Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: April 8, 2017

G. Fairhurst
T. Jones
University of Aberdeen

October 05, 2016

Features of the User Datagram Protocol (UDP) and Lightweight UDP (UDP-Lite) Transport Protocols

draft-fairhurst-taps-transports-usage-udp-03

Abstract

This document describes how the User Datagram Protocol (UDP) and the Lightweight User Datagram Protocol (UDP-Lite) transport protocols expose services to applications and how an application can configure and use the features offered by the transport service. The document is intended as a contribution to the Transport Services (TAPS) working group to assist in analysis of the UDP and UDP-Lite transport interface.

Status of This Memo

Copyright Notice

Table of Contents

1. Terminology

This document uses common terminology defined in [I-D.ietf-taps-transports-usage]. This document also refers to the terminology of [RFC2119], but does not itself define new terms using this terminology.

2. Introduction

This document presents defined interactions between transport protocols and applications in the form of 'primitives' (function calls). Primitives can be invoked by an application or a transport protocol; the latter type is called an "event". The list of transport service features and primitives in this document is strictly based on the parts of protocol specifications that relate to what the protocol provides to an application using it and how the application interacts with it. It does not cover parts of a protocol that are explicitly stated as optional to implement.

This follows the methodology defined in [I-D.ietf-taps-transports-usage], specifically it provides the first pass of this process. It discusses the relevant RFC text describing primitives for each protocol. This also provides documentation that may help users of UDP and UDP-Lite.

3. UDP and UDP-Lite Primitives

This summarizes the relevant text parts of the RFCs describing the UDP and UDP-Lite protocols, focusing on what the transport protocols provide to the application and how the transport is used (based on abstract API descriptions, where they are available).

3.1. Primitives Provided by UDP

The User Datagram Protocol (UDP) [RFC0768] States: "This User Datagram Protocol (UDP) is defined to make available a datagram mode of packet-switched computer communication in the environment of an interconnected set of computer networks." It "provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism (..)".

The User Interface section of [RFC0768] specifies that the user interface to an application should be able to create receive ports, source and destination ports and addresses, and provide operations to receive data based on ports with an indication of source port and address. Operations should be provided that allows datagrams be sent specifying the source and destination ports and addresses to be sent.

UDP for IPv6 is defined by [RFC2460], and API extensions to support this in [RFC3493]. [RFC6935] and [RFC6936] defines an update to the UDP transport specified in RFC 2460. This enables use of a zero UDP checksum mode with a tunnel protocol, providing that the method satisfies the requirements in [RFC6936].

UDP offers only a basic transport interface. UDP datagrams may be directly sent and received, without exchanging messages between the endpoints to setup a connection (i.e., there is no handshake prior to communication). Using the sockets API, applications can receive packets from more than one IP source address on a single UDP socket. Common support allows specification of the local IP address, destination IP address, local port and destination port values. Any or all of these can be indicated, with defaults supplied by the local system when these are not specified. The local endpoint is set using the BIND call and set on the remote endpoint using the CONNECT call. The CLOSE function has local significance only. This does not impact the status of the remote endpoint.

UDP and UDP-Lite do not provide congestion control, retransmission, nor support to optimise fragmentation etc. This means that applications using UDP need to provide additional functions on top of the UDP transport API. This requires parameters to be passed through the API to control the network layer (IPv4 or IPv6). These additional primitives could be considered a part of the network layer (e.g., control of the setting of the Don’t Fragment flag on a transmitted datagram), but are nonetheless essential to allow a user of the UDP API to implement functions that are normally associated with the transport layer (such as probing for Path maximum transmission size). Although this adds complexity to the analysis of the API, this document includes such primitives.

[I-D.ietf-tsvwg-rfc5405bis] also states "many operating systems also allow a UDP socket to be connected, i.e., to bind a UDP socket to a specific pair of addresses and ports. This is similar to the corresponding TCP sockets API functionality. However, for UDP, this is only a local operation that serves to simplify the local send/ receive functions and to filter the traffic for the specified addresses and ports. Binding a UDP socket does not establish a connection - UDP does not notify the remote end when a local UDP socket is bound. Binding a socket also allows configuring options that affect the UDP or IP layers, for example, use of the UDP checksum or the IP Timestamp option. On some stacks, a bound socket also allows an application to be notified when ICMP error messages are received for its transmissions [RFC1122]."

The [POSIX] API offers mechanisms for an application to receive asynchronous data events at the socket layer. Calls such as poll, select or queue allow an application to be notified when data has arrived at a socket or a socket has flushed its buffers. It is possible to structure a callback-driven API to the network interface on top of these calls. There are protocols that allow a macro interface to network primitives, [RFC6458] describes implicit association setup for sending datagram messages using SCTP. Implicit connection setup allows an application to delegate connection life management to the transport API. The transport API uses protocol primitives to offer the automated service to the application via the socket API. By combining UDP primitives (CONNECT.UDP, SEND.UDP), a higher level API could offer a similar service.

Guidance on the use of services provided by UDP is provided in [I-D.ietf-tsvwg-rfc5405bis].

The following primitives are specified:

CONNECT: The CONNECT primitive allows the association of source and port sets to a socket to enable creation of a ’connection’ for UDP traffic. This UDP connection allows an application to be notified of errors received from the network stack and provides a shorthand access to the send and receive primitives. Since UDP is itself connectionless, no datagrams are sent because this primitive is executed. A further connect call can be used to change the association to a source/port pair.

Two forms of usage may be identified for the CONNECT primitive:

1. bind(): A bind operation sets the local port, either implicitly, triggered by a send to operation on an unbound, unconnected socket using an ephemeral port. Or by an explicit bind to makes use of a configured or well-known port.

2. bind(); connect(): A bind operation followed by a CONNECT primitive. The bind operation establishes the use of a known local port for datagrams, rather than using an ephemeral port. The connect operation specifies a known address port combination to be used by default for future datagrams. This form is used either after receiving a datagram from an endpoint causing the creation of a connection or can be triggered by third party configuration or a protocol trigger (such as reception of a UDP Service Description Protocol, SDP [RFC4566], record).

LISTEN: The roles of a client and a server are often not appropriate for UDP, where connections can be peer-to-peer. The listening functions are performed using one of the forms of CONNECT primitive described above.

SEND: The SEND primitive hands over a provided number of bytes that UDP should send to the other side of a UDP connection in a UDP datagram. The primitive can be used by an application to directly send datagrams to an endpoint defined by an address/port pair. If a connection has been created, then the address/port pair is inferred from the current connection for the socket. A connection created on the socket will allow network errors to be returned to the application as a notification on the send primitive. Messages passed to the send primitive that cannot be sent atomically in a datagram will not be sent by the network layer, generating an error.

RECEIVE: The RECEIVE primitive allocates a receiving buffer to accommodate a received datagram. The primitive returns the number of bytes provided from a received UDP datagram. Section 4.1.3.5 of [RFC1122] states "When a UDP datagram is received, its specific-destination address MUST be passed up to the application layer."

DISABLE_CHECKSUM: The CHECKSUM function controls whether a sender disables the UDP checksum when sending datagrams. [RFC0768] and IPv6 [RFC6935] [RFC6936] [I-D.ietf-tsvwg-rfc5405bis]. When set it overrides the default UDP behaviour disabling the checksum on sending. Section 4.1.3.4 of [RFC1122] states "An application MAY optionally be able to control whether a UDP checksum will be generated, but it MUST default to checksumming on."

REQUIRE_CHECKSUM: The REQUIRE_CHECKSUM function determines whether UDP datagrams received with a zero checksum are permitted or discarded. Section 4.1.3.4 of [RFC1122] states "An application MAY optionally be able to control whether UDP datagrams without checksums should be discarded or passed to the application." Section 3.1 of [RFC3828] requires that the checksum field is non-zero, and hence UDP-Lite need to discard all datagrams received with a zero checksum.

SET_IP_OPTIONS: The SET_IP_OPTIONS function enables a datagram to be sent with the specified IP options. Section 4.1.3.2 of[RFC1122] states that an "application MUST be able to specify IP options to be sent in its UDP datagrams, and UDP MUST pass these options to the IP layer."

GET_IP_OPTIONS: The GET_IP_OPTIONS function is a network-layer function that enables a receiver to read the IP options of a received datagram. Section 4.1.3.2 of[RFC1122] states that a UDP receiver "MUST pass any IP option that it receives from the IP layer transparently to the application layer".

SET_DF: The SET_DF function is a network-layer function that sets the Don’t Fragment (DF) flag to be used in the field of an IP header of a packet that carries a UDP datagram. A UDP application should implement a method that avoids IP fragmentation (section 4 of [I-D.ietf-tsvwg-rfc5405bis]). It can use Packetization-Layer-Path MTU Discovery (PLPMTUD) [RFC4821] or Path MTU Discovery [RFC1191]. NOTE: In many other IETF transports (e.g. TCP) the transport provides the support needed to use DF, when using UDP, the application is responsible for the techniques needed to discover the path MTU, coordinating with the network layer.

GET_INTERFACE_MTU: The GET_INTERFACE_MTU function a network-layer function that indicates the largest unfragmented IP packet that may be sent. A UDP endpoint can subtract the size of all network and transport headers to determine the maximum size of unfragmented UDP payload. UDP applications should use this value as part of a method to avoid sending UDP datagrams that would result in IP packets that exceed the effective path maximum transmission unit (PMTU) allowed on the network path. The effective PMTU specified in Section 1 of [RFC1191] is equivalent to the "effective MTU for sending" specified in [RFC1122]. [RFC4821] states: "If PLPMTUD updates the MTU for a particular path, all Packetization Layer sessions that share the path representation (as described in Section 5.2) SHOULD be notified to make use of the new MTU and make the required congestion control adjustments."

SET_TTL: The SET_TTL function a network-layer function that sets the hop limit (TTL field) to be used in the field of an IPv4 header of a packet that carries an UDP datagram. This is used to limit the scope of unicast datagrams. Section 3.2.2.4 of [RFC1122] states an "incoming Time Exceeded message MUST be passed to the transport layer".

