signals would never be detected and again congestion would deteriorate further. The guidelines given later require protocol designers to carefully consider incremental deployment, and suggest various safe approaches for different circumstances.

Of course, the IETF does not have standards authority over every link layer protocol. So this document gives guidelines for designing propagation of congestion notification across the interface between IP and protocols that may encapsulate IP (i.e. that can be layered beneath IP). Each lower layer technology will exhibit different issues and compromises, so the IETF or the relevant standards body must be free to define the specifics of each lower layer congestion notification scheme. Nonetheless, if the guidelines are followed, congestion notification should interwork between different technologies, using IP in its role as a 'portability layer'.

Therefore, the capitalised term 'SHOULD' or 'SHOULD NOT' are often used in preference to 'MUST' or 'MUST NOT', because it is difficult to know the compromises that will be necessary in each protocol design. If a particular protocol design chooses to contradict a

'SHOULD (NOT)' given in the advice below, it MUST include a sound justification.

It has not been possible to give common guidelines for all lower layer technologies, because they do not all fit a common pattern. Instead they have been divided into a few distinct modes of operation: feed-forward-and-upward; feed-upward-and-forward; feed-backward; and null mode. These modes are described in Section 3, then in the following sections separate guidelines are given for each mode.

This document updates the advice to subnetwork designers about ECN in Section 13 of [RFC3819].

1.1. Scope

This document only concerns wire protocol processing of explicit notification of congestion and makes no changes or recommendations concerning algorithms for congestion marking or for congestion response (algorithm issues should be independent of the layer the algorithm operates in).

The question of congestion notification signals with different semantics to those of ECN in IP is touched on in a couple of specific cases (e.g. QCN [IEEE802.1Qau]) and with schemes with multiple severity levels such as PCN [RFC6660]). However, no attempt is made to give guidelines about schemes with different semantics that are yet to be invented.

The semantics of congestion signals can be relative to the traffic class. Therefore correct propagation of congestion signals could depend on correct propagation of any traffic class field between the layers. In this document, correct propagation of traffic class information is assumed, while what 'correct' means and how it is achieved is covered elsewhere (e.g. [RFC2983]) and is outside the scope of the present document.

Note that these guidelines do not require the subnet wire protocol to be changed to accommodate congestion notification. Another way to add congestion notification without consuming header space in the subnet protocol might be to use a parallel control plane protocol.

This document focuses on the congestion notification interface between IP and lower layer protocols that can encapsulate IP, where the term 'IP' includes v4 or v6, unicast, multicast or anycast. However, it is likely that the guidelines will also be useful when a lower layer protocol or tunnel encapsulates itself (e.g. Ethernet MAC in MAC [IEEE802.1Qah]) or when it encapsulates other protocols. In

the feed-backward mode, propagation of congestion signals for multicast and anycast packets is out-of-scope (because it would be so complicated that it is hoped no-one would attempt such an abomination).

2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

Further terminology used within this document:

Protocol data unit (PDU): Information that is delivered as a unit among peer entities of a layered network consisting of protocol control information (typically a header) and possibly user data (payload) of that layer. The scope of this document includes layer 2 and layer 3 networks, where the PDU is respectively termed a frame or a packet (or a cell in ATM). PDU is a general term for any of these. This definition also includes a payload with a shim header lying somewhere between layer 2 & 3.

Transport: The end-to-end transmission control function, conventionally considered at layer-4 in the OSI reference model. Given the audience for this document will often use the word transport to mean low level bit carriage, whenever the term is used it will be qualified, e.g. 'L4 transport'.

Encapsulator: The link or tunnel endpoint function that adds an outer header to a PDU (also termed the 'link ingress', the 'subnet ingress', the 'ingress tunnel endpoint' or just the 'ingress' where the context is clear).

Decapsulator: The link or tunnel endpoint function that removes an outer header from a PDU (also termed the 'link egress', the 'subnet egress', the 'egress tunnel endpoint' or just the 'egress' where the context is clear).

Incoming header: The header of an arriving PDU before encapsulation.

Outer header: The header added to encapsulate a PDU.

Inner header: The header encapsulated by the outer header.

Outgoing header: The header forwarded by the decapsulator.

CE: Congestion Experienced [RFC3168]

ECT: ECN-Capable Transport [RFC3168]

Not-ECT: Not ECN-Capable Transport [RFC3168]

ECN-PDU: A PDU that is part of a feedback loop within which all the nodes that need to propagate explicit congestion notifications back to the Load Regulator are ECN-capable. An IP packet with a non-zero ECN field implies that the endpoints are ECN-capable, so this would be an ECN-PDU. However, ECN-PDU is intended to be a general term for a PDU at any layer, not just IP.

Not-ECN-PDU: A PDU that is part of a feedback-loop within which some nodes necessary to propagate explicit congestion notifications back to the load regulator are not ECN-capable.

Load Regulator: For each flow of PDUs, the transport function that is capable of controlling the data rate. Typically located at the data source, but in-path nodes can regulate load in some congestion control arrangements (e.g. admission control or policing nodes). Note the term "a function capable of controlling the load" deliberately includes a transport that doesn't actually control the load but ideally it ought to (e.g. a sending application without congestion control that uses UDP).

Congestion Baseline: The location of the function on the path that initialised the values of all congestion notification fields in a sequence of packets, before any are set to the congestion experienced (CE) codepoint if they experience congestion further downstream. Typically the original data source at layer-4.

3. Modes of Operation

This section sets down the different modes by which congestion information is passed between the lower layer and the higher one. It acts as a reference framework for the following sections, which give normative guidelines for designers of explicit congestion notification protocols, taking each mode in turn:

Feed-Forward-and-Up: Nodes feed forward congestion notification towards the egress within the lower layer then up and along the layers towards the end-to-end destination at the transport layer. The following local optimisation is possible:

Feed-Up-and-Forward: A lower layer switch feeds-up congestion notification directly into the ECN field in the higher layer (e.g. IP) header, irrespective of whether the node is at the egress of a subnet.

Feed-Backward: Nodes feed back congestion signals towards the ingress of the lower layer and (optionally) attempt to control congestion within their own layer.

Null: Nodes cannot experience congestion at the lower layer except at ingress nodes (which are IP-aware or equivalently higher-layer-aware).

3.1. Feed-Forward-and-Up Mode

Like IP and MPLS, many subnet technologies are based on self-contained protocol data units (PDUs) or frames sent unreliably. They provide no feedback channel at the subnetwork layer, instead relying on higher layers (e.g. TCP) to feed back loss signals.

In these cases, ECN may best be supported by standardising explicit notification of congestion into the lower layer protocol that carries the data forwards. It will then also be necessary to define how the egress of the lower layer subnet propagates this explicit signal into the forwarded upper layer (IP) header. It can then continue forwards until it finally reaches the destination transport (at L4). Then typically the destination will feed this congestion notification back to the source transport using an end-to-end protocol (e.g. TCP). This is the arrangement that has already been used to add ECN to IP-in-IP tunnels [RFC6040], IP-in-MPLS and MPLS-in-MPLS [RFC5129].

This mode is illustrated in Figure 1. Along the middle of the figure, layers 2, 3 & 4 of the protocol stack are shown, and one packet is shown along the bottom as it progresses across the network from source to destination, crossing two subnets connected by a router, and crossing two switches on the path across each subnet. Congestion at the output of the first switch (shown as *) leads to a congestion marking in the L2 header (shown as C in the illustration of the packet). The chevrons show the progress of the resulting congestion indication. It is propagated from link to link across the subnet in the L2 header, then when the router removes the marked L2 header, it propagates the marking up into the L3 (IP) header. The router forwards the marked L3 header into subnet 2, and when it adds a new L2 header it copies the L3 marking into the L2 header as well, as shown by the 'C's in both layers (assuming the technology of subnet 2 also supports explicit congestion marking).

Note that there is no implication that each 'C' marking is encoded the same; a different encoding might be used for the 'C' marking in each protocol.

Finally, for completeness, we show the L3 marking arriving at the destination, where the host transport protocol (e.g. TCP) feeds it

back to the source in the L4 acknowledgement (the 'C' at L4 in the packet at the top of the diagram).

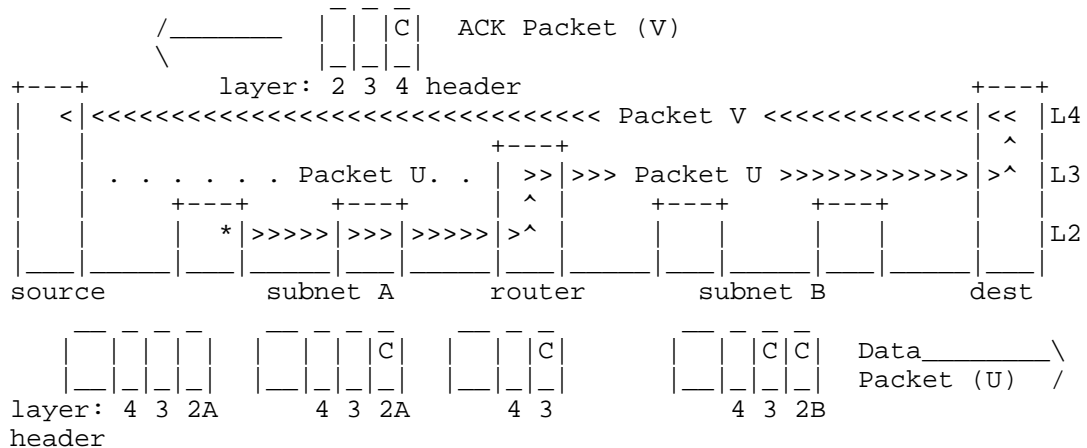


Figure 1: Feed-Forward-and-Up Mode

Of course, modern networks are rarely as simple as this text-book example, often involving multiple nested layers. For example, a 3GPP mobile network may have two IP-in-IP (GTP) tunnels in series and an MPLS backhaul between the base station and the first router. Nonetheless, the example illustrates the general idea of feeding congestion notification forward then upward whenever a header is removed at the egress of a subnet.

Note that the FECN (forward ECN) bit in Frame Relay and the explicit forward congestion indication (EFCI [ITU-T.I.371]) bit in ATM user data cells follow a feed-forward pattern. However, in ATM, this is only as part of a feed-forward-and-backward pattern at the lower layer, not feed-forward-and-up out of the lower layer--the intention was never to interface to IP ECN at the subnet egress. To our knowledge, Frame Relay FECN is solely used to detect where more capacity should be provisioned [Buck00].

3.2. Feed-Up-and-Forward Mode

Ethernet is particularly difficult to extend incrementally to support explicit congestion notification. One way to support ECN in such cases has been to use so called 'layer-3 switches'. These are Ethernet switches that bury into the Ethernet payload to find an IP header and manipulate or act on certain IP fields (specifically Diffserv & ECN). For instance, in Data Center TCP [DCTCP], layer-3 switches are configured to mark the ECN field of the IP header within

the Ethernet payload when their output buffer becomes congested. With respect to switching, a layer-3 switch acts solely on the addresses in the Ethernet header; it doesn't use IP addresses, and it doesn't decrement the TTL field in the IP header.

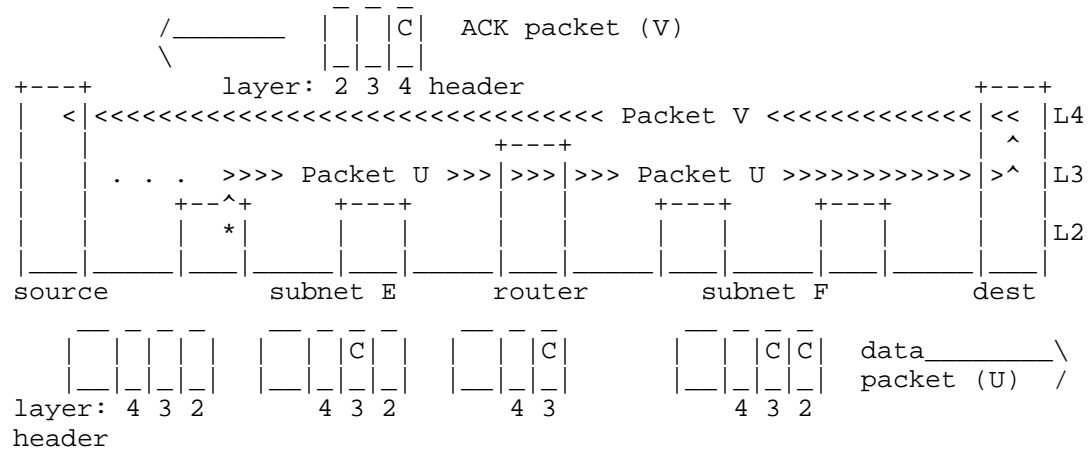


Figure 2: Feed-Up-and-Forward Mode

By comparing Figure 2 with Figure 1, it can be seen that subnet E (perhaps a subnet of layer-3 Ethernet switches) works in feed-up-and-forward mode by notifying congestion directly into L3 at the point of congestion, even though the congested switch does not otherwise act at L3. In this example, the technology in subnet F (e.g. MPLS) does support ECN natively, so when the router adds the layer-2 header it copies the ECN marking from L3 to L2 as well.

3.3. Feed-Backward Mode

In some layer 2 technologies, explicit congestion notification has been defined for use internally within the subnet with its own feedback and load regulation, but typically the interface with IP for ECN has not been defined.

For instance, for the available bit-rate (ABR) service in ATM, the relative rate mechanism was one of the more popular mechanisms for managing traffic, tending to supersede earlier designs. In this approach ATM switches send special resource management (RM) cells in both the forward and backward directions to control the ingress rate of user data into a virtual circuit. If a switch buffer is approaching congestion or congested it sends an RM cell back towards the ingress with respectively the No Increase (NI) or Congestion

Indication (CI) bit set in its message type field [ATM-TM-ABR]. The ingress then holds or decreases its sending bit-rate accordingly.

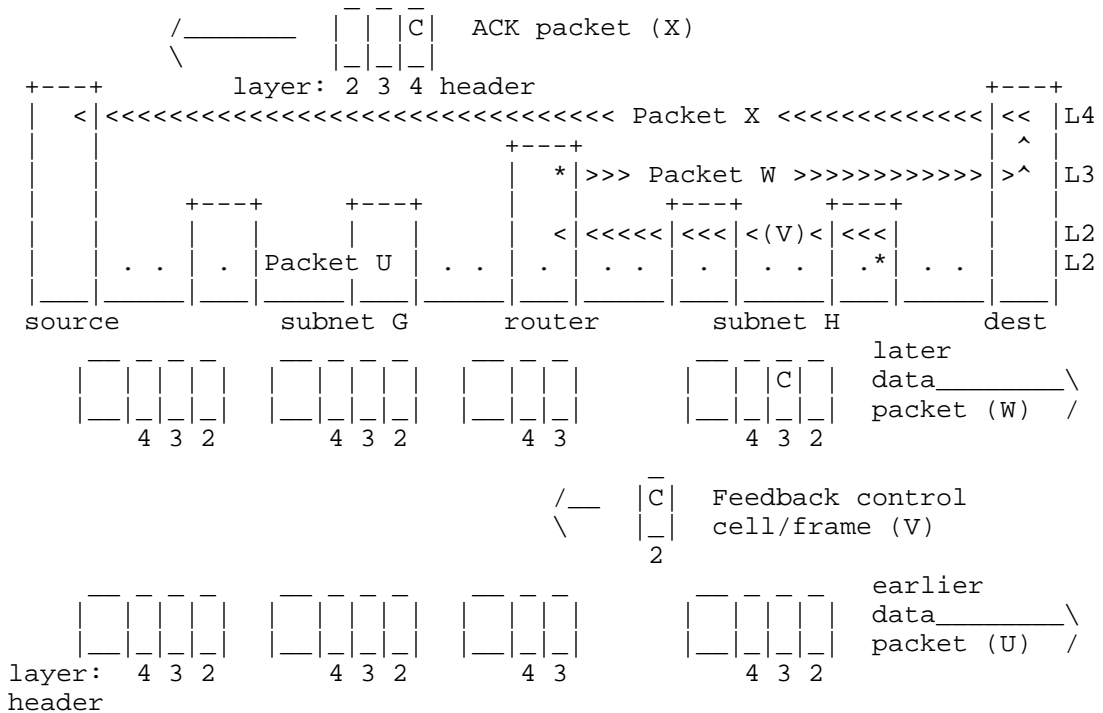


Figure 3: Feed-Backward Mode

ATM's feed-backward approach doesn't fit well when layered beneath IP's feed-forward approach--unless the initial data source is the same node as the ATM ingress. Figure 3 shows the feed-backward approach being used in subnet H. If the final switch on the path is congested (*), it doesn't feed-forward any congestion indications on packet (U). Instead it sends a control cell (V) back to the router at the ATM ingress.

However, the backward feedback doesn't reach the original data source directly because IP doesn't support backward feedback (and subnet G is independent of subnet H). Instead, the router in the middle throttles down its sending rate but the original data sources don't reduce their rates. The resulting rate mismatch causes the middle router's buffer at layer 3 to back up until it becomes congested, which it signals forwards on later data packets at layer 3 (e.g. packet W). Note that the forward signal from the middle router is not triggered directly by the backward signal. Rather, it is

triggered by congestion resulting from the middle router's mismatched rate response to the backward signal.

In response to this later forward signalling, end-to-end feedback at layer-4 finally completes the tortuous path of congestion indications back to the origin data source, as before.

3.4. Null Mode

Often link and physical layer resources are 'non-blocking' by design. In these cases congestion notification may be implemented but it does not need to be deployed at the lower layer; ECN in IP would be sufficient.

A degenerate example is a point-to-point Ethernet link. Excess loading of the link merely causes the queue from the higher layer to back up, while the lower layer remains immune to congestion. Even a whole meshed subnetwork can be made immune to interior congestion by limiting ingress capacity and careful sizing of links, particularly if multi-path routing is used to ensure even worst-case patterns of load cannot congest any link.

4. Feed-Forward-and-Up Mode: Guidelines for Adding Congestion Notification

Feed-forward-and-up is the mode already used for signalling ECN up the layers through MPLS into IP [RFC5129] and through IP-in-IP tunnels [RFC6040]. These RFCs take a consistent approach and the following guidelines are designed to ensure this consistency continues as ECN support is added to other protocols that encapsulate IP. The guidelines are also designed to ensure compliance with the more general best current practice for the design of alternate ECN schemes given in [RFC4774].

The rest of this section is structured as follows:

- o Section 4.1 addresses the most straightforward cases, where [RFC6040] can be applied directly to add ECN to tunnels that are effectively the same as IP-in-IP tunnels.
- o The subsequent sections give guidelines for adding ECN to a subnet technology that uses feed-forward-and-up mode like IP, but it is not so similar to IP that [RFC6040] rules can be applied directly. Specifically:
 - * Sections 4.2, 4.3 and 4.4 respectively address how to add ECN support to the wire protocol and to the encapsulators and decapsulators at the ingress and egress of the subnet.

- * Section 4.5 deals with the special, but common, case of sequences of tunnels or subnets that all use the same technology
- * Section 4.6 deals with the question of reframing when IP packets do not map 1:1 into lower layer frames.

4.1. IP-in-IP Tunnels with Tightly Coupled Shim Headers

A common pattern for many tunnelling protocols is to encapsulate an inner IP header with shim header(s) then an outer IP header. In many cases the shim header(s) always have to be tightly coupled to the outer IP header because they are not sufficient as outer headers in their own right. In such cases the shim header(s) and the outer IP header are always added (or removed) in the same operation. Therefore, in all such tightly coupled IP-in-IP tunnelling protocols, the rules in [RFC6040] for propagating the ECN field between the two IP headers SHOULD be applied directly.

Examples of tightly coupled IP-in-IP tunnelling protocols where [RFC6040] can be applied directly are:

- o L2TP [RFC2661]
- o GRE [RFC1701], [RFC2784]
- o PPTP [RFC2637]
- o GTP [GTPv1], [GTPv1-U], [GTPv2-C]
- o VXLAN [vxlan].

4.2. Wire Protocol Design: Indication of ECN Support

This section is intended to guide the redesign of any lower layer protocol that encapsulate IP to add native ECN support at the lower layer. It reflects the approaches used in [RFC6040] and in [RFC5129]. Therefore IP-in-IP tunnels or IP-in-MPLS or MPLS-in-MPLS encapsulations that already comply with [RFC6040] or [RFC5129] will already satisfy this guidance.

A lower layer (or subnet) congestion notification system:

1. SHOULD NOT apply explicit congestion notifications to PDUs that are destined for legacy layer-4 transport implementations that will not understand ECN, and

2. SHOULD NOT apply explicit congestion notifications to PDUs if the egress of the subnet might not propagate congestion notifications onward into the higher layer.

We use the term ECN-PDUs for a PDU on a feedback loop that will propagate congestion notification properly because it meets both the above criteria. And a Not-ECN-PDU is a PDU on a feedback loop that does not meet both criteria, and will therefore not propagate congestion notification properly. A corollary of the above is that a lower layer congestion notification protocol:

3. SHOULD be able to distinguish ECN-PDUs from Not-ECN-PDUs.

Note that there is no need for all interior nodes within a subnet to be able to mark congestion explicitly. A mix of ECN and drop signals from different nodes is fine. However, if any interior nodes might generate ECN markings, guideline 2 above says that all relevant egress node(s) SHOULD be able to propagate those markings up to the higher layer.

In IP, if the ECN field in each PDU is cleared to the Not-ECT (not ECN-capable transport) codepoint, it indicates that the L4 transport will not understand congestion markings. A congested buffer must not mark these Not-ECT PDUs, and therefore drops them instead.

The mechanism a lower layer uses to distinguish the ECN-capability of PDUs need not mimic that of IP. All the above guidelines say is that the lower layer system, as a whole, should achieve the same outcome. For instance, ECN-capable feedback loops might use PDUs that are identified by a particular set of labels or tags. Alternatively, logical link protocols that use flow state might determine whether a PDU can be congestion marked by checking for ECN-support in the flow state. Other protocols might depend on out-of-band control signals.

The per-domain checking of ECN support in MPLS [RFC5129] is a good example of a way to avoid sending congestion markings to transports that will not understand them, without using any header space in the subnet protocol.

In MPLS, header space is extremely limited, therefore RFC5129 does not provide a field in the MPLS header to indicate whether the PDU is an ECN-PDU or a Not-ECN-PDU. Instead, interior nodes in a domain are allowed to set explicit congestion indications without checking whether the PDU is destined for a transport that will understand them. Nonetheless, this is made safe by requiring that the network operator upgrades all decapsulating edges of a whole domain at once, as soon as even one switch within the domain is configured to mark rather than drop during congestion. Therefore, any edge node that

might decapsulate a packet will be capable of checking whether the higher layer transport is ECN-capable. When decapsulating a CE-marked packet, if the decapsulator discovers that the higher layer (inner header) indicates the transport is not ECN-capable, it drops the packet--effectively on behalf of the earlier congested node (see Decapsulation Guideline 1 in Section 4.4).

It was only appropriate to define such an incremental deployment strategy because MPLS is targeted solely at professional operators, who can be expected to ensure that a whole subnetwork is consistently configured. This strategy might not be appropriate for other link technologies targeted at zero-configuration deployment or deployment by the general public (e.g. Ethernet). For such 'plug-and-play' environments it will be necessary to invent a failsafe approach that ensures congestion markings will never fall into black holes, no matter how inconsistently a system is put together. Alternatively, congestion notification relying on correct system configuration could be confined to flavours of Ethernet intended only for professional network operators, such as IEEE 802.1ah Provider Backbone Bridges (PBB).

QCN [IEEE802.1Qau] provides another example of how to indicate to lower layer devices that the end-points will not understand ECN. An operator can define certain 802.1p classes of service to indicate non-QCN frames and an ingress bridge is required to map arriving not-QCN-capable IP packets to one of these non-QCN 802.1p classes.

4.3. Encapsulation Guidelines

This section is intended to guide the redesign of any node that encapsulates IP with a lower layer header when adding native ECN support to the lower layer protocol. It reflects the approaches used in [RFC6040] and in [RFC5129]. Therefore IP-in-IP tunnels or IP-in-MPLS or MPLS-in-MPLS encapsulations that already comply with [RFC6040] or [RFC5129] will already satisfy this guidance.

1. Egress Capability Check: A subnet ingress needs to be sure that the corresponding egress of a subnet will propagate any congestion notification added to the outer header across the subnet. This is necessary in addition to checking that an incoming PDU indicates an ECN-capable (L4) transport. Examples of how this guarantee might be provided include:
 - * by configuration (e.g. if any label switches in a domain support ECN marking, [RFC5129] requires all egress nodes to have been configured to propagate ECN)

- * by the ingress explicitly checking that the egress propagates ECN (e.g. TRILL uses IS-IS to check path capabilities before using critical options [trill-rbridge-options])
 - * by inherent design of the protocol (e.g. by encoding ECN marking on the outer header in such a way that a legacy egress that does not understand ECN will consider the PDU corrupt and discard it, thus at least propagating a form of congestion signal).
2. Egress Fails Capability Check: If the ingress cannot guarantee that the egress will propagate congestion notification, the ingress SHOULD disable ECN when it forwards the PDU at the lower layer. An example of how the ingress might disable ECN at the lower layer would be by setting the outer header of the PDU to identify it as a Not-ECN-PDU, assuming the subnet technology supports such a concept.
 3. Standard Congestion Monitoring Baseline: Once the ingress to a subnet has established that the egress will correctly propagate ECN, on encapsulation it SHOULD encode the same level of congestion in outer headers as is arriving in incoming headers. For example it might copy any incoming congestion notification into the outer header of the lower layer protocol.

This ensures that all outer headers reflect congestion accumulated along the whole upstream path since the Load Regulator, not just since the ingress of the subnet. A node that is not the Load Regulator SHOULD NOT re-initialise the level of CE markings in the outer to zero.

This guideline is intended to ensure that any bulk congestion monitoring of outer headers (e.g. by a network management node monitoring ECN in passing frames) is most meaningful. For instance, if an operator measures CE in 0.4% of passing outer headers, this information is only useful if the operator knows where the proportion of CE markings was last initialised to 0% (the Congestion Baseline). Such monitoring information will not be useful if some subnet ingress nodes reset all outer CE markings while others copy incoming CE markings into the outer.

Most information can be extracted if the Congestion Baseline is standardised at the node that is regulating the load (the Load Regulator--typically the data source). Then the operator can measure both congestion since the Load Regulator, and congestion since the subnet ingress. The latter might be measurable by subtracting the level of CE markings on inner headers from that on outer headers (see Appendix C of [RFC6040]).

4.4. Decapsulation Guidelines

This section is intended to guide the redesign of any node that decapsulates IP from within a lower layer header when adding native ECN support to the lower layer protocol. It reflects the approaches used in [RFC6040] and in [RFC5129]. Therefore IP-in-IP tunnels or IP-in-MPLS or MPLS-in-MPLS encapsulations that already comply with [RFC6040] or [RFC5129] will already satisfy this guidance.

A subnet egress SHOULD NOT simply copy congestion notification from outer headers to the forwarded header. It SHOULD calculate the outgoing congestion notification field from the inner and outer headers using the following guidelines. If there is any conflict, rules earlier in the list take precedence over rules later in the list:

1. If the arriving inner header is a Not-ECN-PDU it implies the L4 transport will not understand explicit congestion markings.
Then:
 - * If the outer header carries an explicit congestion marking, the packet SHOULD be dropped--the only indication of congestion that the L4 transport will understand.
 - * If the outer is an ECN-PDU that carries no indication of congestion or a Not-ECN-PDU the PDU SHOULD be forwarded, but still as a Not-ECN-PDU.
2. If the outer header does not support explicit congestion notification (a Not-ECN-PDU), but the inner header does (an ECN-PDU), the inner header SHOULD be forwarded unchanged.
3. In some lower layer protocols congestion may be signalled as a numerical level, such as in the control frames of quantised congestion notification [IEEE802.1Qau]. If such a multi-bit encoding encapsulates an ECN-capable IP data packet, a function will be needed to convert the quantised congestion level into the frequency of congestion markings in outgoing IP packets.
4. Congestion indications may be encoded by a severity level. For instance increasing levels of congestion might be encoded by numerically increasing indications, e.g. pre-congestion notification (PCN) can be encoded in each PDU at three severity levels in IP or MPLS [RFC6660].

If the arriving inner header is an ECN-PDU, where the inner and outer headers carry indications of congestion of different

severity, the more severe indication SHOULD be forwarded in preference to the less severe.

5. The inner and outer headers might carry a combination of congestion notification fields that should not be possible given any currently used protocol transitions. For instance, if Encapsulation Guideline 3 in Section 4.3 had been followed, it should not be possible to have a less severe indication of congestion in the outer than in the inner. It MAY be appropriate to log unexpected combinations of headers and possibly raise an alarm.

If a safe outgoing codepoint can be defined for such a PDU, the PDU SHOULD be forwarded rather than dropped. Some implementers discard PDUs with currently unused combinations of headers just in case they represent an attack. However, an approach using alarms and policy-mediated drop is preferable to hard-coded drop, so that operators can keep track of possible attacks but currently unused combinations are not precluded from future use through new standards actions.

4.5. Sequences of Similar Tunnels or Subnets

In some deployments, particularly in 3GPP networks, an IP packet may traverse two or more IP-in-IP tunnels in sequence that all use identical technology (e.g. GTP).

In such cases, it would be sufficient for every encapsulation and decapsulation in the chain to comply with RFC 6040. Alternatively, as an optimisation, a node that decapsulates a packet and immediately re-encapsulates it for the next tunnel MAY copy the incoming outer ECN field directly to the outgoing outer and the incoming inner ECN field directly to the outgoing inner. Then the overall behavior across the sequence of tunnel segments would still be consistent with RFC 6040.

Appendix C of RFC6040 describes how a tunnel egress can monitor how much congestion has been introduced within a tunnel. A network operator might want to monitor how much congestion had been introduced within a whole sequence of tunnels. Using the technique in Appendix C of RFC6040 at the final egress, the operator could monitor the whole sequence of tunnels, but only if the above optimisation were used consistently along the sequence of tunnels, in order to make it appear as a single tunnel. Therefore, tunnel endpoint implementations SHOULD allow the operator to configure whether this optimisation is enabled.

When ECN support is added to a subnet technology, consideration SHOULD be given to a similar optimisation between subnets in sequence if they all use the same technology.

4.6. Reframing and Congestion Markings

The guidance in this section is worded in terms of framing boundaries, but it applies equally whether the protocol data units are frames, cells or packets.

Where framing boundaries are different between two layers, congestion indications SHOULD be propagated on the basis that a congestion indication on a PDU applies to all the octets in the PDU. On average, an encapsulator or decapsulator SHOULD approximately preserve the number of marked octets arriving and leaving (counting the size of inner headers, but not added encapsulating headers).

The next departing frame SHOULD be immediately marked even if only enough incoming marked octets have arrived for part of the departing frame. This ensures that any outstanding congestion marked octets are propagated immediately, rather than held back waiting for a frame no bigger than the outstanding marked octets--which might involve a long wait.

For instance, an algorithm for marking departing frames could maintain a counter representing the balance of arriving marked octets minus departing marked octets. It adds the size of every marked frame that arrives and if the counter is positive it marks the next frame to depart and subtracts its size from the counter. This will often leave a negative remainder in the counter, which is deliberate.

5. Feed-Up-and-Forward Mode: Guidelines for Adding Congestion Notification

The guidance in this section is applicable when IP packets:

- o are encapsulated in Ethernet headers;
- o are forwarded by the eNode-B (base station) of a 3GPP radio access network, which is required to apply ECN marking during congestion [LTE-RA].

This guidance also generalises to encapsulation by other subnet technologies with no native support for explicit congestion notification at the lower layer, but with support for finding and processing an IP header. It is unlikely to be applicable or necessary for IP-in-IP encapsulation, where feed-forward-and-up mode based on [RFC6040] would be more appropriate.

Marking the IP header while switching at layer-2 (by using a layer-3 switch) or while forwarding in a radio access network seems to represent a layering violation. However, it can be considered as a benign optimisation if the guidelines below are followed. Feed-up-and-forward is certainly not a general alternative to implementing feed-forward congestion notification in the lower layer, because:

- o IPv4 and IPv6 are not the only layer-3 protocols that might be encapsulated by lower layer protocols
- o Link-layer encryption might be in use, making the layer-2 payload inaccessible
- o Many Ethernet switches do not have 'layer-3 switch' capabilities so they cannot read or modify an IP payload
- o It might be costly to find an IP header (v4 or v6) when it may be encapsulated by more than one lower layer header, e.g. Ethernet MAC in MAC [IEEE802.1Qah].

Nonetheless, configuring lower layer equipment to look for an ECN field in an encapsulated IP header is a useful optimisation. If the implementation follows the guidelines below, this optimisation does not have to be confined to a controlled environment such as within a data centre; it could usefully be applied on any network--even if the operator is not sure whether the above issues will never apply:

1. If a native lower-layer congestion notification mechanism exists for a subnet technology, it is safe to mix feed-up-and-forward with feed-forward-and-up on other switches in the same subnet. However, it will generally be more efficient to use the native mechanism.
2. The depth of the search for an IP header SHOULD be limited. If an IP header is not found soon enough, or an unrecognised or unreadable header is encountered, the switch SHOULD resort to an alternative means of signalling congestion (e.g. drop, or the native lower layer mechanism if available).
3. It is sufficient to use the first IP header found in the stack; the egress of the relevant tunnel can propagate congestion notification upwards to any more deeply encapsulated IP headers later.

6. Feed-Backward Mode: Guidelines for Adding Congestion Notification

It can be seen from Section 3.3 that congestion notification in a subnet using feed-backward mode has generally not been designed to be directly coupled with IP layer congestion notification. The subnet attempts to minimise congestion internally, and if the incoming load at the ingress exceeds the capacity somewhere through the subnet, the layer 3 buffer into the ingress backs up. Thus, a feed-backward mode subnet is in some sense similar to a null mode subnet, in that there is no need for any direct interaction between the subnet and higher layer congestion notification. Therefore no detailed protocol design guidelines are appropriate. Nonetheless, a more general guideline is appropriate:

1. A subnetwork technology intended to eventually interface to IP SHOULD NOT be designed using only the feed-backward mode, which is certainly best for a stand-alone subnet, but would need to be modified to work efficiently as part of the wider Internet, because IP uses feed-forward-and-up mode.

The feed-backward approach at least works beneath IP, where the term 'works' is used only in a narrow functional sense because feed-backward can result in very inefficient and sluggish congestion control--except if it is confined to the subnet directly connected to the original data source, when it is faster than feed-forward. It would be valid to design a protocol that could work in feed-backward mode for paths that only cross one subnet, and in feed-forward-and-up mode for paths that cross subnets.

In the early days of TCP/IP, a similar feed-backward approach was tried for explicit congestion signalling, using source-quench (SQ) ICMP control packets. However, SQ fell out of favour and is now formally deprecated [RFC6633]. The main problem was that it is hard for a data source to tell the difference between a spoofed SQ message and a quench request from a genuine buffer on the path. It is also hard for a lower layer buffer to address an SQ message to the original source port number, which may be buried within many layers of headers, and possibly encrypted.

Quantised congestion notification (QCN--also known as backward congestion notification or BCN) [IEEE802.1Qau] uses a feed-backward mode structurally similar to ATM's relative rate mechanism. However, QCN confines its applicability to scenarios such as some data centres where all endpoints are directly attached by the same Ethernet technology. If a QCN subnet were later connected into a wider IP-based internetwork (e.g. when attempting to interconnect multiple data centres) it would suffer the inefficiency shown Figure 3.

7. IANA Considerations (to be removed by RFC Editor)

This memo includes no request to IANA.

8. Security Considerations

If a lower layer wire protocol is redesigned to include explicit congestion signalling in-band in the protocol header, care SHOULD be taken to ensure that the field used is specified as mutable during transit. Otherwise interior nodes signalling congestion would invalidate any authentication protocol applied to the lower layer header--by altering a header field that had been assumed as immutable.

The redesign of protocols that encapsulate IP in order to propagate congestion signals between layers raises potential signal integrity concerns. Experimental or proposed approaches exist for assuring the end-to-end integrity of in-band congestion signals, e.g.:

- o Congestion exposure (ConEx) for networks to audit that their congestion signals are not being suppressed by other networks or by receivers, and for networks to police that senders are responding sufficiently to the signals, irrespective of the transport protocol used [I-D.ietf-conex-abstract-mech].
- o The ECN nonce [RFC3540] for a TCP sender to detect whether a network or the receiver is suppressing congestion signals.
- o A test with the same goals as the ECN nonce, but without the need for the receiver to co-operate with the protocol [I-D.moncaster-tcpm-rcv-cheat].

Given these end-to-end approaches are already being specified, it would make little sense to attempt to design hop-by-hop congestion signal integrity into a new lower layer protocol, because end-to-end integrity inherently achieves hop-by-hop integrity.

9. Conclusions

Following the guidance in the document enables ECN support to be extended to numerous protocols that encapsulate IP (v4 & v6) in a consistent way, so that IP continues to fulfil its role as an end-to-end interoperability layer. This includes:

- o A wide range of tunnelling protocols with various forms of shim header between two IP headers;

- o A wide range of subnet technologies, particularly those that work in the same 'feed-forward-and-up' mode that is used to support ECN in IP and MPLS.

Guidelines have been defined for supporting propagation of ECN between Ethernet and IP on so-called Layer-3 Ethernet switches, using a 'feed-up-and-forward' mode. This approach could enable other subnet technologies to pass ECN signals into the IP layer, even if they do not support ECN natively.

Finally, attempting to add ECN to a subnet technology in feed-backward mode is deprecated except in special cases, due to its likely sluggish response to congestion.

10. Acknowledgements

Thanks to Gorry Fairhurst for extensive reviews. Thanks also to the following reviewers: Ingemar Johansson and Piers O'Hanlon and Michael Welzl, who pointed out that lower layer congestion notification signals may have different semantics to those in IP.

Bob Briscoe was part-funded by the European Community under its Seventh Framework Programme through the Trilogy project (ICT-216372) for initial drafts and through the Reducing Internet Transport Latency (RITE) project (ICT-317700) subsequently. The views expressed here are solely those of the authors.

11. Comments Solicited

Comments and questions are encouraged and very welcome. They can be addressed to the IETF Transport Area working group mailing list <tsvwg@ietf.org>, and/or to the authors.

12. References

12.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.
- [RFC3819] Karn, P., Bormann, C., Fairhurst, G., Grossman, D., Ludwig, R., Mahdavi, J., Montenegro, G., Touch, J., and L. Wood, "Advice for Internet Subnetwork Designers", BCP 89, RFC 3819, July 2004.

- [RFC4774] Floyd, S., "Specifying Alternate Semantics for the Explicit Congestion Notification (ECN) Field", BCP 124, RFC 4774, November 2006.
- [RFC5129] Davie, B., Briscoe, B., and J. Tay, "Explicit Congestion Marking in MPLS", RFC 5129, January 2008.
- [RFC6040] Briscoe, B., "Tunnelling of Explicit Congestion Notification", RFC 6040, November 2010.

12.2. Informative References

- [ATM-TM-ABR] Cisco, "Understanding the Available Bit Rate (ABR) Service Category for ATM VCs", Design Technote 10415, June 2005.
- [Buck00] Buckwalter, J., "Frame Relay: Technology and Practice", Pub. Addison Wesley ISBN-13: 978-0201485240, 2000.
- [DCTCP] Alizadeh, M., Greenberg, A., Maltz, D., Padhye, J., Patel, P., Prabhakar, B., Sengupta, S., and M. Sridharan, "Data Center TCP (DCTCP)", ACM SIGCOMM CCR 40(4)63--74, October 2010, <<http://portal.acm.org/citation.cfm?id=1851192>>.
- [GTPv1-U] 3GPP, "General Packet Radio System (GPRS) Tunnelling Protocol User Plane (GTPv1-U)", Technical Specification TS 29.281, .
- [GTPv1] 3GPP, "GPRS Tunnelling Protocol (GTP) across the Gn and Gp interface", Technical Specification TS 29.060, .
- [GTPv2-C] 3GPP, "Evolved General Packet Radio Service (GPRS) Tunnelling Protocol for Control plane (GTPv2-C)", Technical Specification TS 29.274, .
- [I-D.ietf-conex-abstract-mech] Mathis, M. and B. Briscoe, "Congestion Exposure (ConEx) Concepts and Abstract Mechanism", draft-ietf-conex-abstract-mech-08 (work in progress), October 2013.
- [I-D.moncaster-tcpm-rcv-cheat] Moncaster, T., "A TCP Test to Allow Senders to Identify Receiver Non-Compliance", draft-moncaster-tcpm-rcv-cheat-01 (work in progress), June 2007.

[IEEE802.1Qah]

IEEE, "IEEE Standard for Local and Metropolitan Area Networks--Virtual Bridged Local Area Networks--Amendment 6: Provider Backbone Bridges", IEEE Std 802.1Qah-2008, August 2008, <<http://www.ieee802.org/1/pages/802.1ah.html>>.

(Access Controlled link within page)

[IEEE802.1Qau]

Finn, N., Ed., "IEEE Standard for Local and Metropolitan Area Networks--Virtual Bridged Local Area Networks - Amendment 13: Congestion Notification", IEEE Std 802.1Qau-2010, March 2010, <<http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5454061>>.

(Access Controlled link within page)

[ITU-T.I.371]

ITU-T, "Traffic Control and Congestion Control in B-ISDN", ITU-T Rec. I.371 (03/04), March 2004, <<http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=5454061>>.

[LTE-RA] 3GPP, "Evolved Universal Terrestrial Radio Access (E-UTRA) and Evolved Universal Terrestrial Radio Access Network (E-UTRAN); Overall description; Stage 2", Technical Specification TS 36.300, .

[RFC1323] Jacobson, V., Braden, B., and D. Borman, "TCP Extensions for High Performance", RFC 1323, May 1992.

[RFC1701] Hanks, S., Li, T., Farinacci, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 1701, October 1994.

[RFC2003] Perkins, C., "IP Encapsulation within IP", RFC 2003, October 1996.

[RFC2637] Hamzeh, K., Pall, G., Verthein, W., Taarud, J., Little, W., and G. Zorn, "Point-to-Point Tunneling Protocol", RFC 2637, July 1999.

[RFC2661] Townsley, W., Valencia, A., Rubens, A., Pall, G., Zorn, G., and B. Palter, "Layer Two Tunneling Protocol "L2TP"", RFC 2661, August 1999.

- [RFC2784] Farinacci, D., Li, T., Hanks, S., Meyer, D., and P. Traina, "Generic Routing Encapsulation (GRE)", RFC 2784, March 2000.
- [RFC2884] Hadi Salim, J. and U. Ahmed, "Performance Evaluation of Explicit Congestion Notification (ECN) in IP Networks", RFC 2884, July 2000.
- [RFC2983] Black, D., "Differentiated Services and Tunnels", RFC 2983, October 2000.
- [RFC3540] Spring, N., Wetherall, D., and D. Ely, "Robust Explicit Congestion Notification (ECN) Signaling with Nonces", RFC 3540, June 2003.
- [RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.
- [RFC6633] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, May 2012.
- [RFC6660] Briscoe, B., Moncaster, T., and M. Menth, "Encoding Three Pre-Congestion Notification (PCN) States in the IP Header Using a Single Diffserv Codepoint (DSCP)", RFC 6660, July 2012.
- [trill-rbridge-options] Eastlake, D., Ghanwani, A., Manral, V., and C. Bestler, "RBridges: Further TRILL Header Extensions", draft-ietf-trill-rbridge-options-07 (work in progress), June 2012.
- [vxlan] Mahalingam, M., Dutt, D., Duda, K., Agarwal, P., Kreeger, L., Sridhar, T., Bursell, M., and C. Wright, "VXLAN: A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks", draft-mahalingam-dutt-dcops-vxlan-08 (work in progress), February 2014.

Appendix A. Outstanding Document Issues

1. [GF] Concern that certain guidelines warrant a MUST (NOT) rather than a SHOULD (NOT). Given the guidelines say that if any SHOULD (NOT)s are not followed, a strong justification will be needed, they have been left as SHOULD (NOT) pending further list discussion. In particular:
 - * If inner is a Not-ECN-PDU and Outer is CE (or highest severity congestion level), MUST (not SHOULD) drop?
2. Consider whether an IETF Standard Track doc will be needed to Update the IP-in-IP protocols listed in Section 4.1--at least those that the IET

Appendix B. Changes in This Version (to be removed by RFC Editor)

From briscoe-03 to 04:

- * Re-arranged the introduction to describe the purpose of the document first before introducing ECN in more depth. And clarified the introduction throughout.
- * Added applicability to 3GPP TS 36.300.

From briscoe-02 to 03:

- * Scope section:
 - + Added dependence on correct propagation of traffic class information
 - + For the feed-backward mode, deemed multicast and anycast out of scope
- * Ensured all guidelines referring to subnet technologies also refer to tunnels and vice versa by adding applicability sentences at the start of sections 4.1, 4.2, 4.3, 4.4, 4.6 and 5.
- * Added Security Considerations on ensuring congestion signal fields are classed as immutable and on using end-to-end congestion signal integrity technologies rather than hop-by-hop.

From briscoe-01 to 02:

- * Added authors: JK & PT

- * Added
 - + Section 4.1 "IP-in-IP Tunnels with Tightly Coupled Shim Headers"
 - + Section 4.5 "Sequences of Similar Tunnels or Subnets"
 - + roadmap at the start of Section 4, given the subsections have become quite fragmented.
 - + Section 9 "Conclusions"
- * Clarified why transports are starting to be able to saturate interior links
- * Under Section 1.1, addressed the question of alternative signal semantics and included multicast & anycast.
- * Under Section 3.1, included a 3GPP example.
- * Section 4.2. "Wire Protocol Design":
 - + Altered guideline 2. to make it clear that it only applies to the immediate subnet egress, not later ones
 - + Added a reminder that it is only necessary to check that ECN propagates at the egress, not whether interior nodes mark ECN
 - + Added example of how QCN uses 802.1p to indicate support for QCN.
- * Added references to Appendix C of RFC6040, about monitoring the amount of congestion signals introduced within a tunnel
- * Appendix A: Added more issues to be addressed, including plan to produce a standards track update to IP-in-IP tunnel protocols.
- * Updated acks and references

From briscoe-00 to 01:

- * Intended status: BCP (was Informational) & updates 3819 added.
- * Briefer Introduction: Introductory para justifying benefits of ECN. Moved all but a brief enumeration of modes of operation

to their own new section (from both Intro & Scope). Introduced incr. deployment as most tricky part.

- * Tightened & added to terminology section
- * Structured with Modes of Operation, then Guidelines section for each mode.
- * Tightened up guideline text to remove vagueness / passive voice / ambiguity and highlight main guidelines as numbered items.
- * Added Outstanding Document Issues Appendix
- * Updated references

Authors' Addresses

Bob Briscoe
BT
B54/77, Adastral Park
Martlesham Heath
Ipswich IP5 3RE
UK

Phone: +44 1473 645196
EMail: bob.briscoe@bt.com
URI: <http://bobbbriscoe.net/>

John Kaippallimalil
Huawei
5340 Legacy Drive, Suite 175
Plano, Texas 75024
USA

EMail: john.kaippallimalil@huawei.com

Pat Thaler
Broadcom Corporation
5025 Keane Drive
Carmichael, CA 95608
USA

EMail: pthaler@broadcom.com

Internet Engineering Task Force
Internet-Draft
Intended status: Standards Track
Expires: August 29, 2013

G. Fairhurst
University of Aberdeen
M. Westerlund
Ericsson
February 25, 2013

Applicability Statement for the use of IPv6 UDP Datagrams with Zero
Checksums
draft-ietf-6man-udpzero-12

Abstract

This document provides an applicability statement for the use of UDP transport checksums with IPv6. It defines recommendations and requirements for the use of IPv6 UDP datagrams with a zero UDP checksum. It describes the issues and design principles that need to be considered when UDP is used with IPv6 to support tunnel encapsulations and examines the role of the IPv6 UDP transport checksum. The document also identifies issues and constraints for deployment on network paths that include middleboxes. An appendix presents a summary of the trade-offs that were considered in evaluating the safety of the update to RFC 2460 that updates use of the UDP checksum with IPv6.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on August 29, 2013.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal

Provisions Relating to IETF Documents
<http://trustee.ietf.org/license-info>) in effect on the date of
publication of this document. Please review these documents
carefully, as they describe your rights and restrictions with respect
to this document. Code Components extracted from this document must
include Simplified BSD License text as described in Section 4.e of
the Trust Legal Provisions and are provided without warranty as
described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Document Structure	5
1.2. Terminology	5
1.3. Use of UDP Tunnels	5
1.3.1. Motivation for new approaches	6
1.3.2. Reducing forwarding cost	6
1.3.3. Need to inspect the entire packet	7
1.3.4. Interactions with middleboxes	7
1.3.5. Support for load balancing	8
2. Standards-Track Transports	9
2.1. UDP with Standard Checksum	9
2.2. UDP-Lite	9
2.2.1. Using UDP-Lite as a Tunnel Encapsulation	10
2.3. General Tunnel Encapsulations	10
2.4. Relation to UDP-Lite and UDP with checksum	10
3. Issues Requiring Consideration	12
3.1. Effect of packet modification in the network	13
3.1.1. Corruption of the destination IP address	14
3.1.2. Corruption of the source IP address	15
3.1.3. Corruption of Port Information	16
3.1.4. Delivery to an unexpected port	16
3.1.5. Corruption of Fragmentation Information	17
3.2. Where Packet Corruption Occurs	19
3.3. Validating the network path	20
3.4. Applicability of method	21
3.5. Impact on non-supporting devices or applications	21
4. Constraints on implementation of IPv6 nodes supporting zero checksum	22
5. Requirements on usage of the zero UDP checksum	24
6. Summary	26
7. Acknowledgements	28
8. IANA Considerations	28
9. Security Considerations	28
10. References	29
10.1. Normative References	29
10.2. Informative References	29

Appendix A. Evaluation of proposal to update RFC 2460 to support zero checksum	31
A.1. Alternatives to the Standard Checksum	31
A.2. Comparison	33
A.2.1. Middlebox Traversal	33
A.2.2. Load Balancing	34
A.2.3. Ingress and Egress Performance Implications	34
A.2.4. Deployability	34
A.2.5. Corruption Detection Strength	35
A.2.6. Comparison Summary	35
Appendix B. Document Change History	38
Authors' Addresses	41

1. Introduction

The User Datagram Protocol (UDP) [RFC0768] transport is defined for the Internet Protocol (IPv4) [RFC0791] and is defined in "Internet Protocol, Version 6 (IPv6) [RFC2460] for IPv6 hosts and routers. The UDP transport protocol has a minimal set of features. This limited set has enabled a wide range of applications to use UDP, but these application do need to provide many important transport functions on top of UDP. The UDP Usage Guidelines [RFC5405] provides overall guidance for application designers, including the use of UDP to support tunneling. The key difference between UDP usage with IPv4 and IPv6 is that RFC 2460 mandates use of a calculated UDP checksum, i.e. a non-zero value, due to the lack of an IPv6 header checksum. The inclusion of the pseudo header in the checksum computation provides a statistical check that datagrams have been delivered to the intended IPv6 destination node. Algorithms for checksum computation are described in [RFC1071].

The lack of a possibility to use an IPv6 datagram with a zero UDP checksum has been observed as a real problem for certain classes of application, primarily tunnel applications. This class of application has been deployed with a zero UDP checksum using IPv4. The design of IPv6 raises different issues when considering the safety of using a UDP checksum with IPv6. These issues can significantly affect applications, both when an endpoint is the intended user and when an innocent bystander (when a packet is received by a different endpoint to that intended).

This document examines the issues and an appendix compares the strengths and weaknesses of a number of proposed solutions. This identifies a set of issues that must be considered and mitigated to be able to safely deploy IPv6 applications that use a zero UDP checksum. The provided comparison of methods is expected to also be useful when considering applications that have different goals from the ones that initiated the writing of this document, especially the use of already standardized methods. The analysis concludes that using a zero UDP checksum is the best method of the proposed alternatives to meet the goals for certain tunnel applications.

This document defines recommendations and requirements for use of IPv6 datagrams with a zero UDP checksum. This usage is expected to have initial deployment issues related to middleboxes, limiting the usability more than desired in the currently deployed Internet. However, this limitation will be largest initially and will reduce as updates are provided in middleboxes that support the zero UDP checksum for IPv6. The document therefore derives a set of constraints required to ensure safe deployment of a zero UDP checksum.

Finally, the document also identifies some issues that require future consideration and possibly additional research.

1.1. Document Structure

Section 1 provides a background to key issues, and introduces the use of UDP as a tunnel transport protocol.

Section 2 describes a set of standards-track datagram transport protocols that may be used to support tunnels.

Section 3 discusses issues with a zero UDP checksum for IPv6. It considers the impact of corruption, the need for validation of the path and when it is suitable to use a zero UDP checksum.

Section 4 is an applicability statement that defines requirements and recommendations on the implementation of IPv6 nodes that support the use of a zero UDP checksum.

Section 5 provides an applicability statement that defines requirements and recommendations for protocols and tunnel encapsulations that are transported over an IPv6 transport that does not perform a UDP checksum calculation to verify the integrity at the transport endpoints.

Section 6 provides the recommendations for standardization of zero UDP checksum with a summary of the findings and notes remaining issues needing future work.

Appendix A evaluates the set of proposals to update the UDP transport behaviour and other alternatives intended to improve support for tunnel protocols. It concludes by assessing the trade-offs of the various methods, identifying advantages and disadvantages for each method.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

1.3. Use of UDP Tunnels

One increasingly popular use of UDP is as a tunneling protocol, where a tunnel endpoint encapsulates the packets of another protocol inside UDP datagrams and transmits them to another tunnel endpoint. Using UDP as a tunneling protocol is attractive when the payload protocol is not supported by the middleboxes that may exist along the path,

because many middleboxes support transmission using UDP. In this use, the receiving endpoint decapsulates the UDP datagrams and forwards the original packets contained in the payload [RFC5405]. Tunnels establish virtual links that appear to directly connect locations that are distant in the physical Internet topology and can be used to create virtual (private) networks.

1.3.1. Motivation for new approaches

A number of tunnel encapsulations deployed over IPv4 have used the UDP transport with a zero checksum. Users of these protocols expect a similar solution for IPv6.

A number of tunnel protocols are also currently being defined (e.g. Automated Multicast Tunnels, AMT [I-D.ietf-mboned-auto-multicast], and the Locator/Identifier Separation Protocol, LISP [LISP]). These protocols motivated an update to IPv6 UDP checksum processing to benefit from simpler checksum processing for various reasons:

- o Reducing forwarding costs, motivated by redundancy present in the encapsulated packet header, since in tunnel encapsulations, payload integrity and length verification may be provided by higher layer encapsulations (often using the IPv4, UDP, UDP-Lite, or TCP checksums).
- o Eliminating a need to access the entire packet when forwarding the packet by a tunnel endpoint.
- o Enhancing ability to traverse and function with middleboxes.
- o A desire to use the port number space to enable load-sharing.

1.3.2. Reducing forwarding cost

It is a common requirement to terminate a large number of tunnels on a single router/host. The processing cost per tunnel includes both state (memory requirements) and per-packet processing at the tunnel ingress and egress.

Automatic IP Multicast Tunneling, known as AMT [I-D.ietf-mboned-auto-multicast] currently specifies UDP as the transport protocol for packets carrying tunneled IP multicast packets. The current specification for AMT states that the UDP checksum in the outer packet header should be zero (see Section 6.6 of [I-D.ietf-mboned-auto-multicast]). This argues that the computation of an additional checksum is an unwarranted burden on nodes implementing lightweight tunneling protocols when an inner packet is already adequately protected, . The AMT protocol needs to

replicate a multicast packet to each gateway tunnel. In this case, the outer IP addresses are different for each tunnel and therefore require a different pseudo header to be built for each UDP replicated encapsulation.

The argument concerning redundant processing costs is valid regarding the integrity of a tunneled packet. In some architectures (e.g. PC-based routers), other mechanisms may also significantly reduce checksum processing costs: There are implementations that have optimised checksum processing algorithms, including the use of checksum-offloading. This processing is readily available for IPv4 packets at high line rates. Such processing may be anticipated for IPv6 endpoints, allowing receivers to reject corrupted packets without further processing. However, there are certain classes of tunnel end-points where this off-loading is not available and unlikely to become available in the near future.

1.3.3. Need to inspect the entire packet

The currently-deployed hardware in many routers uses a fast-path processing that only provides the first n bytes of a packet to the forwarding engine, where typically $n \leq 128$.

When this design is used to support a tunnel ingress and egress, it prevents fast processing of a transport checksum over an entire (large) packet. Hence the currently defined IPv6 UDP checksum is poorly suited to use within a router that is unable to access the entire packet and does not provide checksum-offloading. Thus enabling checksum calculation over the complete packet can impact router design, performance improvement, energy consumption and/or cost.

1.3.4. Interactions with middleboxes

Many paths in the Internet include one or more middleboxes of various types. There exist large classes of middleboxes that will handle zero UDP checksum packets, which would not support UDP-Lite or the other investigated proposals. These middleboxes includes load balancers (see Section 1.3.5) including Equal Cost Multipath Routing, traffic classifiers and other functions that reads some fields in the UDP headers but does not validate the UDP checksum.

There are also middleboxes that either validates or modify the UDP checksum. The two most common classes are Firewalls and NATs. In IPv4, UDP-encapsulation may be desirable for NAT traversal, since UDP support is commonly provided. It is also necessary due to the almost ubiquitous deployment of IPv4 NATs. There has also been discussion of NAT for IPv6, although not for the same reason as in IPv4. If

IPv6 NAT becomes a reality they hopefully do not present the same protocol issues as for IPv4. If NAT is defined for IPv6, it should take into consideration the use of a zero UDP checksum.

The requirements for IPv6 firewall traversal are likely to be similar to those for IPv4. In addition, it can be reasonably expected that a firewall conforming to RFC 2460 will not regard datagrams with a zero UDP checksum as valid. Use of a zero UDP checksum with IPv6 requires firewalls to be updated before the full utility of the change is available.

It can be expected that datagrams with zero UDP checksum will initially not have the same middlebox traversal characteristics as regular UDP (RFC 2460). However when implementations follow the requirements specified in this document, we expect the traversal capabilities to improve over time. We also note that deployment of IPv6-capable middleboxes is still in its initial phases. Thus, it might be that the number of non-updated boxes quickly become a very small percentage of the deployed middleboxes.

1.3.5. Support for load balancing

The UDP port number fields have been used as a basis to design load-balancing solutions for IPv4. This approach has also been leveraged for IPv6. An alternate method would be to utilise the IPv6 Flow Label [RFC6437] as a basis for entropy for load balancing. This would have the desirable effect of releasing IPv6 load-balancing devices from the need to assume semantics for the use of the transport port field and also works for all type of transport protocols.

This use of the flow-label for load balancing is consistent with the intended use, although further clarity was needed to ensure the field can be consistently used for this purpose, therefore an updated IPv6 Flow Label [RFC6437] and Equal-Cost Multi-Path routing usage, (ECMP) [RFC6438] was produced. Router vendors could be encouraged to start using the IPv6 Flow Label as a part of the flow hash, providing support for ECMP without requiring use of UDP.

However, the method for populating the outer IPv6 header with a value for the flow label is not trivial: If the inner packet uses IPv6, then the flow label value could be copied to the outer packet header. However, many current end-points set the flow label to a zero value (thus no entropy). The ingress of a tunnel seeking to provide good entropy in the flow label field would therefore need to create a random flow label value and keep corresponding state, so that all packets that were associated with a flow would be consistently given the same flow label. Although possible, this complexity may not be

desirable in a tunnel ingress.

The end-to-end use of flow labels for load balancing is a long-term solution. Even if the usage of the flow label is clarified, there would be a transition time before a significant proportion of end-points start to assign a good quality flow label to the flows that they originate, with continued use of load balancing using the transport header fields until any widespread deployment is finally achieved.

2. Standards-Track Transports

The IETF has defined a set of transport protocols that may be applicable for tunnels with IPv6. There are also a set of network layer encapsulation tunnels such as IP-in-IP and GRE. These already standardized solutions are discussed here prior to the issues, as background for the issue description and some comparison of where the issue may already occur.

2.1. UDP with Standard Checksum

UDP [RFC0768] with standard checksum behaviour, as defined in RFC 2460, has already been discussed. UDP usage guidelines are provided in [RFC5405].

2.2. UDP-Lite

UDP-Lite [RFC3828] offers an alternate transport to UDP, specified as a proposed standard, RFC 3828. A MIB is defined in [RFC5097] and unicast usage guidelines in [RFC5405]. There is at least one open source implementation as a part of the Linux kernel since version 2.6.20.

UDP-Lite provides a checksum with optional partial coverage. When using this option, a datagram is divided into a sensitive part (covered by the checksum) and an insensitive part (not covered by the checksum). When the checksum covers the entire packet, UDP-Lite is fully equivalent with UDP, with the exception that it uses a different value in the Next Header field in the IPv6 header. Errors/corruption in the insensitive part will not cause the datagram to be discarded by the transport layer at the receiving endpoint. A minor side-effect of using UDP-Lite is that this was specified for damage-tolerant payloads and some link-layers may employ different link encapsulations when forwarding UDP-Lite segments (e.g. radio access bearers). Most link-layers will cover the insensitive part with the same strong layer 2 frame CRC that covers the sensitive part.

2.2.1. Using UDP-Lite as a Tunnel Encapsulation

Tunnel encapsulations can use UDP-Lite (e.g. Control And Provisioning of Wireless Access Points, CAPWAP [RFC5415]), since UDP-Lite provides a transport-layer checksum, including an IP pseudo header checksum, in IPv6, without the need for a router/middlebox to traverse the entire packet payload. This provides most of the verification required for delivery and still keeps a low complexity for the checksumming operation. UDP-Lite may set the length of checksum coverage on a per packet basis. This feature could be used if a tunnel protocol is designed to only verify delivery of the tunneled payload and uses a calculated checksum for control information.

There is currently poor support for middlebox traversal using UDP-Lite, because UDP-Lite uses a different IPv6 network-layer Next Header value to that of UDP, and few middleboxes are able to interpret UDP-Lite and take appropriate actions when forwarding the packet. This makes UDP-Lite less suited to protocols needing general Internet support, until such time that UDP-Lite has achieved better support in middleboxes and end-points.

2.3. General Tunnel Encapsulations

The IETF has defined a set of tunneling protocols or network layer encapsulations, e.g., IP-in-IP and GRE. These either do not include a checksum or use a checksum that is optional, since tunnel encapsulations are typically layered directly over the Internet layer (identified by the upper layer type in the IPv6 Next Header field) and are also not used as endpoint transport protocols. There is little chance of confusing a tunnel-encapsulated packet with other application data that could result in corruption of application state or data.

From the end-to-end perspective, the principal difference is that the network-layer Next Header field identifies a separate transport, which reduces the probability that corruption could result in the packet being delivered to the wrong endpoint or application. Specifically, packets are only delivered to protocol modules that process a specific Next Header value. The Next Header field therefore provides a first-level check of correct demultiplexing. In contrast, the UDP port space is shared by many diverse applications and therefore UDP demultiplexing relies solely on the port numbers.

2.4. Relation to UDP-Lite and UDP with checksum

The operation of IPv6 with UDP with a zero-checksum is not the same as IPv4 with UDP with a zero-checksum. Protocol designers should not

be fooled into thinking the two are the same. The requirements below list a set of additional considerations.

Where possible, existing general tunnel encapsulations, such as GRE, IP-in-IP, should be used. This section assumes that such existing tunnel encapsulations do not offer the functionality required to satisfy the protocol designer's goals. The section considers the standardized alternative solutions, rather than the full set of ideas evaluated in Appendix A. The alternatives to UDP with a zero checksum are UDP with a (calculated) checksum, and UDP-Lite.

UDP with a checksum has the advantage of close to universal support in both endpoints and middleboxes. It also provides statistical verification of delivery to the intended destination (address and port). However, some classes of device have limited support for calculation of a checksum that covers a full datagram. For these devices, this can incur significant processing cost (e.g. requiring processing in the router slow-path) and can hence reduce capacity or fail to function.

UDP-Lite has the advantage of using a checksum that is calculated only over the pseudo header and the UDP header. This provides a statistical verification of delivery to the intended destination (address and port). The checksum can be calculated without access to the datagram payload, only requiring access to the part to be protected. A drawback is that UDP-Lite has currently limited support in both end-points (i.e. is not supported on all operating system platforms) and middleboxes (that require support for the UDP-Lite header type). A path verification method is therefore recommended.

IPv6 and UDP with a zero-checksum can also be used by nodes that do not permit calculation of a payload checksum. Many existing classes of middleboxes do not verify or change the transport checksum. For these middleboxes, IPv6 with a zero UDP checksum is expected to function where UDP-Lite would not. However, support for the zero UDP checksum in middleboxes that do change or verify the checksum is currently limited, and this may result in datagrams with a zero UDP checksum being discarded, therefore a path verification method is recommended.

There are sets of constraints for which no solution exist: A protocol designer that needs to originate or receive datagrams on a device that can not efficiently calculate a checksum over a full datagram and also needs these packets to pass through a middlebox that verifies or changes a UDP checksum, but does not support a zero UDP checksum, can not use the zero UDP checksum method. Similarly, one that originates datagrams on a device with UDP-Lite support, but needs the packets to pass through a middlebox that does not support

UDP-Lite, can not use UDP-Lite. For such cases, there is no optimal solution and the current recommendation is to use or fall-back to using UDP with full checksum coverage.

3. Issues Requiring Consideration

This informative section evaluates issues around the proposal to update IPv6 [RFC2460], to enable the UDP transport checksum to be set to zero. Some of the identified issues are shared with other protocols already in use. The section also provides background to the requirements and recommendations that follow.

The decision in RFC 2460 to omit an integrity check at the network level meant that the IPv6 transport checksum was overloaded with many functions, including validating:

- o the endpoint address was not corrupted within a router, i.e., a packet was intended to be received by this destination and validate that the packet does not consist of a wrong header spliced to a different payload;
- o that extension header processing is correctly delimited - i.e., the start of data has not been corrupted. In this case, reception of a valid Next Header value provides some protection;
- o reassembly processing, when used;
- o the length of the payload;
- o the port values - i.e., the correct application receives the payload (applications should also check the expected use of source ports/addresses);
- o the payload integrity.

