

TAPS
Internet-Draft
Intended status: Informational
Expires: December 22, 2017

S. Gjessing
M. Welzl
University of Oslo
June 20, 2017

A Minimal Set of Transport Services for TAPS Systems
draft-gjessing-taps-minset-05

Abstract

This draft recommends a minimal set of IETF Transport Services offered by end systems supporting TAPS, and gives guidance on choosing among the available mechanisms and protocols. It is based on the set of transport features given in the TAPS document draft-ietf-taps-transport-services-usage-05.

Status of this Memo

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

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

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

This Internet-Draft will expire on December 22, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Terminology	4
3. The Minimal Set of Transport Features	5
3.1. Flow Creation, Connection and Termination	5
3.2. Flow Group Configuration	6
3.3. Flow Configuration	7
3.4. Data Transfer	7
3.4.1. The Sender	7
3.4.2. The Receiver	8
4. An Abstract MinSet API	9
5. Conclusion	14
6. Acknowledgements	14
7. IANA Considerations	15
8. Security Considerations	15
9. References	15
9.1. Normative References	15
9.2. Informative References	15
Appendix A. Deriving the minimal set	17
A.1. Step 1: Categorization -- The Superset of Transport Features	17
A.1.1. CONNECTION Related Transport Features	19
A.1.2. DATA Transfer Related Transport Features	31
A.2. Step 2: Reduction -- The Reduced Set of Transport Features	35
A.2.1. CONNECTION Related Transport Features	36
A.2.2. DATA Transfer Related Transport Features	37
A.3. Step 3: Discussion	37
A.3.1. Sending Messages, Receiving Bytes	38
A.3.2. Stream Schedulers Without Streams	39
A.3.3. Early Data Transmission	40
A.3.4. Sender Running Dry	41
A.3.5. Capacity Profile	42
A.3.6. Security	42
A.3.7. Packet Size	42
Appendix B. Revision information	43
Authors' Addresses	43

1. Introduction

The task of any system that implements TAPS is to offer transport services to its applications, i.e. the applications running on top of TAPS, without binding them to a particular transport protocol. Currently, the set of transport services that most applications use is based on TCP and UDP; this limits the ability for the network stack to make use of features of other protocols. For example, if a protocol supports out-of-order message delivery but applications always assume that the network provides an ordered bytestream, then the network stack can never utilize out-of-order message delivery: doing so would break a fundamental assumption of the application.

By exposing the transport services of multiple transport protocols, a TAPS system can make it possible to use these services without having to statically bind an application to a specific transport protocol. The first step towards the design of such a system was taken by [RFC8095], which surveys a large number of transports, and [TAPS2], which identifies the specific transport features that are exposed to applications by the protocols TCP, MPTCP, UDP(-Lite) and SCTP as well as the LEDBAT congestion control mechanism. The present draft is based on these documents and follows the same terminology (also listed below).

The number of transport features of current IETF transports is large, and exposing all of them has a number of disadvantages: generally, the more functionality is exposed, the less freedom a TAPS system has to automate usage of the various functions of its available set of transport protocols. Some functions only exist in one particular protocol, and if an application would use them, this would statically tie the application to this protocol, counteracting the purpose of a TAPS system. Also, if the number of exposed features is exceedingly large, a TAPS system might become very hard to use for an application programmer. Taking [TAPS2] as a basis, this document therefore develops a minimal set of transport features, removing the ones that could be harmful to the purpose of a TAPS system but keeping the ones that must be retained for applications to benefit from useful transport functionality.

Applications use a wide variety of APIs today. The transport features in the minimal set in this document must be reflected in **all** network APIs in order for the underlying functionality to become usable everywhere. For example, it does not help an application that talks to a middleware if only the Berkeley Sockets API is extended to offer "unordered message delivery", but the middleware only offers an ordered bytestream. Both the Berkeley Sockets API and the middleware would have to expose the "unordered message delivery" transport feature (alternatively, there may be

interesting ways for certain types of middleware to use some transport features without exposing them, based on knowledge about the applications -- but this is not the general case). In most situations, in the interest of being as flexible and efficient as possible, the best choice will be for a middleware or library to expose at least all of the transport features that are recommended as a "minimal set" here.

This "minimal set" can be implemented one-sided with a fall-back to TCP: i.e., a sender-side TAPS system can talk to a non-TAPS TCP receiver, and a receiver-side TAPS system can talk to a non-TAPS TCP sender. For systems that do not have this requirement, [I-D.trammell-taps-post-sockets] describes a way to extend the functionality of the minimal set such that several of its limitations are removed.

2. Terminology

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 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).

Application-specific knowledge: knowledge that only applications have.

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.

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

3. The Minimal Set of Transport Features

Based on the categorization, reduction and discussion in Appendix A, this section describes the minimal set of transport features that is offered by end systems supporting TAPS.

3.1. Flow Creation, Connection and Termination

A TAPS flow must be "created" before it is connected, to allow for initial configurations to be carried out. All configuration parameters in Section 3.2 and Section 3.3 can be used initially, although some of them may only take effect when the flow has been connected. Configuring a flow early helps a TAPS system make the right decisions. In particular, the "group number" can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not.

A created flow can be queried for the maximum amount of data that an application can possibly expect to have transmitted before or during connection establishment. An application can also give the flow a message for transmission before or during connection establishment; the TAPS system will try to transmit it as early as possible. An application can facilitate sending the message particularly early by marking it as "idempotent"; in this case, the receiving application must be prepared to potentially receive multiple copies of the message.

To be compatible with multiple transports, including streams of a multi-streaming protocol (used as if they were transports themselves), the semantics of opening and closing need to be the most restrictive subset of all of them. For example, TCP's support of half-closed connections can be seen as a feature on top of the more restrictive "ABORT"; this feature cannot be supported because not all protocols used by a TAPS system (including streams of an association) support half-closed connections.

After creation, a flow can be actively connected to the other side using "Connect", or passively listen for incoming connection requests with "Listen". Note that "Connect" may or may not trigger a notification on the listening side. It is possible that the first notification on the listening side is the arrival of the first data that the active side sends (a receiver-side TAPS system could handle

this by continuing a blocking "Listen" call, immediately followed by issuing "Receive", for example). This also means that the active opening side is assumed to be the first side sending data.

A TAPS system can actively close a connection, i.e. terminate it after reliably delivering all remaining data to the peer, or it can abort it, i.e. terminate it without delivering remaining data. Unless all data transfers only used unreliable frame transmission without congestion control, closing a connection is guaranteed to cause an event to notify the peer application that the connection has been closed. Similarly, for anything but unreliable non-congestion-controlled data transfer, aborting a connection will cause an event to notify the peer application that the connection has been aborted. A timeout can be configured to abort a flow when data could not be delivered for too long; timeout-based abortion does not notify the peer application that the connection has been aborted. Because half-closed connections are not supported, when a TAPS host receives a notification that the peer is closing or aborting the flow, the other side may not be able to read outstanding data. This means that unacknowledged data residing in the TAPS system's send buffer may have to be dropped from that buffer upon arrival of a notification to close or abort the flow from the peer.

3.2. Flow Group Configuration

A flow group can be configured with a number of transport features, and there are some notifications to applications about a flow group. Here we list transport features and notifications from Appendix A.2 that sometimes automatically apply to groups of flows (e.g., when a flow is mapped to a stream of a multi-streaming protocol).

Timeout, error notifications:

- o Change timeout for aborting connection (using retransmit limit or time value)
- o Suggest timeout to the peer
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
- o Notification of ICMP error message arrival

Others:

- o Choose a scheduler to operate between flows of a group
- o Obtain ECN field

The following transport features are new or changed, based on the discussion in Appendix A.3:

- o Capacity profile
This describes how an application wants to use its available capacity. Choices can be "lowest possible latency at the expense

of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtcweb-qos]).

3.3. Flow Configuration

Here we list transport features and notifications from Appendix A.2 that only apply to a single flow.

Configure priority or weight for a scheduler

Checksums:

- o Disable checksum when sending
- o Disable checksum requirement when receiving
- o Specify checksum coverage used by the sender
- o Specify minimum checksum coverage required by receiver

3.4. Data Transfer

3.4.1. The Sender

This section discusses how to send data after flow establishment. Section 3.1 discusses the possibility to hand over a message to send before or during establishment.

Here we list per-frame properties that a sender can optionally configure if it hands over a delimited frame for sending with congestion control, taken from Appendix A.2:

- o Configurable Message Reliability
- o Choice between unordered (potentially faster) or ordered delivery of messages
- o Request not to bundle messages
- o Request not to delay the acknowledgement (SACK) of a message

Additionally, an application can hand over delimited frames for unreliable transmission without congestion control (note that such applications should perform congestion control in accordance with [RFC2914]). Then, none of the per-frame properties listed above have any effect, but it is possible to use the transport feature "Specify DF field" to allow/disallow fragmentation.

Following Appendix A.3.7, there are three transport features (two old, one new) and a notification:

- o Get max. transport frame size that may be sent without fragmentation from the configured interface
- This is optional for a TAPS system to offer. It can aid

applications implementing Path MTU Discovery.

- o Get max. transport frame size that may be received from the configured interface
This is optional for a TAPS system to offer.
- o Get maximum transport frame size
Irrespective of fragmentation, there is a size limit for the messages that can be handed over to SCTP or UDP(-Lite); because a TAPS system is independent of the transport, it must allow a TAPS application to query this value -- the maximum size of a frame in an Application-Framed-Bytestream.

There are two more sender-side notifications. These are unreliable, i.e. a TAPS system cannot be assumed to implement them, but they may occur:

- o Notification of send failures
A TAPS system may inform a sender application of a failure to send a specific frame. This was taken over unchanged from Appendix A.2.
- o Notification of draining below a low water mark
A TAPS system can notify a sender application when the TAPS system's filling level of the buffer of unsent data is below a configurable threshold in bytes. Even for TAPS systems that do implement this notification, supporting thresholds other than 0 is optional.

"Notification of draining below a low water mark" is a generic notification that tries to enable uniform access to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification (as discussed in Appendix A.3.4 -- SCTP's "SENDER DRY" is a special case where the threshold (for unsent data) is 0 and there is also no more unacknowledged data in the send buffer). Note that this threshold and its notification should operate across the buffers of the whole TAPS system, i.e. also any potential buffers that the TAPS system itself may use on top of the transport's send buffer.

3.4.2. The Receiver

A receiving application obtains an Application-Framed Bytestream. Similar to TCP's receiver semantics, it is just stream of bytes. If frame boundaries were specified by the sender, a receiver-side TAPS system will still not inform the receiving application about them. Within the bytestream, frames themselves will always stay intact (partial frames are not supported - see Appendix A.3.1). Different from TCP's semantics, there is no guarantee that all frames in the

bytestream are transmitted from the sender to the receiver, and that all of them are in the same sequence in which they were handed over by the sender. If an application is aware of frame delimiters in the bytestream, and if the sender-side application has informed the TAPS system about these boundaries and about potentially relaxed requirements regarding the sequence of frames or per-frame reliability, frames within the receiver-side bytestream may be out-of-order or missing.

4. An Abstract MinSet API

Here we present an abstract API that a TAPS system can implement. This API is derived from the description in the previous section. The primitives of this API can be implemented in various ways. For example, information that is provided to an application can either be offered via a primitive that is polled, or via an asynchronous notification. The API offers specific primitives to configure such asynchronous call-backs.

CREATE (flow-group-id)
Returns: flow-id

Create a flow and associate it with an existing or new flow group number. The group number can influence the TAPS system to implement a TAPS flow as a stream of a multi-streaming protocol's existing association or not.

CONFIGURE_TIMEOUT (flow-group-id [timeout] [peer_timeout]
[retrans_notify])

This configures timeouts for all flows in a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

timeout: a timeout value for aborting connections, in seconds
peer_timeout: a timeout value to be suggested to the peer (if possible), in seconds
retrans_notify: the number of retransmissions after which the application should be notified of "Excessive Retransmissions"

CONFIGURE_CHECKSUM (flow-id [send [send_length]] [receive
[receive_length]])

This configures the usage of checksums for a flow in a group.

Configuration should generally be carried out as early as possible, ideally before the flow is connected, to aid the TAPS system's decision taking. "send" parameters concern using a checksum when sending, "receive" parameters concern requiring a checksum when receiving. There is no guarantee that any checksum limitations will indeed be enforced; all defaults are: "full coverage, checksum enabled".

PARAMETERS:

send: boolean, enable / disable usage of a checksum
send_length: if send is true, this optional parameter can provide the desired coverage of the checksum in bytes
receive: boolean, enable / disable requiring a checksum
receive_length: if receive is true, this optional parameter can provide the required minimum coverage of the checksum in bytes

CONFIGURE_URGENCY (flow-group-id [scheduler] [capacity_profile] [low_watermark])

This carries out configuration related to the urgency of sending data on flows of a group. Configuration should generally be carried out as early as possible, ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

scheduler: a number to identify the type of scheduler that should be used to operate between flows in the group (no guarantees given). Future versions of this document will be self contained, but for now we suggest the schedulers defined in [I-D.ietf-tsvwg-sctp-ndata].
capacity_profile: a number to identify how an application wants to use its available capacity. Future versions of this document will be self contained, but for now choices can be "lowest possible latency at the expense of overhead" (which would disable any Nagle-like algorithm), "scavenger", and some more values that help determine the DSCP value for a flow (e.g. similar to table 1 in [I-D.ietf-tsvwg-rtweb-qos]).
low_watermark: a buffer limit (in bytes); when the sender has less than low_watermark bytes in the buffer, the application may be notified. Notifications are not guaranteed, and supporting watermark numbers greater than 0 is not guaranteed.

CONFIGURE_PRIORITY (flow-id priority)

This configures a flow's priority or weight for a scheduler. Configuration should generally be carried out as early as possible,

ideally before flows are connected, to aid the TAPS system's decision taking.

PARAMETERS:

priority: future versions of this document will be self contained, but for now we suggest the priority as described in [I-D.ietf-tsvwg-sctp-ndata].

NOTIFICATIONS

Returns: flow-group-id notification_type

This is fired when an event occurs, notifying the application about something happening in relation to a flow group. Notification types are:

Excessive Retransmissions: the configured (or a default) number of retransmissions has been reached, yielding this early warning below an abortion threshold

ICMP Arrival (parameter: ICMP message): an ICMP packet carrying the conveyed ICMP message has arrived.

ECN Arrival (parameter: ECN value): a packet carrying the conveyed ECN value has arrived. This can be useful for applications implementing congestion control.

Timeout (parameter: s seconds): data could not be delivered for s seconds.

Close: the peer has closed the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

Abort: the peer has aborted the connection. The peer has no more data to send, and will not read more data. Data that is in transit or resides in the local send buffer will be discarded.

Drain: the send buffer has either drained below the configured low water mark or it has become completely empty.

Path Change (parameter: path identifier): the path has changed; the path identifier is a number that can be used to determine a previously used path is used again (e.g., the TAPS system has switched from one interface to the other and back).

Send Failure (parameter: frame identifier): this informs the application of a failure to send a specific frame. There can be a send failure without this notification happening.

QUERY_PROPERTIES (flow-group-id property_identifier)

Returns: requested property (see below)

This allows to query some properties of a flow group. Return values per property identifier are:

- o The maximum frame size that may be sent without fragmentation, in bytes
- o The maximum transport frame size that can be sent, in bytes
- o The maximum transport frame size that can be received, in bytes
- o The maximum amount of data that can possibly be sent before or during connection establishment, in bytes

CONNECT (flow-id dst_addr)

Connects a flow. This primitive may or may not trigger a notification (continuing LISTEN) on the listening side. If a send precedes this call, then data may be transmitted with this connect.

PARAMETERS:

dst_addr: the destination transport address to connect to

LISTEN (flow-id)

Blocking passive connect, listening on all interfaces. This may not be the direct result of the peer calling CONNECT - it may also be invoked upon reception of the first block of data. In this case, RECEIVE_FRAME is invoked immediately after.

SEND_FRAME (flow-id frame [reliability] [ordered] [bundle] [delack] [fragment] [idempotent])

Sends an application frame. No guarantees are given about the preservation of frame boundaries to the peer; if frame boundaries are needed, the receiving application at the peer must know about them beforehand. Note that this call can already be used before a flow is connected. All parameters refer to the frame that is being handed over.

PARAMETERS:

reliability: this parameter is used to convey a choice of: fully reliable, unreliable without congestion control (which is guaranteed), unreliable, partially reliable (how to configure: TBD, probably using a time value). The latter two choices are not guaranteed and may result in full reliability.

ordered: this boolean parameter lets an application choose between ordered message delivery (true) and possibly unordered, potentially faster message delivery (false).

bundle: a boolean that expresses a preference for allowing to bundle frames (true) or not (false). No guarantees are given.

delack: a boolean that, if false, lets an application request that the peer would not delay the acknowledgement for this frame.

fragment: a boolean that expresses a preference for allowing to fragment frames (true) or not (false), at the IP level. No guarantees are given.

idempotent: a boolean that expresses whether a frame is idempotent (true) or not (false). Idempotent frames may arrive multiple times at the receiver. When data is idempotent it can be used by the receiver immediately on a connection establishment attempt. Thus, if SEND_FRAME is used before connecting, stating that a frame is idempotent facilitates transmitting it to the peer application particularly early.

CLOSE (flow-id)

Closes the flow after all outstanding data is reliably delivered to the peer (if reliable data delivery was requested). In case reliable or partially reliable data delivery was requested earlier, the peer is notified of the CLOSE.

ABORT (flow-id)

Aborts the flow without delivering outstanding data to the peer. In case reliable or partially reliable data delivery was requested earlier, the peer is notified of the ABORT.

RECEIVE_FRAME (flow-id buffer)

This receives a block of data. This block may or may not correspond to a sender-side frame, i.e. the receiving application is not informed about frame boundaries. However, if the sending application has allowed that frames are not fully reliably transferred, or delivered out of order, then such re-ordering or unreliability may be reflected per frame in the arriving data. Frames will always stay intact - i.e. if an incomplete frame is contained at the end of the arriving data block, this frame is guaranteed to continue in the next arriving data block.

PARAMETERS:

buffer: the buffer where the received data will be stored.

5. Conclusion

By decoupling applications from transport protocols, a TAPS system provides a different abstraction level than the Berkeley sockets interface. As with high- vs. low-level programming languages, a higher abstraction level allows more freedom for automation below the interface, yet it takes some control away from the application programmer. This is the design trade-off that a TAPS system developer is facing, and this document provides guidance on the design of this abstraction level. Some transport features are currently rarely offered by APIs, yet they must be offered or they can never be used ("functional" transport features). Other transport features are offered by the APIs of the protocols covered here, but not exposing them in a TAPS API would allow for more freedom to automate protocol usage in a TAPS system.

The minimal set presented in this document is an effort to find a middle ground that can be recommended for TAPS systems to implement, on the basis of the transport features discussed in [TAPS2]. This middle ground eliminates a large number of transport features because they do not require application-specific knowledge, but rather rely on knowledge about the network or the Operating System. This leaves us with an unanswered question about how exactly a TAPS system should automate using all these transport features.

In some cases, it may be best to not entirely automate the decision making, but leave it up to a system-wide policy. For example, when multiple paths are available, a system policy could guide the decision on whether to connect via a WiFi or a cellular interface. Such high-level guidance could also be provided by application developers, e.g. via a primitive that lets applications specify such preferences. As long as this kind of information from applications is treated as advisory, it will not lead to a permanent protocol binding and does therefore not limit the flexibility of a TAPS system. Decisions to add such primitives are therefore left open to TAPS system designers.

6. Acknowledgements

The authors would like to thank the participants of the TAPS Working Group and the NEAT research project for valuable input to this document. We especially thank Michael Tuexen for help with TAPS flow connection establishment/teardown and Gorry Fairhurst for his

suggestions regarding fragmentation and packet sizes. 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

Authentication, confidentiality protection, and integrity protection are identified as transport features by [RFC8095]. As currently deployed in the Internet, these features are generally provided by a protocol or layer on top of the transport protocol; no current full-featured standards-track transport protocol provides all of these transport features on its own. Therefore, these transport features are not considered in this document, with the exception of native authentication capabilities of TCP and SCTP for which the security considerations in [RFC5925] and [RFC4895] apply.

9. References

9.1. Normative References

[RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<http://www.rfc-editor.org/info/rfc8095>>.

[TAPS2] Welzl, M., Tuexen, M., and N. Khademi, "On the Usage of Transport Features Provided by IETF Transport Protocols", draft-ietf-taps-transport-services-usage-05 (work in progress), May 2017.

9.2. Informative References

[COBS] Cheshire, S. and M. Baker, "Consistent Overhead Byte Stuffing", September 1997, <<http://stuartcheshire.org/papers/COBSforToN.pdf>>.

[I-D.ietf-tsvwg-rtcweb-qos]

Jones, P., Dhesikan, S., Jennings, C., and D. Druta, "DSCP Packet Markings for WebRTC QoS", draft-ietf-tsvwg-rtcweb-qos-18 (work in progress), August 2016.

[I-D.ietf-tsvwg-sctp-ndata]

Stewart, R., Tuexen, M., Loreto, S., and R. Seggelmann, "Stream Schedulers and User Message Interleaving for the Stream Control Transmission Protocol", draft-ietf-tsvwg-sctp-ndata-10 (work in progress), April 2017.

[I-D.trammell-taps-post-sockets]

Trammell, B., Perkins, C., Pauly, T., and M. Kuehlewind, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-00 (work in progress), March 2017.

[LBE-draft]

Bless, R., "A Lower Effort Per-Hop Behavior (LE PHB)", draft-tsvwg-le-phb-01 (work in progress), February 2017.

[RFC2914] Floyd, S., "Congestion Control Principles", BCP 41, RFC 2914, DOI 10.17487/RFC2914, September 2000, <<http://www.rfc-editor.org/info/rfc2914>>.

[RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, DOI 10.17487/RFC4895, August 2007, <<http://www.rfc-editor.org/info/rfc4895>>.

[RFC4987] Eddy, W., "TCP SYN Flooding Attacks and Common Mitigations", RFC 4987, DOI 10.17487/RFC4987, August 2007, <<http://www.rfc-editor.org/info/rfc4987>>.

[RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<http://www.rfc-editor.org/info/rfc5925>>.

[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>>.

[RFC6525] Stewart, R., Tuexen, M., and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration", RFC 6525, DOI 10.17487/RFC6525, February 2012,

<<http://www.rfc-editor.org/info/rfc6525>>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.

[WWDC2015] Lakhera, P. and S. Cheshire, "Your App and Next Generation Networks", Apple Worldwide Developers Conference 2015, San Francisco, USA, June 2015, <<https://developer.apple.com/videos/wwdc/2015/?id=719>>.

Appendix A. Deriving the minimal set

We approach the construction of a minimal set of transport features in the following way:

1. Categorization: the superset of transport features from [TAPS2] is presented, and transport features are categorized for later reduction.
2. Reduction: a shorter list of transport features is derived from the categorization in the first step. This removes all transport features that do not require application-specific knowledge or cannot be implemented with TCP.
3. Discussion: the resulting list shows a number of peculiarities that are discussed, to provide a basis for constructing the minimal set.
4. Construction: Based on the reduced set and the discussion of the transport features therein, a minimal set is constructed.

The first three steps as well as the underlying rationale for constructing the minimal set are described in this appendix. The minimal set itself is described in Section 3.

A.1. Step 1: Categorization -- The Superset of Transport Features

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

1. CONNECTION related transport features
 - ESTABLISHMENT
 - AVAILABILITY
 - MAINTENANCE
 - TERMINATION
2. DATA Transfer Related transport features
 - Sending Data
 - Receiving Data
 - Errors

We assume that TAPS applications have no specific requirements that

need knowledge about the network, e.g. regarding the choice of network interface or the end-to-end path. Even with these assumptions, there are 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 transport features that we call "Functional".

Functional transport features provide functionality that cannot be used without the application knowing about them, or else they violate assumptions that might cause the application to fail. For example, unordered message delivery is a functional transport feature: it cannot be used without the application knowing about it because the application's assumption could be that messages arrive in order. Failure includes any change of the application behavior that is not performance oriented, e.g. security.

"Change DSCP" and "Disable Nagle algorithm" are examples of transport features that we call "Optimizing": if a TAPS system autonomously decides to enable or disable them, an application will not fail, but a TAPS system may be able to communicate more efficiently if the application is in control of this optimizing transport feature. These transport features require application-specific knowledge (e.g., about delay/bandwidth requirements or the length of future data blocks that are to be transmitted).

The transport features of IETF transport protocols that do not require application-specific knowledge and could therefore be transparently utilized by a TAPS system are called "Automatable".

Finally, some transport features are aggregated and/or slightly changed in the TAPS API. These transport features are marked as "ADDED". The corresponding transport features are automatable, and they are listed immediately below the "ADDED" transport feature.

In this description, transport services are presented following the nomenclature "CATEGORY.[SUBCATEGORY].SERVICENAME.PROTOCOL", equivalent to "pass 2" in [TAPS2]. The PROTOCOL name "UDP(-Lite)" is used when transport features are equivalent for UDP and UDP-Lite; the PROTOCOL name "TCP" refers to both TCP and MPTCP. We also sketch how some of the TAPS transport services can be implemented. For all transport features that are categorized as "functional" or "optimizing", and for which no matching TCP primitive exists in "pass 2" of [TAPS2], a brief discussion on how to fall back to TCP is included.

We designate some transport features as "automatable" on the basis of a broader decision that affects multiple transport features:

- o Most transport features that are related to multi-streaming were designated as "automatable". This was done because the decision on whether to use multi-streaming or not does not depend on application-specific knowledge. This means that a connection that is exhibited to an application could be implemented by using a single stream of an SCTP association instead of mapping it to a complete SCTP association or TCP connection. This could be achieved by using more than one stream when an SCTP association is first established (CONNECT.SCTP parameter "outbound stream count"), maintaining an internal stream number, and using this stream number when sending data (SEND.SCTP parameter "stream number"). Closing or aborting a connection could then simply free the stream number for future use. This is discussed further in Appendix A.3.2.
- o All transport features that are related to using multiple paths or the choice of the network interface were designated as "automatable". Choosing a path or an interface does not depend on application-specific knowledge. For example, "Listen" could always listen on all available interfaces and "Connect" could use the default interface for the destination IP address.

A.1.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
Protocols: TCP, SCTP, UDP(-Lite)
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 transport feature that is defined by the application protocol.
Implementation: via CONNECT.TCP, CONNECT.SCTP or CONNECT.UDP(-Lite).
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Request multiple streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require

application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Specify number of attempts and/or timeout for the first establishment message
Protocols: TCP, SCTP
Functional because this is closely related to potentially assumed reliable data delivery for data that is sent before or during connection establishment.
Implementation: Using a parameter of CONNECT.TCP and CONNECT.SCTP.
- o Obtain multiple sockets
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
Implementation: via a boolean parameter in CONNECT.MPTCP.
Fall-back to TCP: Do nothing.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in CONNECT.TCP and CONNECT.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in CONNECT.SCTP.
Fall-back to TCP: not possible.

- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in CONNECT.SCTP.
- o Hand over a message to transfer (possibly multiple times) before connection establishment
Protocols: TCP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.TCP.
- o Hand over a message to transfer during connection establishment
Protocols: SCTP
Functional because this can only work if the message is limited in size, making it closely tied to properties of the data that an application sends or expects to receive.
Implementation: via a parameter in CONNECT.SCTP.
- o Enable UDP encapsulation with a specified remote UDP port number
Protocols: SCTP
Automatable because UDP encapsulation relates to knowledge about the network, not the application.

AVAILABILITY:

- o Listen
Protocols: TCP, SCTP, UDP(-Lite)
Functional because the notion of accepting connection requests is often reflected in applications as an expectation to be able to communicate after a "Listen" succeeded, with a communication sequence relating to this transport feature that is defined by the application protocol.
ADDED. This differs from the 3 automatable transport features below in that it leaves the choice of interfaces for listening open.
Implementation: by listening on all interfaces via LISTEN.TCP (not providing a local IP address) or LISTEN.SCTP (providing SCTP port number / address pairs for all local IP addresses).
- o Listen, 1 specified local interface
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.

- o Listen, N specified local interfaces
Protocols: SCTP
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Listen, all local interfaces
Protocols: TCP, SCTP, UDP(-Lite)
Automatable because decisions about local interfaces relate to knowledge about the network and the Operating System, not the application.
- o Specify which IP Options must always be used
Protocols: TCP, UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Disable MPTCP
Protocols: MPTCP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Configure authentication
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via parameters in LISTEN.TCP and LISTEN.SCTP.
Fall-back to TCP: With TCP, this allows to configure Master Key Tuples (MKTs) to authenticate complete segments (including the TCP IPv4 pseudoheader, TCP header, and TCP data). With SCTP, this allows to specify which chunk types must always be authenticated. Authenticating only certain chunk types creates a reduced level of security that is not supported by TCP; to be compatible, this should therefore only allow to authenticate all chunk types. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Obtain requested number of streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Limit the number of inbound streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.

- o Indicate (and/or obtain upon completion) an Adaptation Layer via an adaptation code point
Protocols: SCTP
Functional because it allows to send extra data for the sake of identifying an adaptation layer, which by itself is application-specific.
Implementation: via a parameter in LISTEN.SCTP.
Fall-back to TCP: not possible.
- o Request to negotiate interleaving of user messages
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in LISTEN.SCTP.

MAINTENANCE:

- o 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.
Implementation: via CHANGE-TIMEOUT.TCP or CHANGE-TIMEOUT.SCTP.
- o Suggest timeout to the peer
Protocols: TCP
Functional because this is closely related to potentially assumed reliable data delivery.
Implementation: via CHANGE-TIMEOUT.TCP.
- 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.
Implementation: via DISABLE-NAGLE.TCP and DISABLE-NAGLE.SCTP.
- o Request an immediate heartbeat, returning success/failure
Protocols: SCTP
Automatable because this informs about network-specific knowledge.
- 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.
Implementation: via ERROR.TCP.

- o Add path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Remove path
Protocols: MPTCP, SCTP
MPTCP Parameters: source-IP; source-Port; destination-IP;
destination-Port
SCTP Parameters: local IP address
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Set primary path
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Suggest primary path to the peer
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Configure Path Switchover
Protocols: SCTP
Automatable because the usage of multiple paths to communicate to the same end host relates to knowledge about the network, not the application.
- o Obtain status (query or notification)
Protocols: SCTP, MPTCP
SCTP parameters: association connection state; destination transport address list; destination transport address reachability states; current local and peer receiver window size; current local 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; MTU per path; interleaving supported yes/no
MPTCP parameters: subflow-list (identified by source-IP; source-Port; destination-IP; destination-Port)
Automatable because these parameters relate to knowledge about the

network, not the application.

- o Specify DSCP field
Protocols: TCP, SCTP, UDP(-Lite)
Optimizing because choosing a suitable DSCP value requires application-specific knowledge.
Implementation: via SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite)
- o Notification of ICMP error message arrival
Protocols: TCP, UDP(-Lite)
Optimizing because these messages can inform about success or failure of functional transport features (e.g., host unreachable relates to "Connect")
Implementation: via ERROR.TCP or ERROR.UDP(-Lite).
- o Obtain information about interleaving support
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: via a parameter in GETINTERL.SCTP.
- o Change authentication parameters
Protocols: TCP, SCTP
Functional because this has a direct influence on security.
Implementation: via SET_AUTH.TCP and SET_AUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to adjust key_id, key, and hmac_id. With TCP, this allows to change the preferred outgoing MKT (current_key) and the preferred incoming MKT (rnext_key), respectively, for a segment that is sent on the connection. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].
- o Obtain authentication information
Protocols: SCTP
Functional because authentication decisions may have been made by the peer, and this has an influence on the necessary application-level measures to provide a certain level of security.
Implementation: via GETAUTH.SCTP.
Fall-back to TCP: With SCTP, this allows to obtain key_id and a chunk list. With TCP, this allows to obtain current_key and rnext_key from a previously received segment. Key material must be provided in a way that is compatible with both [RFC4895] and [RFC5925].

- o Reset Stream
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Notification of Stream Reset
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Reset Association
Protocols: SCTP
Functional because it affects "Obtain a message delivery number", which is functional.
Implementation: via RESETASSOC.SCTP.
Fall-back to TCP: not possible.
- o Notification of Association Reset
Protocols: STCP
Functional because it affects "Obtain a message delivery number", which is functional.
Implementation: via RESETASSOC-EVENT.SCTP.
Fall-back to TCP: not possible.
- o Add Streams
Protocols: SCTP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Notification of Added Stream
Protocols: STCP
Automatable because using multi-streaming does not require application-specific knowledge.
Implementation: see Appendix A.3.2.
- o Choose a scheduler to operate between streams of an association
Protocols: SCTP
Optimizing because the scheduling decision requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using SETSTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.

- o Configure priority or weight for a scheduler
Protocols: SCTP
Optimizing because the priority or weight requires application-specific knowledge. However, if a TAPS system would not use this, or wrongly configure it on its own, this would only affect the performance of data transfers; the outcome would still be correct within the "best effort" service model.
Implementation: using CONFIGURESTREAMSCHEDULER.SCTP.
Fall-back to TCP: do nothing.
- o Configure send buffer size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application (see also the discussion in Appendix A.3.4).
- o Configure receive buffer (and rwnd) size
Protocols: SCTP
Automatable because this decision relates to knowledge about the network and the Operating System, not the application.
- o Configure message fragmentation
Protocols: SCTP
Automatable because fragmentation relates to knowledge about the network and the Operating System, not the application.
Implementation: by always enabling it with CONFIG_FRAGMENTATION.SCTP and auto-setting the fragmentation size based on network or Operating System conditions.
- o Configure PMTUD
Protocols: SCTP
Automatable because Path MTU Discovery relates to knowledge about the network, not the application.
- o Configure delayed SACK timer
Protocols: SCTP
Automatable because the receiver-side decision to delay sending SACKs relates to knowledge about the network, not the application (it can be relevant for a sending application to request not to delay the SACK of a message, but this is a different transport feature).
- o Set Cookie life value
Protocols: SCTP
Functional because it relates to security (possibly weakened by keeping a cookie very long) versus the time between connection establishment attempts. Knowledge about both issues can be application-specific.

Fall-back to TCP: the closest specified TCP functionality is the cookie in TCP Fast Open; for this, [RFC7413] states that the server "can expire the cookie at any time to enhance security" and section 4.1.2 describes an example implementation where updating the key on the server side causes the cookie to expire. Alternatively, for implementations that do not support TCP Fast Open, this transport feature could also affect the validity of SYN cookies (see Section 3.6 of [RFC4987]).

- o Set maximum burst
Protocols: SCTP
Automatable because it relates to knowledge about the network, not the application.
- o Configure size where messages are broken up for partial delivery
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: do nothing. Since TCP does not deliver messages, partial or not, this will have no effect on TCP.
- o Disable checksum when sending
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_ENABLED.UDP.
Fall-back to TCP: do nothing.
- o Disable checksum requirement when receiving
Protocols: UDP
Functional because application-specific knowledge is necessary to decide whether it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_REQUIRED.UDP.
Fall-back to TCP: do nothing.
- o Specify checksum coverage used by the sender
Protocols: UDP-Lite
Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.
Implementation: via SET_CHECKSUM_COVERAGE.UDP-Lite.
Fall-back to TCP: do nothing.
- o Specify minimum checksum coverage required by receiver
Protocols: UDP-Lite
Functional because application-specific knowledge is necessary to decide for which parts of the data it can be acceptable to lose data integrity.

Implementation: via SET_MIN_CHECKSUM_COVERAGE.UDP-Lite.
Fall-back to TCP: do nothing.

- o Specify DF field
Protocols: UDP(-Lite)
Optimizing because the DF field can be used to carry out Path MTU Discovery, which can lead an application to choose message sizes that can be transmitted more efficiently.
Implementation: via MAINTENANCE.SET_DF.UDP(-Lite) and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing. With TCP the sender is not in control of transport message sizes, making this functionality irrelevant.
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can lead an application to choose message sizes that can be transmitted more efficiently.
- o Get max. transport-message size that may be received from the configured interface
Protocols: UDP(-Lite)
Optimizing because this can, for example, influence an application's memory management.
- o Specify TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because a TAPS system can use a large enough system default to avoid communication failures. Allowing an application to configure it differently can produce notifications of ICMP error message arrivals that yield information which only relates to knowledge about the network, not the application.
- o Obtain TTL/Hop count field
Protocols: UDP(-Lite)
Automatable because the TTL/Hop count field relates to knowledge about the network, not the application.
- o Specify ECN field
Protocols: UDP(-Lite)
Automatable because the ECN field relates to knowledge about the network, not the application.
- o Obtain ECN field
Protocols: UDP(-Lite)
Optimizing because this information can be used by an application to better carry out congestion control (this is relevant when

choosing a data transmission transport service that does not already do congestion control).