GET_TTL: The GET_TTL function is a network-layer function that reads the value of the TTL field from the IPv4 header of a received UDP datagram. Section 3.2.2.4 of [RFC1122] states that a UDP receiver "MAY pass the received TOS up to the application layer" When used for applications such as the Generalized TTL Security Mechanism (GTSM) [RFC5082], this needs the UDP receiver API to pass the received value of this field to the application.

SET_IPV6_UNICAST_HOPS: The SET_IPV6_UNICAST_HOPS function is a network-layer function that sets the hop limit field to be used in the field of an IPv6 header of a packet that carries a UDP datagram. For IPv6 unicast datagrams, this is functionally equivalent to the SET_TTL IPv4 function.

GET_IPV6_UNICAST_HOPS: The GET_IPV6_UNICAST_HOPS function is a network-layer function that reads the value from the hop count field in the IPv6 header from the IP header information of a received UDP datagram. For IPv6 unicast datagrams, this is functionally equivalent to the GET_TTL IPv4 function.

SET_DSCP: The SET_DSCP function is a network-layer function that sets the DSCP (or legacy TOS) value to be used in the field of an IP header of a packet that carries a UDP Datagram. Section 2.4 of [RFC1122] states that "Applications MUST select appropriate TOS values when they invoke transport layer services, and these values MUST be configurable.". The application should be able to change the TOS during the connection lifetime, and the TOS value should be passed to the IP layer unchanged. Section 4.1.4 of [RFC1122] also states that on reception the "UDP MAY pass the received TOS value up to the application layer". [RFC2475] [RFC3260] replaces this field in the IP Header assigning the six most significant bits to carry the Differentiated Services Code Point (DSCP) field. Preserving the intention of [RFC1122] to allow the application to specify the "Type of Service", this should be interpreted to mean that an API should allow the application to set the DSCP. Section 3.1.6 of [I-D.ietf-tsvwg-rfc5405bis] describes the way UDP applications should use this field. Normally a UDP socket will assign a single DSCP value to all Datagrams in a flow, but it is allowed to use different DSCP values for datagrams within the same flow in some cases, as described in [I-D.ietf-tsvwg-rfc5405bis]. Guidelines for WebRTC that illustrate this use are provided in [RFC7657].

SET_ECN: The SET_ECN function is a network-layer function that sets the ECN field in the IP Header of a UDP Datagram. When use of the TOS field was redefined [RFC3260], 2 bits of the field were assigned to support Explicit Congestion Notification (ECN) [RFC3168]. Section 3.1.5 [I-D.ietf-tsvwg-rfc5405bis] describes the way UDP applications should use this field. NOTE: In many other IETF transports (e.g. TCP) the transport provides the support needed to use ECN, when using UDP, the application itself is responsible for the techniques needed to use ECN.

GET_ECN: The GET_ECN function is a network-layer function that returns the value of the ECN field in the IP Header of a received UDP Datagram. Section 3.1.5 [I-D.ietf-tsvwg-rfc5405bis] states that a UDP receiver "MUST check the ECN field at the receiver for each UDP datagram that it receives on this port", requiring the UDP receiver API to pass to pass the received ECN field up to the application layer to enable appropriate congestion feedback.

ERROR_REPORT The ERROR_REPORT event informs an application of "soft errors", including the arrival of an ICMP or ICMPv6 error message. Section 4.1.4 of [RFC1122] states "UDP MUST pass to the application layer all ICMP error messages that it receives from the IP layer." For example, this event is required to implement ICMP-based Path MTU Discovery [RFC1191] [RFC1981].

CLOSE: The close primitive closes a connection. No further datagrams may be sent/received. Since UDP is itself connectionless, no datagrams are sent because this command is executed.

3.1.1. Excluded Primitives

Section 3.4 of [RFC1122] also describes "GET_MAXSIZES: - replaced, GET_SRCADDR (Section 3.3.4.3) and ADVISE_DELIVPROB:". These mechanisms are no longer used. It also specifies use of the Source Quench ICMP message, which has since been deprecated [RFC6633]. The IPV6_V6ONLY function defined in Section 5.3 of [RFC3493] restricts the use of information from the name resolver to only allow communication of AF_INET6 sockets to use IPv6 only. This is not considered part of the transport service.

3.2. Primitives Provided by UDP-Lite

The Lightweight User Datagram Protocol (UDP-Lite) [RFC3828] provides similar services to UDP. It changed the semantics of the UDP "payload length" field to that of a "checksum coverage length" field. UDP-Lite requires the pseudo-header checksum to be computed at the sender and checked at a receiver. Apart from the length and coverage changes, UDP-Lite is semantically identical to UDP.

The sending interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that communicates the checksum coverage length. This specifies the intended checksum coverage, with the remaining unprotected part of the payload called the "error-insensitive part".

The receiving interface of UDP-Lite differs from that of UDP by the addition of a single (socket) option that specifies the minimum acceptable checksum coverage.

The UDP-Lite Management Information Base (MIB) further defines the checksum coverage method [RFC5097]. Guidance on the use of services provided by UDP-Lite is provided in [I-D.ietf-tsvwg-rfc5405bis].

UDP-Lite requires use of the UDP or UDP-Lite checksum, and hence it is not permitted to use the "DISABLE_CHECKSUM:" function to disable use of a checksum, nor is it possible to disable receiver checksum processing using the "REQUIRE_CHECKSUM:" function . All other primitives and functions for UDP are permitted.

In addition, the following are defined:

SET_CHECKSUM_COVERAGE: The SET_CHECKSUM_COVERAGE function sets the coverage area for a sent datagram. UDP-Lite traffic uses this primitive to set the coverage length provided by the UDP checksum. Section 3.3 of [RFC5097] states that "Applications that wish to define the payload as partially insensitive to bit errors ... Should do this by an explicit system call on the sender side." The default is to provide the same coverage as for UDP.

SET_MIN_COVERAGE The SET_MIN_COVERAGE function sets the minimum a acceptable coverage protection for received datagrams. UDP-Lite traffic uses this primitive to set the coverage length that is checked on receive (section 1.1 of [RFC5097] describes the corresponding MIB entry as udpliteEndpointMinCoverage). Section 3.3 of [RFC3828] states that "applications that wish to receive payloads that were only partially covered by a checksum should inform the receiving system by an explicit system call".

The default is to require only minimal coverage of the datagram payload.

4. Acknowledgements

This work was partially funded by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s). Thanks to all who have commented or contributed, including Joe Touch, Ted Hardie, Aaron Falk.

5. IANA Considerations

This memo includes no request to IANA.

If there are no requirements for IANA, the section will be removed during conversion into an RFC by the RFC Editor.

6. Security Considerations

Security considerations for the use of UDP and UDP-Lite are provided in the referenced RFCs. Security guidance for application usage is provide in the UDP-Guidelines [I-D.ietf-tsvwg-rfc5405bis].

7. References

7.1. Normative References

[I-D.ietf-taps-transports-usage]
Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Service Features Provided by IETF Transport Protocols", draft-ietf-taps-transports-usage-01 (work in progress), July 2016.

[I-D.ietf-tsvwg-rfc5405bis]
Eggert, L., Fairhurst, G., and G. Shepherd, "UDP Usage Guidelines", draft-ietf-tsvwg-rfc5405bis-07 (work in progress), November 2015.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <http://www.rfc-editor.org/info/rfc768>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <http://www.rfc-editor.org/info/rfc2460>.

[RFC2553] Gilligan, R., Thomson, S., Bound, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2553, DOI 10.17487/RFC2553, March 1999, <http://www.rfc-editor.org/info/rfc2553>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, DOI 10.17487/RFC3493, February 2003, <http://www.rfc-editor.org/info/rfc3493>.

[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, <http://www.rfc-editor.org/info/rfc3828>.

[RFC6935] Eubanks, M., Chimento, P., and M. Westerlund, "IPv6 and UDP Checksums for Tunneled Packets", RFC 6935, DOI 10.17487/RFC6935, April 2013, <http://www.rfc-editor.org/info/rfc6935>.

7.2. Informative References

[POSIX] "IEEE Std. 1003.1-2001, , "Standard for Information Technology - Portable Operating System Interface (POSIX)", Open Group Technical Standard: Base Specifications Issue 6, ISO/IEC 9945:2002", December 2001.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <http://www.rfc-editor.org/info/rfc1191>.

[RFC1981] McCann, J., Deering, S., and J. Mogul, "Path MTU Discovery for IP version 6", RFC 1981, DOI 10.17487/RFC1981, August 1996, <http://www.rfc-editor.org/info/rfc1981>.

[RFC2475] Blake, S., Black, D., Carlson, M., Davies, E., Wang, Z., and W. Weiss, "An Architecture for Differentiated Services", RFC 2475, DOI 10.17487/RFC2475, December 1998, <http://www.rfc-editor.org/info/rfc2475>.

[RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <http://www.rfc-editor.org/info/rfc3260>.

[RFC3678] Thaler, D., Fenner, B., and B. Quinn, "Socket Interface Extensions for Multicast Source Filters", RFC 3678, DOI 10.17487/RFC3678, January 2004, <http://www.rfc-editor.org/info/rfc3678>.

[RFC4566] Handley, M., Jacobson, V., and C. Perkins, "SDP: Session Description Protocol", RFC 4566, DOI 10.17487/RFC4566, July 2006, <http://www.rfc-editor.org/info/rfc4566>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <http://www.rfc-editor.org/info/rfc4821>.

[RFC5082] Gill, V., Heasley, J., Meyer, D., Savola, P., Ed., and C. Pignataro, "The Generalized TTL Security Mechanism (GTSM)", RFC 5082, DOI 10.17487/RFC5082, October 2007, <http://www.rfc-editor.org/info/rfc5082>.

[RFC5097] Renker, G. and G. Fairhurst, "MIB for the UDP-Lite protocol", RFC 5097, DOI 10.17487/RFC5097, January 2008, <http://www.rfc-editor.org/info/rfc5097>.

[RFC5790] Liu, H., Cao, W., and H. Asaeda, "Lightweight Internet Group Management Protocol Version 3 (IGMPv3) and Multicast Listener Discovery Version 2 (MLDv2) Protocols", RFC 5790, DOI 10.17487/RFC5790, February 2010, <http://www.rfc-editor.org/info/rfc5790>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <http://www.rfc-editor.org/info/rfc6458>.