In IPv4, the first four checks are performed using the IPv4 header checksum.

In IPv6, these checks occur within the endpoint stack using the UDP checksum information. An IPv6 node also relies on the header information to determine whether to send an ICMPv6 error message [RFC4443] and to determine the node to which this is sent. Corrupted information may lead to misdelivery to an unintended application socket on an unexpected host.

3.1. Effect of packet modification in the network

IP packets may be corrupted as they traverse an Internet path. Older evidence in "When the CRC and TCP Checksum Disagree" [Sigcomm2000] show that this was once an issue in year 2000 with IPv4 routers, and occasional corruption could result from bad internal router processing in routers or hosts. These errors are not detected by the strong frame checksums employed at the link-layer [RFC3819]. During the development of this document in 2009, individuals provided reports of observed rates for received UDP datagrams using IPv4 where the UDP checksum had been detected as corrupt. These rates were as high as 1.39E-4 for some paths, but also close to zero for some other paths.

There is extensive experience of deployment using tunnel protocols in well-managed networks (e.g. corporate networks or service provider core networks). This has shown the robustness of methods such as PWE and MPLS that do not employ a transport protocol checksum and have not specified mechanisms to protect from corruption of the unprotected headers (such as the VPN Identifier in MPLS). Reasons for the robustness may include:

- o A reduced probability of corruption on paths through well-managed networks.
- o IP forms the majority of the inner traffic carried by these tunnels. Hence from a transport perspective, endpoint verification is already being performed when processing a received IPv4 packet or by the transport pseudo-header for an IPv6 packet. This update to UDP does not change this behaviour.
- o In certain cases, a combination of additional filtering (e.g. filter of a MAC destination address in a L2 tunnel) significantly reduces the probability of final mis-delivery to the IP stack.
- o The tunnel protocols did not use a UDP transport header, any corruption is therefore unlikely to result in misdelivery to another UDP-based application. This concern is specific to the use of UDP with IPv6.

While this experience can guide the present recommendations, any update to UDP must preserve operation in the general Internet. This is heterogeneous and can include links and systems of very varying characteristics. Transport protocols used by hosts need to be designed with this in mind, especially when there is need to traverse edge networks, where middlebox deployments are common.

For the general Internet, there is no current evidence that

corruption is rare, nor that this may not be applicable to IPv6. It therefore seems prudent not to relax checks on misdelivery . The emergence of low-end IPv6 routers and the proposed use of NAT with IPv6 further motivate the need to protect from misdelivery.

Corruption in the network may result in:

- o A datagram being misdelivered to the wrong host/router or the wrong transport entity within an endpoint. Such a datagram needs to be discarded;
- o A datagram payload being corrupted, but still delivered to the intended host/router transport entity. Such a datagram needs to be either discarded or correctly processed by an application that provides its own integrity checks;
- o A datagram payload being truncated by corruption of the length field. Such a datagram needs to be discarded.

When a checksum is used, this significantly reduces the impact of errors, reducing the probability of undetected corruption of state (and data) on both the host stack and the applications using the transport service.

The following sections examine the impact of modifying each of these header fields.

3.1.1. Corruption of the destination IP address

An IPv6 endpoint destination address could be modified in the network (e.g. corrupted by an error). This is not a concern for IPv4, because the IP header checksum will result in this packet being discarded by the receiving IP stack. Such modification in the network can not be detected at the network layer when using IPv6. Detection of this corruption by a UDP receiver relies on the IPv6 pseudo header incorporated in the transport checksum.

There are two possible outcomes:

- o Delivery to a destination address that is not in use (the packet will not be delivered, but could result in an error report);
- o Delivery to a different destination address. This modification will normally be detected by the transport checksum, resulting in silent discard. Without a computed checksum, the packet would be passed to the endpoint port demultiplexing function. If an application is bound to the associated ports, the packet payload will be passed to the application (see the subsequent section on

port processing).

3.1.2. Corruption of the source IP address

This section examines what happens when the source address is corrupted in transit. This is not a concern in IPv4, because the IP header checksum will normally result in this packet being discarded by the receiving IP stack. Detection of this corruption by a UDP receiver relies on the IPv6 pseudo header incorporated in the transport checksum.

Corruption of an IPv6 source address does not result in the IP packet being delivered to a different endpoint protocol or destination address. If only the source address is corrupted, the datagram will likely be processed in the intended context, although with erroneous origin information. When using Unicast Reverse Path Forwarding [RFC2827], a change in address may result in the router discarding the packet when the route to the modified source address is different to that of the source address of the original packet.

The result will depend on the application or protocol that processes the packet. Some examples are:

- o An application that requires a per-established context may disregard the datagram as invalid, or could map this to another context (if a context for the modified source address was already activated).
- o A stateless application will process the datagram outside of any context, a simple example is the ECHO server, which will respond with a datagram directed to the modified source address. This would create unwanted additional processing load, and generate traffic to the modified endpoint address.
- o Some datagram applications build state using the information from packet headers. A previously unused source address would result in receiver processing and the creation of unnecessary transport-layer state at the receiver. For example, Real Time Protocol (RTP) [RFC3550] sessions commonly employ a source independent receiver port. State is created for each received flow. Reception of a datagram with a corrupted source address will therefore result in accumulation of unnecessary state in the RTP state machine, including collision detection and response (since the same synchronization source, SSRC, value will appear to arrive from multiple source IP addresses).
- o ICMP messages relating to a corrupted packet can be misdirected to the wrong source node.

In general, the effect of corrupting the source address will depend upon the protocol that processes the packet and its robustness to this error. For the case where the packet is received by a tunnel endpoint, the tunnel application is expected to correctly handle a corrupted source address.

The impact of source address modification is more difficult to quantify when the receiving application is not that originally intended and several fields have been modified in transit.

3.1.3. Corruption of Port Information

This section describes what happens if one or both of the UDP port values are corrupted in transit. This can also happen with IPv4 is used with a zero UDP checksum, but not when UDP checksums are calculated or when UDP-Lite is used. If the ports carried in the transport header of an IPv6 packet were corrupted in transit, packets may be delivered to the wrong application process (on the intended machine) and/or responses or errors sent to the wrong application process (on the intended machine).

3.1.4. Delivery to an unexpected port

If one combines the corruption effects, such as destination address and ports, there is a number of potential outcomes when traffic arrives at an unexpected port. This section discusses these possibilities and their outcomes for a packet that does not use the UDP checksum validation:

- o Delivery to a port that is not in use. The packet is discarded, but could generate an ICMPv6 message (e.g. port unreachable).
- o It could be delivered to a different node that implements the same application, where the packet may be accepted, generating side-effects or accumulated state.
- o It could be delivered to an application that does not implement the tunnel protocol, where the packet may be incorrectly parsed, and may be misinterpreted, generating side-effects or accumulated state.

The probability of each outcome depends on the statistical probability that the address or the port information for the source or destination becomes corrupt in the datagram such that they match those of an existing flow or server port. Unfortunately, such a match may be more likely for UDP than for connection-oriented transports, because:

1. There is no handshake prior to communication and no sequence numbers (as in TCP, DCCP, or SCTP). Together, this makes it hard to verify that an application process is given only the application data associated with a specific transport session.
2. Applications writers often bind to wild-card values in endpoint identifiers and do not always validate correctness of datagrams they receive (guidance on this topic is provided in [RFC5405]).

While these rules could, in principle, be revised to declare naive applications as "Historic". This remedy is not realistic: the transport owes it to the stack to do its best to reject bogus datagrams.

If checksum coverage is suppressed, the application therefore needs to provide a method to detect and discard the unwanted data. A tunnel protocol would need to perform its own integrity checks on any control information if transported in datagrams with a zero UDP checksum. If the tunnel payload is another IP packet, the packets requiring checksums can be assumed to have their own checksums provided that the rate of corrupted packets is not significantly larger due to the tunnel encapsulation. If a tunnel transports other inner payloads that do not use IP, the assumptions of corruption detection for that particular protocol must be fulfilled, this may require an additional checksum/CRC and/or integrity protection of the payload and tunnel headers.

A protocol that uses a zero UDP checksum can not assume that it is the only protocol using a zero UDP checksum. Therefore, it needs to gracefully handle misdelivery. It must be robust to reception of malformed packets received on a listening port and expect that these packets may contain corrupted data or data associated with a completely different protocol.

3.1.5. Corruption of Fragmentation Information

The fragmentation information in IPv6 employs a 32-bit identity field, compared to only a 16-bit field in IPv4, a 13-bit fragment offset and a 1-bit flag, indicating if there are more fragments. Corruption of any of these field may result in one of two outcomes:

Reassembly failure: An error in the "More Fragments" field for the last fragment will for example result in the packet never being considered complete and will eventually be timed out and discarded. A corruption in the ID field will result in the fragment not being delivered to the intended context thus leaving the rest incomplete, unless that packet has been duplicated prior to corruption. The incomplete packet will eventually be timed out

and discarded.

Erroneous reassembly: The re-assembled packet did not match the original packet. This can occur when the ID field of a fragment is corrupted, resulting in a fragment becoming associated with another packet and taking the place of another fragment. Corruption in the offset information can cause the fragment to be misaligned in the reassembly buffer, resulting in incorrect reassembly. Corruption can cause the packet to become shorter or longer, however completion of reassembly is much less probable, since this would require consistent corruption of the IPv6 headers payload length field and the offset field. The possibility of mis-assembly requires the reassembling stack to provide strong checks that detect overlap or missing data, note however that this is not guaranteed and has been clarified in "Handling of Overlapping IPv6 Fragments" [RFC5722].

The erroneous reassembly of packets is a general concern and such packets should be discarded instead of being passed to higher layer processes. The primary detector of packet length changes is the IP payload length field, with a secondary check by the transport checksum. The Upper-Layer Packet length field included in the pseudo header assists in verifying correct reassembly, since the Internet checksum has a low probability of detecting insertion of data or overlap errors (due to misplacement of data). The checksum is also incapable of detecting insertion or removal of all zero-data that occurs in a multiple of a 16-bit chunk.

The most significant risk of corruption results following mis-association of a fragment with a different packet. This risk can be significant, since the size of fragments is often the same (e.g. fragments resulting when the path MTU results in fragmentation of a larger packet, common when addition of a tunnel encapsulation header expands the size of a packet). Detection of this type of error requires a checksum or other integrity check of the headers and the payload. Such protection is anyway desirable for tunnel encapsulations using IPv4, since the small fragmentation ID can easily result in wrap-around [RFC4963], this is especially the case for tunnels that perform flow aggregation [I-D.ietf-intarea-tunnels].

Tunnel fragmentation behavior matters. There can be outer or inner fragmentation "Tunnels in the Internet Architecture" [I-D.ietf-intarea-tunnels]. If there is inner fragmentation by the tunnel, the outer headers will never be fragmented and thus a zero UDP checksum in the outer header will not affect the reassembly process. When a tunnel performs outer header fragmentation, the tunnel egress needs to perform reassembly of the outer fragments into an inner packet. The inner packet is either a complete packet or a

fragment. If it is a fragment, the destination endpoint of the fragment will perform reassembly of the received fragments. The complete packet or the reassembled fragments will then be processed according to the packet Next Header field. The receiver may only detect reassembly anomalies when it uses a protocol with a checksum. The larger the number of reassembly processes to which a packet has been subjected, the greater the probability of an error.

- o An IP-in-IP tunnel that performs inner fragmentation has similar properties to a UDP tunnel with a zero UDP checksum that also performs inner fragmentation.
- o An IP-in-IP tunnel that performs outer fragmentation has similar properties to a UDP tunnel with a zero UDP checksum that performs outer fragmentation.
- o A tunnel that performs outer fragmentation can result in a higher level of corruption due to both inner and outer fragmentation, enabling more chances for reassembly errors to occur.
- o Recursive tunneling can result in fragmentation at more than one header level, even for inner fragmentation unless it goes to the inner-most IP header.
- o Unless there is verification at each reassembly, the probability for undetected error will increase with the number of times fragmentation is recursively applied, making IP-in-IP and UDP with zero UDP checksum both vulnerable to undetected errors.

In conclusion, fragmentation of datagrams with a zero UDP checksum does not worsen the performance compared to some other commonly used tunnel encapsulations. However, caution is needed for recursive tunneling without any additional verification at the different tunnel layers.

3.2. Where Packet Corruption Occurs

Corruption of IP packets can occur at any point along a network path, during packet generation, during transmission over the link, in the process of routing and switching, etc. Some transmission steps include a checksum or Cyclic Redundancy Check (CRC) that reduces the probability for corrupted packets being forwarded, but there still exists a probability that errors may propagate undetected.

Unfortunately the community lacks reliable information to identify the most common functions or equipment that result in packet corruption. However, there are indications that the place where corruption occurs can vary significantly from one path to another.

There is therefore a risk in applying evidence from one domain of usage to infer characteristics for another. Methods intended for general Internet usage must therefore assume that corruption can occur and deploy mechanisms to mitigate the effect of corruption and/or resulting misdelivery.

3.3. Validating the network path

IP transports designed for use in the general Internet should not assume specific path characteristics. Network protocols may reroute packets that change the set of routers and middleboxes along a path. Therefore transports such as TCP, SCTP and DCCP have been designed to negotiate protocol parameters, adapt to different network path characteristics, and receive feedback to verify that the current path is suited to the intended application. Applications using UDP and UDP-Lite need to provide their own mechanisms to confirm the validity of the current network path.

A zero value in the UDP checksum field is explicitly disallowed in RFC2460. Thus it may be expected that any device on the path that has a reason to look beyond the IP header, for example to validate the UDP checksum, will consider such a packet as erroneous or illegal and may discard it, unless the device is updated to support the new behavior. Any middlebox that modifies the UDP checksum, for example a NAT that changes the values of the IP and UDP header in such a way that the checksum over the pseudo header changes value, will need to be updated to support this behavior. Until then, a zero UDP checksum packet is likely to be discarded either directly in the middlebox or at the destination, when a zero UDP checksum has been modified to a non-zero by an incremental update.

A pair of end-points intending to use a new behavior will therefore not only need to ensure support at each end-point, but also that the path between them will deliver packets with the new behavior. This may require using negotiation or an explicit mandate to use the new behavior by all nodes that support the new protocol.

Enabling the use of a zero checksum places new requirements on equipment deployed within the network, such as middleboxes. A middlebox (e.g. Firewalls, Network Address Translators) may enable zero checksum usage for a particular range of ports. Note that checksum off-loading and operating system design may result in all IPv6 UDP traffic being sent with a calculated checksum. This requires middleboxes that are configured to enable a zero UDP checksum to continue to work with bidirectional UDP flows that use a zero UDP checksum in only one direction, and therefore they must not maintain separate state for a UDP flow based on its checksum usage.

Support along the path between end points can be guaranteed in limited deployments by appropriate configuration. In general, it can be expected to take time for deployment of any updated behaviour to become ubiquitous.

A sender will need to probe the path to verify the expected behavior. Path characteristics may change, and usage therefore should be robust and able to detect a failure of the path under normal usage and re-negotiate. Note that a bidirectional path does not necessarily support the same checksum usage in both the forward and return directions: Receipt of a datagram with a zero UDP checksum, does not imply that the remote endpoint can also receive a datagram with a zero UDP checksum. This will require periodic validation of the path, adding complexity to any solution using the new behavior.

3.4. Applicability of method

The update to the IPv6 specification defined in [I-D.ietf-6man-udpchecksums] only modifies IPv6 nodes that implement specific protocols designed to permit omission of a UDP checksum. This document therefore provides an applicability statement for the updated method indicating when the mechanism can (and can not) be used. Enabling this, and ensuring correct interactions with the stack, implies much more than simply disabling the checksum algorithm for specific packets at the transport interface.

When the method is widely available, it may be expected to be used by applications that are perceived to gain benefit. Any solution that uses an end-to-end transport protocol, rather than an IP-in-IP encapsulation, needs to minimise the possibility that application processes could confuse a corrupted or wrongly delivered UDP datagram with that of data addressed to the application running on their endpoint.

The protocol or application that uses the zero checksum method must ensure that the lack of checksum does not affect the protocol operation. This includes being robust to receiving a unintended packet from another protocol or context following corruption of a destination or source address and/or port value. It also includes considering the need for additional implicit protection mechanisms required when using the payload of a UDP packet received with a zero checksum.

3.5. Impact on non-supporting devices or applications

It is important to consider the potential impact of using a zero UDP checksum on end-point devices or applications that are not modified to support the new behavior or by default or preference, use the

regular behavior. These applications must not be significantly impacted by the update.

To illustrate why this necessary, consider the implications of a node that enables use of a zero UDP checksum at the interface level: This would result in all applications that listen to a UDP socket receiving datagrams where the checksum was not verified. This could have a significant impact on an application that was not designed with the additional robustness needed to handle received packets with corruption, creating state or destroying existing state in the application.

A zero UDP checksum therefore needs to be enabled only for individual ports using an explicit request by the application. In this case, applications using other ports would maintain the current IPv6 behavior, discarding incoming datagrams with a zero UDP checksum. These other applications would not be affected by this changed behavior. An application that allows the changed behavior should be aware of the risk of corruption and the increased level of misdirected traffic, and can be designed robustly to handle this risk.

4. Constraints on implementation of IPv6 nodes supporting zero checksum

This section is an applicability statement that defines requirements and recommendations on the implementation of IPv6 nodes that support use of a zero value in the checksum field of a UDP datagram.

All implementations that support this zero UDP checksum method **MUST** conform to the requirements defined below.

1. An IPv6 sending node **MAY** use a calculated RFC 2460 checksum for all datagrams that it sends. This explicitly permits an interface that supports checksum offloading to insert an updated UDP checksum value in all UDP datagrams that it forwards, however note that sending a calculated checksum requires the receiver to also perform the checksum calculation. Checksum offloading can normally be switched off for a particular interface to ensure that datagrams are sent with a zero UDP checksum.
2. IPv6 nodes **SHOULD** by default **NOT** allow the zero UDP checksum method for transmission.
3. IPv6 nodes **MUST** provide a way for the application/protocol to indicate the set of ports that will be enabled to send datagrams with a zero UDP checksum. This may be implemented by enabling a

transport mode using a socket API call when the socket is established, or a similar mechanism. It may also be implemented by enabling the method for a pre-assigned static port used by a specific tunnel protocol.

4. IPv6 nodes MUST provide a method to allow an application/protocol to indicate that a particular UDP datagram is required to be sent with a UDP checksum. This needs to be allowed by the operating system at any time (e.g. to send keep-alive datagrams), not just when a socket is established in the zero checksum mode.
5. The default IPv6 node receiver behaviour MUST discard all IPv6 packets carrying datagrams with a zero UDP checksum.
6. IPv6 nodes MUST provide a way for the application/protocol to indicate the set of ports that will be enabled to receive datagrams with a zero UDP checksum. This may be implemented via a socket API call, or similar mechanism. It may also be implemented by enabling the method for a pre-assigned static port used by a specific tunnel protocol.
7. IPv6 nodes supporting usage of zero UDP checksums MUST also allow reception using a calculated UDP checksum on all ports configured to allow zero UDP checksum usage. (The sending endpoint, e.g. encapsulating ingress, may choose to compute the UDP checksum, or may calculate this by default.) The receiving endpoint MUST use the reception method specified in RFC2460 when the checksum field is not zero.
8. RFC 2460 specifies that IPv6 nodes SHOULD log received datagrams with a zero UDP checksum. This remains the case for any datagram received on a port that does not explicitly enable processing of a zero UDP checksum. A port for which the zero UDP checksum has been enabled MUST NOT log the datagram solely because the checksum value is zero.
9. IPv6 nodes MAY separately identify received UDP datagrams that are discarded with a zero UDP checksum. It SHOULD NOT add these to the standard log, since the endpoint has not been verified. This may be used to support other functions (such as a security policy).
10. IPv6 nodes that receive ICMPv6 messages that refer to packets with a zero UDP checksum MUST provide appropriate checks concerning the consistency of the reported packet to verify that the reported packet actually originated from the node, before acting upon the information (e.g. validating the address and

port numbers in the ICMPv6 message body).

5. Requirements on usage of the zero UDP checksum

This section is an applicability statement that identifies requirements and recommendations for protocols and tunnel encapsulations that are transported over an IPv6 transport flow (e.g. tunnel) that does not perform a UDP checksum calculation to verify the integrity at the transport endpoints. Before deciding to use the zero UDP checksum and loose the integrity verification provided, a protocol developer should seriously consider if they can use checksummed UDP packets or UDP-Lite [RFC3828], because IPv6 with a zero UDP checksum is not equivalent in behavior to IPv4 with zero UDP checksum.

The requirements and recommendations for protocols and tunnel encapsulations using an IPv6 transport flow that does not perform a UDP checksum calculation to verify the integrity at the transport endpoints are:

1. Transported protocols that enable the use of zero UDP checksum MUST only enable this for a specific port or port-range. This needs to be enabled at the sending and receiving endpoints for a UDP flow.
2. An integrity mechanism is always RECOMMENDED at the transported protocol layer to ensure that corruption rates of the delivered payload is not increased (e.g. the inner-most packet of a UDP tunnel). A mechanism that isolates the causes of corruption (e.g. identifying misdelivery, IPv6 header corruption, tunnel header corruption) is expected to also provide additional information about the status of the tunnel (e.g. to suggest a security attack).
3. A transported protocol that encapsulates Internet Protocol (IPv4 or IPv6) packets MAY rely on the inner packet integrity checks, provided that the tunnel protocol will not significantly increase the rate of corruption of the inner IP packet. If a significantly increased corruption rate can occur, then the tunnel protocol MUST provide an additional integrity verification mechanism. Early detection is desirable to avoid wasting unnecessary computation, transmission capacity or storage for packets that will subsequently be discarded.
4. A transported protocol that supports use of a zero UDP checksum, MUST be designed so that corruption of this information does not result in accumulated state for the protocol.

5. A transported protocol with a non-tunnel payload or one that encapsulates non-IP packets **MUST** have a CRC or other mechanism for checking packet integrity, unless the non-IP packet is specifically designed for transmission over a lower layer that does not provide a packet integrity guarantee.
6. A transported protocol with control feedback **SHOULD** be robust to changes in the network path, since the set of middleboxes on a path may vary during the life of an association. The UDP endpoints need to discover paths with middleboxes that drop packets with a zero UDP checksum. Therefore, transported protocols **SHOULD** send keep-alive messages with a zero UDP checksum. An endpoint that discovers an appreciable loss rate for keep-alive packets **MAY** terminate the UDP flow (e.g. tunnel). Section 3.1.3 of RFC 5405 describes requirements for congestion control when using a UDP-based transport.
7. A protocol with control feedback that can fall-back to using UDP with a calculated RFC 2460 checksum is expected to be more robust to changes in the network path. Therefore, keep-alive messages **SHOULD** include both UDP datagrams with a checksum and datagrams with a zero UDP checksum. This will enable the remote endpoint to distinguish between a path failure and dropping of datagrams with a zero UDP checksum.
8. A middlebox implementation **MUST** allow forwarding of an IPv6 UDP datagram with both a zero and standard UDP checksum using the same UDP port.
9. A middlebox **MAY** configure a restricted set of specific port ranges that forward UDP datagrams with a zero UDP checksum. The middlebox **MAY** drop IPv6 datagrams with a zero UDP checksum that are outside a configured range.
10. When a middlebox forwards an IPv6 UDP flow containing datagrams with both a zero and standard UDP checksum, the middlebox **MUST** NOT maintain separate state for flows depending on the value of their UDP checksum field. (This requirement is necessary to enable a sender that always calculates a checksum to communicate via a middlebox with a remote endpoint that uses a zero UDP checksum.)

Special considerations are required when designing a UDP tunnel protocol, where the tunnel ingress or egress may be a router that may not have access to the packet payload. When the node is acting as a host (i.e., sending or receiving a packet addressed to itself), the checksum processing is similar to other hosts. However, when the node (e.g. a router) is acting as a tunnel ingress or egress that

forwards a packet to or from a UDP tunnel, there may be restricted access to the packet payload. This prevents calculating (or verifying) a UDP checksum. In this case, the tunnel protocol may use a zero UDP checksum and must:

- o Ensure that tunnel ingress and tunnel egress router are both configured to use a zero UDP checksum. For example, this may include ensuring that hardware checksum offloading is disabled.
- o The tunnel operator must ensure that middleboxes on the network path are updated to support use of a zero UDP checksum.
- o A tunnel egress should implement appropriate security techniques to protect from overload, including source address filtering to prevent traffic injection by an attacker, and rate-limiting of any packets that incur additional processing, such as UDP datagrams used for control functions that require verification of a calculated checksum to verify the network path. Usage of common control traffic for multiple tunnels between a pair of nodes can assist in reducing the number of packets to be processed.

6. Summary

This document provides an applicability statement for the use of UDP transport checksums with IPv6.

It examines the role of the UDP transport checksum when used with IPv6 and presents a summary of the trade-offs in evaluating the safety of updating RFC 2460 to permit an IPv6 endpoint to use a zero UDP checksum field to indicate that no checksum is present.

Application designers should first examine whether their transport goals may be met using standard UDP (with a calculated checksum) or by using UDP-Lite. The use of UDP with a zero UDP checksum has merits for some applications, such as tunnel encapsulation, and is widely used in IPv4. However, there are different dangers for IPv6: There is an increased risk of corruption and misdelivery when using zero UDP checksum in IPv6 compared to using IPv4 due to the lack of an IPv6 header checksum. Thus, applications need to evaluate the risks of enabling use of a zero UDP checksum and consider a solution that at least provides the same delivery protection as for IPv4, for example by utilizing UDP-Lite, or by enabling the UDP checksum. The use of checksum off-loading may help alleviate the cost of checksum processing and permit use of a checksum using method defined in RFC 2460.

Tunnel applications using UDP for encapsulation can in many cases use

a zero UDP checksum without significant impact on the corruption rate. A well-designed tunnel application should include consistency checks to validate the header information encapsulated with a received packet. In most cases, tunnels encapsulating IP packets can rely on the integrity protection provided by the transported protocol (or tunneled inner packet). When correctly implemented, such an endpoint will not be negatively impacted by omission of the transport-layer checksum. Recursive tunneling and fragmentation is a potential issue that can raise corruption rates significantly, and requires careful consideration.

Other UDP applications at the intended destination node or another node can be impacted if they are allowed to receive datagrams that have a zero UDP checksum. It is important that already deployed applications are not impacted by a change at the transport layer. If these applications execute on nodes that implement RFC 2460, they will discard (and log) all datagrams with a zero UDP checksum. This is not an issue.

In general, UDP-based applications need to employ a mechanism that allows a large percentage of the corrupted packets to be removed before they reach an application, both to protect the data stream of the application and the control plane of higher layer protocols. These checks are currently performed by the UDP checksum for IPv6, or the reduced checksum for UDP-Lite when used with IPv6.

The transport of recursive tunneling and the use of fragmentation pose difficult issues that need to be considered in the design of tunnel protocols. There is an increased risk of an error in the inner-most packet when fragmentation when several layers of tunneling and several different reassembly processes are run without verification of correctness. This requires extra thought and careful consideration in the design of transported tunnels.

Any use of the updated method must consider the implications on firewalls, NATs and other middleboxes. It is not expected that IPv6 NATs handle IPv6 UDP datagrams in the same way that they handle IPv4 UDP datagrams. In many deployed cases this will require an update to support an IPv6 zero UDP checksum. Firewalls are intended to be configured, and therefore may need to be explicitly updated to allow new services or protocols. IPv6 middlebox deployment is not yet as prolific as it is in IPv4, and therefore new devices are expected to follow the methods specified in this document.

Each application should consider the implications of choosing an IPv6 transport that uses a zero UDP checksum, and consider whether other standard methods may be more appropriate, and may simplify application design.

7. Acknowledgements

Brian Haberman, Brian Carpenter, Margaret Wasserman, Lars Eggert, others in the TSV directorate. Barry Leiba, Ronald Bonica, Pete Resnick, and Stewart Bryant are thanked for resulting in a document with much greater applicability. Thanks to P.F. Chimento for careful review and editorial corrections.

Thanks also to: Remi Denis-Courmont, Pekka Savola, Glen Turner, and many others who contributed comments and ideas via the 6man, behave, lisp and mboned lists.

8. IANA Considerations

This document does not require any actions by IANA.

9. Security Considerations

Transport checksums provide the first stage of protection for the stack, although they can not be considered authentication mechanisms. These checks are also desirable to ensure packet counters correctly log actual activity, and can be used to detect unusual behaviours.

Depending on the hardware design, the processing requirements may differ for tunnels that have a zero UDP checksum and those that calculate a checksum. This processing overhead may need to be considered when deciding whether to enable a tunnel and to determine an acceptable rate for transmission. This can become a security risk for designs that can handle a significantly larger number of packets with zero UDP checksums compared to datagrams with a non-zero checksum, such as tunnel egress. An attacker could attempt to inject non-zero checksummed UDP packets into a tunnel forwarding zero checksum UDP packets and cause overload in the processing of the non-zero checksums, e.g. if this happens in a routers slow path. Protection mechanisms should therefore be employed when this threat exists. Protection may include source address filtering to prevent an attacker injecting traffic, as well as throttling the amount of non-zero checksum traffic. The latter may impact the function of the tunnel protocol.

Transmission of IPv6 packets with a zero UDP checksum could reveal additional information to an on-path attacker to identify the operating system or configuration of a sending node. There is a need to probe the network path to determine whether the current path supports using IPv6 packets with a zero UDP checksum. The details of the probing mechanism may differ for different tunnel encapsulations

and if visible in the network (e.g. if not using IPsec in encryption mode) could reveal additional information to an on-path attacker to identify the type of tunnel being used.

IP-in-IP or GRE tunnels offer good traversal of middleboxes that have not been designed for security, e.g. firewalls. However, firewalls may be expected to be configured to block general tunnels as they present a large attack surface. This applicability statement therefore permits this method to be enabled only for specific ranges of ports.

When the zero UDP checksum mode is enabled for a range of ports, nodes and middleboxes must forward received UDP datagrams that have either a calculated checksum or a zero checksum.

10. References

10.1. Normative References

- [I-D.ietf-6man-udpchecksums]
Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", draft-ietf-6man-udpchecksums-08 (work in progress), February 2013.
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, September 1981.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, December 1998.

10.2. Informative References

- [I-D.ietf-intarea-tunnels]
Touch, J. and M. Townsley, "Tunnels in the Internet Architecture", draft-ietf-intarea-tunnels-00 (work in progress), March 2010.
- [I-D.ietf-mboned-auto-multicast]
Bumgardner, G., "Automatic Multicast Tunneling", draft-ietf-mboned-auto-multicast-14 (work in progress),

June 2012.

- [LISP] D. Farinacci et al, "Locator/ID Separation Protocol (LISP)", November 2012.
- [RFC1071] Braden, R., Borman, D., Partridge, C., and W. Plummer, "Computing the Internet checksum", RFC 1071, September 1988.
- [RFC1141] Mallory, T. and A. Kullberg, "Incremental updating of the Internet checksum", RFC 1141, January 1990.
- [RFC1624] Rijssinghani, A., "Computation of the Internet Checksum via Incremental Update", RFC 1624, May 1994.
- [RFC2827] Ferguson, P. and D. Senie, "Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing", BCP 38, RFC 2827, May 2000.
- [RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.
- [RFC3819] Karn, P., Bormann, C., Fairhurst, G., Grossman, D., Ludwig, R., Mahdavi, J., Montenegro, G., Touch, J., and L. Wood, "Advice for Internet Subnetwork Designers", BCP 89, RFC 3819, July 2004.
- [RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004.
- [RFC4443] Conta, A., Deering, S., and M. Gupta, "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", RFC 4443, March 2006.
- [RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, July 2007.
- [RFC5097] Renker, G. and G. Fairhurst, "MIB for the UDP-Lite protocol", RFC 5097, January 2008.
- [RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers", BCP 145, RFC 5405, November 2008.
- [RFC5415] Calhoun, P., Montemurro, M., and D. Stanley, "Control And Provisioning of Wireless Access Points (CAPWAP) Protocol

Specification", RFC 5415, March 2009.

[RFC5722] Krishnan, S., "Handling of Overlapping IPv6 Fragments", RFC 5722, December 2009.

[RFC6437] Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme, "IPv6 Flow Label Specification", RFC 6437, November 2011.

[RFC6438] Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, November 2011.

[Sigcomm2000] Jonathan Stone and Craig Partridge , "When the CRC and TCP Checksum Disagree", 2000.

[UDPTT] G Fairhurst, "The UDP Tunnel Transport mode", Feb 2010.

Appendix A. Evaluation of proposal to update RFC 2460 to support zero checksum

This informative appendix documents the evaluation of the proposal to update IPv6 [RFC2460], to provide the option that some nodes may suppress generation and checking of the UDP transport checksum. It also compares the proposal with other alternatives, and notes that for a particular application some standard methods may be more appropriate than using IPv6 with a zero UDP checksum.

A.1. Alternatives to the Standard Checksum

There are several alternatives to the normal method for calculating the UDP Checksum [RFC1071] that do not require a tunnel endpoint to inspect the entire packet when computing a checksum. These include (in decreasing order of complexity):

- o Delta computation of the checksum from an encapsulated checksum field. Since the checksum is a cumulative sum [RFC1624], an encapsulating header checksum can be derived from the new pseudo header, the inner checksum and the sum of the other network-layer fields not included in the pseudo header of the encapsulated packet, in a manner resembling incremental checksum update [RFC1141]. This would not require access to the whole packet, but does require fields to be collected across the header, and arithmetic operations on each packet. The method would only work for packets that contain a 2's complement transport checksum (i.e., it would not be appropriate for SCTP or when IP fragmentation is used).

- o UDP-Lite with the checksum coverage set to only the header portion of a packet. This requires a pseudo header checksum calculation only on the encapsulating packet header. The computed checksum value may be cached (before adding the Length field) for each flow/destination and subsequently combined with the Length of each packet to minimise per-packet processing. This value is combined with the UDP payload length for the pseudo header, however this length is expected to be known when performing packet forwarding.
- o The proposed UDP Tunnel Transport [UDPTT] suggested a method where UDP would be modified to derive the checksum only from the encapsulating packet protocol header. This value does not change between packets in a single flow. The value may be cached per flow/destination to minimise per-packet processing.
- o There has been a proposal to simply ignore the UDP checksum value on reception at the tunnel egress, allowing a tunnel ingress to insert any value correct or false. For tunnel usage, a non standard checksum value may be used, forcing an RFC 2460 receiver to drop the packet. The main downside is that it would be impossible to identify a UDP datagram (in the network or an endpoint) that is treated in this way compared to a packet that has actually been corrupted.
- o A method has been proposed that uses a new (to be defined) IPv6 Destination Options Header to provide an end-to-end validation check at the network layer. This would allow an endpoint to verify delivery to an appropriate end point, but would also require IPv6 nodes to correctly handle the additional header, and would require changes to middlebox behavior (e.g. when used with a NAT that always adjusts the checksum value).
- o UDP modified to disable checksum processing [I-D.ietf-6man-udpchecksums]. This eliminates the need for a checksum calculation, but would require constraints on appropriate usage and updates to end-points and middleboxes.
- o IP-in-IP tunneling. As this method completely dispenses with a transport protocol in the outer-layer it has reduced overhead and complexity, but also reduced functionality. There is no outer checksum over the packet and also no ports to perform demultiplexing between different tunnel types. This reduces the information available upon which a load balancer may act.

These options are compared and discussed further in the following sections.

A.2. Comparison

This section compares the above listed methods to support datagram tunneling. It includes proposals for updating the behaviour of UDP.

While this comparison focuses on applications that are expected to execute on routers, the distinction between a router and a host is not always clear, especially at the transport level. Systems (such as unix-based operating systems) routinely provide both functions. There is no way to identify the role of the receiving node from a received packet.

A.2.1. Middlebox Traversal

Regular UDP with a standard checksum or the delta encoded optimization for creating correct checksums have the best possibilities for successful traversal of a middlebox. No new support is required.

A method that ignores the UDP checksum on reception is expected to have a good probability of traversal, because most middleboxes perform an incremental checksum update. UDPTT would also have been able to traverse a middlebox with this behaviour. However, a middlebox on the path that attempts to verify a standard checksum will not forward packets using either of these methods, preventing traversal. A method that ignores the checksum has an additional downside in that it prevents improvement of middlebox traversal, because there is no way to identify UDP datagrams that use the modified checksum behaviour.

IP-in-IP or GRE tunnels offer good traversal of middleboxes that have not been designed for security, e.g. firewalls. However, firewalls may be expected to be configured to block general tunnels as they present a large attack surface.

A new IPv6 Destination Options header will suffer traversal issues with middleboxes, especially Firewalls and NATs, and will likely require them to be updated before the extension header is passed.

Datagrams with a zero UDP checksum will not be passed by any middlebox that validates the checksum using RFC 2460 or updates the checksum field, such as NAT or firewalls. This would require an update to correctly handle a datagram with a zero UDP checksum.

UDP-Lite will require an update of almost all type of middleboxes, because it requires support for a separate network-layer protocol number. Once enabled, the method to support incremental checksum update would be identical to that for UDP, but different for checksum

validation.

A.2.2. Load Balancing

The usefulness of solutions for load balancers depends on the difference in entropy in the headers for different flows that can be included in a hash function. All the proposals that use the UDP protocol number have equal behavior. UDP-Lite has the potential for equally good behavior as for UDP. However, UDP-Lite is currently unlikely to be supported by deployed hashing mechanisms, which could cause a load balancer to not use the transport header in the computed hash. A load balancer that only uses the IP header will have low entropy, but could be improved by including the IPv6 the flow label, providing that the tunnel ingress ensures that different flow labels are assigned to different flows. However, a transition to the common use of good quality flow labels is likely to take time to deploy.

A.2.3. Ingress and Egress Performance Implications

IP-in-IP tunnels are often considered efficient, because they introduce very little processing and low data overhead. The other proposals introduce a UDP-like header incurring associated data overhead. Processing is minimised for the method that uses a zero UDP checksum, ignoring the UDP checksum on reception, and only slightly higher for UDPTT, the extension header and UDP-Lite. The delta-calculation scheme operates on a few more fields, but also introduces serious failure modes that can result in a need to calculate a checksum over the complete datagram. Regular UDP is clearly the most costly to process, always requiring checksum calculation over the entire datagram.

It is important to note that the zero UDP checksum method, ignoring checksum on reception, the Option Header, UDPTT and UDP-Lite will likely incur additional complexities in the application to incorporate a negotiation and validation mechanism.

A.2.4. Deployability

The major factors influencing deployability of these solutions are a need to update both end-points, a need for negotiation and the need to update middleboxes. These are summarised below:

- o The solution with the best deployability is regular UDP. This requires no changes and has good middlebox traversal characteristics.
- o The next easiest to deploy is the delta checksum solution. This does not modify the protocol on the wire and only needs changes in

tunnel ingress.

- o IP-in-IP tunnels should not require changes to the end-points, but raise issues when traversing firewalls and other security devices, which are expected to require updates.
- o Ignoring the checksum on reception will require changes at both end-points. The never ceasing risk of path failure requires additional checks to ensure this solution is robust and will require changes or additions to the tunnel control protocol to negotiate support and validate the path.
- o The remaining solutions (including the zero checksum method) offer similar deployability. UDP-Lite requires support at both end-points and in middleboxes. UDPTT and the zero UDP checksum method with or without an extension header require support at both end-points and in middleboxes. UDP-Lite, UDPTT, and the zero UDP checksum method and use of extension headers may additionally require changes or additions to the tunnel control protocol to negotiate support and path validation.

A.2.5. Corruption Detection Strength

The standard UDP checksum and the delta checksum can both provide some verification at the tunnel egress. This can significantly reduce the probability that a corrupted inner packet is forwarded. UDP-Lite, UDPTT and the extension header all provide some verification against corruption, but do not verify the inner packet. They only provide a strong indication that the delivered packet was intended for the tunnel egress and was correctly delimited.

The methods using a zero UDP checksum, ignoring the UDP checksum on reception and IP-and-IP encapsulation all provide no verification that a received datagram was intended to be processed by a specific tunnel egress or that the inner encapsulated packet was correct. Section 3.1 discusses experience using specific protocols in well-managed networks.

A.2.6. Comparison Summary

The comparisons above may be summarised as "there is no silver bullet that will slay all the issues". One has to select which down side(s) can best be lived with. Focusing on the existing solutions, this can be summarized as:

Regular UDP: The method defined in RFC 2460 has good middlebox traversal and load balancing and multiplexing, requiring a checksum in the outer headers covering the whole packet.

IP in IP: A low complexity encapsulation, with limited middlebox traversal, no multiplexing support, and currently poor load balancing support that could improve over time.

UDP-Lite: A medium complexity encapsulation, with good multiplexing support, limited middlebox traversal, but possible to improve over time, currently poor load balancing support that could improve over time, in most cases requiring application level negotiation to select the protocol and validation to confirm the path forwards UDP-Lite.

The delta-checksum is an optimization in the processing of UDP, as such it exhibits some of the drawbacks of using regular UDP.

The remaining proposals may be described in similar terms:

Zero-Checksum: A low complexity encapsulation, with good multiplexing support, limited middlebox traversal that could improve over time, good load balancing support, in most cases requiring application level negotiation and validation to confirm the path forwards a zero UDP checksum.

UDPTT: A medium complexity encapsulation, with good multiplexing support, limited middlebox traversal, but possible to improve over time, good load balancing support, in most cases requiring application level negotiation to select the transport and validation to confirm the path forwards UDPTT datagrams.

IPv6 Destination Option IP in IP tunneling: A medium complexity, with no multiplexing support, limited middlebox traversal, currently poor load balancing support that could improve over time, in most cases requiring negotiation to confirm the option is supported and validation to confirm the path forwards the option.

IPv6 Destination Option combined with UDP Zero-checksumming: A medium complexity encapsulation, with good multiplexing support, limited load balancing support that could improve over time, in most cases requiring negotiation to confirm the option is supported and validation to confirm the path forwards the option.

Ignore the checksum on reception: A low complexity encapsulation, with good multiplexing support, medium middlebox traversal that never can improve, good load balancing support, in most cases requiring negotiation to confirm the option is supported by the

remote endpoint and validation to confirm the path forwards a zero UDP checksum.

There is no clear single optimum solution. If the most important need is to traverse middleboxes, then the best choice is to stay with regular UDP and consider the optimizations that may be required to perform the checksumming. If one can live with limited middlebox traversal, low complexity is necessary and one does not require load balancing, then IP-in-IP tunneling is the simplest. If one wants strengthened error detection, but with currently limited middlebox traversal and load-balancing. UDP-Lite is appropriate. Zero UDP checksum addresses another set of constraints, low complexity and a need for load balancing from the current Internet, providing it can live with currently limited middlebox traversal.

Techniques for load balancing and middlebox traversal do continue to evolve. Over a long time, developments in load balancing have good potential to improve. This time horizon is long since it requires both load balancer and end-point updates to get full benefit. The challenges of middlebox traversal are also expected to change with time, as device capabilities evolve. Middleboxes are very prolific with a larger proportion of end-user ownership, and therefore may be expected to take long time cycles to evolve.

One potential advantage is that the deployment of IPv6-capable middleboxes are still in its initial phase and the quicker a new method becomes standardized, the fewer boxes will be non-compliant.

Thus, the question of whether to permit use of datagrams with a zero UDP checksum for IPv6 under reasonable constraints, is therefore best viewed as a trade-off between a number of more subjective questions:

- o Is there sufficient interest in using a zero UDP checksum with the given constraints (summarised below)?
- o Are there other avenues of change that will resolve the issue in a better way and sufficiently quickly ?
- o Do we accept the complexity cost of having one more solution in the future?

The analysis concludes that the IETF should carefully consider constraints on sanctioning the use of any new transport mode. The 6man working group of the IETF has determined that the answer to the above questions are sufficient to update IPv6 to standardise use of a zero UDP checksum for use by tunnel encapsulations for specific applications.

Each application should consider the implications of choosing an IPv6 transport that uses a zero UDP checksum. In many cases, standard methods may be more appropriate, and may simplify application design. The use of checksum off-loading may help alleviate the checksum processing cost and permit use of a checksum using method defined in RFC 2460.

Appendix B. Document Change History

{RFC EDITOR NOTE: This section must be deleted prior to publication}

Individual Draft 00 This is the first DRAFT of this document - It contains a compilation of various discussions and contributions from a variety of IETF WGs, including: mboned, tsv, 6man, lisp, and behave. This includes contributions from Magnus with text on RTP, and various updates.

Individual Draft 01

- * This version corrects some typos and editorial NiTs and adds discussion of the need to negotiate and verify operation of a new mechanism (3.3.4).

Individual Draft 02

- * Version -02 corrects some typos and editorial NiTs.
- * Added reference to ECMP for tunnels.
- * Clarifies the recommendations at the end of the document.

Working Group Draft 00

- * Working Group Version -00 corrects some typos and removes much of rationale for UDPTT. It also adds some discussion of IPv6 extension header.

Working Group Draft 01

- * Working Group Version -01 updates the rules and incorporates off-list feedback. This version is intended for wider review within the 6man working group.

Working Group Draft 02

- * This version is the result of a major rewrite and re-ordering of the document.
- * A new section comparing the results have been added.
- * The constraints list has been significantly altered by removing some and rewording other constraints.
- * This contains other significant language updates to clarify the intent of this draft.

Working Group Draft 03

- * Editorial updates

Working Group Draft 04

- * Resubmission only updating the AMT and RFC2765 references.

Working Group Draft 05

- * Resubmission to correct editorial NiTs - thanks to Bill Atwood for noting these. Group Draft 05.

Working Group Draft 06

- * Resubmission to keep draft alive (spelling updated from 05).

Working Group Draft 07

- * Interim Version
- * Submission after IESG Feedback Added
- * Updates to enable the document to become a PS Applicability Statement

Working Group Draft 08

- * First Version written as a PS Applicability Statement
- * Changes to reflect decision to update RFC 2460, rather than recommend decision
- * Updates to requirements for middleboxes

- * Inclusion of requirements for security, API, and tunnel
- * Move of the rationale for the update to an Annex (former section 4)

Working Group Draft 09

- * Submission after second WGLC (note mistake corrected in -09).
- * Clarified role of API for supporting full checksum.
- * Clarified that full checksum is required in security considerations, and therefore noting that full checksum should not be treated as an attack - consistent with remainder of document.
- * Added mention that API can set a mode in transport stack - to link to similar statement in RFC 2460 update.
- * Fixed typos.

Working Group Draft 10

- * Submission to correct unwanted removal of text from section 5 bullets 5-7 by GF.
- * Replaced section 5 text with the text from 08, and reapplied the editorial correction.
- * Note to reviewers: Please compare this revision with -08 used in the IETF LC).

Working Group Draft 11

- * Added REF for 5097 (Noted by S.Turner)
- * Added text in response to P. Resnick on place where checksum is calculated.
- * Added text to note experience with MPLS/PWE; Appendix updated to refer to this (S. Bryant)
- * Added text in response to P.Resnick's 2nd comments.
- * Request to make UDP-Lite more clearly recommended (J Touch, P.Resnick)

- * Added considerations around usage of zero checksum in routers.
- * Added text in response to Stewart Bryant's comments on router requirements.

Authors' Addresses

Godred Fairhurst
University of Aberdeen
School of Engineering
Aberdeen, AB24 3UE
Scotland, UK

Email: gorry@erg.abdn.ac.uk
URI: <http://www.erg.abdn.ac.uk/users/gorry>

Magnus Westerlund
Ericsson
Farogatan 6
Stockholm, SE-164 80
Sweden

Phone: +46 8 719 0000
Email: magnus.westerlund@ericsson.com

Transport Area Working Group
Internet-Draft
Updates: 2309 (if approved)
Intended status: BCP
Expires: May 11, 2014

B. Briscoe
BT
J. Manner
Aalto University
November 07, 2013

Byte and Packet Congestion Notification
draft-ietf-tsvwg-byte-pkt-congest-12

Abstract

This document provides recommendations of best current practice for dropping or marking packets using any active queue management (AQM) algorithm, including random early detection (RED), BLUE, pre-congestion notification (PCN) and newer schemes such as CoDel (Controlled Delay) and PIE (Proportional Integral controller Enhanced). We give three strong recommendations: (1) packet size should be taken into account when transports detect and respond to congestion indications, (2) packet size should not be taken into account when network equipment creates congestion signals (marking, dropping), and therefore (3) in the specific case of RED, the byte-mode packet drop variant that drops fewer small packets should not be used. This memo updates RFC 2309 to deprecate deliberate preferential treatment of small packets in AQM algorithms.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on May 11, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Terminology and Scoping	6
1.2. Example Comparing Packet-Mode Drop and Byte-Mode Drop . .	7
2. Recommendations	9
2.1. Recommendation on Queue Measurement	9
2.2. Recommendation on Encoding Congestion Notification	10
2.3. Recommendation on Responding to Congestion	11
2.4. Recommendation on Handling Congestion Indications when Splitting or Merging Packets	12
3. Motivating Arguments	12
3.1. Avoiding Perverse Incentives to (Ab)use Smaller Packets .	12
3.2. Small != Control	14
3.3. Transport-Independent Network	14
3.4. Partial Deployment of AQM	15
3.5. Implementation Efficiency	17
4. A Survey and Critique of Past Advice	17
4.1. Congestion Measurement Advice	18
4.1.1. Fixed Size Packet Buffers	18
4.1.2. Congestion Measurement without a Queue	19
4.2. Congestion Notification Advice	20
4.2.1. Network Bias when Encoding	20
4.2.2. Transport Bias when Decoding	22
4.2.3. Making Transports Robust against Control Packet Losses	23
4.2.4. Congestion Notification: Summary of Conflicting Advice	24
5. Outstanding Issues and Next Steps	25
5.1. Bit-congestible Network	25
5.2. Bit- & Packet-congestible Network	25
6. Security Considerations	26
7. IANA Considerations	26
8. Conclusions	26
9. Acknowledgements	28
10. Comments Solicited	28
11. References	28
11.1. Normative References	28
11.2. Informative References	28
Appendix A. Survey of RED Implementation Status	32
Appendix B. Sufficiency of Packet-Mode Drop	34
B.1. Packet-Size (In)Dependence in Transports	35
B.2. Bit-Congestible and Packet-Congestible Indications	38
Appendix C. Byte-mode Drop Complicates Policing Congestion Response	39
Appendix D. Changes from Previous Versions	40

1. Introduction

This document provides recommendations of best current practice for how we should correctly scale congestion control functions with respect to packet size for the long term. It also recognises that expediency may be necessary to deal with existing widely deployed protocols that don't live up to the long term goal.

When signalling congestion, the problem of how (and whether) to take packet sizes into account has exercised the minds of researchers and practitioners for as long as active queue management (AQM) has been discussed. Indeed, one reason AQM was originally introduced was to reduce the lock-out effects that small packets can have on large packets in drop-tail queues. This memo aims to state the principles we should be using and to outline how these principles will affect future protocol design, taking into account the existing deployments we have already.

The question of whether to take into account packet size arises at three stages in the congestion notification process:

Measuring congestion: When a congested resource measures locally how congested it is, should it measure its queue length in time, bytes or packets?

Encoding congestion notification into the wire protocol: When a congested network resource signals its level of congestion, should it drop / mark each packet dependent on the size of the particular packet in question?

Decoding congestion notification from the wire protocol: When a transport interprets the notification in order to decide how much to respond to congestion, should it take into account the size of each missing or marked packet?

Consensus has emerged over the years concerning the first stage, which Section 2.1 records in the RFC Series. In summary: If possible it is best to measure congestion by time in the queue, but otherwise the choice between bytes and packets solely depends on whether the resource is congested by bytes or packets.

The controversy is mainly around the last two stages: whether to allow for the size of the specific packet notifying congestion i) when the network encodes or ii) when the transport decodes the congestion notification.

Currently, the RFC series is silent on this matter other than a paper trail of advice referenced from [RFC2309], which conditionally

recommends byte-mode (packet-size dependent) drop [pktByteEmail]. Reducing drop of small packets certainly has some tempting advantages: i) it drops less control packets, which tend to be small and ii) it makes TCP's bit-rate less dependent on packet size. However, there are ways of addressing these issues at the transport layer, rather than reverse engineering network forwarding to fix the problems.

This memo updates [RFC2309] to deprecate deliberate preferential treatment of packets in AQM algorithms solely because of their size. It recommends that (1) packet size should be taken into account when transports detect and respond to congestion indications, (2) not when network equipment creates them. This memo also adds to the congestion control principles enumerated in BCP 41 [RFC2914].

In the particular case of Random early Detection (RED), this means that the byte-mode packet drop variant should not be used to drop fewer small packets, because that creates a perverse incentive for transports to use tiny segments, consequently also opening up a DoS vulnerability. Fortunately all the RED implementers who responded to our admittedly limited survey (Section 4.2.4) have not followed the earlier advice to use byte-mode drop, so the position this memo argues for seems to already exist in implementations.

However, at the transport layer, TCP congestion control is a widely deployed protocol that doesn't scale with packet size (i.e. its reduction in rate does not take into account the size of a lost packet). To date this hasn't been a significant problem because most TCP implementations have been used with similar packet sizes. But, as we design new congestion control mechanisms, this memo recommends that we should build in scaling with packet size rather than assuming we should follow TCP's example.

This memo continues as follows. First it discusses terminology and scoping. Section 2 gives the concrete formal recommendations, followed by motivating arguments in Section 3. We then critically survey the advice given previously in the RFC series and the research literature (Section 4), referring to an assessment of whether or not this advice has been followed in production networks (Appendix A). To wrap up, outstanding issues are discussed that will need resolution both to inform future protocol designs and to handle legacy (Section 5). Then security issues are collected together in Section 6 before conclusions are drawn in Section 8. The interested reader can find discussion of more detailed issues on the theme of byte vs. packet in the appendices.

This memo intentionally includes a non-negligible amount of material on the subject. For the busy reader Section 2 summarises the

recommendations for the Internet community.

1.1. Terminology and Scoping

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

This memo applies to the design of all AQM algorithms, for example, Random Early Detection (RED) [RFC2309], BLUE [BLUE02], Pre-Congestion Notification (PCN) [RFC5670], Controlled Delay (CoDel) [I-D.nichols-tsvwg-codel] and the Proportional Integral controller Enhanced (PIE) [I-D.pan-tsvwg-pie]. Throughout, RED is used as a concrete example because it is a widely known and deployed AQM algorithm. There is no intention to imply that the advice is any less applicable to the other algorithms, nor that RED is preferred.

Congestion Notification: Congestion notification is a changing signal that aims to communicate the probability that the network resource(s) will not be able to forward the level of traffic load offered (or that there is an impending risk that they will not be able to).

The 'impending risk' qualifier is added, because AQM systems set a virtual limit smaller than the actual limit to the resource, then notify when this virtual limit is exceeded in order to avoid uncontrolled congestion of the actual capacity.

Congestion notification communicates a real number bounded by the range [0 , 1]. This ties in with the most well-understood measure of congestion notification: drop probability.

Explicit and Implicit Notification: The byte vs. packet dilemma concerns congestion notification irrespective of whether it is signalled implicitly by drop or using Explicit Congestion Notification (ECN [RFC3168] or PCN [RFC5670]). Throughout this document, unless clear from the context, the term marking will be used to mean notifying congestion explicitly, while congestion notification will be used to mean notifying congestion either implicitly by drop or explicitly by marking.

Bit-congestible vs. Packet-congestible: If the load on a resource depends on the rate at which packets arrive, it is called packet-congestible. If the load depends on the rate at which bits arrive it is called bit-congestible.

Examples of packet-congestible resources are route look-up engines and firewalls, because load depends on how many packet headers

they have to process. Examples of bit-congestible resources are transmission links, radio power and most buffer memory, because the load depends on how many bits they have to transmit or store. Some machine architectures use fixed size packet buffers, so buffer memory in these cases is packet-congestible (see Section 4.1.1).

The path through a machine will typically encounter both packet-congestible and bit-congestible resources. However, currently, a design goal of network processing equipment such as routers and firewalls is to size the packet-processing engine(s) relative to the lines in order to keep packet processing uncongested even under worst case packet rates with runs of minimum size packets. Therefore, packet-congestion is currently rare [RFC6077; S.3.3], but there is no guarantee that it will not become more common in future.

Note that information is generally processed or transmitted with a minimum granularity greater than a bit (e.g. octets). The appropriate granularity for the resource in question should be used, but for the sake of brevity we will talk in terms of bytes in this memo.

Coarser Granularity: Resources may be congestible at higher levels of granularity than bits or packets, for instance stateful firewalls are flow-congestible and call-servers are session-congestible. This memo focuses on congestion of connectionless resources, but the same principles may be applicable for congestion notification protocols controlling per-flow and per-session processing or state.

RED Terminology: In RED whether to use packets or bytes when measuring queues is called respectively "packet-mode queue measurement" or "byte-mode queue measurement". And whether the probability of dropping a particular packet is independent or dependent on its size is called respectively "packet-mode drop" or "byte-mode drop". The terms byte-mode and packet-mode should not be used without specifying whether they apply to queue measurement or to drop.

1.2. Example Comparing Packet-Mode Drop and Byte-Mode Drop

Taking RED as a well-known example algorithm, a central question addressed by this document is whether to recommend RED's packet-mode drop variant and to deprecate byte-mode drop. Table 1 compares how packet-mode and byte-mode drop affect two flows of different size packets. For each it gives the expected number of packets and of bits dropped in one second. Each example flow runs at the same bit-

rate of 48Mb/s, but one is broken up into small 60 byte packets and the other into large 1500 byte packets.

To keep up the same bit-rate, in one second there are about 25 times more small packets because they are 25 times smaller. As can be seen from the table, the packet rate is 100,000 small packets versus 4,000 large packets per second (pps).

Parameter	Formula	Small packets	Large packets
Packet size	$s/8$	60B	1,500B
Packet size	s	480b	12,000b
Bit-rate	x	48Mbps	48Mbps
Packet-rate	$u = x/s$	100kpps	4kpps
Packet-mode Drop			
Pkt loss probability	p	0.1%	0.1%
Pkt loss-rate	$p*u$	100pps	4pps
Bit loss-rate	$p*u*s$	48kbps	48kbps
Byte-mode Drop MTU, $M=12,000b$			
Pkt loss probability	$b = p*s/M$	0.004%	0.1%
Pkt loss-rate	$b*u$	4pps	4pps
Bit loss-rate	$b*u*s$	1.92kbps	48kbps

Table 1: Example Comparing Packet-mode and Byte-mode Drop

For packet-mode drop, we illustrate the effect of a drop probability of 0.1%, which the algorithm applies to all packets irrespective of size. Because there are 25 times more small packets in one second, it naturally drops 25 times more small packets, that is 100 small packets but only 4 large packets. But if we count how many bits it drops, there are 48,000 bits in 100 small packets and 48,000 bits in 4 large packets--the same number of bits of small packets as large.

The packet-mode drop algorithm drops any bit with the same probability whether the bit is in a small or a large packet.

For byte-mode drop, again we use an example drop probability of 0.1%, but only for maximum size packets (assuming the link maximum transmission unit (MTU) is 1,500B or 12,000b). The byte-mode algorithm reduces the drop probability of smaller packets proportional to their size, making the probability that it drops a small packet 25 times smaller at 0.004%. But there are 25 times more small packets, so dropping them with 25 times lower probability results in dropping the same number of packets: 4 drops in both cases. The 4 small dropped packets contain 25 times less bits than the 4 large dropped packets: 1,920 compared to 48,000.

The byte-mode drop algorithm drops any bit with a probability proportionate to the size of the packet it is in.

2. Recommendations

This section gives recommendations related to network equipment in Sections 2.1 and 2.2, and in Sections 2.3 and 2.4 we discuss the implications on the transport protocols.

2.1. Recommendation on Queue Measurement

Ideally, an AQM would measure the service time of the queue to measure congestion of a resource. However service time can only be measured as packets leave the queue, where it is not always expedient to implement a full AQM algorithm. To predict the service time as packets join the queue, an AQM algorithm needs to measure the length of the queue.

In this case, if the resource is bit-congestible, the AQM implementation SHOULD measure the length of the queue in bytes and, if the resource is packet-congestible, the implementation SHOULD measure the length of the queue in packets. Subject to the exceptions below, no other choice makes sense, because the number of packets waiting in the queue isn't relevant if the resource gets congested by bytes and vice versa. For example, the length of the queue into a transmission line would be measured in bytes, while the length of the queue into a firewall would be measured in packets.

To avoid the pathological effects of drop tail, the AQM can then transform this service time or queue length into the probability of dropping or marking a packet (e.g. RED's piecewise linear function between thresholds).

What this advice means for RED as a specific example:

1. A RED implementation SHOULD use byte mode queue measurement for measuring the congestion of bit-congestible resources and packet mode queue measurement for packet-congestible resources.
2. An implementation SHOULD NOT make it possible to configure the way a queue measures itself, because whether a queue is bit-congestible or packet-congestible is an inherent property of the queue.

Exceptions to these recommendations might be necessary, for instance where a packet-congestible resource has to be configured as a proxy bottleneck for a bit-congestible resource in an adjacent box that does not support AQM.

The recommended approach in less straightforward scenarios, such as fixed size packet buffers, resources without a queue and buffers comprising a mix of packet and bit-congestible resources, is discussed in Section 4.1. For instance, Section 4.1.1 explains that the queue into a line should be measured in bytes even if the queue consists of fixed-size packet-buffers, because the root-cause of any congestion is bytes arriving too fast for the line--packets filling buffers are merely a symptom of the underlying congestion of the line.

2.2. Recommendation on Encoding Congestion Notification

When encoding congestion notification (e.g. by drop, ECN or PCN), the probability that network equipment drops or marks a particular packet to notify congestion SHOULD NOT depend on the size of the packet in question. As the example in Section 1.2 illustrates, to drop any bit with probability 0.1% it is only necessary to drop every packet with probability 0.1% without regard to the size of each packet.

This approach ensures the network layer offers sufficient congestion information for all known and future transport protocols and also ensures no perverse incentives are created that would encourage transports to use inappropriately small packet sizes.

What this advice means for RED as a specific example:

1. The RED AQM algorithm SHOULD NOT use byte-mode drop, i.e. it ought to use packet-mode drop. Byte-mode drop is more complex, it creates the perverse incentive to fragment segments into tiny pieces and it is vulnerable to floods of small packets.
2. If a vendor has implemented byte-mode drop, and an operator has turned it on, it is RECOMMENDED to switch it to packet-mode drop, after establishing if there are any implications on the relative performance of applications using different packet sizes. The unlikely possibility of some application-specific legacy use of byte-mode drop is the only reason that all the above recommendations on encoding congestion notification are not phrased more strongly.

RED as a whole SHOULD NOT be switched off. Without RED, a drop tail queue biases against large packets and is vulnerable to floods of small packets.

Note well that RED's byte-mode queue drop is completely orthogonal to byte-mode queue measurement and should not be confused with it. If a RED implementation has a byte-mode but does not specify what sort of byte-mode, it is most probably byte-mode queue measurement, which is

fine. However, if in doubt, the vendor should be consulted.

A survey (Appendix A) showed that there appears to be little, if any, installed base of the byte-mode drop variant of RED. This suggests that deprecating byte-mode drop will have little, if any, incremental deployment impact.

2.3. Recommendation on Responding to Congestion

When a transport detects that a packet has been lost or congestion marked, it SHOULD consider the strength of the congestion indication as proportionate to the size in octets (bytes) of the missing or marked packet.

In other words, when a packet indicates congestion (by being lost or marked) it can be considered conceptually as if there is a congestion indication on every octet of the packet, not just one indication per packet.

To be clear, the above recommendation solely describes how a transport should interpret the meaning of a congestion indication, as a long term goal. It makes no recommendation on whether a transport should act differently based on this interpretation. It merely aids interoperability between transports, if they choose to make their actions depend on the strength of congestion indications.

This definition will be useful as the IETF transport area continues its programme of;

- o updating host-based congestion control protocols to take account of packet size
- o making transports less sensitive to losing control packets like SYNs and pure ACKs.

What this advice means for the case of TCP:

1. If two TCP flows with different packet sizes are required to run at equal bit rates under the same path conditions, this SHOULD be done by altering TCP (Section 4.2.2), not network equipment (the latter affects other transports besides TCP).
2. If it is desired to improve TCP performance by reducing the chance that a SYN or a pure ACK will be dropped, this SHOULD be done by modifying TCP (Section 4.2.3), not network equipment.

To be clear, we are not recommending at all that TCPs under equivalent conditions should aim for equal bit-rates. We are merely

saying that anyone trying to do such a thing should modify their TCP algorithm, not the network.

These recommendations are phrased as 'SHOULD' rather than 'MUST', because there may be cases where expediency dictates that compatibility with pre-existing versions of a transport protocol make the recommendations impractical.

2.4. Recommendation on Handling Congestion Indications when Splitting or Merging Packets

Packets carrying congestion indications may be split or merged in some circumstances (e.g. at a RTP/RTCP transcoder or during IP fragment reassembly). Splitting and merging only make sense in the context of ECN, not loss.

The general rule to follow is that the number of octets in packets with congestion indications SHOULD be equivalent before and after merging or splitting. This is based on the principle used above; that an indication of congestion on a packet can be considered as an indication of congestion on each octet of the packet.

The above rule is not phrased with the word "MUST" to allow the following exception. There are cases where pre-existing protocols were not designed to conserve congestion marked octets (e.g. IP fragment reassembly [RFC3168] or loss statistics in RTCP receiver reports [RFC3550] before ECN was added [RFC6679]). When any such protocol is updated, it SHOULD comply with the above rule to conserve marked octets. However, the rule may be relaxed if it would otherwise become too complex to interoperate with pre-existing implementations of the protocol.

One can think of a splitting or merging process as if all the incoming congestion-marked octets increment a counter and all the outgoing marked octets decrement the same counter. In order to ensure that congestion indications remain timely, even the smallest positive remainder in the conceptual counter should trigger the next outgoing packet to be marked (causing the counter to go negative).