- o Specify IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Obtain IP Options
Protocols: UDP(-Lite)
Automatable because IP Options relate to knowledge about the network, not the application.
- o Enable and configure a "Low Extra Delay Background Transfer"
Protocols: A protocol implementing the LEDBAT congestion control mechanism
Optimizing because whether this service is appropriate or not depends on application-specific knowledge. However, wrongly using this will only affect the speed of data transfers (albeit including other transfers that may compete with the TAPS transfer in the network), so it is still correct within the "best effort" service model.
Implementation: via CONFIGURE.LEDBAT and/or SET_DSCP.TCP / SET_DSCP.SCTP / SET_DSCP.UDP(-Lite) [LBE-draft].
Fall-back to TCP: do nothing.

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 transport feature that is defined by the application protocol.
Implementation: via CLOSE.TCP and CLOSE.SCTP.
- 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. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.TCP and ABORT.SCTP.

- o Abort without delivering remaining data, not causing an event informing the application on the other side
Protocols: UDP(-Lite)
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. On both sides of a connection, an application protocol may define a communication sequence relating to this transport feature.
Implementation: via ABORT.UDP(-Lite).
Fall-back to TCP: stop using the connection, wait for a timeout.
- o 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.
Implementation: via TIMEOUT.TCP and TIMEOUT.SCTP.

A.1.2. DATA Transfer Related Transport Features

A.1.2.1. Sending Data

- o Reliably transfer data, with congestion control
Protocols: TCP, SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.TCP and SEND.SCTP.
- o Reliably transfer a message, with congestion control
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP and SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver. Inform the application of the result.
- o Unreliably transfer a message
Protocols: SCTP, UDP(-Lite)
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
ADDED. This differs from the 2 automatable transport features below in that it leaves the choice of congestion control open.
Implementation: via SEND.SCTP or SEND.UDP or SEND.TCP. With SEND.TCP, messages will not be identifiable by the receiver. Inform the application of the result.

- o Unreliably transfer a message, with congestion control
Protocols: SCTP
Automatable because congestion control relates to knowledge about the network, not the application.
- o Unreliably transfer a message, without congestion control
Protocols: UDP(-Lite)
Automatable because congestion control relates to knowledge about the network, not the application.
- o Configurable Message Reliability
Protocols: SCTP
Optimizing because only applications know about the time criticality of their communication, and reliably transferring a message is never incorrect for the receiver of a potentially unreliable data transfer, it is just slower.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and ignoring this configuration: based on the assumption of the best-effort service model, unnecessarily delivering data does not violate application expectations. Moreover, it is not possible to associate the requested reliability to a "message" in TCP anyway.
- o Choice of stream
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable. Implementation: see Appendix A.3.2.
- o Choice of path (destination address)
Protocols: SCTP
Automatable because it requires using multiple sockets, but obtaining multiple sockets in the CONNECTION.ESTABLISHMENT category is automatable.
- o Choice between unordered (potentially faster) or ordered delivery of messages
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and always sending data ordered: based on the assumption of the best-effort service model, ordered delivery may just be slower and does not violate application expectations. Moreover, it is not possible to associate the requested delivery order to a "message" in TCP anyway.

- o 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.
Implementation: via SEND.SCTP.
Fall-back to TCP: By using SEND.TCP and DISABLE-NAGLE.TCP to disable the Nagle algorithm when the request is made and enable it again when the request is no longer made. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
- o Specifying a "payload protocol-id" (handed over as such by the receiver)
Protocols: SCTP
Functional because it allows to send extra application data with every message, for the sake of identification of data, which by itself is application-specific.
Implementation: SEND.SCTP.
Fall-back to TCP: not possible.
- o Specifying a key id to be used to authenticate a message
Protocols: SCTP
Functional because this has a direct influence on security.
Implementation: via a parameter in SEND.SCTP.
Fall-back to TCP: This could be emulated by using SET_AUTH.TCP before and after the message is sent. Note that this is not fully equivalent because it relates to the time of issuing the request rather than a specific message.
- o Request not to delay the acknowledgement (SACK) of a message
Protocols: SCTP
Optimizing because only an application knows for which message it wants to quickly be informed about success / failure of its delivery.
Fall-back to TCP: do nothing.

A.1.2.2. Receiving Data

- o Receive data (with no message delineation)
Protocols: TCP
Functional because a TAPS system must be able to send and receive data.
Implementation: via RECEIVE.TCP
- o Receive a message
Protocols: SCTP, UDP(-Lite)
Functional because this is closely tied to properties of the data

that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP and RECEIVE.UDP(-Lite).
Fall-back to TCP: not possible.

- o Choice of stream to receive from
Protocols: SCTP
Automatable because it requires using multiple streams, but requesting multiple streams in the CONNECTION.ESTABLISHMENT category is automatable.
Implementation: see Appendix A.3.2.
- o 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.
Implementation: via RECEIVE.SCTP.
Fall-back to TCP: do nothing: this information is not available with TCP.
- o Obtain a message delivery number
Protocols: SCTP
Functional because this number can let applications detect and, if desired, correct reordering. Whether messages are in the correct order or not is closely tied to properties of the data that an application sends or expects to receive.
Implementation: via RECEIVE.SCTP.
Fall-back to TCP: not possible.

A.1.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o Notification of send failures
Protocols: SCTP, UDP(-Lite)
Functional because this notifies that potentially assumed reliable data delivery is no longer provided.
ADDED. This differs from the 2 automatable transport features below in that it does not distinguish between unsent and unacknowledged messages.
Implementation: via SENDFAILURE-EVENT.SCTP and SEND_FAILURE.UDP(-Lite).
Fall-back to TCP: do nothing: this notification is not available and will therefore not occur with TCP.

- o Notification of an unsent (part of a) message
Protocols: SCTP, UDP(-Lite)
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification of an unacknowledged (part of a) message
Protocols: SCTP
Automatable because the distinction between unsent and unacknowledged is network-specific.
- o Notification that the stack has no more user data to send
Protocols: SCTP
Optimizing because reacting to this notification requires the application to be involved, and ensuring that the stack does not run dry of data (for too long) can improve performance.
Fall-back to TCP: do nothing. See also the discussion in Appendix A.3.4.
- o Notification to a receiver that a partial message delivery has been aborted
Protocols: SCTP
Functional because this is closely tied to properties of the data that an application sends or expects to receive.
Fall-back to TCP: do nothing. This notification is not available and will therefore not occur with TCP.

A.2. Step 2: Reduction -- The Reduced Set of Transport Features

By hiding automatable transport features from the application, a TAPS system can gain opportunities to automate the usage of 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, system-wide configurations regarding the usage of multiple interfaces can better be exploited if the choice of the interface is not entirely up to the application. Therefore, since they are not strictly necessary to expose in a TAPS system, we do not include automatable transport features in the reduced set of transport features. This leaves us with only the transport features that are either optimizing or functional.

A TAPS system should be able to fall back to TCP or UDP if alternative transport protocols are found not to work. Here we only consider falling back to TCP. For some transport features, it was identified that no fall-back to TCP is possible. This eliminates the possibility to use TCP whenever an application makes use of one of these transport features. Thus, we only keep the functional and

optimizing transport features for which a fall-back to TCP is possible in our reduced set. "Reset Association" and "Notification of Association Reset" are only functional because of their relationship to "Obtain a message delivery number", which is functional. Because "Obtain a message delivery number" does not have a fall-back to TCP, none of these three transport features are included in the reduced set.

A.2.1. CONNECTION Related Transport Features

ESTABLISHMENT:

- o Connect
- o Specify number of attempts and/or timeout for the first establishment message
- o Configure authentication
- o Hand over a message to transfer (possibly multiple times) before connection establishment
- o Hand over a message to transfer during connection establishment

AVAILABILITY:

- o Listen
- o Configure authentication

MAINTENANCE:

- o Change timeout for aborting connection (using retransmit limit or time value)
- o Suggest timeout to the peer
- o Disable Nagle algorithm
- o Notification of Excessive Retransmissions (early warning below abortion threshold)
- o Specify DSCP field
- o Notification of ICMP error message arrival
- o Change authentication parameters
- o Obtain authentication information
- o Set Cookie life value
- o Choose a scheduler to operate between streams of an association
- o Configure priority or weight for a scheduler
- o Configure size where messages are broken up for partial delivery
- o Disable checksum when sending
- o Disable checksum requirement when receiving
- o Specify checksum coverage used by the sender
- o Specify minimum checksum coverage required by receiver
- o Specify DF field
- o Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface
- o Get max. transport-message size that may be received from the configured interface

- o Obtain ECN field
- o Enable and configure a "Low Extra Delay Background Transfer"

TERMINATION:

- o Close after reliably delivering all remaining data, causing an event informing the application on the other side
- o Abort without delivering remaining data, causing an event informing the application on the other side
- o Abort without delivering remaining data, not causing an event informing the application on the other side
- o Timeout event when data could not be delivered for too long

A.2.2. DATA Transfer Related Transport Features

A.2.2.1. Sending Data

- o Reliably transfer data, with congestion control
- o Reliably transfer a message, with congestion control
- o Unreliably transfer a message
- o Configurable Message Reliability
- o Choice between unordered (potentially faster) or ordered delivery of messages
- o Request not to bundle messages
- o Specifying a key id to be used to authenticate a message
- o Request not to delay the acknowledgement (SACK) of a message

A.2.2.2. Receiving Data

- o Receive data (with no message delineation)
- o Information about partial message arrival

A.2.2.3. Errors

This section describes sending failures that are associated with a specific call to in the "Sending Data" category (Appendix A.1.2.1).

- o Notification of send failures
- o Notification that the stack has no more user data to send
- o Notification to a receiver that a partial message delivery has been aborted

A.3. Step 3: Discussion

The reduced set in the previous section exhibits a number of peculiarities, which we will discuss in the following.

A.3.1. Sending Messages, Receiving Bytes

There are several transport features related to sending, but only a single transport feature related to receiving: "Receive data (with no message delineation)" (and, strangely, "information about partial message arrival"). Notably, the transport feature "Receive a message" is also the only non-automatable transport feature of UDP(-Lite) that had to be removed because no fall-back to TCP is possible.

To support these TCP receiver semantics, we define an "Application-Framed Bytestream" (AFra-Bytestream). AFra-Bytestreams allow senders to operate on messages while minimizing changes to the TCP socket API. In particular, nothing changes on the receiver side - data can be accepted via a normal TCP socket.

In an AFra-Bytestream, the sending application can optionally inform the transport about frame boundaries and required properties per frame (configurable order and reliability, or embedding a request not to delay the acknowledgement of a frame). Whenever the sending application specifies per-frame properties that relax the notion of reliable in-order delivery of bytes, it must assume that the receiving application is 1) able to determine frame boundaries, provided that frames are always kept intact, and 2) able to accept these relaxed per-frame properties. Any signaling of such information to the peer is up to an application-layer protocol and considered out of scope of this document.

For example, if an application requests to transfer fixed-size messages of 100 bytes with partial reliability, this needs the receiving application to be prepared to accept data in chunks of 100 bytes. If, then, some of these 100-byte messages are missing (e.g., if SCTP with Configurable Reliability is used), this is the expected application behavior. With TCP, no messages would be missing, but this is also correct for the application, and the possible retransmission delay is acceptable within the best effort service model. Still, the receiving application would separate the byte stream into 100-byte chunks.

Note that this usage of messages does not require all messages to be equal in size. Many application protocols use some form of Type-Length-Value (TLV) encoding, e.g. by defining a header including length fields; another alternative is the use of byte stuffing methods such as COBS [COBS]. If an application needs message numbers, e.g. to restore the correct sequence of messages, these must also be encoded by the application itself, as the sequence number related transport features of SCTP are no longer provided (in the interest of enabling a fall-back to TCP).

!!!NOTE: IMPLEMENTATION DETAILS BELOW WILL BE MOVED TO A SEPARATE DRAFT IN A FUTURE VERSION.!!!

For the implementation of a TAPS system, this has the following consequences:

- o Because the receiver-side transport leaves it up to the application to delineate messages, messages must always remain intact as they are handed over by the transport receiver. Data can be handed over at any time as they arrive, but the byte stream must never "skip ahead" to the beginning of the next message.
- o With SCTP, a "partial flag" informs a receiving application that a message is incomplete. Then, the next receive calls will only deliver remaining parts of the same message (i.e., no messages or partial messages will arrive on other streams until the message is complete) (see Section 8.1.20 in [RFC6458]). This can facilitate the implementation of the receiver buffer in the receiving application, but then such an application does not support message interleaving (which is required by stream schedulers). However, receiving a byte stream from multiple SCTP streams requires a per-stream receiver buffer anyway, so this potential benefit is lost and the "partial flag" (the transport feature "Information about partial message arrival") becomes unnecessary for a TAPS system. With it, the transport features "Configure size where messages are broken up for partial delivery" and "Notification to a receiver that a partial message delivery has been aborted" become unnecessary too.
- o From the above, a TAPS system should always support message interleaving because it enables the use of stream schedulers and comes at no additional implementation cost on the receiver side. Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to an SCTP receiver that does not support interleaving, it cannot assume that stream schedulers will always work as expected.

A.3.2. Stream Schedulers Without Streams

We have already stated that multi-streaming does not require application-specific knowledge. Potential benefits or disadvantages of, e.g., using two streams over an SCTP association versus using two separate SCTP associations or TCP connections are related to knowledge about the network and the particular transport protocol in use, not the application. However, the transport features "Choose a scheduler to operate between streams of an association" and "Configure priority or weight for a scheduler" operate on streams. Here, streams identify communication channels between which a scheduler operates, and they can be assigned a priority. Moreover, the transport features in the MAINTENANCE category all operate on associations in case of SCTP, i.e. they apply to all streams in that

association.

With only these semantics necessary to represent, the interface to a TAPS system becomes easier if we rename connections into "TAPS flows" (the TAPS equivalent of a connection which may be a transport connection or association, but could also become a stream of an existing SCTP association, for example) and allow assigning a "Group Number" to a TAPS flow. Then, all MAINTENANCE transport features can be said to operate on flow groups, not connections, and a scheduler also operates on the flows within a group.

!!!NOTE: IMPLEMENTATION DETAILS BELOW WILL BE MOVED TO A SEPARATE DRAFT IN A FUTURE VERSION.!!!

For the implementation of a TAPS system, this has the following consequences:

- o Streams may be identified in different ways across different protocols. The only multi-streaming protocol considered in this document, SCTP, uses a stream id. The transport association below still uses a Transport Address (which includes one port number) for each communicating endpoint. To implement a TAPS system without exposed streams, an application must be given an identifier for each TAPS flow (akin to a socket), and depending on whether streams are used or not, there will be a 1:1 mapping between this identifier and local ports or not.
- o In SCTP, a fixed number of streams exists from the beginning of an association; streams are not "established", there is no handshake or any other form of signaling to create them: they can just be used. They are also not "gracefully shut down" -- at best, an "SSN Reset Request Parameter" in a "RE-CONFIG" chunk [RFC6525] can be used to inform the peer that of a "Stream Reset", as a rough equivalent of an "Abort". This has an impact on the semantics connection establishment and teardown (see Section 3.1).
- o To support stream schedulers, a receiver-side TAPS system should always support message interleaving because it comes at no additional implementation cost (because of the receiver-side stream reception discussed in Appendix A.3.1). Note, however, that Stream schedulers operate on the sender side. Hence, because a TAPS sender-side application may talk to a native TCP-based receiver-side application, it cannot assume that stream schedulers will always work as expected.

A.3.3. Early Data Transmission

There are two transport features related to transferring a message early: "Hand over a message to transfer (possibly multiple times) before connection establishment", which relates to TCP Fast Open [RFC7413], and "Hand over a message to transfer during connection

establishment", which relates to SCTP's ability to transfer data together with the COOKIE-Echo chunk. Also without TCP Fast Open, TCP can transfer data during the handshake, together with the SYN packet -- however, the receiver of this data may not hand it over to the application until the handshake has completed. This functionality is commonly available in TCP and supported in several implementations, even though the TCP specification does not explain how to provide it to applications.

A TAPS system could differentiate between the cases of transmitting data "before" (possibly multiple times) or during the handshake. Alternatively, it could also assume that data that are handed over early will be transmitted as early as possible, and "before" the handshake would only be used for data that are explicitly marked as "idempotent" (i.e., it would be acceptable to transfer it multiple times).

The amount of data that can successfully be transmitted before or during the handshake depends on various factors: the transport protocol, the use of header options, the choice of IPv4 and IPv6 and the Path MTU. A TAPS system should therefore allow a sending application to query the maximum amount of data it can possibly transmit before (or, if exposed, during) connection establishment.

A.3.4. Sender Running Dry

The transport feature "Notification that the stack has no more user data to send" relates to SCTP's "SENDER DRY" notification. Such notifications can, in principle, be used to avoid having an unnecessarily large send buffer, yet ensure that the transport sender always has data available when it has an opportunity to transmit it. This has been found to be very beneficial for some applications [WWDC2015]. However, "SENDER DRY" truly means that the entire send buffer (including both unsent and unacknowledged data) has emptied -- i.e., when it notifies the sender, it is already too late, the transport protocol already missed an opportunity to send data. Some modern TCP implementations now include the unspecified "TCP_NOTSENT_LOWAT" socket option proposed in [WWDC2015], which limits the amount of unsent data that TCP can keep in the socket buffer; this allows to specify at which buffer filling level the socket becomes writable, rather than waiting for the buffer to run empty.

SCTP allows to configure the sender-side buffer too: the automatable Transport Feature "Configure send buffer size" provides this functionality, but only for the complete buffer, which includes both unsent and unacknowledged data. SCTP does not allow to control these two sizes separately. A TAPS system should allow for uniform access

to "TCP_NOTSENT_LOWAT" as well as the "SENDER DRY" notification.

A.3.5. Capacity Profile

The transport features:

- o Disable Nagle algorithm
- o Enable and configure a "Low Extra Delay Background Transfer"
- o Specify DSCP field

all relate to a QoS-like application need such as "low latency" or "scavenger". In the interest of flexibility of a TAPS system, they could therefore be offered in a uniform, more abstract way, where a TAPS system could e.g. decide by itself how to use combinations of LEDBAT-like congestion control and certain DSCP values, and an application would only specify a general "capacity profile" (a description of how it wants to use the available capacity). A need for "lowest possible latency at the expense of overhead" could then translate into automatically disabling the Nagle algorithm.

In some cases, the Nagle algorithm is best controlled directly by the application because it is not only related to a general profile but also to knowledge about the size of future messages. For fine-grain control over Nagle-like functionality, the "Request not to bundle messages" is available.

A.3.6. Security

Both TCP and SCTP offer authentication. TCP authenticates complete segments. SCTP allows to configure which of SCTP's chunk types must always be authenticated -- if this is exposed as such, it creates an undesirable dependency on the transport protocol. For compatibility with TCP, a TAPS system should only allow to configure complete transport layer packets, including headers, IP pseudo-header (if any) and payload.

Security will be discussed in a separate TAPS document (to be referenced here when it appears). The minimal set presented in the present document therefore excludes all security related transport features: "Configure authentication", "Change authentication parameters", "Obtain authentication information" and "Set Cookie life value" as well as "Specifying a key id to be used to authenticate a message".

A.3.7. Packet Size

UDP(-Lite) has a transport feature called "Specify DF field". This yields an error message in case of sending a message that exceeds the Path MTU, which is necessary for a UDP-based application to be able to implement Path MTU Discovery (a function that UDP-based

applications must do by themselves). The "Get max. transport-message size that may be sent using a non-fragmented IP packet from the configured interface" transport feature yields an upper limit for the Path MTU (minus headers) and can therefore help to implement Path MTU Discovery more efficiently.

This also relates to the fact that the choice of path is automatable: if a TAPS system can switch a path at any time, unknown to an application, yet the application intends to do Path MTU Discovery, this could yield a very inefficient behavior. Thus, a TAPS system should probably avoid automatically switching paths, and inform the application about any unavoidable path changes, when applications request to disallow fragmentation with the "Specify DF field" feature.

Appendix B. Revision information

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

-02: implementation suggestions added, discussion section added, terminology extended, DELETED category removed, various other fixes; list of Transport Features adjusted to -01 version of [TAPS2] except that MPTCP is not included.

-03: updated to be consistent with -02 version of [TAPS2].

-04: updated to be consistent with -03 version of [TAPS2]. Reorganized document, rewrote intro and conclusion, and made a first stab at creating a real "minimal set".

-05: updated to be consistent with -05 version of [TAPS2] (minor changes). Fixed a mistake regarding Cookie Life value. Exclusion of security related transport features (to be covered in a separate document). Reorganized the document (now begins with the minset, derivation is in the appendix). First stab at an abstract API for the minset.

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
Internet-Draft
Intended status: Experimental
Expires: December 3, 2017

K-J. Grinnemo
A. Brunstrom
P. Hurtig
Karlstad University
N. Khademi
University of Oslo
Z. Bozakov
Dell EMC Research Europe
June 2017

Happy Eyeballs for Transport Selection
draft-grinnemo-taps-he-03

Abstract

Ideally, network applications should be able to select an appropriate transport solution from among available transport solutions. However, at present, there is no agreed-upon way to do this. In fact, there is not even an agreed-upon way for a source end host to determine if there is support for a particular transport along a network path. This draft addresses these issues, by proposing a Happy Eyeballs framework. The proposed Happy Eyeballs framework enables the selection of a transport solution that according to application requirements, pre-set policies, and estimated network conditions is the most appropriate one. Additionally, the proposed framework makes it possible for an application to find out whether a particular transport is supported along a network connection towards a specific destination or not.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 3, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Definitions	2
2. Introduction	2
3. Problem Statement	3
4. The Happy Eyeballs Framework	4
5. Design and Implementation Considerations	5
5.1. Candidate List Generation	5
5.2. Caching	7
5.3. Concurrent Connection Attempts	7
6. Example Happy Eyeballs Scenario	7
7. IANA Considerations	8
8. Security Considerations	8
9. Acknowledgements	8
10. References	8
10.1. Normative References	9
10.2. Informative References	9
Authors' Addresses	9

1. Definitions

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

2. Introduction

Information services on the Internet come in varying forms, such as web browsing, email, and on-demand multimedia. The main motivation behind the design of next-generation computer and communications networks is to provide a universal and easy access to these various types of information services on a single multi-service Internet. This means that all forms of communications, e.g., video, voice, data

and control signaling, along with all types of services -- from plain text web pages to multimedia applications -- are bonded in a single-service platform through Internet technology. To enable the next-generation networks, the TAPS Working Group suggests a decoupling between the transport service provided to an application, and the transport stack providing this transport service: An application requests an appropriate transport service on the basis of its transport requirements, and the available transport stack that best meets these requirements is selected. In case the most preferred transport stack is not supported along the network path to the destination, or is not supported by the end host, a less-preferred transport stack is selected instead. As a way to realize the selection of transport stacks, this document suggests a generalization of the Happy Eyeballs (HE) mechanism proposed in Wing et al. [RFC6555] which addresses the selection of complete transport solutions, and which lends itself to arbitrary transport selection criterias. The proposed HE mechanism targets connection-oriented transport solutions, and connectionless transport solutions provided they offer some reasonable way to determine their successful use between endpoints.

The HE mechanism was introduced as a means to promote the use of dual network stacks. Dual-stack client applications should be encouraged to try setting up connections over IPv6 first, and fall back to using IPv4 if IPv6 connection attempts fail. However, serializing tests for IPv6 and IPv4 connectivity can result in large connection latencies. HE for IPv6 minimizes the cost in delay by parallelizing attempts over IPv6 and IPv4. HE has also been proposed as an efficient way to find out the optimal combination of IPv4/IPv6 and TCP/SCTP to use to connect to a server [I-D.wing-tsvwg-happy-eyeballs-sctp]. The HE framework suggested in this document could be seen as a natural continuation of this proposal.

3. Problem Statement

Currently, there is no agreed-upon way for a source end host to select an appropriate transport service for a given application. In fact, there is no common way for a source end-host to find out if a transport stack is supported along a network path between itself and a destination end host. As a consequence, it has become increasingly difficult to introduce new transport stacks, and several applications, including many web applications, run over TCP although there are other transport protocols that better meet the requirements of these applications.

4. The Happy Eyeballs Framework

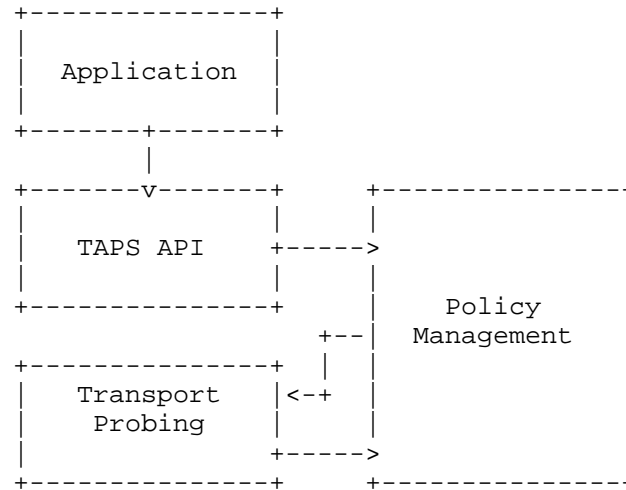


Figure 1: The Happy Eyeballs Framework.

The generalized HE mechanism proposed in this draft is carried out within the framework depicted in Figure 1. It comprises the following steps:

1. The Policy Management component takes as input application requirements from the TAPS API, stored information about previous connection attempts (e.g., whether previous connection attempts succeeded or not), and network conditions and configurations. On the basis of this input and the policies configured in the system, the Policy Management component creates a list of candidate transport solutions, *L*, sorted in decreasing priority order. To be compliant with RFC 6555 [RFC6555], the Policy Management component SHOULD, in those cases there are no policies telling otherwise, following the host's address preference, something which usually means giving preference to IPv6 over IPv4.
2. It is the responsibility of the Transport Probing component to select the most appropriate transport solution. This is done by initiating connection attempts for each transport solution on *L*. To minimize the number of connection attempts that are initiated, the Transport Probing component SHOULD cache the outcome of connection attempts in a repository kept by the Policy Management component. The Policy Management component SHOULD in turn only include those transport solutions on *L* that have not been previously attempted, have valid successful connection-attempt

cache entries, or have previously been attempted but whose cached connection-attempt entries have expired. Cached connection-attempt results SHOULD be valid for a configurable amount of time after which they SHOULD expire and have to be repeated. The transport solutions on L are initiated in priority order. The difference in priority between two consecutive candidates, C1 and C2, is translated according to some criteria to a delay, D. D then governs the delay between the initiation of the connection attempts C1 and C2.

3. After the initiation of the connection attempts, the Transport Probing component waits for the first or winning connection to be established, which becomes the selected transport solution. For the Transport Probing component to be able to efficiently use the connection-attempt cache, already-initiated, non-winning connection attempts SHOULD be given a fair chance to complete. In that way, the connection-attempt cache will be provided with a fairly accurate knowledge of which transport solutions work and does not work against frequently visited transport endpoints. Moreover, it MAY be beneficial to let those transport solutions which have a higher priority than the winning transport solution, live a predetermined amount of time after their establishment, since this enables the reuse of already established connections in later application requests.

5. Design and Implementation Considerations

This section discusses implementation issues that should be considered when a HE mechanism is designed and implemented on the basis of the HE framework proposed in this document.

5.1. Candidate List Generation

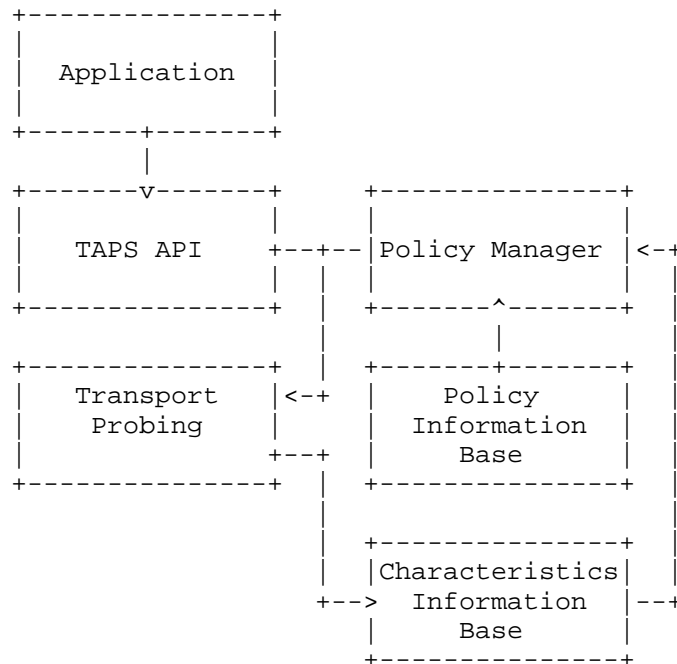


Figure 2: Principle Design of the NEAT Happy Eyeballs Framework.

There are several ways in which the list of candidate transport solutions, L , could be created by the Policy Management component. For example, L could be a list of all available transport solutions in an order that, except for following the host's address preference, is arbitrary; another, more sophisticated, way of creating the list of candidate transport solutions is the one employed by the NEAT System.

The NEAT System is developed as part of the EU Horizon 2020 project, "A New, Evolutive API and Transport-Layer Architecture for the Internet" (NEAT) [NEAT-Webb], and aims to provide a flexible and evolvable transport system that aligns with the charter of the TAPS Working Group. In the NEAT System [NEAT-Git], the HE framework is realized as shown in Figure 2. As follows, the Policy Management component comprises three components in the NEAT HE framework: a Policy Manager (PM), a Policy Information Base (PIB), and a Characteristics Information Base (CIB). PIB is a repository that stores a collection of policies that map application requests to transport solutions, i.e., map application requests to appropriately configured transport protocols, and CIB is a repository that stores information about previous connection attempts, available network

interfaces, supported transport protocols etc. The PM takes as input application requirements from the TAPS API, and information from PIB and CIB. On the basis of this input, the PM creates L.

5.2. Caching

As pointed out in RFC 6555 [RFC6555], a HE algorithm should not waste networking resources by routinely making simultaneous connection attempts. To this end, the HE algorithm should cache the outcome of previous connection attempts to the same peer. The cache lifetime is considered system dependent and should be set on a case-by-case basis. The impact and efficiency of the HE algorithm have been evaluated in [Papastergioul6]. The paper suggests that caching significantly reduces the CPU load imposed by a HE mechanism. It also indicates that the internal-memory footprint of a HE mechanism is essentially the same as for single-flow establishments.

5.3. Concurrent Connection Attempts

As mentioned in Section 4, it is the responsibility of the Transport Probing component to choose the most appropriate transport solution on the list of candidate transport solutions, L. Often this implies that several transport solutions need to be tried out, something which should not be carried out sequentially, but concurrently or partly overlapping depending on the transport-solution priorities. The way this is done is implementation dependent and varies between platforms. The NEAT library [NEAT-Git], which implements the HE framework herein, is built around the libuv asynchronous I/O library [LIBUV] and uses an event-based concurrency model to realize the concurrent initialization of connection attempts. The rationale behind using an event-based concurrency model is at least twofold: The first is that correctly managing concurrency in multi-threaded applications can be challenging with, for example, missing locks or deadlocks. The second is that multi-threading typically offers little or no control over what is scheduled at a given moment in time. Given the complexity of building a general-purpose scheduler that works well in all cases, sometimes the OS will schedule work in a manner that is less than optimal. Those in favor of threads argue that threads are a natural extension of sequential programming in that it maps work to be executed with individual threads. Threads are also a well-known and understood parts of OSes, and are mandatory for exploiting true CPU concurrency.

6. Example Happy Eyeballs Scenario

Consider a scenario in which an IPv6-enabled client using the NEAT System wishes to setup a connection to a server. Assume both the client and server support SCTP and TCP. The Policy Management is

queried about feasible transport solutions to connect to the server. In the NEAT System, this results in PM retrieving information about network connections against this server from the CIB, e.g., supported transport protocols and the outcome of previous connection attempts. In our scenario, the PM learns from the CIB that the server supports SCTP and TCP, and, for the sake of this example, let us assume that the PM is also informed that previous connection attempts against this server, using both SCTP and TCP, were successful. Next, the PM retrieves applicable policies from the PIB, and combines these policies with the previously retrieved CIB information. We assume in this example that the SCTP transport solution has a higher priority than the TCP solution. As a next step, the PM puts together the feasible candidate transport solutions in a list with SCTP over IPv6 placed at the head of the list followed by TCP over IPv6, and supplies this list to the Transport Probing component. The Transport Probing component traverses the candidate list, and initiates a connection attempt with SCTP against the server followed after a short while (governed by the difference in priorities between the SCTP and TCP transport solutions) by a connection attempt with TCP against the server. In our example, assume both connection attempts are successful, however, the SCTP connection attempt completes before the TCP attempt. The Transport Probing component caches in the CIB the SCTP connection attempt as successful, and returns the SCTP connection as the winning connection. When the TCP connection is established some time later, the Transport Probing component caches that connection attempt as successful as well.

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. 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).

10. References

10.1. Normative References

- [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>>.
- [RFC6555] Wing, D. and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts", RFC 6555, DOI 10.17487/RFC6555, April 2012, <<http://www.rfc-editor.org/info/rfc6555>>.

10.2. Informative References

- [I-D.wing-tsvwg-happy-eyeballs-sctp]
Wing, D. and P. Natarajan, "Happy Eyeballs: Trending Towards Success with SCTP", draft-wing-tsvwg-happy-eyeballs-sctp-02 (work in progress), October 2010.
- [LIBUV] libuv -- Asynchronous I/O Made Simple, "<http://libuv.org>", March 2017.
- [NEAT-Git]
A New, Evolutive API and Transport-Layer Architecture for the Internet (NEAT), "<https://github.com/NEAT-project/neat>", March 2017.
- [NEAT-Webb]
NEAT -- A New, Evolutive API and Transport-Layer Architecture for the Internet, "<https://www.neat-project.org>", March 2017.
- [Papastergiou16]
Papastergiou, G., Grinnemo, K-J., Brunstrom, A., Ros, D., Tuexen, M., Khademi, N., and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection", July 2016.

Authors' Addresses

Karl-Johan Grinnemo
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 24 40
Email: karl-johan.grinnemo@kau.se

Anna Brunstrom
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 17 95
Email: anna.brunstrom@kau.se

Per Hurtig
Karlstad University
Universitetsgatan 2
Karlstad 651 88
Sweden

Phone: +46 54 700 23 35
Email: per.hurtig@kau.se

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

Email: naeemk@ifi.uio.no

Zdravko Bozakov
Dell EMC Research Europe
Ovens, Co.
Cork
Ireland

Phone: +353 21 4945733
Email: Zdravko.Bozakov@dell.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2018

M. Kuehlewind
ETH Zurich
T. Pauly
C. Wood
Apple Inc.
July 03, 2017

Separating Crypto Negotiation and Communication
draft-kuehlewind-taps-crypto-sep-00

Abstract

Due to the latency involved in connection setup and security handshakes, there is an increasing deployment of cryptographic session resumption mechanisms. While cryptographic context and endpoint capabilities need to be known before encrypted application data can be sent, there is otherwise no technical constraint that the crypto handshake must be performed on the same transport connection. This document recommends a logical separation between the mechanism(s) used to negotiate capabilities and set up encryption context (handshake protocol), the application of encryption and authentication state to data (record protocol), and the associated transport connection(s).

Status of This Memo

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

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

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

This Internet-Draft will expire on January 4, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Terminology	3
3. Protocol Interfaces	4
3.1. Handshake-Transport Interface	5
3.2. Handshake-Record Interface	6
3.3. Transport-Record Interface	6
4. Existing Mappings	6
5. Benefits of Separation	8
5.1. Reducing Connection Latency	9
5.2. Protocol Flexibility	9
5.3. Protocol Capability Negotiation	10
6. IANA Considerations	10
7. Security Considerations	10
8. Acknowledgments	10
9. Informative References	10
Authors' Addresses	11

1. Introduction

Secure transport protocols are generally composed of three pieces:

1. A transport protocol to control the transfer of data.
2. A record protocol to frame, encrypt and/or authenticate data
3. A handshake protocol to negotiate cryptographic secrets.

For ease of deployment and standardization, among other reasons, these constituents are often tightly coupled. For example, in TLS [RFC5246], the handshake protocol depends on the record protocol, and vice versa. However, more recent transport protocols such as QUIC [I-D.ietf-quic-tls] keep these pieces separate. QUIC uses TLS to negotiate secrets, and _exports_ those secrets to encrypt packets directly.