[RFC6633] Gont, F., "Deprecation of ICMP Source Quench Messages", RFC 6633, DOI 10.17487/RFC6633, May 2012, <http://www.rfc-editor.org/info/rfc6633>.

[RFC6936] Fairhurst, G. and M. Westerlund, "Applicability Statement for the Use of IPv6 UDP Datagrams with Zero Checksums", RFC 6936, DOI 10.17487/RFC6936, April 2013, <http://www.rfc-editor.org/info/rfc6936>.

[RFC7657] Black, D., Ed. and P. Jones, "Differentiated Services (Diffserv) and Real-Time Communication", RFC 7657, DOI 10.17487/RFC7657, November 2015, <http://www.rfc-editor.org/info/rfc7657>.

[STEVENS] "Stevens, W., Fenner, B., and A. Rudoff, "UNIX Network Programming, The sockets Networking API", Addison-Wesley.", 2004.

Appendix A. Revision Notes

Note to RFC-Editor: please remove this entire section prior to publication.

Individual draft -00:

- o This is the first version. Comments and corrections are welcome directly to the authors or via the IETF TAPS working group mailing list.

Individual draft -01:

- o Includes ability of a UDP receiver to disallow zero checksum datagrams.

- o Fixes to references and some connect on UDP usage.

Individual draft -02:

- o Fixes to address issues noted by WG.

- o Completed Multicast section to specify modern APIs.

- o Noted comments on API usage for UDP.

- o Feedback from various reviewers.

Individual draft -03:

- o Removes pass 2 and 3 of the TAPS analysis from this revision. These are expected to be incorporated into a combined draft of the TAPS WG.

o Fixed Typos.

Appendix B. Notes Based on Typical Usage

This appendix contains notes to assist in a later revision.

The de facto standard application programming interface (API) for TCP/IP applications is the "sockets" interface[POSIX]. Some platforms also offer applications the ability to directly assemble and transmit IP packets through "raw sockets" or similar facilities. This is a second, more cumbersome method of using UDP. The use of this API is discussed in the RFC series in [I-D.ietf-tsvwg-rfc5405bis].

The UDP sockets API differs from that for TCP in several key ways. Because application programmers are typically more familiar with the TCP sockets API, this section discusses these differences. [STEVENS] provides usage examples of the UDP sockets API.

This section provides notes on some topics relating to implemented UDP APIs.

A UDP application can use the recv() and send() POSIX functions as well as the recvfrom() and sendto() and recvmsg and sendmsg() functions.

SO_REUSEADDR specifies that the rules used in validating addresses supplied to bind() should allow reuse of local addresses.

SO_REUSEPORT specifies that the rules used in validating ports supplied to bind() should allow reuse of a local port

Accessing TTL From applications: If the IP_RECVTTL option is enabled on a SOCK_DGRAM socket, the recvmsg(2) call will return the IP TTL (time to live) field for a UDP datagram. The msg_control field in the msghdr structure points to a buffer that contains a cmsghdr structure followed by the TTL.

Appendix C. UDP Multicast

UDP and UDP-Lite Multicast may be considered in later versions of this document. This appendix contains notes to assist in this later revision.

A host must request the ability to broadcast before it can send/ receive ipv4 broadcast traffic. A host must become a member of a multicast group at the network layer before it can receive datagrams sent to the group.

C.1. Multicast Primitives

UDP and UDP-Lite support IPv4 broadcast and IPv4/IPv6 Multicast. Use of multicast requires additional functions at the transport API that must be called to coordinate operation of the IPv4 and IPv6 network layer protocols.

Guidance on the use of UDP and UDP-Lite for multicast services is provided in [I-D.ietf-tsvwg-rfc5405bis].

The following are defined:

JoinLocalGroup: 1 of [RFC3493] provides a function that allows joining of a local IPv4 multicast group.

IPV6_MULTICAST_IF: Section 5.2 of [RFC2553] states that this sets the interface to use for outgoing multicast packets.

IP_MULTICAST_TTL: This sets the hop limit to use for outgoing multicast packets. This is used to limit scope of multicast datagrams. When used for applications such as GTSM, this needs the UDP receiver API to pass the received value of this field to the application. (This is equivalent to IPV6_MULTICAST_HOPS for IPv6 multicast and TTL/IPV6_UNICAST_HOPS for unicast datagrams).

IPV6_MULTICAST_HOPS: Section 5.2 of [RFC2553] states that this sets the hop limit to use for outgoing multicast packets. When used for applications such as GTSM, this needs the UDP receiver API to pass the received value of this field to the application. (This is equivalent to IP_MULTICAST_TTL for IPv4 multicast and TTL/IPV6_UNICAST_HOPS for unicast datagrams).

IPV6_MULTICAST_LOOP: Section 5.2 of [RFC2553] states that this sets whether a copy of a datagram is looped back by the IP layer for local delivery when the datagram is sent to a group to which the sending host itself belongs).

IPV6_JOIN_GROUP: Section 5.2 of [RFC2553] provides a function that allows joining of an IPv6 multicast group.

SIOCGIPMSFILTER: Section 8.1 of [RFC3678] provides a function that allows reading the multicast source filters.

SIOCSIPMSFILTER: Section 8.1 of [RFC3678] provides a function that allows setting/modifying the multicast source filters.

IPV6_LEAVE_GROUP: Section 5.2 of [RFC2553] provides a function that allows leaving of a multicast group.

LeaveHostGroup: Section 7.1 of [RFC3493] provides a function that allows joining of an IPv4 multicast group.

LeaveLocalGroup: Section 7.1 of [RFC3493] provides a function that allows joining of a local IPv4 multicast group.

Section 4.1.1 of [RFC3678] updates the interface to add support for Multicast Source Filters (MSF) to IGMPv3 for Any Source Multicast (ASM):

This identifies three sets of API functionality:

1. IPv4 Basic (Delta-based) API. "Each function call specifies a single source address which should be added to or removed from the existing filter for a given multicast group address on which to listen."

2. IPv4 Advanced (Full-state) API. "This API allows an application to define a complete source-filter comprised of zero or more source addresses, and replace the previous filter with a new one."

3. Protocol-Independent Basic MSF (Delta-based) API

4. Protocol-Independent Advanced MSF (Full-state) API

It specifies the following primitives:

IP_ADD_MEMBERSHIP: This is used to join an ASM group.

IP_BLOCK_SOURCE: This is a MSF that can be used to block data from a given multicast source to a given group for ASM or SSM.

IP_UNBLOCK_SOURCE: This updates an MSF to undo a previous call to IP_UNBLOCK_SOURCE for ASM or SSM.

IP_DROP_MEMBERSHIP: This is used to leave an ASM or SSM group. (In SSM this drops all sources that have been joined for a particular group and interface. The operations are the same as if the socket had been closed.)

Section 4.1.2 of [RFC3678] updates the interface to add Multicast Source Filter (MSF) support for IGMPv3 with Any Source Multicast (ASM) using IPv4:

IP_ADD_SOURCE_MEMBERSHIP: This is used to join an SSM group.

IP_DROP_SOURCE_MEMBERSHIP: This is used to leave an SSM group.

Section 4.1.2 of [RFC3678] defines the Advanced (Full-state) API:

setipv4sourcefilter This is used to join an IPv4 multicast group, or to enable multicast from a specified source.

getipv4sourcefilter: This is used to leave an IPv4 multicast group, or to filter multicast from a specified source.

Section 5.1 of [RFC3678] specifies Protocol-Independent Multicast API functions:

MCAST_JOIN_GROUP This is used to join an ASM group.

MCAST_JOIN_SOURCE_GROUP This is used to join an SSM group.

MCAST_BLOCK_SOURCE: This is used to block a source in an ASM group.

MCAST_UNBLOCK_SOURCE: This removes a previous MSF set by MCAST_BLOCK_SOURCE:

MCAST_LEAVE_GROUP: This leaves a SSM group.

MCAST_LEAVE_GROUP: This leaves a ASM or SSM group.

Section 5.2 of [RFC3678] specifies the Protocol-Independent Advanced MSF (Full-state) API applicable for both IPv4 and IPv6 multicast:

setsourcefilter This is used to join an IPv4 or IPv6 multicast group, or to enable multicast from a specified source.

getsourcefilter: This is used to leave an IPv4 or IPv6 multicast group, or to filter multicast from a specified source.

Section 7.2 of [RFC5790] updates the interface to specify support for Lightweight IGMPv3 (LW_IGMPv3) and MLDv2.

According to the MSF API definition [RFC3678], "an LW-IGMPv3 host should implement either the IPv4 Basic MSF API or the Protocol-Independent Basic MSF API, and an LW-MLDv2 host should implement the Protocol- Independent Basic MSF API. Other APIs, IPv4 Advanced MSF API and Protocol-Independent Advanced MSF API, are optional to implement in an LW-IGMPv3/LW-MLDv2 host."

Authors’ Addresses

Godred Fairhurst
University of Aberdeen
School of Engineering
Fraser Noble Building
Fraser Noble Building Aberdeen AB24 3UE
UK

Email: gorry@erg.abdn.ac.uk

Tom Jones
University of Aberdeen
School of Engineering
Fraser Noble Building
Aberdeen AB24 3UE
UK

Email: tom@erg.abdn.ac.uk

TAPS S. Gjessing
Internet-Draft M. Welzl
Intended status: Informational University of Oslo
Expires: September 19, 2016 March 18, 2016

A Minimal Set of Transport Services for TAPS Systems
draft-gjessing-taps-minset-01

Abstract

This draft will eventually recommend a minimal set of IETF Transport Services offered by end systems supporting TAPS, and give guidance on choosing among the available mechanisms and protocols. It categorizes the set of transport services given in the TAPS document draft-ietf-taps-transports-usage-00, assuming that the eventual minimal set of transport services will be based on a similar form of categorization.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

An application has an intended usage and demands for transport services, and the task of any system that implements TAPS is to offer these services to its applications, i.e. the applications running on top of TAPS, without binding an application to a particular transport protocol.