3. Motivating Arguments

This section is informative. It justifies the recommendations given in the previous section.

3.1. Avoiding Perverse Incentives to (Ab)use Smaller Packets

Increasingly, it is being recognised that a protocol design must take care not to cause unintended consequences by giving the parties in

the protocol exchange perverse incentives [Evol_cc][RFC3426]. Given there are many good reasons why larger path maximum transmission units (PMTUs) would help solve a number of scaling issues, we do not want to create any bias against large packets that is greater than their true cost.

Imagine a scenario where the same bit rate of packets will contribute the same to bit-congestion of a link irrespective of whether it is sent as fewer larger packets or more smaller packets. A protocol design that caused larger packets to be more likely to be dropped than smaller ones would be dangerous in both the following cases:

Malicious transports: A queue that gives an advantage to small packets can be used to amplify the force of a flooding attack. By sending a flood of small packets, the attacker can get the queue to discard more traffic in large packets, allowing more attack traffic to get through to cause further damage. Such a queue allows attack traffic to have a disproportionately large effect on regular traffic without the attacker having to do much work.

Non-malicious transports: Even if an application designer is not actually malicious, if over time it is noticed that small packets tend to go faster, designers will act in their own interest and use smaller packets. Queues that give advantage to small packets create an evolutionary pressure for applications or transports to send at the same bit-rate but break their data stream down into tiny segments to reduce their drop rate. Encouraging a high volume of tiny packets might in turn unnecessarily overload a completely unrelated part of the system, perhaps more limited by header-processing than bandwidth.

Imagine two unresponsive flows arrive at a bit-congestible transmission link each with the same bit rate, say 1Mbps, but one consists of 1500B and the other 60B packets, which are 25x smaller. Consider a scenario where gentle RED [gentle_RED] is used, along with the variant of RED we advise against, i.e. where the RED algorithm is configured to adjust the drop probability of packets in proportion to each packet's size (byte mode packet drop). In this case, RED aims to drop 25x more of the larger packets than the smaller ones. Thus, for example if RED drops 25% of the larger packets, it will aim to drop 1% of the smaller packets (but in practice it may drop more as congestion increases [RFC4828; Appx B.4]). Even though both flows arrive with the same bit rate, the bit rate the RED queue aims to pass to the line will be 750kbps for the flow of larger packets but 990kbps for the smaller packets (because of rate variations it will actually be a little less than this target).

Note that, although the byte-mode drop variant of RED amplifies small

packet attacks, drop-tail queues amplify small packet attacks even more (see Security Considerations in Section 6). Wherever possible neither should be used.

3.2. Small != Control

Dropping fewer control packets considerably improves performance. It is tempting to drop small packets with lower probability in order to improve performance, because many control packets tend to be smaller (TCP SYNs & ACKs, DNS queries & responses, SIP messages, HTTP GETs, etc). However, we must not give control packets preference purely by virtue of their smallness, otherwise it is too easy for any data source to get the same preferential treatment simply by sending data in smaller packets. Again we should not create perverse incentives to favour small packets rather than to favour control packets, which is what we intend.

Just because many control packets are small does not mean all small packets are control packets.

So, rather than fix these problems in the network, we argue that the transport should be made more robust against losses of control packets (see 'Making Transports Robust against Control Packet Losses' in Section 4.2.3).

3.3. Transport-Independent Network

TCP congestion control ensures that flows competing for the same resource each maintain the same number of segments in flight, irrespective of segment size. So under similar conditions, flows with different segment sizes will get different bit-rates.

To counter this effect it seems tempting not to follow our recommendation, and instead for the network to bias congestion notification by packet size in order to equalise the bit-rates of flows with different packet sizes. However, in order to do this, the queuing algorithm has to make assumptions about the transport, which become embedded in the network. Specifically:

- o The queuing algorithm has to assume how aggressively the transport will respond to congestion (see Section 4.2.4). If the network assumes the transport responds as aggressively as TCP NewReno, it will be wrong for Compound TCP and differently wrong for Cubic TCP, etc. To achieve equal bit-rates, each transport then has to guess what assumption the network made, and work out how to replace this assumed aggressiveness with its own aggressiveness.

- o Also, if the network biases congestion notification by packet size it has to assume a baseline packet size--all proposed algorithms use the local MTU (for example see the byte-mode loss probability formula in Table 1). Then if the non-Reno transports mentioned above are trying to reverse engineer what the network assumed, they also have to guess the MTU of the congested link.

Even though reducing the drop probability of small packets (e.g. RED's byte-mode drop) helps ensure TCP flows with different packet sizes will achieve similar bit rates, we argue this correction should be made to any future transport protocols based on TCP, not to the network in order to fix one transport, no matter how predominant it is. Effectively, favouring small packets is reverse engineering of network equipment around one particular transport protocol (TCP), contrary to the excellent advice in [RFC3426], which asks designers to question "Why are you proposing a solution at this layer of the protocol stack, rather than at another layer?"

In contrast, if the network never takes account of packet size, the transport can be certain it will never need to guess any assumptions the network has made. And the network passes two pieces of information to the transport that are sufficient in all cases: i) congestion notification on the packet and ii) the size of the packet. Both are available for the transport to combine (by taking account of packet size when responding to congestion) or not. Appendix B checks that these two pieces of information are sufficient for all relevant scenarios.

When the network does not take account of packet size, it allows transport protocols to choose whether to take account of packet size or not. However, if the network were to bias congestion notification by packet size, transport protocols would have no choice; those that did not take account of packet size themselves would unwittingly become dependent on packet size, and those that already took account of packet size would end up taking account of it twice.

3.4. Partial Deployment of AQM

In overview, the argument in this section runs as follows:

- o Because the network does not and cannot always drop packets in proportion to their size, it shouldn't be given the task of making drop signals depend on packet size at all.
- o Transports on the other hand don't always want to make their rate response proportional to the size of dropped packets, but if they want to, they always can.

The argument is similar to the end-to-end argument that says "Don't do X in the network if end-systems can do X by themselves, and they want to be able to choose whether to do X anyway." Actually the following argument is stronger; in addition it says "Don't give the network task X that could be done by the end-systems, if X is not deployed on all network nodes, and end-systems won't be able to tell whether their network is doing X, or whether they need to do X themselves." In this case, the X in question is "making the response to congestion depend on packet size".

We will now re-run this argument taking each step in more depth. The argument applies solely to drop, not to ECN marking.

A queue drops packets for either of two reasons: a) to signal to host congestion controls that they should reduce the load and b) because there is no buffer left to store the packets. Active queue management tries to use drops as a signal for hosts to slow down (case a) so that drop due to buffer exhaustion (case b) should not be necessary.

AQM is not universally deployed in every queue in the Internet; many cheap Ethernet bridges, software firewalls, NATs on consumer devices, etc implement simple tail-drop buffers. Even if AQM were universal, it has to be able to cope with buffer exhaustion (by switching to a behaviour like tail-drop), in order to cope with unresponsive or excessive transports. For these reasons networks will sometimes be dropping packets as a last resort (case b) rather than under AQM control (case a).

When buffers are exhausted (case b), they don't naturally drop packets in proportion to their size. The network can only reduce the probability of dropping smaller packets if it has enough space to store them somewhere while it waits for a larger packet that it can drop. If the buffer is exhausted, it does not have this choice. Admittedly tail-drop does naturally drop somewhat fewer small packets, but exactly how few depends more on the mix of sizes than the size of the packet in question. Nonetheless, in general, if we wanted networks to do size-dependent drop, we would need universal deployment of (packet-size dependent) AQM code, which is currently unrealistic.

A host transport cannot know whether any particular drop was a deliberate signal from an AQM or a sign of a queue shedding packets due to buffer exhaustion. Therefore, because the network cannot universally do size-dependent drop, it should not do it all.

Whereas universality is desirable in the network, diversity is desirable between different transport layer protocols - some, like

NewReno TCP [RFC5681], may not choose to make their rate response proportionate to the size of each dropped packet, while others will (e.g. TFRC-SP [RFC4828]).

3.5. Implementation Efficiency

Biasing against large packets typically requires an extra multiply and divide in the network (see the example byte-mode drop formula in Table 1). Allowing for packet size at the transport rather than in the network ensures that neither the network nor the transport needs to do a multiply operation--multiplication by packet size is effectively achieved as a repeated add when the transport adds to its count of marked bytes as each congestion event is fed to it. Also the work to do the biasing is spread over many hosts, rather than concentrated in just the congested network element. These aren't principled reasons in themselves, but they are a happy consequence of the other principled reasons.

4. A Survey and Critique of Past Advice

This section is informative, not normative.

The original 1993 paper on RED [RED93] proposed two options for the RED active queue management algorithm: packet mode and byte mode. Packet mode measured the queue length in packets and dropped (or marked) individual packets with a probability independent of their size. Byte mode measured the queue length in bytes and marked an individual packet with probability in proportion to its size (relative to the maximum packet size). In the paper's outline of further work, it was stated that no recommendation had been made on whether the queue size should be measured in bytes or packets, but noted that the difference could be significant.

When RED was recommended for general deployment in 1998 [RFC2309], the two modes were mentioned implying the choice between them was a question of performance, referring to a 1997 email [pktByteEmail] for advice on tuning. A later addendum to this email introduced the insight that there are in fact two orthogonal choices:

- o whether to measure queue length in bytes or packets (Section 4.1)
- o whether the drop probability of an individual packet should depend on its own size (Section 4.2).

The rest of this section is structured accordingly.

4.1. Congestion Measurement Advice

The choice of which metric to use to measure queue length was left open in RFC2309. It is now well understood that queues for bit-congestible resources should be measured in bytes, and queues for packet-congestible resources should be measured in packets [pktByteEmail].

Congestion in some legacy bit-congestible buffers is only measured in packets not bytes. In such cases, the operator has to set the thresholds mindful of a typical mix of packets sizes. Any AQM algorithm on such a buffer will be oversensitive to high proportions of small packets, e.g. a DoS attack, and under-sensitive to high proportions of large packets. However, there is no need to make allowances for the possibility of such legacy in future protocol design. This is safe because any under-sensitivity during unusual traffic mixes cannot lead to congestion collapse given the buffer will eventually revert to tail drop, discarding proportionately more large packets.

4.1.1. Fixed Size Packet Buffers

The question of whether to measure queues in bytes or packets seems to be well understood. However, measuring congestion is confusing when the resource is bit congestible but the queue into the resource is packet congestible. This section outlines the approach to take.

Some, mostly older, queuing hardware allocates fixed sized buffers in which to store each packet in the queue. This hardware forwards to the line in one of two ways:

- o With some hardware, any fixed sized buffers not completely filled by a packet are padded when transmitted to the wire. This case, should clearly be treated as packet-congestible, because both queuing and transmission are in fixed MTU-sized units. Therefore the queue length in packets is a good model of congestion of the link.
- o More commonly, hardware with fixed size packet buffers transmits packets to line without padding. This implies a hybrid forwarding system with transmission congestion dependent on the size of packets but queue congestion dependent on the number of packets, irrespective of their size.

Nonetheless, there would be no queue at all unless the line had become congested--the root-cause of any congestion is too many bytes arriving for the line. Therefore, the AQM should measure the queue length as the sum of all the packet sizes in bytes that

are queued up waiting to be serviced by the line, irrespective of whether each packet is held in a fixed size buffer.

In the (unlikely) first case where use of padding means the queue should be measured in packets, further confusion is likely because the fixed buffers are rarely all one size. Typically pools of different sized buffers are provided (Cisco uses the term 'buffer carving' for the process of dividing up memory into these pools [IOSArch]). Usually, if the pool of small buffers is exhausted, arriving small packets can borrow space in the pool of large buffers, but not vice versa. However, there is no need to consider all this complexity, because the root-cause of any congestion is still line overload--buffer consumption is only the symptom. Therefore, the length of the queue should be measured as the sum of the bytes in the queue that will be transmitted to line, including any padding. In the (unusual) case of transmission with padding this means the sum of the sizes of the small buffers queued plus the sum of the sizes of the large buffers queued.

We will return to borrowing of fixed sized buffers when we discuss biasing the drop/marketing probability of a specific packet because of its size in Section 4.2.1. But here we can repeat the simple rule for how to measure the length of queues of fixed buffers: no matter how complicated the buffering scheme is, ultimately a transmission line is nearly always bit-congestible so the number of bytes queued up waiting for the line measures how congested the line is, and it is rarely important to measure how congested the buffering system is.

4.1.1.2. Congestion Measurement without a Queue

AQM algorithms are nearly always described assuming there is a queue for a congested resource and the algorithm can use the queue length to determine the probability that it will drop or mark each packet. But not all congested resources lead to queues. For instance, power limited resources are usually bit-congestible if energy is primarily required for transmission rather than header processing, but it is rare for a link protocol to build a queue as it approaches maximum power.

Nonetheless, AQM algorithms do not require a queue in order to work. For instance spectrum congestion can be modelled by signal quality using target bit-energy-to-noise-density ratio. And, to model radio power exhaustion, transmission power levels can be measured and compared to the maximum power available. [ECNFixedWireless] proposes a practical and theoretically sound way to combine congestion notification for different bit-congestible resources at different layers along an end to end path, whether wireless or wired, and whether with or without queues.

In wireless protocols that use request to send / clear to send (RTS / CTS) control, such as some variants of IEEE802.11, it is reasonable to base an AQM on the time spent waiting for transmission opportunities (TXOPs) even though wireless spectrum is usually regarded as congested by bits (for a given coding scheme). This is because requests for TXOPs queue up as the spectrum gets congested by all the bits being transferred. So the time that TXOPs are queued directly reflects bit congestion of the spectrum.

4.2. Congestion Notification Advice

4.2.1. Network Bias when Encoding

4.2.1.1. Advice on Packet Size Bias in RED

The previously mentioned email [pktByteEmail] referred to by [RFC2309] advised that most scarce resources in the Internet were bit-congestible, which is still believed to be true (Section 1.1). But it went on to offer advice that is updated by this memo. It said that drop probability should depend on the size of the packet being considered for drop if the resource is bit-congestible, but not if it is packet-congestible. The argument continued that if packet drops were inflated by packet size (byte-mode dropping), "a flow's fraction of the packet drops is then a good indication of that flow's fraction of the link bandwidth in bits per second". This was consistent with a referenced policing mechanism being worked on at the time for detecting unusually high bandwidth flows, eventually published in 1999 [pBox]. However, the problem could and should have been solved by making the policing mechanism count the volume of bytes randomly dropped, not the number of packets.

A few months before RFC2309 was published, an addendum was added to the above archived email referenced from the RFC, in which the final paragraph seemed to partially retract what had previously been said. It clarified that the question of whether the probability of dropping/markings a packet should depend on its size was not related to whether the resource itself was bit congestible, but a completely orthogonal question. However the only example given had the queue measured in packets but packet drop depended on the size of the packet in question. No example was given the other way round.

In 2000, Cnoder et al [REDbyte] pointed out that there was an error in the part of the original 1993 RED algorithm that aimed to distribute drops uniformly, because it didn't correctly take into account the adjustment for packet size. They recommended an algorithm called RED_4 to fix this. But they also recommended a further change, RED_5, to adjust drop rate dependent on the square of relative packet size. This was indeed consistent with one implied

motivation behind RED's byte mode drop--that we should reverse engineer the network to improve the performance of dominant end-to-end congestion control mechanisms. This memo makes a different recommendations in Section 2.

By 2003, a further change had been made to the adjustment for packet size, this time in the RED algorithm of the ns2 simulator. Instead of taking each packet's size relative to a 'maximum packet size' it was taken relative to a 'mean packet size', intended to be a static value representative of the 'typical' packet size on the link. We have not been able to find a justification in the literature for this change, however Eddy and Allman conducted experiments [REDbias] that assessed how sensitive RED was to this parameter, amongst other things. However, this changed algorithm can often lead to drop probabilities of greater than 1 (which gives a hint that there is probably a mistake in the theory somewhere).

On 10-Nov-2004, this variant of byte-mode packet drop was made the default in the ns2 simulator. It seems unlikely that byte-mode drop has ever been implemented in production networks (Appendix A), therefore any conclusions based on ns2 simulations that use RED without disabling byte-mode drop are likely to behave very differently from RED in production networks.

4.2.1.2. Packet Size Bias Regardless of AQM

The byte-mode drop variant of RED (or a similar variant of other AQM algorithms) is not the only possible bias towards small packets in queueing systems. We have already mentioned that tail-drop queues naturally tend to lock-out large packets once they are full.

But also queues with fixed sized buffers reduce the probability that small packets will be dropped if (and only if) they allow small packets to borrow buffers from the pools for larger packets (see Section 4.1.1). Borrowing effectively makes the maximum queue size for small packets greater than that for large packets, because more buffers can be used by small packets while less will fit large packets. Incidentally, the bias towards small packets from buffer borrowing is nothing like as large as that of RED's byte-mode drop.

Nonetheless, fixed-buffer memory with tail drop is still prone to lock-out large packets, purely because of the tail-drop aspect. So, fixed size packet-buffers should be augmented with a good AQM algorithm and packet-mode drop. If an AQM is too complicated to implement with multiple fixed buffer pools, the minimum necessary to prevent large packet lock-out is to ensure smaller packets never use the last available buffer in any of the pools for larger packets.

4.2.2. Transport Bias when Decoding

The above proposals to alter the network equipment to bias towards smaller packets have largely carried on outside the IETF process. Whereas, within the IETF, there are many different proposals to alter transport protocols to achieve the same goals, i.e. either to make the flow bit-rate take account of packet size, or to protect control packets from loss. This memo argues that altering transport protocols is the more principled approach.

A recently approved experimental RFC adapts its transport layer protocol to take account of packet sizes relative to typical TCP packet sizes. This proposes a new small-packet variant of TCP-friendly rate control [RFC5348] called TFRC-SP [RFC4828]. Essentially, it proposes a rate equation that inflates the flow rate by the ratio of a typical TCP segment size (1500B including TCP header) over the actual segment size [PktSizeEquCC]. (There are also other important differences of detail relative to TFRC, such as using virtual packets [CCvarPktSize] to avoid responding to multiple losses per round trip and using a minimum inter-packet interval.)

Section 4.5.1 of this TFRC-SP spec discusses the implications of operating in an environment where queues have been configured to drop smaller packets with proportionately lower probability than larger ones. But it only discusses TCP operating in such an environment, only mentioning TFRC-SP briefly when discussing how to define fairness with TCP. And it only discusses the byte-mode dropping version of RED as it was before Cnoddler et al pointed out it didn't sufficiently bias towards small packets to make TCP independent of packet size.

So the TFRC-SP spec doesn't address the issue of which of the network or the transport should handle fairness between different packet sizes. In its Appendix B.4 it discusses the possibility of both TFRC-SP and some network buffers duplicating each other's attempts to deliberately bias towards small packets. But the discussion is not conclusive, instead reporting simulations of many of the possibilities in order to assess performance but not recommending any particular course of action.

The paper originally proposing TFRC with virtual packets (VP-TFRC) [CCvarPktSize] proposed that there should perhaps be two variants to cater for the different variants of RED. However, as the TFRC-SP authors point out, there is no way for a transport to know whether some queues on its path have deployed RED with byte-mode packet drop (except if an exhaustive survey found that no-one has deployed it!--see Appendix A). Incidentally, VP-TFRC also proposed that byte-mode RED dropping should really square the packet-size compensation-factor

(like that of Cnoder's RED_5, but apparently unaware of it).

Pre-congestion notification [RFC5670] is an IETF technology to use a virtual queue for AQM marking for packets within one Diffserv class in order to give early warning prior to any real queuing. The PCN marking algorithms have been designed not to take account of packet size when forwarding through queues. Instead the general principle has been to take account of the sizes of marked packets when monitoring the fraction of marking at the edge of the network, as recommended here.

4.2.3. Making Transports Robust against Control Packet Losses

Recently, two RFCs have defined changes to TCP that make it more robust against losing small control packets [RFC5562] [RFC5690]. In both cases they note that the case for these two TCP changes would be weaker if RED were biased against dropping small packets. We argue here that these two proposals are a safer and more principled way to achieve TCP performance improvements than reverse engineering RED to benefit TCP.

Although there are no known proposals, it would also be possible and perfectly valid to make control packets robust against drop by requesting a scheduling class with lower drop probability, by re-marking to a Diffserv code point [RFC2474] within the same behaviour aggregate.

Although not brought to the IETF, a simple proposal from Wischik [DupTCP] suggests that the first three packets of every TCP flow should be routinely duplicated after a short delay. It shows that this would greatly improve the chances of short flows completing quickly, but it would hardly increase traffic levels on the Internet, because Internet bytes have always been concentrated in the large flows. It further shows that the performance of many typical applications depends on completion of long serial chains of short messages. It argues that, given most of the value people get from the Internet is concentrated within short flows, this simple expedient would greatly increase the value of the best efforts Internet at minimal cost. A similar but more extensive approach has been evaluated on Google servers [GentleAggro].

The proposals discussed in this sub-section are experimental approaches that are not yet in wide operational use, but they are existence proofs that transports can make themselves robust against loss of control packets. The examples are all TCP-based, but applications over non-TCP transports could mitigate loss of control packets by making similar use of Diffserv, data duplication, FEC etc.

4.2.4. Congestion Notification: Summary of Conflicting Advice

transport cc	RED_1 (packet mode drop)	RED_4 (linear byte mode drop)	RED_5 (square byte mode drop)
TCP or TFRC	s/\sqrt{p}	$\sqrt{s/p}$	$1/\sqrt{p}$
TFRC-SP	$1/\sqrt{p}$	$1/\sqrt{sp}$	$1/(s.\sqrt{p})$

Table 2: Dependence of flow bit-rate per RTT on packet size, s , and drop probability, p , when network and/or transport bias towards small packets to varying degrees

Table 2 aims to summarise the potential effects of all the advice from different sources. Each column shows a different possible AQM behaviour in different queues in the network, using the terminology of Cnoder et al outlined earlier (RED_1 is basic RED with packet-mode drop). Each row shows a different transport behaviour: TCP [RFC5681] and TFRC [RFC5348] on the top row with TFRC-SP [RFC4828] below. Each cell shows how the bits per round trip of a flow depends on packet size, s , and drop probability, p . In order to declutter the formulae to focus on packet-size dependence they are all given per round trip, which removes any RTT term.

Let us assume that the goal is for the bit-rate of a flow to be independent of packet size. Suppressing all inessential details, the table shows that this should either be achievable by not altering the TCP transport in a RED_5 network, or using the small packet TFRC-SP transport (or similar) in a network without any byte-mode dropping RED (top right and bottom left). Top left is the 'do nothing' scenario, while bottom right is the 'do-both' scenario in which bit-rate would become far too biased towards small packets. Of course, if any form of byte-mode dropping RED has been deployed on a subset of queues that congest, each path through the network will present a different hybrid scenario to its transport.

Whatever, we can see that the linear byte-mode drop column in the middle would considerably complicate the Internet. It's a half-way house that doesn't bias enough towards small packets even if one believes the network should be doing the biasing. Section 2 recommends that all bias in network equipment towards small packets should be turned off--if indeed any equipment vendors have implemented it--leaving packet-size bias solely as the preserve of the transport layer (solely the leftmost, packet-mode drop column).

In practice it seems that no deliberate bias towards small packets

has been implemented for production networks. Of the 19% of vendors who responded to a survey of 84 equipment vendors, none had implemented byte-mode drop in RED (see Appendix A for details).

5. Outstanding Issues and Next Steps

5.1. Bit-congestible Network

For a connectionless network with nearly all resources being bit-congestible the recommended position is clear--that the network should not make allowance for packet sizes and the transport should. This leaves two outstanding issues:

- o How to handle any legacy of AQM with byte-mode drop already deployed;
- o The need to start a programme to update transport congestion control protocol standards to take account of packet size.

A survey of equipment vendors (Section 4.2.4) found no evidence that byte-mode packet drop had been implemented, so deployment will be sparse at best. A migration strategy is not really needed to remove an algorithm that may not even be deployed.

A programme of experimental updates to take account of packet size in transport congestion control protocols has already started with TFRC-SP [RFC4828].

5.2. Bit- & Packet-congestible Network

The position is much less clear-cut if the Internet becomes populated by a more even mix of both packet-congestible and bit-congestible resources (see Appendix B.2). This problem is not pressing, because most Internet resources are designed to be bit-congestible before packet processing starts to congest (see Section 1.1).

The IRTF Internet congestion control research group (ICCRG) has set itself the task of reaching consensus on generic forwarding mechanisms that are necessary and sufficient to support the Internet's future congestion control requirements (the first challenge in [RFC6077]). The research question of whether packet congestion might become common and what to do if it does may in the future be explored in the IRTF (the "Challenge 3: Packet Size" in [RFC6077]).

Note that sometimes it seems that resources might be congested by neither bits nor packets, e.g. where the queue for access to a wireless medium is in units of transmission opportunities. However,

the root cause of congestion of the underlying spectrum is overload of bits (see Section 4.1.2).

6. Security Considerations

This memo recommends that queues do not bias drop probability due to packets size. For instance dropping small packets less often than large creates a perverse incentive for transports to break down their flows into tiny segments. One of the benefits of implementing AQM was meant to be to remove this perverse incentive that drop-tail queues gave to small packets.

In practice, transports cannot all be trusted to respond to congestion. So another reason for recommending that queues do not bias drop probability towards small packets is to avoid the vulnerability to small packet DDoS attacks that would otherwise result. One of the benefits of implementing AQM was meant to be to remove drop-tail's DoS vulnerability to small packets, so we shouldn't add it back again.

If most queues implemented AQM with byte-mode drop, the resulting network would amplify the potency of a small packet DDoS attack. At the first queue the stream of packets would push aside a greater proportion of large packets, so more of the small packets would survive to attack the next queue. Thus a flood of small packets would continue on towards the destination, pushing regular traffic with large packets out of the way in one queue after the next, but suffering much less drop itself.

Appendix C explains why the ability of networks to police the response of any transport to congestion depends on bit-congestible network resources only doing packet-mode not byte-mode drop. In summary, it says that making drop probability depend on the size of the packets that bits happen to be divided into simply encourages the bits to be divided into smaller packets. Byte-mode drop would therefore irreversibly complicate any attempt to fix the Internet's incentive structures.

7. IANA Considerations

This document has no actions for IANA.

8. Conclusions

This memo identifies the three distinct stages of the congestion notification process where implementations need to decide whether to take packet size into account. The recommendations provided in Section 2 of this memo are different in each case:

- o When network equipment measures the length of a queue, if it is not feasible to use time it is recommended to count in bytes if the network resource is congested by bytes, or to count in packets if is congested by packets.
- o When network equipment decides whether to drop (or mark) a packet, it is recommended that the size of the particular packet should not be taken into account
- o However, when a transport algorithm responds to a dropped or marked packet, the size of the rate reduction should be proportionate to the size of the packet.

In summary, the answers are 'it depends', 'no' and 'yes' respectively

For the specific case of RED, this means that byte-mode queue measurement will often be appropriate but the use of byte-mode drop is very strongly discouraged.

At the transport layer the IETF should continue updating congestion control protocols to take account of the size of each packet that indicates congestion. Also the IETF should continue to make protocols less sensitive to losing control packets like SYN's, pure ACKs and DNS exchanges. Although many control packets happen to be small, the alternative of network equipment favouring all small packets would be dangerous. That would create perverse incentives to split data transfers into smaller packets.

The memo develops these recommendations from principled arguments concerning scaling, layering, incentives, inherent efficiency, security and policeability. But it also addresses practical issues such as specific buffer architectures and incremental deployment. Indeed a limited survey of RED implementations is discussed, which shows there appears to be little, if any, installed base of RED's byte-mode drop. Therefore it can be deprecated with little, if any, incremental deployment complications.

The recommendations have been developed on the well-founded basis that most Internet resources are bit-congestible not packet-congestible. We need to know the likelihood that this assumption will prevail longer term and, if it might not, what protocol changes will be needed to cater for a mix of the two. The IRTF Internet Congestion Control Research Group (ICCRG) is currently working on these problems [RFC6077].

9. Acknowledgements

Thank you to Sally Floyd, who gave extensive and useful review comments. Also thanks for the reviews from Philip Eardley, David Black, Fred Baker, David Taht, Toby Moncaster, Arnaud Jacquet and Mirja Kuehlewind as well as helpful explanations of different hardware approaches from Larry Dunn and Fred Baker. We are grateful to Bruce Davie and his colleagues for providing a timely and efficient survey of RED implementation in Cisco's product range. Also grateful thanks to Toby Moncaster, Will Dormann, John Regnault, Simon Carter and Stefaan De Cnodder who further helped survey the current status of RED implementation and deployment and, finally, thanks to the anonymous individuals who responded.

Bob Briscoe and Jukka Manner were partly funded by Trilogy, a research project (ICT- 216372) supported by the European Community under its Seventh Framework Programme. The views expressed here are those of the authors only.

10. Comments Solicited

Comments and questions are encouraged and very welcome. They can be addressed to the IETF Transport Area working group mailing list <tsvwg@ietf.org>, and/or to the authors.

11. References

11.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, September 2001.

11.2. Informative References

- [BLUE02] Feng, W-c., Shin, K., Kandlur, D., and D. Saha, "The BLUE active queue management algorithms", IEEE/ACM Transactions on Networking 10(4) 513--528, August 2002, <<http://dx.doi.org/10.1109/TNET.2002.801399>>.
- [CCvarPktSize] Widmer, J., Boutremans, C., and J-Y. Le

- Boudec, "Congestion Control for Flows with Variable Packet Size", ACM CCR 34(2) 137--151, 2004, <<http://doi.acm.org/10.1145/997150.997162>>.
- [CHOKE_Var_Pkt] Psounis, K., Pan, R., and B. Prabhaker, "Approximate Fair Dropping for Variable Length Packets", IEEE Micro 21(1):48--56, January-February 2001, <<http://www.stanford.edu/~balaji/papers/01approximatefair.pdf>>.
- [DRQ] Shin, M., Chong, S., and I. Rhee, "Dual-Resource TCP/AQM for Processing-Constrained Networks", IEEE/ACM Transactions on Networking Vol 16, issue 2, April 2008, <<http://dx.doi.org/10.1109/TNET.2007.900415>>.
- [DupTCP] Wischik, D., "Short messages", Philosophical Transactions of the Royal Society A 366(1872):1941-1953, June 2008, <<http://rsta.royalsocietypublishing.org/content/366/1872/1941.full.pdf+html>>.
- [ECNFixedWireless] Siris, V., "Resource Control for Elastic Traffic in CDMA Networks", Proc. ACM MOBICOM'02 , September 2002, <http://www.ics.forth.gr/netlab/publications/resource_control_elastic_cdma.html>.
- [Evol_cc] Gibbens, R. and F. Kelly, "Resource pricing and the evolution of congestion control", Automatica 35(12):1969--1985, December 1999, <<http://www.statslab.cam.ac.uk/~frank/evol.html>>.
- [GentleAggro] Flach, T., Dukkupati, N., Terzis, A., Raghavan, B., Cardwell, N., Cheng, Y., Jain, A., Hao, S., Katz-Bassett, E., and R. Govindan, "Reducing Web Latency: the Virtue of Gentle Aggression", ACM SIGCOMM CCR 43(4):159--170, August 2013, <<http://doi.acm.org/10.1145/2486001.2486014>>.
- [I-D.nichols-tsvwg-codel] Nichols, K. and V. Jacobson, "Controlled Delay Active Queue Management",

- draft-nichols-tsvwg-codel-01 (work in progress), February 2013.
- [I-D.pan-tsvwg-pie] Pan, R., Natarajan, P., Piglione, C., and M. Prabhu, "PIE: A Lightweight Control Scheme To Address the Bufferbloat Problem", draft-pan-tsvwg-pie-00 (work in progress), December 2012.
- [IOSArch] Bollapragada, V., White, R., and C. Murphy, "Inside Cisco IOS Software Architecture", Cisco Press: CCIE Professional Development ISBN13: 978-1-57870-181-0, July 2000.
- [PktSizeEquCC] Vasallo, P., "Variable Packet Size Equation-Based Congestion Control", ICSI Technical Report tr-00-008, 2000, <<http://http.icsi.berkeley.edu/ftp/global/pub/techreports/2000/tr-00-008.pdf>>.
- [RED93] Floyd, S. and V. Jacobson, "Random Early Detection (RED) gateways for Congestion Avoidance", IEEE/ACM Transactions on Networking 1(4) 397--413, August 1993, <<http://www.icir.org/floyd/papers/red/red.html>>.
- [REDBias] Eddy, W. and M. Allman, "A Comparison of RED's Byte and Packet Modes", Computer Networks 42(3) 261--280, June 2003, <<http://www.ir.bbn.com/documents/articles/redbias.ps>>.
- [REDbyte] De Cnodder, S., Elloumi, O., and K. Pauwels, "RED behavior with different packet sizes", Proc. 5th IEEE Symposium on Computers and Communications (ISCC) 793--799, July 2000, <<http://www.icir.org/floyd/red/Elloumi99.pdf>>.
- [RFC2309] Braden, B., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, G., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J., and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet",

RFC 2309, April 1998.

- [RFC2474] Nichols, K., Blake, S., Baker, F., and D. Black, "Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers", RFC 2474, December 1998.
- [RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, September 2000.
- [RFC3426] Floyd, S., "General Architectural and Policy Considerations", RFC 3426, November 2002.
- [RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.
- [RFC3714] Floyd, S. and J. Kempf, "IAB Concerns Regarding Congestion Control for Voice Traffic in the Internet", RFC 3714, March 2004.
- [RFC4828] Floyd, S. and E. Kohler, "TCP Friendly Rate Control (TFRC): The Small-Packet (SP) Variant", RFC 4828, April 2007.
- [RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 5348, September 2008.
- [RFC5562] Kuzmanovic, A., Mondal, A., Floyd, S., and K. Ramakrishnan, "Adding Explicit Congestion Notification (ECN) Capability to TCP's SYN/ACK Packets", RFC 5562, June 2009.
- [RFC5670] Eardley, P., "Metering and Marking Behaviour of PCN-Nodes", RFC 5670, November 2009.
- [RFC5681] Allman, M., Paxson, V., and E. Blanton, "TCP Congestion Control", RFC 5681, September 2009.

- [RFC5690] Floyd, S., Arcia, A., Ros, D., and J. Iyengar, "Adding Acknowledgement Congestion Control to TCP", RFC 5690, February 2010.
- [RFC6077] Papadimitriou, D., Welzl, M., Scharf, M., and B. Briscoe, "Open Research Issues in Internet Congestion Control", RFC 6077, February 2011.
- [RFC6679] Westerlund, M., Johansson, I., Perkins, C., O'Hanlon, P., and K. Carlberg, "Explicit Congestion Notification (ECN) for RTP over UDP", RFC 6679, August 2012.
- [RFC6789] Briscoe, B., Woundy, R., and A. Cooper, "Congestion Exposure (ConEx) Concepts and Use Cases", RFC 6789, December 2012.
- [Rate_fair_Dis] Briscoe, B., "Flow Rate Fairness: Dismantling a Religion", ACM CCR 37(2)63--74, April 2007, <<http://portal.acm.org/citation.cfm?id=1232926>>.
- [gentle_RED] Floyd, S., "Recommendation on using the "gentle_" variant of RED", Web page , March 2000, <<http://www.icir.org/floyd/red/gentle.html>>.
- [pBox] Floyd, S. and K. Fall, "Promoting the Use of End-to-End Congestion Control in the Internet", IEEE/ACM Transactions on Networking 7(4) 458--472, August 1999, <<http://www.aciri.org/floyd/end2end-paper.html>>.
- [pktByteEmail] Floyd, S., "RED: Discussions of Byte and Packet Modes", email , March 1997, <<http://www-nrg.ee.lbl.gov/floyd/REDaveraging.txt>>.

Appendix A. Survey of RED Implementation Status

This Appendix is informative, not normative.

In May 2007 a survey was conducted of 84 vendors to assess how widely drop probability based on packet size has been implemented in RED Table 3. About 19% of those surveyed replied, giving a sample size

of 16. Although in most cases we do not have permission to identify the respondents, we can say that those that have responded include most of the larger equipment vendors, covering a large fraction of the market. The two who gave permission to be identified were Cisco and Alcatel-Lucent. The others range across the large network equipment vendors at L3 & L2, firewall vendors, wireless equipment vendors, as well as large software businesses with a small selection of networking products. All those who responded confirmed that they have not implemented the variant of RED with drop dependent on packet size (2 were fairly sure they had not but needed to check more thoroughly). At the time the survey was conducted, Linux did not implement RED with packet-size bias of drop, although we have not investigated a wider range of open source code.

Response	No. of vendors	%age of vendors
Not implemented	14	17%
Not implemented (probably)	2	2%
Implemented	0	0%
No response	68	81%
Total companies/orgs surveyed	84	100%

Table 3: Vendor Survey on byte-mode drop variant of RED (lower drop probability for small packets)

Where reasons have been given, the extra complexity of packet bias code has been most prevalent, though one vendor had a more principled reason for avoiding it--similar to the argument of this document.

Our survey was of vendor implementations, so we cannot be certain about operator deployment. But we believe many queues in the Internet are still tail-drop. The company of one of the co-authors (BT) has widely deployed RED, but many tail-drop queues are bound to still exist, particularly in access network equipment and on middleboxes like firewalls, where RED is not always available.

Routers using a memory architecture based on fixed size buffers with borrowing may also still be prevalent in the Internet. As explained in Section 4.2.1, these also provide a marginal (but legitimate) bias towards small packets. So even though RED byte-mode drop is not prevalent, it is likely there is still some bias towards small packets in the Internet due to tail drop and fixed buffer borrowing.

Appendix B. Sufficiency of Packet-Mode Drop

This Appendix is informative, not normative.

Here we check that packet-mode drop (or marking) in the network gives sufficiently generic information for the transport layer to use. We check against a 2x2 matrix of four scenarios that may occur now or in the future (Table 4). The horizontal and vertical dimensions have been chosen because each tests extremes of sensitivity to packet size in the transport and in the network respectively.

Note that this section does not consider byte-mode drop at all. Having deprecated byte-mode drop, the goal here is to check that packet-mode drop will be sufficient in all cases.

Network	Transport	a) Independent of packet size of congestion notifications	b) Dependent on packet size of congestion notifications
1) Predominantly bit-congestible network		Scenario a1)	Scenario b1)
2) Mix of bit-congestible and pkt-congestible network		Scenario a2)	Scenario b2)

Table 4: Four Possible Congestion Scenarios

Appendix B.1 focuses on the horizontal dimension of Table 4 checking that packet-mode drop (or marking) gives sufficient information, whether or not the transport uses it--scenarios b) and a) respectively.

Appendix B.2 focuses on the vertical dimension of Table 4, checking that packet-mode drop gives sufficient information to the transport whether resources in the network are bit-congestible or packet-congestible (these terms are defined in Section 1.1).

Notation: To be concrete, we will compare two flows with different packet sizes, s_1 and s_2 . As an example, we will take $s_1 = 60B = 480b$ and $s_2 = 1500B = 12,000b$.

A flow's bit rate, x [bps], is related to its packet rate, u [pps], by

$$x(t) = s.u(t).$$

In the bit-congestible case, path congestion will be denoted by `p_b`, and in the packet-congestible case by `p_p`. When either case is implied, the letter `p` alone will denote path congestion.

B.1. Packet-Size (In)Dependence in Transports

In all cases we consider a packet-mode drop queue that indicates congestion by dropping (or marking) packets with probability `p` irrespective of packet size. We use an example value of loss (marking) probability, `p=0.1%`.

A transport like RFC5681 TCP treats a congestion notification on any packet whatever its size as one event. However, a network with just the packet-mode drop algorithm does give more information if the transport chooses to use it. We will use Table 5 to illustrate this.

We will set aside the last column until later. The columns labelled "Flow 1" and "Flow 2" compare two flows consisting of 60B and 1500B packets respectively. The body of the table considers two separate cases, one where the flows have equal bit-rate and the other with equal packet-rates. In both cases, the two flows fill a 96Mbps link. Therefore, in the equal bit-rate case they each have half the bit-rate (48Mbps). Whereas, with equal packet-rates, flow 1 uses 25 times smaller packets so it gets 25 times less bit-rate--it only gets $1/(1+25)$ of the link capacity ($96\text{Mbps}/26 = 4\text{Mbps}$ after rounding). In contrast flow 2 gets 25 times more bit-rate (92Mbps) in the equal packet rate case because its packets are 25 times larger. The packet rate shown for each flow could easily be derived once the bit-rate was known by dividing bit-rate by packet size, as shown in the column labelled "Formula".

Parameter	Formula	Flow 1	Flow 2	Combined
-----	-----	-----	-----	-----
Packet size	$s/8$	60B	1,500B	(Mix)
Packet size	s	480b	12,000b	(Mix)
Pkt loss probability	p	0.1%	0.1%	0.1%
EQUAL BIT-RATE CASE				
Bit-rate	x	48Mbps	48Mbps	96Mbps
Packet-rate	$u = x/s$	100kpps	4kpps	104kpps
Absolute pkt-loss-rate	$p*u$	100pps	4pps	104pps
Absolute bit-loss-rate	$p*u*s$	48kbps	48kbps	96kbps
Ratio of lost/sent pkts	$p*u/u$	0.1%	0.1%	0.1%
Ratio of lost/sent bits	$p*u*s/(u*s)$	0.1%	0.1%	0.1%
EQUAL PACKET-RATE CASE				
Bit-rate	x	4Mbps	92Mbps	96Mbps
Packet-rate	$u = x/s$	8kpps	8kpps	15kpps
Absolute pkt-loss-rate	$p*u$	8pps	8pps	15pps
Absolute bit-loss-rate	$p*u*s$	4kbps	92kbps	96kbps
Ratio of lost/sent pkts	$p*u/u$	0.1%	0.1%	0.1%
Ratio of lost/sent bits	$p*u*s/(u*s)$	0.1%	0.1%	0.1%

Table 5: Absolute Loss Rates and Loss Ratios for Flows of Small and Large Packets and Both Combined

So far we have merely set up the scenarios. We now consider congestion notification in the scenario. Two TCP flows with the same round trip time aim to equalise their packet-loss-rates over time. That is the number of packets lost in a second, which is the packets per second (u) multiplied by the probability that each one is dropped (p). Thus TCP converges on the "Equal packet-rate" case, where both flows aim for the same "Absolute packet-loss-rate" (both 8pps in the table).

Packet-mode drop actually gives flows sufficient information to measure their loss-rate in bits per second, if they choose, not just packets per second. Each flow can count the size of a lost or marked packet and scale its rate-response in proportion (as TFRC-SP does). The result is shown in the row entitled "Absolute bit-loss-rate", where the bits lost in a second is the packets per second (u) multiplied by the probability of losing a packet (p) multiplied by the packet size (s). Such an algorithm would try to remove any imbalance in bit-loss-rate such as the wide disparity in the "Equal packet-rate" case (4kbps vs. 92kbps). Instead, a packet-size-dependent algorithm would aim for equal bit-loss-rates, which would drive both flows towards the "Equal bit-rate" case, by driving them to equal bit-loss-rates (both 48kbps in this example).

The explanation so far has assumed that each flow consists of packets of only one constant size. Nonetheless, it extends naturally to flows with mixed packet sizes. In the right-most column of Table 5 a flow of mixed size packets is created simply by considering flow 1 and flow 2 as a single aggregated flow. There is no need for a flow to maintain an average packet size. It is only necessary for the transport to scale its response to each congestion indication by the size of each individual lost (or marked) packet. Taking for example the "Equal packet-rate" case, in one second about 8 small packets and 8 large packets are lost (making closer to 15 than 16 losses per second due to rounding). If the transport multiplies each loss by its size, in one second it responds to $8 \times 480\text{b}$ and $8 \times 12,000\text{b}$ lost bits, adding up to 96,000 lost bits in a second. This double checks correctly, being the same as 0.1% of the total bit-rate of 96Mbps. For completeness, the formula for absolute bit-loss-rate is $p(u_1 \cdot s_1 + u_2 \cdot s_2)$.

Incidentally, a transport will always measure the loss probability the same irrespective of whether it measures in packets or in bytes. In other words, the ratio of lost to sent packets will be the same as the ratio of lost to sent bytes. (This is why TCP's bit rate is still proportional to packet size even when byte-counting is used, as recommended for TCP in [RFC5681], mainly for orthogonal security reasons.) This is intuitively obvious by comparing two example flows; one with 60B packets, the other with 1500B packets. If both flows pass through a queue with drop probability 0.1%, each flow will lose 1 in 1,000 packets. In the stream of 60B packets the ratio of bytes lost to sent will be 60B in every 60,000B; and in the stream of 1500B packets, the loss ratio will be 1,500B out of 1,500,000B. When the transport responds to the ratio of lost to sent packets, it will measure the same ratio whether it measures in packets or bytes: 0.1% in both cases. The fact that this ratio is the same whether measured in packets or bytes can be seen in Table 5, where the ratio of lost to sent packets and the ratio of lost to sent bytes is always 0.1% in all cases (recall that the scenario was set up with $p=0.1\%$).

This discussion of how the ratio can be measured in packets or bytes is only raised here to highlight that it is irrelevant to this memo! Whether a transport depends on packet size or not depends on how this ratio is used within the congestion control algorithm.

So far we have shown that packet-mode drop passes sufficient information to the transport layer so that the transport can take account of bit-congestion, by using the sizes of the packets that indicate congestion. We have also shown that the transport can choose not to take packet size into account if it wishes. We will now consider whether the transport can know which to do.

B.2. Bit-Congestible and Packet-Congestible Indications

As a thought-experiment, imagine an idealised congestion notification protocol that supports both bit-congestible and packet-congestible resources. It would require at least two ECN flags, one for each of bit-congestible and packet-congestible resources.

1. A packet-congestible resource trying to code congestion level p_p into a packet stream should mark the idealised 'packet congestion' field in each packet with probability p_p irrespective of the packet's size. The transport should then take a packet with the packet congestion field marked to mean just one mark, irrespective of the packet size.
2. A bit-congestible resource trying to code time-varying byte-congestion level p_b into a packet stream should mark the 'byte congestion' field in each packet with probability p_b , again irrespective of the packet's size. Unlike before, the transport should take a packet with the byte congestion field marked to count as a mark on each byte in the packet.

This hides a fundamental problem--much more fundamental than whether we can magically create header space for yet another ECN flag, or whether it would work while being deployed incrementally. Distinguishing drop from delivery naturally provides just one implicit bit of congestion indication information--the packet is either dropped or not. It is hard to drop a packet in two ways that are distinguishable remotely. This is a similar problem to that of distinguishing wireless transmission losses from congestive losses.

This problem would not be solved even if ECN were universally deployed. A congestion notification protocol must survive a transition from low levels of congestion to high. Marking two states is feasible with explicit marking, but much harder if packets are dropped. Also, it will not always be cost-effective to implement AQM at every low level resource, so drop will often have to suffice.

We are not saying two ECN fields will be needed (and we are not saying that somehow a resource should be able to drop a packet in one of two different ways so that the transport can distinguish which sort of drop it was!). These two congestion notification channels are a conceptual device to illustrate a dilemma we could face in the future. Section 3 gives four good reasons why it would be a bad idea to allow for packet size by biasing drop probability in favour of small packets within the network. The impracticality of our thought experiment shows that it will be hard to give transports a practical way to know whether to take account of the size of congestion indication packets or not.

Fortunately, this dilemma is not pressing because by design most equipment becomes bit-congested before its packet-processing becomes congested (as already outlined in Section 1.1). Therefore transports can be designed on the relatively sound assumption that a congestion indication will usually imply bit-congestion.

Nonetheless, although the above idealised protocol isn't intended for implementation, we do want to emphasise that research is needed to predict whether there are good reasons to believe that packet congestion might become more common, and if so, to find a way to somehow distinguish between bit and packet congestion [RFC3714].

Recently, the dual resource queue (DRQ) proposal [DRQ] has been made on the premise that, as network processors become more cost effective, per packet operations will become more complex (irrespective of whether more function in the network is desirable). Consequently the premise is that CPU congestion will become more common. DRQ is a proposed modification to the RED algorithm that folds both bit congestion and packet congestion into one signal (either loss or ECN).

Finally, we note one further complication. Strictly, packet-congestible resources are often cycle-congestible. For instance, for routing look-ups load depends on the complexity of each look-up and whether the pattern of arrivals is amenable to caching or not. This also reminds us that any solution must not require a forwarding engine to use excessive processor cycles in order to decide how to say it has no spare processor cycles.

Appendix C. Byte-mode Drop Complicates Policing Congestion Response

This section is informative, not normative.

There are two main classes of approach to policing congestion response: i) policing at each bottleneck link or ii) policing at the edges of networks. Packet-mode drop in RED is compatible with either, while byte-mode drop precludes edge policing.

The simplicity of an edge policer relies on one dropped or marked packet being equivalent to another of the same size without having to know which link the drop or mark occurred at. However, the byte-mode drop algorithm has to depend on the local MTU of the line--it needs to use some concept of a 'normal' packet size. Therefore, one dropped or marked packet from a byte-mode drop algorithm is not necessarily equivalent to another from a different link. A policing function local to the link can know the local MTU where the congestion occurred. However, a policer at the edge of the network cannot, at least not without a lot of complexity.

The early research proposals for type (i) policing at a bottleneck link [pBox] used byte-mode drop, then detected flows that contributed disproportionately to the number of packets dropped. However, with no extra complexity, later proposals used packet mode drop and looked for flows that contributed a disproportionate amount of dropped bytes [CHOKe_Var_Pkt].

Work is progressing on the congestion exposure protocol (ConEx [RFC6789]), which enables a type (ii) edge policer located at a user's attachment point. The idea is to be able to take an integrated view of the effect of all a user's traffic on any link in the internetwork. However, byte-mode drop would effectively preclude such edge policing because of the MTU issue above.

Indeed, making drop probability depend on the size of the packets that bits happen to be divided into would simply encourage the bits to be divided into smaller packets in order to confuse policing. In contrast, as long as a dropped/marked packet is taken to mean that all the bytes in the packet are dropped/marked, a policer can remain robust against bits being re-divided into different size packets or across different size flows [Rate_fair_Dis].

Appendix D. Changes from Previous Versions

To be removed by the RFC Editor on publication.

Full incremental diffs between each version are available at
<<http://tools.ietf.org/wg/tsvwg/draft-ietf-tsvwg-byte-pkt-congest/>>
(courtesy of the rfcdiff tool):

From -11 to -12: Following the second pass through the IESG:

- * Section 2.1 [Barry Leiba]:
 - + s/No other choice makes sense,/Subject to the exceptions below, no other choice makes sense,/
 - + s/Exceptions to these recommendations MAY be necessary /Exceptions to these recommendations may be necessary /
- * Sections 3.2 and 4.2.3 [Joel Jaeggli]:
 - + Added comment to section 4.2.3 that the examples given are not in widespread production use, but they give evidence that it is possible to follow the advice given.
 - + Section 4.2.3:

- OLD: Although there are no known proposals, it would also be possible and perfectly valid to make control packets robust against drop by explicitly requesting a lower drop probability using their Diffserv code point [RFC2474] to request a scheduling class with lower drop.
NEW: Although there are no known proposals, it would also be possible and perfectly valid to make control packets robust against drop by requesting a scheduling class with lower drop probability, by re-marking to a Diffserv code point [RFC2474] within the same behaviour aggregate.
- appended "Similarly applications, over non-TCP transports could make any packets that are effectively control packets more robust by using Diffserv, data duplication, FEC etc."
- + Updated Wischik ref and added "Reducing Web Latency: the Virtue of Gentle Aggression" ref.
- * Expanded more abbreviations (CoDel, PIE, MTU).
- * Section 1. Intro [Stephen Farrell]:
 - + In the places where the doc describes the dichotomy between 'long-term goal' and 'expediency' the words long term goal and expedient have been introduced, to more explicitly refer back to this introductory para (S.2.1 & S.2.3).
 - + Added explanation of what scaling with packet size means.
- * Conclusions [Benoit Claise]:
 - + OLD: For the specific case of RED, this means that byte-mode queue measurement will often be appropriate although byte-mode drop is strongly deprecated.
NEW: For the specific case of RED, this means that byte-mode queue measurement will often be appropriate but the use of byte-mode drop is very strongly discouraged.

From -10 to -11: Following a further WGLC:

- * Abstract: clarified that advice applies to all AQMs including newer ones
- * Abstract & Intro: changed 'read' to 'detect', because you don't read losses, you detect them.

- * S.1. Introduction: Disambiguated summary of advice on queue measurement.
- * Clarified that the doc deprecates any preference based solely on packet size, it's not only against preferring smaller packets.
- * S.4.1.2. Congestion Measurement without a Queue: Explained that a queue of TXOPs represents a queue into spectrum congested by too many bits.
- * S.5.2: Bit- & Packet-congestible Network: Referred to explanation in S.4.1.2 to make the point that TXOPs are not a primary unit of workload like bits and packets are, even though you get queues of TXOPs.
- * 6. Security: Disambiguated 'bias towards'.
- * 8. Conclusions: Made consistent with recommendation to use time if possible for queue measurement.

From -09 to -10: Following IESG review:

- * Updates 2309: Left header unchanged reflecting eventual IESG consensus [Sean Turner, Pete Resnick].
- * S.1 Intro: This memo adds to the congestion control principles enumerated in BCP 41 [Pete Resnick]
- * Abstract, S.1, S.1.1, s.1.2 Intro, Scoping and Example: Made applicability to all AQMs clearer listing some more example AQMs and explained that we always use RED for examples, but this doesn't mean it's not applicable to other AQMs. [A number of reviewers have described the draft as "about RED"]
- * S.1 & S.2.1 Queue measurement: Explained that the choice between measuring the queue in packets or bytes is only relevant if measuring it in time units is infeasible [So as not to imply that we haven't noticed the advances made by PDPC & CoDel]
- * S.1.1. Terminology: Better explained why hybrid systems congested by both packets and bytes are often designed to be treated as bit-congestible [Richard Barnes].
- * S.2.1. Queue measurement advice: Added examples. Added a counter-example to justify SHOULDs rather than MUSTs. Pointed to S.4.1 for a list of more complicated scenarios. [Benson]

Schliesser, OpsDir]

- * S2.2. Recommendation on Encoding Congestion Notification: Removed SHOULD treat packets equally, leaving only SHOULD NOT drop dependent on packet size, to avoid it sounding like we're saying QoS is not allowed. Pointed to possible app-specific legacy use of byte-mode as a counter-example that prevents us saying MUST NOT. [Pete Resnick]
- * S.2.3. Recommendation on Responding to Congestion: capitalised the two SHOULDs in recommendations for TCP, and gave possible counter-examples. [noticed while dealing with Pete Resnick's point]
- * S2.4. Splitting & Merging: RTCP -> RTP/RTCP [Pete McCann, Gen-ART]
- * S.3.2 Small != Control: many control packets are small -> ...tend to be small [Stephen Farrell]
- * S.3.1 Perverse incentives: Changed transport designers to app developers [Stephen Farrell]
- * S.4.1.1. Fixed Size Packet Buffers: Nearly completely re-written to simplify and to reverse the advice when the underlying resource is bit-congestible, irrespective of whether the buffer consists of fixed-size packet buffers. [Richard Barnes & Benson Schliesser]
- * S.4.2.1.2. Packet Size Bias Regardless of AQM: Largely re-written to reflect the earlier change in advice about fixed-size packet buffers, and to primarily focus on getting rid of tail-drop, not various nuances of tail-drop. [Richard Barnes & Benson Schliesser]
- * Editorial corrections [Tim Bray, AppsDir, Pete McCann, Gen-ART and others]
- * Updated refs (two I-Ds have become RFCs). [Pete McCann]

From -08 to -09: Following WG last call:

- * S.2.1: Made RED-related queue measurement recommendations clearer
- * S.2.3: Added to "Recommendation on Responding to Congestion" to make it clear that we are definitely not saying transports have to equalise bit-rates, just how to do it and not do it, if you

want to.

- * S.3: Clarified motivation sections S.3.3 "Transport-Independent Network" and S.3.5 "Implementation Efficiency"
- * S.3.4: Completely changed motivating argument from "Scaling Congestion Control with Packet Size" to "Partial Deployment of AQM".

From -07 to -08:

- * Altered abstract to say it provides best current practice and highlight that it updates RFC2309
- * Added null IANA section
- * Updated refs

From -06 to -07:

- * A mix-up with the corollaries and their naming in 2.1 to 2.3 fixed.

From -05 to -06:

- * Primarily editorial fixes.

From -04 to -05:

- * Changed from Informational to BCP and highlighted non-normative sections and appendices
- * Removed language about consensus
- * Added "Example Comparing Packet-Mode Drop and Byte-Mode Drop"
- * Arranged "Motivating Arguments" into a more logical order and completely rewrote "Transport-Independent Network" & "Scaling Congestion Control with Packet Size" arguments. Removed "Why Now?"
- * Clarified applicability of certain recommendations
- * Shifted vendor survey to an Appendix
- * Cut down "Outstanding Issues and Next Steps"

- * Re-drafted the start of the conclusions to highlight the three distinct areas of concern
- * Completely re-wrote appendices
- * Editorial corrections throughout.

From -03 to -04:

- * Reordered Sections 2 and 3, and some clarifications here and there based on feedback from Colin Perkins and Mirja Kuehlewind.

From -02 to -03 (this version)

- * Structural changes:
 - + Split off text at end of "Scaling Congestion Control with Packet Size" into new section "Transport-Independent Network"
 - + Shifted "Recommendations" straight after "Motivating Arguments" and added "Conclusions" at end to reinforce Recommendations
 - + Added more internal structure to Recommendations, so that recommendations specific to RED or to TCP are just corollaries of a more general recommendation, rather than being listed as a separate recommendation.
 - + Renamed "State of the Art" as "Critical Survey of Existing Advice" and retitled a number of subsections with more descriptive titles.
 - + Split end of "Congestion Coding: Summary of Status" into a new subsection called "RED Implementation Status".
 - + Removed text that had been in the Appendix "Congestion Notification Definition: Further Justification".
- * Reordered the intro text a little.
- * Made it clearer when advice being reported is deprecated and when it is not.
- * Described AQM as in network equipment, rather than saying "at the network layer" (to side-step controversy over whether functions like AQM are in the transport layer but in network

equipment).

- * Minor improvements to clarity throughout

From -01 to -02:

- * Restructured the whole document for (hopefully) easier reading and clarity. The concrete recommendation, in RFC2119 language, is now in Section 8.

From -00 to -01:

- * Minor clarifications throughout and updated references

From briscoe-byte-pkt-mark-02 to ietf-byte-pkt-congest-00:

- * Added note on relationship to existing RFCs
- * Posed the question of whether packet-congestion could become common and deferred it to the IRTF ICCRG. Added ref to the dual-resource queue (DRQ) proposal.
- * Changed PCN references from the PCN charter & architecture to the PCN marking behaviour draft most likely to imminently become the standards track WG item.

From -01 to -02:

- * Abstract reorganised to align with clearer separation of issue in the memo.
- * Introduction reorganised with motivating arguments removed to new Section 3.
- * Clarified avoiding lock-out of large packets is not the main or only motivation for RED.
- * Mentioned choice of drop or marking explicitly throughout, rather than trying to coin a word to mean either.
- * Generalised the discussion throughout to any packet forwarding function on any network equipment, not just routers.
- * Clarified the last point about why this is a good time to sort out this issue: because it will be hard / impossible to design new transports unless we decide whether the network or the transport is allowing for packet size.

- * Added statement explaining the horizon of the memo is long term, but with short term expediency in mind.
- * Added material on scaling congestion control with packet size (Section 3.4).
- * Separated out issue of normalising TCP's bit rate from issue of preference to control packets (Section 3.2).
- * Divided up Congestion Measurement section for clarity, including new material on fixed size packet buffers and buffer carving (Section 4.1.1 & Section 4.2.1) and on congestion measurement in wireless link technologies without queues (Section 4.1.2).
- * Added section on 'Making Transports Robust against Control Packet Losses' (Section 4.2.3) with existing & new material included.
- * Added tabulated results of vendor survey on byte-mode drop variant of RED (Table 3).

From -00 to -01:

- * Clarified applicability to drop as well as ECN.
- * Highlighted DoS vulnerability.
- * Emphasised that drop-tail suffers from similar problems to byte-mode drop, so only byte-mode drop should be turned off, not RED itself.
- * Clarified the original apparent motivations for recommending byte-mode drop included protecting SYN's and pure ACK's more than equalising the bit rates of TCP's with different segment sizes. Removed some conjectured motivations.
- * Added support for updates to TCP in progress (ackcc & ecn-syn-ack).
- * Updated survey results with newly arrived data.
- * Pulled all recommendations together into the conclusions.
- * Moved some detailed points into two additional appendices and a note.

* Considerable clarifications throughout.

* Updated references

Authors' Addresses

Bob Briscoe
BT
B54/77, Adastral Park
Martlesham Heath
Ipswich IP5 3RE
UK

Phone: +44 1473 645196
EMail: bob.briscoe@bt.com
URI: <http://bobbriscoe.net/>

Jukka Manner
Aalto University
Department of Communications and Networking (Comnet)
P.O. Box 13000
FIN-00076 Aalto
Finland

Phone: +358 9 470 22481
EMail: jukka.manner@aalto.fi
URI: <http://www.netlab.tkk.fi/~jmanner/>

Network Working Group
Internet-Draft
Intended status: Informational
Expires: April 13, 2012

R. Stewart
Adara Networks
M. Tuexen
Muenster Univ. of Appl. Sciences
K. Poon
Oracle Corporation
P. Lei
Cisco Systems, Inc.
V. Yasevich
HP
October 11, 2011

Sockets API Extensions for Stream Control Transmission Protocol (SCTP)
draft-ietf-tsvwg-sctpsocket-32.txt

Abstract

This document describes a mapping of the Stream Control Transmission Protocol (SCTP) into a sockets API. The benefits of this mapping include compatibility for TCP applications, access to new SCTP features and a consolidated error and event notification scheme.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 13, 2012.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of

publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Introduction	7
2. Data Types	8
3. One-to-Many Style Interface	8
3.1. Basic Operation	8
3.1.1. socket()	10
3.1.2. bind()	10
3.1.3. listen()	12
3.1.4. sendmsg() and recvmsg()	12
3.1.5. close()	14
3.1.6. connect()	15
3.2. Non-blocking mode	16
3.3. Special considerations	17
4. One-to-One Style Interface	18
4.1. Basic Operation	18
4.1.1. socket()	19
4.1.2. bind()	20
4.1.3. listen()	21
4.1.4. accept()	21
4.1.5. connect()	22
4.1.6. close()	23
4.1.7. shutdown()	23
4.1.8. sendmsg() and recvmsg()	24
4.1.9. getpeername()	25
5. Data Structures	25
5.1. The msghdr and cmsghdr Structures	25
5.2. Ancillary Data Considerations and Semantics	26
5.2.1. Multiple Items and Ordering	26
5.2.2. Accessing and Manipulating Ancillary Data	27
5.2.3. Control Message Buffer Sizing	27
5.3. SCTP msg_control Structures	28
5.3.1. SCTP Initiation Structure (SCTP_INIT)	29
5.3.2. SCTP Header Information Structure (SCTP_SNDRCV) - DEPRECATED	30
5.3.3. Extended SCTP Header Information Structure (SCTP_EXTRCV) - DEPRECATED	33
5.3.4. SCTP Send Information Structure (SCTP_SNDINFO)	34
5.3.5. SCTP Receive Information Structure (SCTP_RCVINFO) . .	36
5.3.6. SCTP Next Receive Information Structure (SCTP_NXTINFO)	37
5.3.7. SCTP PR-SCTP Information Structure (SCTP_PRINFO) . .	39
5.3.8. SCTP AUTH Information Structure (SCTP_AUTHINFO) . . .	39
5.3.9. SCTP Destination IPv4 Address Structure (SCTP_DSTADDRV4)	40
5.3.10. SCTP Destination IPv6 Address Structure (SCTP_DSTADDRV6)	40
6. SCTP Events and Notifications	40

6.1.	SCTP Notification Structure	41
6.1.1.	SCTP_ASSOC_CHANGE	43
6.1.2.	SCTP_PEER_ADDR_CHANGE	44
6.1.3.	SCTP_REMOTE_ERROR	46
6.1.4.	SCTP_SEND_FAILED - DEPRECATED	46
6.1.5.	SCTP_SHUTDOWN_EVENT	48
6.1.6.	SCTP_ADAPTATION_INDICATION	48
6.1.7.	SCTP_PARTIAL_DELIVERY_EVENT	49
6.1.8.	SCTP_AUTHENTICATION_EVENT	50
6.1.9.	SCTP_SENDER_DRY_EVENT	51
6.1.10.	SCTP_NOTIFICATIONS_STOPPED_EVENT	51
6.1.11.	SCTP_SEND_FAILED_EVENT	52
6.2.	Notification Interest Options	53
6.2.1.	SCTP_EVENTS option - DEPRECATED	53
6.2.2.	SCTP_EVENT option	55
7.	Common Operations for Both Styles	56
7.1.	send(), recv(), sendto(), and recvfrom()	56
7.2.	setsockopt() and getsockopt()	58
7.3.	read() and write()	60
7.4.	getsockname()	60
7.5.	Implicit Association Setup	60
8.	Socket Options	61
8.1.	Read / Write Options	63
8.1.1.	Retransmission Timeout Parameters (SCTP_RTOINFO)	63
8.1.2.	Association Parameters (SCTP_ASSOCINFO)	64
8.1.3.	Initialization Parameters (SCTP_INITMSG)	65
8.1.4.	SO_LINGER	66
8.1.5.	SCTP_NODELAY	66
8.1.6.	SO_RCVBUF	67
8.1.7.	SO_SNDBUF	67
8.1.8.	Automatic Close of Associations (SCTP_AUTOCLOSE)	67
8.1.9.	Set Primary Address (SCTP_PRIMARY_ADDR)	68
8.1.10.	Set Adaptation Layer Indicator (SCTP_ADAPTATION_LAYER)	68
8.1.11.	Enable/Disable Message Fragmentation (SCTP_DISABLE_FRAGMENTS)	68
8.1.12.	Peer Address Parameters (SCTP_PEER_ADDR_PARAMS)	69
8.1.13.	Set Default Send Parameters (SCTP_DEFAULT_SEND_PARAM) - DEPRECATED	71
8.1.14.	Set Notification and Ancillary Events (SCTP_EVENTS) - DEPRECATED	72
8.1.15.	Set/Clear IPv4 Mapped Addresses (SCTP_I_WANT_MAPPED_V4_ADDR)	72
8.1.16.	Get or Set the Maximum Fragmentation Size (SCTP_MAXSEG)	72
8.1.17.	Get or Set the List of Supported HMAC Identifiers (SCTP_HMAC_IDENT)	73
8.1.18.	Get or Set the Active Shared Key	

(SCTP_AUTH_ACTIVE_KEY)	73
8.1.19. Get or Set Delayed SACK Timer (SCTP_DELAYED_SACK) . .	74
8.1.20. Get or Set Fragmented Interleave (SCTP_FRAGMENT_INTERLEAVE)	75
8.1.21. Set or Get the SCTP Partial Delivery Point (SCTP_PARTIAL_DELIVERY_POINT)	76
8.1.22. Set or Get the Use of Extended Receive Info (SCTP_USE_EXT_RCVINFO) - DEPRECATED	77
8.1.23. Set or Get the Auto ASCONF Flag (SCTP_AUTO_ASCONF) .	77
8.1.24. Set or Get the Maximum Burst (SCTP_MAX_BURST)	77
8.1.25. Set or Get the Default Context (SCTP_CONTEXT)	78
8.1.26. Enable or Disable Explicit EOR Marking (SCTP_EXPLICIT_EOR)	78
8.1.27. Enable SCTP Port Reusage (SCTP_REUSE_PORT)	79
8.1.28. Set Notification Event (SCTP_EVENT)	79
8.1.29. Enable or Disable the Delivery of SCTP_RCVINFO as Ancillary Data (SCTP_RECVRCVINFO)	79
8.1.30. Enable or Disable the Delivery of SCTP_NXTINFO as Ancillary Data (SCTP_RECVNXTINFO)	79
8.1.31. Set Default Send Parameters (SCTP_DEFAULT_SNDINFO) .	80
8.1.32. Set Default PR-SCTP Parameters (SCTP_DEFAULT_PRINFO)	80
8.2. Read-Only Options	80
8.2.1. Association Status (SCTP_STATUS)	80
8.2.2. Peer Address Information (SCTP_GET_PEER_ADDR_INFO) .	82
8.2.3. Get the List of Chunks the Peer Requires to be Authenticated (SCTP_PEER_AUTH_CHUNKS)	83
8.2.4. Get the List of Chunks the Local Endpoint Requires to be Authenticated (SCTP_LOCAL_AUTH_CHUNKS)	84
8.2.5. Get the Current Number of Associations (SCTP_GET_ASSOC_NUMBER)	84
8.2.6. Get the Current Identifiers of Associations (SCTP_GET_ASSOC_ID_LIST)	85
8.3. Write-Only Options	85
8.3.1. Set Peer Primary Address (SCTP_SET_PEER_PRIMARY_ADDR)	85
8.3.2. Add a Chunk that must be Authenticated (SCTP_AUTH_CHUNK)	86
8.3.3. Set a Shared Key (SCTP_AUTH_KEY)	86
8.3.4. Deactivate a Shared Key (SCTP_AUTH_DEACTIVATE_KEY) .	87
8.3.5. Delete a Shared Key (SCTP_AUTH_DELETE_KEY)	87
9. New Functions	88
9.1. sctp_bindx()	88
9.2. sctp_peeloff()	90
9.3. sctp_getpaddrs()	90
9.4. sctp_freepaddrs()	91
9.5. sctp_getladdrs()	91
9.6. sctp_freeladdrs()	92

9.7. sctp_sendmsg() - DEPRECATED	92
9.8. sctp_rcvmsg() - DEPRECATED	93
9.9. sctp_connectx()	94
9.10. sctp_send() - DEPRECATED	95
9.11. sctp_sendx() - DEPRECATED	96
9.12. sctp_sendv()	97
9.13. sctp_rcvv()	100
10. IANA Considerations	102
11. Security Considerations	102
12. Acknowledgments	103
13. References	103
13.1. Normative References	103
13.2. Informative References	104
Appendix A. One-to-One Style Code Example	104
Appendix B. One-to-Many Style Code Example	107
Authors' Addresses	112

1. Introduction

The sockets API has provided a standard mapping of the Internet Protocol suite to many operating systems. Both TCP [RFC0793] and UDP [RFC0768] have benefited from this standard representation and access method across many diverse platforms. SCTP is a new protocol that provides many of the characteristics of TCP but also incorporates semantics more akin to UDP. This document defines a method to map the existing sockets API for use with SCTP, providing both a base for access to new features and compatibility so that most existing TCP applications can be migrated to SCTP with few (if any) changes.