Separating these pieces is important, as new secure transport protocols increasingly rely on session resumption mechanisms where cryptographic context can be resumed to transmit application data with the first packet without delay for connection setup and negotiation. In the case where there is no cryptographic context available when an application expresses the need to transmit data to a certain endpoint, it must first run the handshake protocol on a transport connection before being able to transmit application data. If the handshake protocol can be separated from the other components, then it can use another transport connection to establish secrets without blocking the application's main transport connection. This also opens up the possibility to run the handshake protocol well in advance of the need to send application data, to avoid unnecessary delays. For example, a client system could maintain a database of endpoints it is likely to communicate with, and establish keying material with a handshake protocol at periodic intervals to ensure fresh keys for new transport connections.

[I-D.moskowitz-sse] proposes a similar approach. However while [I-D.moskowitz-sse] proposes a new protocol to negotiate and maintain long-term cryptographic sessions, this document relies on the use of existing protocols and only discusses requirements for the evolution of these protocols and exchange of information within one endpoint locally.

2. Terminology

- o Transport Protocol: A protocol that can transport messages between two endpoints. This may represent the service offered to applications to allow them to send and receive data before encryption; and also represent the protocol that can transmit handshake data and encrypted records.
- o Handshake Protocol: A protocol that can validate and authenticate endpoints, encrypt and authenticate its negotiation, and ultimately generate keying material.
- o Record Protocol: A protocol that can use keying material to transform messages. A record will generally add a frame around application data, and authenticate and/or encrypt the data.
- o Keying Material: One or more pre-shared keys that can be used to encrypt and authenticate data, generated by a handshake protocol and used by a record protocol.

3. Protocol Interfaces

In traditional models in which the protocols are not separated out into the three elements of handshake, record, and transport protocols, there are two basic approaches to the interactions:

1. The transport protocol provides data to the security protocol and gets back an encrypted version of the data to be sent (handshake and record protocols are combined)
2. The security protocol provides keying material to the transport protocol, and the transport protocol is responsible for encrypting data (transport and record protocols are combined)

By teasing apart all three portions as separate protocols, there end up being six interface points:

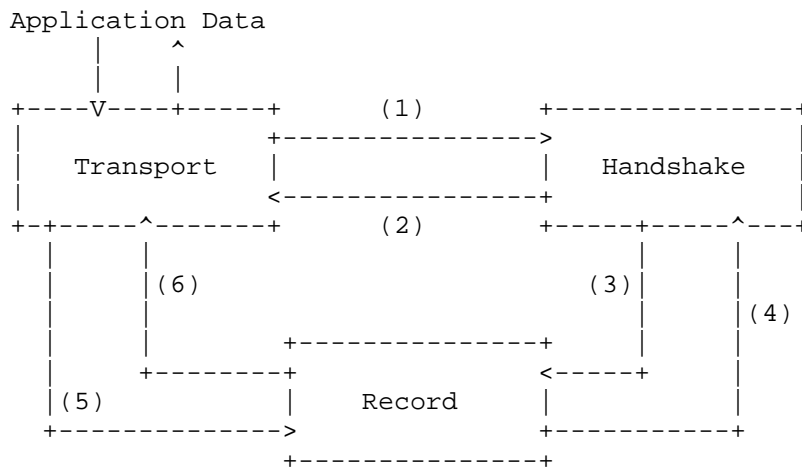


Figure 1: Secure Transport Protocol Components and Interactions

1. A transport protocol depends upon a handshake protocol to establish keying material to protect application data being sent through the transport. The main interface it relies upon is starting the handshake, or ensuring that the material is ready.
2. A handshake protocol depends upon a transport protocol in order to send and receive negotiation messages with the remote peer.
3. A handshake protocol sends its keying material and cryptographic context to the record protocol to use

4. A record protocol may signal state expiration events to a handshake protocol
5. A transport protocol uses a record protocol to send and receive application data
6. A record protocol uses a transport protocol to send and receive encrypted data

3.1. Handshake-Transport Interface

Note that for the purposes of this interface description, it is assumed that the application is primarily interacting with the transport protocol, and thus the handshake protocol interacts with the application primarily through the abstraction of the transport protocol.

- o Start negotiation: The interface MUST provide an indication to start the protocol handshake for key negotiation, and have a way to be notified when the handshake is complete.
- o Identity constraints: The interface MUST allow the application to constrain the identities that it will accept a connection to, such as the hostname it expects to be provided in certificate SAN.
- o Local identities: The interface MUST allow the local identity to be set via a raw private key or interface to one to perform cryptographic operations such as signing and decryption.
- o State changes: The interface SHOULD provide a way for the transport to be notified of important state changes during the protocol execution and session lifetime, e.g., when the handshake begins, ends, or when a key update occurs.
- o Validation: The interface MUST provide a way for the application to participate in the endpoint authentication and validation, which can either be specified as parameters to define how the peer's authentication can be validated, or when the protocol provides the authentication information for the application to inspect directly.
- o Caching domain and lifetime: The application SHOULD be able to specify the instances of the protocol that can share cached keys, as well as the lifetime of cached resources.
- o The protocol SHOULD allow applications to negotiate application protocols and related information.

- o The protocol SHOULD allow applications to specify negotiable cryptographic algorithm suites.
- o The protocol SHOULD expose the peer's identity information.

3.2. Handshake-Record Interface

- o Key export: The interface MUST provide a way to export keying material from a handshake protocol to a record protocol with well-defined cryptographic properties, e.g., "forward-secure" or "perfectly forward secure"
- o Key lifetime and rotation: The interface MUST provide a way for the handshake protocol to define key lifetime bounds in terms of `_time_` or `_bytes encrypted_` and, additionally, provide a way to forcefully update cryptographic session keys at will. The record protocol MUST be able to signal back to the handshake protocol that a lifetime has been reached and that rotation is required. These values SHOULD be configurable by the application.

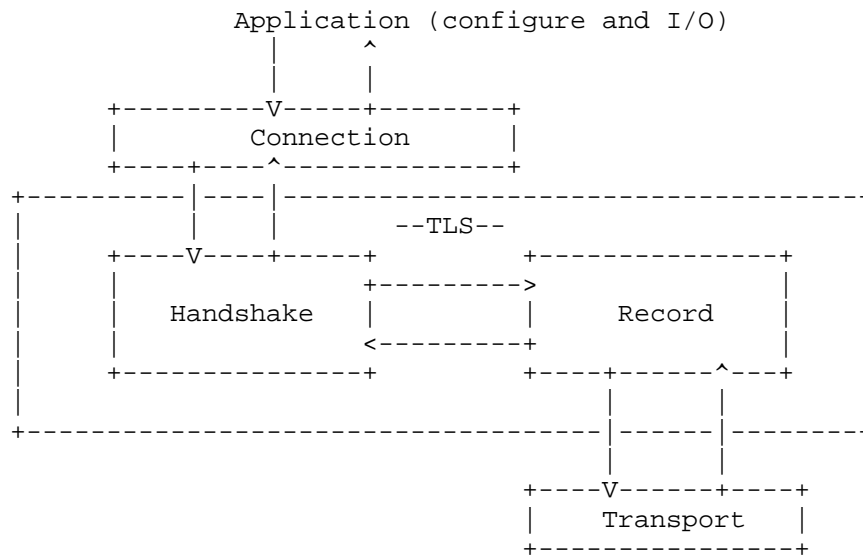
3.3. Transport-Record Interface

- o Transform data: The interface MUST provide a way to send raw application data from the transport protocol to a record protocol to transform it based on the keying material. This data is then sent out by the transport protocol. The same applies for inbound data, in which inbound transport data is transformed by the record protocol into raw application data.
- o Reliability: The transport MUST specify if messages are transmitted reliable and in order.
- o Maximum message size (optional): The transport may specify a maximum message size for the encrypted data if e.g. a datagram transport is used

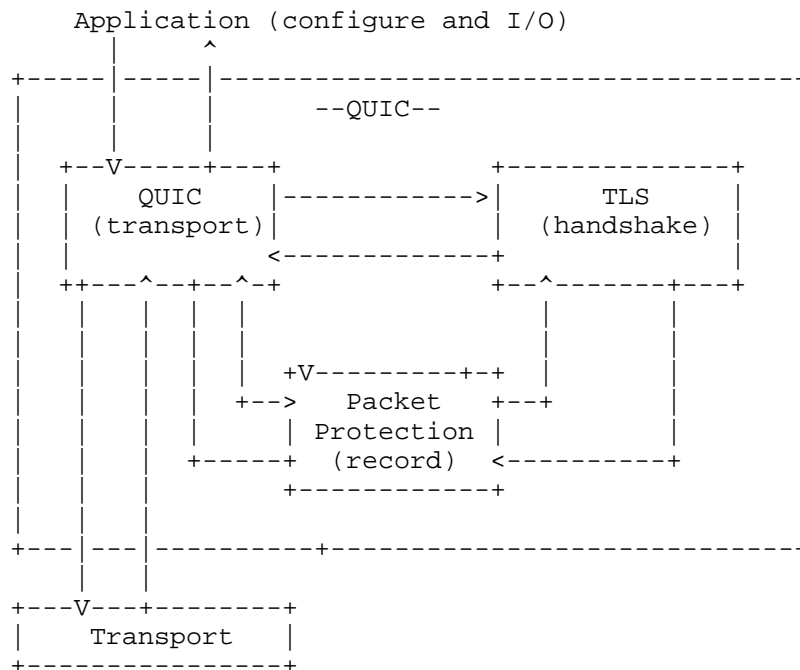
4. Existing Mappings

In this section we document existing mappings between common transport security protocols and the three components described in Section I.

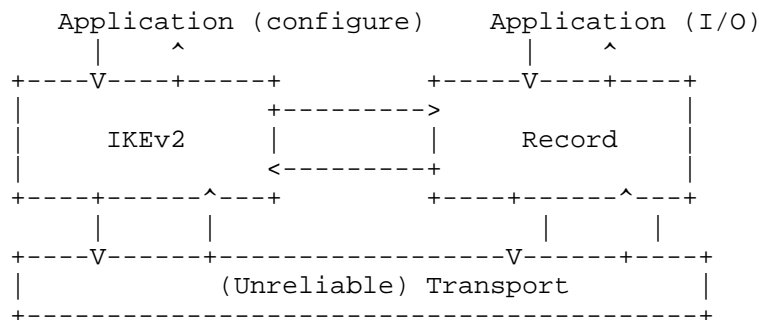
- o TLS/DTLS: TLS [RFC5246] and DTLS [RFC6347] is a combination of a handshake and record protocol, with a dependency on some underlying transport.



- o QUIC + TLS: The emerging QUIC standard is decomposed into the three pieces outlined in Section I [I-D.ietf-quic-tls]. TLS is used as the handshake protocol running on a dedicated QUIC stream, a QUIC-specific record protocol encrypts and encapsulates stream frames, and the main QUIC component handles the transport of these frames.



- o **IKEv2 + ESP:** IKEv2 [RFC7296] is a handshake protocol commonly used to establish keys for use in IPsec (often VPN) deployments. It is already a distinct protocol from its commonly paired record protocol, which is ESP [RFC4303]. ESP encrypts and authenticates IP datagrams, and sends them as datagrams over a transport mechanism such, e.g., IP or UDP.



5. Benefits of Separation

5.1. Reducing Connection Latency

One of the clearest benefits of separating the handshake protocol from the record protocol is that the handshake can be performed out-of-band from the application's data transfer. This should essentially reduce the number of RTTs required before being able to send data by the full length of the handshake (which is commonly 1 or 2 RTTs in the best cases for TLS 1.2 and IKEv2, potentially more if cookie challenges or extended authentication are required).

To avoid long-lived transport connections that wouldn't be actively used, and thus would be vulnerable to timeouts on NATs or firewalls, an obvious approach to separating the handshake and record protocols is to use different transport connections for the early handshake and the data transfer. However, this approach of using separate connections will not always save RTTs if the handshake and data transfer are back-to-back. Each connection may require its own transport protocol handshake, and if the data transfer must wait for two transport protocols to establish and the cryptographic handshake to be finished before sending, then it may experience higher latency. Implementations SHOULD avoid this by either allowing the handshake and record protocols to share a single transport connection or open two connections in parallel when the handshake protocol has not pre-fetched keys. Latency benefits, however, can even be achieved when ensuring that this scenario does not occur by always having the handshake protocol refresh the keys whenever old ones are near expiry.

5.2. Protocol Flexibility

Separation of the handshake, record, and transport protocols also allows for more flexible composition of protocols with one another. If a deployment uses a handshake protocol like TLS, which requires a stream-based transport protocol like TCP, separation of protocols will allow it to use the resulting keys for record protocols that run on datagram transport protocols like UDP.

This flexibility may be useful for implementations that are optimizing for packet size by choosing minimal/lightweight record protocols, while being able to use commonly supported handshake protocols like TLS. One example here is the approach of a VPN tunnel that uses ESP or Diet-ESP [I-D.mglt-ipsecme-diet-esp] to encrypt datagrams, but uses TLS for establishing keys.

5.3. Protocol Capability Negotiation

Enabling the use of a different transport protocol for the actual data transmission than for the cryptographic handshakes opens also the possibility to negotiate protocol capabilities for the data transmission. For TLS, usually TCP is the appropriate transport protocol to use, as it is also widely supported by endpoints. Allowing an endpoint to indicate the support of other, new transport protocols within the TCP connection that is used for the handshake, provides a dynamic transition path to enable easy deployment of new protocols.

6. IANA Considerations

This document has on request to IANA.

7. Security Considerations

(editor's note: this section will be added later. However, this document discusses the use of cryptographic context for transport connections and as such it has security relevant consideration within the whole document.)

8. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement.

9. Informative References

[I-D.ietf-quic-tls]

Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-04 (work in progress), June 2017.

[I-D.mglt-ipsecme-diet-esp]

Migault, D., Guggemos, T., and C. Bormann, "ESP Header Compression and Diet-ESP", draft-mglt-ipsecme-diet-esp-04 (work in progress), June 2017.

[I-D.moskowitz-sse]

Moskowitz, R., Faynberg, I., Lu, H., Hares, S., and P. Giacomin, "Session Security Envelope", draft-moskowitz-sse-05 (work in progress), June 2017.

- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<http://www.rfc-editor.org/info/rfc4303>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<http://www.rfc-editor.org/info/rfc6347>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<http://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Christopher A. Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 1, 2019

M. Kuehlewind
ETH Zurich
T. Pauly
C. Wood
Apple Inc.
June 30, 2018

Separating Crypto Negotiation and Communication
draft-kuehlewind-taps-crypto-sep-03

Abstract

Secure transport protocols often consist of three logically distinct components: transport, control (handshake), and record protection. Typically, such a protocol contains a single module that is responsible for all three functions. However, in many cases, this coupling is unnecessary. For example, while cryptographic context and endpoint capabilities need to be known before encrypted application data can be sent on a specific transport connection, there is otherwise no technical constraint that a cryptographic handshake must be performed on said connection. This document recommends a logical separation between transport, control, and record components of secure transport protocols. We compare existing protocols such as Transport Layer Security, QUIC, and IKEv2+ESP in the context of this logical separation.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 1, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	2
2. Terminology	3
3. Protocol Interfaces	4
3.1. Control-Transport Interface	5
3.1.1. Passive Configuration Interface	5
3.1.2. Active Control and Introspection Interface	6
3.2. Control-Record Interface	6
3.3. Transport-Record Interface	6
4. Existing Mappings	7
5. Benefits of Separation	10
5.1. Reducing Connection Latency	10
5.2. Protocol Flexibility	11
5.3. Protocol Capability and Upgrade Negotiation	11
6. Transport Service Architecture Integration	11
7. IANA Considerations	12
8. Security Considerations	12
9. Acknowledgments	12
10. Informative References	12
Authors' Addresses	13

1. Introduction

Secure transport protocols are generally composed of three pieces:

1. A transport protocol to handle the transfer of data.
2. A record protocol to frame, encrypt and/or authenticate data
3. A control protocol to perform cryptographic handshakes, negotiate shared secrets, and maintain state during the lifetime of cryptographic session including session resumption and key

refreshment. (In the context of TLS, the control protocol is called the handshake protocol.)

For ease of deployment and standardization, among other reasons, these constituents are often tightly coupled. For example, in TLS [RFC5246], the control protocol depends on the record protocol, and vice versa. However, more recent transport protocols such as QUIC [I-D.ietf-quic-tls] keep these pieces separate. For example, QUIC uses TLS to negotiate secrets, and exports those secrets to encrypt packets independent of TLS.

Separating these pieces is important, as new secure transport protocols increasingly rely on session resumption mechanisms where cryptographic context can be resumed to transmit application data with the first packet without delay for connection setup and negotiation. In the case where there is no cryptographic context available when an application expresses the need to transmit data to a certain endpoint, it must first run the control protocol on a transport connection before being able to transmit application data. If the control protocol can be separated from the other components, then it can use another transport connection to establish secrets without blocking the application's main transport connection. This also opens up the possibility to run the control protocol well in advance of the need to send application data, to avoid unnecessary delays. For example, a client system could maintain a database of endpoints it is likely to communicate with, and establish keying material with a control protocol at periodic intervals to ensure fresh keys for new transport connections.

[I-D.moskowitz-sse] proposes a similar approach. However while [I-D.moskowitz-sse] proposes a new protocol to negotiate and maintain long-term cryptographic sessions, this document relies on the use of existing protocols and only discusses requirements for the evolution of these protocols and exchange of information within one endpoint locally.

2. Terminology

- o Transport Protocol: A protocol that can transport messages between two endpoints. This may represent the service offered to applications to allow them to send and receive data before encryption; and also represent the protocol that can transmit control data and encrypted records.
- o Control Protocol: A protocol that performs a cryptographic handshake and, in addition, can validate and authenticate endpoints, encrypt and authenticate its negotiation, and ultimately generate keying material.

- o Record Protocol: A protocol that can use keying material to transform messages. A record will generally add a frame around application data, and authenticate and/or encrypt the data.
- o Keying Material: A shared secret from which pre-shared keys can be derived and subsequently used to encrypt and authenticate data, generated by a control protocol and used by a record protocol.

3. Protocol Interfaces

In traditional models in which the protocols are not separated out into the three elements of control, record, and transport protocols, there are two basic approaches to the interactions:

1. The transport protocol provides data to the security protocol and gets back an encrypted version of the data to be sent (control and record protocols are combined).
2. The security protocol provides keying material to the transport protocol, and the transport protocol is responsible for encrypting data (transport and record protocols are combined).

By teasing apart all three portions as separate protocols, there end up being six interface points:

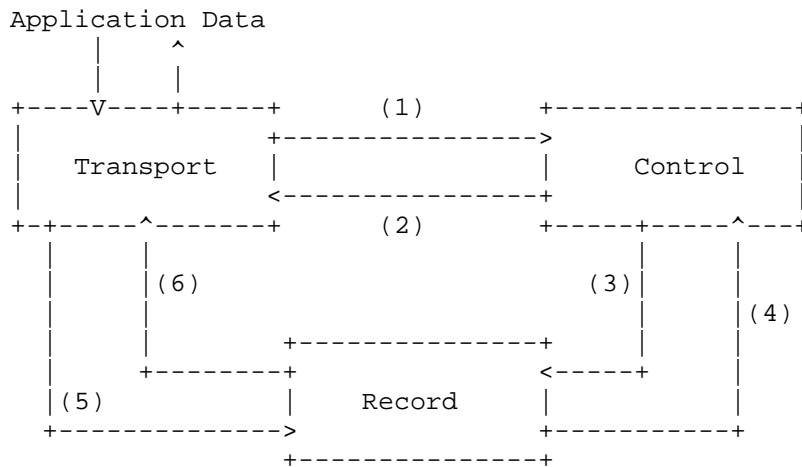


Figure 1: Secure Transport Protocol Components and Interactions

1. A transport protocol depends upon a control protocol to establish keying material to protect application data being sent through the transport. The main interface it relies upon is starting the

control channel, or handshake, or ensuring that the material is ready.

2. A control protocol depends upon a transport protocol in order to send and receive negotiation messages with the remote peer.
3. A control protocol sends its keying material and cryptographic context to the record protocol to use.
4. A record protocol may signal state expiration events to a control protocol.
5. A transport protocol uses a record protocol to send and receive application data.
6. A record protocol uses a transport protocol to send and receive encrypted data.

3.1. Control-Transport Interface

Note that for the purposes of this interface description, it is assumed that the application is primarily interacting with the transport protocol, and thus the control protocol interacts with the application primarily through the abstraction of the transport protocol. Since security protocol interfaces often require pre-connection and active behavior on behalf of clients, we further categorize the following interfaces based on whether they are meant for passive configuration or active control.

3.1.1. Passive Configuration Interface

- o Start negotiation: The interface MUST provide an indication to start the protocol handshake for key negotiation, and have a way to be notified when the handshake is complete.
- o Identity constraints: The interface MUST allow the application to constrain the identities that it will accept a connection to, such as the hostname it expects to be provided in certificate SAN.
- o Local identities: The interface MUST allow the local identity to be set via a raw private key or interface to one to perform cryptographic operations such as signing and decryption.
- o Caching domain and lifetime: The application SHOULD be able to specify the instances of the protocol that can share cached keys, as well as the lifetime of cached resources.

- o Pre-shared keying material: The application SHOULD be able to specify pre-share keying material to use to bootstrap connections. The control protocol can pass this directly to the record protocol for use.
- o The protocol SHOULD allow applications to negotiate application protocols and related information.
- o The protocol SHOULD allow applications to specify negotiable cryptographic algorithm suites.

3.1.2. Active Control and Introspection Interface

- o State changes: The interface SHOULD provide a way for the transport to be notified of important state changes during the protocol execution and session lifetime, e.g., when the handshake begins, ends, or when a key update occurs.
- o Validation: The interface MUST provide a way for the application to participate in the endpoint authentication and validation, which can either be specified as parameters to define how the peer's authentication can be validated, or when the protocol provides the authentication information for the application to inspect directly.
- o The protocol SHOULD expose the peer's identity information during and after connection establishment.

3.2. Control-Record Interface

- o Key export: The interface MUST provide a way to export keying material from a control protocol to a record protocol with well-defined cryptographic properties, e.g., "forward-secure."
- o Key lifetime and rotation: The interface MUST provide a way for the control protocol to define key lifetime bounds in terms of `_time_` or `_bytes encrypted_` and, additionally, provide a way to forcefully update cryptographic session keys at will. The record protocol MUST be able to signal back to the control protocol that a lifetime has been reached and that rotation is required. These values SHOULD be configurable by the application.

3.3. Transport-Record Interface

- o Transform data: The interface MUST provide a way to send raw application data from the transport protocol to a record protocol to transform it based on the keying material. This data is then sent out by the transport protocol. The same applies for inbound

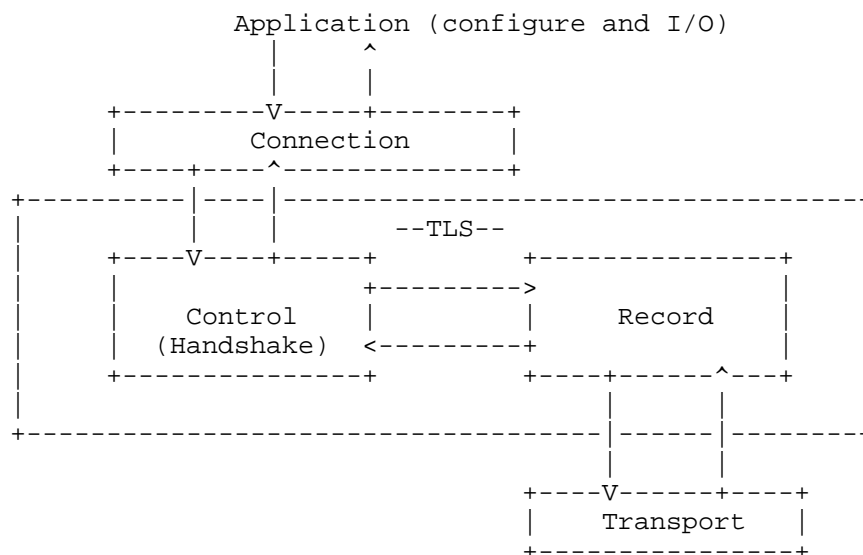
data, in which inbound transport data is transformed by the record protocol into raw application data.

- o Reliability: The transport MUST specify if messages are transmitted reliable and in order.
- o Maximum message size (optional): The transport may specify a maximum message size for the encrypted data if e.g. a datagram transport is used

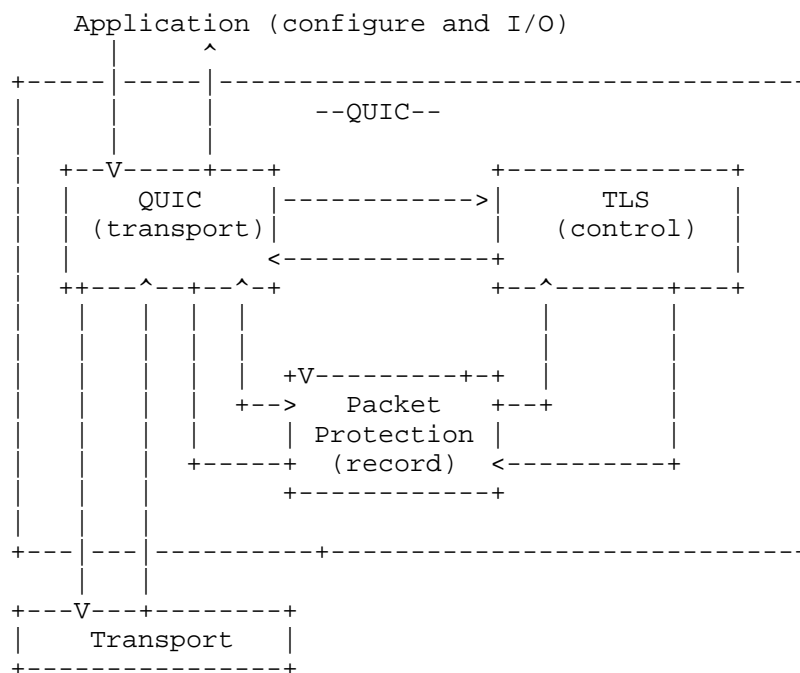
4. Existing Mappings

In this section we document existing mappings between common transport security protocols and the three components described in Section I.

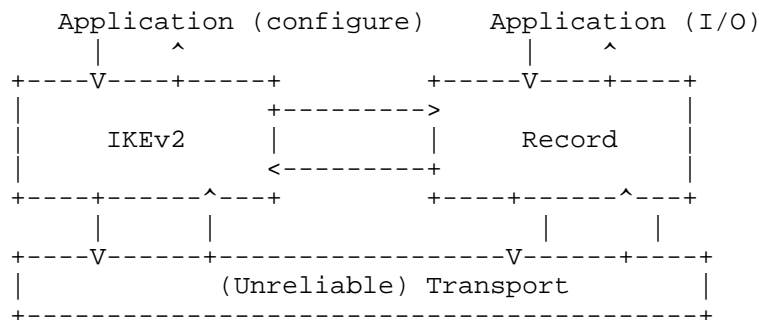
- o TLS/DTLS: TLS [RFC5246] and DTLS [RFC6347] is a combination of a control (handshake) and record protocol, with a dependency on some underlying transport.



- o QUIC + TLS: The emerging QUIC standard is decomposed into the three pieces outlined in Section I [I-D.ietf-quic-tls]. TLS is used as the control protocol running on a dedicated QUIC stream, a QUIC-specific record protocol encrypts and encapsulates stream frames, and the main QUIC component handles the transport of these frames.

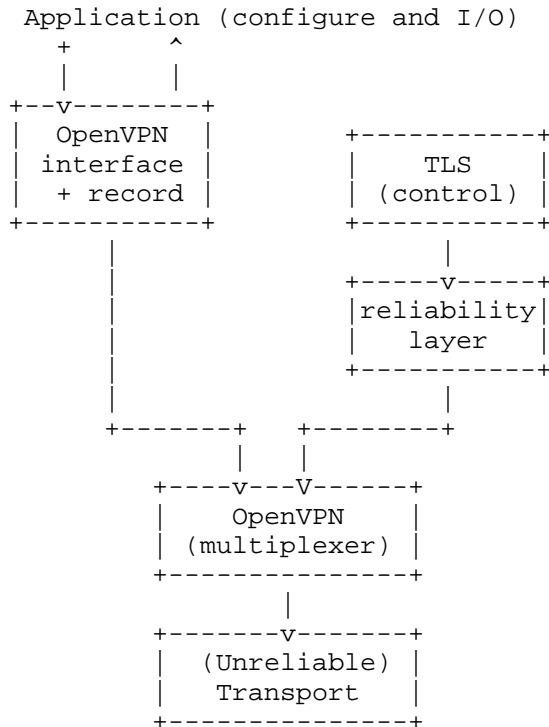


- o IKEv2 + ESP: IKEv2 [RFC7296] is a control protocol commonly used to establish keys for use in IPsec (often VPN) deployments. It is already a distinct protocol from its commonly paired record protocol, which is ESP [RFC4303]. ESP encrypts and authenticates IP datagrams, and sends them as datagrams over a transport mechanism such, e.g., IP or UDP.

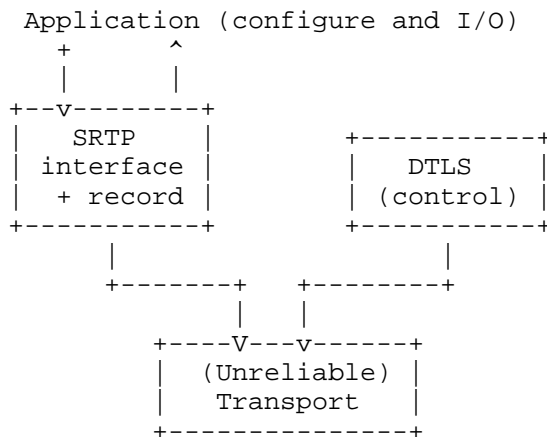


- o OpenVPN [OpenVPN]: OpenVPN consists of two separate stacks - one for TLS, which is used for key exchange and derivation, and the other as an interface to tunnel IP packets over UDP. A common multiplexing layer is used to send TLS and OpenVPN framed packets over an unreliable transport layer. OpenVPN adds a reliability

layer to TLS to ensure packets are sent and processed in order. Running over TCP naturally provides this reliability. After the TLS connection finishes, OpenVPN extracts encryption and authentication keys from TLS, via the PRF, and uses them to encrypt and authenticate IP packets. Packets are framed using a simple length-type-value envelope, wherein the type specifies the contents of the packet, e.g., channel control (TLS ciphertext) bytes.



- o DTLS-SRTP: DTLS [RFC5764] is commonly used as a way to perform mutual authentication and key agreement for SRTP [RFC5763]. (Here, certificates marshal public keys between endpoints. Thus, self-signed certificates may be used if peers do not mutually trust one another, as is common on the Internet.) When DTLS is used, certificate fingerprints are transmitted out-of-band using SIP. Peers typically verify that DTLS-offered certificates match that which are offered over SIP. This prevents active attacks on RTP, but not on the signaling (SIP or WebRTC) channel.



5. Benefits of Separation

5.1. Reducing Connection Latency

One of the clearest benefits of separating the control protocol from the record protocol is that the cryptographic handshake can be performed out-of-band from the application's data transfer. This should essentially reduce the number of RTTs required before being able to send data by the full length of the handshake (which is commonly 1 or 2 RTTs in the best cases for TLS 1.2 and IKEv2, potentially more if cookie challenges or extended authentication are required).

To avoid long-lived transport connections that wouldn't be actively used, and thus would be vulnerable to timeouts on NATs or firewalls, an obvious approach to separating the control and record protocols is to use different transport connections for the early handshake and the data transfer. However, this approach of using separate connections will not always save RTTs if the cryptographic handshake and data transfer are back-to-back. Each connection may require its own transport protocol handshake, and if the data transfer must wait for two transport protocols to establish and the cryptographic handshake to be finished before sending, then it may experience higher latency. Implementations SHOULD avoid this by either allowing the control and record protocols to share a single transport connection or open two connections in parallel when the control protocol has not pre-fetched keys. Latency benefits, however, can even be achieved when ensuring that this scenario does not occur by always having the control protocol refresh the keys whenever old ones are near expiry.

5.2. Protocol Flexibility

Separation of the control, record, and transport protocols also allows for more flexible composition of protocols with one another. If a deployment uses a control protocol like TLS, which requires a stream-based transport protocol like TCP, separation of protocols will allow it to use the resulting keys for record protocols that run on datagram transport protocols like UDP.

This flexibility may be useful for implementations that are optimizing for packet size by choosing minimal/lightweight record protocols, while being able to use commonly supported control protocols like TLS. One example here is the approach of a VPN tunnel that uses ESP or Diet-ESP [I-D.mglt-ipsecme-diet-esp] to encrypt datagrams, but uses TLS for establishing keys. This design is similar to that used by OpenVPN [OpenVPN], as described above.

5.3. Protocol Capability and Upgrade Negotiation

Enabling the use of a different transport protocol for the actual data transmission than for the cryptographic handshakes opens also the possibility to negotiate protocol capabilities for the data transmission. For TLS, usually TCP is the appropriate transport protocol to use, as it is also widely supported by endpoints. Allowing an endpoint to indicate the support of other, new transport protocols within the TCP connection that is used for the cryptographic handshake, provides a dynamic transition path to enable easy deployment of new protocols. Another example is providing an upgrade path from TCP+TLS to QUIC. If TLS could negotiate the use of other transport layers, such as QUIC, applications could perform an abbreviated upgrade from TCP+TLS connections to QUIC, i.e., without doing a full QUIC handshake.

6. Transport Service Architecture Integration

The Transport Services Architecture ([I-D.ietf-taps-arch]) describes a system that can provide transport security functionality behind a common interface. Such systems and their APIs provide applications with the ability to establish connections for sending and receiving data. The lifetime of a connection is comprised of a pre-establishment configuration stage, established (connected) stage, and terminated stage. Pre-establishment properties configured include: Local and Remote Endpoint, protocol selection properties, and specific protocol options. Applications configure security protocols during pre-establishment using the passive interfaces described in Section 3.1. Active control interfaces are exercised during connection establishment, i.e., from pre-establishment to established states. Applications can query connection metadata or state

information, e.g., peer identity information, during and after connection establishment.

7. IANA Considerations

This document has on request to IANA.

8. Security Considerations

(editor's note: this section will be added later. However, this document discusses the use of cryptographic context for transport connections and as such it has security relevant consideration within the whole document.)

9. Acknowledgments

This work is partially supported by the European Commission under Horizon 2020 grant agreement no. 688421 Measurement and Architecture for a Middleboxed Internet (MAMI), and by the Swiss State Secretariat for Education, Research, and Innovation under contract no. 15.0268. This support does not imply endorsement. Thanks to Brian Trammell for reviewing this draft.

10. Informative References

[I-D.ietf-quic-tls]

Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-13 (work in progress), June 2018.

[I-D.ietf-taps-arch]

Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-ietf-taps-arch-00 (work in progress), April 2018.

[I-D.mglt-ipsecme-diet-esp]

Migault, D., Guggemos, T., Bormann, C., and D. Schinazi, "ESP Header Compression and Diet-ESP", draft-mglt-ipsecme-diet-esp-06 (work in progress), May 2018.

[I-D.moskowitz-sse]

Moskowitz, R., Faynberg, I., Lu, H., Hares, S., and P. Giacomin, "Session Security Envelope", draft-moskowitz-sse-05 (work in progress), June 2017.

- [OpenVPN] "OpenVPN Security Overview", n.d.,
<<https://openvpn.net/index.php/open-source/documentation/security-overview.html>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)",
RFC 4303, DOI 10.17487/RFC4303, December 2005,
<<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security
(TLS) Protocol Version 1.2", RFC 5246,
DOI 10.17487/RFC5246, August 2008,
<<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework
for Establishing a Secure Real-time Transport Protocol
(SRTP) Security Context Using Datagram Transport Layer
Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May
2010, <<https://www.rfc-editor.org/info/rfc5763>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer
Security (DTLS) Extension to Establish Keys for the Secure
Real-time Transport Protocol (SRTP)", RFC 5764,
DOI 10.17487/RFC5764, May 2010,
<<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer
Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347,
January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T.
Kivinen, "Internet Key Exchange Protocol Version 2
(IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October
2014, <<https://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan,
"Transport Layer Security (TLS) Application-Layer Protocol
Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301,
July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.