The present draft is based on [TAPS1] and [TAPS2] and follows the same terminology (also listed below). The purpose of these two drafts is, according to the TAPS charter, to "Define a set of Transport Services, identifying the services provided by existing IETF protocols and congestion control mechanisms." This is item 1 in the list of working group tasks. Also according to the TAPS charter, the working group will then "Specify the subset of those Transport Services, as identified in item 1, that end systems supporting TAPS will provide, and give guidance on choosing among available mechanisms and protocols. Note that not all the capabilities of IETF Transport protocols need to be exposed as Transport Services." Hence it is necessary to minimize the number of services that are offered. We begin this by grouping the transport features.

Following [TAPS2], we divide the transport service features into two main groups as follows:

1. Connection related transport service features
   - Establishment
   - Availability

- Maintenance
- Termination

2. Data Transfer Related Transport Service Features
   - Sending Data
   - Receiving Data
   - Errors

Because QoS is out of scope of TAPS, this document assumes a "best effort" service model [RFC5290], [RFC7305]. Applications using a TAPS system can therefore not make any assumptions about e.g. the time it will take to send a message. There are however certain requirements that are strictly kept by transport protocols today, and these must also be kept by a TAPS system. Some of these requirements relate to features that we call "Functional".

Functional features provide functionality that cannot be used without the application knowing about them, or else they violate assumptions that might cause the application to break. For example, unordered message delivery is a functional feature: it cannot be used without the application knowing about it because the application’s assumption could be that messages arrive in-order, and in this case unordered delivery could cause the application to break. Change DSCP and data bundling (Nagle in TCP) are optimizing features: if a TAPS system autonomously decides to enable or disable them, an application will not break, but a TAPS system may be able to communicate more efficiently if the application is in control of this optimizing feature. Change DSCP and data bundling are examples of features that require application-specific knowledge (about delay/bandwidth requirements and the length of future data blocks that are to be transmitted, respectively). Some features, however, do not always require application-specific knowledge, and could therefore sometimes be used by a TAPS system without exposing them to the application. We call these features potentially automatable.

To summarize, features offered to applications are divided into two groups as follows:

- o Potentially automatable
  It may sometimes be possible to use this feature without support by the application.
- o Application-specific
  It is not possible to use this feature without support by the application.

The Application-specific features are further divided into two groups:

o Functional
   This feature is application-specific, and using it without explicitly involving the application could lead to incorrect operation.

o Optimizing
   This feature is application-specific, and can allow an application to improve its performance.

In the following, some features are additionally marked as DELETED. These features are IETF Transport protocol features that are not exposed to the TAPS user because they include functionality that is automatable. A few features are marked as "ADDED". These provide non-automatable functionality of DELETED features.

2. Terminology (as defined by draft-ietf-taps-transports-10)

The following terms are used throughout this document, and in subsequent documents produced by TAPS that describe the composition and decomposition of transport services.

Transport Service Feature: a specific end-to-end feature that the transport layer provides to an application. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire.

Transport Service Instance: an arrangement of transport protocols with a selected set of features and configuration parameters that implements a single transport service, e.g., a protocol stack (RTP over UDP).

Application: an entity that uses the transport layer for end-to-end delivery data across the network (this may also be an upper layer protocol or tunnel encapsulation).

3. The superset of transport service features

This section is based on the classification of the transport service features in pass 3 of [TAPS2]. As noted earlier, whether the usage of potentially automatable features can be automatized in a TAPS system depends on how much network-specific information an application wants to manipulate (e.g., to directly expose to its user). Therefore, in the following, "application-specific knowledge" refers to knowledge that only applications have, as opposed to all knowledge that applications may want to have.

3.1. CONNECTION Related Transport Service Features

ESTABLISHMENT:

o Connect
  Protocols: TCP, SCTP
  Functional because the notion of a connection is often reflected in applications as an expectation to be able to communicate after a "Connect" succeeded, with a communication sequence relating to this feature that is defined by the application protocol.
  ADDED.

o Specify IP Options
  Protocols: TCP
  Potentially automatable because IP Options relate to knowledge about the network, not the application.
  DELETED.

o Request multiple streams
  Protocols: SCTP
  Potentially automatable because using multi-streaming does not require application-specific knowledge.
  DELETED.

o Obtain multiple sockets
  Protocols: SCTP
  Potentially automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
  DELETED.

AVAILABILITY:

o Listen
  Protocols: All
  Functional because the notion of accepting connection requests is often reflected in application as an expectation to be able to communicate after a "Listen" succeeded, with a communication

sequence relating to this feature that is defined by the application protocol.
ADDED.

- Listen, 1 specified local interface
  Protocols: TCP, SCTP
  Potentially automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
  DELETED.

- Listen, N specified local interfaces
  Protocols: SCTP
  Potentially automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
  DELETED.

- Listen, all local interfaces (unspecified)
  Protocols: TCP, SCTP
  Potentially automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
  DELETED.

- Obtain requested number of streams
  Protocols: SCTP
  Potentially automatable because using multi-streaming does not require application-specific knowledge.

MAINTENANCE:

- Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP
  Functional because this is closely related to potentially assumed reliable data delivery.

o Control advertising timeout for aborting connection to remote endpoint
   Protocols: TCP
   Functional because this is closely related to potentially assumed reliable data delivery.

o Disable Nagle algorithm
   Protocols: TCP, SCTP
   Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.

o Request an immediate heartbeat, returning success/failure
   Protocols: SCTP
   Potentially automatable because this informs about network-specific knowledge.

o Set protocol parameters
   Protocols: SCTP
   SCTP parameters: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst
   Potentially automatable because these parameters relate to knowledge about the network, not the application.

o Notification of Excessive Retransmissions (early warning below abortion threshold)
   Protocols: TCP
   Optimizing because it is an early warning to the application, informing it of an impending functional event.

o Notification of ICMP error message arrival
   Protocols: TCP
   Optimizing because these messages can inform about success or failure of functional features (e.g., host unreachable relates to "Connect")

- o Status (query or notification)
  Protocols: SCTP
  SCTP parameters: association connection state; socket list; socket reachability states; current receiver window size; current congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; socket becoming active / inactive
  Potentially automatable because these parameters relate to knowledge about the network, not the application.

- o Set primary path
  Protocols: SCTP
  Potentially automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is potentially automatable.

- o Change DSCP
  Protocols: TCP
  Optimizing because choosing a suitable DSCP value requires application-specific knowledge.

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Functional because the notion of a connection is often reflected in applications as an expectation to have all outstanding data delivered and no longer be able to communicate after a "Close" succeeded, with a communication sequence relating to this feature that is defined by the application protocol.

- o Abort without delivering remaining data, causing an event informing the application on the other side
  Protocols: TCP, SCTP
  Functional because the notion of a connection is often reflected in applications as an expectation to potentially not have all outstanding data delivered and no longer be able to communicate

after an "Abort" succeeded, with a communication sequence relating to this feature that is defined by the application protocol.

- Timeout event when data could not be delivered for too long
  Protocols: TCP, SCTP
  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.

3.2. DATA Transfer Related Transport Service Features

3.2.1. Sending Data

- Reliably transfer data
  Protocols: TCP, SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.

- Notifying the receiver to promptly hand over data to application
  Protocols: TCP
  Optimizing because this is meant to control sleep times of the application's receiving process.

- Message identification
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.

- Choice of stream
  Protocols: SCTP
  Potentially automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is potentially automatable.

- Choice of path (destination address)
  Protocols: SCTP

Potentially automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is potentially automatable.

- Message lifetime
  Protocols: SCTP
  Optimizing because only applications know about the time criticality of their communication.

- Choice between unordered (potentially faster) or ordered delivery
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.

- Request not to bundle messages
  Protocols: SCTP
  Optimizing because this decision depends on knowledge about the size of future data blocks and the delay between them.

- Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP
  Functional because it allows application data with every message, for the sake of identification of data, which by itself is application-specific.

3.2.2. Receiving Data

- Receive data
  Protocols: TCP, SCTP
  Functional because a TAPS system must be able to send and receive data.

- Choice of stream to receive from
  Protocols: SCTP

Potentially automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is potentially automatable.

- Message identification
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.

- Information about partial message arrival
  Protocols: SCTP
  Functional because this is closely tied to properties of the data that an application sends or expects to receive.

3.2.3. Errors

- Notification of send failures
  Protocols: All
  Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
  ADDED.

- Notification of unsent messages
  Protocols: SCTP
  Automatable because the distinction between unsent and unacknowledged is network-specific.
  DELETED.

- Notification of unacknowledged messages
  Protocols: SCTP
  Automatable because the distinction between unsent and unacknowledged is network-specific.
  DELETED.

4. Conclusion

The eventual recommendations are:

- A TAPS system should exhibit all functional features that are offered by the transport protocols that it uses because these features could otherwise not be utilized by the TAPS system. It can still be possible to implement a TAPS system that does not offer all functional features, e.g. for the sake of uniform application operation across a broader set of protocols, but then the corresponding functionality of transport protocols is not exploited.
- A TAPS system should exhibit all application-specific optimizing features. If an application-specific optimizing feature is only available in a subset of the transport protocols used by the TAPS system, it should be acceptable for the TAPS system to ignore its usage when the transport protocol that is currently used does not provide it because of the performance-optimizing nature of the feature and the initially mentioned assumption of "best effort" operation.
- By hiding potentially automatable features from the application, a TAPS system can gain opportunities to automatize network-related functionality. This can facilitate using the TAPS system for the application programmer and it allows for optimizations that may not be possible for an application. For instance, a kernel-level TAPS system that hides SCTP multi-streaming from applications could theoretically map application-level connections from multiple applications onto the same SCTP association. Similarly, system-wide configurations regarding the usage of multiple interfaces could be exploited if the choice of the interface is not given to the application. However, if an application wants to directly expose such choices to its user, not offering this functionality can become a disadvantage of a TAPS system. This is a trade-off that must be considered in TAPS system design.

Given that the intention of TAPS is to break the design-time binding between applications and transport protocols, the decision on which features a TAPS system provides should also depend on the protocols that support them. Features that are provided by only one particular transport protocol have the potential to tie applications to that protocol. They should either not be offered, or replaced by fall-back functionality that allows for semantically correct operation (for example, ordered data delivery is correct but potentially slower for an application that requests unordered data delivery. "Potentially slower" is not a hindrance to correct operation within the "best effort" service model).