There are three basic design objectives:

1. Maintain consistency with existing sockets APIs: We define a sockets mapping for SCTP that is consistent with other sockets API protocol mappings (for instance UDP, TCP, IPv4, and IPv6).
2. Support a one-to-many style interface: This set of semantics is similar to that defined for connection-less protocols, such as UDP. A one-to-many style SCTP socket should be able to control multiple SCTP associations. This is similar to a UDP socket, which can communicate with many peer endpoints. Each of these associations is assigned an association identifier so that an application can use the ID to differentiate them. Note that SCTP is connection-oriented in nature, and it does not support broadcast or multicast communications, as UDP does.
3. Support a one-to-one style interface: This interface supports a similar semantics as sockets for connection-oriented protocols, such as TCP. A one-to-one style SCTP socket should only control one SCTP association. One purpose of defining this interface is to allow existing applications built on other connection-oriented protocols to be ported to use SCTP with very little effort. Developers familiar with these semantics can easily adapt to SCTP. Another purpose is to make sure that existing mechanisms in most operating systems that support sockets, such as `select()`, should continue to work with this style of socket. Extensions are added to this mapping to provide mechanisms to exploit new features of SCTP.

Goals 2 and 3 are not compatible, so this document defines two modes of mapping, namely the one-to-many style mapping and the one-to-one style mapping. These two modes share some common data structures and operations, but will require the use of two different application programming styles. Note that all new SCTP features can be used with both styles of socket. The decision on which one to use depends mainly on the nature of applications.

A mechanism is defined to extract a one-to-many style SCTP association into a one-to-one style socket.

Some of the SCTP mechanisms cannot be adequately mapped to an existing socket interface. In some cases, it is more desirable to have a new interface instead of using existing socket calls. Section 9 of this document describes these new interfaces.

Please note that some elements of the SCTP socket API are declared as deprecated. During the evolution of this document, elements of the API were introduced, implemented and later on replaced by other elements. These replaced elements are declared as deprecated since they are still available in some implementations and the replacement functions are not. This applies especially to older versions of operating systems supporting SCTP. New SCTP socket implementations must implement at least the non deprecated elements. Implementations intending interoperability with older versions of the API should also include the deprecated functions.

2. Data Types

Whenever possible, POSIX data types defined in [IEEE-1003.1-2008] are used: `uintN_t` means an unsigned integer of exactly N bits (e.g. `uint16_t`). This document also assumes the argument data types from POSIX when possible (e.g. the final argument to `setsockopt()` is a `socklen_t` value). Whenever buffer sizes are specified, the POSIX `size_t` data type is used.

3. One-to-Many Style Interface

In the one-to-many style interface there is a 1 to many relationship between sockets and associations.

3.1. Basic Operation

A typical server in this style uses the following socket calls in sequence to prepare an endpoint for servicing requests:

- o `socket()`
- o `bind()`
- o `listen()`
- o `recvmsg()`

- o `sendmsg()`
- o `close()`

A typical client uses the following calls in sequence to setup an association with a server to request services:

- o `socket()`
- o `sendmsg()`
- o `recvmsg()`
- o `close()`

In this style, by default, all the associations connected to the endpoint are represented with a single socket. Each association is assigned an association identifier (type is `sctp_assoc_t`) so that an application can use it to differentiate among them. In some implementations, the peer endpoints' addresses can also be used for this purpose. But this is not required for performance reasons. If an implementation does not support using addresses to differentiate between different associations, the `sendto()` call can only be used to setup an association implicitly. It cannot be used to send data to an established association as the association identifier cannot be specified.

Once an association identifier is assigned to an SCTP association, that identifier will not be reused until the application explicitly terminates the use of the association. The resources belonging to that association will not be freed until that happens. This is similar to the `close()` operation on a normal socket. The only exception is when the `SCTP_AUTOCLOSE` option (Section 8.1.8) is set. In this case, after the association is terminated gracefully and automatically, the association identifier assigned to it can be reused. All applications using this option should be aware of this to avoid the possible problem of sending data to an incorrect peer endpoint.

If the server or client wishes to branch an existing association off to a separate socket, it is required to call `sctp_peeloff()` and to specify the association identifier. The `sctp_peeloff()` call will return a new one-to-one style socket which can then be used with `recv()` and `send()` functions for message passing. See Section 9.2 for more on branched-off associations.

Once an association is branched off to a separate socket, it becomes completely separated from the original socket. All subsequent

control and data operations to that association must be done through the new socket. For example, the close operation on the original socket will not terminate any associations that have been branched off to a different socket.

One-to-many style socket calls are discussed in more detail in the following subsections.

3.1.1. `socket()`

Applications use `socket()` to create a socket descriptor to represent an SCTP endpoint.

The function prototype is

```
int socket(int domain,
           int type,
           int protocol);
```

and one uses `PF_INET` or `PF_INET6` as the domain, `SOCK_SEQPACKET` as the type and `IPPROTO_SCTP` as the protocol.

Here, `SOCK_SEQPACKET` indicates the creation of a one-to-many style socket.

The function returns a socket descriptor or -1 in case of an error.

Using the `PF_INET` domain indicates the creation of an endpoint which can use only IPv4 addresses, while `PF_INET6` creates an endpoint which can use both IPv6 and IPv4 addresses.

3.1.2. `bind()`

Applications use `bind()` to specify which local address and port the SCTP endpoint should associate itself with.

An SCTP endpoint can be associated with multiple addresses. To do this, `sctp_bindx()` is introduced in Section 9.1 to help applications do the job of associating multiple addresses. But note that an endpoint can only be associated with one local port.

These addresses associated with a socket are the eligible transport addresses for the endpoint to send and receive data. The endpoint will also present these addresses to its peers during the association initialization process, see [RFC4960].

After calling `bind()`, if the endpoint wishes to accept new associations on the socket, it must call `listen()` (see

Section 3.1.3).

The function prototype of `bind()` is

```
int bind(int sd,
         struct sockaddr *addr,
         socklen_t addrlen);
```

and the arguments are

`sd`: The socket descriptor returned by `socket()`.

`addr`: The address structure (`struct sockaddr_in` for an IPv4 address or `struct sockaddr_in6` for an IPv6 address, see [RFC3493]).

`addrlen`: The size of the address structure.

It returns 0 on success and -1 in case of an error.

If `sd` is an IPv4 socket, the address passed must be an IPv4 address. If the `sd` is an IPv6 socket, the address passed can either be an IPv4 or an IPv6 address.

Applications cannot call `bind()` multiple times to associate multiple addresses to an endpoint. After the first call to `bind()`, all subsequent calls will return an error.

If the IP address part of `addr` is specified as a wildcard (`INADDR_ANY` for an IPv4 address, or as `IN6ADDR_ANY_INIT` or `in6addr_any` for an IPv6 address), the operating system will associate the endpoint with an optimal address set of the available interfaces. If the IPv4 `sin_port` or IPv6 `sin6_port` is set to 0, the operating system will choose an ephemeral port for the endpoint.

If a `bind()` is not called prior to a `sendmsg()` call that initiates a new association, the system picks an ephemeral port and will choose an address set equivalent to binding with a wildcard address. One of those addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

The completion of this `bind()` process does not allow the SCTP endpoint to accept inbound SCTP association requests. Until a `listen()` system call, described below, is performed on the socket, the SCTP endpoint will promptly reject an inbound SCTP INIT request with an SCTP ABORT.

3.1.3. listen()

By default, a one-to-many style socket does not accept new association requests. An application uses `listen()` to mark a socket as being able to accept new associations.

The function prototype is

```
int listen(int sd,
           int backlog);
```

and the arguments are

`sd`: The socket descriptor of the endpoint.

`backlog`: If `backlog` is non-zero, enable listening, else disable listening.

It returns 0 on success and -1 in case of an error.

Note that one-to-many style socket consumers do not need to call `accept` to retrieve new associations. Calling `accept()` on a one-to-many style socket should return `EOPNOTSUPP`. Rather, new associations are accepted automatically, and notifications of the new associations are delivered via `recvmsg()` with the `SCTP_ASSOC_CHANGE` event (if these notifications are enabled). Clients will typically not call `listen()`, so that they can be assured that only actively initiated associations are possible on the socket. Server or peer-to-peer sockets, on the other hand, will always accept new associations, so a well-written application using server one-to-many style sockets must be prepared to handle new associations from unwanted peers.

Also note that the `SCTP_ASSOC_CHANGE` event provides the association identifier for a new association, so if applications wish to use the association identifier as a parameter to other socket calls, they should ensure that the `SCTP_ASSOC_CHANGE` event is enabled.

3.1.4. sendmsg() and recvmsg()

An application uses the `sendmsg()` and `recvmsg()` call to transmit data to and receive data from its peer.

The function prototypes are

```
ssize_t sendmsg(int sd,
                const struct msghdr *message,
                int flags);
```

and

```
ssize_t recvmsg(int sd,  
                struct msghdr *message,  
                int flags);
```

using the arguments:

sd: The socket descriptor of the endpoint.

message: Pointer to the msghdr structure which contains a single user message and possibly some ancillary data. See Section 5 for complete description of the data structures.

flags: No new flags are defined for SCTP at this level. See Section 5 for SCTP specific flags used in the msghdr structure.

sendmsg() returns the number of bytes accepted by the kernel or -1 in case of an error. recvmsg() returns the number of bytes received or -1 in case of an error.

As described in Section 5, different types of ancillary data can be sent and received along with user data. When sending, the ancillary data is used to specify the sent behavior, such as the SCTP stream number to use. When receiving, the ancillary data is used to describe the received data, such as the SCTP stream sequence number of the message.

When sending user data with sendmsg(), the msg_name field in the msghdr structure will be filled with one of the transport addresses of the intended receiver. If there is no existing association between the sender and the intended receiver, the sender's SCTP stack will set up a new association and then send the user data (see Section 7.5 for more on implicit association setup). If sendmsg() is called with no data and there is no existing association, a new one will be established. The SCTP_INIT type ancillary data can be used to change some of the parameters used to set up a new association. If sendmsg() is called with NULL data, and there is no existing association but the SCTP_ABORT or SCTP_EOF flags are set as described in Section 5.3.4, then -1 is returned and errno is set to EINVAL. Sending a message using sendmsg() is atomic unless explicit EOR marking is enabled on the socket specified by sd (see Section 8.1.26).

If a peer sends a SHUTDOWN, an SCTP_SHUTDOWN_EVENT notification will be delivered if that notification has been enabled, and no more data can be sent to that association. Any attempt to send more data will cause sendmsg() to return with an ESHUTDOWN error. Note that the

socket is still open for reading at this point so it is possible to retrieve notifications.

When receiving a user message with `recvmsg()`, the `msg_name` field in the `msg_hdr` structure will be populated with the source transport address of the user data. The caller of `recvmsg()` can use this address information to determine to which association the received user message belongs. Note that if `SCTP_ASSOC_CHANGE` events are disabled, applications must use the peer transport address provided in the `msg_name` field by `recvmsg()` to perform correlation to an association, since they will not have the association identifier.

If all data in a single message has been delivered, `MSG_EOR` will be set in the `msg_flags` field of the `msg_hdr` structure (see Section 5.1).

If the application does not provide enough buffer space to completely receive a data message, `MSG_EOR` will not be set in `msg_flags`. Successive reads will consume more of the same message until the entire message has been delivered, and `MSG_EOR` will be set.

If the SCTP stack is running low on buffers, it may partially deliver a message. In this case, `MSG_EOR` will not be set, and more calls to `recvmsg()` will be necessary to completely consume the message. Only one message at a time can be partially delivered in any stream. The socket option `SCTP_FRAGMENT_INTERLEAVE` controls various aspects of what interlacing of messages occurs for both the one-to-one and the one-to-many model sockets. Please consult Section 8.1.20 for further details on message delivery options.

3.1.5. `close()`

Applications use `close()` to perform graceful shutdown (as described in Section 10.1 of [RFC4960]) on all the associations currently represented by a one-to-many style socket.

The function prototype is

```
int close(int sd);
```

and the argument is

`sd`: The socket descriptor of the associations to be closed.

0 is returned on success and -1 in case of an error.

To gracefully shutdown a specific association represented by the one-to-many style socket, an application should use the `sendmsg()` call, and include the `SCTP_EOF` flag. A user may optionally terminate an

association non-gracefully by sending with the `SCTP_ABORT` flag set and possibly passing a user specified abort code in the data field. Both flags `SCTP_EOF` and `SCTP_ABORT` are passed with ancillary data (see Section 5.3.4) in the `sendmsg()` call.

If `sd` in the `close()` call is a branched-off socket representing only one association, the shutdown is performed on that association only.

3.1.6. `connect()`

An application may use the `connect()` call in the one-to-many style to initiate an association without sending data.

The function prototype is

```
int connect(int sd,
            const struct sockaddr *nam,
            socklen_t len);
```

and the arguments are

`sd`: The socket descriptor to have a new association added to.

`nam`: The address structure (`struct sockaddr_in` for an IPv4 address or `struct sockaddr_in6` for an IPv6 address, see [RFC3493]).

`len`: The size of the address.

0 is returned on success and -1 in case of an error.

Multiple `connect()` calls can be made on the same socket to create multiple associations. This is different from the semantics of `connect()` on a UDP socket.

Note that SCTP allows data exchange, similar to T/TCP [RFC1644], during the association set up phase. If an application wants to do this, it cannot use the `connect()` call. Instead, it should use `sendto()` or `sendmsg()` to initiate an association. If it uses `sendto()` and it wants to change the initialization behavior, it needs to use the `SCTP_INITMSG` socket option before calling `sendto()`. Or it can use `sendmsg()` with `SCTP_INIT` type ancillary data to initiate an association without calling `setsockopt()`. Note that the implicit setup is supported for the one-to-many style sockets.

SCTP does not support half close semantics. This means that unlike T/TCP, `MSG_EOF` should not be set in the flags parameter when calling `sendto()` or `sendmsg()` when the call is used to initiate a connection. `MSG_EOF` is not an acceptable flag with an SCTP socket.

3.2. Non-blocking mode

Some SCTP application may wish to avoid being blocked when calling a socket interface function.

Once a `bind()` and/or subsequent `sctp_bindx()` calls are complete on a one-to-many style socket, an application may set the non-blocking option by a `fcntl()` (such as `O_NONBLOCK`). After setting the socket to non-blocking mode, the `sendmsg()` function returns immediately. The success or failure of sending the data message (with possible `SCTP_INITMSG` ancillary data) will be signaled by the `SCTP_ASSOC_CHANGE` event with `SCTP_COMM_UP` or `SCTP_CANT_START_ASSOC`. If user data could not be sent (due to a `SCTP_CANT_START_ASSOC`), the sender will also receive an `SCTP_SEND_FAILED_EVENT` event. Events can be received by the user calling `recvmsg()`. A server (having called `listen()`) is also notified of an association up event by the reception of an `SCTP_ASSOC_CHANGE` with `SCTP_COMM_UP` via the calling of `recvmsg()` and possibly the reception of the first data message.

To shutdown the association gracefully, the user must call `sendmsg()` with no data and with the `SCTP_EOF` flag set as described in Section 5.3.4. The function returns immediately, and completion of the graceful shutdown is indicated by an `SCTP_ASSOC_CHANGE` notification of type `SHUTDOWN_COMPLETE` (see Section 6.1.1). Note that this can also be done using the `sctp_sendv()` call described in Section 9.12.

An application is recommended to use caution when using `select()` (or `poll()`) for writing on a one-to-many style socket. The reason being that the interpretation of `select` on write is implementation specific. Generally a positive return on a `select` on write would only indicate that one of the associations represented by the one-to-many socket is writable. An application that writes after the `select()` returns may still block since the association that was writeable is not the destination association of the write call. Likewise `select()` (or `poll()`) for reading from a one-to-many socket will only return an indication that one of the associations represented by the socket has data to be read.

An application that wishes to know that a particular association is ready for reading or writing should either use the one-to-one style or use the `sctp_peeloff()` (see Section 9.2) function to separate the association of interest from the one-to-many socket.

Note some implementations may have an extended `select` call such as `epoll` or `kqueue` that may escape this limitation and allow a `select` on a specific association of a one-to-many socket, but this is an implementation specific detail that a portable application cannot

depend on.

3.3. Special considerations

The fact that a one-to-many style socket can provide access to many SCTP associations through a single socket descriptor, has important implications for both application programmers and system programmers implementing this API. A key issue is how buffer space inside the sockets layer is managed. Because this implementation detail directly affects how application programmers must write their code to ensure correct operation and portability, this section provides some guidance to both implementers and application programmers.

An important feature that SCTP shares with TCP is flow control. Specifically, a sender may not send data faster than the receiver can consume it.

For TCP, flow control is typically provided for in the sockets API as follows. If the reader stops reading, the sender queues messages in the socket layer until the send socket buffer is completely filled. This results in a "stalled connection". Further attempts to write to the socket will block or return the error EAGAIN or EWOULDBLOCK for a non-blocking socket. At some point, either the connection is closed, or the receiver begins to read again freeing space in the output queue.

For one-to-one style SCTP sockets (this includes sockets descriptors that were separated from a one-to-many style socket with `sctp_peeloff()`) the behavior is identical. For one-to-many style SCTP sockets there are multiple associations for a single socket, which makes the situation more complicated. If the implementation uses a single buffer space allocation shared by all associations, a single stalled association can prevent the further sending of data on all associations active on a particular one-to-many style socket.

For a blocking socket, it should be clear that a single stalled association can block the entire socket. For this reason, application programmers may want to use non-blocking one-to-many style sockets. The application should at least be able to send messages to the non-stalled associations.

But a non-blocking socket is not sufficient if the API implementer has chosen a single shared buffer allocation for the socket. A single stalled association would eventually cause the shared allocation to fill, and it would become impossible to send even to non-stalled associations.

The API implementer can solve this problem by providing each

association with its own allocation of outbound buffer space. Each association should conceptually have as much buffer space as it would have if it had its own socket. As a bonus, this simplifies the implementation of `sctp_peeloff()`.

To ensure that a given stalled association will not prevent other non-stalled associations from being writable, application programmers should either:

- o demand that the underlying implementation dedicates independent buffer space reservation to each association (as suggested above), or
- o verify that their application layer protocol does not permit large amounts of unread data at the receiver (this is true of some request-response protocols, for example), or
- o use one-to-one style sockets for association which may potentially stall (either from the beginning, or by using `sctp_peeloff` before sending large amounts of data that may cause a stalled condition).

4. One-to-One Style Interface

The goal of this style is to follow as closely as possible the current practice of using the sockets interface for a connection oriented protocol, such as TCP. This style enables existing applications using connection oriented protocols to be ported to SCTP with very little effort.

One-to-one style sockets can be connected (explicitly or implicitly) at most once, similar to TCP sockets.

Note that some new SCTP features and some new SCTP socket options can only be utilized through the use of `sendmsg()` and `recvmsg()` calls, see Section 4.1.8.

4.1. Basic Operation

A typical server in one-to-one style uses the following system call sequence to prepare an SCTP endpoint for servicing requests:

- o `socket()`
- o `bind()`
- o `listen()`

- o `accept()`

The `accept()` call blocks until a new association is set up. It returns with a new socket descriptor. The server then uses the new socket descriptor to communicate with the client, using `recv()` and `send()` calls to get requests and send back responses.

Then it calls

- o `close()`

to terminate the association.

A typical client uses the following system call sequence to setup an association with a server to request services:

- o `socket()`

- o `connect()`

After returning from `connect()`, the client uses `send()/sendmsg()` and `recv()/recvmsg()` calls to send out requests and receive responses from the server.

The client calls

- o `close()`

to terminate this association when done.

4.1.1. `socket()`

Applications call `socket()` to create a socket descriptor to represent an SCTP endpoint.

The function prototype is

```
int socket(int domain,
           int type,
           int protocol);
```

and one uses `PF_INET` or `PF_INET6` as the domain, `SOCK_STREAM` as the type and `IPPROTO_SCTP` as the protocol.

Here, `SOCK_STREAM` indicates the creation of a one-to-one style socket.

Using the `PF_INET` domain indicates the creation of an endpoint which

can use only IPv4 addresses, while `PF_INET6` creates an endpoint which can use both IPv6 and IPv4 addresses.

4.1.2. `bind()`

Applications use `bind()` to specify which local address and port the SCTP endpoint should associate itself with.

An SCTP endpoint can be associated with multiple addresses. To do this, `sctp_bindx()` is introduced in Section 9.1 to help applications do the job of associating multiple addresses. But note that an endpoint can only be associated with one local port.

These addresses associated with a socket are the eligible transport addresses for the endpoint to send and receive data. The endpoint will also present these addresses to its peers during the association initialization process, see [RFC4960].

The function prototype of `bind()` is

```
int bind(int sd,
         struct sockaddr *addr,
         socklen_t addrlen);
```

and the arguments are

`sd`: The socket descriptor returned by `socket()`.

`addr`: The address structure (`struct sockaddr_in` for an IPv4 address or `struct sockaddr_in6` for an IPv6 address, see [RFC3493]).

`addrlen`: The size of the address structure.

If `sd` is an IPv4 socket, the address passed must be an IPv4 address. If `sd` is an IPv6 socket, the address passed can either be an IPv4 or an IPv6 address.

Applications cannot call `bind()` multiple times to associate multiple addresses to the endpoint. After the first call to `bind()`, all subsequent calls will return an error.

If the IP address part of `addr` is specified as a wildcard (`INADDR_ANY` for an IPv4 address, or as `IN6ADDR_ANY_INIT` or `in6addr_any` for an IPv6 address), the operating system will associate the endpoint with an optimal address set of the available interfaces. If the IPv4 `sin_port` or IPv6 `sin6_port` is set to 0, the operating system will choose an ephemeral port for the endpoint.

If a `bind()` is not called prior to the `connect()` call, the system picks an ephemeral port and will choose an address set equivalent to binding with a wildcard address. One of these addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

The completion of this `bind()` process does not allow the SCTP endpoint to accept inbound SCTP association requests. Until a `listen()` system call, described below, is performed on the socket, the SCTP endpoint will promptly reject an inbound SCTP INIT request with an SCTP ABORT.

4.1.3. `listen()`

Applications use `listen()` to allow the SCTP endpoint to accept inbound associations.

The function prototype is

```
int listen(int sd,
           int backlog);
```

and the arguments are

`sd`: the socket descriptor of the SCTP endpoint.

`backlog`: this specifies the max number of outstanding associations allowed in the socket's accept queue. These are the associations that have finished the four-way initiation handshake (see Section 5 of [RFC4960]) and are in the ESTABLISHED state. Note, a backlog of '0' indicates that the caller no longer wishes to receive new associations.

It returns 0 on success and -1 in case of an error.

4.1.4. `accept()`

Applications use the `accept()` call to remove an established SCTP association from the accept queue of the endpoint. A new socket descriptor will be returned from `accept()` to represent the newly formed association.

The function prototype is

```
int accept(int sd,
           struct sockaddr *addr,
           socklen_t *addrlen);
```

and the arguments are

sd: The listening socket descriptor.

addr: On return, addr (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address, see [RFC3493]) will contain the primary address of the peer endpoint.

addrlen: On return, addrlen will contain the size of addr.

The function returns the socket descriptor for the newly formed association on success and -1 in case of an error.

4.1.5. connect()

Applications use connect() to initiate an association to a peer.

The function prototype is

```
int connect(int sd,
            const struct sockaddr *addr,
            socklen_t addrlen);
```

and the arguments are

sd: The socket descriptor of the endpoint.

addr: The peer's (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address, see [RFC3493]) address.

addrlen: The size of the address.

It returns 0 on success and -1 on error.

This operation corresponds to the ASSOCIATE primitive described in Section 10.1 of [RFC4960].

The number of outbound streams the new association has is stack dependent. Applications can use the SCTP_INITMSG option described in Section 8.1.3 before connecting to change the number of outbound streams.

If a bind() is not called prior to the connect() call, the system picks an ephemeral port and will choose an address set equivalent to binding with INADDR_ANY and IN6ADDR_ANY_INIT for IPv4 and IPv6 socket respectively. One of the addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

Note that SCTP allows data exchange, similar to T/TCP [RFC1644], during the association set up phase. If an application wants to do this, it cannot use the `connect()` call. Instead, it should use `sendto()` or `sendmsg()` to initiate an association. If it uses `sendto()` and it wants to change the initialization behavior, it needs to use the `SCTP_INITMSG` socket option before calling `sendto()`. Or it can use `sendmsg()` with `SCTP_INIT` type ancillary data to initiate an association without calling `setsockopt()`. Note that the implicit setup is supported for the one-to-one style sockets.

SCTP does not support half close semantics. This means that unlike T/TCP, `MSG_EOF` should not be set in the `flags` parameter when calling `sendto()` or `sendmsg()` when the call is used to initiate a connection. `MSG_EOF` is not an acceptable flag with an SCTP socket.

4.1.6. `close()`

Applications use `close()` to gracefully close down an association.

The function prototype is

```
int close(int sd);
```

and the argument is

`sd`: The socket descriptor of the association to be closed.

It returns 0 on success and -1 in case of an error.

After an application calls `close()` on a socket descriptor, no further socket operations will succeed on that descriptor.

4.1.7. `shutdown()`

SCTP differs from TCP in that it does not have half closed semantics. Hence the `shutdown()` call for SCTP is an approximation of the TCP `shutdown()` call, and solves some different problems. Full TCP-compatibility is not provided, so developers porting TCP applications to SCTP may need to recode sections that use `shutdown()`. (Note that it is possible to achieve the same results as half close in SCTP using SCTP streams.)

The function prototype is

```
int shutdown(int sd,  
             int how);
```

and the arguments are

`sd`: The socket descriptor of the association to be closed.

`how`: Specifies the type of shutdown. The values are as follows:

`SHUT_RD`: Disables further receive operations. No SCTP protocol action is taken.

`SHUT_WR`: Disables further send operations, and initiates the SCTP shutdown sequence.

`SHUT_RDWR`: Disables further send and receive operations and initiates the SCTP shutdown sequence.

It returns 0 on success and -1 in case of an error.

The major difference between SCTP and TCP `shutdown()` is that SCTP `SHUT_WR` initiates immediate and full protocol shutdown, whereas TCP `SHUT_WR` causes TCP to go into the half closed state. `SHUT_RD` behaves the same for SCTP as TCP. The purpose of SCTP `SHUT_WR` is to close the SCTP association while still leaving the socket descriptor open. This allows the caller to receive back any data which SCTP is unable to deliver (see Section 6.1.4 for more information) and receive event notifications.

To perform the ABORT operation described in [RFC4960] Section 10.1, an application can use the socket option `SO_LINGER`. It is described in Section 8.1.4.

4.1.8. `sendmsg()` and `recvmsg()`

With a one-to-one style socket, the application can also use `sendmsg()` and `recvmsg()` to transmit data to and receive data from its peer. The semantics is similar to those used in the one-to-many style (see Section 3.1.4), with the following differences:

1. When sending, the `msg_name` field in the `msghdr` is not used to specify the intended receiver, rather it is used to indicate a preferred peer address if the sender wishes to discourage the stack from sending the message to the primary address of the receiver. If the socket is connected and the transport address given is not part of the current association, the data will not be sent and an `SCTP_SEND_FAILED_EVENT` event will be delivered to the application if send failure events are enabled.
2. Using `sendmsg()` on a non-connected one-to-one style socket for implicit connection setup may or may not work depending on the SCTP implementation.

4.1.9. getpeername()

Applications use `getpeername()` to retrieve the primary socket address of the peer. This call is for TCP compatibility, and is not multi-homed. It may not work with one-to-many style sockets depending on the implementation. See Section 9.3 for a multi-homed style version of the call.

The function prototype is

```
int getpeername(int sd,
                struct sockaddr *address,
                socklen_t *len);
```

and the arguments are:

`sd`: The socket descriptor to be queried.

`address`: On return, the peer primary address is stored in this buffer. If the socket is an IPv4 socket, the address will be IPv4. If the socket is an IPv6 socket, the address will be either an IPv6 or IPv4 address.

`len`: The caller should set the length of address here. On return, this is set to the length of the returned address.

It returns 0 on success and -1 in case of an error.

If the actual length of the address is greater than the length of the supplied `sockaddr` structure, the stored address will be truncated.

5. Data Structures

This section discusses important data structures which are specific to SCTP and are used with `sendmsg()` and `recvmsg()` calls to control SCTP endpoint operations and to access ancillary information and notifications.

5.1. The `msghdr` and `cmsghdr` Structures

The `msghdr` structure used in the `sendmsg()` and `recvmsg()` calls, as well as the ancillary data carried in the structure, is the key for the application to set and get various control information from the SCTP endpoint.

The `msghdr` and the related `cmsghdr` structures are defined and discussed in detail in [RFC3542]. They are defined as:

```
struct msghdr {
    void *msg_name;           /* ptr to socket address structure */
    socklen_t msg_namelen;    /* size of socket address structure */
    struct iovec *msg_iov;     /* scatter/gather array */
    int msg_iovlen;           /* # elements in msg_iov */
    void *msg_control;         /* ancillary data */
    socklen_t msg_controllen; /* ancillary data buffer length */
    int msg_flags;            /* flags on received message */
};

struct cmsghdr {
    socklen_t cmsg_len; /* #bytes, including this header */
    int cmsg_level;     /* originating protocol */
    int cmsg_type;      /* protocol-specific type */
    /* followed by unsigned char cmsg_data[]; */
};
```

In the `msghdr` structure, the usage of `msg_name` has been discussed in previous sections (see Section 3.1.4 and Section 4.1.8).

The scatter/gather buffers, or I/O vectors (pointed to by the `msg_iov` field) are treated by SCTP as a single user message for both `sendmsg()` and `recvmsg()`.

SCTP stack uses the ancillary data (`msg_control` field) to communicate the attributes, such as `SCTP_RCVINFO`, of the message stored in `msg_iov` to the socket end point. The different ancillary data types are described in Section 5.3.

The `msg_flags` are not used when sending a message with `sendmsg()`.

If a notification has arrived, `recvmsg()` will return the notification in `msg_iov` field and set `MSG_NOTIFICATION` flag in `msg_flags`. If the `MSG_NOTIFICATION` flag is not set, `recvmsg()` will return data. See Section 6 for more information about notifications.

If all portions of a data frame or notification have been read, `recvmsg()` will return with `MSG_EOR` set in `msg_flags`.

5.2. Ancillary Data Considerations and Semantics

Programming with ancillary socket data (`msg_control`) contains some subtleties and pitfalls, which are discussed below.

5.2.1. Multiple Items and Ordering

Multiple ancillary data items may be included in any call to `sendmsg()` or `recvmsg()`; these may include multiple SCTP or non-SCTP,

such as IP level items, or both.

The ordering of ancillary data items (either by SCTP or another protocol) is not significant and is implementation-dependent, so applications must not depend on any ordering.

SCTP_SNDRCV/SCTP_SNDINFO/SCTP_RCVINFO type ancillary data always correspond to the data in the msghdr's msg_iov member. There can be only one single such type ancillary data for each sendmsg() or recvmsg() call.

5.2.2. Accessing and Manipulating Ancillary Data

Applications can infer the presence of data or ancillary data by examining the msg_iovlen and msg_controllen msghdr members, respectively.

Implementations may have different padding requirements for ancillary data, so portable applications should make use of the macros CMSG_FIRSTHDR, CMSG_NXTHDR, CMSG_DATA, CMSG_SPACE, and CMSG_LEN. See [RFC3542] and the SCTP implementation's documentation for more information. The following is an example, from [RFC3542], demonstrating the use of these macros to access ancillary data:

```
struct msghdr msg;
struct cmsghdr *cmsgptr;

/* fill in msg */

/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
    if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
        u_char *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
    }
}
```

5.2.3. Control Message Buffer Sizing

The information conveyed via SCTP_SNDRCV/SCTP_SNDINFO/SCTP_RCVINFO ancillary data will often be fundamental to the correct and sane operation of the sockets application. This is particularly true of the one-to-many semantics, but also of the one-to-one semantics. For example, if an application needs to send and receive data on

different SCTP streams, SCTP_SNDRCV/SCTP_SNDINFO/SCTP_RCVINFO ancillary data is indispensable.

Given that some ancillary data is critical, and that multiple ancillary data items may appear in any order, applications should be carefully written to always provide a large enough buffer to contain all possible ancillary data that can be presented by `recvmsg()`. If the buffer is too small, and crucial data is truncated, it may pose a fatal error condition.

Thus, it is essential that applications be able to deterministically calculate the maximum required buffer size to pass to `recvmsg()`. One constraint imposed on this specification that makes this possible is that all ancillary data definitions are of a fixed length. One way to calculate the maximum required buffer size might be to take the sum the sizes of all enabled ancillary data item structures, as calculated by `CMSG_SPACE`. For example, if we enabled `SCTP_SNDRCV_INFO` and `IPV6_RECVPKTINFO` [RFC3542], we would calculate and allocate the buffer size as follows:

```
size_t total;
void *buf;

total = CMSG_SPACE(sizeof(struct sctp_sndrcvinfo)) +
        CMSG_SPACE(sizeof(struct in6_pktinfo));

buf = malloc(total);
```

We could then use this buffer (`buf`) for `msg_control` on each call to `recvmsg()` and be assured that we would not lose any ancillary data to truncation.

5.3. SCTP `msg_control` Structures

A key element of all SCTP specific socket extensions is the use of ancillary data to specify and access SCTP specific data via the `struct msghdr`'s `msg_control` member used in `sendmsg()` and `recvmsg()`. Fine-grained control over initialization and sending parameters are handled with ancillary data.

Each ancillary data item is proceeded by a `struct cmsghdr` (see Section 5.1), which defines the function and purpose of the data contained in the `cmsg_data[]` member.

By default on either style socket, SCTP will pass no ancillary data; Specific ancillary data items can be enabled with socket options defined for SCTP; see Section 6.2.

Note that all ancillary types are fixed length; see Section 5.2 for further discussion on this. These data structures use struct `sockaddr_storage` (defined in [RFC3493]) as a portable, fixed length address format.

Other protocols may also provide ancillary data to the socket layer consumer. These ancillary data items from other protocols may intermingle with SCTP data. For example, the IPv6 socket API definitions ([RFC3542] and [RFC3493]) define a number of ancillary data items. If a socket API consumer enables delivery of both SCTP and IPv6 ancillary data, they both may appear in the same `msg_control` buffer in any order. An application may thus need to handle other types of ancillary data besides those passed by SCTP.

The sockets application must provide a buffer large enough to accommodate all ancillary data provided via `recvmsg()`. If the buffer is not large enough, the ancillary data will be truncated and the `msg_hdr`'s `msg_flags` will include `MSG_CTRUNC`.

5.3.1. SCTP Initiation Structure (SCTP_INIT)

This `cmsghdr` structure provides information for initializing new SCTP associations with `sendmsg()`. The `SCTP_INITMSG` socket option uses this same data structure. This structure is not used for `recvmsg()`.

<code>cmsgh_level</code>	<code>cmsgh_type</code>	<code>cmsgh_data[]</code>
<code>IPPROTO_SCTP</code>	<code>SCTP_INIT</code>	<code>struct sctp_initmsg</code>

The `sctp_initmsg` structure is defined below:

```
struct sctp_initmsg {
    uint16_t sinit_num_ostreams;
    uint16_t sinit_max_instreams;
    uint16_t sinit_max_attempts;
    uint16_t sinit_max_init_timeo;
};
```

`sinit_num_ostreams`: This is an integer number representing the number of streams that the application wishes to be able to send to. This number is confirmed in the `SCTP_COMM_UP` notification and must be verified since it is a negotiated number with the remote endpoint. The default value of 0 indicates to use the endpoint default value.

`sinit_max_instreams`: This value represents the maximum number of inbound streams the application is prepared to support. This value is bounded by the actual implementation. In other words the user may be able to support more streams than the Operating System. In such a case, the Operating System limit overrides the value requested by the user. The default value of 0 indicates to use the endpoints default value.

`sinit_max_attempts`: This integer specifies how many attempts the SCTP endpoint should make at resending the INIT. This value overrides the system SCTP 'Max.Init.Retransmits' value. The default value of 0 indicates to use the endpoints default value. This is normally set to the system's default 'Max.Init.Retransmit' value.

`sinit_max_init_timeo`: This value represents the largest Time-Out or RTO value (in milliseconds) to use in attempting an INIT. Normally the 'RTO.Max' is used to limit the doubling of the RTO upon timeout. For the INIT message this value may override 'RTO.Max'. This value must not influence 'RTO.Max' during data transmission and is only used to bound the initial setup time. A default value of 0 indicates to use the endpoints default value. This is normally set to the system's 'RTO.Max' value (60 seconds).

5.3.2. SCTP Header Information Structure (SCTP_SNDRCV) - DEPRECATED

This `cmsghdr` structure specifies SCTP options for `sendmsg()` and describes SCTP header information about a received message through `recvmsg()`. This structure mixes the send and receive path. `SCTP_SNDINFO` described in Section 5.3.4 and `SCTP_RCVINFO` described in Section 5.3.5 split this information. These structures should be used, when possible, since `SCTP_SNDRCV` is deprecated.

```
+-----+-----+-----+
| cmsg_level | cmsg_type | cmsg_data[] |
+-----+-----+-----+
| IPPROTO_SCTP | SCTP_SNDRCV | struct sctp_sndrcvinfo |
+-----+-----+-----+
```

The `sctp_sndrcvinfo` structure is defined below:

```
struct sctp_sndrcvinfo {
    uint16_t sinfo_stream;
    uint16_t sinfo_ssn;
    uint16_t sinfo_flags;
    uint32_t sinfo_ppid;
    uint32_t sinfo_context;
    uint32_t sinfo_timetolive;
    uint32_t sinfo_tsn;
    uint32_t sinfo_cumtsn;
    sctp_assoc_t sinfo_assoc_id;
};
```

sinfo_stream: For `recvmsg()` the SCTP stack places the message's stream number in this value. For `sendmsg()` this value holds the stream number that the application wishes to send this message to. If a sender specifies an invalid stream number an error indication is returned and the call fails.

sinfo_ssn: For `recvmsg()` this value contains the stream sequence number that the remote endpoint placed in the DATA chunk. For fragmented messages this is the same number for all deliveries of the message (if more than one `recvmsg()` is needed to read the message). The `sendmsg()` call will ignore this parameter.

sinfo_flags: This field may contain any of the following flags and is composed of a bitwise OR of these values.

recvmsg() flags:

SCTP_UNORDERED: This flag is present when the message was sent un-ordered.

sendmsg() flags:

SCTP_UNORDERED: This flag requests the un-ordered delivery of the message. If this flag is clear the datagram is considered an ordered send.

SCTP_ADDR_OVER: This flag, in the one-to-many style, requests the SCTP stack to override the primary destination address with the address found with the `sendto/sendmsg` call.

SCTP_ABORT: Setting this flag causes the specified association to abort by sending an ABORT message to the peer. The ABORT chunk will contain an error cause 'User Initiated Abort' with cause code 12. The cause specific information of this error cause is provided in `msg_iov`.

SCTP_EOF: Setting this flag invokes the SCTP graceful shutdown procedure on the specified association. Graceful shutdown assures that all data queued by both endpoints is successfully transmitted before closing the association.

SCTP_SENDALL: This flag, if set, will cause a one-to-many model socket to send the message to all associations that are currently established on this socket. For the one-to-one socket, this flag has no effect.

sinfo_ppid: This value in `sendmsg()` is an unsigned integer that is passed to the remote end in each user message. In `recvmsg()` this value is the same information that was passed by the upper layer in the peer application. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the `htonl()` computation.

sinfo_context: This value is an opaque 32 bit context datum that is used in the `sendmsg()` function. This value is passed back to the upper layer if an error occurs on the send of a message and is retrieved with each undelivered message.

sinfo_timetolive: For the sending side, this field contains the message time to live in milliseconds. The sending side will expire the message within the specified time period if the message as not been sent to the peer within this time period. This value will override any default value set using any socket option. Also note that the value of 0 is special in that it indicates no timeout should occur on this message.

sinfo_tsn: For the receiving side, this field holds a TSN that was assigned to one of the SCTP Data Chunks. For the sending side it is ignored.

sinfo_cumtsn: This field will hold the current cumulative TSN as known by the underlying SCTP layer. Note this field is ignored when sending.

sinfo_assoc_id: The association handle field, `sinfo_assoc_id`, holds the identifier for the association announced in the `SCTP_COMM_UP` notification. All notifications for a given association have the same identifier. Ignored for one-to-one style sockets.

An `sctp_sndrcvinfo` item always corresponds to the data in `msg_iov`.

5.3.3. Extended SCTP Header Information Structure (SCTP_EXTRCV) - DEPRECATED

This `cmsghdr` structure specifies SCTP options for SCTP header information about a received message via `recvmsg()`. Note that this structure is an extended version of `SCTP_SNDRCV` (see Section 5.3.2) and will only be received if the user has set the socket option `SCTP_USE_EXT_RCVINFO` to true in addition to any event subscription needed to receive ancillary data. See Section 8.1.22 on this socket option. Note that next message data is not valid unless the current message is completely read, i.e. the `MSG_EOR` is set, in other words if the application has more data to read from the current message then no next message information will be available.

`SCTP_NXTINFO` described in Section 5.3.6 should be used when possible, since `SCTP_EXTRCV` is considered deprecated.

+-----+-----+-----+			
<code>cmsg_level</code>	<code>cmsg_type</code>	<code>cmsg_data[]</code>	
+-----+-----+-----+			
<code>IPPROTO_SCTP</code>	<code>SCTP_EXTRCV</code>	<code>struct sctp_extrcvinfo</code>	
+-----+-----+-----+			

The `sctp_extrcvinfo` structure is defined below:

```
struct sctp_extrcvinfo {
    uint16_t sinfo_stream;
    uint16_t sinfo_ssn;
    uint16_t sinfo_flags;
    uint32_t sinfo_ppid;
    uint32_t sinfo_context;
    uint32_t sinfo_pr_value;
    uint32_t sinfo_tsn;
    uint32_t sinfo_cumtsn;
    uint16_t serinfo_next_flags;
    uint16_t serinfo_next_stream;
    uint32_t serinfo_next_aid;
    uint32_t serinfo_next_length;
    uint32_t serinfo_next_ppid;
    sctp_assoc_t sinfo_assoc_id;
};
```

`sinfo_*`: Please see Section 5.3.2 for the details for these fields.

`serinfo_next_flags`: This bitmask will hold one or more of the following values:

SCTP_NEXT_MSG_AVAIL: This bit, when set to 1, indicates that next message information is available i.e.: `next_stream`, `next_asocid`, `next_length` and `next_ppid` fields all have valid values. If this bit is set to 0, then these fields are not valid and should be ignored.

SCTP_NEXT_MSG_ISCOMPLETE: This bit, when set, indicates that the next message is completely in the receive buffer. The `next_length` field thus contains the entire message size. If this flag is set to 0, then the `next_length` field only contains part of the message size since the message is still being received (it is being partially delivered).

SCTP_NEXT_MSG_IS_UNORDERED: This bit, when set, indicates that the next message to be received was sent by the peer as unordered. If this bit is not set (i.e. the bit is 0) the next message to be read is an ordered message in the stream specified.

SCTP_NEXT_MSG_IS_NOTIFICATION: This bit, when set, indicates that the next message to be received is not a message from the peer, but instead is a `MSG_NOTIFICATION` from the local SCTP stack.

`serinfo_next_stream`: This value, when valid (see `serinfo_next_flags`), contains the next stream number that will be received on a subsequent call to one of the receive message functions.

`serinfo_next_aid`: This value, when valid (see `serinfo_next_flags`), contains the next association identifier that will be received on a subsequent call to one of the receive message functions.

`serinfo_next_length`: This value, when valid (see `serinfo_next_flags`), contains the length of the next message that will be received on a subsequent call to one of the receive message functions. Note that this length may be a partial length depending on the settings of `next_flags`.

`serinfo_next_ppid`: This value, when valid (see `serinfo_next_flags`), contains the `ppid` of the next message that will be received on a subsequent call to one of the receive message functions.

5.3.4. SCTP Send Information Structure (`SCTP_SNDINFO`)

This `cmsghdr` structure specifies SCTP options for `sendmsg()`.

+-----+-----+-----+			
cmsg_level	cmsg_type	cmsg_data[]	
+-----+-----+-----+			
IPPROTO_SCTP	SCTP_SNDINFO	struct sctp_sndinfo	
+-----+-----+-----+			

The sctp_sndinfo structure is defined below:

```
struct sctp_sndinfo {
    uint16_t snd_sid;
    uint16_t snd_flags;
    uint32_t snd_ppid;
    uint32_t snd_context;
    sctp_assoc_t snd_assoc_id;
};
```

snd_sid: This value holds the stream number that the application wishes to send this message to. If a sender specifies an invalid stream number an error indication is returned and the call fails.

snd_flags: This field may contain any of the following flags and is composed of a bitwise OR of these values.

SCTP_UNORDERED: This flag requests the un-ordered delivery of the message. If this flag is clear the datagram is considered an ordered send.

SCTP_ADDR_OVER: This flag, in the one-to-many style, requests the SCTP stack to override the primary destination address with the address found with the sendto()/sendmsg call.

SCTP_ABORT: Setting this flag causes the specified association to abort by sending an ABORT message to the peer. The ABORT chunk will contain an error cause 'User Initiated Abort' with cause code 12. The cause specific information of this error cause is provided in msg_iov.

SCTP_EOF: Setting this flag invokes the SCTP graceful shutdown procedures on the specified association. Graceful shutdown assures that all data queued by both endpoints is successfully transmitted before closing the association.

SCTP_SENDALL: This flag, if set, will cause a one-to-many model socket to send the message to all associations that are currently established on this socket. For the one-to-one socket, this flag has no effect.

snd_ppid: This value in `sendmsg()` is an unsigned integer that is passed to the remote end in each user message. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the `htonl()` computation.

snd_context: This value is an opaque 32 bit context datum that is used in the `sendmsg()` function. This value is passed back to the upper layer if an error occurs on the send of a message and is retrieved with each undelivered message.

snd_assoc_id: The association handle field, `sinfo_assoc_id`, holds the identifier for the association announced in the `SCTP_COMM_UP` notification. All notifications for a given association have the same identifier. Ignored for one-to-one style sockets.

An `sctp_sndinfo` item always corresponds to the data in `msg_iov`.

5.3.5. SCTP Receive Information Structure (`SCTP_RCVINFO`)

This `cmsghdr` structure describes SCTP receive information about a received message through `recvmsg()`.

To enable the delivery of this information an application must use the `SCTP_RECVRCVINFO` socket option (see Section 8.1.29).

<code>msg_level</code>	<code>msg_type</code>	<code>msg_data[]</code>
<code>IPPROTO_SCTP</code>	<code>SCTP_RCVINFO</code>	<code>struct sctp_rcvinfo</code>

The `sctp_rcvinfo` structure is defined below:

```
struct sctp_rcvinfo {
    uint16_t rcv_sid;
    uint16_t rcv_ssn;
    uint16_t rcv_flags;
    uint32_t rcv_ppid;
    uint32_t rcv_tsn;
    uint32_t rcv_cumtsn;
    uint32_t rcv_context;
    sctp_assoc_t rcv_assoc_id;
};
```

`rcv_sid`: The SCTP stack places the message's stream number in this value.

`rcv_ssn`: This value contains the stream sequence number that the remote endpoint placed in the DATA chunk. For fragmented messages this is the same number for all deliveries of the message (if more than one `recvmsg()` is needed to read the message).

`rcv_flags`: This field may contain any of the following flags and is composed of a bitwise OR of these values.

`SCTP_UNORDERED`: This flag is present when the message was sent un-ordered.

`rcv_ppid`: This value is the same information that was passed by the upper layer in the peer application. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the `ntohl()` computation.

`rcv_tsn`: This field holds a TSN that was assigned to one of the SCTP Data Chunks.

`rcv_cumtsn`: This field will hold the current cumulative TSN as known by the underlying SCTP layer.

`rcv_assoc_id`: The association handle field, `sinfo_assoc_id`, holds the identifier for the association announced in the `SCTP_COMM_UP` notification. All notifications for a given association have the same identifier. Ignored for one-to-one style sockets.

`rcv_context`: This value is an opaque 32 bit context datum that was set by the user with the `SCTP_CONTEXT` socket option. This value is passed back to the upper layer if an error occurs on the send of a message and is retrieved with each undelivered message.

An `sctp_rcvinfo` item always corresponds to the data in `msg_iov`.

5.3.6. SCTP Next Receive Information Structure (`SCTP_NXTINFO`)

This `cmsghdr` structure describes SCTP receive information of the next message which will be delivered through `recvmsg()` if this information is already available when delivering the current message.

To enable the delivery of this information an application must use the `SCTP_RECVNXTINFO` socket option (see Section 8.1.30).

+-----+-----+-----+
cmsg_level cmsg_type cmsg_data[]
+-----+-----+-----+
IPPROTO_SCTP SCTP_NXTINFO struct sctp_nxtinfo
+-----+-----+-----+

The `sctp_nxtinfo` structure is defined below:

```
struct sctp_nxtinfo {  
    uint16_t nxt_sid;  
    uint16_t nxt_flags;  
    uint32_t nxt_ppid;  
    uint32_t nxt_length;  
    sctp_assoc_t nxt_assoc_id;  
};
```

`nxt_sid`: The SCTP stack places the next message's stream number in this value.

`nxt_flags`: This field may contain any of the following flags and is composed of a bitwise OR of these values.

`SCTP_UNORDERED`: This flag is present when the next message was sent un-ordered.

`SCTP_COMPLETE`: This flag indicates that the entire message has been received and is in the socket buffer. Note that this has special implications with respect to the `nxt_length` field, see `nxt_length` description below.

`SCTP_NOTIFICATION`: This flag is present when the next message is not a user message but instead is a notification.

`nxt_ppid`: This value is the same information that was passed by the upper layer in the peer application for the next message. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the `ntohl()` computation.

`nxt_length`: This value is the length of the message currently within the socket buffer. This might NOT be the entire length of the message since a partial delivery may be in progress. Only if the flag `SCTP_COMPLETE` is set in the `nxt_flags` field does this field represent the entire next message size.

`nxt_assoc_id`: The association handle field of the next message, `nxt_assoc_id`, holds the identifier for the association announced in the `SCTP_COMM_UP` notification. All notifications for a given association have the same identifier. Ignored for one-to-one style sockets.

5.3.7. SCTP PR-SCTP Information Structure (`SCTP_PRINFO`)

This `cmsghdr` structure specifies SCTP options for `sendmsg()`.

<code>cmsgh_level</code>	<code>cmsgh_type</code>	<code>cmsgh_data[]</code>
<code>IPPROTO_SCTP</code>	<code>SCTP_PRINFO</code>	<code>struct sctp_prinfo</code>

The `sctp_prinfo` structure is defined below:

```
struct sctp_prinfo {
    uint16_t pr_policy;
    uint32_t pr_value;
};
```

`pr_policy`: This specifies which PR-SCTP policy is used. Using `SCTP_PR_SCTP_NONE` results in a reliable transmission. When `SCTP_PR_SCTP_TTL` is used, the PR-SCTP policy "timed reliability" defined in [RFC3758] is used. In this case, the lifetime is provided in `pr_value`.

`pr_value`: The meaning of this field depends on the PR-SCTP policy specified by the `pr_policy` field. It is ignored when `SCTP_PR_SCTP_NONE` is specified. In case of `SCTP_PR_SCTP_TTL` the lifetime in milliseconds is specified.

An `sctp_prinfo` item always corresponds to the data in `msg_iov`.

5.3.8. SCTP AUTH Information Structure (`SCTP_AUTHINFO`)

This `cmsghdr` structure specifies SCTP options for `sendmsg()`.

<code>cmsgh_level</code>	<code>cmsgh_type</code>	<code>cmsgh_data[]</code>
<code>IPPROTO_SCTP</code>	<code>SCTP_AUTHINFO</code>	<code>struct sctp_authinfo</code>

The `sctp_authinfo` structure is defined below:

```

struct sctp_authinfo {
    uint16_t auth_keynumber;
};

```

`auth_keynumber`: This specifies the shared key identifier used for sending the user message.

An `sctp_authinfo` item always corresponds to the data in `msg_iov`. Please note that the SCTP implementation must not bundle user messages that needs to be authenticated using different shared key identifiers.

5.3.9. SCTP Destination IPv4 Address Structure (SCTP_DSTADDRV4)

This `cmsghdr` structure specifies SCTP options for `sendmsg()`.

```

+-----+-----+-----+
| cmsg_level | cmsg_type | cmsg_data[] |
+-----+-----+-----+
| IPPROTO_SCTP | SCTP_DSTADDRV4 | struct in_addr |
+-----+-----+-----+

```

This ancillary data can be used to provide more than one destination address to `sendmsg()`. It can be used to implement `sctp_sendv()` using `sendmsg()`.

5.3.10. SCTP Destination IPv6 Address Structure (SCTP_DSTADDRV6)

This `cmsghdr` structure specifies SCTP options for `sendmsg()`.

```

+-----+-----+-----+
| cmsg_level | cmsg_type | cmsg_data[] |
+-----+-----+-----+
| IPPROTO_SCTP | SCTP_DSTADDRV6 | struct in6_addr |
+-----+-----+-----+

```

This ancillary data can be used to provide more than one destination address to `sendmsg()`. It can be used to implement `sctp_sendv()` using `sendmsg()`.

6. SCTP Events and Notifications

An SCTP application may need to understand and process events and errors that happen on the SCTP stack. These events include network status changes, association startups, remote operational errors and undeliverable messages. All of these can be essential for the application.

When an SCTP application layer does a `recvmsg()` the message read is normally a data message from a peer endpoint. If the application wishes to have the SCTP stack deliver notifications of non-data events, it sets the appropriate socket option for the notifications it wants. See Section 6.2 for these socket options. When a notification arrives, `recvmsg()` returns the notification in the application-supplied data buffer via `msg_iov`, and sets `MSG_NOTIFICATION` in `msg_flags`.

This section details the notification structures. Every notification structure carries some common fields which provide general information.

A `recvmsg()` call will return only one notification at a time. Just as when reading normal data, it may return part of a notification if the `msg_iov` buffer is not large enough. If a single read is not sufficient, `msg_flags` will have `MSG_EOR` clear. The user must finish reading the notification before subsequent data can arrive.

6.1. SCTP Notification Structure

The notification structure is defined as the union of all notification types.

```
union sctp_notification {
    struct sctp_tlv {
        uint16_t sn_type; /* Notification type. */
        uint16_t sn_flags;
        uint32_t sn_length;
    } sn_header;
    struct sctp_assoc_change sn_assoc_change;
    struct sctp_paddr_change sn_paddr_change;
    struct sctp_remote_error sn_remote_error;
    struct sctp_send_failed sn_send_failed;
    struct sctp_shutdown_event sn_shutdown_event;
    struct sctp_adaptation_event sn_adaptation_event;
    struct sctp_pdapi_event sn_pdapi_event;
    struct sctp_authkey_event sn_auth_event;
    struct sctp_sender_dry_event sn_sender_dry_event;
    struct sctp_send_failed_event sn_send_failed_event;
};
```

`sn_type`: The following list describes the SCTP notification and event types for the field `sn_type`.

SCTP_ASSOC_CHANGE: This tag indicates that an association has either been opened or closed. Refer to Section 6.1.1 for details.

SCTP_PEER_ADDR_CHANGE: This tag indicates that an address that is part of an existing association has experienced a change of state (e.g. a failure or return to service of the reachability of an endpoint via a specific transport address). Please see Section 6.1.2 for data structure details.

SCTP_REMOTE_ERROR: The attached error message is an Operational Error received from the remote peer. It includes the complete TLV sent by the remote endpoint. See Section 6.1.3 for the detailed format.

SCTP_SEND_FAILED_EVENT: The attached datagram could not be sent to the remote endpoint. This structure includes the original SCTP_SNDINFO that was used in sending this message i.e. this structure uses the sctp_sndinfo per Section 6.1.11.

SCTP_SHUTDOWN_EVENT: The peer has sent a SHUTDOWN. No further data should be sent on this socket.

SCTP_ADAPTATION_INDICATION: This notification holds the peer's indicated adaptation layer. Please see Section 6.1.6.

SCTP_PARTIAL_DELIVERY_EVENT: This notification is used to tell a receiver that the partial delivery has been aborted. This may indicate the association is about to be aborted. Please see Section 6.1.7.

SCTP_AUTHENTICATION_EVENT: This notification is used to tell a receiver that either an error occurred on authentication, or a new key was made active. See Section 6.1.8.

SCTP_SENDER_DRY_EVENT: This notification is used to inform the application that the sender has no more user data queued for transmission nor retransmission. See Section 6.1.9.

sn_flags: These are notification-specific flags.

sn_length: This is the length of the whole sctp_notification structure including the sn_type, sn_flags, and sn_length fields.

6.1.1. SCTP_ASSOC_CHANGE

Communication notifications inform the application that an SCTP association has either begun or ended. The identifier for a new association is provided by this notification. The notification information has the following format:

```
struct sctp_assoc_change {  
    uint16_t sac_type;  
    uint16_t sac_flags;  
    uint32_t sac_length;  
    uint16_t sac_state;  
    uint16_t sac_error;  
    uint16_t sac_outbound_streams;  
    uint16_t sac_inbound_streams;  
    sctp_assoc_t sac_assoc_id;  
    uint8_t  sac_info[];  
};
```

sac_type: It should be SCTP_ASSOC_CHANGE.

sac_flags: Currently unused.

sac_length: This field is the total length of the notification data, including the notification header.

sac_state: This field holds one of a number of values that communicate the event that happened to the association. They include:

SCTP_COMM_UP: A new association is now ready and data may be exchanged with this peer. When an association has been established successfully, this notification should be the first one.

SCTP_COMM_LOST: The association has failed. The association is now in the closed state. If SEND_FAILED notifications are turned on, an SCTP_COMM_LOST is accompanied by a series of SCTP_SEND_FAILED_EVENT events, one for each outstanding message.

SCTP_RESTART: SCTP has detected that the peer has restarted.

SCTP_SHUTDOWN_COMP: The association has gracefully closed.

`SCTP_CANT_STR_ASSOC`: The association failed to setup. If non blocking mode is set and data was sent (on a one-to-many style socket), an `SCTP_CANT_STR_ASSOC` is accompanied by a series of `SCTP_SEND_FAILED_EVENT` events, one for each outstanding message.

`sac_error`: If the state was reached due to an error condition (e.g. `SCTP_COMM_LOST`) any relevant error information is available in this field. This corresponds to the protocol error codes defined in [RFC4960].

`sac_outbound_streams`:

`sac_inbound_streams`: The maximum number of streams allowed in each direction are available in `sac_outbound_streams` and `sac_inbound_streams`.

`sac_assoc_id`: The `sac_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

`sac_info`: If the `sac_state` is `SCTP_COMM_LOST` and an ABORT chunk was received for this association, `sac_info[]` contains the complete ABORT chunk as defined in the SCTP specification [RFC4960] Section 3.3.7. If the `sac_state` is `SCTP_COMM_UP` or `SCTP_RESTART`, `sac_info` may contain an array of `uint8_t` describing the features that the current association supports. Features may include

`SCTP_ASSOC_SUPPORTS_PR`: Both endpoints support the protocol extension described in [RFC3758].

`SCTP_ASSOC_SUPPORTS_AUTH`: Both endpoints support the protocol extension described in [RFC4895].

`SCTP_ASSOC_SUPPORTS_ASCONF`: Both endpoints support the protocol extension described in [RFC5061].

`SCTP_ASSOC_SUPPORTS_MULTIBUF`: For a one-to-many style socket, the local endpoints use separate send and/or receive buffers for each SCTP association.

6.1.2. `SCTP_PEER_ADDR_CHANGE`

When a destination address of a multi-homed peer encounters a state change a peer address change event is sent. The notification has the following format:

```
struct sctp_paddr_change {  
    uint16_t spc_type;  
    uint16_t spc_flags;  
    uint32_t spc_length;  
    struct sockaddr_storage spc_aaddr;  
    uint32_t spc_state;  
    uint32_t spc_error;  
    sctp_assoc_t spc_assoc_id;  
}
```

spc_type: It should be SCTP_PEER_ADDR_CHANGE.

spc_flags: Currently unused.

spc_length: This field is the total length of the notification data, including the notification header.

spc_aaddr: The affected address field holds the remote peer's address that is encountering the change of state.

spc_state: This field holds one of a number of values that communicate the event that happened to the address. They include:

SCTP_ADDR_AVAILABLE: This address is now reachable. This notification is provided whenever an address becomes reachable.

SCTP_ADDR_UNREACHABLE: The address specified can no longer be reached. Any data sent to this address is rerouted to an alternate until this address becomes reachable. This notification is provided whenever an address becomes unreachable.

SCTP_ADDR_REMOVED: The address is no longer part of the association.

SCTP_ADDR_ADDED: The address is now part of the association.

SCTP_ADDR_MADE_PRIM: This address has now been made to be the primary destination address. This notification is provided whenever an address is made primary.

spc_error: If the state was reached due to any error condition (e.g. SCTP_ADDR_UNREACHABLE) any relevant error information is available in this field.

`spc_assoc_id`: The `spc_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

6.1.3. SCTP_REMOTE_ERROR

A remote peer may send an Operational Error message to its peer. This message indicates a variety of error conditions on an association. The entire ERROR chunk as it appears on the wire is included in an SCTP_REMOTE_ERROR event. Please refer to the SCTP specification [RFC4960] and any extensions for a list of possible error formats. An SCTP error notification has the following format:

```
struct sctp_remote_error {
    uint16_t sre_type;
    uint16_t sre_flags;
    uint32_t sre_length;
    uint16_t sre_error;
    sctp_assoc_t sre_assoc_id;
    uint8_t sre_data[];
};
```

`sre_type`: It should be SCTP_REMOTE_ERROR.

