# TAPS: an abstract application interface for QUIC

Mirja Kühlewind
ETH Zurich
mirja.kuehlewind@tik.ee.ethz.ch

Brian Trammell
ETH Zurich
trammell@tik.ee.ethz.ch

Anna Brunstrom
Karlstad University
anna.brunstrom@kau.se

Michael Welzl
michawe@ifi.uio.no
UniversityofOslo

Gorry Fairhurst
gorry@erg.abdn.ac.uk
UniversityofAberdeen

## ABSTRACT

The TAPS Architecture for Transport Services [6] provides a framework for the design of a protocol-independent asynchronous message-based transport API. While the traditional BSD socket API [1] requires the user to make a choice about which protocol to use from the beginning, the TAPS API focuses on transport features and gives the transport stack itself the opportunity to select the most appropriate protocol. This flexibility also supports deployment of new protocols, such as QUIC, because the application does not need to change in order to benefit from such new innovations. This poster defines two API mappings for QUIC, either exposing multistreaming explicitly to the application or providing a way to utilize multistreaming without application input.

**ACM Reference Format:**
Mirja Kühlewind, Brian Trammell, Anna Brunstrom, Michael Welzl, and Gorry Fairhurst. 2018. TAPS: an abstract application interface for QUIC. In *Proceedings of CoNEXT '18.* ACM, New York, NY, USA, 3 pages. https://doi.org/TBA

## 1 INTRODUCTION

QUIC is a new, encrypted transport protocol originally designed by Google that is currently under standardization in the IETF. Many of the benefits of QUIC revolve around the use of multi-streaming — the ability to multiplex separate sequences of application data onto a single transport connection without causing head-of-line blocking between them. QUIC provides this functionality as an optimization especially for HTTP. However, this functionality is actually not new: multi-streaming is, for example, also supported by SCTP [7], Adobe's RTMFP [8] and Structured Streams (SST) [2]. It can even be retro-fitted into TCP [3, 4].

Even though the mechanisms provided by these protocols are similar, not all protocols are always available on all platforms. Moreover, protocols typically expose their own API. Therefore, an application programmer needs to decide at development time which protocols to use in which scenario and then write customized code for each of them. This need to upgrade applications *per transport protocol*, and not *per function* can waste effort, and may well have contributed to the "ossification" of the Internet's transport layer [5].

By defining an API that exposes a set of transport services instead of a specific protocol, applications can be decoupled from the transport protocols that they use. This decoupling can have multiple benefits—for example, if a feature such as multi-streaming is unavailable (e.g., because standard TCP is the only available choice), a fall back to a reasonable behavior can be generically implemented in a library below the transport API, relieving the application programmer from this burden. As new protocols (or protocol mechanisms) become available (e.g., an upgrade to TCP that allows multistreaming), only this library would have to be updated, not the applications that use it.

The IETF Transport Services (TAPS) Working Group defines such an API [9]. To enable QUIC support within TAPs, this poster proposes two approaches: an interface mapping for QUIC where multistreaming is explicitly exposed to the application and an interface where multistreaming can be utilized even if the application does not explicitly demand this transport feature. The next section introduces the TAPS architecture and general concepts. The two approaches for a TAPS interface to QUIC are presented and are discussed in the concluding section.

## 2 AN ARCHITECTURE FOR TRANSPORT SERVICES

The TAPS architecture [6] provides a framework for design of a protocol-independent, asynchronous, and message-based transport API. In TAPS, communication starts with the creation of a **Preconnection** that can be configured and used


*CoNEXT '18, December 4–7, 2018, Heraklion/Crete, Greece*

ACM ISBN TBA…$TBA
https://doi.org/TBA

to start a rendezvous process, open listening ports, or trigger the initiation of a connection. At the point where a actual transport channel becomes ready, the user is handed a **Connection** to transmit or receive data.

Data transmission is realized by the transport layer based on **Messages** (chunks of data, with certain properties), which only have a value to the application if received as a whole. An application could choose not to segment its data and send a single large message. Usually the decision about how to segment the data is easy, beccause applications use some kind of message framing anyway.

At the receiver side, data is also provided to the appliaction as Messages. However, the boundaries between messages may not be known for all transports by the receiver. If unknown, the receiver delivers data in chunks whose size can be configured and the application itself has to reassemble a message.

# 3 APPROACHES FOR A TAPS INTERFACE

## 3.1 Transport Connection as QUIC Stream

In the TAPS architecture, multiple streams of the same transport connection are exposed to the application by grouping multiple connections together. Streams can "clone" an existing connection, instead of initiating a new connection. If a multistreaming-capable connection, e.g. using QUIC, is cloned, this leads to creation of a new QUIC stream. If however TCP was selected by the transport system for this connection, cloning will open another TCP connection, configured with the same connection parameters. This enables the transport system to automate fallback if QUIC is unavailable or not supported across the path. Similar functionality is implemented for SCTP streams in the NEAT library[1] [10].