5. Acknowledgements

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

6. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

7. Security Considerations

Security will be considered in future versions of this document.

8. Informative References

[RFC5290] Floyd, S. and M. Allman, "Comments on the Usefulness of Simple Best-Effort Traffic", RFC 5290, DOI 10.17487/RFC5290, July 2008, <http://www.rfc-editor.org/info/rfc5290>.

[RFC7305] Lear, E., Ed., "Report from the IAB Workshop on Internet Technology Adoption and Transition (ITAT)", RFC 7305, DOI 10.17487/RFC7305, July 2014, <http://www.rfc-editor.org/info/rfc7305>.

[TAPS1] Fairhurst, G., Trammell, B., and M. Kuehlewind, "Services provided by IETF transport protocols and congestion control mechanisms", Internet-draft draft-ietf-taps-transports-10, March 2016.

[TAPS2] Welzl, M., Tuexen, M., and N. Khademi, "An Approach to Identify Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", Internet-draft draft-ietf-taps-transports-usage-00, June 2015.

Authors' Addresses

Stein Gjessing
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 44
Email: steing@ifi.uio.no

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

TAPS M. Welzl
Internet-Draft University of Oslo
Intended status: Informational M. Tuexen
Expires: July 11, 2016 Muenster Univ. of Appl. Sciences
N. Khademi
University of Oslo
January 8, 2016

On the Usage of Transport Service Features Provided by IETF Transport Protocols
draft-ietf-taps-transports-usage-00

Abstract

This document describes how transport protocols expose services to applications and how an application can configure and use the features of a transport service.

Status of this Memo

Copyright Notice

Table of Contents

1. Terminology

Transport Service Feature: a specific end-to-end feature that a transport service provides to its clients. Examples include confidentiality, reliable delivery, ordered delivery, message-versus-stream orientation, etc.

Transport Service: a set of transport service features, without an association to any given framing protocol, which provides a complete service to an application.

Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire.

Transport Protocol Component: an implementation of a transport service feature within a protocol.

Transport Service Instance: an arrangement of transport protocols with a selected set of features and configuration parameters that implements a single transport service, e.g., a protocol stack (RTP over UDP).

Application: an entity that uses the transport layer for end-to-end delivery of data across the network (this may also be an upper layer protocol or tunnel encapsulation).

Endpoint: an entity that communicates with one or more other endpoints using a transport protocol.

Connection: shared state of two or more endpoints that persists across messages that are transmitted between these endpoints.

Primitive: a function call that is used to locally communicate between an application and a transport endpoint and is related to one or more Transport Service Features.

Parameter: a value passed between an application and a transport protocol by a primitive.

Socket: the combination of a destination IP address and a destination port number.

2. Introduction

This document presents defined interactions between transport protocols and applications in the form of 'primitives' (function calls). Primitives can be invoked by an application or a transport protocol; the latter type is called an "event". The list of transport service features and primitives in this document is strictly based on the parts of protocol specifications that relate to what the protocol provides to an application using it and how the application interacts with it. It does not cover parts of a protocol that are explicitly stated as optional to implement.

The document presents a three-pass process to arrive at a list of transport service features. In the first pass, the relevant RFC text is discussed per protocol. In the second pass, this discussion is used to derive a list of primitives that are uniformly categorized across protocols. Here, an attempt is made to present or -- where text describing primitives does not yet exist -- construct primitives in a slightly generalized form to highlight similarities. This is, for example, achieved by renaming primitives of protocols or by avoiding a strict 1:1-mapping between the primitives in the protocol specification and primitives in the list. Finally, the third pass presents transport service features based on pass 2, identifying which protocols implement them.

In the list resulting from the second pass, some transport service features are missing because they are implicit in some protocols, and they only become explicit when we consider the superset of all features offered by all protocols. For example, TCP’s reliability includes integrity via a checksum, but we have to include a protocol like UDP-Lite as specified in [RFC3828] (which has a configurable checksum) in the list before we can consider an always-on checksum as a transport service feature. Similar arguments apply to other protocol functions (e.g. congestion control). The complete list of features across all protocols is therefore only available after pass 3.

This document discusses unicast transport protocols. [AUTHOR’S NOTE: we skip "congestion control mechanisms" for now. This simplifies the discussion; the congestion control mechanisms part is about LEDBAT, which should be easy to add later.] Transport protocols provide communication between processes that operate on network endpoints, which means that they allow for multiplexing of communication between the same IP addresses, and normally this multiplexing is achieved using port numbers. Port multiplexing is therefore assumed to be always provided and not discussed in this document.

Some protocols are connection-oriented. Connection-oriented protocols often use an initial call to a specific transport primitive to open a connection before communication can progress, and require communication to be explicitly terminated by issuing another call to a transport primitive (usually called "close"). A "connection" is the common state that some transport primitives refer to, e.g., to adjust general configuration settings. Connection establishment, maintenance and termination are therefore used to categorize transport primitives of connection-oriented transport protocols in pass 2 and pass 3.

3. Pass 1

This first iteration summarizes the relevant text parts of the RFCs describing the protocols, focusing on what each transport protocol provides to the application and how it is used (abstract API descriptions, where they are available).

3.1. Primitives Provided by TCP

[RFC0793] states: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks". Section 3.8 in [RFC0793] further specifies the interaction with the application by listing several transport primitives. It is also assumed that an Operating System provides a means for TCP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. This section describes the relevant primitives.

open: this is either active or passive, to initiate a connection or listen for incoming connections. All other primitives are associated with a specific connection, which is assumed to first have been opened. An active open call contains a socket. A passive open call with a socket waits for a particular connection; alternatively, a passive open call can leave the socket unspecified to accept any incoming connection. A fully specified passive call can later be made active by calling 'send'. Optionally, a timeout can be specified, after which TCP will abort the connection if data has not been successfully delivered to the destination (else a default timeout value is used). [RFC1122] describes a procedure for aborting the connection that must be used to avoid excessive retransmissions, and states that an application must be able to control the threshold used to determine the condition for aborting -- and that this threshold may be measured in time units or as a count of retransmission. This indicates that the timeout could also be specified as a count of retransmission.

Also optional, for multihomed hosts, the local IP address can be provided [RFC1122]. If it is not provided, a default choice will be made in case of active open calls. A passive open call will await incoming connection requests to all local addresses and then maintain usage of the local IP address where the incoming connection request has arrived. Finally, the 'options' parameter is explained in [RFC1122] to allow the application to specify IP options such as source route, record route, or timestamp. It is not stated on which segments of a connection these options should be applied, but probably all segments, as this is also stated in a specification given for the usage of source route (section 4.2.3.8 of [RFC1122]). Source route is the only non-optional IP option in this parameter, allowing an application to specify a source route when it actively opens a TCP connection.

send: this is the primitive that an application uses to give the local TCP transport endpoint a number of bytes that TCP should reliably send to the other side of the connection. The URGENT flag, if set, states that the data handed over by this send call is urgent and this urgency should be indicated to the receiving process in case the receiving application has not yet consumed all non-urgent data preceding it. An optional timeout parameter can be provided that updates the connection's timeout (see 'open').

receive: This primitive allocates a receiving buffer for a provided number of bytes. It returns the number of received bytes provided in the buffer when these bytes have been received and written into the buffer by TCP. The application is informed of urgent data via an URGENT flag: if it is on, there is urgent data. If it is off, there is no urgent data or this call to 'receive' has returned all the urgent data.

close: This primitive closes one side of a connection. It is semantically equivalent to "I have no more data to send" but does not mean "I will not receive any more", as the other side may still have data to send. This call reliably delivers any data that has already been given to TCP (and if that fails, 'close' becomes 'abort').

abort: This primitive causes all pending 'send' and 'receive' calls to be aborted. A TCP RESET message is sent to the TCP endpoint on the other side of the connection [RFC0793].

close event: TCP uses this primitive to inform an application that the application on the other side has called the 'close' primitive, so the local application can also issue a 'close' and terminate the connection gracefully. See [RFC0793], Section 3.5.

abort event: When TCP aborts a connection upon receiving a "Reset" from the peer, it "advises the user and goes to the CLOSED state." See [RFC0793], Section 3.4.

USER TIMEOUT event: This event, described in Section 3.9 of [RFC0793], is executed when the user timeout expires (see 'open'). All queues are flushed and the application is informed that the connection had to be aborted due to user timeout.

ERROR_REPORT event: This event, described in Section 4.2.4.1 of [RFC1122], informs the application of "soft errors" that can be safely ignored [RFC5461], including the arrival of an ICMP error message or excessive retransmissions (reaching a threshold below the threshold where the connection is aborted).

Type-of-Service: Section 4.2.4.2 of [RFC1122] states that the application layer MUST be able to specify the Type-of-Service (TOS) for segments that are sent on a connection. The application should be able to change the TOS during the connection lifetime, and the TOS value should be passed to the IP layer unchanged. Since then the TOS field has been redefined. A part of the field has been assigned to ECN [RFC3168] and the six most significant bits have been assigned to carry the DiffServ CodePoint, DSField [RFC3260]. Staying with the intention behind the application's ability to specify the "Type of Service", this should probably be interpreted to mean the value in the DSField, which is the Differentiated Services Codepoint (DSCP).

Nagle: The Nagle algorithm, described in Section 4.2.3.4 of [RFC1122], delays sending data for some time to increase the likelihood of sending a full-sized segment. An application can disable the Nagle algorithm for an individual connection.

User Timeout Option: The User Timeout Option (UTO) [RFC5482] allows one end of a TCP connection to advertise its current user timeout value so that the other end of the TCP connection can adapt its own user timeout accordingly. In addition to the configurable value of the User Timeout (see 'send'), [RFC5482] introduces three per-connection state variables that an application can adjust to control the operation of the User Timeout Option (UTO): ADV_UTO is the value of the UTO advertised to the remote TCP peer (default: system-wide default user timeout); ENABLED (default false) is a boolean-type flag that controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. CHANGEABLE is a boolean-type flag (default true) that controls whether the user timeout may be changed based on a UTO option received from the other end of the connection. CHANGEABLE becomes false when an application explicitly sets the user timeout (see 'send').