`sre_flags`: Currently unused.

`sre_length`: This field is the total length of the notification data, including the notification header and the contents of `sre_data`.

`sre_error`: This value represents one of the Operational Error causes defined in the SCTP specification, in network byte order.

`sre_assoc_id`: The `sre_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

`sre_data`: This contains the ERROR chunk as defined in the SCTP specification [RFC4960] Section 3.3.10.

6.1.4. SCTP_SEND_FAILED - DEPRECATED

Please note that this notification is deprecated. Use SCTP_SEND_FAILED_EVENT instead.

If SCTP cannot deliver a message, it can return back the message as a notification if the SCTP_SEND_FAILED event is enabled. The

notification has the following format:

```
struct sctp_send_failed {
    uint16_t ssf_type;
    uint16_t ssf_flags;
    uint32_t ssf_length;
    uint32_t ssf_error;
    struct sctp_sndrcvinfo ssf_info;
    sctp_assoc_t ssf_assoc_id;
    uint8_t ssf_data[];
};
```

ssf_type: It should be SCTP_SEND_FAILED.

ssf_flags: The flag value will take one of the following values:

SCTP_DATA_UNSENT: Indicates that the data was never put on the wire.

SCTP_DATA_SENT: Indicates that the data was put on the wire.
Note that this does not necessarily mean that the data was (or was not) successfully delivered.

ssf_length: This field is the total length of the notification data, including the notification header and the payload in ssf_data.

ssf_error: This value represents the reason why the send failed, and if set, will be an SCTP protocol error code as defined in [RFC4960] Section 3.3.10.

ssf_info: The ancillary data (struct sctp_sndrcvinfo) used to send the undelivered message. Regardless of if ancillary data is used or not, the ssf_info.sinfo_flags field indicates if the complete message or only part of the message is returned in ssf_data. If only part of the message is returned, it means that the part which is not present has been sent successfully to the peer.

If the complete message cannot be sent, the SCTP_DATA_NOT_FRAG flag is set in ssf_info.sinfo_flags. If the first part of the message is sent successfully, the SCTP_DATA_LAST_FRAG is set. This means that the tail end of the message is returned in ssf_data.

ssf_assoc_id: The ssf_assoc_id field, ssf_assoc_id, holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

`ssf_data`: The undelivered message or part of the undelivered message will be present in the `ssf_data` field. Note that the `ssf_info.sinfo_flags` field as noted above should be used to determine if a complete message is present or just a piece of the message. Note that only user data is present in this field, any chunk headers or SCTP common headers must be removed by the SCTP stack.

6.1.5. SCTP_SHUTDOWN_EVENT

When a peer sends a SHUTDOWN, SCTP delivers this notification to inform the application that it should cease sending data.

```
struct sctp_shutdown_event {
    uint16_t sse_type;
    uint16_t sse_flags;
    uint32_t sse_length;
    sctp_assoc_t sse_assoc_id;
};
```

`sse_type`: It should be `SCTP_SHUTDOWN_EVENT`.

`sse_flags`: Currently unused.

`sse_length`: This field is the total length of the notification data, including the notification header. It will generally be `sizeof(struct sctp_shutdown_event)`.

`sse_flags`: Currently unused.

`sse_assoc_id`: The `sse_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

6.1.6. SCTP_ADAPTATION_INDICATION

When a peer sends an Adaptation Layer Indication parameter as described in [RFC5061], SCTP delivers this notification to inform the application about the peer's adaptation layer indication.

```
struct sctp_adaptation_event {
    uint16_t sai_type;
    uint16_t sai_flags;
    uint32_t sai_length;
    uint32_t sai_adaptation_ind;
    sctp_assoc_t sai_assoc_id;
};
```


sai_type: It should be SCTP_ADAPTATION_INDICATION.

sai_flags: Currently unused.

sai_length: This field is the total length of the notification data, including the notification header. It will generally be sizeof(struct sctp_adaptation_event).

sai_adaptation_ind: This field holds the bit array sent by the peer in the adaptation layer indication parameter.

sai_assoc_id: The sai_assoc_id field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

6.1.7. SCTP_PARTIAL_DELIVERY_EVENT

When a receiver is engaged in a partial delivery of a message this notification will be used to indicate various events.

```
struct sctp_pdapi_event {
    uint16_t pdapi_type;
    uint16_t pdapi_flags;
    uint32_t pdapi_length;
    uint32_t pdapi_indication;
    uint32_t pdapi_stream;
    uint32_t pdapi_seq;
    sctp_assoc_t pdapi_assoc_id;
};
```

pdapi_type: It should be SCTP_PARTIAL_DELIVERY_EVENT.

pdapi_flags: Currently unused.

pdapi_length: This field is the total length of the notification data, including the notification header. It will generally be sizeof(struct sctp_pdapi_event).

pdapi_indication: This field holds the indication being sent to the application. Currently there is only one defined value:

SCTP_PARTIAL_DELIVERY_ABORTED: This indicates that the partial delivery of a user message has been aborted. This happens, for example, if an association is aborted while a partial delivery is going on or the user message gets abandoned using PR-SCTP while the partial delivery of this message is going on.

pdapi_stream: This field holds the stream on which the partial delivery event happened.

pdapi_seq: This field holds the stream sequence number which was being partially delivered.

pdapi_assoc_id: The pdapi_assoc_id field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket this field is ignored.

6.1.1.8. SCTP_AUTHENTICATION_EVENT

[RFC4895] defines an extension to authenticate SCTP messages. The following notification is used to report different events relating to the use of this extension.

```
struct sctp_authkey_event {
    uint16_t auth_type;
    uint16_t auth_flags;
    uint32_t auth_length;
    uint16_t auth_keynumber;
    uint32_t auth_indication;
    sctp_assoc_t auth_assoc_id;
};
```

auth_type: It should be SCTP_AUTHENTICATION_EVENT.

auth_flags: Currently unused.

auth_length: This field is the total length of the notification data, including the notification header. It will generally be sizeof(struct sctp_authkey_event).

auth_keynumber: This field holds the keynumber for the affected key indicated in the event (depends on auth_indication).

auth_indication: This field holds the error or indication being reported. The following values are currently defined:

SCTP_AUTH_NEW_KEY: This report indicates that a new key has been made active (used for the first time by the peer) and is now the active key. The auth_keynumber field holds the user specified key number.

SCTP_AUTH_NO_AUTH: This report indicates that the peer does not support SCTP AUTH as defined in [RFC4895].

SCTP_AUTH_FREE_KEY: This report indicates that the SCTP implementation will no longer use the key identifier specified in `auth_keynumber`.

auth_assoc_id: The `auth_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket this field is ignored.

6.1.9. SCTP_SENDER_DRY_EVENT

When the SCTP stack has no more user data to send or retransmit, this notification is given to the user. Also, at the time when a user app subscribes to this event, if there is no data to be sent or retransmit, the stack will immediately send up this notification.

```
struct sctp_sender_dry_event {
    uint16_t sender_dry_type;
    uint16_t sender_dry_flags;
    uint32_t sender_dry_length;
    sctp_assoc_t sender_dry_assoc_id;
};
```

sender_dry_type: It should be `SCTP_SENDER_DRY_EVENT`.

sender_dry_flags: Currently unused.

sender_dry_length: This field is the total length of the notification data, including the notification header. It will generally be `sizeof(struct sctp_sender_dry_event)`.

sender_dry_assoc_id: The `sender_dry_assoc_id` field holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket this field is ignored.

6.1.10. SCTP_NOTIFICATIONS_STOPPED_EVENT

SCTP notifications, when subscribed to, are reliable. They are always delivered as long as there is space in the socket receive buffer. However, if an implementation experiences a notification storm, it may run out of socket buffer space. When this occurs it may wish to disable notifications. If the implementation chooses to do this, it will append a final notification `SCTP_NOTIFICATIONS_STOPPED_EVENT`. This notification is a union

sctp_notification, where only the struct sctp_tlv (see the union above) is used. It only contains this type in the sn_type field, the sn_length field set to the size of an sctp_tlv structure and the sn_flags set to 0. If an application receives this notification, it will need to re-subscribe to any notifications of interest to it, except for the sctp_data_io_event (note that SCTP_EVENTS is deprecated).

An endpoint is automatically subscribed to this event as soon as it is subscribed to any event other than data io events.

6.1.11. SCTP_SEND_FAILED_EVENT

If SCTP cannot deliver a message, it can return back the message as a notification if the SCTP_SEND_FAILED_EVENT event is enabled. The notification has the following format:

```
struct sctp_send_failed_event {
    uint16_t ssfe_type;
    uint16_t ssfe_flags;
    uint32_t ssfe_length;
    uint32_t ssfe_error;
    struct sctp_sndinfo ssfe_info;
    sctp_assoc_t ssfe_assoc_id;
    uint8_t ssfe_data[];
};
```

ssfe_type: It should be SCTP_SEND_FAILED_EVENT.

ssfe_flags: The flag value will take one of the following values:

SCTP_DATA_UNSENT: Indicates that the data was never put on the wire.

SCTP_DATA_SENT: Indicates that the data was put on the wire.
Note that this does not necessarily mean that the data was (or was not) successfully delivered.

ssfe_length: This field is the total length of the notification data, including the notification header and the payload in ssf_data.

ssfe_error: This value represents the reason why the send failed, and if set, will be an SCTP protocol error code as defined in [RFC4960] Section 3.3.10.

`ssfe_info`: The ancillary data (struct `sctp_sndinfo`) used to send the undelivered message. Regardless of if ancillary data is used or not, the `ssfe_info.sinfo_flags` field indicates if the complete message or only part of the message is returned in `ssf_data`. If only part of the message is returned, it means that the part which is not present has been sent successfully to the peer.

If the complete message cannot be sent, the `SCTP_DATA_NOT_FRAG` flag is set in `ssfe_info.sinfo_flags`. If the first part of the message is sent successfully, the `SCTP_DATA_LAST_FRAG` is set. This means that the tail end of the message is returned in `ssf_data`.

`ssfe_assoc_id`: The `ssfe_assoc_id` field, `ssf_assoc_id`, holds the identifier for the association. All notifications for a given association have the same association identifier. For a one-to-one style socket, this field is ignored.

`ssf_data`: The undelivered message or part of the undelivered message will be present in the `ssf_data` field. Note that the `ssf_info.sinfo_flags` field as noted above should be used to determine if a complete message is present or just a piece of the message. Note that only user data is present in this field, any chunk headers or SCTP common headers must be removed by the SCTP stack.

6.2. Notification Interest Options

6.2.1. SCTP_EVENTS option - DEPRECATED

Please note that this option is deprecated. Use the `SCTP_EVENT` option described in Section 6.2.2 instead.

To receive SCTP event notifications, an application registers its interest by setting the `SCTP_EVENTS` socket option. The application then uses `recvmsg()` to retrieve notifications. A notification is stored in the data part (`msg_iov`) of the struct `msg_hdr`. The socket option uses the following structure:

```
struct sctp_event_subscribe {
    uint8_t sctp_data_io_event;
    uint8_t sctp_association_event;
    uint8_t sctp_address_event;
    uint8_t sctp_send_failure_event;
    uint8_t sctp_peer_error_event;
    uint8_t sctp_shutdown_event;
    uint8_t sctp_partial_delivery_event;
    uint8_t sctp_adaptation_layer_event;
    uint8_t sctp_authentication_event;
    uint8_t sctp_sender_dry_event;
};
```

`sctp_data_io_event`: Setting this flag to 1 will cause the reception of SCTP_SNDRCV information on a per message basis. The application will need to use the `recvmsg()` interface so that it can receive the event information contained in the `msg_control` field. Setting the flag to 0 will disable the reception of the message control information. Note that this is not really a notification and this is stored in the ancillary data (`msg_control`), not in the data part (`msg_iov`).

`sctp_association_event`: Setting this flag to 1 will enable the reception of association event notifications. Setting the flag to 0 will disable association event notifications.

`sctp_address_event`: Setting this flag to 1 will enable the reception of address event notifications. Setting the flag to 0 will disable address event notifications.

`sctp_send_failure_event`: Setting this flag to 1 will enable the reception of send failure event notifications. Setting the flag to 0 will disable send failure event notifications.

`sctp_peer_error_event`: Setting this flag to 1 will enable the reception of peer error event notifications. Setting the flag to 0 will disable peer error event notifications.

`sctp_shutdown_event`: Setting this flag to 1 will enable the reception of shutdown event notifications. Setting the flag to 0 will disable shutdown event notifications.

`sctp_partial_delivery_event`: Setting this flag to 1 will enable the reception of partial delivery notifications. Setting the flag to 0 will disable partial delivery event notifications.

sctp_adaptation_layer_event: Setting this flag to 1 will enable the reception of adaptation layer notifications. Setting the flag to 0 will disable adaptation layer event notifications.

sctp_authentication_event: Setting this flag to 1 will enable the reception of authentication layer notifications. Setting the flag to 0 will disable authentication layer event notifications.

sctp_sender_dry_event: Setting this flag to 1 will enable the reception of sender dry notifications. Setting the flag to 0 will disable sender dry event notifications.

An example where an application would like to receive data_io_events and association_events but no others would be as follows:

```
{
    struct sctp_event_subscribe events;

    memset(&events, 0, sizeof(events));

    events.sctp_data_io_event = 1;
    events.sctp_association_event = 1;

    setsockopt(sd, IPPROTO_SCTP, SCTP_EVENTS, &events, sizeof(events));
}
```

Note that for one-to-many style SCTP sockets, the caller of `recvmsg()` receives ancillary data and notifications for all associations bound to the file descriptor. For one-to-one style SCTP sockets, the caller receives ancillary data and notifications only for the single association bound to the file descriptor.

By default both the one-to-one style and the one-to-many style socket do not subscribe to any notification.

6.2.2. SCTP_EVENT option

The `SCTP_EVENTS` socket option has one issue for future compatibility. As new features are added the structure (`sctp_event_subscribe`) must be expanded. This can cause an application binary interface (ABI) issue unless an implementation has added padding at the end of the structure. To avoid this problem, `SCTP_EVENTS` has been deprecated and a new socket option `SCTP_EVENT` has taken its place. The option is used with the following structure:

```
struct sctp_event {
    sctp_assoc_t se_assoc_id;
    uint16_t     se_type;
    uint8_t      se_on;
};
```

se_assoc_id: The `se_assoc_id` field is ignored for one-to-one style sockets. For one-to-many style sockets this field can be a particular association identifier or `SCTP_{FUTURE|CURRENT|ALL}_ASSOC`.

se_type: The `se_type` field can be filled with any value that would show up in the respective `sn_type` field (in the `sctp_tlv` structure of the notification).

se_on: The `se_on` field is set to 1 to turn on an event and set to 0 to turn off an event.

To use this option the user fills in this structure and then calls the `setsockopt()` to turn on or off an individual event. The following is an example use of this option:

```
{
    struct sctp_event event;

    memset(&event, 0, sizeof(event));

    event.se_assoc_id = SCTP_FUTURE_ASSOC;
    event.se_type = SCTP_SENDER_DRY_EVENT;
    event.se_on = 1;
    setsockopt(sd, IPPROTO_SCTP, SCTP_EVENT, &event, sizeof(event));
}
```

By default both the one-to-one style and the one-to-many style socket do not subscribe to any notification.

7. Common Operations for Both Styles

7.1. `send()`, `recv()`, `sendto()`, and `recvfrom()`

Applications can use `send()` and `sendto()` to transmit data to the peer of an SCTP endpoint. `recv()` and `recvfrom()` can be used to receive data from the peer.

The function prototypes are


```
ssize_t send(int sd,
             const void *msg,
             size_t len,
             int flags);

ssize_t sendto(int sd,
              const void *msg,
              size_t len,
              int flags,
              const struct sockaddr *to,
              socklen_t tolen);

ssize_t recv(int sd,
            void *buf,
            size_t len,
            int flags);

ssize_t recvfrom(int sd,
                void *buf,
                size_t len,
                int flags,
                struct sockaddr *from,
                socklen_t *fromlen);
```

and the arguments are

sd: The socket descriptor of an SCTP endpoint.

msg: The message to be sent.

len: The size of the message or the size of the buffer.

to: One of the peer addresses of the association to be used to send the message.

tolen: The size of the address.

buf: The buffer to store a received message.

from: The buffer to store the peer address used to send the received message.

fromlen: The size of the from address.

flags: (described below).

These calls give access to only basic SCTP protocol features. If either peer in the association uses multiple streams, or sends unordered data, these calls will usually be inadequate, and may deliver the data in unpredictable ways.

SCTP has the concept of multiple streams in one association. The above calls do not allow the caller to specify on which stream a message should be sent. The system uses stream 0 as the default stream for `send()` and `sendto()`. `recv()` and `recvfrom()` return data from any stream, but the caller can not distinguish the different streams. This may result in data seeming to arrive out of order. Similarly, if a data chunk is sent unordered, `recv()` and `recvfrom()` provide no indication.

SCTP is message based. The msg buffer above in `send()` and `sendto()` is considered to be a single message. This means that if the caller wants to send a message that is composed by several buffers, the caller needs to combine them before calling `send()` or `sendto()`. Alternately, the caller can use `sendmsg()` to do that without combining them. Sending a message using `send()` or `sendto()` is atomic unless explicit EOR marking is enabled on the socket specified by `sd`. Using `sendto()` on a non-connected one-to-one style socket for implicit connection setup may or may not work depending on the SCTP implementation. `recv()` and `recvfrom()` cannot distinguish message boundaries (i.e. there is no way to observe the `MSG_EOR` flag to detect partial delivery).

In receiving, if the buffer supplied is not large enough to hold a complete message, the receive call acts like a stream socket and returns as much data as will fit in the buffer.

Note, the `send()` and `recv()` calls may not be used for a one-to-many style socket.

Note, if an application calls a `send()` or `sendto()` function with no user data the SCTP implementation should reject the request with an appropriate error message. An implementation is not allowed to send a DATA chunk with no user data [RFC4960].

7.2. `setsockopt()` and `getsockopt()`

Applications use `setsockopt()` and `getsockopt()` to set or retrieve socket options. Socket options are used to change the default behavior of socket calls. They are described in Section 8.

The function prototypes are

```
int getsockopt(int sd,
               int level,
               int optname,
               void *optval,
               socklen_t *optlen);
```

and

```
int setsockopt(int sd,
               int level,
               int optname,
               const void *optval,
               socklen_t optlen);
```

and the arguments are

sd: The socket descriptor.

level: Set to IPPROTO_SCTP for all SCTP options.

optname: The option name.

optval: The buffer to store the value of the option.

optlen: The size of the buffer (or the length of the option returned).

They return 0 on success and -1 in case of an error.

All socket options set on a one-to-one style listening socket also apply to all future accepted sockets. For one-to-many style sockets often a socket option will pass a structure that includes an `assoc_id` field. This field can be filled with the association identifier of a particular association and unless otherwise specified can be filled with one of the following constants:

SCTP_FUTURE_ASSOC: Specifies that only future associations created after this socket option will be affected by this call.

SCTP_CURRENT_ASSOC: Specifies that only currently existing associations will be affected by this call, future associations will still receive the previous default value.

SCTP_ALL_ASSOC: Specifies that all current and future associations will be affected by this call.

7.3. read() and write()

Applications can use read() and write() to send and receive data to and from a peer. They have the same semantics as send() and recv() except that the flags parameter cannot be used.

7.4. getsockname()

Applications use getsockname() to retrieve the locally-bound socket address of the specified socket. This is especially useful if the caller let SCTP choose a local port. This call is for single homed endpoints. It does not work well with multi-homed endpoints. See Section 9.5 for a multi-homed version of the call.

The function prototype is

```
int getsockname(int sd,
                struct sockaddr *address,
                socklen_t *len);
```

and the arguments are

sd: The socket descriptor to be queried.

address: On return, one locally bound address (chosen by the SCTP stack) is stored in this buffer. If the socket is an IPv4 socket, the address will be IPv4. If the socket is an IPv6 socket, the address will be either an IPv6 or IPv4 address.

len: The caller should set the length of the address here. On return, this is set to the length of the returned address.

It returns 0 on success and -1 in case of an error.

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by address is unspecified.

7.5. Implicit Association Setup

The application can begin sending and receiving data using the sendmsg()/recvmsg() or sendto()/recvfrom() calls, without going through any explicit association setup procedures (i.e., no connect())

calls required).

Whenever `sendmsg()` or `sendto()` is called and the SCTP stack at the sender finds that no association exists between the sender and the intended receiver (identified by the address passed either in the `msg_name` field of `msg_hdr` structure in the `sendmsg()` call or the `dest_addr` field in the `sendto()` call), the SCTP stack will automatically setup an association to the intended receiver.

Upon the successful association setup an `SCTP_COMM_UP` notification will be dispatched to the socket at both the sender and receiver side. This notification can be read by the `recvmsg()` system call (see Section 3.1.4).

Note, if the SCTP stack at the sender side supports bundling, the first user message may be bundled with the COOKIE ECHO message [RFC4960].

When the SCTP stack sets up a new association implicitly, the `SCTP_INIT` type ancillary data may also be passed along (see Section 5.3.1 for details of the data structures) to change some parameters used in setting up a new association.

If this information is not present in the `sendmsg()` call, or if the implicit association setup is triggered by a `sendto()` call, the default association initialization parameters will be used. These default association parameters may be set with respective `setsockopt()` calls or be left to the system defaults.

Implicit association setup cannot be initiated by `send()` calls.

8. Socket Options

The following sub-section describes various SCTP level socket options that are common to both styles. SCTP associations can be multi-homed. Therefore, certain option parameters include a `sockaddr_storage` structure to select which peer address the option should be applied to.

For the one-to-many style sockets, an `sctp_assoc_t` (association identifier) parameter is used to identify the association instance that the operation affects. So it must be set when using this style.

For the one-to-one style sockets and branched off one-to-many style sockets (see Section 9.2) this association ID parameter is ignored.

Note that socket or IP level options are set or retrieved per socket.

This means that for one-to-many style sockets, the options will be applied to all associations (similar to using `SCTP_ALL_ASSOC` as the association identifier) belonging to the socket. For one-to-one style, these options will be applied to all peer addresses of the association controlled by the socket. Applications should be careful in setting those options.

For some IP stacks `getsockopt()` is read-only; so a new interface will be needed when information must be passed both into and out of the SCTP stack. The syntax for `sctp_opt_info()` is

```
int sctp_opt_info(int sd,
                  sctp_assoc_t id,
                  int opt,
                  void *arg,
                  socklen_t *size);
```

The `sctp_opt_info()` call is a replacement for `getsockopt()` only and will not set any options associated with the specified socket. A `setsockopt()` must be used to set any writeable option.

For one-to-many style sockets, `id` specifies the association to query. For one-to-one style sockets, `id` is ignored. For one-to-many sockets, any association identifier in the structure provided as `arg` is ignored and `id` takes precedence.

Note that `SCTP_CURRENT_ASSOC` and `SCTP_ALL_ASSOC` cannot be used with `sctp_opt_info()` or in `getsockopt()` calls. Using them will result in an error (returning `-1` and `errno` set to `EINVAL`). `SCTP_FUTURE_ASSOC` can be used to query information for future associations.

The field `opt` specifies which SCTP socket option to get. It can get any socket option currently supported that requests information (either read/write options or read only) such as:

`SCTP_RTOINFO`

`SCTP_ASSOCINFO`

`SCTP_PRIMARY_ADDR`

`SCTP_PEER_ADDR_PARAMS`

`SCTP_DEFAULT_SEND_PARAM`

SCTP_MAX_SEG
SCTP_AUTH_ACTIVE_KEY
SCTP_DELAYED_SACK
SCTP_MAX_BURST
SCTP_CONTEXT
SCTP_EVENT
SCTP_DEFAULT_SNDINFO
SCTP_DEFAULT_PRINFO
SCTP_STATUS
SCTP_GET_PEER_ADDR_INFO
SCTP_PEER_AUTH_CHUNKS
SCTP_LOCAL_AUTH_CHUNKS

The `arg` field is an option-specific structure buffer provided by the caller. See the rest of this sections subsections for more information on these options and option-specific structures.

`sctp_opt_info()` returns 0 on success, or on failure returns -1 and sets `errno` to the appropriate error code.

8.1. Read / Write Options

8.1.1. Retransmission Timeout Parameters (SCTP_RTOINFO)

The protocol parameters used to initialize and limit the retransmission timeout (RTO) are tunable. See [RFC4960] for more information on how these parameters are used in RTO calculation.

The following structure is used to access and modify these parameters:

```
struct sctp_rtoinfo {  
    sctp_assoc_t srto_assoc_id;  
    uint32_t srto_initial;  
    uint32_t srto_max;  
    uint32_t srto_min;  
};
```

`srto_initial`: This contains the initial RTO value.

`srto_max` and `srto_min`: These contain the maximum and minimum bounds for all RTOs.

`srto_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the application may fill in an association identifier or `SCTP_FUTURE_ASSOC`. It is an error to use `SCTP_{CURRENT|ALL}_ASSOC` in `srto_assoc_id`.

All times are given in milliseconds. A value of 0, when modifying the parameters, indicates that the current value should not be changed.

To access or modify these parameters, the application should call `getsockopt()` or `setsockopt()` respectively with the option name `SCTP_RTOINFO`.

8.1.2. Association Parameters (`SCTP_ASSOCINFO`)

This option is used to both examine and set various association and endpoint parameters. See [RFC4960] for more information on how this parameter is used.

The following structure is used to access and modify these parameters:

```
struct sctp_assocparams {
    sctp_assoc_t sasoc_assoc_id;
    uint16_t sasoc_asocmaxrxt;
    uint16_t sasoc_number_peer_destinations;
    uint32_t sasoc_peer_rwnd;
    uint32_t sasoc_local_rwnd;
    uint32_t sasoc_cookie_life;
};
```

`sasoc_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the application may fill in an association identifier or `SCTP_FUTURE_ASSOC`. It is an error to use `SCTP_{CURRENT|ALL}_ASSOC` in `sasoc_assoc_id`.

`sasoc_asocmaxrxt`: This contains the maximum retransmission attempts to make for the association.

`sasoc_number_peer_destinations`: This is the number of destination addresses that the peer has.

`sasoc_peer_rwnd`: This holds the current value of the peers `rwnd` (reported in the last SACK) minus any outstanding data (i.e. data in flight).

`sasoc_local_rwnd`: This holds the last reported `rwnd` that was sent to the peer.

`sasoc_cookie_life`: This is the association's cookie life value used when issuing cookies.

The values of the `sasoc_peer_rwnd` is meaningless when examining endpoint information (i.e. it is only valid when examining information on a specific association).

All time values are given in milliseconds. A value of 0, when modifying the parameters, indicates that the current value should not be changed.

The values of the `sasoc_asocmaxrxt` and `sasoc_cookie_life` may be set on either an endpoint or association basis. The `rwnd` and destination counts (`sasoc_number_peer_destinations`, `sasoc_peer_rwnd`, `sasoc_local_rwnd`) are not settable and any value placed in these is ignored.

To access or modify these parameters, the application should call `getsockopt()` or `setsockopt()` respectively with the option name `SCTP_ASSOCINFO`.

The maximum number of retransmissions before an address is considered unreachable is also tunable, but is address-specific, so it is covered in a separate option. If an application attempts to set the value of the association maximum retransmission parameter to more than the sum of all maximum retransmission parameters, `setsockopt()` may return an error. The reason for this, from [RFC4960] Section 8.2:

Note: When configuring the SCTP endpoint, the user should avoid having the value of 'Association.Max.Retrans' (`sasoc_maxrxt` in this option) larger than the summation of the 'Path.Max.Retrans' (see Section 8.1.12 on `spp_pathmaxrxt`) of all the destination addresses for the remote endpoint. Otherwise, all the destination addresses may become inactive while the endpoint still considers the peer endpoint reachable.

8.1.3. Initialization Parameters (SCTP_INITMSG)

Applications can specify protocol parameters for the default association initialization. The structure used to access and modify

these parameters is defined in Section 5.3.1. The option name argument to `setsockopt()` and `getsockopt()` is `SCTP_INITMSG`.

Setting initialization parameters is effective only on an unconnected socket (for one-to-many style sockets only future associations are affected by the change).

8.1.4. `SO_LINGER`

An application can use this option to perform the SCTP ABORT primitive. This option affects all associations related to the socket.

The linger option structure is:

```
struct linger {
    int l_onoff; /* option on/off */
    int l_linger; /* linger time */
};
```

To enable the option, set `l_onoff` to 1. If the `l_linger` value is set to 0, calling `close()` is the same as the ABORT primitive. If the value is set to a negative value, the `setsockopt()` call will return an error. If the value is set to a positive value `linger_time`, the `close()` can be blocked for at most `linger_time`. Please note that the time unit is seconds according to POSIX, but might be different on specific platforms. If the graceful shutdown phase does not finish during this period, `close()` will return but the graceful shutdown phase will continue in the system.

Note, this is a socket level option, not an SCTP level option. When using this option, an application must specify a level of `SOL_SOCKET` in the call.

8.1.5. `SCTP_NODELAY`

Turn on/off any Nagle-like algorithm. This means that packets are generally sent as soon as possible and no unnecessary delays are introduced, at the cost of more packets in the network. In particular, not using any Nagle-like algorithm might reduce the bundling of small user messages in cases where this would require an additional delay.

Turning this option on disables any Nagle-like algorithm.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.6. SO_RCVBUF

Sets the receive buffer size in octets. For SCTP one-to-one style sockets, this controls the receiver window size. For one-to-many style sockets the meaning is implementation dependent. It might control the receive buffer for each association bound to the socket descriptor or it might control the receive buffer for the whole socket. This option expects an integer.

Note, this is a socket level option, not an SCTP level option. When using this option, an application must specify a level of SOL_SOCKET in the call.

8.1.7. SO_SNDBUF

Sets the send buffer size. For SCTP one-to-one style sockets, this controls the amount of data SCTP may have waiting in internal buffers to be sent. This option therefore bounds the maximum size of data that can be sent in a single send call. For one-to-many style sockets, the effect is the same, except that it applies to one or all associations (see Section 3.3) bound to the socket descriptor used in the setsockopt() or getsockopt() call. The option applies to each association's window size separately. This option expects an integer.

Note, this is a socket level option, not an SCTP level option. When using this option, an application must specify a level of SOL_SOCKET in the call.

8.1.8. Automatic Close of Associations (SCTP_AUTOCLOSE)

This socket option is applicable to the one-to-many style socket only. When set it will cause associations that are idle for more than the specified number of seconds to automatically close using the graceful shutdown procedure. An association being idle is defined as an association that has not sent or received user data. The special value of '0' indicates that no automatic close of any association should be performed, this is the default value. This option expects an integer defining the number of seconds of idle time before an association is closed.

An application using this option should enable receiving the association change notification. This is the only mechanism an application is informed about the closing of an association. After an association is closed, the association identifier assigned to it can be reused. An application should be aware of this to avoid the possible problem of sending data to an incorrect peer endpoint.

8.1.9. Set Primary Address (SCTP_PRIMARY_ADDR)

Requests that the local SCTP stack uses the enclosed peer address as the association's primary. The enclosed address must be one of the association peer's addresses.

The following structure is used to make a set peer primary request:

```
struct sctp_setprim {  
    sctp_assoc_t ssp_assoc_id;  
    struct sockaddr_storage ssp_addr;  
};
```

`ssp_addr`: The address to set as primary. No wildcard address is allowed.

`ssp_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets it identifies the association for this request. Note that the special `sctp_assoc_t` `SCTP_{FUTURE|ALL|CURRENT}_ASSOC` are not allowed.

8.1.10. Set Adaptation Layer Indicator (SCTP_ADAPTATION_LAYER)

Requests that the local endpoint set the specified Adaptation Layer Indication parameter for all future INIT and INIT-ACK exchanges.

The following structure is used to access and modify this parameter:

```
struct sctp_setadaptation {  
    uint32_t ssb_adaptation_ind;  
};
```

`ssb_adaptation_ind`: The adaptation layer indicator that will be included in any outgoing Adaptation Layer Indication parameter.

8.1.11. Enable/Disable Message Fragmentation (SCTP_DISABLE_FRAGMENTS)

This option is a on/off flag and is passed as an integer where a non-zero is on and a zero is off. If enabled no SCTP message fragmentation will be performed. The effect of enabling this option are that if a message being sent exceeds the current PMTU size, the message will not be sent and instead an error will be indicated to the user. If this option is disabled (the default) then a message exceeding the size of the PMTU will be fragmented and reassembled by the peer.

8.1.12. Peer Address Parameters (SCTP_PEER_ADDR_PARAMS)

Applications can enable or disable heartbeats for any peer address of an association, modify an address's heartbeat interval, force a heartbeat to be sent immediately, and adjust the address's maximum number of retransmissions sent before an address is considered unreachable.

The following structure is used to access and modify an address's parameters:

```
struct sctp_paddrparams {
    sctp_assoc_t spp_assoc_id;
    struct sockaddr_storage spp_address;
    uint32_t spp_hbinterval;
    uint16_t spp_pathmaxrxt;
    uint32_t spp_pathmtu;
    uint32_t spp_flags;
    uint32_t spp_ipv6_flowlabel;
    uint8_t spp_dscp;
};
```

spp_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the application may fill in an association identifier or SCTP_FUTURE_ASSOC for this query. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in spp_assoc_id.

spp_address: This specifies which address is of interest. If a wildcard address is provided it applies to all current and future paths.

spp_hbinterval: This contains the value of the heartbeat interval, in milliseconds (HB.Interval in [RFC4960]). Note that unless the spp_flag is set to SPP_HB_ENABLE the value of this field is ignored. Note also that a value of zero indicates the current setting should be left unchanged. To set an actual value of zero the use of the flag SPP_HB_TIME_IS_ZERO should be used. Even when it is set to 0, it does not mean that SCTP will continuously send out heartbeat since the actual interval also includes the current RTO and jitter (see Section 8.3 in [RFC4960]).

spp_pathmaxrxt: This contains the maximum number of retransmissions before this address shall be considered unreachable. Note that a value of zero indicates the current setting should be left unchanged.

`spp_pathmtu`: The current path MTU of the peer address. It is the number of bytes available in an SCTP packet for chunks. Providing a value of 0 does not change the current setting. If a positive value is provided and `SPP_PMTUD_DISABLE` is set in the `spp_flags`, the given value is used as the path MTU. If `SPP_PMTUD_ENABLE` is set in the `spp_flags`, the `spp_pathmtu` field is ignored.

`spp_ipv6_flowlabel`: This field is used in conjunction with the `SPP_IPV6_FLOWLABEL` flag and contains the IPv6 flowlabel. The 20 least significant bits are used for the flowlabel. This setting has precedence over any IPv6 layer setting.

`spp_dscp`: This field is used in conjunction with the `SPP_DSCP` flag and contains the Differentiated Services Code Point (DSCP). The 6 most significant bits are used for the DSCP. This setting has precedence over any IPv4 or IPv6 layer setting.

`spp_flags`: These flags are used to control various features on an association. The flag field is a bit mask which may contain zero or more of the following options:

`SPP_HB_ENABLE`: Enable heartbeats on the specified address.

`SPP_HB_DISABLE`: Disable heartbeats on the specified address.
Note that `SPP_HB_ENABLE` and `SPP_HB_DISABLE` are mutually exclusive, only one of these two should be specified. Enabling both fields will have undetermined results.

`SPP_HB_DEMAND`: Request a user initiated heartbeat to be made immediately. This must not be used in conjunction with a wildcard address.

`SPP_HB_TIME_IS_ZERO`: Specifies that the time for heartbeat delay is to be set to the value of 0 milliseconds.

`SPP_PMTUD_ENABLE`: This field will enable PMTU discovery upon the specified address.

`SPP_PMTUD_DISABLE`: This field will disable PMTU discovery upon the specified address. Note that if the address field is empty then all addresses on the association are affected. Note also that `SPP_PMTUD_ENABLE` and `SPP_PMTUD_DISABLE` are mutually exclusive. Enabling both will have undetermined results.

`SPP_IPV6_FLOWLABEL`: Setting this flag enables the setting of the IPv6 flowlabel value. The value is contained in the `spp_ipv6_flowlabel` field.

Upon retrieval, this flag will be set to indicate that the `spp_ipv6_flowlabel` field has a valid value returned. If a specific destination address is set (in the `spp_address` field), then the value returned is that of the address. If just an association is specified (and no address), then the association's default flowlabel is returned. If neither an association nor a destination is specified, then the socket's default flowlabel is returned. For non IPv6 sockets, this flag will be left cleared.

SPP_DSCP: Setting this flag enables the setting of the DSCP value associated with either the association or a specific address. The value is obtained in the `spp_dscp` field.

Upon retrieval, this flag will be set to indicate that the `spp_dscp` field has a valid value returned. If a specific destination address is set when called (in the `spp_address` field) then that specific destination address' DSCP value is returned. If just an association is specified then the association default DSCP is returned. If neither an association nor a destination is specified, then the sockets default DSCP is returned.

Please note that changing the flowlabel or DSCP value will affect all packets sent by the SCTP stack after setting these parameters. The flowlabel might also be set via the `sin6_flowinfo` field of the `sockaddr_in6` structure.

8.1.13. Set Default Send Parameters (`SCTP_DEFAULT_SEND_PARAM`) - DEPRECATED

Please note that this options is deprecated. Section 8.1.31 should be used instead.

Applications that wish to use the `sendto()` system call may wish to specify a default set of parameters that would normally be supplied through the inclusion of ancillary data. This socket option allows such an application to set the default `sctp_sndrcvinfo` structure. The application that wishes to use this socket option simply passes the `sctp_sndrcvinfo` structure defined in Section 5.3.2 to this call. The input parameters accepted by this call include `sinfo_stream`, `sinfo_flags`, `sinfo_ppid`, `sinfo_context`, and `sinfo_timetolive`. The `sinfo_flags` is composed of a bitwise OR of `SCTP_UNORDERED`, `SCTP_EOF`, and `SCTP_SENDALL`. The `sinfo_assoc_id` field specifies the association to apply the parameters to. For a one-to-many style socket any of the predefined constants are also allowed in this field. The field is ignored on the one-to-one style.

8.1.14. Set Notification and Ancillary Events (SCTP_EVENTS) - DEPRECATED

This socket option is used to specify various notifications and ancillary data the user wishes to receive. Please see Section 6.2.1 for a full description of this option and its usage. Note that this option is considered deprecated and present for backward compatibility. New applications should use the SCTP_EVENT option. See Section 6.2.2 for a full description of that option as well.

8.1.15. Set/Clear IPv4 Mapped Addresses (SCTP_I_WANT_MAPPED_V4_ADDR)

This socket option is a boolean flag which turns on or off the mapping of IPv4 addresses. If this option is turned on, then IPv4 addresses will be mapped to V6 representation. If this option is turned off, then no mapping will be done of V4 addresses and a user will receive both PF_INET6 and PF_INET type addresses on the socket. See [RFC3542] for more details on mapped V6 addresses.

If this socket option is used on a socket of type PF_INET an error is returned.

By default this option is turned off and expects an integer to be passed where a non-zero value turns on the option and a zero value turns off the option.

8.1.16. Get or Set the Maximum Fragmentation Size (SCTP_MAXSEG)

This option will get or set the maximum size to put in any outgoing SCTP DATA chunk. If a message is larger than this size it will be fragmented by SCTP into the specified size. Note that the underlying SCTP implementation may fragment into smaller sized chunks when the PMTU of the underlying association is smaller than the value set by the user. The default value for this option is '0' which indicates the user is not limiting fragmentation and only the PMTU will affect SCTP's choice of DATA chunk size. Note also that values set larger than the maximum size of an IP datagram will effectively let SCTP control fragmentation (i.e. the same as setting this option to 0).

The following structure is used to access and modify this parameter:

```
struct sctp_assoc_value {
    sctp_assoc_t assoc_id;
    uint32_t assoc_value;
};
```


`assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets this parameter indicates which association the user is performing an action upon. It is an error to use `SCTP_{CURRENT|ALL}_ASSOC` in `assoc_id`.

`assoc_value`: This parameter specifies the maximum size in bytes.

8.1.17. Get or Set the List of Supported HMAC Identifiers (`SCTP_HMAC_IDENT`)

This option gets or sets the list of HMAC algorithms that the local endpoint requires the peer to use.

The following structure is used to get or set these identifiers:

```
struct sctp_hmacalgo {
    uint32_t shmac_number_of_idents;
    uint16_t shmac_idents[];
};
```

`shmac_number_of_idents`: This field gives the number of elements present in the array `shmac_idents`.

`shmac_idents`: This parameter contains an array of HMAC identifiers that the local endpoint is requesting the peer to use, in priority order. The following identifiers are valid:

- * `SCTP_AUTH_HMAC_ID_SHA1`
- * `SCTP_AUTH_HMAC_ID_SHA256`

Note that the list supplied must include `SCTP_AUTH_HMAC_ID_SHA1` and may include any of the other values in its preferred order (lowest list position has the highest preference in algorithm selection). Note also that the lack of `SCTP_AUTH_HMAC_ID_SHA1`, or the inclusion of an unknown HMAC identifier (including optional identifiers unknown to the implementation) will cause the set option to fail and return an error.

8.1.18. Get or Set the Active Shared Key (`SCTP_AUTH_ACTIVE_KEY`)

This option will get or set the active shared key to be used to build the association shared key.

The following structure is used to access and modify these parameters:

```
struct sctp_authkeyid {
    sctp_assoc_t scact_assoc_id;
    uint16_t scact_keynumber;
};
```

scact_assoc_id: This parameter sets the active key of the specified association. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can be used. For one-to-one sockets, this parameter is ignored. Note, however, that this option will set the active key on the association if the socket is connected, otherwise this will set the default active key for the endpoint.

scact_keynumber: This parameter is the shared key identifier which the application is requesting to become the active shared key to be used for sending authenticated chunks. The key identifier must correspond to an existing shared key. Note that shared key identifier '0' defaults to a null key.

When used with `setsockopt()` the SCTP implementation must use the indicated shared key identifier for all messages being given to an SCTP implementation via a `send` call after the `setsockopt()` call until changed again. Therefore, the SCTP implementation must not bundle user messages which should be authenticated using different shared key identifiers.

Initially the key with key identifier 0 is the active key.

8.1.19. Get or Set Delayed SACK Timer (`SCTP_DELAYED_SACK`)

This option will affect the way delayed sacks are performed. This option allows the application to get or set the delayed sack time, in milliseconds. It also allows changing the delayed sack frequency. Changing the frequency to 1 disables the delayed sack algorithm. Note that if `sack_delay` or `sack_freq` are 0 when setting this option, the current values will remain unchanged.

The following structure is used to access and modify these parameters:

```
struct sctp_sack_info {
    sctp_assoc_t sack_assoc_id;
    uint32_t sack_delay;
    uint32_t sack_freq;
};
```

sack_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets this parameter indicates which association the user is performing an action upon. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can also be used.

sack_delay: This parameter contains the number of milliseconds that the user is requesting the delayed SACK timer to be set to. Note that this value is defined in the standard to be between 200 and 500 milliseconds.

sack_freq: This parameter contains the number of packets that must be received before a sack is sent without waiting for the delay timer to expire. The default value is 2, setting this value to 1 will disable the delayed sack algorithm.

8.1.20. Get or Set Fragmented Interleave (`SCTP_FRAGMENT_INTERLEAVE`)

Fragmented interleave controls how the presentation of messages occurs for the message receiver. There are three levels of fragment interleave defined. Two of the levels affect the one-to-one model, while the one-to-many model is affected by all three levels.

This option takes an integer value. It can be set to a value of 0, 1 or 2. Attempting to set this level to other values will return an error.

Setting the three levels provides the following receiver interactions:

level 0: Prevents the interleaving of any messages. This means that when a partial delivery begins, no other messages will be received except the message being partially delivered. If another message arrives on a different stream (or association) that could be delivered, it will be blocked waiting for the user to read all of the partially delivered message.

level 1: Allows interleaving of messages that are from different associations. For the one-to-one model, level 0 and level 1 thus have the same meaning since a one-to-one socket always receives messages from the same association. Note that setting the one-to-many model to this level may cause multiple partial deliveries from different associations but for any given association, only one message will be delivered until all parts of a message have been delivered. This means that one large message, being read with an association identifier of "X", will block other messages from association "X" from being delivered.

level 2: Allows complete interleaving of messages. This level requires that the sender carefully observes not only the peer association identifier (or address) but must also pay careful attention to the stream number. With this option enabled a partially delivered message may begin being delivered for association "X" stream "Y" and the next subsequent receive may return a message from association "X" stream "Z". Note that no other messages would be delivered for association "X" stream "Y" until all of stream "Y"'s partially delivered message was read. Note that this option also affects the one-to-one model. Also note that for the one-to-many model not only another stream's message from the same association may be delivered upon the next receive, some other association's message may be delivered upon the next receive.

An implementation should default the one-to-many model to level 1. The reason for this is that otherwise it is possible that a peer could begin sending a partial message and thus block all other peers from sending data. However a setting of level 2 requires the application to not only be aware of the association (via the association identifier or peer's address) but also the stream number. The stream number is not present unless the user has subscribed to the `sctp_data_io_event` (see Section 6.2), which is deprecated, or has enabled the `SCTP_RECVRCVINFO` socket option (see Section 8.1.29). This is also why we recommend that the one-to-one model be defaulted to level 0 (level 1 for the one-to-one model has no effect). Note that an implementation should return an error if an application attempts to set the level to 2 and has not subscribed to the `sctp_data_io_event` event, which is deprecated, or has enabled the `SCTP_RECVRCVINFO` socket option.

For applications that have subscribed to events, those events appear in the normal socket buffer data stream. This means that unless the user has set the fragmentation interleave level to 0, notifications may also be interleaved with partially delivered messages.

8.1.21. Set or Get the SCTP Partial Delivery Point (`SCTP_PARTIAL_DELIVERY_POINT`)

This option will set or get the SCTP partial delivery point. This point is the size of a message where the partial delivery API will be invoked to help free up rwnd space for the peer. Setting this to a lower value will cause partial deliveries to happen more often. This option expects an integer that sets or gets the partial delivery point in bytes. Note also that the call will fail if the user attempts to set this value larger than the socket receive buffer size.

Note that any single message having a length smaller than or equal to the SCTP partial delivery point will be delivered in one single read call as long as the user provided buffer is large enough to hold the message.

8.1.22. Set or Get the Use of Extended Receive Info (SCTP_USE_EXT_RCVINFO) - DEPRECATED

This option will enable or disable the use of the extended version of the `sctp_sndrcvinfo` structure. If this option is disabled, then the normal `sctp_sndrcvinfo` structure is returned in all receive message calls. If this option is enabled then the `sctp_extrcvinfo` structure is returned in all receive message calls. The default is off.

Note that the `sctp_extrcvinfo` structure is never used in any send call.

This option is present for compatibility with older applications and is deprecated. Future applications should use `SCTP_NXTINFO` to retrieve this same information via ancillary data.

8.1.23. Set or Get the Auto ASCONF Flag (SCTP_AUTO_ASCONF)

This option will enable or disable the use of the automatic generation of ASCONF chunks to add and delete addresses to an existing association. Note that this option has two caveats namely: a) it only affects sockets that are bound to all addresses available to the SCTP stack, and b) the system administrator may have an overriding control that turns the ASCONF feature off no matter what setting the socket option may have.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.24. Set or Get the Maximum Burst (SCTP_MAX_BURST)

This option will allow a user to change the maximum burst of packets that can be emitted by this association. Note that the default value is 4, and some implementations may restrict this setting so that it can only be lowered to positive values.

To set or get this option the user fills in the following structure:

```
struct sctp_assoc_value {
    sctp_assoc_t assoc_id;
    uint32_t assoc_value;
};
```

`assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets this parameter indicates which association the user is performing an action upon. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can also be used.

`assoc_value`: This parameter contains the maximum burst. Setting the value to 0 disables burst mitigation.

8.1.25. Set or Get the Default Context (SCTP_CONTEXT)

The context field in the `sctp_sndrcvinfo` structure is normally only used when a failed message is retrieved holding the value that was sent down on the actual send call. This option allows the setting of a default context on an association basis that will be received on reading messages from the peer. This is especially helpful in the one-to-many model for an application to keep some reference to an internal state machine that is processing messages on the association. Note that the setting of this value only affects received messages from the peer and does not affect the value that is saved with outbound messages.

To set or get this option the user fills in the following structure:

```
struct sctp_assoc_value {
    sctp_assoc_t assoc_id;
    uint32_t assoc_value;
};
```

`assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets this parameter indicates which association the user is performing an action upon. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can also be used.

`assoc_value`: This parameter contains the context.

8.1.26. Enable or Disable Explicit EOR Marking (SCTP_EXPLICIT_EOR)

This boolean flag is used to enable or disable explicit end of record (EOR) marking. When this option is enabled, a user may make multiple send system calls to send a record and must indicate that they are finished sending a particular record by including the `SCTP_EOR` flag. If this boolean flag is disabled then each individual send system call is considered to have an `SCTP_EOR` indicator set on it implicitly without the user having to explicitly add this flag. The default is off.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.27. Enable SCTP Port Reusage (SCTP_REUSE_PORT)

This option only supports one-to-one style SCTP sockets. If used on a one-to-many style SCTP socket an error is indicated.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

This socket option must not be used after calling `bind()` or `sctp_bindx()` for a one-to-one style SCTP socket. If using `bind()` or `sctp_bindx()` on a socket with the `SCTP_REUSE_PORT` option, all other SCTP sockets bound to the same port must have set the `SCTP_REUSE_PORT`. Calling `bind()` or `sctp_bindx()` for a socket without having set the `SCTP_REUSE_PORT` option will fail if there are other sockets bound to the same port. At most one socket being bound to the same port may be listening.

It should be noted that the behavior of the socket level socket option to reuse ports and/or addresses for SCTP sockets is unspecified.

8.1.28. Set Notification Event (SCTP_EVENT)

This socket option is used to set a specific notification option. Please see Section 6.2.2 for a full description of this option and its usage.

8.1.29. Enable or Disable the Delivery of SCTP_RCVINFO as Ancillary Data (SCTP_RECVRCVINFO)

Setting this option specifies that `SCTP_RCVINFO` defined in Section 5.3.5 is returned as ancillary data by `recvmsg()`.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.30. Enable or Disable the Delivery of SCTP_NXTINFO as Ancillary Data (SCTP_RECVNXTINFO)

Setting this option specifies that `SCTP_NXTINFO` defined in Section 5.3.6 is returned as ancillary data by `recvmsg()`.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.31. Set Default Send Parameters (SCTP_DEFAULT_SNDINFO)

Applications that wish to use the `sendto()` system call may wish to specify a default set of parameters that would normally be supplied through the inclusion of ancillary data. This socket option allows such an application to set the default `sctp_sndinfo` structure. The application that wishes to use this socket option simply passes the `sctp_sndinfo` structure defined in Section 5.3.4 to this call. The input parameters accepted by this call include `snd_sid`, `snd_flags`, `snd_ppid`, `snd_context`. The `snd_flags` is composed of a bitwise OR of `SCTP_UNORDERED`, `SCTP_EOF`, and `SCTP_SENDALL`. The `snd_assoc_id` field specifies the association to apply the parameters to. For a one-to-many style socket any of the predefined constants are also allowed in this field. The field is ignored on the one-to-one style.

8.1.32. Set Default PR-SCTP Parameters (SCTP_DEFAULT_PRINFO)

This option sets and gets the default parameters for PR-SCTP. They can be overwritten by specific information provided in send calls.

The following structure is used to access and modify these parameters:

```
struct sctp_default_prinfo {
    uint16_t pr_policy;
    uint32_t pr_value;
    sctp_assoc_t pr_assoc_id;
};
```

`pr_policy`: Same as described in Section 5.3.7.

`pr_value`: Same as described in Section 5.3.7.

`pr_assoc_id`: This field is ignored for one-to-one style sockets. For one-to-many style sockets `pr_assoc_id` can be a particular association identifier or `SCTP_{FUTURE|CURRENT|ALL}_ASSOC`.

8.2. Read-Only Options

The options defined in this subsection are read-only. Using this option in a `setsockopt()` call will result in an error indicating `EOPNOTSUPP`.

8.2.1. Association Status (SCTP_STATUS)

Applications can retrieve current status information about an association, including association state, peer receiver window size, number of unacknowledged data chunks, and number of data chunks

pending receipt. This information is read-only.

The following structure is used to access this information:

```
struct sctp_status {
    sctp_assoc_t sstat_assoc_id;
    int32_t sstat_state;
    uint32_t sstat_rwnd;
    uint16_t sstat_unackdata;
    uint16_t sstat_penddata;
    uint16_t sstat_instrms;
    uint16_t sstat_outstrms;
    uint32_t sstat_fragmentation_point;
    struct sctp_paddrinfo sstat_primary;
};
```

sstat_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets it holds the identifier for the association. All notifications for a given association have the same association identifier. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC cannot be used.

sstat_state: This contains the association's current state, i.e. one of the following values:

- * SCTP_CLOSED
- * SCTP_BOUND
- * SCTP_LISTEN
- * SCTP_COOKIE_WAIT
- * SCTP_COOKIE_ECHOED
- * SCTP_ESTABLISHED
- * SCTP_SHUTDOWN_PENDING
- * SCTP_SHUTDOWN_SENT
- * SCTP_SHUTDOWN_RECEIVED
- * SCTP_SHUTDOWN_ACK_SENT

sstat_rwnd: This contains the association peer's current receiver window size.

sstat_unackdata: This is the number of unacknowledged data chunks.

sstat_penddata: This is the number of data chunks pending receipt.

sstat_instrms: The number of streams that the peer will be using outbound.

sstat_outstrms: The number of streams that the endpoint is allowed to use outbound.

sstat_fragmentation_point: The size at which SCTP fragmentation will occur.

sstat_primary: This is information on the current primary peer address.

To access these status values, the application calls `getsockopt()` with the option name `SCTP_STATUS`.

8.2.2. Peer Address Information (`SCTP_GET_PEER_ADDR_INFO`)

Applications can retrieve information about a specific peer address of an association, including its reachability state, congestion window, and retransmission timer values. This information is read-only.

The following structure is used to access this information:

```
struct sctp_paddrinfo {
    sctp_assoc_t spinfo_assoc_id;
    struct sockaddr_storage spinfo_address;
    int32_t spinfo_state;
    uint32_t spinfo_cwnd;
    uint32_t spinfo_srtt;
    uint32_t spinfo_rto;
    uint32_t spinfo_mtu;
};
```

`spinfo_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the following applies: This field may be filled by the application, if so, this field will have priority in looking up the association instead of using the address specified in `spinfo_address`. Note that if the address does not belong to the association specified then this call will fail. If the application does not fill in the `spinfo_assoc_id`,

then the address will be used to lookup the association and on return this field will have the valid association identifier. In other words, this call can be used to translate an address into an association identifier. Note that the predefined constants are not allowed on this option.

`spinfo_address`: This is filled by the application, and contains the peer address of interest.

`spinfo_state`: This contains the peer address' state:

`SCTP_UNCONFIRMED`: The initial state of a peer address.

`SCTP_ACTIVE`: The state is entered the first time after path verification. It can also be entered if the state is `SCTP_INACTIVE` and the path supervision detects that the peer address is reachable again.

`SCTP_INACTIVE`: This state is entered whenever a path failure is detected.

`spinfo_cwnd`: This contains the peer address' current congestion window.

`spinfo_srtt`: This contains the peer address' current smoothed round-trip time calculation in milliseconds.

`spinfo_rto`: This contains the peer address' current retransmission timeout value in milliseconds.

`spinfo_mtu`: The current path MTU of the peer address. It is the number of bytes available in an SCTP packet for chunks.

8.2.3. Get the List of Chunks the Peer Requires to be Authenticated (`SCTP_PEER_AUTH_CHUNKS`)

This option gets a list of chunk types (see [RFC4960]) for a specified association that the peer requires to be received authenticated only.

The following structure is used to access these parameters:

```
struct sctp_authchunks {
    sctp_assoc_t gauth_assoc_id;
    uint32_t gauth_number_of_chunks;
    uint8_t gauth_chunks[];
};
```

`gauth_assoc_id`: This parameter indicates for which association the user is requesting the list of peer authenticated chunks. For one-to-one sockets, this parameter is ignored. Note that the predefined constants are not allowed with this option.

`gauth_number_of_chunks`: This parameter gives the number of elements in the array `gauth_chunks`.

`gauth_chunks`: This parameter contains an array of chunk types that the peer is requesting to be authenticated. If the passed in buffer size is not large enough to hold the list of chunk types, `ENOBUFFS` is returned.

8.2.4. Get the List of Chunks the Local Endpoint Requires to be Authenticated (`SCTP_LOCAL_AUTH_CHUNKS`)

This option gets a list of chunk types (see [RFC4960]) for a specified association that the local endpoint requires to be received authenticated only.

The following structure is used to access these parameters:

```
struct sctp_authchunks {
    sctp_assoc_t gauth_assoc_id;
    uint32_t gauth_number_of_chunks;
    uint8_t gauth_chunks[];
};
```

`gauth_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the application may fill in an association identifier or `SCTP_FUTURE_ASSOC`. It is an error to use `SCTP_{CURRENT|ALL}_ASSOC` in `gauth_assoc_id`.

`gauth_number_of_chunks`: This parameter gives the number of elements in the array `gauth_chunks`.

`gauth_chunks`: This parameter contains an array of chunk types that the local endpoint is requesting to be authenticated. If the passed in buffer is not large enough to hold the list of chunk types, `ENOBUFFS` is returned.

8.2.5. Get the Current Number of Associations (`SCTP_GET_ASSOC_NUMBER`)

This option gets the current number of associations that are attached to a one-to-many style socket. The option value is an `uint32_t`. Note that this number is only a snap shot. This means that the number of associations may have changed when the caller gets back the option result.

For a one-to-one style socket, this socket option results in an error.

8.2.6. Get the Current Identifiers of Associations (SCTP_GET_ASSOC_ID_LIST)

This option gets the current list of SCTP association identifiers of the SCTP associations handled by a one-to-many style socket.

The option value has the structure

```
struct sctp_assoc_ids {
    uint32_t gaids_number_of_ids;
    sctp_assoc_t gaids_assoc_id[];
};
```

The caller must provide a large enough buffer to hold all association identifiers. If the buffer is too small, an error must be returned. The user can use the SCTP_GET_ASSOC_NUMBER socket option to get an idea how large the buffer has to be. `gaids_number_of_ids` gives the number of elements in the array `gaids_assoc_id`. Note also that some or all of `sctp_assoc_t` returned in the array may become invalid by the time the caller gets back the result.

For a one-to-one style socket, this socket option results in an error.

8.3. Write-Only Options

The options defined in this subsection are write-only. Using this option in a `getsockopt()` or `sctp_opt_info()` call will result in an error indicating EOPNOTSUPP.

8.3.1. Set Peer Primary Address (SCTP_SET_PEER_PRIMARY_ADDR)

Requests that the peer marks the enclosed address as the association primary (see [RFC5061]). The enclosed address must be one of the association's locally bound addresses.

The following structure is used to make a set peer primary request:

```
struct sctp_setpeerprim {
    sctp_assoc_t sspp_assoc_id;
    struct sockaddr_storage sspp_addr;
};
```

`sspp_assoc_id`: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets it identifies the association for this request. Note that the predefined constants are not allowed on this option.

`sspp_addr`: The address to set as primary.

8.3.2. Add a Chunk that must be Authenticated (SCTP_AUTH_CHUNK)

This set option adds a chunk type that the user is requesting to be received only in an authenticated way. Changes to the list of chunks will only affect future associations on the socket.

The following structure is used to add a chunk:

```
struct sctp_authchunk {
    uint8_t sauth_chunk;
};
```

`sauth_chunk`: This parameter contains a chunk type that the user is requesting to be authenticated.

The chunk types for INIT, INIT-ACK, SHUTDOWN-COMplete, and AUTH chunks must not be used. If they are used, an error must be returned. The usage of this option enables SCTP AUTH in cases where it is not required by other means (for example the use of dynamic address reconfiguration).

8.3.3. Set a Shared Key (SCTP_AUTH_KEY)

This option will set a shared secret key that is used to build an association shared key.

The following structure is used to access and modify these parameters:

```
struct sctp_authkey {
    sctp_assoc_t sca_assoc_id;
    uint16_t sca_keynumber;
    uint16_t sca_keylength;
    uint8_t sca_key[];
};
```

`sca_assoc_id`: This parameter indicates what association the shared key is being set upon. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can be used. For one-to-one sockets, this parameter is ignored. Note, however on one to one sockets, that this option will set a key on the association if the socket is connected,

otherwise this will set a key on the endpoint.

sca_keynumber: This parameter is the shared key identifier by which the application will refer to this shared key. If a key of the specified index already exists, then this new key will replace the old existing key. Note that shared key identifier '0' defaults to a null key.

sca_keylength: This parameter is the length of the array `sca_key`.

sca_key: This parameter contains an array of bytes that is to be used by the endpoint (or association) as the shared secret key. Note, if the length of this field is zero, a null key is set.

8.3.4. Deactivate a Shared Key (SCTP_AUTH_DEACTIVATE_KEY)

This set option indicates that the application will no longer send user messages using the indicated key identifier.

```
struct sctp_authkeyid {  
    sctp_assoc_t scact_assoc_id;  
    uint16_t scact_keynumber;  
};
```

scact_assoc_id: This parameter indicates which association the shared key identifier is being deleted from. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can be used. For one-to-one sockets, this parameter is ignored. Note, however, that this option will deactivate the key from the association if the socket is connected, otherwise this will deactivate the key from the endpoint.

scact_keynumber: This parameter is the shared key identifier which the application is requesting to be deactivated. The key identifier must correspond to an existing shared key. Note if this parameter is zero, use of the null key identifier '0' is deactivated on the endpoint and/or association.

The currently active key cannot be deactivated.

8.3.5. Delete a Shared Key (SCTP_AUTH_DELETE_KEY)

This set option will delete a shared secret key which has been deactivated of an SCTP association.

```
struct sctp_authkeyid {  
    sctp_assoc_t scact_assoc_id;  
    uint16_t scact_keynumber;  
};
```

```
};
```

`sact_assoc_id`: This parameter indicates which association the shared key identifier is being deleted from. The special `SCTP_{FUTURE|CURRENT|ALL}_ASSOC` can be used. For one-to-one sockets, this parameter is ignored. Note, however, that this option will delete the key from the association if the socket is connected, otherwise this will delete the key from the endpoint.

`sact_keynumber`: This parameter is the shared key identifier which the application is requesting to be deleted. The key identifier must correspond to an existing shared key and must not be in use for any packet being sent by the SCTP implementation. This means in particular, that it must be deactivated first. Note if this parameter is zero, use of the null key identifier '0' is deleted from the endpoint and/or association.

Only deactivated keys that are no longer used by an association can be deleted.

9. New Functions

Depending on the system, the following interface can be implemented as a system call or library function.

9.1. `sctp_bindx()`

This function allows the user to bind a specific subset of addresses or, if the SCTP extension described in [RFC5061] is supported, add or delete specific addresses.

The function prototype is

```
int sctp_bindx(int sd,
               struct sockaddr *addrs,
               int addrcnt,
               int flags);
```

If `sd` is an IPv4 socket, the addresses passed must be IPv4 addresses. If the `sd` is an IPv6 socket, the addresses passed can either be IPv4 or IPv6 addresses.

A single address may be specified as `INADDR_ANY` or `IN6ADDR_ANY`, see Section 3.1.2 for this usage.

`addrs` is a pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure. For an IPv6

socket, an array of `sockaddr_in6` is used. For a IPv4 socket, an array of `sockaddr_in` is used. The caller specifies the number of addresses in the array with `addrcnt`. Note that the wildcard addresses cannot be used in combination with non wildcard addresses on a socket with this function, doing so will result in an error.

On success, `sctp_bindx()` returns 0. On failure, `sctp_bindx()` returns -1 and sets `errno` to the appropriate error code.

For SCTP, the port given in each socket address must be the same, or `sctp_bindx()` will fail, setting `errno` to `EINVAL`.

The flags parameter is formed from the bitwise OR of zero or more of the following currently defined flags:

- o `SCTP_BINDX_ADD_ADDR`
- o `SCTP_BINDX_REM_ADDR`

`SCTP_BINDX_ADD_ADDR` directs SCTP to add the given addresses to the socket (i.e. endpoint), and `SCTP_BINDX_REM_ADDR` directs SCTP to remove the given addresses from the socket. The two flags are mutually exclusive; if both are given, `sctp_bindx()` will fail with `EINVAL`. A caller may not remove all addresses from a socket; `sctp_bindx()` will reject such an attempt with `EINVAL`.

An application can use `sctp_bindx(SCTP_BINDX_ADD_ADDR)` to associate additional addresses with an endpoint after calling `bind()`. Or use `sctp_bindx(SCTP_BINDX_REM_ADDR)` to remove some addresses a listening socket is associated with, so that no new association accepted will be associated with these addresses. If the endpoint supports dynamic address reconfiguration, an `SCTP_BINDX_REM_ADDR` or `SCTP_BINDX_ADD_ADDR` may cause an endpoint to send the appropriate message to its peers to change the peers' address lists.

Adding and removing addresses from established associations is an optional functionality. Implementations that do not support this functionality should return -1 and set `errno` to `EOPNOTSUPP`.

`sctp_bindx()` can be called on an already bound socket or on an unbound socket. If the socket is unbound and the first port number in the `addrs` is zero, the kernel will choose a port number. All port numbers after the first one being 0 must also be zero. If the first port number is not zero, the following port numbers must be zero or have the same value as the first one. For an already bound socket, all port numbers provided must be the bound one or 0.

`sctp_bindx()` is an atomic operation. Therefore, the binding will be

either successful on all addresses or fail on all addresses. If multiple addresses are provided and the `sctp_bindx()` call fails there is no indication which address is responsible for the failure. The only way to identify the specific error indication is to call `sctp_bindx()` sequentially with only one address per call.

