# Transparent Flow Mapping for NEAT

Felix Weinrank, Michael Tüxen
Münster University of Applied Sciences
Department of Electrical Engineering and Computer Science
Stegerwaldstrasse 39
D-48565 Steinfurt
Germany
{weinrank, tuexen}@fh-muenster.de

***Abstract*—The NEAT library provides application developers with a unified and platform independent API for network communication, regardless of the underlying network protocol. NEAT's abstraction layer approach allows the integration of new network protocols and transport features, transparently to the user. With QUIC, RTMFP and WebRTC, several widely deployed protocols make use of mapping multiple data streams to a single transport connection. However, the usage of multiplexing requires application developers to spend additional effort and has to be supported by both endpoints. This paper describes an approach to integrate multiplexing functionality into the NEAT library, giving application developers a simple way to use the benefits of mapping multiple data streams to a single transport connection without additional coding effort. We describe our considerations about feature negotiation, connection handling and data transmission for multiplexed data streams, an introduction to the NEAT library, the implementation details as well as measurement results and future steps.**

## I. Introduction

The internet is dominated by two transport protocols, TCP and UDP, supported by nearly every operating system and network. However, within the last years, several new transport protocols have been developed to better address the needs of modern network communication, providing new features and improved techniques. Many of these protocols are developed on top of the existing TCP / UDP stack provided by the operating system, increasing the compatibility with existing networks. By deploying them in user space, shorter software update cycles can be realized. The usage of multiplexing to bundle several data paths to a single transport connection has become a key technology for many of these protocols. Adobe's Secure Real-Time Media Flow Protocol (RTMFP) [1], Google's Quick UDP Internet Connections (QUIC) [2] and Web Real-Time Communication Data Channel (WebRTC) [3] are some examples for widely used protocols and protocol stacks using multiplexing. Especially for delay-sensitive applications with a low transmission rate, multiplexing can be very beneficial. The inherent transport protocol mechanisms, like flow-control and congestion-control, improve their effectiveness when larger quantities of data have to be transmitted. In case of packet loss, a higher packet rate per flow will result in faster retransmissions and less application-to-application delay. Also, sharing a common congestion window (cwnd) is beneficial for newly created connections or connections with a low sending rate. Additionally, the reduced amount of parallel connections improves the capacity of servers.

ISBN 978-3-901882-94-4 

Having the choice between several network protocols with specific characteristics give application developers the ability to use the best matching solution for their use case, but also causes new difficulties. Every protocol, regardless of whether accessed via the operating systems socket API or by a third party library on application level, requires a different API usage.

The NEAT library [4] addresses this issue by offering a unified and cross-platform API for network communication. This includes not only transport protocols offered by the underlying operating system, like TCP, UDP or Stream Control Transmission Protocol (SCTP), but also network protocols which operate at application level. For example, on platforms without native support for the SCTP protocol, like macOS and NetBSD, NEAT can seamlessly include an SCTP userland implementation [5].

Multiplexing has to be implemented at application level on top of the network stack of the operating system, requiring additional coding effort. The developer either has to implement it from scratch or use an existing library providing an application level protocol which includes this feature. Especially when the application has only limited knowledge of its peer's multiplexing capabilities, a fallback solution is required to guarantee a successful communication. Even if the effort for multiplexing is high and its usage is not beneficial for all traffic patterns, previous investigations [6] have shown that the advantages outweigh the disadvantages for many use-cases.

Our work introduces a multi-purpose multiplexing solution for the NEAT library, providing application developers with the benefits of multiplexing without additional effort. This includes an automatic negotiation mechanism which ensures a maximum of compatibility and transparency to the application. After introducing the NEAT library, its concept of flows and how they map to transport connections, we will explain the concept and implementation of transparently mapping multiple flows to a single SCTP transport connection without prior knowledge about the peer's capabilities. The section is followed by some of our measurement results, considerations about alternative transport protocols and an outlook for our ongoing and future work.

## II. NEAT Library

The NEAT library offers application developers a new interface for network communication. Instead of using the traditional socket API, which requires a lot of protocol and platform specific coding effort, NEAT provides a unified cross-platform API for network communication. This includes DNS-name resolution, connection handling, buffer management and encryption. NEAT is built on top of the libuv [7] event-loop library, and, therefore, it offers a non-blocking and callback-based API. Additionally to the functionality provided by the NEAT library, the developer has full access to the libuv library's functionality. A detailed insight about the concept and architecture of NEAT has been given in [8].