3.1.1. Excluded Primitives

The 'open' primitive specified in [RFC0793] can be handed optional Precedence or security/compartment information according to [RFC0793], but this was not included here because it is mostly irrelevant today, as explained in [RFC7414].

The 'status' primitive was not included because [RFC0793] describes this primitive as "implementation dependent" and states that it "could be excluded without adverse effect". Moreover, while a data block containing specific information is described, it is also stated that not all of this information may always be available. The 'send' primitive described in [RFC0793] includes an optional PUSH flag which, if set, requires data to be promptly transmitted to the receiver without delay; the 'receive' primitive described in [RFC0793] can (under some conditions) yield the status of the PUSH flag. Because PUSH functionality is made optional to implement for both the 'send' and 'receive' primitives in [RFC1122], this functionality is not included here. [RFC1122] also introduces keep-alives to TCP, but these are optional to implement and hence not considered here. [RFC1122] describes that "some TCP implementations have included a FLUSH call", indicating that this call is also optional to implement. It is therefore not considered here.

3.2. Primitives Provided by SCTP

Section 1.1 of [RFC4960] lists limitations of TCP that SCTP removes. Three of the four mentioned limitations directly translate into a transport service features that are visible to an application using SCTP: 1) it allows for preservation of message delineations; 2) these messages, while reliably transferred, do not require to be in order unless the application wants it; 3) multi-homing is supported. In SCTP, connections are called "association" and they can be between not only two (as in TCP) but multiple addresses at each endpoint.

Section 10 of [RFC4960] further specifies the interaction with the application (which RFC [RFC4960] calls the "Upper Layer Protocol" (ULP)). It is assumed that the Operating System provides a means for SCTP to asynchronously signal the application; the primitives representing such signals are called 'events' in this section. Here, we describe the relevant primitives.

Initialize: Initialize creates a local SCTP instance that it binds to a set of local addresses (and, if provided, port number). Initialize needs to be called only once per set of local addresses.

Associate: This creates an association (the SCTP equivalent of a connection) between the local SCTP instance and a remote SCTP instance. Most primitives are associated with a specific association, which is assumed to first have been created. Associate can return a list of destination transport addresses so that multiple paths can later be used. One of the returned sockets will be selected by the local endpoint as default primary path for sending SCTP packets to this peer, but this choice can be changed by the application using the list of destination addresses. Associate is also given the number of outgoing streams to request and optionally returns the number of outgoing streams negotiated.

Send: This sends a message of a certain length in bytes over an association. A number can be provided to later refer to the correct message when reporting an error, and a stream id is provided to specify the stream to be used inside an association (we consider this as a mandatory parameter here for simplicity: if not provided, the stream id defaults to 0). An optional maximum life time can specify the time after which the message should be discarded rather than sent. A choice (advisory, i.e. not guaranteed) of the preferred path can be made by providing a socket, and the message can be delivered out-of-order if the unordered flag is set. Another advisory flag indicates whether the application prefers to avoid bundling user data with other outbound DATA chunks (i.e., in the same packet). A payload protocol-id can be provided to pass a value that indicates the type of payload protocol data to the peer.

Receive: Messages are received from an association, and optionally a stream within the association, with their size returned. The application is notified of the availability of data via a DATA ARRIVE notification. If the sender has included a payload protocol-id, this value is also returned. If the received message is only a partial delivery of a whole message, a partial flag will indicate so, in which case the stream id and a stream sequence number are provided to the application.

Shutdown: This primitive gracefully closes an association, reliably delivering any data that has already been handed over to SCTP. A return code informs about success or failure of this procedure.

Abort: This ungracefully closes an association, by discarding any locally queued data and informing the peer that the association was aborted. Optionally, an abort reason to be passed to the peer may be provided by the application. A return code informs about success or failure of this procedure.

Change Heartbeat / Request Heartbeat: This allows the application to enable/disable heartbeats and optionally specify a heartbeat frequency as well as requesting a single heartbeat to be carried out upon a function call, with a notification about success or failure of transmitting the HEARTBEAT chunk to the destination.

Set Protocol Parameters: This allows to set values for protocol parameters per association; for some parameters, a setting can be made per socket. The set listed in [RFC4960] is: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst.

Set Primary: This allows to set a new primary default path for an association by providing a socket. Optionally, a default source address to be used in IP datagrams can be provided.

Status: The 'Status' primitive returns a data block with information about a specified association, containing: association connection state; socket list; destination transport address reachability states; current receiver window size; current congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses.

COMMUNICATION UP notification: When a lost communication to an endpoint is restored or when SCTP becomes ready to send or receive user messages, this notification informs the application process about the affected association, the type of event that has occurred, the complete set of sockets of the peer, the maximum number of allowed streams and the inbound stream count (the number of streams the peer endpoint has requested).

DATA ARRIVE notification: When a message is ready to be retrieved via the Receive primitive, the application is informed by this notification.

SEND FAILURE notification / Receive Unsent Message / Receive Unacknowledged Message: When a message cannot be delivered via an association, the sender can be informed about it and learn whether the message has just not been acknowledged or (e.g. in case of lifetime expiry) if it has not even been sent.

NETWORK STATUS CHANGE notification: The NETWORK STATUS CHANGE notification informs the application about a socket becoming active/inactive.

COMMUNICATION LOST notification: When SCTP loses communication to an endpoint (e.g. via Heartbeats or excessive retransmission) or detects an abort, this notification informs the application process of the affected association and the type of event (failure OR termination in response to a shutdown or abort request).

SHUTDOWN COMPLETE notification: When SCTP completes the shutdown procedures, this notification is passed to the upper layer, informing it about the affected assocation.

3.2.1. Excluded Primitives

The 'Receive' primitive can return certain additional information, but this is optional to implement and therefore not considered. With a COMMUNICATION LOST notification, some more information may optionally be passed to the application (e.g., identification to retrieve unsent and unacknowledged data). SCTP "can invoke" a COMMUNICATION ERROR notification and "may send" a RESTART notification, making these two notifications optional to implement. The list provided under 'Status' includes "etc", indicating that more information could be provided. The primitive 'Get SRTT Report' returns information that is included in the information that 'Status' provides and is therefore not discussed. Similarly, 'Set Failure Threshold' sets only one out of various possible parameters included in 'Set Protocol Parameters'. The 'Destroy SCTP Instance' API function was excluded: it erases the SCTP instance that was created by 'Initialize', but is not a Primitive as defined in this document because it does not relate to a Transport Service Feature.

4. Pass 2

This pass categorizes the primitives from pass 1 based on whether they relate to a connection or to data transmission. Primitives are presented following the nomenclature: "CATEGORY.[SUBCATEGORY].PRIMITIVENAME.PROTOCOL". A connection is a general protocol-independent concept and refers to, e.g., TCP connections (identifiable by a unique pair of IP addresses and TCP port numbers) as well as SCTP associations (identifiable by multiple IP address and port number pairs).

Some minor details are omitted for the sake of generalization -- e.g., SCTP's 'close' [RFC4960] returns success or failure, whereas this is not described in the same way for TCP in [RFC0793], but this detail plays no significant role for the primitives provided by either TCP or SCTP.

The TCP 'send' and 'receive' primitives include usage of an "URGENT" mechanism. This mechanism is required to implement the "synch signal" used by telnet [RFC0854], but SHOULD NOT be used by new applications [RFC6093]. Because pass 2 is meant as a basis for the creation of TAPS systems, the "URGENT" mechanism is excluded. This also concerns the notification "Urgent pointer advance" in the

ERROR_REPORT described in Section 4.2.4.1 of [RFC1122].

4.1. CONNECTION Related Primitives

ESTABLISHMENT:
Active creation of a connection from one transport endpoint to one or more transport endpoints.

- CONNECT.TCP:
  Pass 1 primitive / event: 'open' (active) or 'open' (passive) with socket, followed by 'send'
  Parameters: 1 local IP address (optional); 1 destination transport address (for active open; else the socket and the local IP address of the succeeding incoming connection request will be maintained); timeout (optional); options (optional)
  Comments: If the local IP address is not provided, a default choice will automatically be made. The timeout can also be a retransmission count. The options are IP options to be used on all segments of the connection. At least the Source Route option is mandatory for TCP to provide.

- CONNECT.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'associate'
  Parameters: list of local SCTP port number / IP address pairs (initialize); 1 socket; outbound stream count
  Returns: socket list
  Comments: 'initialize' needs to be called only once per list of local SCTP port number / IP address pairs. One socket will automatically be chosen; it can later be changed in MAINTENANCE.

AVAILABILITY:
Preparing to receive incoming connection requests.

- LISTEN.TCP:
  Pass 1 primitive / event: 'open' (passive)
  Parameters: 1 local IP address (optional); 1 socket (optional); timeout (optional)
  Comments: if the socket and/or local IP address is provided, this waits for incoming connections from only and/or to only the provided address. Else this waits for incoming connections without this / these constraint(s). ESTABLISHMENT can later be performed with 'send'.

- LISTEN.SCTP:
  Pass 1 primitive / event: 'initialize', followed by 'COMMUNICATION UP' notification
  Parameters: list of local SCTP port number / IP address pairs

(initialize)
Returns: socket list; outbound stream count; inbound stream count
Comments: initialize needs to be called only once per list of local SCTP port number / IP address pairs. COMMUNICATION UP can also follow a COMMUNICATION LOST notification, indicating that the lost communication is restored.

MAINTENANCE:
Adjustments made to an open connection, or notifications about it. These are out-of-band messages to the protocol that can be issued at any time, at least after a connection has been established and before it has been terminated (with one exception: CHANGE-TIMEOUT.TCP can only be issued when DATA.SEND.TCP is called).

o CHANGE-TIMEOUT.TCP:
Pass 1 primitive / event: 'send' combined with unspecified control of per-connection state variables
Parameters: timeout value (optional); ADV_UTO (optional); boolean UTO_ENABLED (optional, default false); boolean CHANGEABLE (optional, default true)
Comments: when sending data, an application can adjust the connection's timeout value (time after which the connection will be aborted if data could not be delivered). If UTO_ENABLED is true, the user timeout value (or, if provided, the value ADV_UTO) will be advertised for the TCP on the other side of the connection to adapt its own user timeout accordingly. UTO_ENABLED controls whether the UTO option is enabled for a connection. This applies to both sending and receiving. CHANGEABLE controls whether the user timeout may be changed based on a UTO option received from the other end of the connection; it becomes false when 'timeout value' is used.

o CHANGE-TIMEOUT.SCTP:
Pass 1 primitive / event: 'Change HeartBeat' combined with 'Set Protocol Parameters'
Parameters: 'Change HeartBeat': heartbeat frequency; 'Set Protocol Parameters': Association.Max.Retrans (whole association) or Path.Max.Retrans (per socket)
Comments: Change Heartbeat can enable / disable heartbeats in SCTP as well as change their frequency. The parameter Association.Max.Retrans defines after how many unsuccessful heartbeats the connection will be terminated; thus these two primitives / parameters together can yield a similar behavior to CHANGE-TIMEOUT.TCP.