9.2. `sctp_peeloff()`

After an association is established on a one-to-many style socket, the application may wish to branch off the association into a separate socket/file descriptor.

This is particularly desirable when, for instance, the application wishes to have a number of sporadic message senders/receivers remain under the original one-to-many style socket, but branch off these associations carrying high volume data traffic into their own separate socket descriptors.

The application uses the `sctp_peeloff()` call to branch off an association into a separate socket (Note the semantics are somewhat changed from the traditional one-to-one style `accept()` call). Note that the new socket is a one-to-one style socket. Thus it will be confined to operations allowed for a one-to-one style socket.

The function prototype is

```
int sctp_peeloff(int sd,
                 sctp_assoc_t assoc_id);
```

and the arguments are

`sd`: The original one-to-many style socket descriptor returned from the `socket()` system call (see Section 3.1.1).

`assoc_id`: the specified identifier of the association that is to be branched off to a separate file descriptor (Note, in a traditional one-to-one style `accept()` call, this would be an out parameter, but for the one-to-many style call, this is an in parameter).

The function returns a non-negative file descriptor representing the branched-off association, or -1 if an error occurred. The variable `errno` is then set appropriately.

9.3. `sctp_getpaddrs()`

`sctp_getpaddrs()` returns all peer addresses in an association.

The function prototype is:

```
int sctp_getpaddrs(int sd,
                   sctp_assoc_t id,
                   struct sockaddr **addrs);
```

On return, `addrs` will point to a dynamically allocated array of `sockaddr` structures of the appropriate type for the socket type. The caller should use `sctp_freepaddrs()` to free the memory. Note that the in/out parameter `addrs` must not be `NULL`.

If `sd` is an IPv4 socket, the addresses returned will be all IPv4 addresses. If `sd` is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses, with IPv4 addresses returned according to the `SCTP_I_WANT_MAPPED_V4_ADDR` option setting.

For one-to-many style sockets, `id` specifies the association to query. For one-to-one style sockets, `id` is ignored.

On success, `sctp_getpaddrs()` returns the number of peer addresses in the association. If there is no association on this socket, `sctp_getpaddrs()` returns 0, and the value of `*addrs` is undefined. If an error occurs, `sctp_getpaddrs()` returns -1, and the value of `*addrs` is undefined.

9.4. `sctp_freepaddrs()`

`sctp_freepaddrs()` frees all resources allocated by `sctp_getpaddrs()`.

The function prototype is

```
void sctp_freepaddrs(struct sockaddr *addrs);
```

and `addrs` is the array of peer addresses returned by `sctp_getpaddrs()`.

9.5. `sctp_getladdrs()`

`sctp_getladdrs()` returns all locally bound address(es) on a socket.

The function prototype is

```
int sctp_getladdrs(int sd,
                   sctp_assoc_t id,
                   struct sockaddr **addrs);
```

On return, `addrs` will point to a dynamically allocated array of `sockaddr` structures of the appropriate type for the socket type. The caller should use `sctp_freeladdrs()` to free the memory. Note that the in/out parameter `addrs` must not be `NULL`.

If `sd` is an IPv4 socket, the addresses returned will be all IPv4 addresses. If `sd` is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses, with IPv4 addresses returned according to the `SCTP_I_WANT_MAPPED_V4_ADDR` option setting.

For one-to-many style sockets, `id` specifies the association to query. For one-to-one style sockets, `id` is ignored.

If the `id` field is set to the value '0' then the locally bound addresses are returned without regard to any particular association.

On success, `sctp_getladdrs()` returns the number of local addresses bound to the socket. If the socket is unbound, `sctp_getladdrs()` returns 0, and the value of `*addrs` is undefined. If an error occurs, `sctp_getladdrs()` returns -1, and the value of `*addrs` is undefined.

9.6. `sctp_freeladdrs()`

`sctp_freeladdrs()` frees all resources allocated by `sctp_getladdrs()`.

The function prototype is

```
void sctp_freeladdrs(struct sockaddr *addrs);
```

and `addrs` is the array of local addresses returned by `sctp_getladdrs()`.

9.7. `sctp_sendmsg()` - DEPRECATED

This function is deprecated, `sctp_sendv()` (see Section 9.12) should be used instead.

An implementation may provide a library function (or possibly system call) to assist the user with the advanced features of SCTP.

The function prototype is

```
ssize_t sctp_sendmsg(int sd,
                    const void *msg,
                    size_t len,
                    const struct sockaddr *to,
                    socklen_t tolen,
                    uint32_t ppid,
                    uint32_t flags,
                    uint16_t stream_no,
                    uint32_t timetolive,
                    uint32_t context);
```

and the arguments are:

sd: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

to: The destination address of the message.

tolen: The length of the destination address.

ppid: The same as `sinfo_ppid` (see Section 5.3.2).

flags: The same as `sinfo_flags` (see Section 5.3.2).

stream_no: The same as `sinfo_stream` (see Section 5.3.2).

timetolive: The same as `sinfo_timetolive` (see Section 5.3.2).

context: The same as `sinfo_context` (see Section 5.3.2).

The call returns the number of characters sent, or -1 if an error occurred. The variable `errno` is then set appropriately.

Sending a message using `sctp_sendmsg()` is atomic (unless explicit EOR marking is enabled on the socket specified by `sd`).

Using `sctp_sendmsg()` on a non-connected one-to-one style socket for implicit connection setup may or may not work depending on the SCTP implementation.

9.8. `sctp_rcvmsg()` - DEPRECATED

This function is deprecated, `sctp_rcvv()` (see Section 9.13) should be used instead.

An implementation may provide a library function (or possibly system call) to assist the user with the advanced features of SCTP. Note that in order for the `sctp_sndrcvinfo` structure to be filled in by `sctp_rcvmsg()` the caller must enable the `sctp_data_io_event` with the `SCTP_EVENTS` option. Note that the setting of the `SCTP_USE_EXT_RCVINFO` will affect this function as well, causing the `sctp_sndrcvinfo` information to be extended.

The function prototype is

```
ssize_t sctp_recvmmsg(int sd,  
                      void *msg,  
                      size_t len,  
                      struct sockaddr *from,  
                      socklen_t *fromlen,  
                      struct sctp_sndrcvinfo *sinfo,  
                      int *msg_flags);
```

and the arguments are

sd: The socket descriptor.

msg: The message buffer to be filled.

len: The length of the message buffer.

from: A pointer to an address to be filled with the sender of this messages address.

fromlen: An in/out parameter describing the from length.

sinfo: A pointer to an sctp_sndrcvinfo structure to be filled upon receipt of the message.

msg_flags: A pointer to an integer to be filled with any message flags (e.g. MSG_NOTIFICATION). Note that this field is an in-out field. Options for the receive may also be passed into the value (e.g. MSG_PEEK). On return from the call, the msg_flags value will be different than what was sent in to the call. If implemented via a recvmmsg() call, the msg_flags should only contain the value of the flags from the recvmmsg() call.

The call returns the number of bytes received, or -1 if an error occurred. The variable errno is then set appropriately.

9.9. sctp_connectx()

An implementation may provide a library function (or possibly system call) to assist the user with associating to an endpoint that is multi-homed. Much like sctp_bindx() this call allows a caller to specify multiple addresses at which a peer can be reached. The way the SCTP stack uses the list of addresses to set up the association is implementation dependent. This function only specifies that the stack will try to make use of all the addresses in the list when needed.

Note that the list of addresses passed in is only used for setting up the association. It does not necessarily equal the set of addresses

the peer uses for the resulting association. If the caller wants to find out the set of peer addresses, it must use `sctp_getpaddrs()` to retrieve them after the association has been set up.

The function prototype is

```
int sctp_connectx(int sd,
                  struct sockaddr *addrs,
                  int addrcnt,
                  sctp_assoc_t *id);
```

and the arguments are:

`sd`: The socket descriptor.

`addrs`: An array of addresses.

`addrcnt`: The number of addresses in the array.

`id`: An output parameter that if passed in as a non-NULL will return the association identifier for the newly created association (if successful).

The call returns 0 on success or -1 if an error occurred. The variable `errno` is then set appropriately.

9.10. `sctp_send()` - DEPRECATED

This function is deprecated, `sctp_sendv()` should be used instead.

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the CMSG header structures.

The function prototype is

```
ssize_t sctp_send(int sd,
                  const void *msg,
                  size_t len,
                  const struct sctp_sndrcvinfo *sinfo,
                  int flags);
```

and the arguments are

`sd`: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

sinfo: A pointer to an sctp_sndrcvinfo structure used as described in Section 5.3.2 for a sendmsg() call.

flags: The same flags as used by the sendmsg() call flags (e.g. MSG_DONTROUTE).

The call returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

This function call may also be used to terminate an association using an association identifier by setting the sinfo.sinfo_flags to SCTP_EOF and the sinfo.sinfo_assoc_id to the association that needs to be terminated. In such a case the len of the message can be zero.

Using sctp_send() on a non-connected one-to-one style socket for implicit connection setup may or may not work depending on the SCTP implementation.

Sending a message using sctp_send() is atomic unless explicit EOR marking is enabled on the socket specified by sd.

9.11. sctp_sendx() - DEPRECATED

This function is deprecated, sctp_sendv() should be used instead.

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the CMSG header structures that also gives a list of addresses. The list of addresses is provided for implicit association setup. In such a case the list of addresses serves the same purpose as the addresses given in sctp_connectx() (see Section 9.9).

The function prototype is

```
ssize_t sctp_sendx(int sd,
                  const void *msg,
                  size_t len,
                  struct sockaddr *addrs,
                  int addrcnt,
                  struct sctp_sndrcvinfo *sinfo,
                  int flags);
```

and the arguments are:

sd: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

addrs: Is an array of addresses.

addrcnt: The number of addresses in the array.

sinfo: A pointer to an `sctp_sndrcvinfo` structure used as described in Section 5.3.2 for a `sendmsg()` call.

flags: The same flags as used by the `sendmsg()` call flags (e.g. `MSG_DONTROUTE`).

The call returns the number of bytes sent, or -1 if an error occurred. The variable `errno` is then set appropriately.

Note that in case of implicit connection setup, on return from this call the `sinfo_assoc_id` field of the `sinfo` structure will contain the new association identifier.

This function call may also be used to terminate an association using an association identifier by setting the `sinfo.sinfo_flags` to `SCTP_EOF` and the `sinfo.sinfo_assoc_id` to the association that needs to be terminated. In such a case the `len` of the message would be zero.

Sending a message using `sctp_sendx()` is atomic unless explicit EOR marking is enabled on the socket specified by `sd`.

Using `sctp_sendx()` on a non-connected one-to-one style socket for implicit connection setup may or may not work depending on the SCTP implementation.

9.12. `sctp_sendv()`

The function prototype is

```
ssize_t sctp_sendv(int sd,
                  const struct iovec *iov,
                  int iovcnt,
                  struct sockaddr *addrs,
                  int addrcnt,
                  void *info,
                  socklen_t infolen,
                  unsigned int infotype,
```

```
int flags);
```

The function `sctp_sendv()` provides an extensible way for an application to communicate different send attributes to the SCTP stack when sending a message. An implementation may provide `sctp_sendv()` as a library function or a system call.

This document defines three types of attributes which can be used to describe a message to be sent. They are `struct sctp_sndinfo` (Section 5.3.4), `struct sctp_prinfo` (Section 5.3.7), and `struct sctp_authinfo` (Section 5.3.8). The following structure `sctp_sendv_spa` is defined to be used when more than one of the above attributes are needed to describe a message to be sent.

```
struct sctp_sendv_spa {
    uint32_t sendv_flags;
    struct sctp_sndinfo sendv_sndinfo;
    struct sctp_prinfo sendv_prinfo;
    struct sctp_authinfo sendv_authinfo;
};
```

The `sendv_flags` field holds a bit wise OR of `SCTP_SEND_SNDINFO_VALID`, `SCTP_SEND_PRINFO_VALID` and `SCTP_SEND_AUTHINFO_VALID` indicating if the `sendv_sndinfo/sendv_prinfo/sendv_authinfo` fields contain valid information.

In future, when new send attributes are needed, new structures can be defined. But those new structures do not need to be based on any of the above defined structures.

The function takes the following arguments:

`sd`: The socket descriptor.

`iov`: The gather buffer. The data in the buffer is treated as one single user message.

`iovcnt`: The number of elements in `iov`.

`addrs`: An array of addresses to be used to set up an association or one single address to be used to send the message. Pass in `NULL` if the caller does not want to set up an association nor want to send the message to a specific address.

`addrcnt`: The number of addresses in the `addrs` array.

info: A pointer to the buffer containing the attribute associated with the message to be sent. The type is indicated by the info_type parameter.

infoflen: The length in bytes of info.

infotype: Identifies the type of the information provided in info. The current defined values are:

SCTP_SENDDV_NOINFO: No information is provided. The parameter info is a NULL pointer and infoflen is 0.

SCTP_SENDDV_SNDINFO: The parameter info is pointing to a struct sctp_sndinfo.

SCTP_SENDDV_PRINFO: The parameter info is pointing to a struct sctp_prinfo.

SCTP_SENDDV_AUTHINFO: The parameter info is pointing to a struct sctp_authinfo.

SCTP_SENDDV_SPA: The parameter info is pointing to a struct sctp_sendv_spa.

flags: The same flags as used by the sendmsg() call flags (e.g. MSG_DONTROUTE).

The call returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

A note on one-to-many style socket. The struct sctp_sndinfo attribute must always be used in order to specify the association the message is to be sent on. The only case where it is not needed is when this call is used to set up a new association.

The caller provides a list of addresses in the addrs parameter to set up an association. This function will behave like calling sctp_connectx() (see Section 9.9) first using the list of addresses and then calling sendmsg() with the given message and attributes. For an one-to-many style socket, if struct sctp_sndinfo attribute is provided, the snd_assoc_id field must be 0. When this function returns, the snd_assoc_id field will contain the association identifier of the newly established association. Note that struct sctp_sndinfo attribute is not required to set up an association for one-to-many style socket. If this attribute is not provided, the caller can enable the SCTP_ASSOC_CHANGE notification and use the SCTP_COMM_UP message to find out the association identifier.

If the caller wants to send the message to a specific peer address (hence overriding the primary address), it can provide the specific address in the `addrs` parameter and provide a `struct sctp_sndinfo` attribute with the field `snd_flags` set to `SCTP_ADDR_OVER`.

This function call may also be used to terminate an association. The caller provides an `sctp_sndinfo` attribute with the `snd_flags` set to `SCTP_EOF`. In this case the `len` of the message would be zero.

Sending a message using `sctp_sendv()` is atomic unless explicit EOR marking is enabled on the socket specified by `sd`.

9.13. `sctp_rcvv()`

The function prototype is

```
ssize_t sctp_rcvv(int sd,
                  const struct iovec *iov,
                  int iovlen,
                  struct sockaddr *from,
                  socklen_t *fromlen,
                  void *info,
                  socklen_t *infolen,
                  unsigned int *infotype,
                  int *flags);
```

The function `sctp_rcvv()` provides an extensible way for the SCTP stack to pass up different SCTP attributes associated with a received message to an application. An implementation may provide `sctp_rcvv()` as a library function or as a system call.

This document defines two types of attributes which can be returned by this call, the attribute of the received message and the attribute of the next message in receive buffer. The caller enables the `SCTP_RECVRCVINFO` and `SCTP_RECVNXTINFO` socket option to receive these attributes respectively. Attributes of the received message are returned in `struct sctp_rcvinfo` (Section 5.3.5) and attributes of the next message are returned in `struct sctp_nxtinfo` (Section 5.3.6). If both options are enabled, both attributes are returned using the following structure.

```
struct sctp_rcvv_rn {
    struct sctp_rcvinfo rcvv_rcvinfo;
    struct sctp_nxtinfo rcvv_nxtinfo;
};
```

In future, new structures can be defined to hold new types of attributes. The new structures do not need to be based on `struct`

sctp_rcvv_rn or struct sctp_rcvinfo.

This function takes the following arguments:

sd: The socket descriptor.

iov: The scatter buffer. Only one user message is returned in this buffer.

iovlen: The number of elements in iov.

from: A pointer to an address to be filled with the sender of the received message's address.

fromlen: An in/out parameter describing the from length.

info: A pointer to the buffer to hold the attributes of the received message. The structure type of info is determined by the info_type parameter.

infolen: An in/out parameter describing the size of the info buffer.

infotype: In return, *info_type is set to the type of the info buffer. The current defined values are:

SCTP_RECVV_NOINFO: If both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are not enabled, no attribute will be returned. If only the SCTP_RECVNXTINFO option is enabled but there is no next message in the buffer, there will also no attribute be returned. In these cases *info_type will be set to SCTP_RECVV_NOINFO.

SCTP_RECVV_RCVINFO: The type of info is struct sctp_rcvinfo and the attribute is about the received message.

SCTP_RECVV_NXTINFO: The type of info is struct sctp_nxtinfo and the attribute is about the next message in receive buffer. This is the case when only the SCTP_RECVNXTINFO option is enabled and there is a next message in buffer.

SCTP_RECVV_RN: The type of info is struct sctp_rcvv_rn. The rcvv_rcvinfo field is the attribute of the received message and the rcvv_nxtinfo field is the attribute of the next message in buffer. This is the case when both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are enabled and there is a next message in the receive buffer.

flags: A pointer to an integer to be filled with any message flags (e.g. MSG_NOTIFICATION). Note that this field is an in/out parameter. Options for the receive may also be passed into the value (e.g. MSG_PEEK). On return from the call, the flags value will be different than what was sent in to the call. If implemented via a `recvmsg()` call, the flags should only contain the value of the flags from the `recvmsg()` call when calling `sctp_recvv()` and on return it has the value from `msg_flags`.

The call returns the number of bytes received, or -1 if an error occurred. The variable `errno` is then set appropriately.

10. IANA Considerations

This document requires no actions from IANA.

11. Security Considerations

Many TCP and UDP implementations reserve port numbers below 1024 for privileged users. If the target platform supports privileged users, the SCTP implementation should restrict the ability to call `bind()` or `sctp_bindx()` on these port numbers to privileged users.

Similarly unprivileged users should not be able to set protocol parameters that could result in the congestion control algorithm being more aggressive than permitted on the public Internet. These parameters are:

- o `struct sctp_rtoinfo`

If an unprivileged user inherits a one-to-many style socket with open associations on a privileged port, it may be permitted to accept new associations, but it should not be permitted to open new associations. This could be relevant for the *r** family of protocols.

Applications using the one-to-many style sockets and using the interleave level if 0 are subject to denial of service attacks as described in Section 8.1.20.

Applications needing transport layer security can use DTLS/SCTP as specified in [RFC6083]. This can be implemented using the socket API described in this document.

12. Acknowledgments

Special acknowledgment is given to Ken Fujita, Jonathan Woods, Qiaobing Xie, and La Monte Yarroll, who helped extensively in the early formation of this document.

The authors also wish to thank Kavitha Baratakke, Mike Bartlett, Martin Becke, Jon Berger, Mark Butler, Thomas Dreibholz, Andreas Fink, Scott Kimble, Jonathan Leighton, Renee Revis, Irene Ruengeler, Dan Wing, and many others on the TSVWG mailing list for contributing valuable comments.

A special thanks to Phillip Conrad, for his suggested text, quick and constructive insights, and most of all his persistent fighting to keep the interface to SCTP usable for the application programmer.

13. References

13.1. Normative References

- [IEEE-1003.1-2008]
Institute of Electrical and Electronics Engineers,
"Information Technology - Portable Operating System
Interface (POSIX)", IEEE Standard 1003.1, 2008.
- [RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W.
Stevens, "Basic Socket Interface Extensions for IPv6",
RFC 3493, February 2003.
- [RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei,
"Advanced Sockets Application Program Interface (API) for
IPv6", RFC 3542, May 2003.
- [RFC3758] Stewart, R., Ramalho, M., Xie, Q., Tuexen, M., and P.
Conrad, "Stream Control Transmission Protocol (SCTP)
Partial Reliability Extension", RFC 3758, May 2004.
- [RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla,
"Authenticated Chunks for the Stream Control Transmission
Protocol (SCTP)", RFC 4895, August 2007.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol",
RFC 4960, September 2007.
- [RFC5061] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and M.
Kozuka, "Stream Control Transmission Protocol (SCTP)
Dynamic Address Reconfiguration", RFC 5061,

September 2007.

13.2. Informative References

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.
- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.
- [RFC1644] Braden, B., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.
- [RFC6083] Tuexen, M., Seggelmann, R., and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", RFC 6083, January 2011.

Appendix A. One-to-One Style Code Example

The following code is an implementation of a simple client which sends a number of messages marked for unordered delivery to an echo server making use of all outgoing streams. The example shows how to use some features of one-to-one style IPv4 SCTP sockets, including:

- o Creating and connecting an SCTP socket.
- o Requesting to negotiate a number of outgoing streams.
- o Determining the negotiated number of outgoing streams.
- o Setting an adaptation layer indication.
- o Sending messages with a given payload protocol identifier on a particular stream using `sctp_sendv()`.

/*

Copyright (c) 2011 IETF Trust and the persons identified as authors of the code. All rights reserved.

Redistribution and use in source and binary forms, with or without modification, is permitted pursuant to, and subject to the license terms contained in, the Simplified BSD License set forth in Section 4.c of the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>).


```
*/

#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define PORT 9
#define ADDR "127.0.0.1"
#define SIZE_OF_MESSAGE 1000
#define NUMBER_OF_MESSAGES 10
#define PPID 1234

int
main(void) {
    unsigned int i;
    int sd;
    struct sockaddr_in addr;
    char buffer[SIZE_OF_MESSAGE];
    struct iovec iov;
    struct sctp_status status;
    struct sctp_initmsg init;
    struct sctp_sndinfo info;
    struct sctp_setadaptation ind;
    socklen_t opt_len;

    /* Create a one-to-one style SCTP socket. */
    if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) < 0) {
        perror("socket");
        exit(1);
    }

    /* Prepare for requesting 2048 outgoing streams. */
    memset(&init, 0, sizeof(init));
    init.sinit_num_ostreams = 2048;
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_INITMSG,
                   &init, (socklen_t)sizeof(init)) < 0) {
        perror("setsockopt");
        exit(1);
    }

    ind.ssb_adaptation_ind = 0x01020304;
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_ADAPTATION_LAYER,
```

```
        &ind, (socklen_t)sizeof(ind)) < 0) {
    perror("setsockopt");
    exit(1);
}

/* Connect to the discard server. */
memset(&addr, 0, sizeof(addr));
#ifdef HAVE_SIN_LEN
    addr.sin_len      = sizeof(struct sockaddr_in);
#endif
addr.sin_family      = AF_INET;
addr.sin_port        = htons(PORT);
addr.sin_addr.s_addr = inet_addr(ADDR);
if (connect(sd,
            (const struct sockaddr *)&addr,
            sizeof(struct sockaddr_in)) < 0) {
    perror("connect");
    exit(1);
}

/* Get the actual number of outgoing streams. */
memset(&status, 0, sizeof(status));
opt_len = (socklen_t)sizeof(status);
if (getsockopt(sd, IPPROTO_SCTP, SCTP_STATUS,
            &status, &opt_len) < 0) {
    perror("getsockopt");
    exit(1);
}

memset(&info, 0, sizeof(info));
info.snd_ppid = htonl(PPID);
info.snd_flags = SCTP_UNORDERED;
memset(buffer, 'A', SIZE_OF_MESSAGE);
iov.iov_base = buffer;
iov.iov_len = SIZE_OF_MESSAGE;
for (i = 0; i < NUMBER_OF_MESSAGES; i++) {
    info.snd_sid = i % status.sstat_outstrms;
    if (sctp_sendv(sd,
                (const struct iovec *)&iov, 1,
                NULL, 0,
                &info, sizeof(info), SCTP_SENDV_SNDINFO,
                0) < 0) {
        perror("sctp_sendv");
        exit(1);
    }
}

if (close(sd) < 0) {
```

```
        perror("close");
        exit(1);
    }
    return(0);
}
```

Appendix B. One-to-Many Style Code Example

The following code is a simple implementation of a discard server over SCTP. The example shows how to use some features of one-to-many style IPv6 SCTP sockets, including:

- o Opening and binding of a socket.
- o Enabling notifications.
- o Handling notifications.
- o Configuring the auto close timer.
- o Using `sctp_rcvv()` to receive messages.

Please note that this server can be used in combination with the client described in Appendix A.

```
/*
```

```
    Copyright (c) 2011 IETF Trust and the persons identified
    as authors of the code. All rights reserved.
```

```
    Redistribution and use in source and binary forms, with
    or without modification, is permitted pursuant to, and subject
    to the license terms contained in, the Simplified BSD License
    set forth in Section 4.c of the IETF Trust's Legal Provisions
    Relating to IETF Documents (http://trustee.ietf.org/license-info).
```

```
*/
```

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/sctp.h>
#include <arpa/inet.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
#define BUFFER_SIZE (1<<16)
#define PORT 9
#define ADDR "0.0.0.0"
#define TIMEOUT 5

static void
print_notification(void *buf)
{
    struct sctp_assoc_change *sac;
    struct sctp_paddr_change *spc;
    struct sctp_adaptation_event *sad;
    union sctp_notification *snp;
    char addrbuf[INET6_ADDRSTRLEN];
    const char *ap;
    struct sockaddr_in *sin;
    struct sockaddr_in6 *sin6;

    snp = buf;

    switch (snp->sn_header.sn_type) {
    case SCTP_ASSOC_CHANGE:
        sac = &snp->sn_assoc_change;
        printf("^^^ Association change: ");
        switch (sac->sac_state) {
        case SCTP_COMM_UP:
            printf("Communication up (streams (in/out)=(%u/%u)).\n",
                   sac->sac_inbound_streams, sac->sac_outbound_streams);
            break;
        case SCTP_COMM_LOST:
            printf("Communication lost (error=%d).\n", sac->sac_error);
            break;
        case SCTP_RESTART:
            printf("Communication restarted (streams (in/out)=(%u/%u)).\n",
                   sac->sac_inbound_streams, sac->sac_outbound_streams);
            break;
        case SCTP_SHUTDOWN_COMP:
            printf("Communication completed.\n");
            break;
        case SCTP_CANT_STR_ASSOC:
            printf("Communication couldn't be started.\n");
            break;
        default:
            printf("Unknown state: %d.\n", sac->sac_state);
            break;
        }
        break;
    case SCTP_PEER_ADDR_CHANGE:
        spc = &snp->sn_paddr_change;
```

```

    if (spc->spc_aaddr.ss_family == AF_INET) {
        sin = (struct sockaddr_in *)&spc->spc_aaddr;
        ap = inet_ntop(AF_INET, &sin->sin_addr,
                        addrbuf, INET6_ADDRSTRLEN);
    } else {
        sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
        ap = inet_ntop(AF_INET6, &sin6->sin6_addr,
                        addrbuf, INET6_ADDRSTRLEN);
    }
    printf("^^^ Peer Address change: %s ", ap);
    switch (spc->spc_state) {
    case SCTP_ADDR_AVAILABLE:
        printf("is available.\n");
        break;
    case SCTP_ADDR_UNREACHABLE:
        printf("is not available (error=%d).\n", spc->spc_error);
        break;
    case SCTP_ADDR_REMOVED:
        printf("was removed.\n");
        break;
    case SCTP_ADDR_ADDED:
        printf("was added.\n");
        break;
    case SCTP_ADDR_MADE_PRIM:
        printf("is primary.\n");
        break;
    default:
        printf("unknown state (%d).\n", spc->spc_state);
        break;
    }
    break;
case SCTP_SHUTDOWN_EVENT:
    printf("^^^ Shutdown received.\n");
    break;
case SCTP_ADAPTATION_INDICATION:
    sad = &snp->sn_adaptation_event;
    printf("^^^ Adaptation indication 0x%08x received.\n",
          sad->sai_adaptation_ind);
    break;
default:
    printf("^^^ Unknown event of type: %u.\n",
          snp->sn_header.sn_type);
    break;
};
}

int
main(void) {

```

```
int sd, flags, timeout, on;
ssize_t n;
unsigned int i;
union {
    struct sockaddr sa;
    struct sockaddr_in sin;
    struct sockaddr_in6 sin6;
} addr;
socklen_t fromlen, infolen;
struct sctp_rcvinfo info;
unsigned int infotype;
struct iovec iov;
char buffer[BUFFER_SIZE];
struct sctp_event event;
uint16_t event_types[] = {SCTP_ASSOC_CHANGE,
                           SCTP_PEER_ADDR_CHANGE,
                           SCTP_SHUTDOWN_EVENT,
                           SCTP_ADAPTATION_INDICATION};

/* Create a 1-to-many style SCTP socket. */
if ((sd = socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0) {
    perror("socket");
    exit(1);
}

/* Enable the events of interest. */
memset(&event, 0, sizeof(event));
event.se_assoc_id = SCTP_FUTURE_ASSOC;
event.se_on = 1;
for (i = 0; i < sizeof(event_types)/sizeof(uint16_t); i++) {
    event.se_type = event_types[i];
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_EVENT,
                  &event, sizeof(event)) < 0) {
        perror("setsockopt");
        exit(1);
    }
}

/* Configure auto-close timer. */
timeout = TIMEOUT;
if (setsockopt(sd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
              &timeout, sizeof(timeout)) < 0) {
    perror("setsockopt SCTP_AUTOCLOSE");
    exit(1);
}

/* Enable delivery of SCTP_RCVINFO. */
on = 1;
```

```
if (setsockopt(sd, IPPROTO_SCTP, SCTP_RECVRCVINFO,
               &on, sizeof(on)) < 0) {
    perror("setsockopt SCTP_RECVRCVINFO");
    exit(1);
}

/* Bind the socket to all local addresses. */
memset(&addr, 0, sizeof(addr));
#ifdef HAVE_SIN6_LEN
    addr.sin6.sin6_len = sizeof(addr.sin6);
#endif
addr.sin6.sin6_family = AF_INET6;
addr.sin6.sin6_port = htons(PORT);
addr.sin6.sin6_addr = in6addr_any;
if (bind(sd, &addr.sa, sizeof(addr.sin6)) < 0) {
    perror("bind");
    exit(1);
}
/* Enable accepting associations. */
if (listen(sd, 1) < 0) {
    perror("listen");
    exit(1);
}

for (;;) {
    flags = 0;
    memset(&addr, 0, sizeof(addr));
    fromlen = (socklen_t)sizeof(addr);
    memset(&info, 0, sizeof(info));
    infolen = (socklen_t)sizeof(info);
    infotype = 0;
    iov.iov_base = buffer;
    iov.iov_len = BUFFER_SIZE;

    n = sctp_rcvv(sd, &iov, 1,
                  &addr.sa, &fromlen,
                  &info, &infolen, &infotype,
                  &flags);

    if (flags & MSG_NOTIFICATION) {
        print_notification(iov.iov_base);
    } else {
        char addrbuf[INET6_ADDRSTRLEN];
        const char *ap;
        in_port_t port;

        if (addr.sa.sa_family == AF_INET) {
            ap = inet_ntop(AF_INET, &addr.sin.sin_addr,
```

```
        addrbuf, INET6_ADDRSTRLEN);
    port = ntohs(addr.sin.sin_port);
} else {
    ap = inet_ntop(AF_INET6, &addr.sin6.sin6_addr,
        addrbuf, INET6_ADDRSTRLEN);
    port = ntohs(addr.sin6.sin6_port);
}
printf("Message received from %s:%u: len=%d",
    ap, port, (int)n);
switch (infotype) {
case SCTP_RECVV_RCVINFO:
    printf(", sid=%u", info.rcv_sid);
    if (info.rcv_flags & SCTP_UNORDERED) {
        printf(", unordered");
    } else {
        printf(", ssn=%u", info.rcv_ssn);
    }
    printf(", tsu=%u", info.rcv_tsn);
    printf(", ppid=%u.\n", ntohl(info.rcv_ppid));
    break;
case SCTP_RECVV_NOINFO:
case SCTP_RECVV_NXTINFO:
case SCTP_RECVV_RN:
    printf(".\n");
    break;
default:
    printf(" unknown infotype.\n");
}
}
}

if (close(sd) < 0) {
    perror("close");
    exit(1);
}

return (0);
}
```


Authors' Addresses

Randall R. Stewart
Adara Networks
Chapin, SC 29036
USA

Email: randall@lakerest.net

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstr. 39
48565 Steinfurt
Germany

Email: tuexen@fh-muenster.de

Kacheong Poon
Oracle Corporation

Email: ka-cheong.poon@oracle.com

Peter Lei
Cisco Systems, Inc.
9501 Technology Blvd
West Office Center
Rosemont, IL 60018
USA

Email: peterlei@cisco.com

Vladislav Yasevich
HP
110 Spitbrook Rd
Nashua, NH 03062
USA

Email: vladislav.yasevich@hp.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 13, 2012

Y. Nishida
WIDE Project
P. Natarajan
Cisco Systems
A. Caro
BBN Technologies
March 12, 2012

Quick Failover Algorithm in SCTP
draft-nishida-tsvwg-sctp-failover-05

Abstract

One of the major advantages in SCTP is supporting multi-homing communication. If a multi-homed end-point has redundant network connections, SCTP sessions can have a good chance to survive from network failures by migrating inactive network to active one. However, if we follow the SCTP standard, there can be significant delay for the network migration. During this migration period, SCTP cannot transmit much data to the destination. This issue drastically impairs the usability of SCTP in some situations. This memo describes the issue of SCTP failover mechanism and discuss its solutions which require minimal modification to the current standard.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on September 13, 2012.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal

Provisions Relating to IETF Documents
(<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Conventions and Terminology	4
3. Issue in SCTP Path Management Process	5
4. Existing Solutions for Smooth Failover	6
4.1. Reduce Path.Max.Retrans	6
4.2. Adjust RTO related parameters	7
5. Proposed Solution: SCTP with Potentially-Failed Destination State (SCTP-PF)	8
5.1. SCTP-PF Description	8
5.2. Effect of Path Bouncing	10
5.3. Permanent Failover	10
5.4. Handling Error Counter	10
6. Socket API Considerations	12
6.1. Peer Address Thresholds (SCTP_PEER_ADDR_THLDS) socket option	12
7. Security Considerations	13
8. IANA Considerations	14
9. References	15
9.1. Normative References	15
9.2. Informative References	15
Authors' Addresses	17

1. Introduction

The Stream Control Transmission Protocol (SCTP) [RFC4960] natively supports multihoming at the transport layer -- an SCTP association can bind to multiple IP addresses at each endpoint. SCTP's multihoming features include failure detection and failover procedures to provide network interface redundancy and improved end-to-end fault tolerance.

In SCTP's current failure detection procedure, the sender must experience Path.Max.Retrans (PMR) number of consecutive timeouts on a destination before detecting path failure. The sender fails over to an alternate active destination only after failure detection. Until failover, the sender transmits data on the failed path, degrading SCTP performance. Concurrent Multipath Transfer (CMT) [IYENGAR06] is an extension to SCTP and allows the sender to transmit data on multiple paths simultaneously. Research [NATARAJAN09] shows that the current failure detection procedure worsens CMT performance during failover and can be significantly improved by employing a better failover algorithm.

This document proposes an alternative failure detection procedure for SCTP (and CMT) that improves SCTP (CMT) performance during failover.

2. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. Issue in SCTP Path Management Process

SCTP can utilize multiple IP addresses for a single SCTP association. Each SCTP endpoint exchanges the list of available addresses on the node during initial negotiation. After this, endpoints select one address from the list and define this as the primary destination. During normal transmission, SCTP sends all data to the primary destination. Also, it sends heartbeat packets to other (non-primary) destinations at a certain interval to check the reachability of the path.

If sender has multiple active destination addresses, it can retransmit data to secondary destination address when the transmission to the primary times out.

When sender receives the acknowledgment for data or heartbeat packets from one of the destination addresses, it considers the destination is active. If it fails to receive acknowledgments, the error count for the address is increased. If the error counter exceeds the protocol parameter 'Path.Max.Retrans', SCTP endpoint considers the address is inactive.

The failover process of SCTP is initiated when the primary path becomes inactive (error counter for the primary path exceeds Path.Max.Retrans). If the primary path is marked inactive, SCTP chooses new destination address from one of the active destinations and start using this address to send data. If the primary path becomes active again, SCTP uses the primary destination for subsequent data transmissions and stop using non-primary one.

An issue in this failover process is that it usually takes significant amount of time before SCTP switches to the new destination. Let's say the primary path on a multi-homed host becomes unavailable and the RTO value for the primary path at that time is around 1 second, it usually takes over 60 seconds before SCTP starts to use the secondary path. This is because the recommended value for Path.Max.Retrans in the standard is 5, which requires 6 consecutive timeouts before failover takes place. Before SCTP switches to the secondary address, SCTP keeps trying to send packets to the primary and only retransmitted packets are sent to the secondary can be reached at the receiver. This slow failover process can cause significant performance degradation and will not be acceptable in some situations.

4. Existing Solutions for Smooth Failover

The following approach are conceivable for the solutions of this issue.

4.1. Reduce Path.Max.Retrans

If we choose smaller value for Path.Max.Retrans, we can shorten the duration of failover process. In fact, this is recommended in some research results [JUNGMAIER02] [GRINNEMO04] [FALLON08]. For example, if we set Path.Max.Retrans to 0, SCTP switches to another destination on a single timeout. However, smaller value for Path.Max.Retrans might cause spurious failover. In addition, if we use smaller value for Path.Max.Retrans, we may also need to choose smaller value for 'Association.Max.Retrans'. The Association.Max.Retrans indicates the threshold for the total number of consecutive error count for the entire SCTP association. If the total of the error count for all paths exceeds this value, the endpoint considers the peer endpoint unreachable and terminates the association. According to the Section 8.2 in [RFC4960], we should avoid having the value of Association.Max.Retrans larger than the summation of the Path.Max.Retrans of all the destination addresses. Otherwise, even if all the destination addresses become inactive, the endpoint still considers the peer endpoint reachable. The behavior in this situation is not defined in the RFC and depends on each implementation. In order to avoid inconsistent behavior between implementations, we had better use smaller value for Association.Max.Retrans. However, if we choose smaller value for Association.Max.Retrans, associations will prone to be terminated with minor congestion.

Another issue is that the interval of heartbeat packet: 'HB.interval' may not be small. (recommended value is 30 seconds) This means once failover takes place, an endpoint might need a certain amount of time to use the primary path again. This can cause undesirable effects in case of spurious failover. If we choose smaller value for HB.interval, the traffic used for path probing in a session will be increased.

The advantage of tuning Path.Max.Retrans is that it requires no modification to the current standard, although it needs to ignore several recommendations. In addition, some research results indicate path bouncing caused by spurious failover does not cause serious problems. We discuss the effect of path bouncing in the section 5.

4.2. Adjust RTO related parameters

As several research results indicate, we can also shorten the duration of failover process by adjusting RTO related parameters [JUNGMAIER02] [FALLON08]. During failover process. RTO keeps being doubled. However, if we can choose smaller value for RTO.max, we can stop the exponential growth of RTO at some point. Also, choosing smaller values for RTO.initial or RTO.min can contribute to keep RTO value small.

Similar to reducing Path.Max.Retrans, the advantage of this approach is that it requires no modification to the current standard, although it needs to ignore several recommendations. However, this approach requires to have enough knowledge about the network characteristics between end points. Otherwise, it can introduce adverse side-effects such as spurious timeouts.

5. Proposed Solution: SCTP with Potentially-Failed Destination State (SCTP-PF)

5.1. SCTP-PF Description

Our proposal stems from the following two observations about SCTP's failure detection procedure:

- o In order to minimize performance impact during failover, the sender should avoid transmitting data to the failed destination as early as possible. In the current SCTP path management scheme, the sender stops transmitting data to a destination only after the destination is marked Failed. Thus, a smaller PMR value is ideal so that the sender transitions a destination to the Failed state quicker.
- o Smaller PMR values increase the chances of spurious failure detection where the sender incorrectly marks a destination as Failed during periods of temporary congestion. Larger PMR values are preferable to avoid spurious failure detection.

From the above observations it is clear that tweaking the PMR value involves the following tradeoff -- a lower value improves performance but increases the chances of spurious failure detection, whereas a higher value degrades performance and reduces spurious failure detection in a wide range of path conditions. Thus, tweaking the association's PMR value is an incomplete solution to address performance impact during failure.

We propose a new "Potentially-failed" (PF) destination state in SCTP's path management procedure. The PF state was originally proposed to improve CMT performance [NATARAJAN09]. The PF state is an intermediate state between Active and Failed states. SCTP's failure detection procedure is modified to include the PF state. The new failure detection algorithm assumes that loss detected by a timeout implies either severe congestion or failure en-route. After a single timeout on a path, a sender is unsure, and marks the corresponding destination as PF. A PF destination is not used for data transmission except in special cases (discussed below). The new failure detection algorithm requires only sender-side changes. Details are:

1. The sender maintains a new tunable parameter called Potentially-failed.Max.Retrans (PFMR). The recommended value of PFMR = 0 when quick failover is used. When an association's PFMR \geq PMR, quick failover is turned off.

2. Each time the T3-rtx timer expires on an active or idle destination, the error counter of that destination address will be incremented. When the value in the error counter exceeds PFMR, the endpoint should mark the destination transport address as PF. SCTP MUST NOT send any notification to the upper layer about the active to PF state transition.
3. The sender SHOULD avoid data transmission to PF destinations. When all destinations are in either PF or Inactive state, the sender MAY either move the destination from PF to active state (and transmit data to the active destination) or the sender MAY transmit data to a PF destination. In the former scenario, (i) the sender MUST NOT notify the ULP about the state transition, and (ii) MUST NOT clear the destination's error counter. It is recommended that the sender picks the PF destination with least error count (fewest consecutive timeouts) for data transmission. In case of a tie (multiple PF destinations with same error count), the sender MAY choose the last active destination.
4. Only heartbeats MUST be sent to PF destination(s) once per RTO. This means the sender SHOULD ignore HB.interval for PF destinations. If an heartbeat is unanswered, the sender increments the error counter and exponentially backs off the RTO value. If error counter is less than PMR, the sender SHOULD transmit another heartbeat immediately after T3-timer expiration.
5. When the sender receives an heartbeat ACK from a PF destination, the sender clears the destination's error counter and transitions the PF destination back to active state. This state transition MUST NOT be notified to the ULP. This destination's cwnd is set to 1 MTU (TODO: or 2? Needs more text discussing rationale; can revisit later?)
6. An additional (PMR - PFMR) consecutive timeouts on a PF destination confirm the path failure, upon which the destination transitions to the Inactive state. As described in [RFC4960], the sender (i) SHOULD notify ULP about this state transition, and (ii) transmit heartbeats to the Inactive destination at a lower frequency as described in Section 8.3 of [RFC4960].
7. When all destinations are in the Inactive state, the sender picks one of the Inactive destinations for data transmission. This proposal recommends that the sender picks the Inactive destination with least error count (fewest consecutive timeouts) for data transmission. In case of a tie (multiple Inactive destinations with same error count), the sender MAY choose the last active destination.

8. ACKs for retransmissions do not transition a PF destination back to the active state, since a sender cannot disambiguate whether the ack was for the original transmission or the retransmission(s).

5.2. Effect of Path Bouncing

The methods described above can accelerate failover process. Hence, it might introduce path bouncing effect which keeps changing the data transmission path frequently. This sounds harmful for data transfer, however several research results indicate that there is no serious problem with SCTP in terms of path bouncing effect [CAR004] [CAR005].

There are two main reasons for this. First, SCTP is basically designed for multipath communication, which means SCTP maintains all path related parameters (cwnd, ssthresh, RTT, error count, etc) per each destination address. These parameters cannot be affected by path bouncing. In addition, when SCTP migrates to another path, it starts with minimal cwnd because of slow-start. Hence, there is little chance for packet reordering or duplicating.

Second, even if all communication paths between end-nodes share the same bottleneck, the proposed method does not make situations worse. In case of congestion, the current standard tries to transmit data packets to the primary during failover, while the proposed method tries to explore other destinations. In any case, the same amount of data packets sent to the same bottleneck.

5.3. Permanent Failover

When primary path becomes active again after failover, SCTP migrates back to the primary path. After this, SCTP starts data transfer with minimal cwnd. This is because SCTP must perform slow-start when it migrates to new path. However, this might degrade the communication performance in case that the performance of the alternative path is relatively good. In order to mitigate this effect of slow-start, permanent failover was proposed in [CAR002]. Permanent failover allows SCTP to remain the alternative path even if the primary path becomes active again. This approach can improve performance in some cases, however, it will require more detail analysis since it might impact on SCTP failover algorithm. Since we prefer to keep the current behavior of the standard as possible, we recommend not to take this approach for now.

5.4. Handling Error Counter

When multiple destinations are in the PF state, the sender may transmit heartbeats to multiple destinations at the same time. This

allows sender to quickly track and respond to network status change. However, when all PF destinations become unavailable, this approach increases the total number of consecutive retransmissions rather aggressively than the current SCTP spec does. Because of this aggressive increase, an SCTP association may be terminated rather earlier than the standard [RFC4960].

One way to avoid early termination is to send retransmitted data or HB to only one PF destination at a time, but this approach may delay path status tracking. An alternative solution is to exclude HB timeouts from incrementing the error count. The latter approach is preferred but requires an update to Section 8.3 of [RFC4960].

6. Socket API Considerations

This section describes how the socket API defined in [I-D.ietf-tsvwg-sctpsocket] is extended to provide a way for the application to control the quick failover behavior.

Please note that this section is informational only.

A socket API implementation based on [I-D.ietf-tsvwg-sctpsocket] is extended by adding a new read/write socket option for the level IPPROTO_SCTP and the name SCTP_PEER_ADDR_THLDS as described below. This socket option is used to read/write the value of PFMR parameter described in Section 5.

Support for the SCTP_PEER_ADDR_THLDS socket option needs also to be added to the function sctp_opt_info().

6.1. Peer Address Thresholds (SCTP_PEER_ADDR_THLDS) socket option

Applications can control the quick failover behavior by getting or setting the number of timeouts before a peer address is considered potentially failed or unreachable.

The following structure is used to access and modify the thresholds:

```
struct sctp_paddrthlds {  
    sctp_assoc_t spt_assoc_id;  
    struct sockaddr_storage spt_address;  
    uint16_t spt_pathmaxrxt;  
    uint16_t spt_pathpfthld;  
};
```

spt_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets the application may fill in an association identifier or SCTP_FUTURE_ASSOC for this query. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in spt_assoc_id.

spt_address: This specifies which peer address is of interest. If a wildcard address is provided, this socket option applies to all current and future peer addresses.

spt_pathmaxrxt: Each peer address of interest is considered unreachable, if its path error counter exceeds spt_pathmaxrxt.

spt_pathpfthld: Each peer address of interest is considered potentially failed, if its path error counter exceeds spt_pathpfthld.

7. Security Considerations

There are no new security considerations introduced in this document.

8. IANA Considerations

This document does not create any new registries or modify the rules for any existing registries managed by IANA.

9. References

9.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

9.2. Informative References

- [CARO02] Caro Jr., A., Iyengar, J., Amer, P., Heinz, G., and R. Stewart, "A Two-level Threshold Recovery Mechanism for SCTP", Tech report, CIS Dept, University of Delaware , 7 2002.
- [CARO04] Caro Jr., A., Amer, P., and R. Stewart, "End-to-End Failover Thresholds for Transport Layer Multihoming", MILCOM 2004 , 11 2004.
- [CARO05] Caro Jr., A., "End-to-End Fault Tolerance using Transport Layer Multihoming", Ph.D Thesis, University of Delaware , 1 2005.
- [FALLON08] Fallon, S., Jacob, P., Qiao, Y., Murphy, L., Fallon, E., and A. Hanley, "SCTP Switchover Performance Issues in WLAN Environments", IEEE CCNC 2008, 1 2008.
- [GRINNEMO04] Grinnemo, K-J. and A. Brunstrom, "Performance of SCTP-controlled failovers in M3UA-based SIGTRAN networks", Advanced Simulation Technologies Conference , 4 2004.
- [I-D.ietf-tsvwg-sctpsocket] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)", draft-ietf-tsvwg-sctpsocket-31 (work in progress), August 2011.
- [IYENGAR06] Iyengar, J., Amer, P., and R. Stewart, "Concurrent Multipath Transfer using SCTP Multihoming over Independent End-to-end Paths.", IEEE/ACM Trans on Networking 14(5), 10 2006.

[JUNGMAIER02]

Jungmaier, A., Rathgeb, E., and M. Tuexen, "On the use of SCTP in failover scenarios", World Multiconference on Systemics, Cybernetics and Informatics , 7 2002.

[NATARAJAN09]

Natarajan, P., Ekiz, N., Amer, P., and R. Stewart, "Concurrent Multipath Transfer during Path Failure", Computer Communications , 5 2009.

Authors' Addresses

Yoshifumi Nishida
WIDE Project
Endo 5322
Fujisawa, Kanagawa 252-8520
Japan

Email: nishida@wide.ad.jp

Preethi Natarajan
Cisco Systems
510 McCarthy Blvd
Milpitas, CA 95035
USA

Email: prenatar@cisco.com

Armando Caro
BBN Technologies
10 Moulton St.
Cambridge, MA 02138
USA

Email: acar@bbn.com

TSVWG WG
Internet-Draft

Expires: September 14, 2011

Intended Status: Standards Track (PS)

Updates: RFC 2205, 2210, & 4495 (if published as an RFC)

James Polk
Subha Dhesikan
Cisco Systems
March 14, 2011

Integrated Services (IntServ) Extension to Allow Signaling of Multiple
Traffic Specifications and Multiple Flow Specifications in RSVPv1
draft-polk-tsvwg-intserv-multiple-tspec-06

Abstract

This document defines extensions to Integrated Services (IntServ) allowing multiple traffic specifications and multiple flow specifications to be conveyed in the same Resource Reservation Protocol (RSVPv1) reservation message exchange. This ability helps optimize an agreeable bandwidth through a network between endpoints in a single round trip.

Legal

This documents and the information contained therein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION THEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Status of this Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at
<http://www.ietf.org/ietf/lid-abstracts.txt>.

The list of Internet-Draft Shadow Directories can be accessed at
<http://www.ietf.org/shadow.html>.

This Internet-Draft will expire on September 14, 2011.

Copyright Notice

Copyright (c) 2011 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the BSD License.

Table of Contents

1. Introduction	3
2. Overview of the Proposal for including multiple TSPECs and FLOWSPECs	6
3. Multi_TSPEC and MULTI_FLOWSPEC Solution	8
3.1 New MULTI_TSPEC and MULTI_RSPEC Parameters	9
3.2 Multiple TSPEC in a PATH Message	9
3.3 Multiple FLOWSPEC for Controlled Load Service	12
3.4 Multiple FLOWSPEC for Guaranteed Service	14
4. Rules of Usage	17
4.1 Backward Compatibility	17
4.2 Applies to Only a Single Session	17
4.3 No Special Error Handling for PATH Message	17
4.4 Preference Order to be Maintained	18
4.5 Bandwidth Reduction in Downstream Routers	18
4.6 Merging Rules	19
4.7 Applicability to Multicast	19
4.8 MULTI_TSPEC Specific Error	20
4.9 Other Considerations	20
4.10 Known Open Issues	21
5. Security considerations	21
6. IANA considerations	22
7. Acknowledgments	22
8. References	22
8.1. Normative References	23
8.2. Informative References	23
Authors' Addresses	23
Appendix A. Alternatives for Sending Multiple TSPECs.	23

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC 2119].

1. Introduction

This document defines how Integrated Services (IntServ) [RFC2210] includes multiple traffic specifications and multiple flow specifications in the same Resource Reservation Protocol (RSVPv1) [RFC2205] message. This ability helps optimize an agreeable bandwidth through a network between endpoints in a single round trip.

There is a separation of function between RSVP and IntServ, in which RSVP does not define the internal objects to establish controlled load or guarantee services. These are generally left to be opaque in RSVP. At the same time, IntServ does not require that RSVP be the only reservation protocol for transporting both the controlled load or guaranteed service objects - but RSVP does often carry the objects anyway. This makes the two independent - yet related in usage, but are also frequently talked about as if they are one and the same. They are not.

The 'traffic specification' contains the traffic characteristics of a sender's data flow and is a required object in a PATH message. The TSPEC object is defined in RFC 2210 to convey the traffic specification from the sender and is opaque to RSVP. The ADSPEC object - for 'advertising specification' - is used to gather information along the downstream data path to aid the receiver in the computation of QoS properties of this data path. The ADSPEC is also opaque to RSVP and is defined in RFC 2210. Both of these IntServ objects are part of the Sender Descriptor [RFC2205].

Once the Sender Descriptor is received at its destination node, after having traveled through the network of routers, the SENDER_TSPEC information is matched with the information gathered in the ADSPEC, if present, about the data path. Together, these two objects help the receiver build its flow specification (encoded in the FLOWSPEC object) for the RESV message. The RESV message establishes the reservation through the network of routers on the data path established by the PATH message. If the ADSPEC is not present in the Sender_Descriptor, it cannot aid the receiver in building the flow specification.

The SENDER_TSPEC is not changed in transit between endpoints (i.e., there are no bandwidth request adjustments along the way). However, the ADSPEC is changed, based on the conditions experienced through the network (i.e., bandwidth availability within each router) as the RSVP message travels hop-by-hop.

Today, real-time applications have evolved such that they are able to dynamically adapt to available bandwidth, not only by dropping and adding layers, but also by reducing frame rates and resolution. It is therefore limiting to have a single bandwidth request in Integrated Services, and by extension, RSVP.

With only one traffic specification in a PATH message and only one flow specification in a RESV message (with some styles of reservations a RESV message may actually contain multiple flow specifications, but then there is only one per sender), applications will either have to give up altogether on session establishment in case of failure of the reservation establishment for the highest "bandwidth or will have to resort to multiple successive RSVP signaling attempts in a trial-and-error manner until they finally establish the reservation a lower "bandwidth". These multiple signaling round-trip would affect the session establishment time and in turn would negatively impact the end user experience.

The objective of this document is to avoid such roundtrips as well as allow applications to successfully receive some level of bandwidth allotment that it can use for its sessions.

While the ADSPEC provides an indication of the bandwidth available along the path and can be used by the receiver in creating the FLOWSPEC, it does not prevent failures or multiple round-trips as described above. The intermediary routers provide a best attempt estimate of available bandwidth in the ADSPEC object. However, it does not take into account external policy considerations (RFC 2215). In addition, the available bandwidth at the time of creating the ADSPEC may not be available at the time of an actual request in an RESV message. These reasons may cause the RESV message to be rejected. Therefore, the ADSPEC object cannot, by itself, satisfy the requirements of the current generations of real-time applications.

It needs to be noted that the ADSPEC is unchanged by this new mechanism. If ADSPEC is included in the PATH message, it is suggested that the receiver use this object in determining the flow specification.

This document creates a means for conveying more than one "bandwidth" within the same RSVP reservation set-up (both PATH and RESV) messages to optimize the determination of an agreed upon bandwidth for this reservation. Allowing multiple traffic specifications within the same PATH message allows the sender to communicate to the receiver multiple "bandwidths" that match the different sending rates that the sender is capable of transmitting at. This allows the receiver to convey this multiple "bandwidths" in the RESV so those can be considered when RSVP makes the actual reservation admission into the network. This allows the applications to dynamically adapt their data stream to available network resources.

The concept of RSVP signaling is shown in a single direction below, in Figure 1. Although the TSPEC is opaque to RSVP, it is shown along with the RSVP messages for completeness. The RSVP messages themselves need not be the focus of the reader. Instead, the number of round trips it takes to establish a reservation is the

focus here.

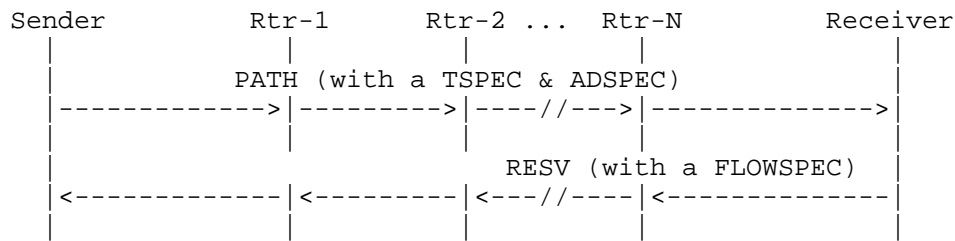


Figure 1. Concept of RSVP in a Single Direction

Figure 1 shows a successful one-way reservation using RSVP and IntServ.

Figure 2 shows a scenario where the RESV message, containing a FLOWSPEC, which is generated by the Receiver, after considering both the Sender TSPEC and the ADSPEC, is rejected by an intermediary router.

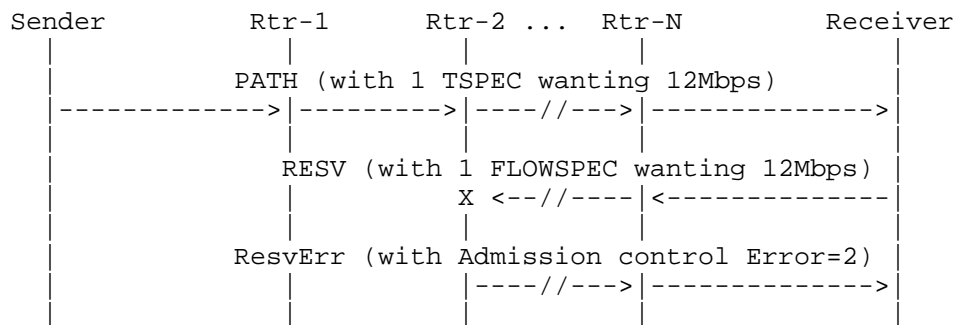


Figure 2. Concept of RSVP Rejection due to Limited Bandwidth

The scenario above is where multiple TSPEC and multiple FLOWSPEC optimization helps. The Sender may support multiple bandwidths for a given application (i.e., more than one codec for voice or video) and therefore might want to establish a reservation with the highest (or best) bandwidth that the network can provide for a particular codec.

For example, bandwidths of:

12Mbps,
4Mbps, and
1.5Mbps

for the three video codecs the Sender supports.

This document will discuss the overview of the proposal to include multiple TSPECs and FLOWSPECs RSVP in section 2. In section 3, the overview of the entire solution is provided. This section also contains the new parameters which are defined in this document. The multiple TSPECs in a PATH message and the multiple FLOWSPEC in a RESV message, both for controlled load and guaranteed service are described in this section. Section 4 will cover the rules of usage of this IntServ extension. This section contains how this document needs to extend the scenario of when a router in the middle of a reservation cannot accept a preferred bandwidth (i.e., FLOWSPEC), meaning previous routers that accepted that greater bandwidth now have too much bandwidth reserved. This requires an extension to RFC 4495 (RSVP Bandwidth Reduction) to cover reservations being established, as well as existing reservations. Section 4 also includes the merging rules.

2. Overview of Proposal for Including Multiple TSPECs and FLOWSPECs

Presently, this is the format of a PATH message [RFC2205]:

```

<PATH Message> ::= <Common Header> [ <INTEGRITY> ]

                                <SESSION> <RSVP_HOP>

                                <TIME_VALUES>

                                [ <POLICY_DATA> ... ]

                                [ <sender descriptor> ]

<sender descriptor> ::= <SENDER_TEMPLATE> <SENDER_TSPEC>
                                ^^^^^^^^^^^^^
                                [ <ADSPEC> ]

```

where the SENDER_TSPEC object contains a single traffic specification.

For the PATH message, the focus of this document is to modify the <sender_descriptor> in such a way to include more than one traffic specification. This solution does this by retaining the existing SENDER_TSPEC object above, highlighted by the '^^^^' characters, and complementing it with a new optional MULTI_TSPEC object to convey additional traffic specifications in this PATH message. No other object within the PATH message is affected by this IntServ extension.

This extension modifies the sender descriptor by specifically augmenting it to allow an optional <MULTI_TSPEC> object after the optional <ADSPEC>, as shown below.

```

<sender descriptor> ::= <SENDER_TEMPLATE> <SENDER_TSPEC>
                        [ <ADSPEC> ] [ <MULTI_TSPEC> ]
                                   ^^^^^^^^^^^^^

```

As can be seen above, the MULTI_TSPEC is in addition to the SENDER_TSPEC - and is only to be used, per this extension, when more than one TSPEC is to be included in the PATH message.

Here is another way of looking at the proposal choices:

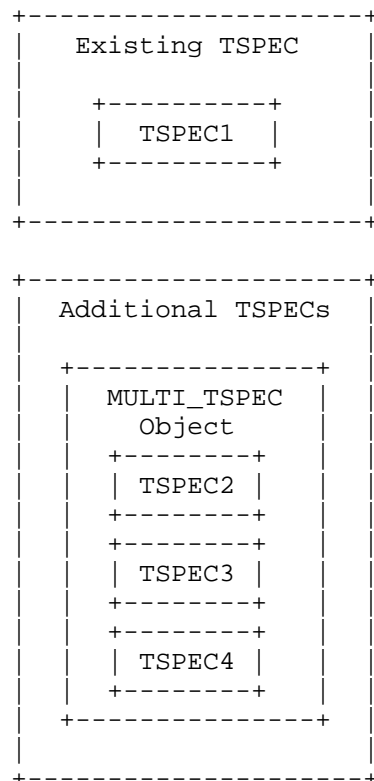


Figure 3. Encoding of Multiple Traffic Specifications in the TSPEC and MULTI_TSPEC objects

This solution is backwards compatible with existing implementations of [RFC2205] and [RFC2210], as the multiple TSPECs and FLOWSPECs are inserted as optional objects and such objects do not need to be processed, especially if they are not understood.

This solution defines a similar approach for encoding multiple flow specifications in the RESV message. Flow specifications beyond the first one can be encoded in a new "MULTI_FLOWSPEC" object contained

in the RESV message.

In this proposal, the original SENDER_TSPEC and the FLOWSPEC are left untouched, allowing routers not supporting this extension to process the PATH and the RESV message without issue. Two new additional objects are defined in this document. They are the MULTI_TSPEC and the MULTI_FLOWSPEC for the PATH and the RESV message, respectively. The additional TSPECs (in the new MULTI_TSPEC Object) are included in the PATH and the additional FLOWSPECs (in the new MULTI_FLOWSPEC Object) are included in the RESV message as new (optional) objects. These additional objects will have a class number of 11bbbbbb, allowing older routers to ignore the object(s) and forward each unexamined and unchanged, as defined in section 3.10 of [RFC 2205].

NOTE: it is important to emphasize here that including more than one FLOWSPEC in the RESV message does not cause more than one FLOWSPEC to be granted. This document requires that the receiver arrange these multiple FLOWSPECs in the order of preference according to the order remaining from the MULTI_TSPECs in the PATH message. The benefit of this arrangement is that RSVP does not have to process the rest of the FLOWSPEC if it can admit the first one.

3. Multi_TSPEC and MULTI_FLOWSPEC Solution

For the Sender Descriptor within the PATH message, the original TSPEC remains where it is, and is untouched by this IntServ extension. What is new is the use of a new <MULTI_TSPEC> object inside the sender descriptor as shown here:

```
<sender descriptor> ::= <SENDER_TEMPLATE> <SENDER_TSPEC>
                        [ <ADSPEC> ] [ <MULTI_TSPEC> ]
                                   ^^^^^^^^^^^^^^
```

The preferred order of TSPECs sent by the sender is this:

- preferred TSPEC is in the original SENDER_TSPEC
- the next in line preferred TSPEC is the first TSPEC in the MULTI_TSPEC object
- the next in line preferred TSPEC is the second TSPEC in the MULTI_TSPEC object
- and so on...

The composition of the flow descriptor list in a Resv message depends upon the reservation style. Therefore, the following shows

the inclusion of the MULTI_FLOWSPEC object with each of the styles:

WF Style:

<flow descriptor list> ::= <WF flow descriptor>

<WF flow descriptor> ::= <FLOWSPEC> [MULTI_FLOWSPEC]

FF style:

<flow descriptor list> ::=

<FLOWSPEC> <FILTER_SPEC> [MULTI_FLOWSPEC] |

<flow descriptor list> <FF flow descriptor>

<FF flow descriptor> ::=

[<FLOWSPEC>] <FILTER_SPEC> [MULTI_FLOWSPEC]

SE style:

<flow descriptor list> ::= <SE flow descriptor>

<SE flow descriptor> ::=

<FLOWSPEC> <filter spec list> [MULTI_FLOWSPEC]

<filter spec list> ::= <FILTER_SPEC>

| <filter spec list> <FILTER_SPEC>

3.1 New MULTI_TSPEC and MULTI_RSPEC Parameters

This extension to Integrated Services defines two new parameters They are:

1. <parameter name> Multiple-Token-Bucket-Tspec, with a parameter number of 125.
2. <parameter name> Multiple_Guaranteed_Service_RSPEC with a parameter number of 124

These are IANA registered in this document.

The original SENDER_TSPEC and FLOWSPEC for Controlled Service maintain the <parameter name> of Token_Bucket_Tspec with a parameter number of 127. The original FLOWSPEC for Guaranteed Service maintains the <parameter name> of Guaranteed_Service_RSPEC with a parameter number of 130.

3.2 Multiple TSPEC in a PATH Message

Here is the object from [RFC2210]. It is used as a SENDER_TSPEC in a PATH message:

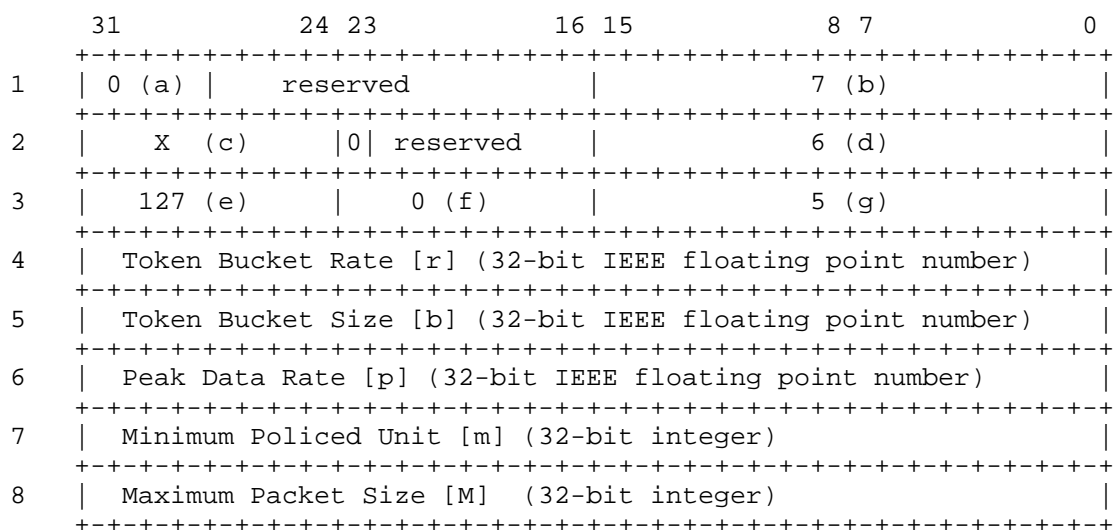
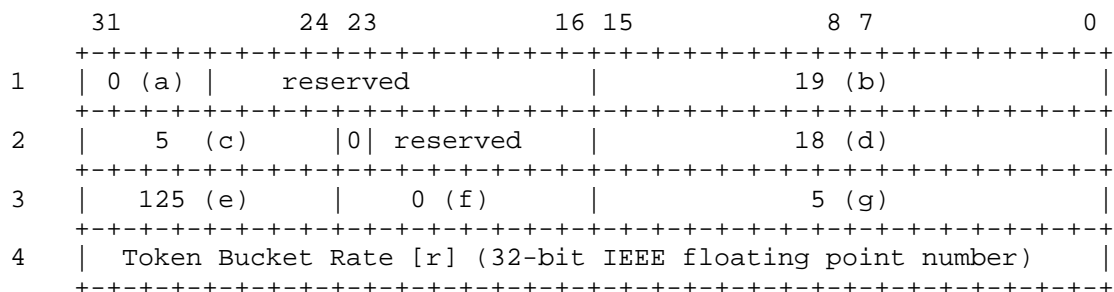


Figure 4. SENDER_TSPEC in PATH

- (a) - Message format version number (0)
- (b) - Overall length (7 words not including header)
- (c) - Service header, service number
 - '1' (Generic information) if in a PATH message;
- (d) - Length of service data, 6 words not including per-service header
- (e) - Parameter ID, parameter 127 (Token Bucket TSpec)
- (f) - Parameter 127 flags (none set)
- (g) - Parameter 127 length, 5 words not including per-service header

For completeness, Figure 4 is included in its original form for backwards compatibility reasons, as if there were only 1 TSPEC in the PATH. What is new when there are more than one TSPEC in this reservation message is the new MULTI_TSPEC object in Figure 5 containing, for example, 3 (Multiple-Token-Bucket-Tspec) TSPECs in a PATH message.