Authors' Addresses

Mirja Kuehlewind
ETH Zurich
Gloriastrasse 35
8092 Zurich
Switzerland

Email: mirja.kuehlewind@tik.ee.ethz.ch

Tommy Pauly
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Christopher A. Wood
Apple Inc.
One Apple Park Way
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2018

T. Pauly
C. Wood
Apple Inc.
July 03, 2017

A Survey of Transport Security Protocols
draft-pauly-taps-transport-security-00

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, and Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP). This survey is not limited to protocols developed within the scope or context of the IETF.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 4, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	3
2. Terminology	3
3. Transport Security Protocol Descriptions	4
3.1. TLS	5
3.1.1. Protocol Description	5
3.1.2. Protocol Features	6
3.1.3. Protocol Dependencies	6
3.2. DTLS	6
3.2.1. Protocol Description	7
3.2.2. Protocol Features	7
3.2.3. Protocol Dependencies	7
3.3. QUIC with TLS	8
3.3.1. Protocol Description	8
3.3.2. Protocol Features	8
3.3.3. Protocol Dependencies	9
3.4. MinimalT	9
3.4.1. Protocol Description	9
3.4.2. Protocol Features	10
3.4.3. Protocol Dependencies	10
3.5. CurveCP	10
3.5.1. Protocol Description	10
3.5.2. Protocol Features	12
3.5.3. Protocol Dependencies	12
3.6. tcpcrypt	12
3.6.1. Protocol Description	12
3.6.2. Protocol Features	13
3.6.3. Protocol Dependencies	13
3.7. IKEv2 with ESP	13
3.7.1. Protocol descriptions	13
3.7.2. Protocol features	14
3.7.3. Protocol dependencies	15
4. Common Transport Security Features	15
4.1. Mandatory Features	16
4.1.1. Handshake	16
4.1.2. Record	16
4.2. Optional Features	16
4.2.1. Handshake	16
4.2.2. Record	17
5. Transport Security Protocol Interfaces	17

5.1. Configuration Interfaces	17
5.2. Handshake Interfaces	17
5.3. Record Interfaces	18
6. IANA Considerations	20
7. Security Considerations	20
8. Acknowledgments	20
9. Normative References	20
Authors' Addresses	22

1. Introduction

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, and Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP). This survey is not limited to protocols developed within the scope or context of the IETF.

For each protocol, this document provides a brief description, the security features it provides, and the dependencies it has on the underlying transport. This is followed by defining the set of transport security features shared by these protocols. Finally, we distill the application and transport interfaces provided by the transport security protocols.

2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols:

- o Transport 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.
- o Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides a functionality to an application.
- o Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire. A Transport Protocol services an application.

- o Application: an entity that uses a transport protocol for end-to-end delivery of data across the network (this may also be an upper layer protocol or tunnel encapsulation).
- o Security Feature: a specific feature that a network security layer provides to applications. Examples include authentication, encryption, key generation, session resumption, and privacy. A feature may be considered to be Mandatory or Optional to an application's implementation.
- o Security Protocol: a defined network protocol that implements one or more security features. Security protocols may be used alongside transport protocols, and in combination with one another when appropriate.
- o Handshake Protocol: a security protocol that performs a handshake to validate peers and establish a shared cryptographic key.
- o Record Protocol: a security protocol that allows data to be encrypted in records or datagrams based on a shared cryptographic key.
- o Session: an ephemeral security association between applications.
- o Connection: the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.
- o Connection Mobility: a property of a connection that allows it to be multihomed or resilient across network interface or address changes.
- o Peer: an endpoint application party to a session.
- o Client: the peer responsible for initiating a session.
- o Server: the peer responsible for responding to a session initiation.

3. Transport Security Protocol Descriptions

This section contains descriptions of security protocols that currently used to protect data being sent over a network.

For each protocol, we describe the features it provides and its dependencies on other protocols.

3.1. TLS

TLS (Transport Layer Security) [RFC5246] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal (possibly encrypted) data from one peer to the other. This data may contain handshake messages or raw application data.

3.1.1. Protocol Description

TLS is the composition of a handshake and record protocol [I-D.ietf-tls-tls13]. The record protocol is designed to marshal an arbitrary, in-order stream of bytes from one endpoint to the other. It handles segmenting, compressing (when enabled), and encrypting data into discrete records. When configured to use an AEAD algorithm, it also handles nonce generation and encoding for each record. The record protocol is hidden from the client behind a byte stream-oriented API.

The handshake protocol serves several purposes, including: peer authentication, protocol option (key exchange algorithm and ciphersuite) negotiation, and key derivation. Peer authentication may be mutual. However, commonly, only the server is authenticated. X.509 certificates are commonly used in this authentication step, though other mechanisms, such as raw public keys [RFC7250], exist. The client is not authenticated unless explicitly requested by the server with a CertificateRequest handshake message.

The handshake protocol is also extensible. It allows for a variety of extensions to be included by either the client or server. These extensions are used to specify client preferences, e.g., the application-layer protocol to be driven with the TLS connection [RFC7301], or signals to the server to aid operation, e.g., the server name [RFC6066]. Various extensions also exist to tune the parameters of the record protocol, e.g., the maximum fragment length [RFC6066].

Alerts are used to convey errors and other atypical events to the endpoints. There are two classes of alerts: closure and error alerts. A closure alert is used to signal to the other peer that the sender wishes to terminate the connection. The sender typically follows a close alert with a TCP FIN segment to close the connection. Error alerts are used to indicate problems with the handshake or individual records. Most errors are fatal and are followed by

connection termination. However, warning alerts may be handled at the discretion of each respective implementation.

Once a session is disconnected all session keying material must be torn down, unless resumption information was previously negotiated. TLS supports stateful and stateless resumption. (Here, the state refers to the information requirements for the server. It is assumed that the client must always store some state information in order to resume a session.)

3.1.2. Protocol Features

- o Key exchange and ciphersuite algorithm negotiation.
- o Stateful and stateless session resumption.
- o Certificate- and raw public-key-based authentication.
- o Mutual client and server authentication.
- o Byte stream confidentiality and integrity.
- o Extensibility via well-defined extensions.
- o 0-RTT data support (in TLS 1.3 only).
- o Application-layer protocol negotiation.
- o Transparent data segmentation.

3.1.3. Protocol Dependencies

- o TCP for in-order, reliable transport.
- o (Optionally) A PKI trust store for certificate validation.

3.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] is based on TLS, but differs in that it is designed to run over UDP instead of TCP. Since UDP does not guarantee datagram ordering or reliability, DTLS modifies the protocol to make sure it can still provide the same security guarantees as TLS. DTLS was designed to be as close to TLS as possible, so this document will assume that all properties from TLS are carried over except where specified.

3.2.1. Protocol Description

DTLS is modified from TLS to account for packet loss and reordering that occur when operating over a datagram-based transport, i.e., UDP. Each message is assigned an explicit sequence number to be used to reorder on the receiving end. This removes the inter-record dependency and allows each record to be decrypt in isolation of the rest. However, DTLS does not deviate from TLS in that it still provides in-order delivery of data to the application.

With respect to packet loss, if one peer has sent a handshake message and has not yet received its expected response, it will retransmit the handshake message after a configurable timeout.

To account for long records that cannot fit within a single UDP datagram, DTLS supports fragmentation of records across datagrams, keeping track of fragment offsets and lengths in each datagram. The receiving peer must re-assemble records before decrypting.

DTLS relies on UDP's port numbers to allow peers with multiple DTLS sessions between them to demultiplex 'streams' of encrypted packets that share a single TLS session.

Since datagrams may be replayed, DTLS provides anti-replay detection based on a window of acceptable sequence numbers [RFC4303].

3.2.2. Protocol Features

- o Anti-replay protection between datagrams.
- o Basic reliability for handshake messages.
- o See also the features from TLS.

3.2.3. Protocol Dependencies

- o Since DTLS runs over an unreliable, unordered datagram transport, it does not require any reliability features.
- o DTLS contains its own length, so although it runs over a datagram transport, it does not rely on the transport protocol supporting framing.
- o UDP for port numbers used for demultiplexing.
- o Path MTU discovery.

3.3. QUIC with TLS

QUIC (Quick UDP Internet Connections) is a new transport protocol that runs over UDP, and was originally designed with a tight integration with its security protocol and application protocol mappings. The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC over TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

3.3.1. Protocol Description

Since QUIC integrates TLS with its transport, it relies on specific integration points between its security and transport sides. Specifically, these points are:

- o Starting the handshake to generate keys and provide authentication (and providing the transport for the handshake).
- o Client address validation.
- o Key ready events from TLS to notify the QUIC transport.
- o Exporting secrets from TLS to the QUIC transport.

The QUIC transport layer support multiple streams over a single connection. The first stream is reserved specifically for a TLS connection. The TLS handshake, along with further records, are sent over this stream. This TLS connection follows the TLS standards and inherits the security properties of TLS. The handshake generates keys, which are then exported to the rest of the QUIC connection, and are used to protect the rest of the streams.

The initial QUIC messages are sent without encryption in order to start the TLS handshake. Once the handshake has generated keys, the subsequent messages are encrypted. The TLS 1.3 handshake for QUIC is used in either a single-RTT mode or a fast-open zero-RTT mode. When zero-RTT handshakes are possible, the encryption first transitions to use the zero-RTT keys before using single-RTT handshake keys after the next TLS flight.

3.3.2. Protocol Features

- o Handshake properties of TLS.
- o Multiple encrypted streams over a single connection without head-of-line blocking.

- o Packet payload encryption and complete packet authentication (with the exception of the Public Reset packet, which is not authenticated).

3.3.3. Protocol Dependencies

- o QUIC transport relies on UDP.
- o QUIC transport relies on TLS 1.3 for authentication and initial key derivation.
- o TLS within QUIC relies on a reliable stream abstraction for its handshake.

3.4. MinimalT

MinimalT is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility [MinimalT]. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimalT uses a variant of TCP's congestion control algorithm.

3.4.1. Protocol Description

MinimalT is a secure transport protocol built on top of a widespread directory service. Clients and servers interact with local directory services to (a) resolve server information and (b) public ephemeral state information, respectively. Clients connect to a local resolver once at boot time. Through this resolver they recover the IP address(es) and public key(s) of each server to which they want to connect.

Connections are instances of user-authenticated, mobile sessions between two endpoints. Connections run within tunnels between hosts. A tunnel is a server-authenticated container that multiplexes multiple connections between the same hosts. All connections in a tunnel share the same transport state machine and encryption. Each tunnel has a dedicated control connection used to configure and manage the tunnel over time. Moreover, since tunnels are independent of the network address information, they may be reused as both ends of the tunnel move about the network. This does however imply that the connection establishment and packet encryption mechanisms are coupled.

Before a client connects to a remote service, it must first establish a tunnel to the host providing or offering the service. Tunnels are established in 1-RTT using an ephemeral key obtained from the

directory service. Tunnel initiators provide their own ephemeral key and, optionally, a DoS puzzle solution such that the recipient (server) can verify the authenticity of the request and derive a shared secret. Within a tunnel, new connections to services may be established.

3.4.2. Protocol Features

- o 0-RTT forward secrecy for new connections.
- o DoS prevention by client-side puzzles.
- o Tunnel-based mobility.
- o (Transport Feature) Connection multiplexing between hosts across shared tunnels.
- o (Transport Feature) Congestion control state is shared across connections between the same host pairs.

3.4.3. Protocol Dependencies

- o A DNS-like resolution service to obtain location information (an IP address) and ephemeral keys.
- o A PKI trust store for certificate validation.

3.5. CurveCP

CurveCP [CurveCP] is a UDP-based transport security protocol from Daniel J. Bernstein. Unlike other transport security protocols, it is based entirely upon highly efficient public key algorithms. This removes many pitfalls associated with nonce reuse and key synchronization.

3.5.1. Protocol Description

CurveCP is a UDP-based transport security protocol. It is built on three principal features: exclusive use of public key authenticated encryption of packets, server-chosen cookies to prohibit memory and computation DoS at the server, and connection mobility with a client-chosen ephemeral identifier.

There are two rounds in CurveCP. In the first round, the client sends its first initialization packet to the server, carrying its (possibly fresh) ephemeral public key C' , with zero-padding encrypted under the server's long-term public key. The server replies with a cookie and its own ephemeral key S' and a cookie that is to be used

by the client. Upon receipt, the client then generates its second initialization packet carrying: the ephemeral key C' , cookie, and an encryption of C' , the server's domain name, and, optionally, some message data. The server verifies the cookie and the encrypted payload and, if valid, proceeds to send data in return. At this point, the connection is established and the two parties can communicate.

The use of only public-key encryption and authentication, or "boxing", is done to simplify problems that come with symmetric key management and synchronization. For example, it allows the sender of a message to be in complete control of each message's nonce. It does not require either end to share secret keying material. And it allows ephemeral public keys to be associated with connections (or sessions).

The client and server do not perform a standard key exchange. Instead, in the initial exchange of packets, the each party provides its own ephemeral key to the other end. The client can choose a new ephemeral key for every new connection. However, the server must rotate these keys on a slower basis. Otherwise, it would be trivial for an attacker to force the server to create and store ephemeral keys with a fake client initialization packet.

Unlike TCP, the server employs cookies to enable source validation. After receiving the client's initial packet, encrypted under the server's long-term public key, the server generates and returns a stateless cookie that must be echoed back in the client's following message. This cookie is encrypted under the client's ephemeral public key. This stateless technique prevents attackers from hijacking client initialization packets to obtain cookie values to flood clients. (A client would detect the duplicate cookies and reject the flooded packets.) Similarly, replaying the client's second packet, carrying the cookie, will be detected by the server.

CurveCP supports a weak form of client authentication. Clients are permitted to send their long-term public keys in the second initialization packet. A server can verify this public key and, if untrusted, drop the connection and subsequent data.

Unlike some other protocols, CurveCP data packets only leave the ephemeral public key, i.e., the connection ID, and the per-message nonce in the clear. Everything else is encrypted.

3.5.2. Protocol Features

- o Forward-secure data encryption and authentication.
- o Per-packet public-key encryption.
- o 1-RTT session bootstrapping.
- o Connection mobility based on a client-chosen ephemeral identifier.
- o Connection establishment message padding to prevent traffic amplification.
- o Sender-chosen explicit nonces, e.g., based on a sequence number.

3.5.3. Protocol Dependencies

- o An unreliable transport protocol such as UDP.

3.6. tcpcrypt

tcpcrypt is a lightweight extension to the TCP protocol to enable opportunistic encryption.

3.6.1. Protocol Description

tcpcrypt extends TCP to enable opportunistic encryption between the two ends of a TCP connection [I-D.ietf-tcpinc-tcpencrypt]. It is a type of TCP Encryption Protocol (TEP). The use of a TEP is negotiated using TCP headers during the initial TCP handshake. Negotiating a TEP also involves agreeing upon a key exchange algorithm. If and when a TEP is negotiated, the tcpcrypt key exchange occurs within the data segments of the first packets exchanged after the handshake completes. The initiator of a connection sends a list of support AEAD algorithms, a random nonce, and an ephemeral public key share. The responder chooses an AEAD algorithm and replies with its own nonce and ephemeral key share. The traffic encryption keys are derived from the key exchange.

Each tcpcrypt session is associated with a unique session ID; the value of which is derived from the current shared secret used for the session. This can be cached and used to later resume a session. Willingness to resume a session is signaled within the TCP-ENO negotiation option during the TCP handshake [I-D.ietf-tcpinc-tcpeno]. Session identifiers are rotated each time they are resumed. Sessions may also be re-keyed if the natural AEAD limit is reached.

tcpcrypt only encrypts the data portion of a TCP packet. It does not encrypt any header information, such as the TCP sequence number.

3.6.2. Protocol Features

- o Forward-secure TCP packet encryption.
- o Session caching and address-agnostic resumption.
- o Session re-keying.

3.6.3. Protocol Dependencies

- o TCP (with option support).

3.7. IKEv2 with ESP

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either as for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

3.7.1. Protocol descriptions

3.7.1.1. IKEv2

IKEv2 is a control protocol that runs on UDP port 500. Its primary goal is to generate keys for Security Associations (SAs). It first uses a Diffie-Hellman key exchange to generate keys for the "IKE SA", which is a set of keys used to encrypt further IKEv2 messages. It then goes through a phase of authentication in which both peers present blobs signed by a shared secret or private key, after which another set of keys is derived, referred to as the "Child SA". These Child SA keys are used by ESP.

IKEv2 negotiates which protocols are acceptable to each peer for both the IKE and Child SAs using "Proposals". Each proposal may contain an encryption algorithm, an authentication algorithm, a Diffie-Hellman group, and (for IKE SAs only) a pseudorandom function algorithm. Each peer may support multiple proposals, and the most preferred mutually supported proposal is chosen during the handshake.

The authentication phase of IKEv2 may use Shared Secrets, Certificates, Digital Signatures, or an EAP (Extensible

Authentication Protocol) method. At a minimum, IKEv2 takes two round trips to set up both an IKE SA and a Child SA. If EAP is used, this exchange may be expanded.

Any SA used by IKEv2 can be rekeyed upon expiration, which is usually based either on time or number of bytes encrypted.

There is an extension to IKEv2 that allows session resumption [RFC5723].

MOBIKE is a Mobility and Multihoming extension to IKEv2 that allows a set of Security Associations to migrate over different addresses and interfaces [RFC4555].

When UDP is not available or well-supported on a network, IKEv2 may be encapsulated in TCP [I-D.ietf-ipsecme-tcp-encaps].

3.7.1.2. ESP

ESP is a protocol that encrypts and authenticates IP and IPv6 packets. The keys used for both encryption and authentication can be derived from an IKEv2 exchange. ESP Security Associations come as pairs, one for each direction between two peers. Each SA is identified by a Security Parameter Index (SPI), which is marked on each encrypted ESP packet.

ESP packets include the SPI, a sequence number, an optional Initialization Vector (IV), payload data, padding, a length and next header field, and an Integrity Check Value.

From [RFC4303], "ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic flow confidentiality."

Since ESP operates on IP packets, it is not directly tied to the transport protocols it encrypts. This means it requires little or no change from transports in order to provide security.

ESP packets are sent directly over IP, except when a NAT is present, in which case they are sent on UDP port 4500, or via TCP encapsulation [I-D.ietf-ipsecme-tcp-encaps].

3.7.2. Protocol features

3.7.2.1. IKEv2

- o Encryption and authentication of handshake packets.
- o Cryptographic algorithm negotiation.
- o Session resumption.
- o Mobility across addresses and interfaces.
- o Peer authentication extensibility based on Shared Secret, Certificates, Digital Signatures, or EAP methods.

3.7.2.2. ESP

- o Data confidentiality and authentication.
- o Connectionless integrity.
- o Anti-replay protection.
- o Limited flow confidentiality.

3.7.3. Protocol dependencies

3.7.3.1. IKEv2

- o Availability of UDP to negotiate, or implementation support for TCP-encapsulation.
- o Some EAP authentication types require accessing a hardware device, such as a SIM card; or interacting with a user, such as password prompting.

3.7.3.2. ESP

- o Since ESP is below transport protocols, it does not have any dependencies on the transports themselves, other than on UDP or TCP for NAT traversal.

4. Common Transport Security Features

There exists a common set of features shared across the transport protocols surveyed in this document. The mandatory features should be provided by any transport security protocol, while the optional features are extensions that a subset of the protocols provide. For clarity, we also distinguish between handshake and record features.

4.1. Mandatory Features

4.1.1. Handshake

- o Forward-secure segment encryption and authentication: Transit data must be protected with an authenticated encryption algorithm.
- o Private key interface or injection: Authentication based on public key signatures is commonplace for many transport security protocols.
- o Endpoint authentication: The endpoint (receiver) of a new connection must be authenticated before any data is sent to said party.
- o Source validation: Source validation must be provided to mitigate server-targeted DoS attacks. This can be done with puzzles or cookies.

4.1.2. Record

- o Pre-shared key support: A record protocol must be able to use a pre-shared key established out-of-band to encrypt individual messages, packets, or datagrams.

4.2. Optional Features

4.2.1. Handshake

- o Mutual authentication: Transport security protocols should allow both endpoints to authenticate one another if needed.
- o Application-layer feature negotiation: The type of application using a transport security protocol often requires features configured at the connection establishment layer, e.g., ALPN [RFC7301]. Moreover, application-layer features may often be used to offload the session to another server which can better handle the request. (The TLS SNI is one example of such a feature.) As such, transport security protocols should provide a generic mechanism to allow for such application-specific features and options to be configured or otherwise negotiated.
- o Configuration extensions: The protocol negotiation should be extensible with addition of new configuration options.
- o Session caching and management: Sessions should be cacheable to enable reuse and amortize the cost of performing session establishment handshakes.

4.2.2. Record

- o Connection mobility: Sessions should not be bound to a network connection (or 5 tuple). This allows cryptographic key material and other state information to be reused in the event of a connection change. Examples of this include a NAT rebinding that occurs without a client's knowledge.

5. Transport Security Protocol Interfaces

This section describes the interface surface exposed by the security protocols described above, with each interface. Note that not all protocols support each interface.

5.1. Configuration Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or the keys are negotiated.

- o Identity and Private Keys
The application can provide its identities (certificates) and private keys, or mechanisms to access these, to the security protocol to use during handshakes.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2
- o Supported Algorithms (Key Exchange, Signatures and Ciphersuites)
The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2
- o Session Cache
The application provides the ability to save and retrieve session state (tickets, keying material, server parameters) that may be used to resume the security session.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT
- o Authentication Delegate
The application provides access to a separate module that will provide authentication, using EAP for example.
Protocols: IKEv2

5.2. Handshake Interfaces

Handshake interfaces are the points of interaction between a handshake protocol and the application, record protocol, and transport once the handshake is active.

- o Send Handshake Messages

The handshake protocol needs to be able to send messages over a transport to the remote peer to establish trust and negotiate keys.

Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2)

- o Receive Handshake Messages

The handshake protocol needs to be able to receive messages from the remote peer over a transport to establish trust and negotiate keys.

Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2)

- o Identity Validation

During a handshake, the security protocol will conduct identity validation of the peer. This can call into the application to offload validation.

Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2)

- o Source Address Validation

The handshake protocol may delegate validation of the remote peer that has sent data to the transport protocol or application. This involves sending a cookie exchange to avoid DoS attacks.

Protocols: QUIC + TLS

- o Key Update

The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event.

Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2

- o Pre-Shared Key Export

The handshake protocol will generate one or more keys to be used for record encryption/decryption and authentication. These may be explicitly exportable to the application, traditionally limited to direct export to the record protocol, or inherently non-exportable because the keys must be used directly in conjunction with the record protocol.

- * Explicit export: TLS (for QUIC), tcpcrypt, IKEv2

- * Direct export: TLS, DTLS, MinimalT

- * Non-exportable: CurveCP

5.3. Record Interfaces

Record interfaces are the points of interaction between a record protocol and the application, handshake protocol, and transport once in use.

- o Pre-Shared Key Import

Either the handshake protocol or the application directly can supply pre-shared keys for the record protocol use for encryption/decryption and authentication. If the application can supply keys directly, this is considered explicit import; if the handshake protocol traditionally provides the keys directly, it is considered direct import; if the keys can only be shared by the handshake, they are considered non-importable.

 - * Explicit import: QUIC, ESP
 - * Direct import: TLS, DTLS, MinimalT, tcpcrypt
 - * Non-importable: CurveCP
- o Encrypt application data

The application can send data to the record protocol to encrypt it into a format that can be sent on the underlying transport. The encryption step may require that the application data is treated as a stream or as datagrams, and that the transport to send the encrypted records present a stream or datagram interface.

 - * Stream-to-Stream Protocols: TLS, tcpcrypt
 - * Datagram-to-Datagram Protocols: DTLS, ESP
 - * Stream-to-Datagram Protocols: QUIC ((Editor's Note: This depends on the interface QUIC exposes to applications.))
- o Decrypt application data

The application can receive data from its transport to be decrypted using record protocol. The decryption step may require that the incoming transport data is presented as a stream or as datagrams, and that the resulting application data is a stream or datagrams.

 - * Stream-to-Stream Protocols: TLS, tcpcrypt
 - * Datagram-to-Datagram Protocols: DTLS, ESP
 - * Datagram-to-Stream Protocols: QUIC ((Editor's Note: This depends on the interface QUIC exposes to applications.))
- o Key Expiration

The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is

often limited to signaling between the record layer and the handshake layer.

Protocols: ESP ((Editor's note: One may consider TLS/DTLS to also have this interface))

- o Transport mobility

The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation.

Protocols: QUIC, MinimalT, CurveCP, ESP

6. IANA Considerations

This document has on request to IANA.

7. Security Considerations

N/A

8. Acknowledgments

The authors would like to thank Mirja Kuehlewind, Brian Trammell, Yannick Sierra, Frederic Jacobs, and Bob Bradley for their input and feedback on earlier versions of this draft.

9. Normative References

[CurveCP] "CurveCP -- Usable security for the Internet", n.d..

[I-D.ietf-ipsecme-tcp-encaps]

Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", draft-ietf-ipsecme-tcp-encaps-10 (work in progress), May 2017.

[I-D.ietf-quic-tls]

Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-04 (work in progress), June 2017.

[I-D.ietf-quic-transport]

Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-04 (work in progress), June 2017.

- [I-D.ietf-tcpinc-tcpencrypt]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpencrypt)", draft-ietf-tcpinc-tcpencrypt-06 (work in progress), March 2017.
- [I-D.ietf-tcpinc-tcpext]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-EXT: Extension Negotiation Option", draft-ietf-tcpinc-tcpext-08 (work in progress), March 2017.
- [I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-20 (work in progress), April 2017.
- [MinimalT]
"MinimalT -- Minimal-latency Networking Through Better Security", n.d..
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<http://www.rfc-editor.org/info/rfc4303>>.
- [RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <<http://www.rfc-editor.org/info/rfc4555>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [RFC5723] Sheffer, Y. and H. Tschofenig, "Internet Key Exchange Protocol Version 2 (IKEv2) Session Resumption", RFC 5723, DOI 10.17487/RFC5723, January 2010, <<http://www.rfc-editor.org/info/rfc5723>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<http://www.rfc-editor.org/info/rfc6066>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<http://www.rfc-editor.org/info/rfc6347>>.

- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<http://www.rfc-editor.org/info/rfc7250>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<http://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<http://www.rfc-editor.org/info/rfc8095>>.

Authors' Addresses

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Christopher A. Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

T. Pauly
Apple Inc.
C. Perkins
University of Glasgow
K. Rose
Akamai Technologies, Inc.
C. Wood
Apple Inc.
March 05, 2018

A Survey of Transport Security Protocols
draft-pauly-taps-transport-security-02

Abstract

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. This survey is not limited to protocols developed within the scope or context of the IETF.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 6, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	3
2. Terminology	4
3. Transport Security Protocol Descriptions	5
3.1. TLS	5
3.1.1. Protocol Description	6
3.1.2. Protocol Features	7
3.1.3. Protocol Dependencies	7
3.2. DTLS	7
3.2.1. Protocol Description	7
3.2.2. Protocol Features	8
3.2.3. Protocol Dependencies	8
3.3. (IETF) QUIC with TLS	9
3.3.1. Protocol Description	9
3.3.2. Protocol Features	9
3.3.3. Protocol Dependencies	10
3.4. gQUIC	10
3.4.1. Protocol Description	10
3.4.2. Protocol Dependencies	10
3.5. MinimalT	10
3.5.1. Protocol Description	10
3.5.2. Protocol Features	11
3.5.3. Protocol Dependencies	11
3.6. CurveCP	12
3.6.1. Protocol Description	12
3.6.2. Protocol Features	13
3.6.3. Protocol Dependencies	13
3.7. tcpcrypt	13
3.7.1. Protocol Description	13
3.7.2. Protocol Features	14
3.7.3. Protocol Dependencies	15
3.8. IKEv2 with ESP	15

3.8.1.	Protocol descriptions	15
3.8.2.	Protocol features	16
3.8.3.	Protocol dependencies	17
3.9.	WireGuard	17
3.9.1.	Protocol description	17
3.9.2.	Protocol features	18
3.9.3.	Protocol dependencies	18
3.10.	SRTP (with DTLS)	18
3.10.1.	Protocol descriptions	19
3.10.2.	Protocol features	20
3.10.3.	Protocol dependencies	20
4.	Common Transport Security Features	20
4.1.	Mandatory Features	20
4.1.1.	Handshake	20
4.1.2.	Record	21
4.2.	Optional Features	21
4.2.1.	Handshake	21
4.2.2.	Record	21
5.	Transport Security Protocol Interfaces	21
5.1.	Configuration Interfaces	22
5.2.	Handshake Interfaces	22
5.3.	Record Interfaces	23
6.	IANA Considerations	25
7.	Security Considerations	25
8.	Acknowledgments	25
9.	Normative References	25
	Authors' Addresses	29

1. Introduction

This document provides a survey of commonly used or notable network security protocols, with a focus on how they interact and integrate with applications and transport protocols. Its goal is to supplement efforts to define and catalog transport services [RFC8095] by describing the interfaces required to add security protocols. It examines Transport Layer Security (TLS), Datagram Transport Layer Security (DTLS), Quick UDP Internet Connections with TLS (QUIC + TLS), MinimalT, CurveCP, tcpcrypt, Internet Key Exchange with Encapsulating Security Protocol (IKEv2 + ESP), SRTP (with DTLS), and WireGuard. This survey is not limited to protocols developed within the scope or context of the IETF.

For each protocol, this document provides a brief description, the security features it provides, and the dependencies it has on the underlying transport. This is followed by defining the set of transport security features shared by these protocols. Finally, we distill the application and transport interfaces provided by the transport security protocols.

Authentication-only protocols such as TCP-AO [RFC5925] and IPsec AH [RFC4302] are excluded from this survey. TCP-AO adds authenticity protections to long-lived TCP connections, e.g., replay protection with per-packet Message Authentication Codes. (This protocol obsoletes TCP MD5 "signature" options specified in [RFC2385].) One prime use case of TCP-AO is for protecting BGP connections. Similarly, AH adds per-datagram authenticity and adds similar replay protection. Despite these improvements, neither protocol sees general use and both lack critical properties important for emergent transport security protocols: confidentiality, privacy protections, and agility. Thus, we omit these and related protocols from our survey.

2. Terminology

The following terms are used throughout this document to describe the roles and interactions of transport security protocols:

- o Transport 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.
- o Transport Service: a set of Transport Features, without an association to any given framing protocol, which provides functionality to an application.
- o Transport Protocol: an implementation that provides one or more different transport services using a specific framing and header format on the wire. A Transport Protocol services an application.
- o Application: an entity that uses a transport protocol for end-to-end delivery of data across the network. This may also be an upper layer protocol or tunnel encapsulation.
- o Security Feature: a specific feature that a network security layer provides to applications. Examples include authentication, encryption, key generation, session resumption, and privacy. A feature may be considered to be Mandatory or Optional to an application's implementation.
- o Security Protocol: a defined network protocol that implements one or more security features. Security protocols may be used alongside transport protocols, and in combination with other security protocols when appropriate.
- o Handshake Protocol: a protocol that enables peers to validate each other and to securely establish shared cryptographic context.

- o Record Protocol: a security protocol that allows data to be divided into manageable blocks and protected using a shared cryptographic context.
- o Session: an ephemeral security association between applications.
- o Cryptographic context: a set of cryptographic parameters, including but not necessarily limited to keys for encryption, authentication, and session resumption, enabling authorized parties to a session to communicate securely.
- o Connection: the shared state of two or more endpoints that persists across messages that are transmitted between these endpoints. A connection is a transient participant of a session, and a session generally lasts between connection instances.
- o Connection Mobility: a property of a connection that allows it to be multihomed or resilient across network interface or address changes.
- o Peer: an endpoint application party to a session.
- o Client: the peer responsible for initiating a session.
- o Server: the peer responsible for responding to a session initiation.

3. Transport Security Protocol Descriptions

This section contains descriptions of security protocols that currently used to protect data being sent over a network.

For each protocol, we describe the features it provides and its dependencies on other protocols.

3.1. TLS

TLS (Transport Layer Security) [RFC5246] is a common protocol used to establish a secure session between two endpoints. Communication over this session "prevents eavesdropping, tampering, and message forgery." TLS consists of a tightly coupled handshake and record protocol. The handshake protocol is used to authenticate peers, negotiate protocol options, such as cryptographic algorithms, and derive session-specific keying material. The record protocol is used to marshal (possibly encrypted) data from one peer to the other. This data may contain handshake messages or raw application data.

3.1.1.1. Protocol Description

TLS is the composition of a handshake and record protocol [I-D.ietf-tls-tls13]. The record protocol is designed to marshal an arbitrary, in-order stream of bytes from one endpoint to the other. It handles segmenting, compressing (when enabled), and encrypting data into discrete records. When configured to use an AEAD algorithm, it also handles nonce generation and encoding for each record. The record protocol is hidden from the client behind a byte stream-oriented API.

The handshake protocol serves several purposes, including: peer authentication, protocol option (key exchange algorithm and ciphersuite) negotiation, and key derivation. Peer authentication may be mutual; however, commonly, only the server is authenticated. X.509 certificates are commonly used in this authentication step, though other mechanisms, such as raw public keys [RFC7250], exist. The client is not authenticated unless explicitly requested by the server with a CertificateRequest handshake message. Assuming strong cryptography, an infrastructure for trust establishment, correctly-functioning endpoints, and communication patterns free from side channels, server authentication is sufficient to establish a channel resistant to eavesdroppers.

The handshake protocol is also extensible. It allows for a variety of extensions to be included by either the client or server. These extensions are used to specify client preferences, e.g., the application-layer protocol to be driven with the TLS connection [RFC7301], or signals to the server to aid operation, e.g., Server Name Indication (SNI) [RFC6066]. Various extensions also exist to tune the parameters of the record protocol, e.g., the maximum fragment length [RFC6066].

Alerts are used to convey errors and other atypical events to the endpoints. There are two classes of alerts: closure and error alerts. A closure alert is used to signal to the other peer that the sender wishes to terminate the connection. The sender typically follows a close alert with a TCP FIN segment to close the connection. Error alerts are used to indicate problems with the handshake or individual records. Most errors are fatal and are followed by connection termination. However, warning alerts may be handled at the discretion of the implementation.

Once a session is disconnected all session keying material must be destroyed, with the exception of secrets previously established expressly for purposes of session resumption. TLS supports stateful and stateless resumption. (Here, "state" refers to bookkeeping on a

per-session basis by the server. It is assumed that the client must always store some state information in order to resume a session.)

3.1.2. Protocol Features

- o Key exchange and ciphersuite algorithm negotiation.
- o Stateful and stateless session resumption.
- o Certificate- and raw public key-based authentication.
- o Mutual client and server authentication.
- o Byte stream confidentiality and integrity.
- o Extensibility via well-defined extensions.
- o 0-RTT data support (starting with TLS 1.3).
- o Application-layer protocol negotiation.
- o Transparent data segmentation.

3.1.3. Protocol Dependencies

- o TCP for in-order, reliable transport.
- o (Optionally) A PKI trust store for certificate validation.

3.2. DTLS

DTLS (Datagram Transport Layer Security) [RFC6347] is based on TLS, but differs in that it is designed to run over UDP instead of TCP. Since UDP does not guarantee datagram ordering or reliability, DTLS modifies the protocol to make sure it can still provide the same security guarantees as TLS. DTLS was designed to be as close to TLS as possible, so this document will assume that all properties from TLS are carried over except where specified.

3.2.1. Protocol Description

DTLS is modified from TLS to account for packet loss, reordering, and duplication that may occur when operating over UDP. To enable out-of-order delivery of application data, the DTLS record protocol itself has no inter-record dependencies. However, as the handshake requires reliability, each handshake message is assigned an explicit sequence number to enable retransmissions of lost packets and in-order processing by the receiver. Handshake message loss is remedied

by sender retransmission after a configurable period in which the expected response has not yet been received.

As the DTLS handshake protocol runs atop the record protocol, to account for long handshake messages that cannot fit within a single record, DTLS supports fragmentation and subsequent reconstruction of handshake messages across records. The receiver must reassemble records before processing.

DTLS relies on unique UDP 4-tuples to allow peers with multiple DTLS connections between them to demultiplex connections, constraining protocol design slightly more than UDP: application-layer demultiplexing over the same 4-tuple is not possible without trial decryption as all application-layer data is encrypted to a connection-specific cryptographic context. Starting with DTLS 1.3 [I-D.ietf-tls-dtls13], a connection identifier extension to permit multiplexing of independent connections over the same 4-tuple is available [I-D.ietf-tls-dtls-connection-id].

Since datagrams may be replayed, DTLS provides optional anti-replay detection based on a window of acceptable sequence numbers [RFC6347].

3.2.2. Protocol Features

- o Anti-replay protection between datagrams.
- o Basic reliability for handshake messages.
- o See also the features from TLS.

3.2.3. Protocol Dependencies

- o Since DTLS runs over an unreliable, unordered datagram transport, it does not require any reliability features.
- o The DTLS record protocol explicitly encodes record lengths, so although it runs over a datagram transport, it does not rely on the transport protocol's framing beyond requiring transport-level reconstruction of datagrams fragmented over packets.
- o UDP 4-tuple uniqueness, or the connection identifier extension, for demultiplexing.
- o Path MTU discovery.