- DISABLE-NAGLE.TCP:
  Pass 1 primitive / event: not specified
  Parameters: one boolean value
  Comments: the Nagle algorithm delays data transmission to increase the chance to send a full-sized segment. An application must be able to disable this algorithm for a connection. This is related to the no-bundle flag in DATA.SEND.SCTP.

- REQUESTHEARTBEAT.SCTP:
  Pass 1 primitive / event: 'Request HeartBeat'
  Parameters: socket
  Returns: success or failure
  Comments: requests an immediate heartbeat on a path, returning success or failure.

- SETPROTOCOLPARAMETERS.SCTP:
  Pass 1 primitive / event: 'Set Protocol Parameters'
  Parameters: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst

- SETPRIMARY.SCTP:
  Pass 1 primitive / event: 'Set Primary'
  Parameters: socket
  Returns: result of attempting this operation
  Comments: update the current primary address to be used, based on the set of available sockets of the association.

- ERROR.TCP:
  Pass 1 primitive / event: 'ERROR_REPORT'
  Returns: reason (encoding not specified); subreason (encoding not specified)
  Comments: soft errors that can be ignored without harm by many applications; an application should be able to disable these notifications. The reported conditions include at least: ICMP error message arrived; Excessive Retransmissions.

- STATUS.SCTP:
  Pass 1 primitive / event: 'Status' and 'NETWORK STATUS CHANGE' notification
  Returns: data block with information about a specified association, containing: association connection state; socket list; destination transport address reachability states; current receiver window size; current congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses. The NETWORK STATUS CHANGE notification informs the application about a socket

becoming active/inactive.

o CHANGE-DSCP.TCP:
  Pass 1 primitive / event: not specified
  Parameters: DSCP value
  Comments: This allows an application to change the DSCP value. For TCP this was originally specified for the TOS field [RFC1122], which is here interpreted to refer to the DSField [RFC3260].

TERMINATION:
Gracefully or forcefully closing a connection, or being informed about this event happening.

o CLOSE.TCP:
  Pass 1 primitive / event: 'close'
  Comments: this terminates the sending side of a connection after reliably delivering all remaining data.

o CLOSE.SCTP:
  Pass 1 primitive / event: 'Shutdown'
  Comments: this terminates a connection after reliably delivering all remaining data.

o ABORT.TCP:
  Pass 1 primitive / event: 'abort'
  Comments: this terminates a connection without delivering remaining data and sends an error message to the other side.

o ABORT.SCTP:
  Pass 1 primitive / event: 'abort'
  Parameters: abort reason to be given to the peer (optional)
  Comments: this terminates a connection without delivering remaining data and sends an error message to the other side.

o TIMEOUT.TCP:
  Pass 1 primitive / event: 'USER TIMEOUT' event
  Comments: the application is informed that the connection is aborted. This event is executed on expiration of the timeout set in CONNECTION.ESTABLISHMENT.CONNECT.TCP (possibly adjusted in CONNECTION.MAINTENANCE.CHANGE-TIMEOUT.TCP).

o TIMEOUT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Comments: the application is informed that the connection is aborted. this event is executed on expiration of the timeout that should be enabled by default (see beginning of section 8.3 in [RFC4960]) and was possibly adjusted in

CONNECTION.MAINTENANCE.CHANGE-TIMEOOUT.SCTP.

- ABORT-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- ABORT-EVENT.SCTP:
  Pass 1 primitive / event: 'COMMUNICATION LOST' event
  Returns: abort reason from the peer (if available)
  Comments: the application is informed that the other side has aborted the connection using CONNECTION.TERMINATION.ABORT.SCTP.

- CLOSE-EVENT.TCP:
  Pass 1 primitive / event: not specified.

- CLOSE-EVENT.SCTP:
  Pass 1 primitive / event: 'SHUTDOWN COMPLETE' event
  Comments: the application is informed that CONNECTION.TERMINATION.CLOSE.SCTP was successfully completed.

4.2. DATA Transfer Related Primitives

All primitives in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data. In addition to the listed parameters, all sending primitives contain a reference to a data block and all receiving primitives contain a reference to available buffer space for the data.

- SEND.TCP:
  Pass 1 primitive / event: 'send'
  Parameters: timeout (optional)
  Comments: this gives TCP a data block for reliable transmission to the TCP on the other side of the connection. The timeout can be configured with this call whenever data are sent (see also CONNECTION.MAINTENANCE.CHANGE-TIMEOUT.TCP).

- SEND.SCTP:
  Pass 1 primitive / event: 'Send'
  Parameters: stream number; context (optional); life time (optional); socket (optional); unordered flag (optional); no-bundle flag (optional); payload protocol-id (optional)
  Comments: this gives SCTP a data block for reliable transmission to the SCTP on the other side of the connection (SCTP association). The 'stream number' denotes the stream to be used. The 'context' number can later be used to refer to the correct message when an error is reported. The 'life time' specifies a time after which this data block will not be sent. The 'socket'

can be used to state which path should be preferred, if there are multiple paths available (see also CONNECTION.MAINTENANCE.SETPRIMARY.SCTP). The data block can be delivered out-of-order if the 'unordered flag' is set. The 'no-bundle flag' can be set to indicate a preference to avoid bundling. The 'payload protocol-id' is a number that will, if provided, be handed over to the receiving application.

o RECEIVE.TCP:
Pass 1 primitive / event: 'receive'.

o RECEIVE.SCTP:
Pass 1 primitive / event: 'DATA ARRIVE' notification, followed by 'Receive'
Parameters: stream number (optional)
Returns: stream sequence number (optional), partial flag (optional)
Comments: if the 'stream number' is provided, the call to receive only receives data on one particular stream. If a partial message arrives, this is indicated by the 'partial flag', and then the 'stream sequence number' must be provided such that an application can restore the correct order of data blocks that comprise an entire message.

o SENDFAILURE-EVENT.SCTP:
Pass 1 primitive / event: 'SEND FAILURE' notification, optionally followed by 'Receive Unsent Message' or 'Receive Unacknowledged Message'
Returns: cause code; context; unsent or unacknowledged message (optional)
Comments: 'cause code' indicates the reason of the failure, and 'context' is the context number if such a number has been provided in DATA.SEND.SCTP, for later use with 'Receive Unsent Message' or 'Receive Unacknowledged Message', respectively. These primitives can be used to retrieve the complete unsent or unacknowledged message if desired.

5. Pass 3

This section presents the superset of all transport service features in all protocols that were discussed in the preceding sections, based on the list of primitives in pass 2 but also on text in pass 1 to include features that can be configured in one protocol and are static properties in another. Again, some minor details are omitted for the sake of generalization -- e.g., TCP may provide various different IP options, but only source route is mandatory to implement, and this detail is not visible in the Pass 3 feature "Specify IP Options".

[AUTHOR’S NOTE: the list here looks pretty similar to the list in pass 2 for now. This will change as more protocols are added. For example, when we add UDP, we will find that UDP does not do congestion control, which is relevant to the application using it. This will have to be reflected in pass 1 and pass 2, only for UDP. In pass 3, we can then derive "no congestion control" as a transport service feature of UDP; however, since it would be strange to call the lack of congestion control a feature, the natural outcome is then to list "congestion control" as a feature of TCP and SCTP.]

5.1. CONNECTION Related Transport Service Features

ESTABLISHMENT:
Active creation of a connection from one transport endpoint to one or more transport endpoints.

- o Specify IP Options
  Protocols: TCP

- o Request multiple streams
  Protocols: SCTP

- o Obtain multiple sockets
  Protocols: SCTP

AVAILABILITY:
Preparing to receive incoming connection requests.

- o Listen, 1 specified local interface
  Protocols: TCP, SCTP

- o Listen, N specified local interfaces
  Protocols: SCTP

- o Listen, all local interfaces (unspecified)
  Protocols: TCP, SCTP

- o Obtain requested number of streams
  Protocols: SCTP

MAINTENANCE:
Adjustments made to an open connection, or notifications about it.
NOTE: all features except "set primary path" in this category apply

to one out of multiple possible paths (identified via sockets) in SCTP, whereas TCP uses only one path (one socket).

o Change timeout for aborting connection (using retransmit limit or time value)
  Protocols: TCP, SCTP

o Control advertising timeout for aborting connection to remote endpoint
  Protocols: TCP

o Disable Nagle algorithm
  Protocols: TCP, SCTP
  Comments: This is not specified in [RFC4960] but in [RFC6458].

o Request an immediate heartbeat, returning success/failure
  Protocols: SCTP

o Set protocol parameters
  Protocols: SCTP
  SCTP parameters: RTO.Initial; RTO.Min; RTO.Max; Max.Burst; RTO.Alpha; RTO.Beta; Valid.Cookie.Life; Association.Max.Retrans; Path.Max.Retrans; Max.Init.Retransmits; HB.interval; HB.Max.Burst
  Comments: in future versions of this document, it might make sense to split out some of these parameters -- e.g., if a different protocol provides means to adjust the RTO calculation there could be a common feature for them called "adjust RTO calculation".

o Notification of Excessive Retransmissions (early warning below abortion threshold)
  Protocols: TCP

o Notification of ICMP error message arrival
  Protocols: TCP

o Status (query or notification)
  Protocols: SCTP
  SCTP parameters: association connection state; socket list; socket reachability states; current receiver window size; current congestion window sizes; number of unacknowledged DATA chunks; number of DATA chunks pending receipt; primary path; most recent SRTT on primary path; RTO on primary path; SRTT and RTO on other destination addresses; socket becoming active / inactive

o Set primary path
  Protocols: SCTP

o Change DSCP
Protocols: TCP
Comments: This is described to be changeable for SCTP too in [RFC6458].