```

5  | Token Bucket Size [b] (32-bit IEEE floating point number) |
  +-----+
6  | Peak Data Rate [p] (32-bit IEEE floating point number) |
  +-----+
7  | Minimum Policed Unit [m] (32-bit integer) |
  +-----+
8  | Maximum Packet Size [M] (32-bit integer) |
  +-----+
9  | 125 (e) | 0 (f) | 5 (g) |
  +-----+
10 | Token Bucket Rate [r] (32-bit IEEE floating point number) |
  +-----+
11 | Token Bucket Size [b] (32-bit IEEE floating point number) |
  +-----+
12 | Peak Data Rate [p] (32-bit IEEE floating point number) |
  +-----+
13 | Minimum Policed Unit [m] (32-bit integer) |
  +-----+
14 | Maximum Packet Size [M] (32-bit integer) |
  +-----+
15 | 125 (e) | 0 (f) | 5 (g) |
  +-----+
16 | Token Bucket Rate [r] (32-bit IEEE floating point number) |
  +-----+
17 | Token Bucket Size [b] (32-bit IEEE floating point number) |
  +-----+
18 | Peak Data Rate [p] (32-bit IEEE floating point number) |
  +-----+
19 | Minimum Policed Unit [m] (32-bit integer) |
  +-----+
20 | Maximum Packet Size [M] (32-bit integer) |
  +-----+

```

Figure 5. MULTI_TSPEC Object

- (a) - Message format version number (0)
- (b) - Overall length (19 words not including header)
- (c) - Service header, service number 5 (Controlled-Load)
- (d) - Length of service data, 18 words not including per-service header
- (e) - Parameter ID, parameter 125 (Multiple Token Bucket TSPEC)
- (f) - Parameter 125 flags (none set)
- (g) - Parameter 125 length, 5 words not including per-service header

Figure 5 shows the 2nd through Nth TSPEC in the PATH in the preferred order. The message format (a) remains the same for a second TSPEC and for other additional TSPECs.

The Overall Length (b) includes all the TSPECs within this object, plus the 2nd Word (containing fields (c) and (d)), which MUST NOT be repeated. The service header fields (e),(f) and(g) are repeated for

each TSPEC.

The Service header, here service number 5 (Controlled-Load) MUST remain the same.

Each TSPEC is six 32-bit Words long (the per-service header plus the 5 values that are 1 Word each in length), therefore the length is in 6 Word increments for each additional TSPEC. Case in point, from the above Figure 5, Words 3-8 are the first TSPEC (2nd preferred), Words 9-14 are the next TSPEC (3rd preferred), and Words 15-20 are the final TSPEC (and 4th preferred) in this example of 3 TSPECs in this MULTI_TSPEC object. There is no limit placed on the number of TSPECs a MULTI_TSPEC object can have. However, it is RECOMMENDED to administratively limit the number of TSPECs in the MULTI_TSPEC object to 9 (making for a total of 10 in the PATH message).

The TSPECs are included in the order of preference by the message generator (PATH) and MUST be maintained in that order all the way to the Receiver. The order of TSPECs that are still grantable, in conjunction with the ADSPEC at the Receiver, MUST retain that order in the FLOWSPEC and MULTI_FLOWSPEC objects.

3.3 Multiple FLOWSPEC for Controlled-Load service

The format of an RSVP FLOWSPEC object requesting Controlled-Load service is the same as the one used for the SENDER_TSPEC given in Figure 4.

The format of the new MULTI_FLOWSPEC object is given below:

	31	24 23	16 15	8 7	0
1	0 (a)	reserved		19 (b)	
2	5 (c)	0 reserved		18 (d)	
3	125 (e)	0 (f)		5 (g)	
4	Token Bucket Rate [r] (32-bit IEEE floating point number)				
5	Token Bucket Size [b] (32-bit IEEE floating point number)				
6	Peak Data Rate [p] (32-bit IEEE floating point number)				
7	Minimum Policed Unit [m] (32-bit integer)				
8	Maximum Packet Size [M] (32-bit integer)				
9	125 (e)	0 (f)		5 (g)	


```

10 | Token Bucket Rate [r] (32-bit IEEE floating point number) |
   +-----+
11 | Token Bucket Size [b] (32-bit IEEE floating point number) |
   +-----+
12 | Peak Data Rate [p] (32-bit IEEE floating point number) |
   +-----+
13 | Minimum Policed Unit [m] (32-bit integer) |
   +-----+
14 | Maximum Packet Size [M] (32-bit integer) |
   +-----+
15 | 125 (e) | 0 (f) | 5 (g) |
   +-----+
16 | Token Bucket Rate [r] (32-bit IEEE floating point number) |
   +-----+
17 | Token Bucket Size [b] (32-bit IEEE floating point number) |
   +-----+
18 | Peak Data Rate [p] (32-bit IEEE floating point number) |
   +-----+
19 | Minimum Policed Unit [m] (32-bit integer) |
   +-----+
20 | Maximum Packet Size [M] (32-bit integer) |
   +-----+

```

Figure 5. Multiple FLOWSPEC for Controlled-Load service

- (a) - Message format version number (0)
- (b) - Overall length (19 words not including header)
- (c) - Service header, service number 5 (Controlled-Load)
- (d) - Length of controlled-load data, 18 words not including per-service header
- (e) - Parameter ID, parameter 125 (Multiple Token Bucket TSPEC)
- (f) - Parameter 125 flags (none set)
- (g) - Parameter 125 length, 5 words not including per-service header

This is for the 2nd through Nth TSPEC in the RESV, in the preferred order.

The message format (a) remains the same for a second TSPEC and for additional TSPECs.

The Overall Length (b) includes the TSPECs, plus the 2nd Word (fields (c) and (d)), which MUST NOT be repeated. The service header fields (e),(f) and(g), which are repeated for each TSPEC.

The Service header, here service number 5 (Controlled-Load) MUST remain the same for the RESV message. The services, Controlled-Load and Guaranteed MUST NOT be mixed within the same RESV message. In other words, if one TSPEC is a Controlled Load service TSPEC, the remaining TSPECs MUST be Controlled Load service. This same rule also is true for Guaranteed Service - if one TSPEC is for Guaranteed

Service, the rest of the TSPECs in this PATH or RESV MUST be for Guaranteed Service.

The Length of controlled-load data (d) also increases to account for the additional TSPECs.

Each FLOWSPEC is six 32-bit Words long (the per-service header plus the 5 values that are 1 Word each in length), therefore the length is in 6 Word increments for each additional TSPEC. Case in point, from the above Figure 5, Words 3-8 are the first TSPEC (2nd preferred), Words 9-14 are the next TSPEC (3rd preferred), and Words 15-20 are the final TSPEC (and 4th preferred) in this example of 3 TSPECs in this FLOWSPEC. There is no limit placed on the number of TSPECs a particular FLOWSPEC can have.

Within the MULTI_FLOWSPEC, any SENDER_TSPEC that cannot be reserved - based on the information gathered in the ADSPEC, is not placed in the RESV or based on other information available to the receiver. Otherwise, the order in which the TSPECs were in the PATH message MUST be in the same order they are in the FLOWSPEC in the RESV. This is the order of preference of the sender, and MUST be maintained throughout the reservation establishment, unless the ADSPEC indicates one or more TSPECs cannot be granted, or the receiver cannot include any TSPEC due to technical or administrative constraints or one or more routers along the RESV path cannot grant a particular TSPEC. At any router that a reservation cannot honor a TSPEC, this TSPEC MUST be removed from the RESV, or else another router along the RESV path might reserve that TSPEC. This rule ensures this cannot happen.

Once one TSPEC has been removed from the RESV, the next in line TSPEC becomes the preferred TSPEC for that reservation. That router MUST generate a ResvErr message, containing an ERROR_SPEC object with a Policy Control Failure with Error code = 2 (Policy Control Failure), and an Error Value sub-code 102 (ERR_PARTIAL_PREEMPT) to the previous routers, clearing the now over allocation of bandwidth for this reservation. The difference between the previously accepted TSPEC bandwidth and the currently accepted TSPEC bandwidth is the amount this error identifies as the amount of bandwidth that is no longer required to be reserved. The ResvErr and the RESV messages are independent, and not normally sent by the same router. This aspect of this document is the extension to RFC 2205 (RSVP).

If a RESV cannot grant the final TSPEC, normal RSVP rules apply with regard to the transmission of a particular ResvErr.

3.4 Multiple FLOWSPEC for Guaranteed service

The FLOWSPEC object, which is used to request guaranteed service contains a TSPEC and RSpec. Here is the FLOWSPEC object from [RFC2215] when requesting Guaranteed service:

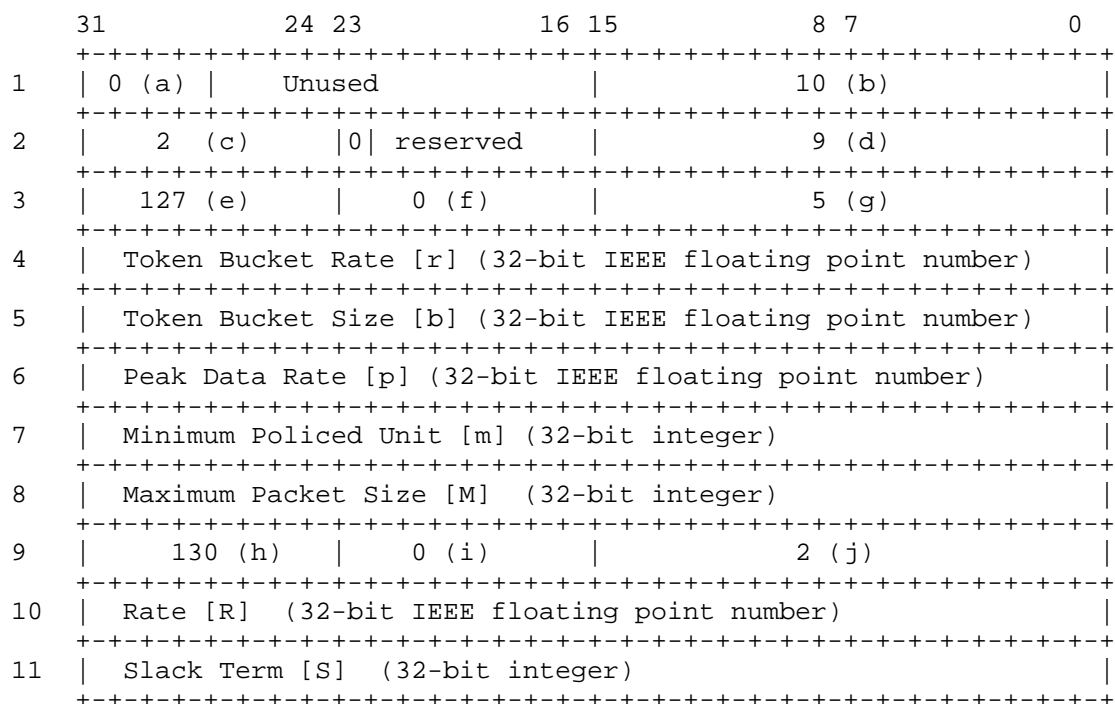
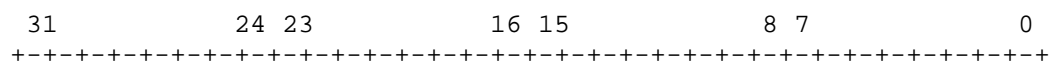


Figure 6. FLOWSPEC for Guaranteed service

- (a) - Message format version number (0)
- (b) - Overall length (9 words not including header)
- (c) - Service header, service number 2 (Guaranteed)
- (d) - Length of per-service data, 9 words not including per-service header
- (e) - Parameter ID, parameter 127 (Token Bucket TSpec)
- (f) - Parameter 127 flags (none set)
- (g) - Parameter 127 length, 5 words not including parameter header
- (h) - Parameter ID, parameter 130 (Guaranteed Service RSpec)
- (i) - Parameter xxx flags (none set)
- (j) - Parameter xxx length, 2 words not including parameter header

The difference in structure between the Controlled-Load FLOWSPEC and Guaranteed FLOWSPEC is the RSpec, defined in [RFC2212].

For completeness, Figure 6 is included in its original form for backwards compatibility reasons, as if there were only 1 FLOWSPEC in the RESV. What is new when there is more than one TSPEC in the FLOWSPEC in a RESV message is the new MULTI_FLOWSPEC object in Figure 7 containing, for example, 3 FLOWSPECs requesting Guaranteed Service.



1	0 (a) Unused 28 (b)
2	2 (c) 0 reserved 27 (d)
3	125 (e) 0 (f) 5 (g)
4	Token Bucket Rate [r] (32-bit IEEE floating point number)
5	Token Bucket Size [b] (32-bit IEEE floating point number)
6	Peak Data Rate [p] (32-bit IEEE floating point number)
7	Minimum Policed Unit [m] (32-bit integer)
8	Maximum Packet Size [M] (32-bit integer)
9	124 (h) 0 (i) 2 (j)
10	Rate [R] (32-bit IEEE floating point number)
11	Slack Term [S] (32-bit integer)
12	125 (e) 0 (f) 5 (g)
13	Token Bucket Rate [r] (32-bit IEEE floating point number)
14	Token Bucket Size [b] (32-bit IEEE floating point number)
15	Peak Data Rate [p] (32-bit IEEE floating point number)
16	Minimum Policed Unit [m] (32-bit integer)
17	Maximum Packet Size [M] (32-bit integer)
18	124 (h) 0 (i) 2 (j)
19	Rate [R] (32-bit IEEE floating point number)
20	Slack Term [S] (32-bit integer)
21	125 (e) 0 (f) 5 (g)
22	Token Bucket Rate [r] (32-bit IEEE floating point number)
23	Token Bucket Size [b] (32-bit IEEE floating point number)
24	Peak Data Rate [p] (32-bit IEEE floating point number)
25	Minimum Policed Unit [m] (32-bit integer)
26	Maximum Packet Size [M] (32-bit integer)

```

27 |      124 (h)      |      0 (i)      |      2 (j)      |
   +-----+-----+-----+-----+-----+-----+-----+
28 | Rate [R]  (32-bit IEEE floating point number) |
   +-----+-----+-----+-----+-----+-----+-----+
29 | Slack Term [S] (32-bit integer) |
   +-----+-----+-----+-----+-----+-----+-----+

```

Figure 7. Multiple FLOWSPECs for Guaranteed service

- (a) - Message format version number (0)
- (b) - Overall length (9 words not including header)
- (c) - Service header, service number 2 (Guaranteed)
- (d) - Length of per-service data, 9 words not including per-service header
- (e) - Parameter ID, parameter 125 (Token Bucket TSPEC)
- (f) - Parameter 125 flags (none set)
- (g) - Parameter 125 length, 5 words not including parameter header
- (h) - Parameter ID, parameter 124 (Guaranteed Service RSpec)
- (i) - Parameter 124 flags (none set)
- (j) - Parameter 124 length, 2 words not including parameter header

There MUST be 1 RSpec per TSPEC for Guaranteed Service. Therefore, there are 5 words for Receiver TSPEC and 3 words for the RSpec. Therefore, for Guaranteed Service, the TSPEC/RSPEC combination occurs in increments of 8 words.

4. Rules of Usage

The following rules apply to nodes adhering to this specification:

4.1 Backward Compatibility

If the recipient does not understand this extension, it ignores this MULTI_TSPEC object, and operates normally for a node receiving this RSVP message.

4.2 Applies to Only a Single Session

When there is more than one TSPEC object or more than one FLOWSPEC object, this MUST NOT be considered for more than one flow created. These are OR choices for the same flow of data. In order to attain three reservations between two endpoints, three different reservation requests are required, not one reservation request with 3 TSPECs.

4.3 No Special Error Handling for PATH Message

If a problem occurs with the PATH message - regardless of this

extension, normal RSVP procedures apply (i.e., there is no new PathErr code created within this extension document) - resulting in a PathErr message being sent upstream towards the sender, as usual.

4.4 Preference Order to be Maintained

When more than one TSPEC is in a PATH message, the order of TSPECs is decided by the Sender and MUST be maintained within the SENDER_TSPEC. The same order MUST be carried to the FLOWSPECs by the receiver. No additional TSPECs can be introduced by the receiver or any router processing these new objects. The deletion of TSPECs from a PATH message is not permitted. The deletion of the TSPECs when forming the FLOWSPEC is allowed by the receiver in the following cases:

- If one or more preferred TSPECs cannot be granted by a router as discovered during processing of the ADSPEC by the receiver, then they can be omitted when creating the FLOWSPEC(s) from the TSPECs.
- If one or more TSPECs arriving from the sender is not preferred by the receiver, then the receiver MAY omit any while creating the FLOWSPEC. A good reason to omit a TSPEC is if, for example, it does not match a codec supported by the receiver's application(s).

The deletion of the TSPECs in the router during the processing of this MULTI_FLOWSPEC object is allowed in the following cases:

- If the original FLOWSPEC cannot be granted by a router then the router may discard that FLOWSPEC and replace it with the topmost FLOWSPEC from the MULTI_FLOWSPEC project. This will cause the topmost FLOWSPEC in the MULTI_FLOWSPEC object to be removed. The next FLOWSPECs becomes the topmost FLOWSPEC.
- If the router merges multiple RESV into a single RESV message, then the FLOWSPEC and the multiple FLOWSPEC may be affected

The preferred order of the remaining TSPECs or FLOWSPECs MUST be kept intact both at the receiver as well as the router processing these objects.

4.5 Bandwidth Reduction in Downstream Routers

If there are multiple FLOWSPECs in a single RESV message, it is quite possible that a higher bandwidth is reserved at a previous downstream device. Thus, any device that grants a reservation that is not the highest will have to inform the previous downstream routers to reduce the bandwidth reserved for this particular session.

The bandwidth reduction RFC [RFC4495] has the ability to partially

preempt existing reservations. However, it does not address the need that this draft addresses. RFC 4495 defines an ability to preempt part of an existing reservation so as to admit a new incoming reservation with a higher priority, in lieu of tearing down the whole reservation with lower priority. It does not specify the capability to reduce the bandwidth a RESV set up along the data path before the reservation is realized (from source to destination), when a subsequent router cannot support a more preferred FLOWSPEC contained in that RESV. This document will extend the RFC 4495 defined error to work for previous hops while a reservation is being established.

4.6 Merging Rules

RFC 2205 defines the rules for merging as combining more than one FLOWSPEC into a single FLOWSPEC. In the case of MULTI_FLOWSPECs, merging of the two (or more) MULTI_FLOWSPEC MUST be done to arrive at a single MULTI_FLOWSPEC. The merged MULTI_FLOWSPEC will contain all the flow specification components of the individual MULTI_FLOWSPECs in descending orders of bandwidth. In other words, the merged FLOWSPEC MUST maintain the relative order of each of the individual FLOWSPECs. For example, if the individual FLOWSPEC order is 1,2,3 and another FLOWSPEC is a,b,c, then this relative ordering cannot be altered in the merged FLOWSPEC.

A byproduct of this is the ordering between the two individual FLOWSPECs cannot be signaled with this extension. If two (or more) FLOWSPECs have the same bandwidth, they are to be merged into one FLOWSPEC using the rules defined in RFC 2205. It is RECOMMENDED that the following rules are used for determining ordering (in TSPEC and FLOWSPEC):

For Controlled Load - in descending order of BW based on the Token Bucket Rate 'r' parameter value

For Guaranteed Service - in descending order of BW based on the RSPEC Rate 'R' parameter value

The resultant FLOWSPEC is added to the MULTI_FLOWSPEC based on its bandwidth in descending orders of bandwidth.

As a result of such merging, the number of FLOWSPECs in a MULTI_FLOWSPEC object should be the sum of the number of FLOWSPECs from individual MULTI_FLOWSPEC that have been merged *minus* the number of duplicates.

4.7 Applicability to Multicast

An RSVP message with a MULTI_TSPEC works just as well in a multicast scenario as it does in a unicast scenario. In a multicast scenario, the bandwidth allotted in each hop is the lowest bandwidth that can

be admitted along the various path. For example:

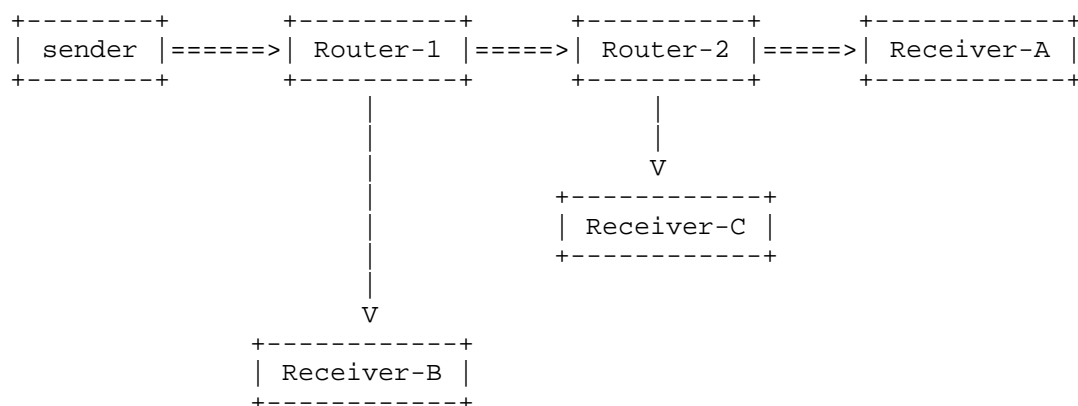


Figure 8. MULTI_TPSEC and Multicast

If the sender (in Figure 8) sends 3 TSPECs (i.e., 1 TSPEC Object, and 2 in the MULTI_TSPEC Object) of 12Mbps, 5Mbps and 1.5Mbps. Let us say the path from Receiver-B to Router-1 admitted 5Mbps, Receiver-C to Router-2 admitted 1.5Mbps and Receiver-A to Router-2 admitted 12Mbps.

When the Resv message is send upstream from Router-2, the combining of 1.5Mbps (to Receiver-C) and 12Mbps (to Receiver-A) will be resolved to 1.5Mbps (lowest that can be admitted). Only a Resv with 1.5Mbps will be sent upstream from Router-2. Likewise, at Router-1, the combining of 1.5Mbps (to Router-2) and 5Mbps (to Receiver-B) will be resolved to 1.5Mbps units.

This is to allow the sender to transmit the flow at a rate that can be accepted by all devices along the path. Without this, if Router-2 receives a flow of 12Mbps, it will not know how to create a flow of 1.5Mbps down to Receiver-B. A differentiated reservation for the various paths along a multicast path is only possible with a Media-aware network device (MANE). The discussion of MANE and how it relates to admission control is outside the scope of this draft.

4.8 MULTI TSPEC Specific Error

Since this mechanism is backward compatible, it is possible that a router without support for this MULTI_TSPEC extension will reject a reservation because the bandwidth indicated in the primary FLOWSPECs is not available. This means that an attempt with a lower bandwidth might have been successful, if one were included in a MULTI_TSPEC Object. Therefore, one should be able to differentiate between an admission control error where there is insufficient bandwidth when all the FLOWSPECs are considered and insufficient bandwidth when

only the primary FLOWSPEC is considered.

This requires the definition of an error code within the ERROR_SPEC Object. When a router does not have sufficient bandwidth even after considering all the FLOWSPEC provided, it issues a new "MULTI_TSPEC bandwidth unavailable" error. This will be an Admission Control Failure (error #1), with a subcode of 6. A router that does not support this MULTI_TSPEC extension will return the "requested bandwidth unavailable" error as defined in RFC 2205 as if there was no MULTI_TSPEC in the message.

4.9 Other Considerations

- RFC 4495 articulates why a ResvErr is more appropriate to use for reducing the bandwidth of an existing reservation vs. a ResvTear.
- Refreshes only include the TSPECs that were accepted. One SHOULD be sent immediately upon the Sender receiving the RESV, to ensure all routers in this flow are synchronized with which TSPEC is in place.
- Periodically, it might be appropriate to attempt to increase the bandwidth of an accepted reservation with one of the TSPECs that were not accepted by the network when the reservation was first installed. This SHOULD NOT occur too regularly. This document currently offers no guidance on the frequency of this bump request for a rejected TSPEC from the PATH.

4.10 Known Open Issues

Here are the know open issues within this document:

- o Both the idea of MULTI_RSPEC and MULTI_FLOWSPEC need to be fleshed out, and IANA registered.
- o Need to ensure the cap on the number of TSPECs and FLOWSPECs is viable, yet controlled.

5. Security considerations

The security considerations for this document do not exceed what is already in RFC 2205 (RESV) or RFC 2210 (IntServ), as nothing in either of those documents prevent a node from requesting a lot of bandwidth in a single TSPEC. This document merely reduces the signaling traffic load on the network by allowing many requests that fall under the same policy controls to be included in a single round-trip message exchange.

Further, this document does not increase the security risk(s) to

that defined in RFC 4495, where this document creates additional meaning to the RFC 4495 created error code 102.

A misbehaving Sender can include too many TSPECs in the MULTI_TSPEC object, which can lead to an amplification attack. That said, a bad implementation can create a reservation for each TSPEC received from within the Resv message. The number of TSPECs in the new MULTI_TSPEC object is limited, and the spec clearly states that only a single reservation is to be set up per Resv message.

6. IANA considerations

This document IANA registers the following new parameter name in the Integ-serv assignments at [IANA]:

Registry Name: Parameter Names

Registry:

Value	Description	Reference
-----	-----	-----
125	Multiple-Token-Bucket-Tspec	[RFCXXXX]
124	Multiple-Guaranteed-Service-RSpec	[RFCXXXX]

Where RFCXXXX is replaced with the RFC number assigned to this Document.

This document IANA registers the following new error subcode in the Error code section, under the Admission Control Failure (error=1), of the rsvp-parameters assignments at [IANA]:

Registry Name: Error Codes and Globally-Defined Error Value
Sub-Codes

Registry:

"Admission Control
Failure"

Error Subcode	meaning	Reference
-----	-----	-----
6	= MULTI_TSPEC bandwidth unavailable	[RFCXXXX]

7. Acknowledgments

The authors wish to thank Fred Baker, Joe Touch, Bruce Davie, Dave Oran, Ashok Narayanan, Lou Berger, Lars Eggert, Arun Kudur and Janet Gunn for their helpful comments and guidance in this effort.

And to Francois Le Faucheur, who provided text in this version.

8. References

8.1. Normative References

- [RFC2119] S. Bradner, "Key words for use in RFCs to Indicate Requirement Levels", RFC 2119, March 1997
- [RFC2205] R. Braden, Ed., L. Zhang, S. Berson, S. Herzog, S. Jamin, "Resource ReSerVation Protocol (RSVP) -- Version 1 Functional Specification", RFC 2205, September 1997
- [RFC2210] J. Wroclawski, "The Use of RSVP with IETF Integrated Services", RFC 2210, September 1997
- [RFC2212] S. Shenker, C. Partridge, R. Guerin, "Specification of Guaranteed Quality of Service", RFC 2212, September 1997
- [RFC2215] S. Shenker, J. Wroclawski, "General Characterization Parameters for Integrated Service Network Elements", RFC 2212, September 1997
- [RFC4495] J. Polk, S. Dhesikan, "A Resource Reservation Protocol (RSVP) Extension for the Reduction of Bandwidth of a Reservation Flow", RFC 4495, May 2006

8.2. Informative References

- [IANA] <http://www.iana.org/assignments/integ-serv>

Authors' Addresses

James Polk
3913 Treemont Circle
Colleyville, Texas, USA
+1.817.271.3552

[mailto: jmpolk@cisco.com](mailto:jmpolk@cisco.com)

Subha Dhesikan
Cisco Systems
170 W. Tasman Drive
San Jose, CA 95134 USA

[mailto: sdhesika@cisco.com](mailto:sdhesika@cisco.com)

Appendix A: Alternatives for Sending Multiple TSPECs

This appendix describes the discussion within the TSVWG of which approach best fits the requirements of sending multiple TSPECs within a single PATH or RESV message. There were 3 different options proposed, of which - 2 were insufficient or caused more harm

than other options.

Looking at the format of a PATH message [RFC2205] again:

```

<PATH Message> ::= <Common Header> [ <INTEGRITY> ]

                                <SESSION> <RSVP_HOP>

                                <TIME_VALUES>

                                [ <POLICY_DATA> ... ]

                                [ <sender descriptor> ]

<sender descriptor> ::= <SENDER_TEMPLATE> <SENDER_TSPEC>
                                ^^^^^^^^^^^^^^^
                                [ <ADSPEC> ]

```

For the PATH message, the focus of this document is with what to do with respect to the <SENDER_TSPEC> above, highlighted by the '^^^^' characters. No other object within the PATH message will be affected by this IntServ extension.

The ADSPEC is optional in IntServ; therefore it might or might not be in the RSVP PATH message. Presently, the SENDER_TSPEC is limited to one bandwidth associated with the session. This is changed in this extension to IntServ to multiple bandwidths for the same session. There are multiple options on how the additional bandwidths may be added:

Option #1 - creating the ability to add one or more additional (and complete) SENDER_TSPECs,

or

Option #2 - create the ability for the one already allowed SENDER_TSPEC to carry more than one bandwidth amount for the same reservation.

or

Option #3 - create the ability for the existing SENDER_TSPEC to remain unchanged, but add an optional <MULTI_TSPEC> object to the <sender descriptor> such as this:

```

<sender descriptor> ::= <SENDER_TEMPLATE> <SENDER_TSPEC>

                                [ <ADSPEC> ] [ <MULTI_TSPEC> ]
                                ^^^^^^^^^^^^^^^

```

Here is another way of looking at the option choices:

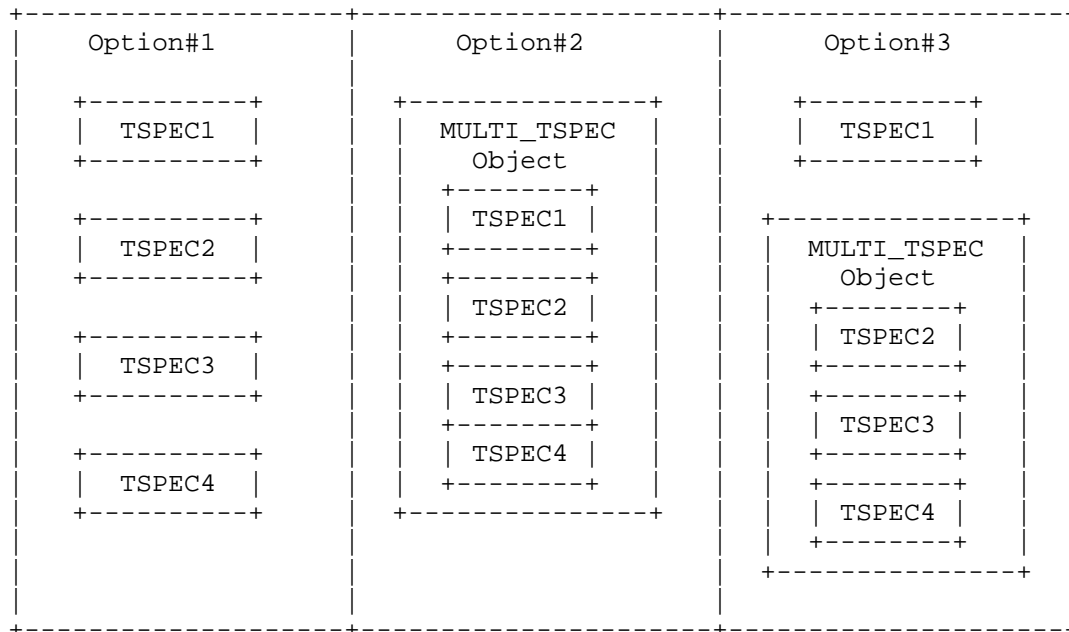


Figure 3. Concept of Option Choice

Option #1 and #2 do not allow for backward compatibility. If the currently used SENDER_TSPEC and FLOWSPEC objects are changed, then unless all the routers requiring RSVP processing are upgraded, this functionality cannot be realized. As it is unlikely that all routers along the path will have the necessary enhancements as per this extension at one given time, therefore, it is necessary this enhancement be made in a way that is backward compatible. Therefore, option #1 and option #2 has been discarded in favor of option #3, which had WG consensus in a recent IETF meeting.

Option #3: This option has the advantage of being backwards compatible with existing implementations of [RFC2205] and [RFC2210], as the multiple TSPECs and FLOWSPECs are inserted as optional objects and such objects do not need to be processed, especially if they are not understood.

Option#3 applies to the FLOWSPEC contained in the RESV message as well. In this option, the original SENDER_TSPEC and the FLOWSPEC are left untouched, allowing routers not supporting this extension to be able to process the PATH and the RESV message without issue. Two new additional objects are defined in this document. They are the MULTI_TSPEC and the MULTI_FLOWSPEC for the PATH and the RESV message, respectively. The additional TSPECs (in the new MULTI_TSPEC Object) are included in the PATH and the additional FLOWSPECs (in the new MULTI_FLOWSPEC Object) are included in the RESV message as new (optional) objects. These additional objects will have a class number of 11bbbbbb, allowing older routers to ignore the object(s)

and forward each unexamined and unchanged, as defined in section 3.10 of [RFC 2205].

We state in the document body that the top most FLOWSPEC of the new MULTI_FLOWSPEC Object in the RESV message replaces the existing FLOWSPEC when it is determined by the receiver (perhaps along with the ADSPEC) that the original FLOWSPEC cannot be granted. Therefore, the ordering of preference issue is solved with Option#3 as well.

NOTE: it is important to emphasize here that including more than one FLOWSPEC in the RESV message does not cause more than one FLOWSPEC to be granted. This document requires that the receiver arrange these multiple FLOWSPECs in the order of preference according to the order remaining from the MULTI_TSPECs in the PATH message. The benefit of this arrangement is that RSVP does not have to process the rest of the FLOWSPEC if it can admit the first one.

Additional details of these options can be found in the draft-polk-tsvwg-...-01 version of this appendix (which includes the RSVP bit mapping of fields in the TSPECs, if the reader wishes to search for that doc.

This Internet-Draft, draft-touch-tsvwg-port-use-00.txt, has expired, and has been deleted from the Internet-Drafts directory. An Internet-Draft expires 185 days from the date that it is posted unless it is replaced by an updated version, or the Secretariat has been notified that the document is under official review by the IESG or has been passed to the RFC Editor for review and/or publication as an RFC. This Internet-Draft was not published as an RFC.

Internet-Drafts are not archival documents, and copies of Internet-Drafts that have been deleted from the directory are not available. The Secretariat does not have any information regarding the future plans of the author or working group, if applicable, with respect to this deleted Internet-Draft. For more information, or to request a copy of the document, please contact the author directly.

Draft Author:

Joseph Touch<touch@isi.edu>

Network Working Group

Internet-Draft

Intended status: Standards Track

Expires: April 12, 2013

M. Tuexen

I. Ruengeler

Muenster Univ. of Appl. Sciences

R. Stewart

Adara Networks

October 9, 2012

SACK-IMMEDIATELY Extension for the Stream Control Transmission Protocol
draft-tuexen-tsvwg-sctp-sack-immediately-10.txt

Abstract

This document defines a method for the sender of a DATA chunk to indicate that the corresponding SACK chunk should be sent back immediately and not be delayed.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 12, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
2. Conventions	3
3. The I-bit in the DATA Chunk Header	3
4. Procedures	4
4.1. Sender Side Considerations	4
4.2. Receiver Side Considerations	5
5. Interoperability Considerations	5
6. Socket API Considerations	5
7. IANA Considerations	5
8. Security Considerations	6
9. Acknowledgments	6
10. References	6
10.1. Normative References	6
10.2. Informative References	6
Authors' Addresses	6

1. Introduction

According to [RFC4960] the receiver of a DATA chunk should use delayed SACKs. This delaying is completely controlled by the receiver of the DATA chunk.

In specific situations the delaying of SACKs results in reduced performance of the protocol. If such a situation can be detected by the receiver, the corresponding SACK can be sent immediately. For example, [RFC4960] recommends the immediate sending if the receiver has detected message loss or message duplication. However, if the situation can only be detected by the sender of the DATA chunk, [RFC4960] provides no method of avoiding the delaying of the SACK. Thus the protocol performance might be reduced.

This document overcomes this limitation and describes a simple extension of the SCTP DATA chunk by defining a new flag, the I-bit. The sender of a DATA chunk indicates by setting this bit that the corresponding SACK chunk should not be delayed.

Upper layers of SCTP using the socket API as defined in [RFC6458] may subscribe to the SCTP_SENDER_DRY_EVENT for getting a notification as soon as no user data is outstanding anymore. To avoid an unnecessary delay while waiting for such an event, the application might set the I-Bit on the last DATA chunk sent before waiting for the event. This enabling is possible using the extension of the socket API described in Section 6.

There are also situations in which the SCTP implementation can set the I-bit without interacting with the upper layer. If the association is in the SHUTDOWN-PENDING state, the I-bit should be set. This reduces the number of simultaneous associations in case of a busy server handling short living associations. Another case is where the sending of a DATA chunk fills the congestion or receiver window. Setting the I-bit in these cases improves the throughput of the transfer.

2. Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

3. The I-bit in the DATA Chunk Header

The following Figure 1 shows the extended DATA chunk.

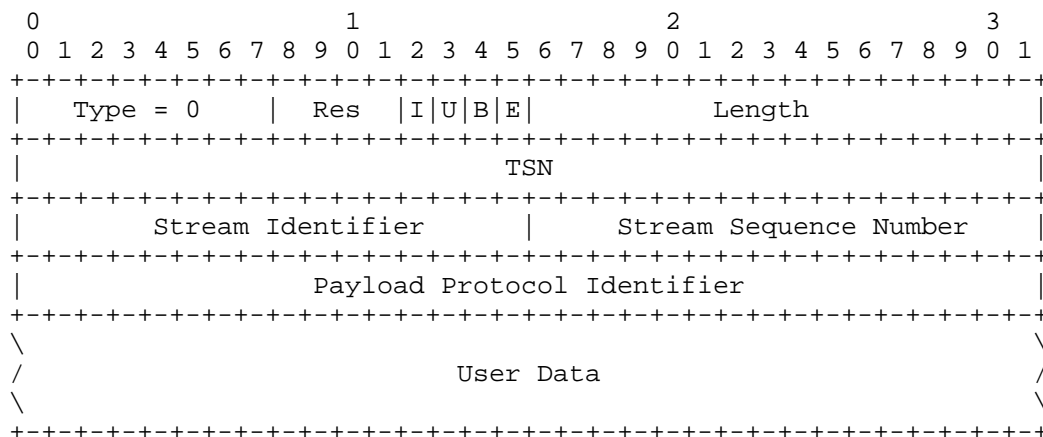


Figure 1: Extended DATA chunk format

The only difference between the DATA chunk in Figure 1 and the DATA chunk defined in [RFC4960] is the addition of the I-bit in the flags field of the chunk header.

4. Procedures

4.1. Sender Side Considerations

Whenever the sender of a DATA chunk can benefit from the corresponding SACK chunk being sent back without delay, the sender MAY set the I-bit in the DATA chunk header. Please note that it is irrelevant to the receiver why the sender has set the I-bit.

Reasons for setting the I-bit include but are not limited to the following:

- o The application requests to set the I-bit of the last DATA chunk of a user message when providing the user message to the SCTP implementation (see Section 6).
- o The sender is in the SHUTDOWN-PENDING state.
- o The sending of a DATA chunk fills the congestion or receiver window.

4.2. Receiver Side Considerations

On reception of an SCTP packet containing a DATA chunk with the I-bit set, the receiver SHOULD NOT delay the sending of the corresponding SACK chunk and SHOULD send it back immediately.

5. Interoperability Considerations

According to [RFC4960] the receiver of a DATA chunk with the I-bit set should ignore this bit when it does not support the extension described in this document. Since the sender of the DATA chunk is able to handle this case, there is no requirement for negotiating the support of the feature described in this document.

6. Socket API Considerations

This section describes how the socket API defined in [RFC6458] is extended to provide a way for the application to set the I-bit.

Please note that this section is informational only.

A socket API implementation based on [RFC6458] is extended by supporting a flag called SCTP_SACK_IMMEDIATELY, which can be set in the `snd_flags` field of the struct `sctp_sndinfo` structure or the `sinfo_flags` field of the struct `sctp_sndrcvinfo` structure, which is deprecated.

If the SCTP_SACK_IMMEDIATELY flag is set when sending a user message, the I-bit of the last DATA chunk of the corresponding user message is set.

7. IANA Considerations

[NOTE to RFC-Editor:

"RFCXXXX" is to be replaced by the RFC number you assign this document.

]

Following the chunk flag registration procedure defined in [RFC6096] IANA should register a new bit, the I-bit, for the DATA chunk. The suggested value is 0x08. The reference for the new chunk flag in the chunk flags table for the DATA chunk available at `sctp-parameters` [1] should be RFCXXXX.

8. Security Considerations

This document does not add any additional security considerations in addition to the ones given in [RFC4960].

9. Acknowledgments

The authors wish to thank Mark Allmann, Brian Bidulock, Janardhan Iyengar, and Kacheong Poon for their invaluable comments.

10. References

10.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
- [RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.
- [RFC6096] Tuexen, M. and R. Stewart, "Stream Control Transmission Protocol (SCTP) Chunk Flags Registration", RFC 6096, January 2011.

10.2. Informative References

- [RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, December 2011.

URIs

- [1] <<http://www.iana.org/assignments/sctp-parameters>>

Authors' Addresses

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstr. 39
48565 Steinfurt
DE

Email: tuexen@fh-muenster.de

Irene Ruengeler
Muenster University of Applied Sciences
Stegerwaldstr. 39
48565 Steinfurt
DE

Email: i.ruengeler@fh-muenster.de

Randall R. Stewart
Adara Networks
Chapin, SC 29036
US

Email: randall@lakerest.net

Network Working Group
Internet-Draft
Intended status: Experimental
Expires: June 20, 2018

L. Wood
Surrey alumni
W. Eddy
MTI Systems
C. Smith
Vallona
W. Ivancic
Syzygy
C. Jackson
SSTL
December 17, 2017

Saratoga: A Scalable Data Transfer Protocol
draft-wood-tsvwg-saratoga-22

Abstract

This document specifies the Saratoga transfer protocol. Saratoga was originally developed to transfer remote-sensing imagery efficiently from a low-Earth-orbiting satellite constellation, but is useful in many other scenarios, including ad-hoc peer-to-peer communications, large-scale scientific sensing, and grid computing. Saratoga is a simple, lightweight, content dissemination protocol that builds on UDP, and optionally uses UDP-Lite. Saratoga is intended for use when moving files or streaming data between peers which may have permanent, sporadic or intermittent connectivity, and is capable of transferring very large amounts of data reliably under adverse conditions. The Saratoga protocol is designed to cope with highly asymmetric link or path capacity between peers, and can support fully-unidirectional data transfer if required. Saratoga can also cope with very large files for exascale computing. In scenarios with dedicated links, Saratoga focuses on high link utilization to make the most of limited connectivity times, while standard congestion control mechanisms can be implemented for operation over shared links. Loss recovery is implemented via a simple negative-ack ARQ mechanism. The protocol specified in this document is considered to be appropriate for experimental use on private IP networks.

Status of This Memo

This Internet-Draft is submitted to IETF in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on June 20, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document.

This document may not be modified, and derivative works of it may not be created, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1. Background and Introduction	3
2. Overview of Saratoga File Transfer	6
3. Optional Parts of Saratoga	11
3.1. Optional but useful functions in Saratoga	11
3.2. Optional congestion control	12
3.3. Optional functionality requiring other protocols	12
4. Packet Types	13
4.1. BEACON	16
4.2. REQUEST	21
4.3. METADATA	25
4.4. DATA	30
4.5. STATUS	34
5. The Directory Entry	41
6. Behaviour of a Saratoga Peer	45
6.1. Saratoga Sessions	45
6.2. Beacons	48
6.3. Upper-Layer Interface	49
6.4. Inactivity Timer	49
6.5. Streams and wrapping	50
6.6. Completing file delivery and ending the session	50
7. Implementation Development	51
8. Security Considerations	51
9. IANA Considerations	52

10. Acknowledgements	52
11. A Note on Naming	52
12. References	53
12.1. Normative References	53
12.2. Informative References	53
Appendix A. Timestamp/Nonce field considerations	55
Authors' Addresses	56

1. Background and Introduction

Saratoga is a file transfer and content delivery protocol capable of efficiently sending both small (kilobyte) and extremely large (yottabyte) files, as well as streaming continuous content. Saratoga was originally designed for the purpose of large file transfer from small low-Earth-orbiting satellites. It has been used in daily operations since 2004 to move mission imaging data files of the order of several hundred megabytes each from the Disaster Monitoring Constellation (DMC) remote-sensing satellites to ground stations.

The DMC satellites, built at the University of Surrey by Surrey Satellite Technology Ltd (SSTL), all use IP for payload communications and delivery of Earth imagery. At the time of this writing, in April 2015, nine DMC satellites have been launched into orbit since 2002, four of those are currently operational in orbit, and three more are under construction. The DMC satellites use Saratoga to provide Earth imagery under the aegis of the International Charter on Space and Major Disasters.

An orbital pass giving a period of visibility and connectivity between a satellite and ground station offers an 8-12 minute time window in which to transfer imagery files, using a minimum of an 8.1 Mbps downlink and a 9.6 kbps uplink. Newer operational DMC satellites can use faster downlinks, capable of 20, 40, 80, 105 or 210 Mbps [Brenchley12]. Planned DMC satellites are expected to use downlinks at up to 320 Mbps, without significant increases in uplink rates. SSTL's TechDemoSat-1 satellite, launched in July 2014 and also carrying Saratoga, uses a 400 Mbps downlink [Brenchley12]. This high degree of link asymmetry, with the need to fully utilize the available downlink capacity to move the volume of data required within the limited time available, motivates much of Saratoga's design.

Further details on how these DMC satellites use IP to communicate with the ground and the terrestrial Internet are discussed elsewhere [Hogie05][Wood07a]. Saratoga has also been implemented for use in high-speed private ground networks supporting radio astronomy sensors [Wood11].

Store-and-forward delivery relies on reliable hop-by-hop transfers of files, removing the need for the final receiver to talk to the original sender across long delays and allowing for the possibility that an end-to-end path may never exist between sender and receiver at any given time. Breaking an end-to-end path into multiple hops allows data to be transferred as quickly as possible across each link; congestion on a longer Internet path is then not detrimental to the transfer rate on a space downlink. Use of store-and-forward hop-by-hop delivery is typical of scenarios in space exploration for both near-Earth and deep-space missions, and useful for other scenarios, such as underwater networking, ad-hoc sensor networks, and some message-ferrying relay scenarios, where efficient delivery must not be constrained by the limitations of a bottleneck in the overall end-to-end path. Saratoga is intended to be useful for relaying data in these scenarios.

Saratoga contains a Selective Negative Acknowledgement (SNACK) 'holestofill' mechanism to provide reliable retransmission of data. This is intended to correct losses of corrupted link-layer frames due to channel noise over a space link. Packet losses in the DMC are due to corruption introducing non-recoverable errors in the frame. The DMC design uses point-to-point links and scheduling of applications in order, so that the link is dedicated to one application transfer at a time, meaning that packet loss cannot be due to congestion when applications compete for link capacity simultaneously. In other wireless environments that may be shared by many nodes and applications, allocation of channel resources to nodes becomes a MAC-layer function. Forward Error Coding (FEC) to get the most reliable transmission through a channel is best left near the physical layer so that it can be tailored for the channel. Use of FEC complements Saratoga's transport-level negative-acknowledgement approach that provides a reliable ARQ mechanism.

Saratoga is scalable in that it is capable of efficiently transferring small or large files, by choosing a width of file offset descriptor appropriate for the filesize, and advertising accepted offset descriptor sizes. 16-bit, 32-bit, 64-bit and 128-bit descriptors can be selected, for maximum file sizes of 64KiB-1 (<64 Kilobytes of disk space), 4GiB-1 (<4 Gigabytes), 16EiB-1 (<16 Exabytes) and 256 EiEiB-1 (<256 Exa-exabytes) respectively.

Earth imaging files currently transferred by Saratoga are mostly up to a few gigabytes in size. Some implementations do transfer more than 4 GiB in size, and so require offset descriptors larger than 32 bits. We believe that supporting a 128-bit descriptor can satisfy many future Big Data needs, but we expect current implementations to only support up to 32-bit or 64-bit descriptors, depending on their application needs. The 16-bit descriptor is useful for small

messages, including messages from 8-bit devices, and is always supported. The 128-bit descriptor can be used for moving very large files stored on a 128-bit filesystem, such as on OpenSolaris ZFS.

As a UDP-based protocol, Saratoga can be used with either IPv4 or IPv6. Compatibility between Saratoga and the wide variety of links that can already carry IP traffic is assured.

High link utilization is important during periods of limited connectivity. Given that Saratoga was originally developed for scheduled peer-to-peer communications over dedicated links in private networks, where each application has the entire link for the duration of its transfer, many Saratoga implementations deliberately lack any form of congestion control and send at line rate to maximise throughput and link utilisation in their limited, carefully controlled, environments. In accordance with UDP Guidelines [RFC5405] for protocols able to traverse the public Internet, newer implementations may perform TCP-Friendly Rate Control (TFRC) [RFC5348] or other congestion control mechanisms. This is described further in [I-D.wood-tsvwg-saratoga-congestion-control].

Saratoga was originally implemented as outlined in [Jackson04], but the specification given here differs substantially, as we have added a number of capabilities while cleaning up the initial Saratoga specification. The original Saratoga code uses a version number of 0, while code that implements this version of the protocol advertises a version number of 1. Further discussion of the history and development of Saratoga is given in [Wood07b].

This document contains an overview of the transfer process and sessions using Saratoga in Section 2, followed by a formal definition of the packet types used by Saratoga in Section 4, and the details of the various protocol mechanisms in Section 6.

Here, Saratoga session types are labelled with underscores around lowercase names (such as a "_get_" session), while Saratoga packet types are labelled in all capitals (such as a "REQUEST" packet) in order to distinguish between the two.

The remainder of this specification uses 'file' as a shorthand for 'binary object', which may be a file, or other type of data. This specification uses 'file' when also discussing streaming of data of indeterminate length. Saratoga uses unsigned integers in its fields, and does not use signed types.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119. [RFC2119]

2. Overview of Saratoga File Transfer

Saratoga is a peer-to-peer protocol in the sense that multiple files may be transferred in both directions simultaneously between two communicating Saratoga peers, and there is not intended to be a strict client-to-server relationship.

Saratoga nodes can act as simple file servers. Saratoga supports several types of operations on files including "pull" downloads, "push" uploads, directory listing, and deletion requests. Each operation is handled as a distinct "session" between the peers.

Saratoga nodes MAY advertise their presence, capabilities, and desires by sending BEACON packets. These BEACONS are sent to either a reserved, unforwardable, multicast address when using IPv4, or a link-local all-Saratoga-peers multicast address when using IPv6. A BEACON might also be unicast to another known node as a sort of "keepalive". Saratoga nodes may dynamically discover other Saratoga nodes, either through listening for BEACONS, through pre-configuration, via some other trigger from a user, lower-layer protocol, or another process. The BEACON is useful in many situations, such as ad-hoc networking, as a simple, explicit, confirmation that another node is present; a BEACON is not required in order to begin a Saratoga session.. BEACONS have been used by the DMC satellites to indicate to ground stations that a link has become functional, a solid-state data recorder is online, and the software is ready to transfer any requested files.

A Saratoga session begins with either a `_get_`, `_put_`, `_getdir_`, or `_delete_` session REQUEST packet corresponding to a desired download, upload, directory listing, or deletion operation. `_put_` sessions MAY instead begin directly with METADATA and DATA, without an initial REQUEST/OKAY STATUS exchange; these rarer cases are known as 'blind puts'. The most common envisioned session is the `_get_`, which begins with a single Saratoga REQUEST packet sent from the peer wishing to receive the file, to the peer who currently has the file. If the session is rejected, then a brief STATUS packet that conveys rejection is generated. If the file-serving peer accepts the session, an OKAY STATUS can be optional; the peer can immediately generate and send a more useful descriptive METADATA packet, along with some number of DATA packets constituting the requested file.

These DATA packets are finished by (and can intermittently include) a DATA packet with a flag bit set that demands the file-receiver send a reception report in the form of a STATUS packet. This DATA-driven cycle is shown in Figure 1. The STATUS packet can include 'holestofill' Selective Negative Acknowledgement (SNACK) information listing spans of octets within the file that have not yet been

received, as well as whether or not the METADATA packet was received, or an error code terminating the transfer session. Once the information in this STATUS packet is received, the file-sender can begin a cycle of selective retransmissions of missing DATA packets, until it sees a STATUS packet that acknowledges total reception of all file data.

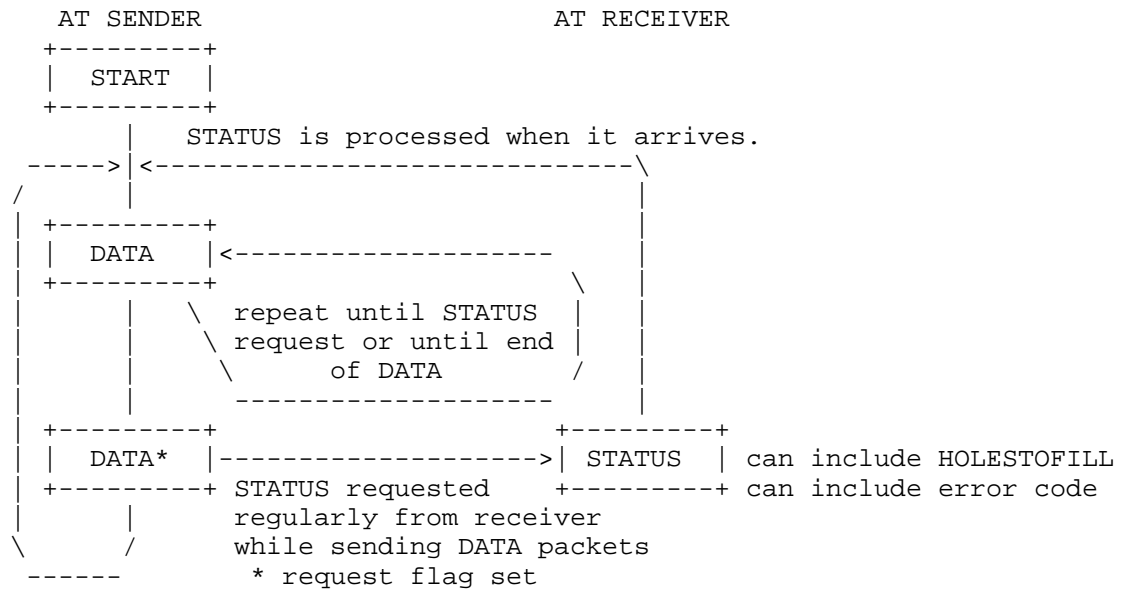


Figure 1: STATUS and DATA cycle

In the example scenario in Figure 2, a `_get_` request is granted. The reliable file delivery experiences loss of a single DATA packet due to channel-induced errors.

```

File-Receiver                                File-Sender

GET REQUEST ----->

(indicating error/reject) <---- STATUS

                                or

                                <----- METADATA
                                <----- DATA #1
STATUS -----> (voluntarily sent at start)
      (lost) <----- DATA #2
                                <----- DATA #3 (bit set
                                requesting STATUS)
STATUS ----->
(indicating that range in DATA #2 was lost)
      <----- DATA #2 (bit set
                                requesting STATUS)
STATUS ----->
(complete file and METADATA received)

```

Figure 2: Example `_get_` session sequence

A `_put_` is similar to `_get_`, although once the OKAY STATUS is received, DATA is sent from the peer that originated the `_put_` request. A less common 'blind `_put_`' does not require an REQUEST and OKAY STATUS to be exchanged before sending DATA packets, and is efficient for long-delay or unidirectional links.

A `_getdir_` request proceeds similarly, though the DATA transfer contains a directory record with one or more directory entries, described later, rather than a given file's bytes. `_getdir_` is the only request to also apply to directories, where one or more directory entries for individual files is received.

The STATUS and DATA packets are allowed to be sent at any time within the scope of a session, in order for the file-sending node to optimize buffer management and transmission order. For example, if the file-receiver already has the first part of a file from a previous disrupted transfer, it may send a STATUS at the beginning of the session indicating that it has the first part of the file, and so only needs the last part of the file. Thus, efficient recovery from interrupted sessions between peers becomes possible, similar to ranged FTP and HTTP requests. (Note that METADATA with a checksum is useful to verify that the parts are of the same file and that the file is reassembled correctly.)

The less common Saratoga 'blind _put_' session is initiated by the file-sender sending an optional METADATA packet followed by immediate DATA packets, without requiring a REQUEST or waiting for a STATUS response. This can be considered an "optimistic" mode of protocol operation, as it assumes the implicit session request will be granted. If the sender of a PUT request sees a STATUS packet indicating that the request was declined, it MUST stop sending any DATA packets within that session immediately. Since this type of _put_ is open-loop for some period of time, it should not be used in scenarios where congestion is a valid concern; in these cases, the file-sender should wait on its METADATA to be acknowledged by a STATUS before sending DATA packets within the session.

Figure 3 illustrates the sequence of packets in an example _put_ session, beginning directly with METADATA and DATA, where the second DATA packet is lost. The METADATA SHOULD be sent at the beginning of the transfer, but MAY be sent (or resent) at any time. Other than the way that it is initiated, the mechanics of data delivery of a _put_ session are similar to a _get_ session.

File-Sender	File-Receiver
REQUEST ----->	
	<----- STATUS
METADATA ----->	
DATA #1 ----->	
(transfer accepted)	<----- STATUS
DATA #2 ---> (lost)	
DATA #3 (bit set ----->	
requesting STATUS)	
(DATA #2 lost)	<----- STATUS
DATA #2 (bit set ----->	
requesting STATUS)	
(transfer complete)	<----- STATUS

Figure 3: Example PUT session sequence

In large-distance scenarios such as for deep space, the large propagation delays and round-trip times involved discourage use of ping-pong packet exchanges (such as TCP's SYN/ACK) for starting sessions, and unidirectional transfers via optimistic 'blind _put_s' are desirable. Blind _puts_, skipping the initial REQUEST/STATUS exchange, are the the only mode of transfer suitable for unidirectional links. Senders sending on unidirectional links SHOULD send a copy of the METADATA in advance of DATA packets, and MAY resend METADATA at intervals.

The `_delete_` sessions are simple single packet requests that trigger a STATUS packet with a status code that indicates whether the file was deleted or not. If the file is not able to be deleted for some reason, this reason can be conveyed in the Status field of the STATUS packet.

A `_get_ REQUEST` packet that does not specify a filename (i.e. the request contains a zero-length File Path field) is specially defined to be a request for any chosen file that the peer wishes to send it. This 'blind `_get_`' allows a Saratoga peer to request any files that the other Saratoga peer has ready for it, without prior knowledge of the directory listing, and without requiring the ability to examine files or decode remote file names/paths for meaningful information such as final destination.

If a file is larger than Saratoga can be expected to transfer during a time-limited contact, there are at least two feasible options:

(1) The application can use proactive fragmentation to create multiple smaller-sized files. Saratoga can transfer some number of these smaller files fully during a contact.

(2) To avoid file fragmentation, a Saratoga file-receiver can retain a partially-transferred file and request transfer of the unreceived bytes during a later contact. This uses a STATUS packet to make clear how much of the file has been successfully received and where transfer should be resumed from, and relies on use of METADATA to identify the file. On resumption of a transfer, the new METADATA (including file length, file timestamps, and possibly a file checksum) MUST match that of the previous METADATA in order to re-establish the transfer. Otherwise, the file-receiver MUST assume that the file has changed and purge the DATA payload received during previous contacts.

Like the BEACON packets, a `_put_` or a response to a `_get_` MAY be sent to the dedicated IPv4 Saratoga multicast address (allocated to 224.0.0.108) or the dedicated IPv6 link-local multicast address (allocated to FF02:0:0:0:0:0:0:6C) for multiple file-receivers on the link to hear. This is at the discretion of the file-sender, if it believes that there is interest from multiple receivers. In-progress DATA transfers MAY also be moved seamlessly from unicast to multicast if the file-sender learns during a transfer, from receipt of further unicast `_get_ REQUEST` packets, that multiple nodes are interested in the file. The associated METADATA packet is multicast when this transition takes place, and is then repeated periodically while the DATA stream is being sent, to inform newly-arrived listeners about the file being multicast. Acknowledgements MUST NOT be demanded by multicast DATA packets, to prevent ack implosion at the file-sender,

and instead status SNACK information is aggregated and sent voluntarily by all file-receivers. File-receivers respond to multicast DATA with multicast STATUS packets. File-receivers SHOULD introduce a short random delay before sending a multicast STATUS packet, to prevent ack implosion after a channel-induced loss, and MUST listen for STATUS packets from others, to avoid duplicating fill requests. The file-sender SHOULD repeat any initial unicast portion of the transfer as multicast last of all, and may repeat and cycle through multicast of the file several times while file-receivers express interest via STATUS or `_get_` packets. Once in multicast and with METADATA being repeated periodically, new file-receivers do not need to send individual REQUEST packets. If a transfer has been started using UDP-Lite and new receivers indicate UDP-only capability, multicast transfers MUST switch to using UDP to accommodate them.

3. Optional Parts of Saratoga

Implementing support for some parts of Saratoga is optional. These parts are grouped into three sections, namely useful capabilities in Saratoga that are likely to be supported by implementations, congestion control that is needed in shared networks and across the public Internet, and functionality requiring other protocols that is less likely to be supported.

3.1. Optional but useful functions in Saratoga

These are useful capabilities in Saratoga that implementations SHOULD support, but may not, depending on scenarios:

- sending and parsing BEACONS.
- sending METADATA. However, sending and receiving METADATA is considered extremely useful, is strongly recommended, and SHOULD be done. A METADATA that is received MUST be parsed.
- streaming data, including real-time streaming of content of unknown length. This streaming can be unreliable (without resend requests) or reliable (with resend requests). Session protocols such as http expect reliable streaming. Although Saratoga data delivery is inherently one-way, where a stream of DATA packets elicits a stream of STATUS packets, bidirectional duplex communication can be established by using two Saratoga transfers flowing in opposite directions.
- multicast DATA transfers, if judged useful for the environment in which Saratoga is deployed, when multiple receivers are participating and are receiving the same file or stream.

- sending and parsing STATUS messages, which are expected for bidirectional communication, but cannot be sent on and are not required for sending over unidirectional links.
- sending and responding to packet timestamps in DATA and STATUS packets. These timestamps are useful for streaming and for giving a file-sender an indication of path latency for rate control. There is no need for a file-receiver to understand the format used for these timestamps for it to be able to receive them from and reflect them back to the file-sender.
- support for descriptor sizes greater than 16 bits, for handling small files, is optional, as is support for descriptor sizes greater than 32 bits, and support for descriptor sizes greater than 64 bits. If a descriptor size is implemented, all sizes below that size MUST be implemented.

3.2. Optional congestion control

Saratoga can be implemented to perform congestion control at the sender, based on feedback from acknowledgement STATUS packets [I-D.wood-tsvwg-saratoga-congestion-control], or have the sender configured to use simple open-loop rate control to only use a fixed amount of link capacity. Congestion control is expected to be undesirable for many of Saratoga's use cases and expected environmental conditions in private networks, where sending as quickly as possible or simple rate control at a fixed output speed are considered useful.

In accordance with the UDP Guidelines [RFC5405], congestion control MUST be supported if Saratoga is being used across the public Internet, and SHOULD be supported in environments where links are shared by traffic flows. Congestion control may not be supported across private, single-flow links engineered for performance: Saratoga's primary use case.

3.3. Optional functionality requiring other protocols

The functionality listed here is useful in rare cases, but requires use of other, optional, protocols. This functionality MAY be supported by Saratoga implementations:

- transfers permitting some errors in content delivered, using UDP-Lite [RFC3828]. These can be useful for decreasing delivery time over unreliable channels, especially for unidirectional links, or in decreasing computational overhead for the UDP Lite checksum. To be really usefully, error tolerance requires that lower-layer frames

permit delivery of unreliable data, while header information is still checked to assure that e.g. destination information is reliable.

If a file contains separate parts that require reliable transmission without errors or that can tolerate errors in delivered content, proactive fragmentation can be used to split the file into separate reliable and unreliable files that can be transferred separately, using UDP or UDP-Lite.

If parts of a file require reliability but the rest can be sent by unreliable transfer, the file-sender can use its knowledge of the internal file structure and vary DATA packet size so that the reliable parts always start after the offset field and are covered by the UDP-Lite checksum.

A file that permits unreliable delivery can be transferred onwards using UDP. If the current sender does not understand the internal file format to be able to decide what parts must be protected with payload checksum coverage, the current sender or receiver does not support UDP-Lite, or the current protocol stack only implements error-free frame delivery below the UDP layer, then the file MAY be delivered using UDP.

4. Packet Types

Saratoga is defined for use with UDP over either IPv4 or IPv6 [RFC0768]. UDP checksums, which are mandatory with IPv6, MUST be used with IPv4. Within either version of IP datagram, a Saratoga packet appears as a typical UDP header followed by an octet indicating how the remainder of the packet is to be interpreted:

```

      1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|          UDP source port          |          UDP destination port          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          UDP length                |          UDP checksum                |
+-----+-----+-----+-----+-----+-----+-----+-----+
|Vers |Pckt Type| other Saratoga fields ... //
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Saratoga data transfers can also be carried out using UDP-Lite [RFC3828]. If Saratoga can be carried over UDP-Lite, the implementation MUST also support UDP. All packet types except DATA MUST be sent using UDP with checksums turned on. For reliable transfers, DATA packets are sent using UDP with checksums turned on. For files where unreliable transfer has been indicated as desired and possible, the sender MAY send DATA packets unreliably over UDP-Lite,

where UDP-Lite protects only the Saratoga headers and parts of the file that must be transmitted reliably.