3.3. (IETF) QUIC with TLS

QUIC (Quick UDP Internet Connections) is a new standards-track transport protocol that runs over UDP, loosely based on Google's original proprietary gQUIC protocol. (See Section 3.4 for more details.) The QUIC transport layer itself provides support for data confidentiality and integrity. This requires keys to be derived with a separate handshake protocol. A mapping for QUIC over TLS 1.3 [I-D.ietf-quic-tls] has been specified to provide this handshake.

3.3.1. Protocol Description

As QUIC relies on TLS to secure its transport functions, it creates specific integration points between its security and transport functions:

- o Starting the handshake to generate keys and provide authentication (and providing the transport for the handshake).
- o Client address validation.
- o Key ready events from TLS to notify the QUIC transport.
- o Exporting secrets from TLS to the QUIC transport.

The QUIC transport layer support multiple streams over a single connection. The first stream is reserved specifically for a TLS connection. The TLS handshake, along with further records, are sent over this stream. This TLS connection follows the TLS standards and inherits the security properties of TLS. The handshake generates keys, which are then exported to the rest of the QUIC connection, and are used to protect the rest of the streams.

Initial QUIC messages (packets) are encrypted using "fixed" keys derived from the QUIC version and public packet information (Connection ID). Packets are later encrypted using keys derived from the TLS traffic secret upon handshake completion. The TLS 1.3 handshake for QUIC is used in either a single-RTT mode or a fast-open zero-RTT mode. When zero-RTT handshakes are possible, the encryption first transitions to use the zero-RTT keys before using single-RTT handshake keys after the next TLS flight.

3.3.2. Protocol Features

- o Handshake properties of TLS.
- o Multiple encrypted streams over a single connection without head-of-line blocking.

- o Packet payload encryption and complete packet authentication (with the exception of the Public Reset packet, which is not authenticated).

3.3.3. Protocol Dependencies

- o QUIC transport relies on UDP.
- o QUIC transport relies on TLS 1.3 for authentication and initial key derivation.
- o TLS within QUIC relies on a reliable stream abstraction for its handshake.

3.4. gQUIC

gQUIC is a UDP-based multiplexed streaming protocol designed and deployed by Google following experience from deploying SPDY, the proprietary predecessor to HTTP/2. gQUIC was originally known as "QUIC": this document uses gQUIC to unambiguously distinguish it from the standards-track IETF QUIC. The proprietary technical forebear of IETF QUIC, gQUIC was originally designed with tightly-integrated security and application data transport protocols.

3.4.1. Protocol Description

((TODO: write me))

3.4.2. Protocol Dependencies

((TODO: write me))

3.5. MinimalT

MinimalT is a UDP-based transport security protocol designed to offer confidentiality, mutual authentication, DoS prevention, and connection mobility [MinimalT]. One major goal of the protocol is to leverage existing protocols to obtain server-side configuration information used to more quickly bootstrap a connection. MinimalT uses a variant of TCP's congestion control algorithm.

3.5.1. Protocol Description

MinimalT is a secure transport protocol built on top of a widespread directory service. Clients and servers interact with local directory services to (a) resolve server information and (b) public ephemeral state information, respectively. Clients connect to a local resolver once at boot time. Through this resolver they recover the IP

address(es) and public key(s) of each server to which they want to connect.

Connections are instances of user-authenticated, mobile sessions between two endpoints. Connections run within tunnels between hosts. A tunnel is a server-authenticated container that multiplexes multiple connections between the same hosts. All connections in a tunnel share the same transport state machine and encryption. Each tunnel has a dedicated control connection used to configure and manage the tunnel over time. Moreover, since tunnels are independent of the network address information, they may be reused as both ends of the tunnel move about the network. This does however imply that the connection establishment and packet encryption mechanisms are coupled.

Before a client connects to a remote service, it must first establish a tunnel to the host providing or offering the service. Tunnels are established in 1-RTT using an ephemeral key obtained from the directory service. Tunnel initiators provide their own ephemeral key and, optionally, a DoS puzzle solution such that the recipient (server) can verify the authenticity of the request and derive a shared secret. Within a tunnel, new connections to services may be established.

3.5.2. Protocol Features

- o 0-RTT forward secrecy for new connections.
- o DoS prevention by client-side puzzles.
- o Tunnel-based mobility.
- o (Transport Feature) Connection multiplexing between hosts across shared tunnels.
- o (Transport Feature) Congestion control state is shared across connections between the same host pairs.

3.5.3. Protocol Dependencies

- o A DNS-like resolution service to obtain location information (an IP address) and ephemeral keys.
- o A PKI trust store for certificate validation.

3.6. CurveCP

CurveCP [CurveCP] is a UDP-based transport security protocol from Daniel J. Bernstein. Unlike other transport security protocols, it is based entirely upon highly efficient public key algorithms. This removes many pitfalls associated with nonce reuse and key synchronization.

3.6.1. Protocol Description

CurveCP is a UDP-based transport security protocol. It is built on three principal features: exclusive use of public key authenticated encryption of packets, server-chosen cookies to prohibit memory and computation DoS at the server, and connection mobility with a client-chosen ephemeral identifier.

There are two rounds in CurveCP. In the first round, the client sends its first initialization packet to the server, carrying its (possibly fresh) ephemeral public key C' , with zero-padding encrypted under the server's long-term public key. The server replies with a cookie and its own ephemeral key S' and a cookie that is to be used by the client. Upon receipt, the client then generates its second initialization packet carrying: the ephemeral key C' , cookie, and an encryption of C' , the server's domain name, and, optionally, some message data. The server verifies the cookie and the encrypted payload and, if valid, proceeds to send data in return. At this point, the connection is established and the two parties can communicate.

The use of only public-key encryption and authentication, or "boxing", is done to simplify problems that come with symmetric key management and synchronization. For example, it allows the sender of a message to be in complete control of each message's nonce. It does not require either end to share secret keying material. Furthermore, it allows connections (or sessions) to be associated with unique ephemeral public keys as a mechanism for enabling forward secrecy given the risk of long-term private key compromise.

The client and server do not perform a standard key exchange. Instead, in the initial exchange of packets, each party provides its own ephemeral key to the other end. The client can choose a new ephemeral key for every new connection. However, the server must rotate these keys on a slower basis. Otherwise, it would be trivial for an attacker to force the server to create and store ephemeral keys with a fake client initialization packet.

Unlike TCP, the server employs cookies to enable source validation. After receiving the client's initial packet, encrypted under the

server's long-term public key, the server generates and returns a stateless cookie that must be echoed back in the client's following message. This cookie is encrypted under the client's ephemeral public key. This stateless technique prevents attackers from hijacking client initialization packets to obtain cookie values to flood clients. (A client would detect the duplicate cookies and reject the flooded packets.) Similarly, replaying the client's second packet, carrying the cookie, will be detected by the server.

CurveCP supports a weak form of client authentication. Clients are permitted to send their long-term public keys in the second initialization packet. A server can verify this public key and, if untrusted, drop the connection and subsequent data.

Unlike some other protocols, CurveCP data packets leave only the ephemeral public key, the connection ID, and the per-message nonce in the clear. Everything else is encrypted.

3.6.2. Protocol Features

- o Forward-secure data encryption and authentication.
- o Per-packet public-key encryption.
- o 1-RTT session bootstrapping.
- o Connection mobility based on a client-chosen ephemeral identifier.
- o Connection establishment message padding to prevent traffic amplification.
- o Sender-chosen explicit nonces, e.g., based on a sequence number.

3.6.3. Protocol Dependencies

- o An unreliable transport protocol such as UDP.

3.7. tcpcrypt

Tcpcrypt is a lightweight extension to the TCP protocol to enable opportunistic encryption with hooks available to the application layer for implementation of endpoint authentication.

3.7.1. Protocol Description

Tcpcrypt extends TCP to enable opportunistic encryption between the two ends of a TCP connection [I-D.ietf-tcpinc-tcpcrypt]. It is a family of TCP encryption protocols (TEP), distinguished by key

exchange algorithm. The use of a TEP is negotiated with a TCP option during the initial TCP handshake via the mechanism described by TCP Encryption Negotiation Option (ENO) [I-D.ietf-tcpinc-tcpno]. In the case of initial session establishment, once a tcpcrypt TEP has been negotiated the key exchange occurs within the data segments of the first few packets exchanged after the handshake completes. The initiator of a connection sends a list of supported AEAD algorithms, a random nonce, and an ephemeral public key share. The responder typically chooses a mutually-supported AEAD algorithm and replies with this choice, its own nonce, and ephemeral key share. An initial shared secret is derived from the ENO handshake, the tcpcrypt handshake, and the initial keying material resulting from the key exchange. The traffic encryption keys on the initial connection are derived from the shared secret. Connections can be re-keyed before the natural AEAD limit for a single set of traffic encryption keys is reached.

Each tcpcrypt session is associated with a ladder of resumption IDs, each derived from the respective entry in a ladder of shared secrets. These resumption IDs can be used to negotiate a stateful resumption of the session in a subsequent connection, resulting in use of a new shared secret and traffic encryption keys without requiring a new key exchange. Willingness to resume a session is signaled via the ENO option during the TCP handshake. Given the length constraints imposed by TCP options, unlike stateless resumption mechanisms (such as that provided by session tickets in TLS) resumption in tcpcrypt requires the maintenance of state on the server, and so successful resumption across a pool of servers implies shared state.

Owing to middlebox ossification issues, tcpcrypt only protects the payload portion of a TCP packet. It does not encrypt any header information, such as the TCP sequence number.

Tcpencrypt exposes a universally-unique connection-specific session ID to the application, suitable for application-level endpoint authentication either in-band or out-of-band.

3.7.2. Protocol Features

- o Forward-secure TCP payload encryption and integrity protection.
- o Session caching and address-agnostic resumption.
- o Connection re-keying.
- o Application-level authentication primitive.

3.7.3. Protocol Dependencies

- o TCP
- o TCP Encryption Negotiation Option (ENO)

3.8. IKEv2 with ESP

IKEv2 [RFC7296] and ESP [RFC4303] together form the modern IPsec protocol suite that encrypts and authenticates IP packets, either as for creating tunnels (tunnel-mode) or for direct transport connections (transport-mode). This suite of protocols separates out the key generation protocol (IKEv2) from the transport encryption protocol (ESP). Each protocol can be used independently, but this document considers them together, since that is the most common pattern.

3.8.1. Protocol descriptions

3.8.1.1. IKEv2

IKEv2 is a control protocol that runs on UDP port 500. Its primary goal is to generate keys for Security Associations (SAs). It first uses a Diffie-Hellman key exchange to generate keys for the "IKE SA", which is a set of keys used to encrypt further IKEv2 messages. It then goes through a phase of authentication in which both peers present blobs signed by a shared secret or private key, after which another set of keys is derived, referred to as the "Child SA". These Child SA keys are used by ESP.

IKEv2 negotiates which protocols are acceptable to each peer for both the IKE and Child SAs using "Proposals". Each proposal may contain an encryption algorithm, an authentication algorithm, a Diffie-Hellman group, and (for IKE SAs only) a pseudorandom function algorithm. Each peer may support multiple proposals, and the most preferred mutually supported proposal is chosen during the handshake.

The authentication phase of IKEv2 may use Shared Secrets, Certificates, Digital Signatures, or an EAP (Extensible Authentication Protocol) method. At a minimum, IKEv2 takes two round trips to set up both an IKE SA and a Child SA. If EAP is used, this exchange may be expanded.

Any SA used by IKEv2 can be rekeyed upon expiration, which is usually based either on time or number of bytes encrypted.

There is an extension to IKEv2 that allows session resumption [RFC5723].

MOBIKE is a Mobility and Multihoming extension to IKEv2 that allows a set of Security Associations to migrate over different addresses and interfaces [RFC4555].

When UDP is not available or well-supported on a network, IKEv2 may be encapsulated in TCP [RFC8229].

3.8.1.2. ESP

ESP is a protocol that encrypts and authenticates IPv4 and IPv6 packets. The keys used for both encryption and authentication can be derived from an IKEv2 exchange. ESP Security Associations come as pairs, one for each direction between two peers. Each SA is identified by a Security Parameter Index (SPI), which is marked on each encrypted ESP packet.

ESP packets include the SPI, a sequence number, an optional Initialization Vector (IV), payload data, padding, a length and next header field, and an Integrity Check Value.

From [RFC4303], "ESP is used to provide confidentiality, data origin authentication, connectionless integrity, an anti-replay service (a form of partial sequence integrity), and limited traffic flow confidentiality."

Since ESP operates on IP packets, it is not directly tied to the transport protocols it encrypts. This means it requires little or no change from transports in order to provide security.

ESP packets may be sent directly over IP, but where network conditions warrant (e.g., when a NAT is present or when a firewall blocks such packets) they may be encapsulated in UDP [RFC3948] or TCP [RFC8229].

3.8.2. Protocol features

3.8.2.1. IKEv2

- o Encryption and authentication of handshake packets.
- o Cryptographic algorithm negotiation.
- o Session resumption.
- o Mobility across addresses and interfaces.
- o Peer authentication extensibility based on shared secret, certificates, digital signatures, or EAP methods.

3.8.2.2. ESP

- o Data confidentiality and authentication.
- o Connectionless integrity.
- o Anti-replay protection.
- o Limited flow confidentiality.

3.8.3. Protocol dependencies

3.8.3.1. IKEv2

- o Availability of UDP to negotiate, or implementation support for TCP-encapsulation.
- o Some EAP authentication types require accessing a hardware device, such as a SIM card; or interacting with a user, such as password prompting.

3.8.3.2. ESP

- o Since ESP is below transport protocols, it does not have any dependencies on the transports themselves, other than on UDP or TCP where encapsulation is employed.

3.9. WireGuard

WireGuard is a layer 3 protocol designed to complement or replace IPsec [WireGuard]. Unlike most transport security protocols, which rely on PKI for peer authentication, WireGuard authenticates peers using pre-shared public keys delivered out-of-band, each of which is bound to one or more IP addresses. Moreover, as a protocol suited for VPNs, WireGuard offers no extensibility, negotiation, or cryptographic agility.

3.9.1. Protocol description

WireGuard is a simple VPN protocol that binds a pre-shared public key to one or more IP addresses. Users configure WireGuard by associating peer public keys with IP addresses. These mappings are stored in a CryptoKey Routing Table. (See Section 2 of [WireGuard] for more details and sample configurations.) These keys are used upon WireGuard packet transmission and reception. For example, upon receipt of a Handshake Initiation message, receivers use the static public key in their CryptoKey routing table to perform necessary cryptographic computations.

WireGuard builds on Noise [Noise] for 1-RTT key exchange with identity hiding. The handshake hides peer identities as per the SIGMA construction [SIGMA]. As a consequence of using Noise, WireGuard comes with a fixed set of cryptographic algorithms:

- o x25519 [Curve25519] and HKDF [RFC5869] for ECDH and key derivation.
- o ChaCha20+Poly1305 [RFC7539] for packet authenticated encryption.
- o BLAKE2s [BLAKE2] for hashing.

There is no cryptographic agility. If weaknesses are found in any of these algorithms, new message types using new algorithms must be introduced.

WireGuard is designed to be entirely stateless, modulo the CryptoKey routing table, which has size linear with the number of trusted peers. If a WireGuard receiver is under heavy load and cannot process a packet, e.g., cannot spare CPU cycles for point multiplication, it can reply with a cookie similar to DTLS and IKEv2. This cookie only proves IP address ownership. Any rate limiting scheme can be applied to packets coming from non-spoofed addresses.

3.9.2. Protocol features

- o Optional PSK-based session creation.
- o Mutual client and server authentication.
- o Stateful, timestamp-based replay prevention.
- o Cookie-based DoS mitigation similar to DTLS and IKEv2.

3.9.3. Protocol dependencies

- o Datagram transport.
- o Out-of-band key distribution and management.

3.10. SRTP (with DTLS)

SRTP - Secure RTP - is a profile for RTP that provides confidentiality, message authentication, and replay protection for data and control packets [RFC3711]. SRTP packets are encrypted using a session key, which is derived from a separate master key. Master keys are derived and managed externally, e.g., via DTLS, as specified

in RFC 5763 [RFC5763], under the control of a signaling protocol such as SIP [RFC3261] or WebRTC [I-D.ietf-rtcweb-security-arch].

3.10.1. Protocol descriptions

SRTP adds confidentiality and optional integrity protection to RTP data packets, and adds confidentiality and mandatory integrity protection to RTP control (RTCP) packets. For RTP data packets, this is done by encrypting the payload section of the packet and optionally appending an authentication tag (MAC) as a packet trailer, with the RTP header authenticated but not encrypted. The RTP header itself is left unencrypted to enable RTP header compression [RFC2508][RFC3545]. For RTCP packets, the first packet in the compound RTCP packet is partially encrypted, leaving the first eight octets of the header as cleartext to allow identification of the packet as RTCP, while the remainder of the compound packet is fully encrypted. The entire RTCP packet is then authenticated by appending a MAC as packet trailer.

Packets are encrypted using session keys, which are ultimately derived from a master key and some additional master salt and session salt. SRTP packets carry a 2-byte sequence number to partially identify the unique packet index. SRTP peers maintain a separate rollover counter (ROC) for RTP data packets that is incremented whenever the sequence number wraps. The sequence number and ROC together determine the packet index. RTCP packets have a similar, yet differently named, field called the RTCP index which serves the same purpose.

Numerous encryption modes are supported. For popular modes of operation, e.g., AES-CTR, the (unique) initialization vector (IV) used for each encryption mode is a function of the RTP SSRC (synchronization source), packet index, and session "salting key".

SRTP offers replay detection by keeping a replay list of already seen and processed packet indices. If a packet arrives with an index that matches one in the replay list, it is silently discarded.

DTLS [RFC5764] is commonly used as a way to perform mutual authentication and key agreement for SRTP [RFC5763]. (Here, certificates marshal public keys between endpoints. Thus, self-signed certificates may be used if peers do not mutually trust one another, as is common on the Internet.) When DTLS is used, certificate fingerprints are transmitted out-of-band using SIP. Peers typically verify that DTLS-offered certificates match that which are offered over SIP. This prevents active attacks on RTP, but not on the signaling (SIP or WebRTC) channel.

3.10.2. Protocol features

- o Optional replay protection with tunable replay windows.
- o Out-of-order packet receipt.
- o (RFC5763) Mandatory mutually authenticated key exchange.
- o Partial encryption, protecting media payloads and control packets but not data packet headers.
- o Optional authentication of data packets; mandatory authentication of control packets.

3.10.3. Protocol dependencies

- o External key derivation and management mechanism or protocol, e.g., DTLS [RFC5763].
- o External signaling protocol to manage RTP parameters and locate and identify peers, e.g., SIP [RFC3261] or WebRTC [I-D.ietf-rtcweb-security-arch].

4. Common Transport Security Features

There exists a common set of features shared across the transport protocols surveyed in this document. The mandatory features should be provided by any transport security protocol, while the optional features are extensions that a subset of the protocols provide. For clarity, we also distinguish between handshake and record features.

4.1. Mandatory Features

4.1.1. Handshake

- o Forward-secure segment encryption and authentication: Transit data must be protected with an authenticated encryption algorithm.
- o Private key interface or injection: Authentication based on public key signatures is commonplace for many transport security protocols.
- o Endpoint authentication: The endpoint (receiver) of a new connection must be authenticated before any data is sent to said party.

- o Source validation: Source validation must be provided to mitigate server-targeted DoS attacks. This can be done with puzzles or cookies.

4.1.2. Record

- o Pre-shared key support: A record protocol must be able to use a pre-shared key established out-of-band to encrypt individual messages, packets, or datagrams.

4.2. Optional Features

4.2.1. Handshake

- o Mutual authentication: Transport security protocols must allow each endpoint to authenticate the other if required by the application.
- o Application-layer feature negotiation: The type of application using a transport security protocol often requires features configured at the connection establishment layer, e.g., ALPN [RFC7301]. Moreover, application-layer features may often be used to offload the session to another server which can better handle the request. (The TLS SNI is one example of such a feature.) As such, transport security protocols should provide a generic mechanism to allow for such application-specific features and options to be configured or otherwise negotiated.
- o Configuration extensions: The protocol negotiation should be extensible with addition of new configuration options.
- o Session caching and management: Sessions should be cacheable to enable reuse and amortize the cost of performing session establishment handshakes.

4.2.2. Record

- o Connection mobility: Sessions should not be bound to a network connection (or 5-tuple). This allows cryptographic key material and other state information to be reused in the event of a connection change. Examples of this include a NAT rebinding that occurs without a client's knowledge.

5. Transport Security Protocol Interfaces

This section describes the interface surface exposed by the security protocols described above, with each interface. Note that not all protocols support each interface.

5.1. Configuration Interfaces

Configuration interfaces are used to configure the security protocols before a handshake begins or the keys are negotiated.

- o Identity and Private Keys
The application can provide its identities (certificates) and private keys, or mechanisms to access these, to the security protocol to use during handshakes.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP
- o Supported Algorithms (Key Exchange, Signatures, and Ciphersuites)
The application can choose the algorithms that are supported for key exchange, signatures, and ciphersuites.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2, SRTP
- o Session Cache
The application provides the ability to save and retrieve session state (such as tickets, keying material, and server parameters) that may be used to resume the security session.
Protocols: TLS, DTLS, QUIC + TLS, MinimalT
- o Authentication Delegation
The application provides access to a separate module that will provide authentication, using EAP for example.
Protocols: IKEv2, SRTP

5.2. Handshake Interfaces

Handshake interfaces are the points of interaction between a handshake protocol and the application, record protocol, and transport once the handshake is active.

- o Send Handshake Messages
The handshake protocol needs to be able to send messages over a transport to the remote peer to establish trust and to negotiate keys.
Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))
- o Receive Handshake Messages
The handshake protocol needs to be able to receive messages from the remote peer over a transport to establish trust and to negotiate keys.
Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))

- o Identity Validation
During a handshake, the security protocol will conduct identity validation of the peer. This can call into the application to offload validation. Protocols: All (TLS, DTLS, QUIC + TLS, MinimalT, CurveCP, IKEv2, WireGuard, SRTP (DTLS))
- o Source Address Validation
The handshake protocol may delegate validation of the remote peer that has sent data to the transport protocol or application. This involves sending a cookie exchange to avoid DoS attacks. Protocols: QUIC + TLS, DTLS, WireGuard
- o Key Update
The handshake protocol may be instructed to update its keying material, either by the application directly or by the record protocol sending a key expiration event. Protocols: TLS, DTLS, QUIC + TLS, MinimalT, tcpcrypt, IKEv2
- o Pre-Shared Key Export
The handshake protocol will generate one or more keys to be used for record encryption/decryption and authentication. These may be explicitly exportable to the application, traditionally limited to direct export to the record protocol, or inherently non-exportable because the keys must be used directly in conjunction with the record protocol.
 - * Explicit export: TLS (for QUIC), tcpcrypt, IKEv2, DTLS (for SRTP)
 - * Direct export: TLS, DTLS, MinimalT
 - * Non-exportable: CurveCP

5.3. Record Interfaces

Record interfaces are the points of interaction between a record protocol and the application, handshake protocol, and transport once in use.

- o Pre-Shared Key Import
Either the handshake protocol or the application directly can supply pre-shared keys for the record protocol use for encryption/decryption and authentication. If the application can supply keys directly, this is considered explicit import; if the handshake protocol traditionally provides the keys directly, it is considered direct import; if the keys can only be shared by the handshake, they are considered non-importable.

- * Explicit import: QUIC, ESP
- * Direct import: TLS, DTLS, MinimalT, tcpcrypt, WireGuard
- * Non-importable: CurveCP
- o Encrypt application data

The application can send data to the record protocol to encrypt it into a format that can be sent on the underlying transport. The encryption step may require that the application data is treated as a stream or as datagrams, and that the transport to send the encrypted records present a stream or datagram interface.

 - * Stream-to-Stream Protocols: TLS, tcpcrypt
 - * Datagram-to-Datagram Protocols: DTLS, ESP, SRTP, WireGuard
 - * Stream-to-Datagram Protocols: QUIC ((Editor's Note: This depends on the interface QUIC exposes to applications.))
- o Decrypt application data

The application can receive data from its transport to be decrypted using record protocol. The decryption step may require that the incoming transport data is presented as a stream or as datagrams, and that the resulting application data is a stream or datagrams.

 - * Stream-to-Stream Protocols: TLS, tcpcrypt
 - * Datagram-to-Datagram Protocols: DTLS, ESP, SRTP, WireGuard
 - * Datagram-to-Stream Protocols: QUIC ((Editor's Note: This depends on the interface QUIC exposes to applications.))
- o Key Expiration

The record protocol can signal that its keys are expiring due to reaching a time-based deadline, or a use-based deadline (number of bytes that have been encrypted with the key). This interaction is often limited to signaling between the record layer and the handshake layer.
Protocols: ESP ((Editor's note: One may consider TLS/DTLS to also have this interface))
- o Transport mobility

The record protocol can be signaled that it is being migrated to another transport or interface due to connection mobility, which may reset address and state validation.
Protocols: QUIC, MinimalT, CurveCP, ESP, WireGuard (roaming)

6. IANA Considerations

This document has no request to IANA.

7. Security Considerations

This document summarizes existing transport security protocols and their interfaces. It does not propose changes to or recommend usage of reference protocols.

8. Acknowledgments

The authors would like to thank Mirja Kuehlewind, Brian Trammell, Yannick Sierra, Frederic Jacobs, and Bob Bradley for their input and feedback on earlier versions of this draft.

9. Normative References

[BLAKE2] "BLAKE2 -- simpler, smaller, fast as MD5", n.d..

[Curve25519] "Curve25519 - new Diffie-Hellman speed records", n.d..

[CurveCP] "CurveCP -- Usable security for the Internet", n.d..

[I-D.ietf-quic-tls]
Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", draft-ietf-quic-tls-10 (work in progress), March 2018.

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-10 (work in progress), March 2018.

[I-D.ietf-rtcweb-security-arch]
Rescorla, E., "WebRTC Security Architecture", draft-ietf-rtcweb-security-arch-13 (work in progress), October 2017.

[I-D.ietf-tcpinc-tcpcrypt]
Bittau, A., Giffin, D., Handley, M., Mazieres, D., Slack, Q., and E. Smith, "Cryptographic protection of TCP Streams (tcpcrypt)", draft-ietf-tcpinc-tcpcrypt-11 (work in progress), November 2017.

- [I-D.ietf-tcpinc-tcpno]
- Bittau, A., Giffin, D., Handley, M., Mazieres, D., and E. Smith, "TCP-ENO: Encryption Negotiation Option", draft-ietf-tcpinc-tcpno-18 (work in progress), November 2017.
- [I-D.ietf-tls-dtls-connection-id]
- Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", draft-ietf-tls-dtls-connection-id-00 (work in progress), December 2017.
- [I-D.ietf-tls-dtls13]
- Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", draft-ietf-tls-dtls13-26 (work in progress), March 2018.
- [I-D.ietf-tls-tls13]
- Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-26 (work in progress), March 2018.
- [MinimalT]
- "MinimalT -- Minimal-latency Networking Through Better Security", n.d..
- [Noise]
- "The Noise Protocol Framework", n.d..
- [RFC2385]
- Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, DOI 10.17487/RFC2385, August 1998, <<https://www.rfc-editor.org/info/rfc2385>>.
- [RFC2508]
- Casner, S. and V. Jacobson, "Compressing IP/UDP/RTP Headers for Low-Speed Serial Links", RFC 2508, DOI 10.17487/RFC2508, February 1999, <<https://www.rfc-editor.org/info/rfc2508>>.
- [RFC3261]
- Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <<https://www.rfc-editor.org/info/rfc3261>>.
- [RFC3545]
- Koren, T., Casner, S., Geevarghese, J., Thompson, B., and P. Ruddy, "Enhanced Compressed RTP (CRTP) for Links with High Delay, Packet Loss and Reordering", RFC 3545, DOI 10.17487/RFC3545, July 2003, <<https://www.rfc-editor.org/info/rfc3545>>.

- [RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <<https://www.rfc-editor.org/info/rfc3711>>.
- [RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC 3948, DOI 10.17487/RFC3948, January 2005, <<https://www.rfc-editor.org/info/rfc3948>>.
- [RFC4302] Kent, S., "IP Authentication Header", RFC 4302, DOI 10.17487/RFC4302, December 2005, <<https://www.rfc-editor.org/info/rfc4302>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <<https://www.rfc-editor.org/info/rfc4303>>.
- [RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, DOI 10.17487/RFC4555, June 2006, <<https://www.rfc-editor.org/info/rfc4555>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <<https://www.rfc-editor.org/info/rfc5246>>.
- [RFC5723] Sheffer, Y. and H. Tschofenig, "Internet Key Exchange Protocol Version 2 (IKEv2) Session Resumption", RFC 5723, DOI 10.17487/RFC5723, January 2010, <<https://www.rfc-editor.org/info/rfc5723>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <<https://www.rfc-editor.org/info/rfc5763>>.
- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <<https://www.rfc-editor.org/info/rfc5764>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.

- [RFC5925] Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", RFC 5925, DOI 10.17487/RFC5925, June 2010, <<https://www.rfc-editor.org/info/rfc5925>>.
- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <<https://www.rfc-editor.org/info/rfc6066>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <<https://www.rfc-editor.org/info/rfc6347>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7250, DOI 10.17487/RFC7250, June 2014, <<https://www.rfc-editor.org/info/rfc7250>>.
- [RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <<https://www.rfc-editor.org/info/rfc7296>>.
- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<https://www.rfc-editor.org/info/rfc7301>>.
- [RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <<https://www.rfc-editor.org/info/rfc7539>>.
- [RFC8095] Fairhurst, G., Ed., Trammell, B., Ed., and M. Kuehlewind, Ed., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, DOI 10.17487/RFC8095, March 2017, <<https://www.rfc-editor.org/info/rfc8095>>.
- [RFC8229] Pauly, T., Touati, S., and R. Mantha, "TCP Encapsulation of IKE and IPsec Packets", RFC 8229, DOI 10.17487/RFC8229, August 2017, <<https://www.rfc-editor.org/info/rfc8229>>.
- [SIGMA] "SIGMA -- The 'SIGn-and-MAC' Approach to Authenticated Diffie-Hellman and Its Use in the IKE-Protocols", n.d..

[WireGuard]

"WireGuard -- Next Generation Kernel Network Tunnel",
n.d..

Authors' Addresses

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com

Colin Perkins
University of Glasgow
School of Computing Science
Glasgow G12 8QQ
United Kingdom

Email: csp@csp Perkins.org

Kyle Rose
Akamai Technologies, Inc.
150 Broadway
Cambridge, MA 02144
United States of America

Email: krose@krose.org

Christopher A. Wood
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: cawood@apple.com

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: December 29, 2017

P. Tiesel
T. Enhardt
Berlin Institute of Technology
June 27, 2017

Communication Units Granularity Considerations for Multi-Path Aware
Transport Selection
draft-tiesel-taps-communitgrany-00

Abstract

This document provides an abstract framework to reason about the composition of multi-path aware systems in a protocol-independent fashion. It discusses basic mechanisms that are used in multi-path systems and their applicability to different granularities of communication units. This document is targeted as consideration basis for automation of destination, path and transport protocol selection within the transport layer.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 29, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Conventions and Definitions	2
2. Introduction	2
2.1. Communication Units vs. Layering	3
3. Abstract Hierarchy of Communication Units	4
3.1. Object	4
3.2. Stream	4
3.3. Association, Flow	5
3.4. Association Set, Flow Set (Flow-Group)	5
4. Mechanisms Used in Multi-Path Systems	5
4.1. Destination Selection	5
4.2. Path Selection	6
4.3. Chunking	7
4.4. Scheduling	7
4.5. Transport Protocol Stack Instance Selection	8
5. Cost of Transport Option Selection	8
6. Involvement of On-Path Elements	8
7. Security Considerations	9
8. IANA Considerations	9
9. References	9
9.1. Normative References	9
9.2. Informative References	9
Authors' Addresses	10

1. Conventions and Definitions

The words "MUST", "MUST NOT", "SHALL", "SHALL NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

2. Introduction

Today's Internet architecture faces a communication endpoint with a set of choices, including choosing a transport protocol and picking an IP protocol version. In many cases, e.g., when fetching data from a CDN, an endpoint has also the choice of which endpoint instance, [I-D.paully-taps-guidelines] calls these instances "Derived Endpoint", to contact as DNS can return multiple alternative addresses.

If endpoints want to take advantage of multiple available paths, there is another bunch of, partially interdependent, choices:

- o Which path(s) between the endpoints could be used?

- o Which path(s) between the endpoints should be used?
- o Should the paths be used in an active/active way or only as active/fallback?
- o Which protocols or sets of protocols should be used?
- o Which role will other on-path elements, e.g. middle-boxes, take in servicing this flow?

Implementing an heuristic or strategy for choosing from this overwhelming set of transport options by each application puts a huge burden on the application developer. Thus, the decisions regarding all transport options mentioned so far should be supported and, if requested by the application, automated within a the transport layer. In order to build such automatization, we need to be able to compare the product of all transport options (destinations, paths, transport protocols and protocol options) available to choose the most appropriate.

As the protocols to be used are not known a priori and can differ depending on other transport options, this reasoning has to be independent of a specific protocol or implementation and allow to compare them even if they operate on different communication unit granularities.

2.1. Communication Units vs. Layering

When reasoning about network systems, layering traditionally has been the main guidance on where functionality is placed. Looking at modern systems, the classical concept of layers and their mapping to protocols becomes blurry.

In this document, we do not want to take a protocol-centric perspective, but we focus on mechanisms a multi-access system is composed of and the communication units they operate on. This has several advantages:

- o We can much easier abstract from the protocols used and look at the composition itself.
- o By disseminate on which kind of communication unit these mechanisms can operate, we can reason about the overall design space.
- o If seeing the same mechanism multiple times within the same system composition, we can reason about possibly conflicting optimizations.

Overall, this perspective allows us to compare mechanism like distributing requests of an application among different paths, MPTCP and using bandwidth aggregation proxies (as discussed within the IETF in the BANANA working group) despite their different nature and layer of implementation.

3. Abstract Hierarchy of Communication Units

These communication units definitions are primarily used for reasoning about automatic stack composition. Therefore, depending on the protocol stack instance, a communication unit can span multiple protocol instances.

Some of these hierarchy levels correspond to objects in [I-D.gjessing-taps-minset], but in case of Association and Association Set, we have to split categories as they may indeed be separate on the transport. Note the naming confusion concerning the term "flow" deriving from different perspective.

We also annotate the corresponding terminology used in [I-D.trammell-taps-post-sockets] if applicable.

3.1. Object

An Object is a piece of data that has a meaning for the application. It is the smallest communication unit that we consider.

[I-D.gjessing-taps-minset] correspondent: Message

[I-D.trammell-taps-post-sockets] correspondent: Message

Examples:

- o A HTTP-Request/Response-Header/Body for HTTP/2
- o An XML message in XMPP

3.2. Stream

A Stream is an ordered sequence of related Objects that should be treated the same by the transport system.

[I-D.gjessing-taps-minset] correspondent: Flow

[I-D.trammell-taps-post-sockets] correspondent: Stream

Examples:

- o A Stream in QUIC or SCTP
- o A TCP connection used as transport for XMPP

3.3. Association, Flow

An Association multiplexes a set of Objects or Streams within the same Flow with common source and destination. Therefore these communication units become indistinguishable for the network. Association and flow describe the same concept, the former from the perspective of the application, the latter from the perspective of the network.

[I-D.gjessing-taps-minset] correspondent: Flow-Group

[I-D.trammell-taps-post-sockets] correspondent: Association

Examples:

- o A TCP connection carrying HTTP/2 frames
- o A set of IP packets that carry TCP or UDP segments and share the same 5-tuple of src-address, dst-address, protocol, src-port, dest-port.

3.4. Association Set, Flow Set (Flow-Group)

An Association Set or Flow Set is a set of Associations or Flows that belong together from an application point of view.

[I-D.gjessing-taps-minset] correspondent: Flow-Group

[I-D.trammell-taps-post-sockets] correspondent: Association

Examples:

- o Two flows, one carrying RTP payloads and one used for RTCP control messages.

4. Mechanisms Used in Multi-Path Systems

4.1. Destination Selection

Destination Selection refers to selecting one of multiple different destinations. This mechanism is applicable to any kind of communication unit and can occur on all layers.

Typical cases for destination selection include:

- o Choosing one address of a multi-homed server for an upcoming communication.
- o Choosing a server among a list of servers returned by DNS, e.g for servers that host the same content as part of a CDN.
- o Choosing a backend server within a load balancer.