The system ensures that QUIC does not fall back to plain TCP when encryption was required, but instead to TLS over TCP. Complicating matters further, for HTTP2 applications, a fallback should use a single TCP connection. Such a fallback would need to be realized at the application layer where HTTP is implemented. Alternatively, if HTTP is used as pseudo transport, it would also be possible to implement HTTP below the TAPS API. In this case, the multistreaming interface exposed to the application would sit on top of HTTP.

When multiple streams are exposed as connections, all messages sent on the same connection are respectively mapped to the same bidirectional QUIC stream and sent in-order. This requires that applications encode the boundaries of messages within the byte stream, as well as maintain an own message identifier to allow correlation of replies with outbound messages if needed.

[1] Available from https://github.com/NEAT-project/neat

## 3.2 Transport Connection as QUIC Connection

An alternative mapping is also possible that utilises the benefits of multistreaming protocols in TAPS even when streams are not exposed. This is most useful for applications originally developed for the use with TCP. Use of multistreaming, however, would enable transmission of independent messages without head-of-line blocking. However, these applications may not wish to change their interface to the transport (e.g. are deployed where TCP as the only available transport). If the application indicates that messages are independent with no requirement for in-order delivery, it is easy for the transport system to select a multiplstreaming protocol and map exactly one message to one stream. This also enables the receiver to recognize message boundaries and correctly deliver messages to the application. The use of bidirectional streams also make it easy to correlate the responses to previous request messages. Therefore message framing and identification can be provided by the transport system as well, even when a streaming based protocol is used underneath.

# 4 DISCUSSION

While sending only one message per stream seems on the first glance to be wasteful, however, it can provide a huge benefit to applications that do not already define their own message framing; especially in case of QUIC, where it is cheap to create and terminate streams. This approach further allows all messages to be delivered without head-of-line blocking, which can be especially beneficial for scenarios with unstable network conditions. Moreover, any message that expects a reply can use a bidirectional stream, allowing data returned on the same stream to be interpreted as a reply.

However, this approach does not provide a guarantee of ordering. An application, that is only able to process messages in strict order, would need itself to re-order messages which would eliminate the latency reduction from avoiding head-of-line blocking. For this style of application, it may be preferable to send messages that express dependencies on the same stream and to implement message segmentation and stream mapping within the application. An application that already defines message boundaries and semantics can reduce redundant processing and avoid this overhead by using an API that exposes streams to the application

# REFERENCES

[1] 2018. *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7.* Std 1003.1-2017. IEEE. http://www.opengroup.org/onlinepubs/9699919799/functions/contents.html (Revision of IEEE Std 1003.1-2008).

[2] Bryan Ford. 2007. Structured Streams: A New Transport Abstraction. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '07).*

ACM, New York, NY, USA, 361–372. https://doi.org/10.1145/1282380.1282421

[3] S. McQuistin, C. Perkins, and M. Fayed. 2016. TCP Hollywood: An unordered, time-lined, TCP for networked multimedia applications. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*. 422–430. https://doi.org/10.1109/IFIPNetworking.2016.7497221

[4] Michael F. Nowlan, Nabin Tiwari, Janardhan Iyengar, Syed Obaid Aminy, and Bryan Fordy. 2012. Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-compatible with TCP and TLS. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*. USENIX Association, Berkeley, CA, USA, 28–28. http://dl.acm.org/citation.cfm?id=2228298.2228337

[5] G. Papastergiou, G. Fairhurst, D. Ros, A. Brunstrom, K. Grinnemo, P. Hurtig, N. Khademi, M. Ti£¡xen, M. Welzl, D. Damjanovic, and S. Mangiante. 2017. De-Ossifying the Internet Transport Layer: A Survey and Future Perspectives. *IEEE Communications Surveys Tutorials* 19, 1 (Firstquarter 2017), 619–639. https://doi.org/10.1109/COMST.2016.2626780

[6] Tommy Pauly, Brian Trammell, Anna Brunstrom, Gorry Fairhurst, Colin Perkins, Philipp S. Tiesel, and Christopher A. Wood. 2018. *An Architecture for Transport Services*. Internet-Draft draft-ietf-taps-arch-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-taps-arch-01 Work in Progress.

[7] R. Stewart (Ed.). 2007. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard). (Sept. 2007), 152 pages. https://doi.org/10.17487/RFC4960 Updated by RFCs 6096, 6335, 7053.

[8] M. Thornburgh. 2013. Adobe's Secure Real-Time Media Flow Protocol. RFC 7016 (Informational). (Nov. 2013), 113 pages. https://doi.org/10.17487/RFC7016

[9] Brian Trammell, Michael Welzl, Theresa Enghardt, Gorry Fairhurst, Mirja Ki£¡hlewind, Colin Perkins, Philipp S. Tiesel, and Christopher A. Wood. 2018. *An Abstract Application Layer Interface to Transport Services*. Internet-Draft draft-ietf-taps-interface-01. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-taps-interface-01 Work in Progress.

[10] F. Weinrank and M. Ti£¡xen. 2017. Transparent flow mapping for NEAT. In *2017 IFIP Networking Conference (IFIP Networking) and Workshops*. 1–6. https://doi.org/10.23919/IFIPNetworking.2017.8264876