The three-bit Saratoga version field ("Ver") identifies the version of the Saratoga protocol that the packet conforms to. The value 001 MUST be used in this field for implementations conforming to the specification in this document, which specifies version 1 of Saratoga. The value 000 was used in earlier implementations, prior to the formal specification and public submission of the protocol design, and is incompatible with version 001 in many respects.

The five-bit Saratoga "Packet Type" field indicates how the remainder of the packet is intended to be decoded and processed:

#	Type	Use
0	BEACON	Beacon packet indicating peer status.
1	REQUEST	Commands peer to start a transfer.
2	METADATA	Carries file transfer metadata.
3	DATA	Carries octets of file data.
4	STATUS	responds to REQUEST or DATA. Can signal list of unreceived data to sender during a transfer.

Several of these packet types include a Flags field, for which only some of the bits have defined meanings and usages in this document. Other, undefined, bits may be reserved for future use. Following the principle of being conservative in what you send and liberal in what you accept, a packet sender MUST set any undefined bits to zero, and a packet recipient MUST NOT rely on these undefined bits being zero on reception.

The specific formats for the different types of packets are given in this section. Some packet types contain file offset descriptor fields, which contain unsigned integers. The lengths of the offset descriptors are fixed within a transfer, but vary between file transfers. The size is set for each particular transfer, depending on the choice of offset descriptor width made in the METADATA packet, which in turn depends on the size of file being transferred.

In this document, all of the packet structure figures illustrating a packet format assume 32-bit lengths for these offset descriptor fields, and indicate the transfer-dependent length of the fields by using a "(descriptor)" designation within the [field] in all packet diagrams. That is:

The example 32-bit descriptors shown in all diagrams here

```
+-----+
[                (descriptor)                ]
+-----+
```

are suitable for files of up to 4GiB - 1 octets in length, and may be replaced in a file transfer by descriptors using a different length, depending on the size of file to be transferred:

16-bit descriptor for short files of up to 64KiB - 1 octets in size (MUST be supported)

```
+-----+
[          (descriptor)          ]
+-----+
```

64-bit descriptor for longer files of up to 16EiB - 1 octets in size (optional)

```
+-----+
[                (descriptor)                ] /
+-----+
/                (descriptor, continued)        ]
+-----+
```

128-bit descriptor for very long files of up to 256 EiEiB - 1 octets in size (optional)

```
+-----+
[                (descriptor)                ] /
+-----+
/                (descriptor, continued)        /
+-----+
/                (descriptor, continued)        /
+-----+
/                (descriptor, continued)        ]
+-----+
```

Descriptors are used for the descriptor size less one octet, e.g. 16-bit for files up to 64KB - 1 octets in size, before switching to the larger descriptor, e.g. using the 32-bit descriptor for a 64KB file and larger.

For offset descriptors and types of content being transferred, the related flag bits in BEACON and REQUEST indicate capabilities, while in METADATA and DATA those flag bits are used slightly differently, to indicate the content being transferred.

Saratoga packets are intended to fit within link MTUs to avoid the inefficiencies and overheads of lower-layer fragmentation. A Saratoga implementation does not itself perform any form of MTU discovery, but is assumed to be configured with knowledge of usable maximum IP MTUs for the link interfaces it uses.

4.1. BEACON

BEACON packets may be multicast periodically by nodes willing to act as Saratoga peers, or unicast to individual peers to indicate properties for that peer. Some implementations have sent BEACONS every 100 milliseconds, but this rate is arbitrary, and should be chosen to be appropriate for the environment and implementation.

The main purpose for sending BEACONS is to announce the presence of the node to potential peers (e.g. satellites, ground stations) to provide automatic service discovery, and also to confirm the activity or presence of the peer.

The Endpoint Identifier (EID) in the BEACON serves to uniquely identify the Saratoga peer. Whenever the Saratoga peer begins using a new IP address, it SHOULD issue a BEACON on it and repeat the BEACON periodically, to enable listeners to associate the IP address with the EID and the peer.

Format

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 0 1 |  Type  |               Flags               |
+-----+-----+-----+-----+-----+-----+-----+
| [               Available free space (optional)           ] |
+-----+-----+-----+-----+-----+-----+-----+
|               Endpoint identifier...                      //
+-----+-----+-----+-----+-----+-----+-----+//

```

where

Field	Description
Type	0
Flags	convey whether or not the peer is ready to send/receive, what the maximum supported file size range and descriptor is, and whether and how free space is indicated.
Available free space	This optional field can be used to indicate the current free space available for storage.
Endpoint identifier	This MUST be used to uniquely identify the sending Saratoga peer, or the administrative node that the BEACON-sender is associated with.

The Flags field is used to provide some additional information about the peer. The first two octets of the Flags field is currently in use. The later octet is reserved for future use, and MUST be set to zero.

The BEACON flags field, expanding a line of flag bits with descriptions of each flag, is as follows:

BEACON Flags

```

0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
|0|0|1| => Version Field: Saratoga version 1
|   |0|0|0|0|0| => Type field: BEACON Frame designation
|   |X|X| => Descriptor size
|   |0| => Reserved for future functionality
|   |X| => Supports streaming?
|   |X|X| => Sending files
|   |X|X| => Receiving files
|   |X| => Supports UDP Lite?
|   |X| => Includes free space size?
|   |X|X| => Freespace Descriptor
+-----+-----+-----+-----+

```

The two highest-order bits (bits 8 and 9 above) indicate the maximum supported file size parameters that the peer's Saratoga implementation permits. Other Saratoga packet types contain variable-length fields that convey file sizes or offsets into a file -- the file offset descriptors. These descriptors may be 16-bit, 32-bit, 64-bit, or 128-bit in length, depending on the size of the file being transferred and/or the integer types supported by the sending peer.

The indicated bounds for the possible values of these bits are summarized below:

Bit 8	Bit 9	Supported Field Sizes	Maximum File Size
0	0	16 bits	$2^{16} - 1$ octets.
0	1	16 or 32 bits	$2^{32} - 1$ octets.
1	0	16, 32, or 64 bits	$2^{64} - 1$ octets.
1	1	16, 32, 64, or 128 bits	$2^{128} - 1$ octets.

If a Saratoga peer advertises it is capable of receiving a certain size of file, then it **MUST** also be capable of receiving files sent using smaller descriptor values. This avoids overhead on small files, while increasing interoperability between peers.

It is likely when sending unbounded streams that a larger offset descriptor field size will be preferred to minimise problems with offset sequence numbers wrapping. Protecting against sequence number wrapping is discussed in the STATUS section.

Bit	Value	Meaning
10	0	Reserved for future use.

Bit 10 is reserved for possible future use, and its use is not specified here. This bit **MUST** be set to zero by implementations conforming to this specification.

Bit	Value	Meaning
11	0	not capable of supporting streaming.
11	1	capable of supporting streaming.

Bit 11 is used to indicate whether the sender is capable of sending and receiving continuous streams.

Bit 12	Bit 13	Capability and willingness to send files
0	0	cannot send files at all.
0	1	invalid.
1	0	capable of sending, but not willing right now.
1	1	capable of and willing to send files.

Bit 14	Bit 15	Capability and willingness to receive files
0	0	cannot receive files at all.
0	1	invalid.
1	0	capable of receiving, but unwilling. Will reject METADATA or DATA packets.
1	1	capable of and willing to receive files.

Also in the Flags field, bits 12 and 14 act as capability bits, while bits 13 and 15 augment those flags with bits indicating current willingness to use the capability.

Bits 12 and 13 deal with sending, while bits 14 and 15 deal with receiving. If bit 12 is set, then the peer has the capability to send files. If bit 14 is set, then the peer has the capability to receive files. Bits 13 and 15 indicate willingness to send and receive files, respectively.

A peer that is able to act as a file-sender MUST set the capability bit 12 in all BEACONS that it sends, regardless of whether it is willing to send any particular files to a particular peer at a particular time. Bit 13 indicates the current presence of data to send and a willingness to send it in general, in order to augment the capability advertised by bit 12.

If bit 14 is set, then the peer is capable of acting as a receiver, although it still might not currently be ready or willing to receive files (for instance, it may be low on free storage). This bit MUST be set in any BEACON packets sent by nodes capable of acting as file-receivers. Bit 15 augments this by expresses a current general willingness to receive and accept files.

Bit	Value	Meaning
16	0	supports DATA transfers over UDP only.
16	1	supports DATA transfers over both UDP and UDP-Lite.

Bit 16 is used to indicate whether the sender is capable of sending and receiving unreliable transfers via UDP-Lite.

Bit	Value	Meaning
17	0	available free space is not advertised in this BEACON.
17	1	available free space is advertised in this BEACON.

Bit 17 is used to indicate whether the sender includes an optional field in this BEACON packet that tells how much free space is available. If bit 17 is set, then bits 18 and 19 are used to indicate the size in bits of the optional free-space-size field. If bit 17 is not set, then bits 18 and 19 are zero.

Bit 18	Bit 19	Size of free space field
0	0	16 bits.
0	1	32 bits.
1	0	64 bits.
1	1	128 bits.

The free space field size can vary as indicated by a varying-size field indicated in bits 18 and 19 of the flags field. Unlike other offset descriptor use where the value in the descriptor indicates a byte or octet position for retransmission, or gives a file size in bytes, this particular field indicates the available free space in KIBIBYTES (KiB, multiples of 1024 octets), rather than octets. Available free space is rounded down to the nearest KiB, so advertising zero means that less than 1KiB is free and available. Advertising the maximum size possible in the field means that more free space than that is available. While this field is intended to be scalable, it is expected that 32 bits (up to 4TiB) will be most common in use.

A BEACON unicast to an individual peer MAY choose to indicate the free space available for use by that particular peer, and MAY

indicate capabilities only available to that particular peer, overriding or supplementing the properties advertised to all local peers by multicast BEACONS.

Any type of host identifier can be used in the endpoint identifier field, as long as it is a reasonably unique string within the range of operational deployment. This identifier **MUST** be encoded in UTF-8 in the packet. This field encompasses the remainder of the packet. No terminating null byte is included.

4.2. REQUEST

A REQUEST packet is an explicit command to perform either a `_put_`, `_get_`, `_getdir_`, or `_delete_` session.

Format

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 0 1 | Type |           Flags           | Request Type |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Session Id                               |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               variable-length File Path ...           |
+-----+-----+-----+-----+-----+-----+-----+-----+
/                               /
+-----+-----+-----+-----+-----+-----+-----+-----+
/                               | null byte |                               /
+-----+-----+-----+-----+-----+-----+-----+-----+
/   variable-length Authentication Field (optional)   |
+-----+-----+-----+-----+-----+-----+-----+-----+

```

where

Field	Description
Type	1
Flags	provide additional information about the requested file/operation; see table below for definition.
Request Type	identifies the type of request being made; see table further below for request values.
Id	uniquely identifies the session between two peers.
File Path	the path of the requested file/directory following the rules described below.

The Id that is used during sessions serves to uniquely associate a given packet with a particular sessions. This enables multiple simultaneous data transfer or request/status sessions between two peers, with each peer deciding how to multiplex and prioritise the parallel flows it sends. The Id for a session is selected by the initiator so as to not conflict with any other in-progress or recent sessions with the same host. This Id should be unique and generated using properties of the file, which will remain constant across a host reboot. The 3-tuple of both host identifiers and a carefully-generated session Id field can be used to uniquely index a particular session's state.

The REQUEST flags field, expanding a line of flag bits with descriptions of each flag, is as follows:

REQUEST Flags

```

      0              1              2              3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|0|0|1| => Version field: Saratoga version 1
|   |0|0|0|0|1| => Type field: REQUEST Frame designation
|   |X|X| => Descriptor size
|   |   |0| => Reserved for future use.
|   |   |X| => Supports streaming?
|   |   |   |X| => Supports UDP Lite?
|   |   |   |   |Request Type field <= |X|X|X|X|X|X|X|X|X|
+-----+-----+-----+-----+-----+-----+-----+-----+

```

In the Flags field, the bits labelled 8 and 9 in the figure above indicate the maximum supported file length fields that the peer can handle, and are interpreted exactly as the bits 8 and 9 in the BEACON packet described above. Bits 12 and 13, and 14 and 15, indicate capability and willingness to send and receive files, as described above. Making a `_get_` request would require that the requester is capable and willing to receive files. The remaining defined individual bits are as summarised as follows:

Bit	Value	Meaning
10	0	Reserved for future use.
11	0	The requester cannot receive streams.
11	1	The requester is also able to receive streams.
16	0	The requester is able to receive DATA over UDP only.
16	1	The requester is also able to receive DATA over UDP-Lite.

The Request Type field is an octet that contains a value indicated the type of request being made. Possible values are:

Value	Meaning
0	No action is to be taken; similar to a BEACON.
1	A <code>_get_</code> session is requested. The File Path field holds the name of the file to be sent.
2	A <code>_put_</code> session is requested. The File Path field suggests the name of the file that will be delivered only after an OK STATUS is received from the file receiver.
3	A <code>_get_</code> session is requested, and once received successfully, the original copy should be deleted. The File Path field holds the name of the file to be sent. (This get+delete is known as a 'take'.)
4	A <code>_put_</code> session is requested, and once sent successfully, the original copy will be deleted. The File Path field holds the name of the file to be sent. (This put+delete is known as a 'give'.)
5	A <code>_delete_</code> session is requested, and the File Path field specifies the name of the file to be deleted.
6	A <code>_getdir_</code> session is requested. The File Path field holds the name of the directory or file on which the directory record is created.

The File Path portion of a `_get_` packet is a null-terminated UTF-8 encoded string [RFC3629] that represents the path and base file name on the file-sender of the file (or directory) that the file-receiver wishes to perform the `_get_`, `_getdir_`, or `_delete_` operation on. Implementations SHOULD only send as many octets of File Path as are needed for carrying this string, although some implementations MAY choose to send a fixed-size File Path field in all REQUEST packets that is filled with null octets after the last UTF-8 encoded octet of the path. A maximum of 1024 octets for this field, and for the File

Path fields in other Saratoga packet types, is used to limit the total packet size to within a single IPv6 minimum MTU (minus some padding for network layer headers), and thus avoid the need for fragmentation. The 1024-octet maximum applies after UTF-8 encoding and null termination.

As in the standard Internet File Transfer Protocol (FTP) [RFC0959], for path separators, Saratoga allows the local naming convention on the peers to be used. There are security implications to processing these strings without some intelligent filtering and checking on the filesystem items they refer to. See also the Security Considerations section later within this document.

If the File Path field is empty, i.e. is a null-terminated zero-length string one octet long, then this indicates that the file-receiver is ready to receive any file that the file-sender would like to send it, rather than requesting a particular file. This allows the file-sender to determine the order and selection of files that it would like to forward to the receiver in more of a "push" manner. Of course, file retrieval could also follow a "pull" manner, with the file-receiving host requesting specific files from the file-sender. This may be desirable at times if the file-receiver is low on storage space, or other resources. The file-receiver could also use the Saratoga `_getdir_` session results in order to select small files, or make other optimizations, such as using its local knowledge of contact times to pick files of a size likely to be able to be delivered completely. File transfer through pushing sender-selected files implements delivery prioritization decisions made solely at the Saratoga file-sending node. File transfer through pulling specific receiver-selected files implements prioritization involving more participation from the Saratoga file-receiver. This is how Saratoga implements Quality of Service (QoS).

The null-terminated File Path string MAY be followed by an optional Authentication Field that can be used to validate the REQUEST packet. Any value in the Authentication Field is the result of a computation of packet contents that SHOULD include, at a minimum, source and destination IP addresses and port numbers and packet length in a 'pseudo-header', as well as the content of all Saratoga fields from Version to File Path, excluding the predictable null-termination octet. This Authentication Field can be used to allow the REQUEST receiver to discriminate between other peers, and permit and deny various REQUEST actions as appropriate. The format of this field is unspecified for local use.

Combined get+delete (take) and put+delete (give) requests should only have the delete carried out once the deleting peer is certain that the file-receiver has a good copy of the file. This may require the

file receiver to verify checksums before sending a final STATUS message acknowledging successful delivery of the final DATA segment, or aborting the transfer if the checksum fails. If the transfer fails and an error STATUS is sent for any reason, the file should not be deleted.

REQUEST packets may be sent multicast, to learn about all listening nodes. A multicast `_get_` request for a file that elicits multiple METADATA or DATA responses should be followed by unicast STATUS packets with status errors cancelling all but one of the proposed transfers. File timestamps in the Directory Entry can be used to select the most recent version of an offered file, and the host to fetch it from.

If the receiver already has the file at the expected file path and is requesting an update to that file, REQUEST can be sent after a METADATA advertising that file, to allow the sender to determine whether a replacement for the file should be sent.

Delete requests are ignored for files currently being transferred.

4.3. METADATA

METADATA packets are sent as part of a data transfer session (`_get_`, `_getdir_`, and `_put_`). A METADATA packet says how large the file is and what its name is, as well as what size of file offset descriptor is chosen for the session. METADATA packets are optional, but SHOULD be sent. A METADATA packet that is received MUST be parsed. A METADATA packet is normally sent at the start of a DATA transfer, but can be repeated throughout the transfer. Sending METADATA at the start if using checksums allows for incremental checksum calculation by the receiver, and is a good idea.

Format

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 0 1 |  Type   |             Flags             | Sumleng | Sumtype |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Session Id                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                                                                   |
/                                                                                   /
/      example error-detection checksum (128-bit MD5 shown)                       /
/                                                                                   /
/                                                                                   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                                                                   |
/                                                                                   /
/      single Directory Entry describing file                                     /
/      (variable length)                                                         /
/                                                                                   //
+-----+-----+-----+-----+-----+-----+-----+-----+

```

where

Field	Description
Type	2
Flags	indicate additional boolean metadata about a file.
Sumleng	indicates the length of a checksum, as a multiple of 32 bits.
Sumtype	indicates whether a checksum is present after the Id, and what type it is.
Id	identifies the session that this packet describes.
Checksum	an example included checksum covering file contents.
Directory Entry	describes file system information about the file, including file length, file timestamps, etc.; the format is specified in Section 5.

The first octet of the Flags field is currently specified for use. The later two octets are reserved for future use, and MUST be set to zero.

The METADATA flags field is as follows, expanding a line of flag bits with explanations of each field:

METADATA Flags

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|0|0|1| => Version Field: Saratoga version 1
|  |  |0|0|1|0| => Type field: METADATA Frame designation
|  |  |X|X| => Descriptor
|  |  |X|X| => File/stream/dir record
|  |  |X| => Transfer in progress?
|  |  |X| => UDP Lite permitted?
|  |  |Checksum length in no. of 32-bit words <=|X|X|X|X|
|  |  |Error detection checksum type <=|X|X|X|X|
+-----+-----+-----+-----+-----+-----+-----+-----+

```

In the Flags field, the bits labelled 8 and 9 in the figure above indicate the exact size of the offset descriptor fields used in this particular packet and are interpreted exactly as the bits 8 and 9 in the BEACON packet described above. The value of these bits determines the size of the File Length field in the current packet, as well as indicating the size of the offset fields used in DATA and STATUS packets within the session that will follow this packet.

Bit 10	Bit 11	Type of transfer
0	0	a file is being sent.
0	1	the file being sent should be interpreted as a Directory Record.
1	0	Reserved for future use.
1	1	an indefinite-length stream is being sent.

Also inside the Flags field, bits 10 and 11 indicate what is being transferred - a file, special directory record file that contains one or more directory entries, or stream. The value 01 indicates that the METADATA and DATA packets are being generated in response to a `_getdir_ REQUEST`, and that the assembled DATA contents should be interpreted as a Directory Record containing directory entries, as defined in Section 5.

Bit	Value	Meaning
12	0	This transfer is in progress.
12	1	This transfer is no longer in progress, and has been terminated.

Bit 12 indicates whether the transfer is in progress, or has been terminated by the sender. It is normally set to 1 only when METADATA is resent to indicate that a stream transfer has been ended.

Bit	Value	Meaning
13	0	This file's content MUST be delivered reliably without errors using UDP.
13	1	This file's content MAY be delivered unreliably, or partly unreliably, where errors are tolerated, using UDP-Lite.

Bit 13 indicates whether the file must be sent reliably or can be sent at least partly unreliably, using UDP-Lite. This flag SHOULD only be set if the originator of the file knows that at least some of the file content is suitable for sending unreliably and is robust to errors. This flag reflects a property of the file itself. This flag may still be set if the immediate file-receiver is only capable of UDP delivery, on the assumption that this preference will be preserved for later transfers where UDP-Lite transfers may be taken advantage of by senders with knowledge of the internal file structure. The file-sender may know that the receiver is capable of handling UDP-Lite, either from a `_get_ REQUEST`, from exchange of BEACONS, or a-priori.

The high four bits of the Flags field, bits 28-31, are used to indicate if an optional error-detection checksum has been included in the METADATA for the file to be transferred. Here, bits 0000 indicate that no checksum is present, with the implicit assumption that the application will do its own end-to-end check. Other values indicate the type of checksum to use. The choice of checksum depends on the available computing power and the length of the file to be checksummed. Longer files require stronger checksums to ensure error-free delivery. The checksum of the file to be transferred is carried as shown, with a fixed-length field before the varying-length File Length and File Name information fields.

Assigned values for the checksum type field are:

Value (0-15)	Use
0	No checksum is provided.
1	32-bit CRC32 checksum, suitable for small files.
2	128-bit MD5 checksum, suitable for larger files.
3	160-bit SHA-1 checksum, suitable for larger files but slower to process than MD5.

The length of an optional checksum cannot be inferred from the checksum type field, particularly for unknown checksum types. The next-highest four bits of the 32-bit word holding the Flags, bits 24-27, indicate the length of the checksum bit field, as a multiple of 32 bits.

Example Value (0-15)	Use
0	No checksum is provided.
1	32-bit checksum field, e.g. CRC32.
4	128-bit checksum field, e.g. MD5.
5	160-bit checksum field, e.g. SHA-1.

For a 32-bit CRC, the length field holds 1 and the type field holds 1. For MD5, the length field holds 4 and the type field holds 2. For SHA-1, the length field holds 5 and the type field holds 3.

It is expected that higher values will be allocated to new and stronger checksums able to better protect larger files. These checksums can be expected to be longer, with larger checksum length fields.

A checksum **SHOULD** be included for files being transferred. The checksum **SHOULD** be as strong as possible. Streaming of an indefinite-length stream **MUST** set the checksum type field to zero.

It is expected that a minimum of the MD5 checksum will be used, unless the Saratoga implementation is used exclusively for small transfers at the low end of the 16-bit file descriptor range, such as on low-performing hardware, where the weaker CRC-32c checksum can suffice.

The CRC32 checksum is computed as described for the CRC-32c algorithm given in [RFC3309].

The MD5 Sum field is generated via the MD5 algorithm [RFC1321], computed over the entire contents of the file being transferred. The file-receiver can compute the MD5 result over the reassembled Saratoga DATA packet contents, and compare this to the METADATA's MD5 Sum field in order to gain confidence that there were no undetected protocol errors or UDP checksum weaknesses encountered during the transfer. Although MD5 is known to be less than optimal for security uses, it remains excellent for non-security use in error detection (as is done here in Saratoga), and has better performance implications than cryptographically-stronger alternatives given the limited available processing of many use cases [RFC6151]. MD5 use here has similar properties to an Ethernet frame CRC for error detection.

Checksums may be privately keyed for local use, to allow transmission of authenticated or encrypted files delivered in DATA packets. This has limitations, discussed further in Section 8 at end.

Use of the checksum to ensure that a file has been correctly relayed to the receiving node is important. A provided checksum **MUST** be checked against the received data file. If checksum verification fails, either due to corruption or due to the receiving node not having the right key for a keyed checksum), the file **MUST** be discarded. If the file is to be relayed onwards later to another Saratoga peer, the metadata, including the checksum, **MUST** be retained with the file and **SHOULD** be retransmitted onwards unchanged with the file for end-to-end coverage. If it is necessary to recompute the checksum or encrypted data for the new peer, either because a different key is in use or the existing checksum algorithm is not supported, the new checksum **MUST** be computed before the old checksum is verified, to ensure overlapping checksum coverage and detect errors introduced in file storage.

METADATA can be used as an indication to update copies of files. If the METADATA is in response to a `_get_` REQUEST including a file record, and the record information for the held file matches what the requester already has, as has been indicated by a previously-received METADATA advertisement from the requester, then only the METADATA is sent repeating this information and verifying that the file is up to date. If the record information does not match and a newer file can be supplied, the METADATA begins a transfer with following DATA packets to update the file.

4.4. DATA

A series of DATA packets form the main part of a data transfer session (`_get_`, `_put_`, or `_getdir_`). The payloads constitute the actual file data being transferred.

Format

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 0 1 |  Type   |                               Flags          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Session Id                       |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               /                               /
|          Timestamp/nonce information (optional)              /
|                               /                               /
|                               |                               |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Offset (descriptor)             |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                               Payload data...                  //
+-----+-----+-----+-----+-----+-----+-----+-----+

```

where

Field	Description
Type	3
Flags	are described below.
Id	identifies the session to which this packet belongs.
Timestamp/nonce	is an optional 128-bit field providing timing or identification information unique to this packet. See Appendix A for details.
Offset	the offset in octets to the location where the first byte of this packet's payload is to be written.

The DATA packet has a minimum size of ten octets, using sixteen-bit descriptors and no timestamps.

DATA packets are normally checked by the UDP checksum to prevent errors in either the header or the payload content. However, for transfers that can tolerate content errors, DATA packets MAY be sent using UDP-Lite. If UDP-Lite is used, the file-sender must know that the file-receiver is capable of handling UDP-Lite, and the file contents to be transferred should be resilient to errors. The UDP-Lite checksum MUST protect the Saratoga headers, up to and including the offset descriptor, and MAY protect more of each packet's payload,

depending on the file-sender's knowledge of the internal structure of the file and the file's reliability requirements.

The DATA flags field is as follows, expanding a line of flag bits with explanations of each field:

DATA Flags

```

0                               1                               2                               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|0|0|1| => Version Field: Saratoga version 1
|  |  |  |0|0|0|1|1| => Type field: DATA Frame designation
|  |  |  |X|X| => Descriptor
|  |  |  |X|X| => File/stream/dir record
|  |  |  |X| => Includes timestamp?
|  |  |  |X| => STATUS requested
|  |  |  |X| => End of Data (EOD) set
+-----+-----+-----+-----+-----+-----+-----+-----+

```

Bit 8	Bit 9	Type of transfer
0	0	16-bit descriptors are in use in this transfer.
0	1	32-bit descriptors are in use in this transfer.
1	0	64-bit descriptors are in use in this transfer.
1	1	128-bit descriptors are in use in this transfer.

Flag bits 8 and 9 are set to indicate the size of the offset descriptor as described for BEACON and METADATA packets, so that each DATA packet is self-describing. This allows the DATA packet to be used to construct a file even when an initial METADATA is lost and must be resent. The flag values for bits 8 and 9 MUST be the same as indicated in any expected METADATA packet.

Bit 10	Bit 11	Type of transfer
0	0	a file is being sent.
0	1	the file being sent should be interpreted as a directory record.
1	0	Reserved for future use.
1	1	an indefinite-length stream is being sent.

Also inside the Flags field, bits 10 and 11 indicate what is being transferred - a file, special file that contains a Directory Records, or stream. The value 01 indicates that the METADATA and DATA packets are being generated in response to a `_getdir_ REQUEST`, and that the assembled DATA contents should be interpreted as a Directory Record containing directory entries, as defined in Section 5. The flag values for bits 10 and 11 MUST be the same as indicated in the initial METADATA packet.

Bit	Value	Meaning
12	0	This packet does not include an optional timestamp/nonce field.
12	1	This packet includes an optional timestamp/nonce field.

Flag bit 12 indicates that an optional packet timestamp/nonce is carried in the packet before the offset field. This packet timestamp/nonce field is always sixteen octets (128 bits) long. Timestamps can be useful to the sender even when the receiver does not understand them, as the receiver can simply echo any provided timestamps back, as specified for STATUS packets, to allow the sender to monitor flow conditions. Packet timestamps are particularly useful when streaming. Packet timestamps are discussed further in Appendix A.

Bit	Value	Meaning
15	0	No response is requested.
15	1	A STATUS packet is requested.

Within the Flags field, if bit 15 of the packet is set, the file-receiver is expected to immediately generate a STATUS packet to provide the file-sender with up-to-date information regarding the status of the file transfer. This flag is set carefully and rarely. This flag may be set periodically, but infrequently. Asymmetric links with constrained backchannels can only carry a limited amount of STATUS packets before ack congestion becomes a problem. This flag SHOULD NOT be set if an unreliable stream is being transferred, or if multicast is in use. This flag SHOULD be set periodically for reliable file transfers, or reliable streaming. The file-receiver MUST respond to the flag by generating a STATUS packet, unless it knows that doing so will lead to local congestion, in which case it may choose to send a later voluntary STATUS message. Voluntary

STATUS packets MAY be sent if a request for one has not been made within an appropriate time.

Bit	Value	Meaning
16	0	Normal use.
16	1	The EOD End of Data flag is set.

The End of Data flag is set in DATA packets carrying the last byte of a transfer. This is particularly useful for streams and for the rare Saratoga implementations that do not send or receive METADATA.

Immediately following the DATA header is the payload, which consumes the remainder of the packet and whose length is implicitly defined by the end of the packet. The payload octets are directly formed from the continuous octets starting at the specified Offset in the file being transferred. No special coding is performed. A zero-octet payload length is allowable, and a single DATA packet indicating zero payload, consisting only of a header with the EOD flag set, may be useful to simply elicit a STATUS response from the receiver.

The length of the Offset fields used within all DATA packets for a given session MUST be consistent with the length indicated by bits 8 and 9 of any accompanying METADATA packet. If the METADATA packet has not yet been received, a file-receiver that supports METADATA MUST indicate that it has not been received via a STATUS packet, and MAY choose to enqueue received DATA packets for later processing after the METADATA arrives.

4.5. STATUS

The STATUS packet type is the single acknowledgement method that is used for feedback from a Saratoga receiver to a Saratoga sender to indicate session progress, both as a response to a REQUEST, and as a response to a DATA packet when demanded or volunteered.

When responding to a DATA packet, the STATUS packet MAY, as needed, include selective acknowledgement (SNACK) 'hole' information to enable transmission (usually re-transmission) of specific sets of octets within the current session (called "holes"). This 'holestofill' information can be used to clean up losses (or indicate no losses) at the end of, or during, a session, or to efficiently resume a transfer that was interrupted in a previous session.

Format

```

      0               1               2               3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
| 0 0 1 |  Type   |             Flags             |   Status   |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     Session Id                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                     /                                     /
|          Timestamp/nonce information (optional)          /
|                                     /
|                                     |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          Progress Indicator (descriptor)          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          In-Response-To (descriptor)          |
+-----+-----+-----+-----+-----+-----+-----+-----+
|          (possibly, several Hole fields)          /
|                                     ...                                     /
+-----+-----+-----+-----+-----+-----+-----+-----+

```

where

Field	Description
Type	4
Flags	are defined below.
Id	identifies the session that this packet belongs to.
Status	a value of 00 indicates the transfer is successfully proceeding. All other values are errors terminating the transfer, explained below.
Zero-Pad	an octet fixed at 00 to allow later fields to be conveniently aligned for processing.
Timestamp (optional)	an optional fixed 128-bit field, that is only present and used to return a packet timestamp if the timestamp flag is set. If the STATUS packet is voluntary and the voluntary flag is set, this should repeat the timestamp of the DATA packet containing the highest offset seen. If the STATUS packet is in response to a mandatory request, this will repeat the timestamp of the requesting DATA packet. The file-sender may use these timestamps to estimate latency. Packet timestamps are particularly useful when streaming. There are special considerations for streaming, discussed further below, to protect against the ambiguity of wrapped offset descriptor sequence numbers. Packet timestamps are discussed further in Appendix A.
Progress Indicator (descriptor)	the offset of the lowest-numbered octet of the file not yet received, and expected.
In-Response-To (descriptor)	the offset of the octet following the DATA packet that generated this STATUS packet, or the offset of the next expected octet following the highest DATA packet seen if this STATUS is generated voluntarily and the voluntary flag is set.
Holes	indications of offset ranges of missing data, defined below.

The STATUS packet has a minimum size of twelve octets, using sixteen-bit descriptors, a progress indicator but no Hole fields, and no timestamps. The progress indicator is always zero when responding to requests that may initiate a transfer.

The Id field is needed to associate the STATUS packet with the session that it refers to.

The Progress Indicator and In-Response-To fields mark the 'left edge' and 'right edge' of the incomplete working area where holes are being filled in. If there are no holes, these fields will hold the same value. At the start of a transfer, both fields begin by expecting octet zero. When a transfer has completed successfully, these fields will contain the length of the file.

The STATUS flags field is as follows, expanding a line of flag bits with explanations of each field:

STATUS Flags

```

0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
|0|0|1| => Version Field: Saratoga version 1
|      |0|0|1|0|0| => Type field: STATUS Frame designation
|      |X|X| => Descriptor
|      |X| => Timestamp included?
|      |X| => METADATA received?
|      |X| => Hole information complete?
|      |X| => Voluntary STATUS message?
|      |      |      |      |      |      |
|      |      |      |      |      |      |
+-----+-----+-----+-----+
Status code <= |X|X|X|X|X|X|X|X|

```

Flags bits 8 and 9 are set to indicate the size of the offset descriptor as described for BEACON and METADATA packets, so that each STATUS packet is self-describing. The flag values here MUST be the same as indicated in the initial METADATA and DATA packets.

Other bits in the Flags field are defined as:

Bit	Value	Meaning
12	0	This packet does not include a timestamp/nonce field.
12	1	This packet includes an optional timestamp field.

Flag bit 12 indicates that an optional sixteen-byte packet timestamp/nonce field is carried in the packet before the Progress Indicator descriptor, as discussed for the DATA packet format. Packet timestamps are discussed further in Appendix A.

Bit	Value	Meaning
13	0	file's METADATA has been received or is ignored.
13	1	file's METADATA has not been received.

If bit 13 of a STATUS packet has been set to indicate that the METADATA has not yet been received, then any METADATA SHOULD be resent. This flag should normally be clear.

A receiver SHOULD tolerate lost METADATA that is later resent, but MAY insist on receiving METADATA at the start of a transfer. This is done by responding to early DATA packets with a voluntary STATUS packet that sets this flag bit, reports a status error code 10, sets the Progress Indicator field to zero, and does not include HOLESTOFILL information.

Bit	Value	Meaning
14	0	this packet contains the complete current set of holes at the file-receiver.
14	1	this packet contains incomplete hole-state; holes shown in this packet should supplement other incomplete hole-state known to the file-sender.

Bit 14 of a 'holestofill' STATUS packet is only set when there are too many holes to fit within a single STATUS packet due to MTU limitations. This causes the hole list to be spread out over multiple STATUS packets, each of which conveys distinct sets of holes. This could occur, for instance, in a large file `_put_` scenario with a long-delay feedback loop and poor physical layer conditions. These multiple STATUS packets will share In-Response-To information. When losses are light and/or hole reporting and repair is relatively frequent, all holes should easily fit within a single STATUS packet, and this flag will be clear. Bit 14 should normally be clear.

In some rare cases of high loss, there may be too many holes in the received data to convey within a single STATUS's size, which is limited by the link MTU size. In this case, multiple STATUS packets may be generated, and Flags bit 14 should be set on each STATUS packet accordingly, to indicate that each packet holds incomplete results. The complete group of STATUS packets, each containing incomplete information, will share common In-Response-To information to distinguish them from any earlier groups.

Bit	Value	Meaning
15	0	This STATUS was requested by the file-sender.
15	1	This STATUS is sent voluntarily.

Flag bit 15 indicates whether the STATUS is sent voluntarily or due to a request by the sender. It affects content of the In-Response-To timestamp and descriptor fields.

In the case of a transfer proceeding normally, immediately following the STATUS packet header shown above, is a set of "Hole" definitions indicating any lost packets. Each Hole definition is a pair of unsigned integers. For a 32-bit offset descriptor, each Hole definition consists of two four-octet unsigned integers:

Hole Definition Format

```

0          1          2          3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+
[          offset to start of hole (descriptor)          ]
+-----+-----+-----+-----+
[          offset to end of hole (descriptor)            ]
+-----+-----+-----+-----+

```

The start of the hole means the offset of the first unreceived byte in that hole. The end of the hole means the last unreceived byte in that hole.

For 16-bit descriptors, each Hole definition holds two two-octet unsigned integers, while Hole definitions for 64- and 128-bit descriptors require two eight- and two sixteen-octet unsigned integers respectively.

Holes MUST be listed in order, lowest values first.

Since each Hole definition takes up eight octets when 32-bit offset lengths are used, we expect that well over 100 such definitions can fit in a single STATUS packet, given the IPv6 minimum MTU. (There may be cases where there is a very constrained backchannel compared to the forward channel streaming DATA packets. For these cases, implementations might deliberately request large holes that span a number of smaller holes and intermediate areas where DATA has already been received, so that previously-received DATA is deliberately resent. This aggregation of separate holes keeps the backchannel STATUS packet size down to avoid backchannel congestion.)

A 'voluntary' STATUS can be sent at the start of each session. This indicates that the receiver is ready to receive the file, or indicates an error or rejection code, described below. A STATUS indicating a successfully established transfer has a Progress Indicator of zero and an In-Response-To field of zero.

On receiving a STATUS packet, the sender SHOULD prioritize sending the necessary data to fill those holes, in order to advance the Progress Indicator at the receiver.

4.5.1. Errors and aborting sessions

In the case of an error causing a session to be aborted, the Status field holds a code that can be used to explain the cause of the error to the other peer. A zero value indicates that there have been no significant errors (this is called a "success STATUS" within this document), while any non-zero value means the session should be aborted (this is called a "failure STATUS").

Error Code Status Value	Meaning
0x00	Success, No Errors.
0x01	Unspecified Error.
0x02	Unable to send file due to resource constraints.
0x03	Unable to receive file due to resource constraints.
0x04	File not found.
0x05	Access Denied.
0x06	Unknown Id field for session.
0x07	Did not delete file.
0x08	File length is longer than receiver can support.
0x09	File offset descriptors do not match expected use or file length.
0x0A	Unsupported Saratoga packet type received.
0x0B	Unsupported Request Type received.
0x0C	REQUEST is now terminated due to an internal timeout.
0x0D	DATA flag bits describing transfer have changed unexpectedly.
0x0E	Receiver is no longer interested in receiving this file.
0x0F	File is in use.
0x10	METADATA required before transfer can be accepted.
0x11	A STATUS error message has been received unexpectedly, so REQUEST is terminated.
0x12	Receiver did not hear from sender before timeout.

The recipient of a failure STATUS MUST NOT try to process the Progress Indicator, In-Response-To, or Hole offsets, because, in some types of error conditions, the packet's sender may not have any way of setting them to the right length for the session.

5. The Directory Entry

Directory Entries have two uses within Saratoga:

1. Within a METADATA packet, a Directory Entry is used to give information about the file being transferred, in order to facilitate proper reassembly of the file and to help the file-receiver understand how recently the file may have been created or modified.

2. When a peer requests a directory record via a `_getdir_ REQUEST`, the other peer generates a file containing a series of one or more concatenated Directory Entry records, and transfers this file as it would transfer the response to a normal `_get_ REQUEST`, sending the records together within DATA packets. This file may be either temporary or within-memory and not actually a part of the host's file system itself.

Directory Entry Format

```

      0               1               2               3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-----+-----+-----+-----+-----+-----+-----+-----+
|1|          Properties          [          Size (descriptor)          ]
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Mtime file modification time (using year 2000 epoch)      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|      Ctime file creation time (using year 2000 epoch)      |
+-----+-----+-----+-----+-----+-----+-----+-----+
|                                                                    /
+                                                                    /
/                                                                    /
/          File Path (max 1024 octets,variable length)          /
/                                                                    /
/                                                                    ... //
+-----+-----+-----+-----+-----+-----+-----+-----+

```

where

field	description
Properties	if set, bit 7 of this field indicates that the entry corresponds to a directory. Bit 6, if set, indicates that the file is "special". A special file may not be directly transferable as it corresponds to a symbolic link, a named pipe, a device node, or some other "special" filesystem object. A file-sender may simply choose not to include these types of files in the results of a <code>_getdir_</code> request. Bits 8 and 9 are flags that indicate the width of the following descriptor field that gives file size.
Size	the size of each file or directory in octets. This is a descriptor, varying as needed in each entry for the size of the file. For convenience in the figure, it is shown here as a 16-bit descriptor for a small file.
Mtime	a timestamp showing when the file or directory was modified.
Ctime	a timestamp of the last status change for this file or directory.
File Path	contains the file's name relative within the requested path of the <code>_getdir_</code> session, a maximum of 1024-octet UTF-8 string, which is null-terminated to indicate its end. The File Path may contain additional null padding in the null termination to allow Directory Entries to each be allocated a fixed amount of space or to place an integer number of Directory Entries in each DATA packet for debugging purposes.

The first bit of the Directory Entry is always 1, to indicate the start of the record and the end of any padding from previous Directory Entries.

Bit 6	Bit 7	Properties conveyed
0	0	normal file.
0	1	normal directory.
1	0	special file.
1	1	special directory.

Streams listed in a directory should be marked as special. If a stream is being transferred, its size is unknown -- otherwise it

would be a file. The size property of a Directory Entry for a stream is therefore expected to be zero.

Bit 8	Bit 9	Properties conveyed
0	0	File size is indicated in a 16-bit descriptor.
0	1	File size is indicated in a 32-bit descriptor.
1	0	File size is indicated in a 64-bit descriptor.
1	1	File size is indicated in a 128-bit descriptor.

Flag bits 8 and 9 of Properties are descriptor size flags, with similar meaning as before, describing the size of the File Size descriptor that follows the Properties field. When a single Directory Entry appears in the METADATA packet, these flags SHOULD match flag bits 8 and 9 in the METADATA header. (A smaller descriptor size may be indicated in the Directory Entry when doing test transfers of small files using large descriptors.)

Bit 10	Properties conveyed
0	Set to zero. Reserved for future use.

Bit	Use
13	
0	This file's content MUST be delivered reliably without errors using UDP.
1	This file's content MAY be delivered unreliably, or partly unreliably, where errors are tolerated, using UDP-Lite.

Bit 13 indicates whether the file must be sent reliably or can be sent at least partly unreliably, using UDP-Lite. This matches METADATA flag use.

Undefined or unused flag bits of the Properties field default to zero. Bit 0 is always 1, to indicate the start of a Directory Entry. In general, bits 1-7 of Properties are for matters related to the sender's filesystem, while bits 8-15 are for matters related to transport over Saratoga.

It may be reasonable that files are visible in Directory Entries only when they can be transferred to the requester - this may depend on

e.g. having appropriate access permissions or being able to handle large filesizes. But requesters only capable of handling small files MUST be able to skip through large descriptors for large file sizes. Directory sizes are not calculated or sent, and a Size of 0 is given instead for directories, which are considered zero-length files.

The "epoch" format used in file creation and modification timestamps in directory entries indicates the unsigned number of seconds since the start of January 1, 2000 in UTC. The times MUST include all leap seconds. Using unsigned 32-bit values means that these time fields will not wrap until after the year 2136.

Converting from unix CTime/MTime holding a time past January 1, 2000 but with the traditional 1970 epoch means subtracting the fixed value of 946 684 822 seconds, which includes the 22 leap seconds that were added to UTC between 1 January 1970 and 1 January 2000. A unix time before 2000 is rounded to January 1, 2000.

A file-receiver should preserve the timestamp information received in the METADATA for its own copy of the file, to allow newer versions of files to propagate and supercede older versions.

6. Behaviour of a Saratoga Peer

This section describes some details of Saratoga implementations and uses the RFC 2119 standards language to describe which portions are needed for interoperability.

6.1. Saratoga Sessions

Following are descriptions of the packet exchanges between two peers for each type of session. Exchanges rely on use of the Id field to match responses to requests, as described earlier in Section 4.2.

6.1.1. The `_get_` Session

1. A peer (the file-receiver) sends a REQUEST packet to its peer (the file-sender). The Flags bits are set to indicate that this is not a `_delete_` request, nor does the File Path indicate a directory. Each `_get_` session corresponds to a single file, and fetching multiple files requires sending multiple REQUEST packets and using multiple different Session Ids so that responses can be differentiated and matched to REQUESTs based on the Id field. If a specific file is being requested, then its name is filled into the File Path field, otherwise it is left null and the file-sender will send a file of its choice.

2. If the `_get_` request is rejected, then a STATUS packet containing an error code in the Status field is sent and the session is terminated. This STATUS packet MUST be sent to reject and terminate the session. The error code MAY make use of the "Unspecified Error" value for security reasons. Some REQUESTs might also be rejected for specifying files that are too large to have their lengths encoded within the maximum integer field width advertised by bits 8 and 9 of the REQUEST.
3. If the `_get_` request is accepted, then a STATUS packet MAY be sent with an error code of 00 and an In-Response-To field of zero, to indicate acceptance. Sending other packets (METADATA or DATA) also indicates acceptance. The file-sender SHOULD generate and send a METADATA packet. A METADATA packet that is received MUST be parsed. The sender MUST send the contents of the file or stream as a series of DATA packets. In the absence of STATUS packets being requested from the receiver, if the file-sender believes it has finished sending the file and is not on a unidirectional link, it MUST send the last DATA packet with the Flags bit set requesting a STATUS response from the file-receiver. The last DATA packet MUST always have its End of Data (EOD) bit set. This can be followed by empty DATA packets with the Flags bits set with EOD and requesting a STATUS until either a STATUS packet is received, or the inactivity timer expires. All of the DATA packets MUST use field widths for the file offset descriptor fields that match what the Flags of the METADATA packet specified. Some arbitrarily selected DATA packets may have the Flags bit set that requests a STATUS packet. The file-receiver MAY voluntarily send STATUS packets at other times, where the In-Response-To field MUST set to zero. The file-receiver SHOULD voluntarily send a STATUS packet in response to the first DATA packet.
4. As the file-receiver takes in the DATA packets, it writes them into the file locally. The file-receiver keeps track of missing data in a hole list. Periodically the file sender will set the ack flag bit in a DATA packet and request a STATUS packet from the file-receiver. The STATUS packet can include a copy of this hole list if there are holes. File-receivers MUST send a STATUS packet immediately in response to receiving a DATA packet with the Flags bit set requesting a STATUS.
5. If the file-sender receives a STATUS packet with a non-zero number of holes, it re-fetches the file data at the specified offsets and re-transmits it. If the METADATA packet has not been received, this is indicated by a bit in the STATUS packet, and the METADATA packet can be retransmitted. The file-sender MUST

retransmit data from any holes reported by the file-receiver before proceeding further with new DATA packets.

6. When the file-receiver has fully received the file data and any METADATA packet, then it sends a STATUS packet indicating that the session is complete, and it terminates the session locally, although it MUST persist in responding to any further DATA packets received from the file-sender with 'completed' STATUSes, as described in Section 4.5, for some reasonable amount of time. Starting a timer on sending a completed STATUS and resetting it whenever a received DATA/sent 'completed' STATUS session takes place, then removing all session state on timer expiry, is one approach to this.

Given that there may be a high degree of asymmetry in link bandwidth between the file-sender and file-receiver, the STATUS packets should be carefully generated so as to not congest the feedback path. This means that both a file-sender should be cautious in setting the DATA Flags bit requesting STATUSes, and also that a file-receiver should be cautious in gratuitously generating STATUS packets of its own volition. When sending on known unidirectional links, a file-sender cannot reasonably expect to receive STATUS packets, so should never request them.

6.1.2. The `_getdir_` Session

A `_getdir_` session to obtain a Directory Record proceeds through the same states as the `_get_` session. Rather than transferring the contents of a file from the file-receiver to the file-sender, a set of records representing the contents of a directory are transferred as a file. These records can be parsed and dealt with by the file-receiver as desired. There is no requirement that a Saratoga peer send the full contents of a directory listing; a peer may filter the results to only those entries that are actually accessible to the requesting peer.

Any file system entries that would normally be contained in the directory records, but that have sizes greater than the receiver has indicated that it can support in its BEACON, MUST be filtered out.

6.1.3. The `_delete_` Session

1. A peer sends a REQUEST packet with the bit set indicating that it is a deletion request and the path to be deleted is filled into the File Path field. The File Path MUST be filled in for `_delete_` sessions, unlike for `_get_` sessions.

2. The other peer replies with a feedback STATUS packet whose Id matches the Id field of the `_delete_ REQUEST`. This STATUS has a Status code that indicates that the file is not currently present on the filesystem (indicated by the 00 Status field in a success STATUS), or whether some error occurred (indicated by the non-zero Status field in a failure STATUS). This STATUS packet MUST have no Holes and 16-bit width zero-valued Progress Indicator and In-Response-To fields.

If a request is received to delete a file that is already deleted, a STATUS with Status code 00 and other fields as described above is sent back in acknowledgement. This response indicates that the indicated file is not present, not the exact action sequence that led to a not-present file. This idempotent behaviour ensures that loss of STATUS acknowledgements and repeated `_delete_` requests are handled properly.

6.1.4. The `_put_` Session

A `_put_` session proceeds as a `_get_` does, except the file-sender and file-receiver roles are exchanged between peers. In a `_put_` a PUT REQUEST is sent.

However, in a 'blind `_put_`', no REQUEST packet is ever sent. The file-sending end senses that the session is in progress when it receives METADATA or DATA packets for which it has no knowledge of the Id field.

If the file-receiver decides that it will store and handle the `_put_` request (at least provisionally), then it MUST send a voluntary (ie, not requested) success STATUS packet to the file-sender. Otherwise, it sends a failure STATUS packet. After sending a failure STATUS packet, it may ignore future packets with the same Id field from the file-sender, but it should, at a low rate, periodically regenerate the failure STATUS packet if the flow of packets does not stop.

6.2. Beacons

Sending BEACON packets is not required in any of the sessions discussed in this specification, but optional BEACONS can provide useful information in many situations. If a node periodically generates BEACON packets, then it should do so at a low rate which does not significantly affect in-progress data transfers.

A node that supports multiple versions of Saratoga (e.g. version 1 from this specification along with the older version 0), MAY send multiple BEACON packets showing different version numbers. The version number in a single BEACON should not be used to infer the

larger set of protocol versions that a peer is compatible with. Similarly, a node capable of communicating via IPv4 and IPv6 MAY send separate BEACONS via both protocols, or MAY only send BEACONS on its preferred protocol.

If a node receives BEACONS from a peer, then it SHOULD NOT attempt to start any `_get_`, `_getdir_`, or `_delete_` sessions with that peer if bit 14 is not set in the latest received BEACONS. Likewise, if received BEACONS from a peer do not have bit 15 set, then `_put_` sessions SHOULD NOT be attempted to that peer. Unlike the capabilities bits which prevent certain types of sessions from being attempted, the willingness bits are advisory, and sessions MAY be attempted even if the node is not advertising a willingness, as long as it advertises a capability. This avoids waiting for a willingness indication across long-delay links.

6.3. Upper-Layer Interface

No particular application interface functionality is required in implementations of this specification. The means and degree of access to Saratoga configuration settings, and session control that is offered to upper layers and applications, are completely implementation-dependent. In general, it is expected that upper layers (or users) can set timeout values for session requests and for inactivity periods during the session, on a per-peer or per-session basis, but in some implementations where the Saratoga code is restricted to run only over certain interfaces with well-understood operational latency bounds, then these timers MAY be hard-coded.

6.4. Inactivity Timer

In order to determine the liveliness of a session, Saratoga nodes may implement an inactivity timer for each peer they are expecting to see packets from. For each packet received from a peer, its associated inactivity timer is reset. If no packets are received for some amount of time, and the inactivity timer expires, this serves as a signal to the node that it should abort (and optionally retry) any sessions that were in progress with the peer. Information from the link interface (i.e. link down) can override this timer for point-to-point links.

The actual length of time that the inactivity timer runs for is a matter of both implementation and deployment situation. Relatively short timers (on the order of several round-trip times) allow nodes to quickly react to loss of contact, while longer timers allow for session robustness in the presence of transient link problems. This document deliberately does not specify a particular inactivity timer value nor any rules for setting the inactivity timer, because the

protocol is intended to be used in both long- and short-delay regimes.

Specifically, the inactivity timer is started on sending REQUEST or STATUS packets. When sending packets not expected to elicit responses (BEACON, METADATA, or DATA without acknowledgement requests), there is no point to starting the local inactivity timer.

For normal file transfers, there are simple rules for handling expiration of the inactivity timer during a `_get_` or `_put_` session. Once the timer expires, the file-sender SHOULD terminate the session state and cease to send DATA or METADATA packets. The file-receiver SHOULD stop sending STATUS packets, and MAY choose to store the file in some cache location so that the transfer can be recovered. This is possible by waiting for an opportunity to re-attempt the session and immediately sending a STATUS that only lists the parts of the file not yet received if the session is granted. In any case, a partially-received file MUST NOT be handled in any way that would allow another application to think it is complete.

The file-sender may implement more complex timers to allow rate-based pacing or simple congestion control using information provided in STATUS packets, but such possible timers and their effects are deliberately not specified here.

6.5. Streams and wrapping

When sending an indefinite-length stream, the possibility of offset sequence numbers wrapping back to zero must be considered. This can be protected against by using large offsets, and by the stream receiver. The receiver MUST separate out holes before the offset wraps to zero from holes after the wrap, and send Hole definitions in different STATUS packets, with Flag 14 set to mark them as incomplete. Any Hole straddling a sequence wrap MUST be broken into two separate Holes, with the second Hole starting at zero. The timestamps in STATUS packets carrying any pre-wrap holes should be earlier than the timestamp in later packets, and should repeat the timestamp of the last DATA packet seen for that offset sequence before the following wrap to zero occurred. Receivers indicate that they no longer wish to receive streams by sending Status Code 0C.

6.6. Completing file delivery and ending the session

The sender infers a completely-received transfer from the reported receiver window position. In the final STATUS packet sent by the receiver once the file to be transferred has been completely received, bit 14 MUST be 0 (indicating a complete set of holes in this packet), there MUST NOT be any `holestofill` offset pairs

indicating holes, the In-Response-To and Progress Indicator fields contain the length of the file (i.e. point to the next octet after the file), and the voluntary flag MUST be set. This 'completed' STATUS may be repeated, depending on subsequent sender behaviour, while internal state about the transfer remains available to the receiver.

Because METADATA not mandatory for implementations, the file receiver may not know the length of a file if METADATA is never sent. The sender MUST set the EOD End of Data flag in each DATA packet that sends the last byte of the file, and SHOULD request a STATUS acknowledgement when the EOD flag is set. If METADATA has been sent and the EOD comes earlier than a previously reported length of a file, an unspecified error 0x01, as described below, is returned in the STATUS message responding to that DATA packet and EOD flag. If a stream is being marked EOD, the receiver acknowledges this with a Success 0x00 code.

7. Implementation Development

There is a mailing list for discussion of Saratoga and its implementations. Contact Lloyd Wood for details. Further information on the Saratoga protocol is at:
<http://saratoga.sourceforge.net/>

8. Security Considerations

The design of Saratoga provides limited, deliberately lightweight, services for authentication of session requests, and for authentication or encryption of data files via keyed metadata checksums. This document does not specify privacy or access control for data files transferred. Privacy, access, authentication and encryption issues may be addressed within an implementation or deployment in several ways that do not affect the file transfer protocol itself. As examples, IPsec may be used to protect Saratoga implementations from forged packets, to provide privacy, or to authenticate the identity of a peer. Other implementation-specific or configuration-specific mechanisms and policies might also be employed for authentication and authorization of requests. Protection of file data and meta-data can also be provided by a higher-level file encryption facility. If IPsec is not required, use of encryption before the file is given to Saratoga is preferable.

Basic security practices like not accepting paths with "..", not following symbolic links, and using a chroot() system call, among others, should also be considered within an implementation.

Note that Saratoga is intended for single-hop transfers between peers. A METADATA checksum using a previously shared key can be used to decrypt or authenticate delivered DATA files. Saratoga can only provide payload encryption across a single Saratoga transfer, not end-to-end across concatenated separate hop-by-hop transfers through untrusted peers, as checksum verification of file integrity is required at each node. End-to-end data encryption, if required, MUST be implemented by the application using Saratoga.

9. IANA Considerations

IANA has allocated port 7542 (tcp/udp) for use by Saratoga.

saratoga	7542/tcp	Saratoga Transfer Protocol
saratoga	7542/udp	Saratoga Transfer Protocol

IANA has allocated a dedicated IPv4 all-hosts multicast address (224.0.0.108) and a dedicated IPv6 link-local multicast addresses (FF02:0:0:0:0:0:0:6c) for use by Saratoga.

10. Acknowledgements

Developing and deploying the on-orbit IP-based infrastructure of the Disaster Monitoring Constellation, in which Saratoga has proven useful, has taken the efforts of hundreds of people over more than a decade. We thank them all.

We thank James H. McKim as an early contributor to Saratoga implementations and specifications, while working for RSIS Information Systems at NASA Glenn. We regard Jim as an author of this document, but are prevented by the boilerplate five-author limit from naming him earlier.

We thank Stewart Bryant, Dale Mellor, Cathryn Peoples, Kerrin Pine, Abu Zafar Shahriar and Dave Stewart for their review comments.

Work on this specification at NASA's Glenn Research Center was funded by NASA's Earth Science Technology Office (ESTO).

11. A Note on Naming

Saratoga is named for the USS Saratoga (CV-3), the aircraft carrier sunk at Bikini Atoll that is now a popular diving site.

12. References

12.1. Normative References

- [RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <<https://www.rfc-editor.org/info/rfc768>>.
- [RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992, <<https://www.rfc-editor.org/info/rfc1321>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3309] Stone, J., Stewart, R., and D. Otis, "Stream Control Transmission Protocol (SCTP) Checksum Change", RFC 3309, DOI 10.17487/RFC3309, September 2002, <<https://www.rfc-editor.org/info/rfc3309>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.

12.2. Informative References

- [Brenchley12] Brenchley, M., Garner, P., Cawthorne, A., Wisniewska, K., and P. Davies, "Bridging the Abyss - Agile Data Downlink Solutions for the Disaster Monitoring Constellation", Small Satellites Systems and Services (4S) Symposium, European Space Agency, Portoroz, Slovenia, June 2012.
- [Hogie05] Hogie, K., Criscuolo, E., and R. Parise, "Using Standard Internet Protocols and Applications in Space", Computer Networks, Special Issue on Interplanetary Internet, vol. 47, no. 5, pp. 603-650, April 2005.
- [I-D.wood-tsvwg-saratoga-congestion-control] Wood, L., Eddy, W., and W. Ivancic, "Congestion control for the Saratoga protocol", draft-wood-tsvwg-saratoga-congestion-control-12 (work in progress) , December 2017.

- [Jackson04] Jackson, C., "Saratoga File Transfer Protocol", Surrey Satellite Technology Ltd internal technical document , 2004.
- [RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, DOI 10.17487/RFC0959, October 1985, <<https://www.rfc-editor.org/info/rfc959>>.
- [RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., Ed., and G. Fairhurst, Ed., "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, DOI 10.17487/RFC3828, July 2004, <<https://www.rfc-editor.org/info/rfc3828>>.
- [RFC5348] Floyd, S., Handley, M., Padhye, J., and J. Widmer, "TCP Friendly Rate Control (TFRC): Protocol Specification", RFC 5348, DOI 10.17487/RFC5348, September 2008, <<https://www.rfc-editor.org/info/rfc5348>>.
- [RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers", RFC 5405, DOI 10.17487/RFC5405, November 2008, <<https://www.rfc-editor.org/info/rfc5405>>.
- [RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <<https://www.rfc-editor.org/info/rfc6151>>.
- [Wood07a] Wood, L., Ivancic, W., Hodgson, D., Miller, E., Conner, B., Lynch, S., Jackson, C., da Silva Curiel, A., Cooke, D., Shell, D., Walke, J., and D. Stewart, "Using Internet Nodes and Routers Onboard Satellites", International Journal of Satellite Communications and Networking, Special Issue on Space Networks, vol. 25, no. 2, pp. 195-216, March/April 2007.
- [Wood07b] Wood, L., Eddy, W., Ivancic, W., Miller, E., McKim, J., and C. Jackson, "Saratoga: a Delay-Tolerant Networking convergence layer with efficient link utilization", International Workshop on Satellite and Space Communications (IWSSC '07) Salzburg, September 2007.
- [Wood11] Wood, L., Smith, C., Eddy, W., Ivancic, W., and C. Jackson, "Taking Saratoga from space-based ground sensors to ground-based space sensors", IEEE Aerospace Conference Big Sky, Montana, March 2011.

Appendix A. Timestamp/Nonce field considerations

Timestamps are useful in DATA packets when the time that the packet or its payload was generated is of importance; this can be necessary when streaming sensor data recorded and packetized in real time. The format of the optional timestamp, whose presence is indicated by a flag bit, is implementation-dependent within the available fixed-length 128-bit field. How the contents of this timestamp field are used and interpreted depends on local needs and conventions and the local implementation.

However, one simple suggested format for timestamps is to begin with a POSIX `time_t` representation of time, in network byte order. This is either a 32-bit or 64-bit signed integer representing the number of seconds since 1970. The remainder of this field can be used either for a representation of elapsed time within the current second, if that level of accuracy is required, or as a nonce field uniquely identifying the packet or including other information. Any locally-meaningful flags identifying a type of timestamp or timebase can be included before the end of the field. Unused parts of this field MUST be set to zero.

There are many different representations of timestamps and timebases, and this draft is too short to cover them in detail. One suggested flag representation of different timestamp fields is to use the least significant bits at the end of the timestamp/nonce field as:

Status Value	Meaning
00	No flags set, local interpretation of field.
01	32-bit POSIX timestamp at start of field indicating whole seconds from epoch.
02	64-bit POSIX timestamp at start of field indicating whole seconds elapsed from epoch.
03	32-bit POSIX timestamp, as in 01, followed by 32-bit timestamp indicating fraction of the second elapsed.
04	64-bit POSIX timestamp, as in 02, followed by 32-bit timestamp indicating fraction of the second elapsed.
05	32-bit timestamp giving seconds elapsed since the 2000 epoch, as in file timestamps. This option is likely only useful for very slow links.

Other values may indicate specific epochs or timebases, as local requirements dictate. There are many ways to define and use time usefully.

Echoing timestamps back to the file-sender is also useful for tracking flow conditions. This does not require the echoing receiver to understand the timestamp format or values in use. The use of timestamp values may assist in developing algorithms for flow control (including TCP-Friendly Rate Control [I-D.wood-tsvwg-saratoga-congestion-control]) or other purposes. Timestamp values provide a useful mechanism for Saratoga peers to measure path and round-trip latency.

Authors' Addresses

Lloyd Wood
University of Surrey alumni
Sydney, New South Wales
Australia

Email: lloydwood@users.sourceforge.net

Wesley M. Eddy
MTI Systems
MS 500-ASRC
NASA Glenn Research Center
21000 Brookpark Road
Cleveland, OH 44135
USA

Phone: +1-216-433-6682
Email: wes@mti-systems.com

Charles Smith
Vallona Networks
7 Wattle Crescent
Phegans Bay, New South Wales 2256
Australia

Phone: +61-404-05-8974
Email: charlesetsmith@me.com

Will Ivancic
Syzygy Engineering LLC
Westlake, OH 44145
USA

Phone: +1-440-835-8448
Email: ivancic@syzygyengineering.com

Chris Jackson
Surrey Satellite Technology Ltd
Tycho House
Surrey Space Centre
20 Stephenson Road
Guildford, Surrey GU2 7YE
United Kingdom

Phone: +44-1483-803803
Email: C.Jackson@sstl.co.uk