In practice, destination address selection is often tied to name resolution. As name resolution relies on both local decisions on the endpoint as well as decisions within the DNS infrastructure, this mechanism spreads across different administrative domains which each independently contribute to the overall selection result.

4.2. Path Selection

Path Selection refers to choosing which of the available paths to use. and can occur on the network layer and any layer below.

- o Within an end-host, path selection is usually realized by choosing the source IP address and thus choosing one of the local network interfaces for the communication to the remote endpoint.
- o Within a path layer traffic system like an MPTCP-Proxy or a BANANA-Box, path selection is usually realized by choosing the outer source and destination address.
- o In case of an ECMP router, path selection is usually done based on a 3- or 5-tuple and just determines the interface to the next hop.
- o Within MPTCP, each TCP segment has to be assigned to one or more subflows for transmission to the receiver.

While path selection involves a choice of access network it does not need knowledge of or changes to the routing choices within the core network.

When doing path selection on small communication units like TCP segments, it is not uncommon to split path selection into two subproblems: Candidate Path Selection determines feasible and preferred choices, e.g., in case of MPTCP by establishing subflows. Afterwards, Per-Chunk Path Selection selects among these alternatives for each chunk. Thus, the first can be more expensive while the latter should be easy to execute.

TODO: Discuss difference between Multiple Provisioning Domains [RFC7556] or multiple access networks within the same provisioning

domain - especially when it comes to integrating 3GPP mechanisms like IFOM/ [RFC5555].

4.3. Chunking

Chunking refers to splitting an object, a stream or a set of associations into one or more parts. Typically, chunking splits only large objects or streams into multiple ones while keeping smaller entities untouched. Associations or Flows are typically not split, but sets of Associations or Flows might be partitioned. Once split into chunks, each chunk can be transferred individually over different transfer options.

Chunking can and does occur at different layers within a system:

- o A Web site consists of multiple objects or files. Thus, the files can be seen as the natural chunks of a Web site.
- o TCP takes as input a byte stream and chunks it into segments. TCP chunking (segmentation) occurs at arbitrary byte ranges, thus it will most likely not align with boundaries of Objects that were multiplexed within an application layer Association on top of a TCP connection.

In practice, chunking is often constrained in order to maintain certain properties that are desirable for the overall system.

Examples such restrictions include the following:

- o Segmentation in TCP restrict the chunk size, i.e. TCP segment size, to the IP MTU or IP Path MTU to avoid fragmentation at the IP layer.
- o Equal cost multipath routing does not distribute packets, but Flows to avoid reordering.

4.4. Scheduling

Scheduling refers to distributing chunks or sets of chunks across multiple pre-chosen path. Thus, depending on the objectives, it can make sense to see scheduling as is nothing else than per-chunk path selection as defined above. In other cases, e.g. when trying to balance traffic, it makes sense to look at scheduling as a concept itself that uses chunking and per-chunk path selection as sub-mechanisms.

Examples of scheduling strategies include:

- o Schedule all chunks on one path as long as this path is available, otherwise fall back to another.
- o Distribute chunks based on path capacity.

4.5. Transport Protocol Stack Instance Selection

TODO - There are many examples in TAPS - still unsure what will go here or will be cited here.

5. Cost of Transport Option Selection

Transport option selection mechanisms are often intertwined. Which mechanism is used by which layer or which network component depends on the transfer objectives as well as the state of the network, e.g., availability, path throughput, path RTT, server load.

The cost and complexity of transport option selection depends on the network state used and the number of transfer options. If the transfer option selection only uses local state e.g., link availability, and the mechanism is predetermined and/or uses simple mechanisms, e.g., a simple hash function, the cost can even be negligible. An example where transfer option selection is cheap is ECMP within a router. In other cases, the cost can be non-trivial, e.g. when the selection involves queries to remote entities or even active network performance measurements. Such examples include DNS or DHT lookups, as used by some file sharing protocols, or network measurements like RTT and bandwidth estimations used by many video streaming applications. Indeed, costs may be prohibitive, e.g when requiring multiple DNS lookups for every 1 second chunk of a 20 minute video.

6. Involvement of On-Path Elements

It may become necessary to take path layer components (middle-boxes) into account that interfere with the transport layer.

While the classical "End-To-End Arguments in System Design" [End-To-End] advocates for a dumb network and placing functionality as close to the edge and up in the stack as possible, there are always tussles of moving functionality up or down the stack. This document does not argue against pushing some multi-path functionality down the stack, but advocates to maintain the control of the overall system composition at the end host.

Especially in the 3GPP context, a lot of off-loading mechanisms have been specified that are implemented as path level components, within virtual network adapters.

7. Security Considerations

Security related transport service request must take priority over performance, therefore, transport options or stack compositions that don't provide the transport service requested should be ignored for transport option selection.

Note: This discussion is not exhaustive - more considerations will be added in later versions of this draft.

8. IANA Considerations

None

9. References

9.1. Normative References

[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>>.

9.2. Informative References

[End-To-End]

Saltzer, J., Reed, D., and D. Clark, "End-to-end arguments in system design", ACM Transactions on Computer Systems Vol. 2, pp. 277-288, DOI 10.1145/357401.357402, November 1984.

[I-D.gjessing-taps-minset]

Gjessing, S. and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems", draft-gjessing-taps-minset-05 (work in progress), June 2017.

[I-D.pauly-taps-guidelines]

Pauly, T., "Software Guidelines for Protocol Evolution", draft-pauly-taps-guidelines-00 (work in progress), February 2017.

[I-D.trammell-taps-post-sockets]

Trammell, B., Perkins, C., Pauly, T., and M. Kuehlewind, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-00 (work in progress), March 2017.

- [RFC5555] Soliman, H., Ed., "Mobile IPv6 Support for Dual Stack Hosts and Routers", RFC 5555, DOI 10.17487/RFC5555, June 2009, <<http://www.rfc-editor.org/info/rfc5555>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<http://www.rfc-editor.org/info/rfc7556>>.

Authors' Addresses

Philipp S. Tiesel
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enghardt
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: May 7, 2019

P. Tiesel
T. Enghardt
TU Berlin
November 03, 2018

Communication Units Granularity Considerations for Multi-Path Aware
Transport Selection
draft-tiesel-taps-communitgrany-03

Abstract

This document provides an approach how to reason about the composition of multi-path aware transport stacks. It discusses how to compose the functionality needed by stacking existing internet protocols and the fundamental mechanisms that are used in multi-path systems and the consequences of applying them to different granularities of communication units, e.g, on a message or stream granularity. This document is targeted as guidance for automation of destination selection, path selection, and transport protocol selection.

Status of This Memo

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

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

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

This Internet-Draft will expire on May 7, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

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

Table of Contents

1. Introduction	2
1.1. Communication Units vs. Layering	3
2. Abstract Hierarchy of Communication Units	4
2.1. Message	4
2.2. Stream	4
2.3. Association / Connection Group	5
2.4. Association Set / Flow-Group	5
3. Mechanisms Used in Multi-Path Systems	5
3.1. Destination Selection	6
3.2. Path Selection	6
3.3. Chunking	7
3.4. Scheduling	7
4. Cost of Transport Option Selection	8
5. Involvement of On-Path Elements	8
6. Overview of Mechanisms provided by selected IETF Protocols	9
7. Acknowledgements	11
8. Informative References	11
Appendix A. Changes	12
A.1. Since -00	12
A.2. Since -01	12
A.3. Since -02	12
Authors' Addresses	12

1. Introduction

Today's Internet architecture faces a communication endpoint with a set of choices, including choosing a transport protocol and picking an IP protocol version. In many cases, e.g., when fetching data from a CDN, an endpoint has also the choice of which endpoint instance, [I-D.brunstrom-taps-impl] calls these instances "Derived Endpoint", to contact as DNS can return multiple alternative addresses.

If endpoints want to take advantage of multiple available paths, there is another bunch of, partially interdependent, choices:

- o Which path(s) between the endpoints could be used?
- o Which path(s) between the endpoints should be used?

- o Should the paths be used in an active/active way or only as active/fallback?
- o Which protocols or sets of protocols should be used?
- o Which role will other on-path elements, e.g. middle-boxes, take in servicing this flow?

Implementing an heuristic or strategy for choosing from this overwhelming set of transport options by each application puts a huge burden on the application developer. Thus, the decisions regarding all transport options mentioned so far should be supported and, if requested by the application, automated within a the transport layer. In order to build such automatization, we need to be able to compare the product of all transport options (destinations, paths, transport protocols and protocol options) available to choose the most appropriate.

As the protocols to be used are not known a priori and can differ depending on other transport options, this reasoning has to be independent of a specific protocol or implementation and allow to compare them even if they operate on different communication unit granularities.

1.1. Communication Units vs. Layering

When reasoning about network systems, layering traditionally has been the main guidance on where functionality is placed. Looking at modern systems, the classical concept of layers and their mapping to protocols becomes blurry. Protocols can operate on different granularities of communication units, i.e., the semantic units such as messages that the protocols distinguish. These communication units often do not match the PDUs used by the protocols, e.g., TCP segments do not necessarily align with messages at the application layer.

In this document, we do not want to take a protocol-centric perspective, but we focus on mechanisms a multi-access system is composed of and the communication units they operate on. This has several advantages:

- o We can much easier abstract from the protocols used and look at the composition itself.
- o By disseminate on which kind of communication unit these mechanisms can operate, we can reason about the overall design space.

- o If seeing the same mechanism multiple times within the same system composition, we can reason about possibly conflicting optimizations.

Overall, this perspective allows us to compare mechanism like distributing requests of an application among different paths, MPTCP and using bandwidth aggregation proxies (as discussed within the IETF in the BANANA working group) despite their different nature and layer of implementation.

2. Abstract Hierarchy of Communication Units

These communication units definitions are primarily used for reasoning about automatic stack composition. Therefore, depending on the protocol stack instance, a communication unit can span multiple protocol instances.

Some of these hierarchy levels correspond to objects in [I-D.ietf-taps-minset], but in case of Association and Association Set, we have to split categories as they may indeed be separate on the transport. Note the naming confusion concerning the term "flow" deriving from different perspective.

We also annotate the corresponding terminology used in [I-D.ietf-taps-arch] if it differs from the one used in this document.

2.1. Message

An Message is a piece of data that has a meaning for the application. It is the smallest communication unit that we consider.

[I-D.ietf-taps-minset] correspondent: Message

Examples:

- o A HTTP-Request/Response-Header/Body for HTTP/2
- o An XML message in XMPP

2.2. Stream

A Stream is an ordered sequence of related Messages that should be treated the same by the transport system.

[I-D.ietf-taps-minset] correspondent: Flow

Examples:

- o A Stream in QUIC or SCTP
- o A TCP connection used as transport for XMPP

2.3. Association / Connection Group

An Association multiplexes a set of Messages or Streams within the same Flow with common source and destination. Therefore these communication units become indistinguishable for the network. Association and flow describe the same concept, the former from the perspective of the application, the latter from the perspective of the network.

[I-D.ietf-taps-minset] correspondent: Flow-Group

[I-D.ietf-taps-arch] correspondent: Connection Group

Examples:

- o A TCP connection carrying HTTP/2 frames
- o A set of IP packets that carry TCP or UDP segments and share the same 5-tuple of src-address, dst-address, protocol, src-port, dest-port.

2.4. Association Set / Flow-Group

An Association Set or Flow Set is a set of Associations or Flows that belong together from an application point of view.

[I-D.ietf-taps-minset] correspondent: Flow-Group

Examples:

- o Two flows, one carrying RTP payloads and one used for RTCP control messages.

3. Mechanisms Used in Multi-Path Systems

Transport protocols on the Internet provide a large variety of functionality. While the functionality of simple protocols like UDP is easy to describe (multiplexing streams of messages), describing the functionality of complex protocols such as QUIC, MPTCP or SCTP is manyfold as these protocols provide a set of commonly used functionality. Also, the same functionality can be provided at many places throughout the whole stack. In the following, we explore the set of functionality that can be provided by transport protocols.

3.1. Destination Selection

Destination Selection refers to selecting one of multiple different destinations. This mechanism is applicable to any kind of communication unit and can occur on all layers.

Typical cases for destination selection include:

- o Choosing one address of a multi-homed server for an upcoming communication.
- o Choosing a server among a list of servers returned by DNS, e.g for servers that host the same content as part of a CDN.
- o Choosing a backend server within a load balancer.

In practice, destination address selection is often tied to name resolution. As name resolution relies on both local decisions on the endpoint as well as decisions within the DNS infrastructure, this mechanism spreads across different administrative domains which each independently contribute to the overall selection result.

3.2. Path Selection

Path Selection refers to choosing which of the available paths to use. and can occur on the network layer and any layer below.

- o Within an end-host, path selection is usually realized by choosing the source IP address and thus choosing one of the local network interfaces for the communication to the remote endpoint.
- o Within a path layer traffic system like an MPTCP-Proxy or a BANANA-Box, path selection is usually realized by choosing the outer source and destination address.
- o In case of an ECMP router, path selection is usually done based on a 3- or 5-tupel and just determines the interface to the next hop.
- o Within MPTCP, each TCP segment has to be assigned to one or more subflows for transmission to the receiver.

While path selection involves a choice of access network it does not need knowledge of or changes to the routing choices within the core network.

When doing path selection on small communication units like TCP segments, it is not uncommon to split path selection into two subproblems: `_Candidate Path Selection_` determines feasible and

preferred choices, e.g., in case of MPTCP by establishing subflows. Afterwards, Per-Chunk Path Selection selects among these alternatives for each chunk. Thus, the first can be more expensive while the latter should be easy to execute.

TODO: Discuss difference between Multiple Provisioning Domains [RFC7556] or multiple access networks within the same provisioning domain – especially when it comes to integrating 3GPP mechanisms like [RFC5555] or [RFC7864].

3.3. Chunking

Chunking refers to splitting an message, a stream or a set of associations into one or more parts. Typically, chunking splits only large messages or streams into multiple ones while keeping smaller entities untouched. Associations or Flows are typically not split, but sets of Associations or Flows might be partitioned. Once split into chunks, each chunk can be transferred individually over different transfer options.

Chunking can and does occur at different layers within a system:

- o A Web site consists of multiple objects or files. Thus, the files can be seen as the natural chunks of a Web site.
- o TCP takes as input a byte stream and chunks it into segments. TCP chunking (segmentation) occurs at arbitrary byte ranges, thus it will most likely not align with boundaries of Messages that were multiplexed within an application layer Association on top of a TCP connection.

In practice, chunking is often constrained in order to maintain certain properties that are desirable for the overall system. Examples such restrictions include the following:

- o Segmentation in TCP restrict the chunk size, i.e. TCP segment size, to the IP MTU or IP Path MTU to avoid fragmentation at the IP layer.
- o Equal cost multipath routing does not distribute packets, but Flows to avoid reordering.

3.4. Scheduling

Scheduling refers to distributing chunks or sets of chunks across multiple pre-chosen path. Thus, depending on the objectives, it can make sense to see scheduling as is nothing else than per-chunk path selection as defined above. In other cases, e.g. when trying to

balance traffic, it makes sense to look at scheduling as a concept itself that uses chunking and per-chunk path selection as sub-mechanisms.

Examples of scheduling strategies include:

- o Schedule all chunks on one path as long as this path is available, otherwise fall back to another.
- o Distribute chunks based on path capacity.

4. Cost of Transport Option Selection

Transport option selection mechanisms are often intertwined. Which mechanism is used by which layer or which network component depends on the transfer objectives as well as the state of the network, e.g., availability, path throughput, path RTT, server load.

The cost and complexity of transport option selection depends on the network state used and the number of transfer options. If the transfer option selection only uses local state e.g., link availability, and the mechanism is predetermined and/or uses simple mechanisms, e.g., a simple hash function, the cost can even be negligible. An example where transfer option selection is cheap is ECMP within a router. In other cases, the cost can be non-trivial, e.g. when the selection involves queries to remote entities or even active network performance measurements. Such examples include DNS or DHT lookups, as used by some file sharing protocols, or network measurements like RTT and bandwidth estimations used by many video streaming applications. Indeed, costs may be prohibitive, e.g. when requiring multiple DNS lookups for every 1 second chunk of a 20 minute video.

5. Involvement of On-Path Elements

It may become necessary to take path layer components (middle-boxes) into account that interfere with the transport layer.

While the classical "End-To-End Arguments in System Design" [End-To-End] advocates for a dumb network and placing functionality as close to the edge and up in the stack as possible, there are always tussles of moving functionality up or down the stack. This document does not argue against pushing some multi-path functionality down the stack, but advocates to maintain the control of the overall system composition at the end host. Functionality provided by a path can indeed be a reason to choose this path for a given communication unit.

Some flow off-loading mechanisms that come in gestalt of of logical interfaces, e.g., [RFC7847]. These interfaces treat some association sets differently, which can be considered on-path functionality.

6. Overview of Mechanisms provided by selected IETF Protocols

Pro toc ol	Con ges tio n C ont rol	Ord er ing	Reli abil ity	Inte grit y P.	Confid ential ity P.	Authe ntici ty P.	Chunk ing	Multiple xing
HTT P	r	r	r				bytes	requests
HTT PS	r	r	r	r	r	r	bytes	requests
XMP P	r	r	r	(r)	(r)	(r)		messages
SIP			+	(r)	(r)	(r)		messages
DTL S				+	+	+		services ,name
TLS		r	r	+	+	+		services ,name
RTP	+(p rf)	+(p rf)					messa ges(p rf)	messages
SRT P	+(p rf)	+(p rf)		+	+	r(sig)	messa ges(p rf)	messages
QUI C	+	+	+	+	+	+(tls)	bytes	connecti on-id,+(tls)
UDP								ports
DCC P	+							ports

TCP	+	+	+				bytes	ports
MPTCP	+	+	+				bytes	ports
SCTP	+	+	+				bytes	ports, streams
IPsec (ESP)				+	+	r(ike)		spi, next-header
IPsec (AH)				+		r(ike)		spi, next-header
IP		(+ (fr))					(fragments)	address, next-header
NEMO/IoM					r	r	assoc.	

Legend:

r: Protocol requires transport service.

+: Protocol provides transport service.

prf: Realized by content specific profiles.

tls: Uses TLSv1.3 as sub-protocol; imports authenticity protection and multiplexing from TLS.

ike: Realized externally by external protocol IKE/IKEv2.

sig: Realized externally by external signaling protocol (e.g., SIP, XMPP, WebRTC).

fr: :Only when fragmentation is used and only to re-assemble IP PUDs

7. Acknowledgements

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

8. Informative References

[End-To-End]

Saltzer, J., Reed, D., and D. Clark, "End-to-end arguments in system design", ACM Transactions on Computer Systems Vol. 2, pp. 277-288, DOI 10.1145/357401.357402, November 1984.

[I-D.brunstrom-taps-impl]

Brunstrom, A., Pauly, T., Enghardt, T., Grinnemo, K., Jones, T., Tiesel, P., Perkins, C., and M. Welzl, "Implementing Interfaces to Transport Services", draft-brunstrom-taps-impl-00 (work in progress), March 2018.

[I-D.ietf-taps-arch]

Pauly, T., Trammell, B., Brunstrom, A., Fairhurst, G., Perkins, C., Tiesel, P., and C. Wood, "An Architecture for Transport Services", draft-ietf-taps-arch-02 (work in progress), October 2018.

[I-D.ietf-taps-interface]

Trammell, B., Welzl, M., Enghardt, T., Fairhurst, G., Kuehlewind, M., Perkins, C., Tiesel, P., and C. Wood, "An Abstract Application Layer Interface to Transport Services", draft-ietf-taps-interface-02 (work in progress), October 2018.

[I-D.ietf-taps-minset]

Welzl, M. and S. Gjessing, "A Minimal Set of Transport Services for End Systems", draft-ietf-taps-minset-11 (work in progress), September 2018.

[RFC5555] Soliman, H., Ed., "Mobile IPv6 Support for Dual Stack Hosts and Routers", RFC 5555, DOI 10.17487/RFC5555, June 2009, <<https://www.rfc-editor.org/info/rfc5555>>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

- [RFC7847] Melia, T., Ed. and S. Gundavelli, Ed., "Logical-Interface Support for IP Hosts with Multi-Access Support", RFC 7847, DOI 10.17487/RFC7847, May 2016, <<https://www.rfc-editor.org/info/rfc7847>>.
- [RFC7864] Bernardos, CJ., Ed., "Proxy Mobile IPv6 Extensions to Support Flow Mobility", RFC 7864, DOI 10.17487/RFC7864, May 2016, <<https://www.rfc-editor.org/info/rfc7864>>.

Appendix A. Changes

A.1. Since -00

- o Replaced granularity "Object" with "Message" to align with other TAPS documents.
- o Removed empty section on protocol instance selection - this topic will go into a separate document later.
- o Minor clarifications.
- o Removed definition of normative terms not needed for this document
- o Added acknowledgments and updated authors' affiliation (compliance).

A.2. Since -01

- o Updated drafts references
- o Added Overview of Mechanisms provided by selected IETF Protocols
- o Minor clarifications
- o Removed superfluous IANA and Security Considerations section

A.3. Since -02

- o Prevent expiry (minor formatting fixes)

Authors' Addresses

Philipp S. Tiesel
TU Berlin
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enhardt
TU Berlin
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Experimental
Expires: December 17, 2017

P. Tiesel
T. Enhardt
Berlin Institute of Technology
June 15, 2017

Socket Intents
draft-tiesel-taps-socketintents-00

Abstract

This document outlines an API-independent concept that allows applications to share their knowledge about upcoming communication and express their performance preferences in a portable and abstract way: Socket Intents. Socket Intents express what an application knows, assumes, expects or wants to prioritize regarding its own network communication. The information provided by Socket Intents should be taken into account by the network stack in a best-effort way.

Socket Intent can be used to stem against the complexity and make use of multiple provisioning domains as well as new transport protocols and features available to a larger user base by expressing the applications intents in an abstract and portable way.

Status of This Memo

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

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

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

This Internet-Draft will expire on December 17, 2017.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Conventions and Definitions	2
2. Introduction	3
3. Problem Statement	3
4. General Concept	4
4.1. Socket Intent Types	4
4.2. Interactions between Socket Intents and QoS	5
5. Initial Socket Intent Types	5
5.1. Traffic Category	5
5.2. Object Size to be Sent / Received	6
5.3. Duration	6
5.4. Stream Bitrate Sent / Received	6
5.5. Burstiness	6
5.6. Timeliness	7
5.7. Application Resilience	8
5.8. Cost Preferences	8
6. Usage examples	9
6.1. Example 1	9
6.2. Example 2	9
6.3. Example 3	10
7. Implementation Guidelines	10
8. Security Considerations	11
8.1. Performance Degradation Attacks	11
8.2. Information Leakage	11
9. IANA Considerations	11
10. Publications History	11
11. References	12
11.1. Normative References	12
11.2. Informative References	12
Authors' Addresses	13

1. Conventions and Definitions

The words "MUST", "MUST NOT", "SHALL", "SHALL NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

Flow, Association, Stream, or Object are used as defined in [I-D.tiesel-taps-communitgrany]:

2. Introduction

Despite recent advances in the transport area, the adaption of new transport protocols and transport protocol features is slow and only happens in limited domains (primarily in the Web browser and within datacenters). The same problem occurs for taking advantage of multiple available access networks or provisioning domains (PvDs). In both cases, the benefits of the new transport diversity comes at the cost of an increased complexity that has to be mastered by the application programmer.

Enabling features like TCP fast open [RFC7413] or controlling how MPTCP [RFC6824] creates subflows requires specialized APIs that are not part of the standard socket API, often require deep knowledge of the transport protocol internals, and are not portable across different implementations.

Applications that want to use multiple network interfaces usually have to use their own heuristics to select which access network to use. Choosing the right interface is difficult as their characteristics differ, e.g. in terms of performance, and obtaining the necessary information is often not easy since it may require special privileges and differs heavily by implementation.

In all cases mentioned above, an application that wants to take advantage of the available transport diversity is faced with substantially higher complexity regarding network APIs and networking code.

3. Problem Statement

Application programmers opening a communication channel typically know how this channel will be used. Beside the hard requirements already necessary for establishing the communication channels, e.g., reliable in-order stream transport, there is more information available: An application developer has an intuition about optimization preferences, e.g., optimize for bandwidth, latency, or cost, about expectations, e.g. towards data loss, and possibly also about specifics, such as how many bytes will be sent or received.

This information does not directly map to the choice of a transport protocol, to certain protocol parameters, nor to which PvD to use, but the information can imply that the application can benefit from certain transport options or help to choose between multiple PvD as

described in [RFC7556], Section 6.2, and therefore enable the OS to adjust its defaults for this communication channel accordingly.

The preferences, expectations and other information known about the upcoming communication MAY be expressible in an intuitive, abstract way independent of the network- and transport protocol. Its representation SHOULD be independent of the actual API used for network communication, e.g., these SHOULD be expressible in whatever API available, e.g., as "socketopts" for BSD sockets or as part of the address resolution configuration for Post Sockets [I-D.trammell-taps-post-sockets]. Finally, given the expectations and external constraints known, the OS SHOULD use the information provided via Socket Intents in an best-effort fashion and therefore try to choose the best transport protocol, default parameters and PVDs available and MAY try to further optimize based on them.

4. General Concept

With Socket Intents, applications MAY express their communication preferences in order to take advantage of the available transfer diversity. Depending on the API used, Socket Intents can be used on a per Flow, Association, Stream, or Object level. Communication preference refers to desired transport characteristics, e.g., low delay or high throughput, stable transport or minimal cost, and is optional information.

4.1. Socket Intent Types

The following sections contain a list of Socket Intent types and their possible values.

Socket Intents are structured as key / value pair. The key is expressed by a short name, the value has a fixed data type (Enum, Int or Float).

The namespace for the short names is portioned as follows: - Experimental Socket Intent type MUST start with "x-". - Private or vendor specific Socket Intent type MUST start with "y-[vendor]-". - The remaining Socket Intent type namespace SHOULD be managed by an IANA registry. The assignment of new types requires an RFC or expert review.

For Enum data types, a list of valid values MUST be provided by the document specifying that intent.

An implementation faced with unknown intent types or invalid or unknown values MAY ignore that Intent but SHOULD return an error code to the application.

4.2. Interactions between Socket Intents and QoS

Socket Intents are not QoS labels, but have an orthogonal meaning.

- o Socket Intents SHALL be purely advisory.
- o Socket Intents MUST NOT be used to derive IntServ / RSVP style guarantees.
- o Socket Intents SHOULD be taken into account on a best-effort basis and MAY be used to derive DiffServ Service Classes as described in [RFC4594].

5. Initial Socket Intent Types

Note: Recommended default values for Enum types are marked with an asterisk (*) behind the level name.

5.1. Traffic Category

The Traffic Category describes the dominating traffic pattern of the respective communication unit expected by the application.

Short name: category

Applicability: Flow, Association, Stream

Data type: Enum

Level	Description
query	Single request / response style workload, latency bound
control	Long lasting low bandwidth control channel, not bandwidth bound
stream	Stream of bytes/objects with steady data rate
bulk	Bulk transfer of large objects, presumably bandwidth bound
mixed*	Don't know or none of the above

Note: Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the bulk category

suggesting the use of the Object Size intents or the stream category suggesting the Stream Bitrate and Duration intents.

5.2. Object Size to be Sent / Received

This Intent is used to communicate the expected size of a transfer.

Short name: `sndobjsz / rcvobjsz`

Applicability: Flow, Association, Stream, Object

Data type: Int (bytes)

5.3. Duration

This Intent is used to communicate the expected lifetime of the respective communication unit.

Short name: `duration`

Applicability: Flow, Association, Stream

Data type: Int (msec)

5.4. Stream Bitrate Sent / Received

This Intent is used to communicate the bitrate of the respective communication unit.

Short name: `sndrate / rcvrate`

Applicability: Flow, Association, Stream

Data type: Int (bytes/sec)

5.5. Burstiness

This Intent describes the anticipated sender-side burst characteristics of the traffic for this communication unit. It expresses how the traffic sent by the application is expected to vary over time, and, consequently, how long sequences of consecutively sent packets will be. Note that the actual burst characteristics of the traffic at the receiver side will depend on the network.

This Intent can provide hints to the application on what the resource usage pattern for this communication unit will look like, which can be useful for balancing the requirements of different application.

Short name: burst

Applicability: Association, Connection, Stream

Data type: Enum

Level	Description
no_bursts	Application sends traffic at a constant rate
regular_bursts	Application sends bursts of traffic periodically
random_bursts	Application sends bursts of traffic irregularly
bulk	Application sends a bulk of traffic
mixed*	Don't know or none of the above

5.6. Timeliness

This Intent describes the desired delay characteristics for this communication unit. It provides hints for the OS whether to optimize for low delay or for other criteria. There are no hard requirements or implied guarantees on whether these requirements can actually be satisfied.

Short name: timeliness

Applicability: Association, Connection, Stream, Object

Data type: Enum

Level	Description
stream	Delay and packet delay variation should be kept as low as possible
interactive	Delay should be kept as low as possible, but some variation is tolerable
transfer*	Delay and packet delay variation should be reasonable, but are not critical
background	Delay and packet delay variation is no concern

5.7. Application Resilience

This Intent describes how an application deals with disruption of its communication, e.g. connection loss. It communicates how well the application can recover from such disturbance and can have implications on how many resources the OS should allocate to failover techniques for this particular communication unit.

Short name: resilience

Applicability: Association, Connection, Stream, Object

Data type: Enum

Level	Description
sensitive*	Disruptions result in application failure, disrupting user experience
recoverable	Disruptions are inconvenient for the application, but can be recovered from
resilient	Disruptions have minimal impact for the application

5.8. Cost Preferences

This describes the Intents of an Application towards costs caused by the respective communication unit. It should guide the OS how to handle cost vs. performance and reliability tradeoffs.

Short name: cost

Applicability: Association, Connection, Stream, Object

Data type: Enum

Level	Description
no_expense	Avoid expensive transports and consider failing otherwise
optimize_cost	Prefer inexpensive transports and accept service degradation
balance_cost*	Use system policy to balance cost and other criteria
ignore_cost	Ignore cost, choose transport solely based on other criteria

6. Usage examples

6.1. Example 1

Consider a cellphone performing an OS upgrade. This process usually implies downloading a large file. This is a bulk transfer for which the application may already know the file size. Timing is typically noncritical and the data can be downloaded as background traffic with minimal cost and power overhead. It would not hurt if the TCP connection was closed during the transfer as the download can be continued.

For this case, the application should set the "Traffic Category" to "bulk", "Timeliness" to "background", and "Application Resilience" to "resilient". In addition, "Object Size to be Received" can be provided. Finally, the application may set the the "Cost Preferences" to "no_expense".

The OS can use this information and therefore may schedule this transfer on a flaky but not traffic-billed WiFi link and may reject the connection attempt if no cheap access link is available.

6.2. Example 2

Consider a user watching non-live video content using MPEG-DASH [DASH]. This usually means fetching a stream of video chunks. The application should know the size of each chunk and may know the bitrate and the duration of each chunk and the whole video. Disconnection of the TCP connection should be avoided because that might have an effect that is visible to the user.

For this case, the application should set the "Traffic Category" to "stream", the "Timeliness" to "stream", and "Application Resilience" to "sensitive". It may also provide the "Stream Bitrate Received" and "Duration" expected. Finally, the application may set the "Cost Preferences" to "balance_cost".

The OS can use this information and, e.g., use MPTCP [RFC6824] if available to schedule the traffic on the cheaper link (e.g., WiFi) while establishing an additional subflow over an expensive link (e.g., LTE). If the desired bandwidth cannot be matched by the cheaper link, the more expensive link can be added to satisfy the desired bandwidth.

If the application would set the "Cost Preferences" to "optimize_cost", the OS would not schedule traffic on the second subflow and the application would reduce the video quality to adapt to the available data rate.

6.3. Example 3

Consider a user managing a remote machine via SSH. This usually involves at least one long-lived console session and possibly file transfers using SCP or rsync multiplexed on the same association (e.g. TCP connection).

For the console session, the application can set the "Traffic Category" to "control", the "Burstiness" to "random bursts", the timeliness to "interactive" and the resilience to "sensitive".

For the file transfers, SSH may set both, the "Traffic Category" and "Burstiness" to "bulk". It may also know the size of the transfer and therefore sets "Object Size to be Sent" or "Object Size to be Received".

Assuming there are transport opportunities supporting multiple streams in a single association (e.g. SCPT [RFC4960]), the OS can use this information to schedule the streams over different links to meet their requirements (latency vs. bandwidth). In case the OS has to use TCP, it can still optimize by disabling TCP Nagle Algorithm for console session related transmissions.

7. Implementation Guidelines

TBD

8. Security Considerations

8.1. Performance Degradation Attacks

We assume that applications specify their preferences in a selfish, but not malicious way and that it is up to the OS to find a compromise between demands.

A malicious application could confuse the OS in a way that leads to scheduling traffic with certain Intents on amore expensive interface, penalizing this traffic, or even rejecting it. The attack vector added by this is negligible: As the malicious application could also generate the traffic it claims to intent, it already has a much more powerful attack vector.

As a mitigation, the OS could monitor and compare the intents specified with the traffic actually generated and notify the user if the usage of Socket Intents is unusual or defective.

8.2. Information Leakage

Varying the transport or IP layer parameters of packets belonging to different Streams or Objects multiplexed in the same encrypted association might enable an attacker to gain some ground truth about the shares of different kinds of traffic. As this might also be implied by packet timings, application developers might weight the small additional information disclosure against the possible performance gains. Using Socket Intents on Association level can be considered safe.

9. IANA Considerations

The Socket Intents type namespace SHOULD be managed by the IANA registry. Details conforming to [RFC5226] are laid out in Section 4.1, the initial types for the registry are described in Section 5.

10. Publications History

- o The original idea of Socket Intents was published in [CoNEXT2013].
- o A performance study "Socket Intents: OS Support for Using Multiple Access Networks and its Benefits for Web Browsing" is under submission.

11. References

11.1. Normative References

- [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>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, DOI 10.17487/RFC5226, May 2008, <<http://www.rfc-editor.org/info/rfc5226>>.

11.2. Informative References

- [CoNEXT2013] Schmidt, P., Enghardt, T., Khalili, R., and A. Feldmann, "Socket intents", Proceedings of the ninth ACM conference on Emerging networking experiments and technologies - CoNEXT '13, DOI 10.1145/2535372.2535405, 2013.
- [DASH] International Organization for Standardization, "Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats", Standard ISO/IEC 23009-1:2014, June 2011, <<https://www.iso.org/standard/65274.html>>.
- [I-D.pauly-taps-guidelines] Pauly, T., "Software Guidelines for Protocol Evolution", draft-pauly-taps-guidelines-00 (work in progress), February 2017.
- [I-D.tiesel-taps-communitgrany] Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for using Transport Diversity or Multiple Provisioning Domains", draft-tiesel-taps-communitgrany-00 (work in progress), July 2017.
- [I-D.trammell-taps-post-sockets] Trammell, B., Perkins, C., Pauly, T., and M. Kuehlewind, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-00 (work in progress), March 2017.

- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <<http://www.rfc-editor.org/info/rfc4594>>.
- [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<http://www.rfc-editor.org/info/rfc4960>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<http://www.rfc-editor.org/info/rfc6824>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<http://www.rfc-editor.org/info/rfc7556>>.

Authors' Addresses

Philipp S. Tiesel
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enghardt
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Experimental
Expires: April 30, 2018

P. Tiesel
T. Enhardt
A. Feldmann
TU Berlin
October 27, 2017

Socket Intents
draft-tiesel-taps-socketintents-01

Abstract

This document outlines Socket Intents, a concept that allows applications to share their knowledge about upcoming communication and express their performance preferences in a generic, intuitive and, portable way. Using Socket Intents, an application can express what it knows, assumes, expects, or wants regarding its network communication. The information provided by Socket Intents can be used by the network stack to optimize communication in a best-effort way.

Socket Intent can be used to stem against the complexity of exploiting transport diversity, e.g., to automate the choice among multiple paths, provisioning domains or protocols. By shifting this complexity from the application developer to the operating system, it enables the use of these transport features to a wider range of applications.