TERMINATION:
Gracefully or forcefully closing a connection, or being informed about this event happening.

o Close after reliably delivering all remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Comments: A TCP endpoint locally only closes the connection for sending; it may still receive data afterwards.

o Abort without delivering remaining data, causing an event informing the application on the other side
Protocols: TCP, SCTP
Comments: In SCTP a reason can optionally be given by the application on the aborting side, which can then be received by the application on the other side.

o Timeout event when data could not be delivered for too long
Protocols: TCP, SCTP
Comments: the timeout is configured with CONNECTION.MAINTENANCE "Change timeout for aborting connection (using retransmit limit or time value)".

5.2. DATA Transfer Related Transport Service Features

All features in this section refer to an existing connection, i.e. a connection that was either established or made available for receiving data. Reliable data transfer entails delay -- e.g. for the sender to wait until it can transmit data, or due to retransmission in case of packet loss.

5.2.1. Sending Data

All features in this section are provided by DATA.SEND from pass 2. DATA.SEND is given a data block from the application, which we here call a "message".

o Reliably transfer data
Protocols: TCP, SCTP

- o Notifying the receiver to promptly hand over data to application
  Protocols: TCP
  Comments: This seems unnecessary in SCTP, where data arrival causes an event for the application.

- o Message identification
  Protocols: SCTP

- o Choice of stream
  Protocols: SCTP

- o Choice of path (destination address)
  Protocols: SCTP

- o Message lifetime
  Protocols: SCTP

- o Choice between unordered (potentially faster) or ordered delivery
  Protocols: SCTP

- o Request not to bundle messages
  Protocols: SCTP

- o Specifying a "payload protocol-id" (handed over as such by the receiver)
  Protocols: SCTP

5.2.2. Receiving Data

All features in this section are provided by DATA.RECEIVE from pass 2. DATA.RECEIVE fills a buffer provided to the application, with what we here call a "message".

- o Receive data
  Protocols: TCP, SCTP

- o Choice of stream to receive from
  Protocols: SCTP

- o Message identification
  Protocols: SCTP
  Comments: In SCTP, this is optionally achieved with a "stream sequence number". The stream sequence number is always provided in case of partial message arrival.

- Information about partial message arrival
  Protocols: SCTP
  Comments: In SCTP, partial messages are combined with a stream sequence number so that the application can restore the correct order of data blocks an entire message consists of.

5.2.3. Errors

This section describes sending failures that are associated with a specific call to DATA.SEND from pass 2.

- Notification of unsent messages
  Protocols: SCTP

- Notification of unacknowledged messages
  Protocols: SCTP

6. Acknowledgements

The authors would like to thank (in alphabetical order) Bob Briscoe, David Hayes, Gorry Fairhurst, Karen Nielsen and Joe Touch for providing valuable feedback on this document. This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. 644334 (NEAT). The views expressed are solely those of the author(s).

7. IANA Considerations

XX RFC ED - PLEASE REMOVE THIS SECTION XXX

This memo includes no request to IANA.

8. Security Considerations

Security will be considered in future versions of this document.

9. References

9.1. Normative References

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981,

<http://www.rfc-editor.org/info/rfc793>.

[RFC1122] Braden, R., Ed., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, DOI 10.17487/RFC1122, October 1989, <http://www.rfc-editor.org/info/rfc1122>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <http://www.rfc-editor.org/info/rfc4960>.

[RFC5482] Eggert, L. and F. Gont, "TCP User Timeout Option", RFC 5482, DOI 10.17487/RFC5482, March 2009, <http://www.rfc-editor.org/info/rfc5482>.

9.2. Informative References

[FA15] Fairhurst, Ed., G., Trammell, Ed., B., and M. Kuehlewind, Ed., "Services provided by IETF transport protocols and congestion control mechanisms", draft-fairhurst-taps-transports-08.txt (work in progress), December 2015.

[RFC0854] Postel, J. and J. Reynolds, "Telnet Protocol Specification", STD 8, RFC 854, DOI 10.17487/RFC0854, May 1983, <http://www.rfc-editor.org/info/rfc854>.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

[RFC3168] Ramakrishnan, K., Floyd, S., and D. Black, "The Addition of Explicit Congestion Notification (ECN) to IP", RFC 3168, DOI 10.17487/RFC3168, September 2001, <http://www.rfc-editor.org/info/rfc3168>.

[RFC3260] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, DOI 10.17487/RFC3260, April 2002, <http://www.rfc-editor.org/info/rfc3260>.

[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, <http://www.rfc-editor.org/info/rfc3828>.

[RFC5461] Gont, F., "TCP's Reaction to Soft Errors", RFC 5461, DOI 10.17487/RFC5461, February 2009,

<http://www.rfc-editor.org/info/rfc5461>.

[RFC6093] Gont, F. and A. Yourtchenko, "On the Implementation of the TCP Urgent Mechanism", RFC 6093, DOI 10.17487/RFC6093, January 2011, <http://www.rfc-editor.org/info/rfc6093>.

[RFC6458] Stewart, R., Tuexen, M., Poon, K., Lei, P., and V. Yasevich, "Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)", RFC 6458, DOI 10.17487/RFC6458, December 2011, <http://www.rfc-editor.org/info/rfc6458>.

[RFC7414] Duke, M., Braden, R., Eddy, W., Blanton, E., and A. Zimmermann, "A Roadmap for Transmission Control Protocol (TCP) Specification Documents", RFC 7414, DOI 10.17487/RFC7414, February 2015, <http://www.rfc-editor.org/info/rfc7414>.

Appendix A. Overview of RFCs used as input for pass 1

TCP: [RFC0793], [RFC1122], [RFC5482]
SCTP: [RFC4960], planned: [RFC6458]

Appendix B. How to contribute

This document is only concerned with transport service features that are explicitly exposed to applications via primitives. It also strictly follows RFC text: if a feature is truly relevant for an application, the RFCs better say so and in some way describe how to use and configure it. Thus, the approach to follow for contributing to this document is to identify the right RFCs, then analyze and process their text.

Experimental RFCs are excluded, and so are primitives that MAY be implemented (by the transport protocol). To be included, the minimum requirement level for a primitive to be implemented by a protocol is SHOULD. If [RFC2119]-style requirements levels are not used, primitives should be excluded when they are described in conjunction with statements like, e.g.: "some implementations also provide" or "an implementation may also". Briefly describe excluded primitives in a subsection called "excluded primitives".

Pass 1: Identify text that talks about primitives. An API specification, abstract or not, obviously describes primitives -- but note that we are not *only* interested in API specifications. The text describing the 'send' primitive in the API specified in

[RFC0793], for instance, does not say that data transfer is reliable. TCP's reliability is clear, however, from this text in Section 1 of [RFC0793]: "The Transmission Control Protocol (TCP) is intended for use as a highly reliable host-to-host protocol between hosts in packet-switched computer communication networks, and in interconnected systems of such networks."

For the new pass 1 subsection about the protocol you're describing, it is recommendable to begin by copy+pasting all the relevant text parts from the relevant RFCs, then adjust terminology to match the terminology in Section 1 and adjust (shorten!) phrasing to match the general style of the document. Try to formulate everything as a primitive description to make the primitive description as complete as possible (e.g., the "SEND.TCP" primitive in pass 2 is explicitly described as reliably transferring data); if there is text that is relevant for the primitives presented in this pass but still does not fit directly under any primitive, use it as an introduction for your subsection. However, do note that document length is a concern and all the protocols and their services / features are already described in [FA15].

Pass 2: The main goal of this pass is unification of primitives. As input, use your own text from Pass 1, no exterior sources. If you find that something is missing there, fix the text in Pass 1. The list in pass 2 is not done by protocol ("first protocol X, here are all the primitives; then protocol Y, here are all the primitives, ..") but by primitive ("primitive A, implemented this way in protocol X, this way in protocol Y, ..."). We want as many similar pass 2 primitives as possible. This can be achieved, for instance, by not always maintaining a 1:1 mapping between pass 1 and pass 2 primitives, renaming primitives etc. Please consider the primitives that are already there and try to make the ones of the protocol you are describing as much in line with the already existing ones as possible. In other words, we would rather have a primitive with new parameters than a new primitive that allows to send in a particular way.

Please make primitives fit within the already existing categories and subcategories. For each primitive, please follow the style:

o PRIMITIVENAME.PROTOCOL:
  Pass 1 primitive / event:
  Parameters:
  Returns:
  Comments:

The entries "Parameters", "Returns" and "Comments" may be skipped if a primitive has no parameters, no described return value or no comments seem necessary, respectively. Optional parameters must be followed by "(optional)". If a default value is known, provide it too.

Pass 3: the main point of this pass is to identify features that are the result of static properties of protocols, for which all protocols have to be listed together; this is then the final list of all available features. For this, we need a list of features per category (similar categories as in pass 2) along with the protocol supporting it. This should be primarily based on text from pass 2 as input, but text from pass 1 can also be used. Do not use external sources.

Appendix C. Revision information

XXX RFC-Ed please remove this section prior to publication.

-00 (from draft-welzl-taps-transports): this now covers TCP based on all TCP RFCs (this means: if you know of something in any TCP RFC that you think should be addressed, please speak up!) as well as SCTP, exclusively based on [RFC4960]. We decided to also incorporate [RFC6458] for SCTP, but this hasn't happened yet. Terminology made in line with [FA15]. Addressed comments by Karen Nielsen and Gorry Fairhurst; various other fixes. Appendices (TCP overview and how-to-contribute) added.

Authors' Addresses

Michael Welzl
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Phone: +47 22 85 24 20
Email: michawe@ifi.uio.no

Michael Tuexen
Muenster University of Applied Sciences
Stegerwaldstrasse 39
Steinfurt 48565
Germany

Email: tuexen@fh-muenster.de

Naeem Khademi
University of Oslo
PO Box 1080 Blindern
Oslo, N-0316
Norway

Email: naeemk@ifi.uio.no