Status of This Memo

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

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

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

This Internet-Draft will expire on April 30, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Conventions and Definitions	3
2. Introduction	3
3. Problem Statement	3
4. Socket Intents Concept	4
4.1. Interactions between Socket Intents and QoS	5
5. Socket Intent Types	5
6. Initial Socket Intent Types	6
6.1. Traffic Category	6
6.2. Size to be Sent / Received	7
6.3. Duration	7
6.4. Stream Bitrate Sent / Received	7
6.5. Burstiness	7
6.6. Timeliness	8
6.7. Disruption Resilience	9
6.8. Cost Preferences	9
7. Implementation Guidelines	10
8. Security Considerations	10
8.1. Performance Degradation Attacks	10
8.2. Information Leakage	11
9. IANA Considerations	11
10. Publications History	11
11. Acknowledgements	11
12. References	11
12.1. Normative References	11
12.2. Informative References	12
Appendix A. Usage examples	13
A.1. Example 1	13
A.2. Example 2	13
A.3. Example 3	14
Appendix B. Changes	14
B.1. Since -00	14

Authors' Addresses	15
--------------------	----

1. Conventions and Definitions

The words "MUST", "MUST NOT", "SHALL", "SHALL NOT", "SHOULD", and "MAY" are used in this document. It's not shouting; when these words are capitalized, they have a special meaning as defined in [RFC2119].

Association Set, Association, Stream, or Message are used as defined in [I-D.tiesel-taps-communitgrany].

2. Introduction

Despite recent advances in the transport area, the adaption of new transport protocols and transport protocol features is slow. In practice, this only happens in limited fields as Web browsers or within datacenters. The same problem occurs for taking advantage of paths or provisioning domains (PvDs). In both cases, the benefits of the new transport diversity come at the cost of an increased complexity that has to be mastered by the application programmer.

To enable transport features like TCP fast open [RFC7413] or to control how MPTCP [RFC6824] creates subflows requires specialized APIs. These APIs are not part of the standard socket API, usually not portable, and not available in many programming languages. Using them often requires profound knowledge of the transport protocol internals.

To use multiple paths, applications usually have to use their own heuristics to select which paths, provisioning domains, or access network to use. Choosing the right path is difficult as their characteristics differ, e.g., regarding performance. Obtaining the necessary information is difficult since it may require special privileges and non-portable APIs.

In all cases mentioned above, an application that wants to take advantage of the available transport diversity is faced with substantially higher complexity regarding network APIs and networking code.

3. Problem Statement

Application programmers opening a communication channel typically know how this channel will be used. There is more information available than the protocol and destination address needed to establish a communication channel: An application developer has an intuition about many aspects of an upcoming communication. These intuition may include:

preferences: whether to optimize for bandwidth, latency, or cost

characteristics: expected packet rates, byte rates or how many bytes will be sent or received.

expectations: towards path availability or packet loss

resiliences: whether the application can gracefully handle certain error cases

These preferences, expectations and other information known about the upcoming communication should be expressible in an intuitive, generic way, that is independent of the network and transport protocol. Its representation should be independent of the actual API used for network communication and should be expressible in whatever API available, e.g., as socket options for BSD sockets or as part of the address resolution configuration for Post Sockets [I-D.trammell-taps-post-sockets].

Socket Intents should enable the OS to adjust the communication channel according to the application's intents in a best-effort fashion: They should provide the information needed to automatically enabling transport features the application can benefit from or help choosing the most suitable (combination) of paths based on the properties of the access networks or PvD (see [RFC7556], Section 6.2) available. The actual implementation is not part of the Socket Intents concept, it is left to an OS policy that may choose the best transport protocol, default parameters and PvDs available and may also try to further optimize wherever possible.

4. Socket Intents Concept

Socket Intents are pieces of information that allow an application to express what they know about the application's communication. They indicate what the application wants to achieve, knows, or assumes in general, intuitive terms. An application can use them to annotate the characteristics, preferences, and intentions it associates with each communication unit. Depending on the API used, Socket Intents can be used on a per Association Set, Association, Stream or, Message level.

Socket Intents are optional information that can be considered in a best-effort manner. Socket Intents do not include requirements, such as reliable in-order delivery. Typical examples include desired transport characteristics, e.g., low delay, high throughput, or minimal cost, as well as expected application behavior, e.g., will send 500 bytes. As this information captures the intents of an

applications and passes them along with the communication socket, we call these pieces of information Socket Intents.

Applications have an incentive to specify their intents as accurately as possible to take advantage of the most suitable existing resources. Applications are expected to selfishly specify their preferences. It is up to the OS's policy to prevent commitment of excessive resources.

4.1. Interactions between Socket Intents and QoS

Socket Intents are not QoS labels, but have an orthogonal meaning. While the purpose of QoS is to specify what an application requires, Socket Intents are used to specify what an application knows or prefers. Therefore,

- o Socket Intents SHALL be purely advisory.
- o Socket Intents MUST NOT be used to derive IntServ / RSVP style guarantees.
- o Socket Intents SHOULD be taken into account on a best-effort basis and MAY be used to derive DiffServ Service Classes as described in [RFC4594].

5. Socket Intent Types

Socket Intents are structured as key-value-pairs.

The key, called short name, specifies the Socket Intent type. It is identified by a string of the lower-case characters [a-z], numbers [0-9] and the separator "-".

The namespace for the short names is partitioned as follows:

- o All Socket Intent type not starting with "x-" or "y-" are managed by an IANA registry. The assignment of new types requires an RFC or expert review (TO BE DECIDED).
- o Socket Intent type starting with "x-" are for experimental use.
- o Private or vendor specific Socket Intent type MUST start with "y-[vendor]-".

Values can be represented as Enum, Int, Float, ASCII-String [RFC0020] or a sequence of the aforementioned data types. Implementations determine how these types are represented on the respective platform.

The data type for the individual Socket Intents are determined by the document defining the Socket Intent and MUST NOT be changed by an implementation. For Enum data types, a list of valid values MUST be provided by the document specifying that intent as well as a default value that is equivalent to not specifying this intent.

6. Initial Socket Intent Types

The following sections contain a list of Socket Intent types and their possible values. Recommended default values for Enum values are marked with an asterisk (*) behind the level name.

6.1. Traffic Category

The Traffic Category describes the dominating traffic pattern of the respective communication unit expected by the application.

Short name: category

Applicability: Association Set, Association, Stream

Data type: Enum

Level	Description
query	Single request / response style workload, latency bound
control	Long lasting low bandwidth control channel, not bandwidth bound
stream	Stream of bytes/messages with steady data rate
bulk	Bulk transfer of large messages, presumably bandwidth bound
mixed*	Don't know or none of the above

Note: Most categories suggest the use of other intents to further describe the traffic pattern anticipated, e.g., the bulk category suggesting the use of the Size to be Sent intent or the stream category suggesting the Stream Bitrate and Duration intents.

6.2. Size to be Sent / Received

This Intent is used to communicate the expected size of a transfer.

Short name: `send_size / recv_size`

Applicability: Association Set, Association, Stream, Message

Data type: Int (bytes)

6.3. Duration

This Intent is used to communicate the expected lifetime of the respective communication unit.

Short name: `duration`

Applicability: Association Set, Association, Stream

Data type: Int (msec)

6.4. Stream Bitrate Sent / Received

This Intent is used to communicate the bitrate of the respective communication unit.

Short name: `send_bitrate / recv_bitrate`

Applicability: Association Set, Association, Stream

Data type: Int (bits/sec)

6.5. Burstiness

This Intent describes the anticipated burst characteristics of the traffic for this communication unit. It expresses how the traffic sent by the application is expected to vary over time, and, consequently, how long sequences of consecutively sent packets will be. Note that the actual burst characteristics of the traffic at the receiver side will depend on the network.

This Intent can provide hints to the application on what the resource usage pattern for this communication unit will look like, which can be useful for balancing the requirements of different application.

Short name: `bursts`

Applicability: Association Set, Association, Stream

Data type: Enum

Level	Description
no_bursts	Application sends traffic at a constant rate
regular_bursts	Application sends bursts of traffic periodically
random_bursts	Application sends bursts of traffic irregularly
bulk	Application sends a bulk of traffic
mixed*	Don't know or none of the above

6.6. Timeliness

This Intent describes the desired delay characteristics for this communication unit. It provides hints for the OS whether to optimize for low delay or for other criteria. There are no hard requirements or implied guarantees on whether these requirements can actually be satisfied.

Short name: timeliness

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
stream	Delay and packet delay variation should be kept as low as possible
interactive	Delay should be kept as low as possible, but some variation is tolerable
transfer*	Delay and packet delay variation should be reasonable, but are not critical
background	Delay and packet delay variation is no concern

6.7. Disruption Resilience

This Intent describes how an application deals with disruption of its communication, e.g. connection loss. It communicates how well the application can recover from such disturbance and can have implications on how many resources the OS should allocate to failover techniques for this particular communication unit.

Short name: resilience

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
sensitive	Disruptions result in application failure, disrupting user experience
recoverable*	Disruptions are inconvenient for the application, but can be recovered from
resilient	Disruptions have minimal impact for the application

6.8. Cost Preferences

This describes the Intents of an Application towards costs caused by the respective communication unit. It should guide the OS how to handle cost vs. performance and reliability tradeoffs.

Short name: cost

Applicability: Association Set, Association, Stream, Message

Data type: Enum

Level	Description
no_expense	Avoid expensive transports and consider failing otherwise
optimize_cost	Prefer inexpensive transports and accept service degradation
balance_cost*	Do not bias balancing cost and other criteria
ignore_cost	Ignore cost, choose transport solely based on other criteria

Note: the "no_expense" level implicitly asks the OS to fail communication attempts if no inexpensive transports are available.

Application developers MUST be aware that this also no hard requirement and can be ignored or overridden by the OS policy.

7. Implementation Guidelines

Implementations faced with unknown Socket Intent types SHOULD ignore these intents for forward compatibility. The API MAY include a parameter to change this behavior and make specifying unknown Socket Intent types return an error.

Invalid values SHOULD return an error to the application.

For debugging purposes, implementations SHOULD allow to enumerate the Socket Intents that are understood by the implementation. They MAY expose which of the Socket Intents were considered by the implementation.

8. Security Considerations

8.1. Performance Degradation Attacks

We assume that applications specify their preferences in a selfish, but not malicious way and that it is up to the OS to find a compromise between demands.

A malicious application could confuse the OS in a way that leads to scheduling traffic with certain Intents on a more expensive interface, penalizing this traffic, or even rejecting it. The attack vector added by this is negligible: As the malicious application

could also generate the traffic it claims to intend, it already has a much more powerful attack vector.

As a mitigation, the OS could monitor and compare the intents specified with the traffic actually generated and notify the user if the usage of Socket Intents is unusual or defective.

8.2. Information Leakage

Varying the transport or IP layer parameters of packets belonging to different Streams or Messages multiplexed in the same encrypted association might enable an attacker to gain some ground truth about the shares of different kinds of traffic. As this might also be implied by packet timings, application developers might weight the small additional information disclosure against the possible performance gains. Using Socket Intents on Association level can be considered safe.

9. IANA Considerations

The Socket Intents type namespace SHOULD be managed by the IANA registry. Details conforming to [RFC5226] are laid out in Section 5, the initial types for the registry are described in Section 6.

10. Publications History

- o The original idea of Socket Intents was published in [CoNEXT2013].
- o A performance study "Socket Intents: OS Support for Using Multiple Access Networks and its Benefits for Web Browsing" is under submission.

11. Acknowledgements

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

12. References

12.1. Normative References

- [RFC0020] Cerf, V., "ASCII format for network interchange", STD 80, RFC 20, DOI 10.17487/RFC0020, October 1969, <<https://www.rfc-editor.org/info/rfc20>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", RFC 5226, DOI 10.17487/RFC5226, May 2008, <<https://www.rfc-editor.org/info/rfc5226>>.

12.2. Informative References

- [CoNEXT2013] Schmidt, P., Enghardt, T., Khalili, R., and A. Feldmann, "Socket intents", Proceedings of the ninth ACM conference on Emerging networking experiments and technologies - CoNEXT '13, DOI 10.1145/2535372.2535405, 2013.
- [DASH] International Organization for Standardization, "Dynamic adaptive streaming over HTTP (DASH) - Part 1: Media presentation description and segment formats", Standard ISO/IEC 23009-1:2014, June 2011, <<https://www.iso.org/standard/65274.html>>.
- [I-D.pauly-taps-guidelines] Pauly, T., "Guidelines for Racing During Connection Establishment", draft-pauly-taps-guidelines-01 (work in progress), October 2017.
- [I-D.tiesel-taps-communitgrany] Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for Multi-Path Aware Transport Selection", draft-tiesel-taps-communitgrany-01 (work in progress), October 2017.
- [I-D.trammell-taps-post-sockets] Trammell, B., Perkins, C., Pauly, T., Kuehlewind, M., and C. Wood, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-03 (work in progress), October 2017.
- [RFC4594] Babiarz, J., Chan, K., and F. Baker, "Configuration Guidelines for DiffServ Service Classes", RFC 4594, DOI 10.17487/RFC4594, August 2006, <<https://www.rfc-editor.org/info/rfc4594>>.

- [RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <<https://www.rfc-editor.org/info/rfc4960>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<https://www.rfc-editor.org/info/rfc6824>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

Appendix A. Usage examples

A.1. Example 1

Consider a cellphone performing an OS upgrade. This process usually implies downloading a large file. This is a bulk transfer for which the application may already know the file size. Timing is typically noncritical and the data can be downloaded as background traffic with minimal cost and power overhead. It would not hurt if the TCP connection was closed during the transfer as the download can be continued.

For this case, the application should set the "Traffic Category" to "bulk", "Timeliness" to "background", and "Application Resilience" to "resilient". In addition, "Message Size to be Received" can be provided. Finally, the application may set the the "Cost Preferences" to "no_expense".

The OS can use this information and therefore may schedule this transfer on a flaky but not traffic-billed WiFi link and may reject the connection attempt if no cheap access link is available.

A.2. Example 2

Consider a user watching non-live video content using MPEG-DASH [DASH]. This usually means fetching a stream of video chunks. The application should know the size of each chunk and may know the bitrate and the duration of each chunk and the whole video. Disconnection of the TCP connection should be avoided because that might have an effect that is visible to the user.

For this case, the application should set the "Traffic Category" to "stream", the "Timeliness" to "stream", and "Application Resilience" to "sensitive". It may also provide the "Stream Bitrate Received" and "Duration" expected. Finally, the application may set the "Cost Preferences" to "balance_cost".

The OS can use this information and, e.g., use MPTCP [RFC6824] if available to schedule the traffic on the cheaper link (e.g., WiFi) while establishing an additional subflow over an expensive link (e.g., LTE). If the desired bandwidth cannot be matched by the cheaper link, the more expensive link can be added to satisfy the desired bandwidth.

If the application would set the "Cost Preferences" to "optimize_cost", the OS would not schedule traffic on the second subflow and the application would reduce the video quality to adapt to the available data rate.

A.3. Example 3

Consider a user managing a remote machine via SSH. This usually involves at least one long-lived console session and possibly file transfers using SCP or rsync multiplexed on the same association (e.g. TCP connection).

For the packets sent for the console session, the application can set the "Traffic Category" to "control", the "Burstiness" to "random bursts", the timeliness to "interactive" and the resilience to "sensitive". For the packets of the file transfers, SSH may set both, the "Traffic Category" and "Burstiness" to "bulk". It may also know the size of the transfer and therefore sets "Message Size to be Sent" or "Message Size to be Received".

Assuming there are transport opportunities supporting multiple streams in a single association (e.g. SCPT [RFC4960]), the OS can use this information to schedule the streams over different links to meet their requirements (latency vs. bandwidth). In case the OS has to use TCP, it can still optimize by disabling TCP Nagle Algorithm for console session related transmissions.

Appendix B. Changes

B.1. Since -00

- o Updates on Terminology (Object -> Message, Flow -> Association)
- o More detailed Socket Intent Types specification

- o Added implementation guidelines
- o Many clarifications
- o Fixed Authors and affiliations

Authors' Addresses

Philipp S. Tiesel
TU Berlin
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enhardt
TU Berlin
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

Anja Feldmann
TU Berlin
Marchstr. 23
Berlin
Germany

Email: anja@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: January 4, 2018

P. Tiesel
T. Enhardt
Berlin Institute of Technology
July 03, 2017

A Socket Intents Prototype for the BSD Socket API - Experiences, Lessons
Learned and Considerations
draft-tiesel-taps-socketintents-bsdsockets-00

Abstract

This document describes a prototype implementation of Socket Intents [I-D.tiesel-taps-socketintents] for the BSD Socket API as an illustrative example how Socket Intents could be implemented. It described the experiences made with the prototype and lessons learned from trying to extend the BSD Socket API.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 4, 2018.

Copyright Notice

Copyright (c) 2017 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Prototype Architecture	3
3. Multiple Access Manager	4
3.1. Policy	5
3.2. Path characteristics data collectors	6
4. Socket Intents Representation	7
5. The Socket Intents API Variants	7
5.1. Classic API / muacc_context	8
5.1.1. muacc_getaddrinfo()	8
5.1.2. muacc_socket()	9
5.1.3. muacc_setsockopt()	10
5.1.4. muacc_connect()	10
5.1.5. muacc_close()	11
5.2. Classic API / getaddrinfo	11
5.3. Socketconnect API	14
6. API Implementation Experiences & Lessons Learned	15
6.1. The Missing Link to Name Resolution	15
6.2. File Descriptors Considered Harmful	16
6.3. Asynchronous API Anarchy	17
6.4. Here Be Dragons hiding in Shadow Structures	17
7. Conclusion	18
8. Acknowledgments	18
9. References	18
9.1. Informative References	18
9.2. URIs	19
Appendix A. API Usage Examples	19
A.1. Usage Example of the Classic / muacc_context API	19
A.2. Usage Example of the Classic / getaddrinfo API	21
A.3. Usage Example of the Socketconnect API	22
Authors' Addresses	23

1. Introduction

With the proliferation of devices that have multiple paths to the internet and an increasing number of transport protocols available, the number of transport options to serve a communication unit explodes. Implementing a heuristic or strategy for choosing from this overwhelming set of transport options by each application puts a huge burden on the application developer. Thus, the decisions regarding all transport options mentioned so far should be supported and, if requested by the application, automated within the transport layer.

Socket Intents [I-D.tiesel-taps-socketintents] allow an application to express what it knows, assumes, expects or wants to prioritize regarding its own network communication. This information can then be used by the OS to perform destination selection, path selection and transport protocol stack instance selection.

Our Socket Intents prototype for the BSD Socket API is a first attempt to automate transport option selection within the OS. It is primarily targeted at path and destination address selection and tries to be as close as possible to the semantics of the BSD Socket API. The prototype mostly excludes the problem of transport protocol stack instance selection, which is more closely discussed in [I-D.tiesel-taps-communitgrany].

We implemented the prototype as a wrapper for the BSD Socket API that communicates to a central Multiple Access Manager that makes the actual decisions and can optimize across applications. The whole implementation was done in about 15k lines of C code. The code is available at Github [1] under BSD License.

This document describes our Socket Intents prototype for the BSD Socket API. It details important aspects of the implementation and the API variants we developed over time based on lessons learned. Finally, it summarizes these lessons and points out why the BSD Socket API is not particularly well suited to integrate automated transport protocol stack instance selection. Furthermore, it describes the limitations for destination address and path selection within the BSD Socket API.

2. Prototype Architecture

The Socket Intents prototype consists of the following components, also shown in Figure 1:

- o The Socket Intents API, a BSD Socket API wrapper for applications to use, including a representation of the actual Socket Intents.
- o The Socket Intents Library which implements the Socket Intents API. It sends requests to the Multiple Access Manager, e.g. before establishing a connection, and gets back a response regarding what interface to use.
- o The Multiple Access Manager (MAM), a daemon which gets informed about all application requests and has knowledge of the available network interfaces.

- o The Policy, a dynamically loaded library hosted by the MAM. It chooses which of the available interfaces to use based on the available knowledge about them and the Socket Intents.
- o Data collectors that reside inside the MAM and that provide information like bandwidth usage, smoothed RTT estimate and RSSI for wireless links to the policy.

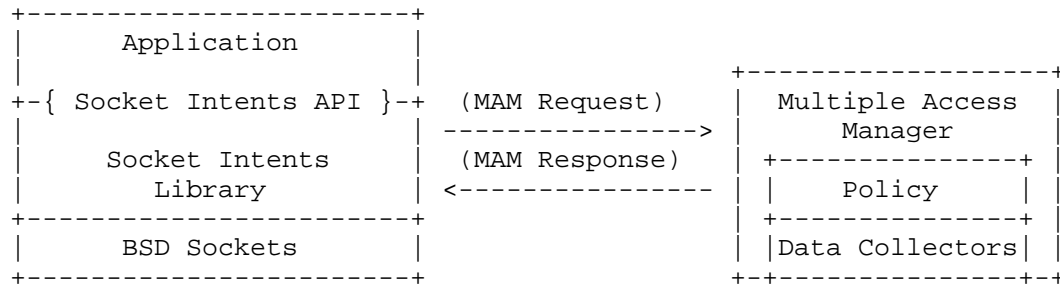


Figure 1: Components of the Socket Intents Prototype

3. Multiple Access Manager

The Multiple Access Manager (MAM) is the central transport option selection instance on a host. It is realized as a daemon that runs in userspace and receives requests from each application that uses the Socket Intents Library.

The MAM hosts the Policy, which is the actual decision making component, e.g., deciding which source address and therefore which source interface to use. Upon events, such as an application requesting to resolve a name or to connect a socket (see Section 5 for details), the Socket Intents Library issues a MAM request and the MAM invokes a callback to the policy - see Section 3.1 for details - which can either communicate its decision right away or defer its decision, e.g., when it has to wait for the results of name resolution. The results and decisions are communicated back to the Socket Intents Library through the MAM response, where they are applied to the actual socket, see also Figure 1.

To support the policy, the MAM maintains a list of IP prefixes that are configured on the local interfaces and available for outgoing communications. As destination address selection and path selection are highly dependent on each other, the MAM integrates DNS resolution and maintains separate resolver configurations per prefix (see [ANRW17-MH] for further discussion on multiple PvDs and DNS resolution). Furthermore, the MAM includes data collectors which

periodically gather statistics on the available paths, see Section 3.2 for details.

3.1. Policy

In the Socket Intents prototype, the Policy to select among the available transport options is hosted by the MAM, see Figure 1. We implement different interchangeable policies as dynamically loaded libraries. In our current implementation, only one policy can be active at a given time. When launching the MAM, the user has to choose a policy and supply a policy configuration, which can contain arbitrary data.

Examples of policy configuration include:

- o A list of IP prefixes configured on local interfaces to consider as source for the communication
- o Name server(s) to use for each of the IP prefixes
- o Preferences to instrument the policy

The policy is initialized with this configuration and then waits for the callback of an incoming MAM request.

Upon a callback, the policy can use information from the MAM request, such as Socket Intents, and information available within the MAM, such as recently measured path characteristics (see Section 3.2), to make decisions.

Policy decisions can include:

- o The source address(es) used for name resolution
- o How to order the results of name resolution (i.e., preferring certain IP addresses over others)
- o Picking an IP protocol version
- o Picking a transport protocol
- o Setting socket options (e.g., disable TCP Nagle)
- o Choosing a source address for the outgoing communication
- o Reusing a socket from a given socket set (only for the API variant described in Section 5.3)

Note that in our current implementation, the policy is a piece of code which can in principle execute arbitrary instructions. We assume this is acceptable for an experimental platform but would prefer an abstract description like a domain-specific language for a production system.

3.2. Path characteristics data collectors

The data collectors are implemented as a component of the MAM, within a callback that is executed periodically, e.g., every 100 ms. When this callback is invoked, the MAM passively gathers statistics about the current usage and properties of the available local interfaces and stores them in per-interface or per-network prefix data structures.

Measured properties include:

- o Minimum Smoothed Round Trip Time (SRTT) of current TCP connections using a network prefix, as an estimate for last-mile latency
- o Transmitted and received bytes per second over an interface within the last callback period, as an estimate for current utilization
- o Smoothed transmitted and received bytes per second over an interface, as an estimate for recent utilization
- o Maximum transmitted and received bytes per second over an interface within the last 5 minutes, as an estimate for maximum available bandwidth
- o On 802.11 interfaces, the Received Signal Strength Indicator (RSSI) of the last received frame on that interface, as an estimate for reception strength
- o On 802.11 interfaces, the modulation rate of the last received and the last transmitted unicast data frame on that interface, as an estimate for the available data transmission rate on the first hop

When a policy callback is invoked, the policy can use the latest measured properties to guide its decisions, see Section 3.1.

Note that we do not perform active measurements from within the MAM to avoid overhead.

4. Socket Intents Representation

As described in [I-D.tiesel-taps-socketintents], Socket Intents are pieces of information about upcoming traffic. An application can share the information that it has available through the Socket Intents API.

In our implementation, Socket Intents are represented as socket options for get/setsockopt on its own socket option level (SOL_INTENTS).

For some of the API variants, we had to introduce socket option lists, i.e., data structures that can hold multiple socket options and therefore multiple Socket Intents.

Which of these variants is actually used depends on the API variant, see Section 5.

5. The Socket Intents API Variants

The Socket Intents API is a wrapper around the BSD Socket API. It sends requests to the Multiple Access Manager (MAM) at certain events, e.g., before a connection is established, and applies the suggestions that it gets from the MAM, e.g., to bind to a certain local interface or to set a certain socket option.

There exist different variants of this API, see Section 5, that try to fit different concepts:

- o The Classic API with muacc_context, see Section 5.1, was attempting to stick as close as possible to the call sequence of BSD Sockets.
- o The second variant of the classic API does all transport option selection in "getaddrinfo", see Section 5.2. This variant tries to simplify the implementation without deriving too much from the usage of BSD Sockets. It minimizes the changes to the BSD Socket API, but adds additional overhead to the application.
- o The "socketconnect" API, see Section 5.3, tries to automate as much functionality as possible and adds support for automating connection caching. It replaces the usual sequence of BSD Socket API calls with a single call.

5.1. Classic API / muacc_context

In the first variant, we add a parameter called "muacc_context" to the BSD Socket API calls and to getaddrinfo. This parameter holds properties provided by the socket calls and retains them across function calls to enable automation of the connection properties by our Socket Intents Prototype. The shadow data structures behind the "muacc_context" parameter are initialized by API wrapper at the time of the first call (which we assume to be muacc_getaddrinfo most of the time) with most of its fields empty. Then within each call to our modified Socket API, it is filled with data.

Properties include:

- o Socket file descriptor
- o API calls that were already performed on this context
- o domain, type, and protocol of the socket
- o remote hostname
- o remote address
- o hints for resolving the remote address
- o local address to bind to that the application requested
- o local address to bind to that the MAM suggested
- o current socket options that were set
- o socket options suggested by MAM

5.1.1. muacc_getaddrinfo()

This function resolves a host name or service to an addrinfo data structure, usually containing an IP address or port. Internally, the Socket Intents prototype sends a "getaddrinfo" request to the MAM, which should do the name resolution. It can, e.g., resolve the name over multiple available interfaces at the same time, and then order the results according to a policy decision, or only return results obtained over a specific interface.

SIGNATURE:

```
int muacc_getaddrinfo(muacc_context_t *ctx, const char *hostname,  
const char *servname, const struct addrinfo *hints, struct addrinfo  
**res)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called with an empty context, which is then filled within the function.

hostname: Remote host name to be resolved

servname: Remote service to be resolved

hints: Hints for resolving the name

res: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by getaddrinfo().

5.1.2. muacc_socket()

This function creates a socket file descriptor just like the regular socket call.

SIGNATURE:

```
int muacc_socket(muacc_context_t *ctx, int domain, int type, int  
protocol)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after muacc_getaddrinfo(), since domain, type, and protocol can depend on the type of resolved address.

domain: Domain of the socket

type: Type of the socket

protocol: Protocol of the socket

RETURN VALUE:

Returns a file descriptor of the new socket on success, or -1 on failure.

5.1.3. `muacc_setsockopt()`

This call allows to set socket options (including Socket Intents). For Socket Intents, this function can be called on a valid "muacc_context" and an invalid file descriptor (-1) to provide assertional hints to "muacc_getaddrinfo()".

SIGNATURE:

```
int muacc_setsockopt(muacc_context_t *ctx, int socket, int level, int
option_name, const void *option_value, socklen_t option_len)
```

ARGUMENTS:

`ctx`: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called to set Intents as socket options within the context.

`socket`: Socket file descriptor

`level`: Level of the socket option to set

`option_name`: Name of the socket option to set

`option_value`: Value of the socket option to set

`option_len`: Length of the socket option to set

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.4. `muacc_connect()`

Like the regular connect call, but also binds to the source address selected by the Socket Intents Policy and applies socket options suggested by the Socket Intents Policy.

SIGNATURE:

```
int muacc_connect(muacc_context_t *ctx, int socket, const struct
sockaddr *address, socklen_t address_len)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after all Socket Intents for this connection have been set via `muacc_setsockopt()`.

socket: Socket file descriptor

address: Remote address to connect to

address_len: Length of the remote address

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.5. `muacc_close()`

Like regular close, but also cleans up state held in shadow structures behind "muacc_context"

SIGNATURE:

```
int muacc_close(muacc_context_t *ctx, int socket)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function deinitializes and releases the context.

socket: Socket file descriptor

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.2. Classic API / `getaddrinfo`

In this variant, Socket Intents are passed directly to "`getaddrinfo()`" as part of the "hints" parameter. The name resolution is done by the MAM, which makes all decisions and stores them in the "result" data structure as list of options ordered by preference. Subsequently, applications can use this information for calls to the unmodified BSD Socket API or other APIs. We provide helpers to apply all socket options from the "result" data structure.

All relevant infos are stored in our `addrinfo` struct (see Figure 2)

SIGNATURE:

```
int muacc_ai_getaddrinfo(const char * hostname, const char * service,  
const struct muacc_addrinfo * hints, struct muacc_addrinfo ** result)
```

ARGUMENTS:

hostname: Remote host name to be resolved

service: Remote service to be resolved

hints: Hints for resolving the name. Contents include family,
socket type, protocol, socket options (including Socket Intents
for this socket/connection), local address to bind to.

result: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by getaddrinfo().

```
/** Extended version of the standard library's struct addrinfo
 *
 * This is used both as hint and as result from the
 * muacc_ai_getaddrinfo * function. This structure
 * differs from struct addrinfo only in the three members
 * ai_bindaddrlen, ai_bindaddr and ai_sockopt.
 */
struct muacc_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;

    /** Not included in struct addrinfo. Purpose:
     * 1. If the structure is given to muacc_ai_getaddrinfo
     *    as hints, you set socket intents that influence MAM's
     *    source and destination as well as transport protocol
     *    selection
     * 2. The recommended socket options MAM will be returned
     *    through this attribute.
     */
    struct socketopt *ai_sockopts;

    int ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;

    /** Not included in struct addrinfo.
     * Length of ai_bindaddr.
     */
    int ai_bindaddrlen;
    /** Not included in struct addrinfo.
     * Contains the address, which the MAM recommends us to bind to.
     */
    struct sockaddr *ai_bindaddr;

    struct muacc_addrinfo *ai_next;
};
```

Figure 2: Definition of the muacc_addrinfo struct

Appendix A.2 shows an example usage of the classic API with most functionality in getaddrinfo.

5.3. Socketconnect API

In this API variant, we move the functionality of resolving a hostname and connecting to the resulting address into one function called "socketconnect()". This API makes it possible to call socketconnect not only for each connection, but also to multiplex messages across multiple existing sockets.

This function returns a file descriptor of a connected socket for the application to use. This socket can either be a newly created one or a socket that existed previously and is now being reused. Furthermore, a socket can belong to a socket set of sockets with common destination and service. These sockets may, e.g., be bound to different local addresses, but are treated as interchangeable by the API implementation. So if the application passes a socket file descriptor to this function, it may get back a different file descriptor to a socket from the same set, e.g., to use the connection over a different local interface for its following communication.

SIGNATURE:

```
int socketconnect(int *socket, const char *host, size_t hostlen,
const char *serv, size_t servlen, struct socketopt *socketopt, int
domain, int type, int proto)
```

ARGUMENTS:

socket: Existing socket file descriptor as representant to a socket set, "-1" to create a new socket, or "0" to automatically try to find a suitable socket set

host: Remote hostname to be resolved

hostlen: Length of remote hostname

serv: Remote service or port

servlen: Length of remote service

socketopt: List of socket options, including Socket Intents

domain: Domain of the socket

type: Type of the socket

proto: Protocol of the socket

RETURN VALUE:

Returns 0 on success if socket is from an existing socket set, 1 on success if socket was newly created, or -1 on fail.

Appendix A.3 shows an example usage of the Socketconnect API.

6. API Implementation Experiences & Lessons Learned

While designing and implementing the different parts of the system as described in this document, we faced several challenges. In the Multiple Access Manager discovering the currently available paths and statistics about their performance turned out to be quite complex and had to be implemented in a partially platform-dependent way. However, the most challenging parts were the Socket Intents API and Library, on which we focus in the following sections.

6.1. The Missing Link to Name Resolution

Transport option selection is most useful if crucial information, such as Socket Intents or other socket options, is available as early as possible, i.e., for name resolution. The primary problem here is the order of the function calls that are involved in name resolution, destination selection, protocol, and path selection, and how they are linked.

In the classic BSD Socket API, most functions either take a socket file descriptor as argument or return it, and thus link different function calls to the same flow. However, "getaddrinfo()" is not linked to a socket file descriptor, and it is typically called before the socket is created. At this point, it is not yet possible to set a socket option, because the socket does not exist yet.

Consequently, across BSD Socket API calls, several choices are being made before it is possible to set a Socket Intent: A call to "getaddrinfo()" returns a linked list of "addrinfo" structs, where each entry contains an "ai_family" (IP version), the pair of "ai_socktype" and "ai_protocol" (transport protocol), and a "sockaddr" struct containing an IP address and port to connect to. Then a socket of the given family, type, and protocol is created. Only after this has been done, socket options can be set on the socket, but at this point destination, IP version, and transport protocol are already fixed. Before calling "connect()", only the path to be used (i.e., the local address to bind to) can still be chosen, but the available paths and which one to prefer may be constrained by the choice of destination.

The three variants described in Section 5 work around this problem in different ways:

- o The approach in Section 5.2 places the whole automation of transport option selection into the "getaddrinfo()" function. The results are returned in an extended "addrinfo" struct and have to be applied manually by the application, including binding to a source address representing the selected path and applying all socket options provided in a list, for each connection attempt.
- o The approach in Section 5.1 adds a context to all socket- and name resolution-related API calls.
- o The approach in Section 5.3 puts all functionality into one call.

All of these approaches add the missing link between name resolution and the other parts of the API, but add a lot of state keeping either to the API, which the application developer has to manage, or to the Socket Intents library.

6.2. File Descriptors Considered Harmful

When using BSD sockets, file descriptors are the abstraction for network flows. Depending on the transport protocol used, their semantics changes and these file handles represent streams (SOCK_STREAM), associations (SOCK_DGRAM) or network interfaces (SOCK_RAW). This does not provide a unified API, but is merely an artifact of squeezing networking into the "Everything is a file" UNIX philosophy.

File descriptors make no good abstraction for automated protocol stack instance selection as applications have to adapt to changed semantics, e.g., whether message boundaries are preserved, depending on the transport protocol chosen.

File descriptors make no good abstraction for destination instance selection and path selection either. Once a socket has been created, its protocol stack instance is fixed, so selecting a path by binding to a local address and connecting to a destination instance is now only possible using this protocol stack instance. If such a connection attempt fails, it is possible to retry using another path and destination, but changing the protocol stack instance requires creating a new socket with a different file descriptor.

For further discussion of other asynchronous I/O weirdness with file descriptors see end of Section 6.3.

6.3. Asynchronous API Anarchy

Network I/O is asynchronous, but asynchronous I/O within the POSIX filesystem API is hard to use. There are at least three different asynchronous I/O APIs for each operating system.

To implement asynchronous I/O for our Socket Intents prototype, we wrapped one of the asynchronous I/O APIs that is available on most platforms: "select()". To make Socket Intents accessible to more applications and on more platforms, a production-grade system would need to wrap all asynchronous I/O APIs and implement most of the socket creation logic, path selection and connection logic within these wrappers. However, mixing asynchronous I/O and multithreading may lead to unintuitive behavior, e.g., calling our prototype's select() from different threads could lead to anything from deadlocks to busy waiting.

Another issue is that we use Unix domain sockets to communicate between our Multiple Access Manager and the Socket Intents API library called by the application, so we need to make sure that the application does not block on communication with the Multiple Access Manager.

Also the problems with using file descriptors get even worse. If a Socket API call should return immediately, it needs to provide the application with a reference to a flow that has not yet been fully set up, i.e., a reference to a "future" socket. An implementation of such an asynchronous API has to return an unconnected socket file descriptor, on which the application then calls, e.g., "select()", and starts using it once it becomes readable and writable. If the destination, path and transport protocol have not been chosen yet at this point, the file descriptor returned by the implementation might not yet have the final family and transport protocol. When the implementation later creates the final socket of the right type, it can re-bind it to the file-id of the originally returned file descriptor using "dup2". This procedure can easily lead to time-of-check to time-of-use confusion. To make things even worse, the application can copy the "future" file descriptor using "dup", which is rarely useful for sockets, but in combination with file descriptors used as "future" it leads to unexpected behavior.

6.4. Here Be Dragons hiding in Shadow Structures

The API variants described in Section 5.3 and Section 5.1 need to keep a lot of state in shadow structures that cannot be passed between the Socket API calls otherwise. This state needs to be cleaned up when the last copy of the file descriptor is closed or the

last socket held for reuse has timed out. In addition, access to these shadow structures has to be thread-safe.

Implementing both has turned out to be extremely error-prone and there is a high amount of unspecified behavior and platform-dependent extensions in the system library. These issues guarantee that an implementation of transport option selection that nicely integrates with BSD Sockets will come with lots of limitations and will not be portable across POSIX-compliant operating systems.

7. Conclusion

Adding transport option selection to BSD Sockets is hard, as the API calls are not designed to defer making and applying choices to a moment where all information needed for transport option selection is available.

After all, if limiting transport option selection to the granularity BSD Sockets typically provide today (TCP connections and UDP associations), the API variant described in Section 5.2 seems to be a good compromise, even if it forces the application to try all candidates itself (either in a sequential or partial parallel fashion). This option is easily deployable, but does not include automation of techniques like connection caching or HTTP pipelining.

The most versatile API variant described in Section 5.3 implements connection caching on the transport layer. This comes at the cost of heavily modifying existing applications. If feasible, given the unnecessary complexity of the file I/O integration of BSD sockets, it seems easier to move to a totally different system like [I-D.trammell-taps-post-sockets].

8. Acknowledgments

Thanks to Tobias Kaiser mail@tb-kaiser.de [2] for drafting and implementing the API variant described in Section 5.2 as part of his BA thesis.

9. References

9.1. Informative References

[ANRW17-MH]

Tiesel, P., May, B., and A. Feldmann, "Multi-Homed on a Single Link", Proceedings of the 2016 workshop on Applied Networking Research Workshop - ANRW 16 , DOI 10.1145/2959424.2959434, 2016.

- [I-D.tiesel-taps-communitgrany]
Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for Multi-Path Aware Transport Selection", draft-tiesel-taps-communitgrany-00 (work in progress), June 2017.
- [I-D.tiesel-taps-socketintents]
Tiesel, P. and T. Enghardt, "Socket Intents", draft-tiesel-taps-socketintents-00 (work in progress), June 2017.
- [I-D.trammell-taps-post-sockets]
Trammell, B., Perkins, C., Pauly, T., and M. Kuehlewind, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-00 (work in progress), March 2017.
- [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>>.
- [RFC6824] Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<http://www.rfc-editor.org/info/rfc6824>>.
- [RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<http://www.rfc-editor.org/info/rfc7413>>.
- [RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<http://www.rfc-editor.org/info/rfc7556>>.

9.2. URIs

- [1] <https://github.com/fg-inet/socket-intents/>
- [2] <mailto:mail@tb-kaiser.de>

Appendix A. API Usage Examples

A.1. Usage Example of the Classic / muacc_context API

In this example, a client application sets up a connection to a remote host and sends data to it. It specifies two Socket Intents on

this connection: The Category of Bulk Transfer and the File Size of 1 MB.

```
#define LENGTH_OF_DATA 1048576

// Create and initialize a context to retain information across function
// calls
muacc_context_t ctx;
muacc_init_context(&ctx);

int socket = -1;

struct addrinfo *result = NULL;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

// Set Socket Intents for this connection. Note that the "socket" is
// still invalid, but it does not yet need to exist at this time. The
// Socket Intents prototype just sets the Intent within the
// muacc_context data structure.

enum intent_category category = INTENT_BULKTRANSFER;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_CATEGORY, &category, sizeof(enum intent_category));

int filesize = LENGTH_OF_DATA;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_FILESIZE, &filesize, sizeof(int));

// Resolve a host name. This involves a request to the MAM, which can
// automatically choose a suitable local interface or other parameters
// for the DNS request and set other parameters, such as preferred
// address family or transport protocol.
muacc_getaddrinfo(&ctx, "example.org", NULL, NULL, &result);

// Create the socket with the address family, type, and protocol
// obtained by getaddrinfo.
socket = muacc_socket(&ctx, result->ai_family, result->ai_socktype,
    result->ai_protocol);

// Connect the socket to the remote endpoint as determined by
// getaddrinfo. This involves another request to MAM, which may at this
// point, e.g., choose to bind the socket to a local IP address before
// connecting it.
muacc_connect(&ctx, socket, result->ai_addr, result->ai_addrlen);
```

```
// Send data to the remote host over the socket.  
write(socket, &buf, LENGTH_OF_DATA);  
  
// Close the socket. This de-initializes any data that was stored within  
// the muacc_context.  
muacc_close(&ctx, socket);
```

A.2. Usage Example of the Classic / getaddrinfo API

As in Appendix A.1, the application sets the Intents "Category" and "File Size".

```
#define LENGTH_OF_DATA 1048576

// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

struct muacc_addrinfo intent_hints = { .ai_flags = 0,
    .ai_family = AF_INET, .ai_socktype = SOCK_STREAM, .ai_protocol = 0,
    .ai_sockopts = &intents, .ai_addr = NULL, .ai_addrlen = 0,
    .ai_bindaddr = NULL, .ai_bindaddrlen = 0, .ai_next = NULL };

struct muacc_addrinfo *result = NULL;

muacc_ai_getaddrinfo("example.org", NULL, &intent_hints,
    &result);

// Create and connect the socket, using the information obtained through
// getaddrinfo
int fd;
fd = socket(result->ai_family, result->ai_socktype,
    result->ai_protocol);
muacc_ai_simple_connect(fd, result);

// Send data to the remote host over the socket, then close it.
write(fd, &buf, LENGTH_OF_DATA);
close(fd);

muacc_ai_freeaddrinfo(result);
```

A.3. Usage Example of the Socketconnect API

As in Appendix A.1, the application sets the Intents "Category" and "File Size". As we provide "-1" as socket, no we do not reuse existing connections.

```
#define LENGTH_OF_DATA 1048576

// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

int socket = -1;

// Get a socket that is connected to the given host and service,
// with the given Intents
socketconnect(&socket, "example.org", 11, "80", 2, &intents, AF_INET,
    SOCK_STREAM, 0);

// Send data to the remote host over the socket.
write(socket, &buf, LENGTH_OF_DATA);

// Close the socket and tear down the data structure kept for it
// in the library
socketclose(socket);
```

Authors' Addresses

Philipp S. Tiesel
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enhardt
Berlin Institute of Technology
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de

TAPS Working Group
Internet-Draft
Intended status: Informational
Expires: January 3, 2019

P. Tiesel
T. Enghardt
TU Berlin
July 02, 2018

A Socket Intents Prototype for the BSD Socket API - Experiences, Lessons
Learned and Considerations
draft-tiesel-taps-socketintents-bsdsockets-02

Abstract

This document describes a prototype implementation of Socket Intents [I-D.tiesel-taps-socketintents] for the BSD Socket API as an illustrative example how Socket Intents could be implemented. It described the experiences made with the prototype and lessons learned from trying to extend the BSD Socket API.

Status of This Memo

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

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

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

This Internet-Draft will expire on January 3, 2019.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
2. Prototype Architecture	3
3. Multiple Access Manager	4
3.1. Policy	5
3.2. Path characteristics data collectors	6
4. Socket Intents Representation	7
5. The Socket Intents API Variants	7
5.1. Classic API / muacc_context	8
5.1.1. muacc_getaddrinfo()	8
5.1.2. muacc_socket()	9
5.1.3. muacc_setsockopt()	10
5.1.4. muacc_connect()	10
5.1.5. muacc_close()	11
5.2. Classic API / getaddrinfo	11
5.3. Socketconnect API	14
6. API Implementation Experiences & Lessons Learned	15
6.1. The Missing Link to Name Resolution	15
6.2. File Descriptors Considered Harmful	16
6.3. Asynchronous API Anarchy	17
6.4. Here Be Dragons hiding in Shadow Structures	17
7. Conclusion	18
8. Acknowledgments	18
9. References	18
9.1. Informative References	19
9.2. URIs	20
Appendix A. API Usage Examples	20
A.1. Usage Example of the Classic / muacc_context API	20
A.2. Usage Example of the Classic / getaddrinfo API	21
A.3. Usage Example of the Socketconnect API	22
Appendix B. Changes	23
B.1. Since -01	23
B.2. Since -00	23
Authors' Addresses	24

1. Introduction

With the proliferation of devices that have multiple paths to the internet and an increasing number of transport protocols available, the number of transport options to serve a communication unit explodes. Implementing a heuristic or strategy for choosing from this overwhelming set of transport options by each application puts a huge burden on the application developer. Thus, the decisions regarding all transport options mentioned so far should be supported

and, if requested by the application, automated within the transport layer.

Socket Intents [I-D.tiesel-taps-socketintents] allow an application to express what it knows, assumes, expects or wants to prioritize regarding its own network communication. This information can then be used by the OS to perform destination selection, path selection and transport protocol stack instance selection.

Our Socket Intents prototype for the BSD Socket API is a first attempt to automate transport option selection within the OS. It is primarily targeted at path and destination address selection and tries to be as close as possible to the semantics of the BSD Socket API. The prototype mostly excludes the problem of transport protocol stack instance selection, which is more closely discussed in [I-D.tiesel-taps-communitgrany].

We implemented the prototype as a wrapper for the BSD Socket API that communicates to a central Multiple Access Manager that makes the actual decisions and can optimize across applications. The whole implementation was done in about 15k lines of C code. The code is available at Github [1] under BSD License.

This document describes our Socket Intents prototype for the BSD Socket API. It details important aspects of the implementation and the API variants we developed over time based on lessons learned. Finally, it summarizes these lessons and points out why the BSD Socket API is not particularly well suited to integrate automated transport protocol stack instance selection. Furthermore, it describes the limitations for destination address and path selection within the BSD Socket API.

2. Prototype Architecture

The Socket Intents prototype consists of the following components, also shown in Figure 1:

- o The Socket Intents API, a BSD Socket API wrapper for applications to use, including a representation of the actual Socket Intents.
- o The Socket Intents Library which implements the Socket Intents API. It sends requests to the Multiple Access Manager, e.g. before establishing a connection, and gets back a response regarding what interface to use.
- o The Multiple Access Manager (MAM), a daemon which gets informed about all application requests and has knowledge of the available network interfaces.

- o The Policy, a dynamically loaded library hosted by the MAM. It chooses which of the available interfaces to use based on the available knowledge about them and the Socket Intents.
- o Data collectors that reside inside the MAM and that provide information like bandwidth usage, smoothed RTT estimate and RSSI for wireless links to the policy.

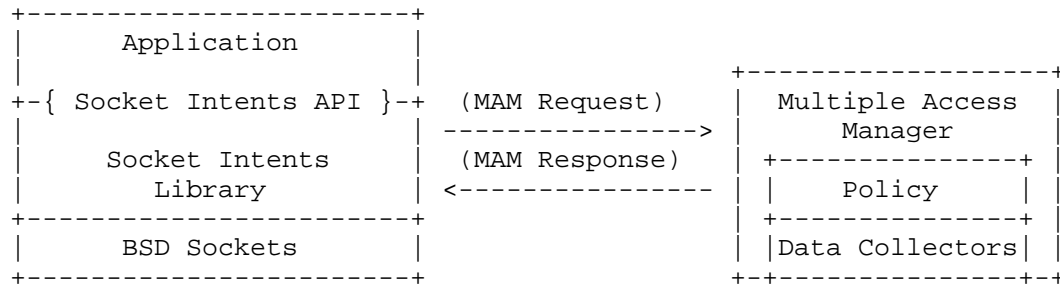


Figure 1: Components of the Socket Intents Prototype

3. Multiple Access Manager

The Multiple Access Manager (MAM) is the central transport option selection instance on a host. It is realized as a daemon that runs in userspace and receives requests from each application that uses the Socket Intents Library.

The MAM hosts the Policy, which is the actual decision making component, e.g., deciding which source address and therefore which source interface to use. Upon events, such as an application requesting to resolve a name or to connect a socket (see Section 5 for details), the Socket Intents Library issues a MAM request and the MAM invokes a callback to the policy - see Section 3.1 for details - which can either communicate its decision right away or defer its decision, e.g., when it has to wait for the results of name resolution. The results and decisions are communicated back to the Socket Intents Library through the MAM response, where they are applied to the actual socket, see also Figure 1.

To support the policy, the MAM maintains a list of IP prefixes that are configured on the local interfaces and available for outgoing communications. As destination address selection and path selection are highly dependent on each other, the MAM integrates DNS resolution and maintains separate resolver configurations per prefix (see [ANRW17-MH] for further discussion on multiple PvDs and DNS resolution). Furthermore, the MAM includes data collectors which

periodically gather statistics on the available paths, see Section 3.2 for details.

3.1. Policy

In the Socket Intents prototype, the Policy implements the decision logic for selecting among available transport options. In our current implementation, only one policy can be active at a given time. We implement different interchangeable policies as dynamically loaded libraries, which are hosted by the Multi Access Manager (MAM), see Figure 1. When launching the MAM, the user has to choose a policy and supply a policy configuration, which can contain additional information to configure the policy.

Examples of policy configuration include:

- o A list of IP prefixes configured on local interfaces to consider as source for the communication
- o Name server(s) to use for each of the IP prefixes
- o Preferences to instrument the policy, e.g., default prefix to use

The policy is initialized with this configuration and then waits for the callback of an incoming MAM request.

Upon a callback, the policy can use information from the MAM request, such as Socket Intents, and information available within the MAM, such as recently measured path characteristics (see Section 3.2), to make decisions.

Policy decisions can include:

- o The source address(es) used for name resolution
- o How to order the results of name resolution (i.e., preferring certain IP addresses over others)
- o Picking an IP protocol version
- o Picking a transport protocol (Note that in our current implementation, we are constrained by the Socket API, so our policy cannot override the transport protocol chosen by an application.)
- o Setting socket options (e.g., disable TCP Nagle)
- o Choosing a source address for the outgoing communication

- o Reusing a socket from a given socket set (only for the API variant described in Section 5.3)

Note that in our current implementation, the policy is a piece of code which can in principle execute arbitrary instructions. We assume this is acceptable for an experimental platform but would prefer an abstract description like a domain-specific language for a production system.

3.2. Path characteristics data collectors

The data collectors are implemented as a component of the MAM, within a callback that is executed periodically, e.g., every 100 ms. When this callback is invoked, the MAM passively gathers statistics about the current usage and properties of the available local interfaces and stores them in per-interface or per-network prefix data structures.

Measured properties include:

- o Minimum Smoothed Round Trip Time (SRTT) of current TCP connections using a network prefix, as an estimate for last-mile latency
- o Median SRTT of current TCP connections using a network prefix, as an alternate estimate for last-mile latency
- o Median of Round Trip Time variations within connections
- o Median variation of Smoothed Round Trip Times across connections
- o Median of percentage of segments deemed lost of all transmitted segments of current TCP connections, as an estimate of upstream packet loss
- o Maximum transmitted and received bytes per second over an interface within the last 5 minutes, as an estimate for maximum available bandwidth
- o On 802.11 interfaces, the Received Signal Strength Indicator (RSSI) of the last received frame on that interface, as an estimate for reception strength
- o On 802.11 interfaces, the modulation rate of the last received and the last transmitted unicast data frame on that interface, as an estimate for the available data transmission rate on the first hop

- o On 802.11 interfaces, the latest Channel Utilization as parsed from a Beacon frame, as an estimate of congestion on the wireless medium

See [ANRW18-Metrics] for more discussion of the gathered metrics.

When a policy callback is invoked, the policy can use the latest measured properties to guide its decisions, see Section 3.1.

Note that we do not perform active measurements from within the MAM to avoid overhead.

4. Socket Intents Representation

As described in [I-D.tiesel-taps-socketintents], Socket Intents are pieces of information about upcoming traffic. An application can share the information that it has available through the Socket Intents API.

In our implementation, Socket Intents are represented as socket options for get/setsockopt on its own socket option level (SOL_INTENTS).

For some of the API variants, we had to introduce socket option lists, i.e., data structures that can hold multiple socket options and therefore multiple Socket Intents.

Which of these variants is actually used depends on the API variant, see Section 5.

5. The Socket Intents API Variants

The Socket Intents API is a wrapper around the BSD Socket API. It sends requests to the Multiple Access Manager (MAM) at certain events, e.g., before a connection is established, and applies the suggestions that it gets from the MAM, e.g., to bind to a certain local interface or to set a certain socket option.

There exist different variants of this API, see Section 5, that try to fit different concepts:

- o The Classic API with muacc_context, see Section 5.1, was attempting to stick as close as possible to the call sequence of BSD Sockets.
- o The second variant of the classic API does all transport option selection in "getaddrinfo", see Section 5.2. This variant tries to simplify the implementation without deriving too much from the

usage of BSD Sockets. It minimizes the changes to the BSD Socket API, but adds additional overhead to the application.

- o The "socketconnect" API, see Section 5.3, tries to automate as much functionality as possible and adds support for automating connection caching. It replaces the usual sequence of BSD Socket API calls with a single call.

5.1. Classic API / muacc_context

In the first variant, we add a parameter called "muacc_context" to the BSD Socket API calls and to getaddrinfo. This parameter holds properties provided by the socket calls and retains them across function calls to enable automation of the connection properties by our Socket Intents Prototype. The shadow data structures behind the "muacc_context" parameter are initialized by API wrapper at the time of the first call (which we assume to be muacc_getaddrinfo most of the time) with most of its fields empty. Then within each call to our modified Socket API, it is filled with data.

Properties include:

- o Socket file descriptor
- o API calls that were already performed on this context
- o domain, type, and protocol of the socket
- o remote hostname
- o remote address
- o hints for resolving the remote address
- o local address to bind to that the application requested
- o local address to bind to that the MAM suggested
- o current socket options that were set
- o socket options suggested by MAM

5.1.1. muacc_getaddrinfo()

This function resolves a host name or service to an addrinfo data structure, usually containing an IP address or port. Internally, the Socket Intents prototype sends a "getaddrinfo" request to the MAM, which should do the name resolution. It can, e.g., resolve the name

over multiple available interfaces at the same time, and then order the results according to a policy decision, or only return results obtained over a specific interface.

SIGNATURE:

```
int muacc_getaddrinfo(muacc_context_t *ctx, const char *hostname,
const char *servname, const struct addrinfo *hints, struct addrinfo
**res)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called with an empty context, which is then filled within the function.

hostname: Remote host name to be resolved

servname: Remote service to be resolved

hints: Hints for resolving the name

res: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by getaddrinfo().

5.1.2. muacc_socket()

This function creates a socket file descriptor just like the regular socket call.

SIGNATURE:

```
int muacc_socket(muacc_context_t *ctx, int domain, int type, int
protocol)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after muacc_getaddrinfo(), since domain, type, and protocol can depend on the type of resolved address.

domain: Domain of the socket

type: Type of the socket

protocol: Protocol of the socket

RETURN VALUE:

Returns a file descriptor of the new socket on success, or -1 on failure.

5.1.3. muacc_setsockopt()

This call allows to set socket options (including Socket Intents). For Socket Intents, this function can be called on a valid "muacc_context" and an invalid file descriptor (-1) to provide assertional hints to "muacc_getaddrinfo()".

SIGNATURE:

```
int muacc_setsockopt(muacc_context_t *ctx, int socket, int level, int
option_name, const void *option_value, socklen_t option_len)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called to set Intents as socket options within the context.

socket: Socket file descriptor

level: Level of the socket option to set

option_name: Name of the socket option to set

option_value: Value of the socket option to set

option_len: Length of the socket option to set

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.4. muacc_connect()

Like the regular connect call, but also binds to the source address selected by the Socket Intents Policy and applies socket options suggested by the Socket Intents Policy.

SIGNATURE:

```
int muacc_connect(muacc_context_t *ctx, int socket, const struct
sockaddr *address, socklen_t address_len)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function is mostly called after all Socket Intents for this connection have been set via `muacc_setsockopt()`.

socket: Socket file descriptor

address: Remote address to connect to

address_len: Length of the remote address

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.1.5. muacc_close()

Like regular close, but also cleans up state held in shadow structures behind "muacc_context"

SIGNATURE:

```
int muacc_close(muacc_context_t *ctx, int socket)
```

ARGUMENTS:

ctx: Context that can contain properties of this socket/connection and retains them across function calls. This function deinitializes and releases the context.

socket: Socket file descriptor

RETURN VALUE:

Returns 0 on success, or -1 on failure.

5.2. Classic API / getaddrinfo

In this variant, Socket Intents are passed directly to "getaddrinfo()" as part of the "hints" parameter. The name resolution is done by the MAM, which makes all decisions and stores them in the "result" data structure as list of options ordered by preference. Subsequently, applications can use this information for

calls to the unmodified BSD Socket API or other APIs. We provide helpers to apply all socket options from the "result" data structure.

All relevant infos are stored in our `addrinfo` struct (see Figure 2)

SIGNATURE:

```
int muacc_ai_getaddrinfo(const char * hostname, const char * service,  
const struct muacc_addrinfo * hints, struct muacc_addrinfo ** result)
```

ARGUMENTS:

`hostname`: Remote host name to be resolved

`service`: Remote service to be resolved

`hints`: Hints for resolving the name. Contents include family, socket type, protocol, socket options (including Socket Intents for this socket/connection), local address to bind to.

`result`: Data structure for result of name resolution

RETURN VALUE:

Returns 0 on success, or an error code as provided by `getaddrinfo()`.

```
/** Extended version of the standard library's struct addrinfo
 *
 * This is used both as hint and as result from the
 * muacc_ai_getaddrinfo * function. This structure
 * differs from struct addrinfo only in the three members
 * ai_bindaddrlen, ai_bindaddr and ai_socketopt.
 */
struct muacc_addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;

    /** Not included in struct addrinfo. Purpose:
     * 1. If the structure is given to muacc_ai_getaddrinfo
     *    as hints, you set socket intents that influence MAM's
     *    source and destination as well as transport protocol
     *    selection
     * 2. The recommended socket options MAM will be returned
     *    through this attribute.
     */
    struct socketopt *ai_sockopts;

    int ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;

    /** Not included in struct addrinfo.
     * Length of ai_bindaddr.
     */
    int ai_bindaddrlen;
    /** Not included in struct addrinfo.
     * Contains the address, which the MAM recommends us to bind to.
     */
    struct sockaddr *ai_bindaddr;

    struct muacc_addrinfo *ai_next;
};
```

Figure 2: Definition of the muacc_addrinfo struct

Appendix A.2 shows an example usage of the classic API with most functionality in getaddrinfo.

5.3. Socketconnect API

In this API variant, we move the functionality of resolving a hostname and connecting to the resulting address into one function called "socketconnect()". This API makes it possible to call socketconnect not only for each connection, but also to multiplex messages across multiple existing sockets.

This function returns a file descriptor of a connected socket for the application to use. This socket can either be a newly created one or a socket that existed previously and is now being reused. Furthermore, a socket can belong to a socket set of sockets with common destination and service. These sockets may, e.g., be bound to different local addresses, but are treated as interchangeable by the API implementation. So if the application passes a socket file descriptor to this function, it may get back a different file descriptor to a socket from the same set, e.g., to use the connection over a different local interface for its following communication.

SIGNATURE:

```
int socketconnect(int *socket, const char *host, size_t hostlen,
const char *serv, size_t servlen, struct socketopt *socketopt, int
domain, int type, int proto)
```

ARGUMENTS:

socket: Existing socket file descriptor as representant to a socket set, "-1" to create a new socket, or "0" to automatically try to find a suitable socket set

host: Remote hostname to be resolved

hostlen: Length of remote hostname

serv: Remote service or port

servlen: Length of remote service

socketopt: List of socket options, including Socket Intents

domain: Domain of the socket

type: Type of the socket

proto: Protocol of the socket

RETURN VALUE:

Returns 0 on success if socket is from an existing socket set, 1 on success if socket was newly created, or -1 on fail.

Appendix A.3 shows an example usage of the Socketconnect API.

6. API Implementation Experiences & Lessons Learned

While designing and implementing the different parts of the system as described in this document, we faced several challenges. In the Multiple Access Manager discovering the currently available paths and statistics about their performance turned out to be quite complex and had to be implemented in a partially platform-dependent way. However, the most challenging parts were the Socket Intents API and Library, on which we focus in the following sections.

6.1. The Missing Link to Name Resolution

Transport option selection is most useful if crucial information, such as Socket Intents or other socket options, is available as early as possible, i.e., for name resolution. The primary problem here is the order of the function calls that are involved in name resolution, destination selection, protocol, and path selection, and how they are linked.

In the classic BSD Socket API, most functions either take a socket file descriptor as argument or return it, and thus link different function calls to the same flow. However, "getaddrinfo()" is not linked to a socket file descriptor, and it is typically called before the socket is created. At this point, it is not yet possible to set a socket option, because the socket does not exist yet.

Consequently, across BSD Socket API calls, several choices are being made before it is possible to set a Socket Intent: A call to "getaddrinfo()" returns a linked list of "addrinfo" structs, where each entry contains an "ai_family" (IP version), the pair of "ai_socktype" and "ai_protocol" (transport protocol), and a "sockaddr" struct containing an IP address and port to connect to. Then a socket of the given family, type, and protocol is created. Only after this has been done, socket options can be set on the socket, but at this point destination, IP version, and transport protocol are already fixed. Before calling "connect()", only the path to be used (i.e., the local address to bind to) can still be chosen, but the available paths and which one to prefer may be constrained by the choice of destination.

The three variants described in Section 5 work around this problem in different ways:

- o The approach in Section 5.2 places the whole automation of transport option selection into the "getaddrinfo()" function. The results are returned in an extended "addrinfo" struct and have to be applied manually by the application, including binding to a source address representing the selected path and applying all socket options provided in a list, for each connection attempt.
- o The approach in Section 5.1 adds a context to all socket- and name resolution-related API calls.
- o The approach in Section 5.3 puts all functionality into one call.

All of these approaches add the missing link between name resolution and the other parts of the API, but add a lot of state keeping either to the API, which the application developer has to manage, or to the Socket Intents library.

6.2. File Descriptors Considered Harmful

When using BSD sockets, file descriptors are the abstraction for network flows. Depending on the transport protocol used, their semantics changes and these file handles represent streams (SOCK_STREAM), associations (SOCK_DGRAM) or network interfaces (SOCK_RAW). This does not provide a unified API, but is merely an artifact of squeezing networking into the "Everything is a file" UNIX philosophy.

File descriptors make no good abstraction for automated protocol stack instance selection as applications have to adapt to changed semantics, e.g., whether message boundaries are preserved, depending on the transport protocol chosen.

File descriptors make no good abstraction for destination instance selection and path selection either. Once a socket has been created, its protocol stack instance is fixed, so selecting a path by binding to a local address and connecting to a destination instance is now only possible using this protocol stack instance. If such a connection attempt fails, it is possible to retry using another path and destination, but changing the protocol stack instance requires creating a new socket with a different file descriptor.

For further discussion of other asynchronous I/O weirdness with file descriptors see end of Section 6.3.

6.3. Asynchronous API Anarchy

Network I/O is asynchronous, but asynchronous I/O within the POSIX filesystem API is hard to use. There are at least three different asynchronous I/O APIs for each operating system.

To implement asynchronous I/O for our Socket Intents prototype, we wrapped one of the asynchronous I/O APIs that is available on most platforms: "select()". To make Socket Intents accessible to more applications and on more platforms, a production-grade system would need to wrap all asynchronous I/O APIs and implement most of the socket creation logic, path selection and connection logic within these wrappers. However, mixing asynchronous I/O and multithreading may lead to unintuitive behavior, e.g., calling our prototype's select() from different threads could lead to anything from deadlocks to busy waiting.

Another issue is that we use Unix domain sockets to communicate between our Multiple Access Manager and the Socket Intents API library called by the application, so we need to make sure that the application does not block on communication with the Multiple Access Manager.

Also the problems with using file descriptors get even worse. If a Socket API call should return immediately, it needs to provide the application with a reference to a flow that has not yet been fully set up, i.e., a reference to a "future" socket. An implementation of such an asynchronous API has to return an unconnected socket file descriptor, on which the application then calls, e.g., "select()", and starts using it once it becomes readable and writable. If the destination, path and transport protocol have not been chosen yet at this point, the file descriptor returned by the implementation might not yet have the final family and transport protocol. When the implementation later creates the final socket of the right type, it can re-bind it to the file-id of the originally returned file descriptor using "dup2". This procedure can easily lead to time-of-check to time-of-use confusion. To make things even worse, the application can copy the "future" file descriptor using "dup", which is rarely useful for sockets, but in combination with file descriptors used as "future" it leads to unexpected behavior.

6.4. Here Be Dragons hiding in Shadow Structures

The API variants described in Section 5.3 and Section 5.1 need to keep a lot of state in shadow structures that cannot be passed between the Socket API calls otherwise. This state needs to be cleaned up when the last copy of the file descriptor is closed or the

last socket held for reuse has timed out. In addition, access to these shadow structures has to be thread-safe.

Implementing both has turned out to be extremely error-prone and there is a high amount of unspecified behavior and platform-dependent extensions in the system library. These issues guarantee that an implementation of transport option selection that nicely integrates with BSD Sockets will come with lots of limitations and will not be portable across POSIX-compliant operating systems.

7. Conclusion

Adding transport option selection to BSD Sockets is hard, as the API calls are not designed to defer making and applying choices to a moment where all information needed for transport option selection is available.

After all, if limiting transport option selection to the granularity BSD Sockets typically provide today (TCP connections and UDP associations), the API variant described in Section 5.2 seems to be a good compromise, even if it forces the application to try all candidates itself (either in a sequential or partial parallel fashion). This option is easily deployable, but does not include automation of techniques like connection caching or HTTP pipelining.

The most versatile API variant described in Section 5.3 implements connection caching on the transport layer. This comes at the cost of heavily modifying existing applications. If feasible, given the unnecessary complexity of the file I/O integration of BSD sockets, it seems easier to move to a totally different system like [I-D.trammell-taps-post-sockets].

8. Acknowledgments

The API variant described in Section 5.2 was originally drafted and implemented by Tobias Kaiser mail@tb-kaiser.de [2] as part of his BA thesis.

This work has been supported by Leibniz Prize project funds of DFG - German Research Foundation: Gottfried Wilhelm Leibniz-Preis 2011 (FKZ FE 570/4-1).

9. References

9.1. Informative References

[ANRW17-MH]

Tiesel, P., May, B., and A. Feldmann, "Multi-Homed on a Single Link", Proceedings of the 2016 workshop on Applied Networking Research Workshop - ANRW 16, DOI 10.1145/2959424.2959434, 2016.

[ANRW18-Metrics]

"Metrics for access network selection (ANRW 2018)", n.d..

[I-D.tiesel-taps-communitgrany]

Tiesel, P. and T. Enghardt, "Communication Units Granularity Considerations for Multi-Path Aware Transport Selection", draft-tiesel-taps-communitgrany-02 (work in progress), May 2018.

[I-D.tiesel-taps-socketintents]

Tiesel, P., Enghardt, T., and A. Feldmann, "Socket Intents", draft-tiesel-taps-socketintents-01 (work in progress), October 2017.

[I-D.trammell-taps-post-sockets]

Trammell, B., Perkins, C., Pauly, T., Kuehlewind, M., and C. Wood, "Post Sockets, An Abstract Programming Interface for the Transport Layer", draft-trammell-taps-post-sockets-03 (work in progress), October 2017.

[RFC2119]

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

[RFC6824]

Ford, A., Raiciu, C., Handley, M., and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses", RFC 6824, DOI 10.17487/RFC6824, January 2013, <<https://www.rfc-editor.org/info/rfc6824>>.

[RFC7413]

Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <<https://www.rfc-editor.org/info/rfc7413>>.

[RFC7556]

Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <<https://www.rfc-editor.org/info/rfc7556>>.

9.2. URIs

[1] <https://github.com/fg-inet/socket-intents/>

[2] <mailto:mail@tb-kaiser.de>

Appendix A. API Usage Examples

A.1. Usage Example of the Classic / muacc_context API

In this example, a client application sets up a connection to a remote host and sends data to it. It specifies two Socket Intents on this connection: The Category of Bulk Transfer and the File Size of 1 MB.

```
#define LENGTH_OF_DATA 1048576

// Create and initialize a context to retain information across function
// calls
muacc_context_t ctx;
muacc_init_context(&ctx);

int socket = -1;

struct addrinfo *result = NULL;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

// Set Socket Intents for this connection. Note that the "socket" is
// still invalid, but it does not yet need to exist at this time. The
// Socket Intents prototype just sets the Intent within the
// muacc_context data structure.

enum intent_category category = INTENT_BULKTRANSFER;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_CATEGORY, &category, sizeof(enum intent_category));

int filesize = LENGTH_OF_DATA;
muacc_setsockopt(&ctx, socket, SOL_INTENTS,
    INTENT_FILESIZE, &filesize, sizeof(int));

// Resolve a host name. This involves a request to the MAM, which can
// automatically choose a suitable local interface or other parameters
// for the DNS request and set other parameters, such as preferred
// address family or transport protocol.
```

```
muacc_getaddrinfo(&ctx, "example.org", NULL, NULL, &result);

// Create the socket with the address family, type, and protocol
// obtained by getaddrinfo.
socket = muacc_socket(&ctx, result->ai_family, result->ai_socktype,
    result->ai_protocol);

// Connect the socket to the remote endpoint as determined by
// getaddrinfo. This involves another request to MAM, which may at this
// point, e.g., choose to bind the socket to a local IP address before
// connecting it.
muacc_connect(&ctx, socket, result->ai_addr, result->ai_addrlen);

// Send data to the remote host over the socket.
write(socket, &buf, LENGTH_OF_DATA);

// Close the socket. This de-initializes any data that was stored within
// the muacc_context.
muacc_close(&ctx, socket);
```

A.2. Usage Example of the Classic / getaddrinfo API

As in Appendix A.1, the application sets the Intents "Category" and "File Size".

```
#define LENGTH_OF_DATA 1048576

// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

struct muacc_addrinfo intent_hints = { .ai_flags = 0,
    .ai_family = AF_INET, .ai_socktype = SOCK_STREAM, .ai_protocol = 0,
    .ai_sockopts = &intents, .ai_addr = NULL, .ai_addrlen = 0,
    .ai_bindaddr = NULL, .ai_bindaddrlen = 0, .ai_next = NULL };

struct muacc_addrinfo *result = NULL;

muacc_ai_getaddrinfo("example.org", NULL, &intent_hints,
    &result);

// Create and connect the socket, using the information obtained through
// getaddrinfo
int fd;
fd = socket(result->ai_family, result->ai_socktype,
    result->ai_protocol);
muacc_ai_simple_connect(fd, result);

// Send data to the remote host over the socket, then close it.
write(fd, &buf, LENGTH_OF_DATA);
close(fd);

muacc_ai_freeaddrinfo(result);
```

A.3. Usage Example of the Socketconnect API

As in Appendix A.1, the application sets the Intents "Category" and "File Size". As we provide "-1" as socket, no we do not reuse existing connections.

```
#define LENGTH_OF_DATA 1048576

// Define Intents to be set later
enum intent_category category = INTENT_BULKTRANSFER;
int filesize = LENGTH_OF_DATA;

struct socketopt intents = { .level = SOL_INTENTS,
    .optname = INTENT_CATEGORY, .optval = &category, .next = NULL};
struct socketopt filesize_intent = { .level = SOL_INTENTS,
    .optname = INTENT_FILESIZE, .optval = &filesize, .next = NULL};

intents.next = &filesize_intent;

// Initialize a buffer of data to send later.
char buf[LENGTH_OF_DATA];
memset(&buf, 0, LENGTH_OF_DATA);

int socket = -1;

// Get a socket that is connected to the given host and service,
// with the given Intents
socketconnect(&socket, "example.org", 11, "80", 2, &intents, AF_INET,
    SOCK_STREAM, 0);

// Send data to the remote host over the socket.
write(socket, &buf, LENGTH_OF_DATA);

// Close the socket and tear down the data structure kept for it
// in the library
socketclose(socket);
```

Appendix B. Changes

B.1. Since -01

- o Updated list of gathered path characteristics
- o Reordered start of Policy section to make it clearer

B.2. Since -00

- o Fixed Author's affiliations and funding
- o Fixed acknowledgments

Authors' Addresses

Philipp S. Tiesel
TU Berlin
Marchstr. 23
Berlin
Germany

Email: philipp@inet.tu-berlin.de

Theresa Enhardt
TU Berlin
Marchstr. 23
Berlin
Germany

Email: theresa@inet.tu-berlin.de