Instead of specifying a transport protocol, the developer specifies his requirements for the properties provided by the transport service for every path. These requirements are for example ordered/unordered delivery, message preservation or reliability. Taking the preferences and requirements of the application into account, the NEAT library chooses the best matching protocol at run-time and cares for the protocol specific connection handling. In addition to the widely used TCP and UDP protocols, NEAT supports the SCTP protocol, the native SCTP implementations on FreeBSD and Linux, as well as the SCTP userland implementation on platforms not having a native support for SCTP, such as macOS and NetBSD.

## III. NEAT Flows

In NEAT, a communication channel between two application endpoints is called *flow*. Flows offer applications a bi-directional data transmission interface to the network.

In order to create a new flow, the client application provides a DNS-name or IP-address and the port-number of the remote endpoint and an optional set of properties. These properties offer a flexible way of configuring the requirements for the new flow, allowing a high level transport feature requirements specification. This includes demanding a reliable data transport and message preserving boundaries, as well as a lower level approach by setting the transport protocol(s) or protocol features, like SCTP's multihoming, and encryption. There is a distinction between required and optional flow properties. For example, an application may require the SCTP protocol for a flow and optionally enable SCTP's multihoming feature. Flows are assigned to flow groups where they have a specific priority within the group, affecting the share of the available bandwidth. If not specified, all flows are assigned to flow zero.

Based on this information and collected data from previous connections, available address-protocol candidates are built, and the NEAT library tries to establish a connection, based on the flow specific properties.

If multiple address-protocol candidates are available, NEAT probes all available candidates by using the Happy-Eyeballs algorithm [9]. In case of several successfully established connections, NEAT will select the best matching connection and close all spare connections. This selection is based on the flows properties, taking the transport protocol specific characteristics

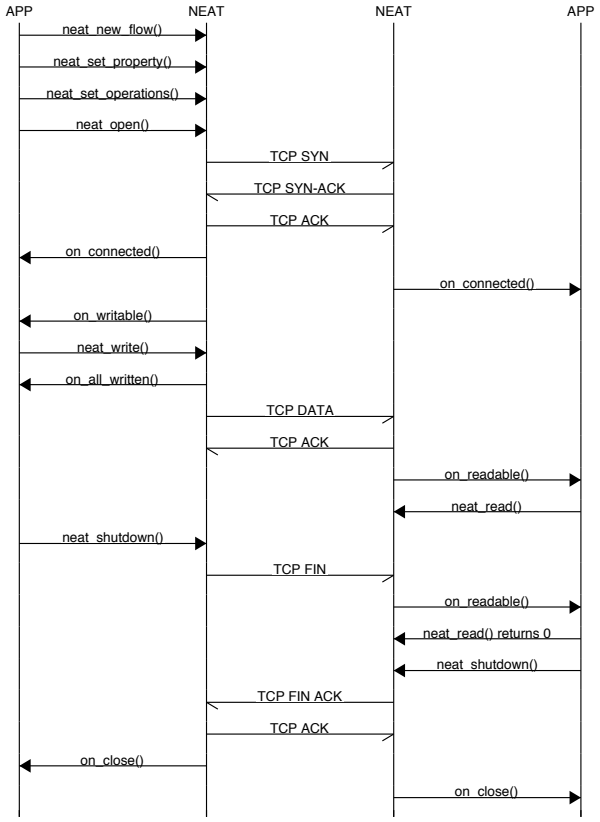


Fig. 1. NEAT message and function sequence example using TCP

and user specified priorities into account. For example, the TCP connection setup takes less round-trips than the SCTP connection setup. If the NEAT library probes TCP and SCTP candidates, the TCP connection will probably be established before the SCTP connection. To overcome this disadvantage for SCTP, NEAT will wait for an additional period of time before evaluating the results. When a connection for a candidate has successfully been established, the NEAT flow changes its state from *connecting* to *open*, and the application will be notified by means of the *on_connected* callback. Figure 1 illustrates the usage and operation of a NEAT flow using the TCP protocol.

When NEAT uses the stream-based TCP transport protocol, the flow is reported ready for data transmission to the application by calling the *on_connected* callback, right after the network socket becomes writable, followed by calling the *on_writable* callback. The application may now send and receive data via the flow's data transmission functions. Due to NEAT's non-blocking-io constraint, applications can write data to connected flows at any time. The NEAT library will try to send the data directly to the network. However, if the socket is not writable or the amount of data cannot be sent at once, the unsent data is buffered in a dedicated flow buffer. The data will be sent as soon as the underlying network socket becomes writable again. When all data has been transmitted to the network and no outstanding data is left in the outgoing

flow buffer, NEAT will notify the application by calling the *on_all_written* callback. This callback allows applications to saturate a network connection without bloating the outgoing flow buffer.

When the flow's network socket becomes readable, the NEAT layer notifies the application by triggering the *on_readable* callback. The application can now read data from the flow by using the *neat_read* function and by providing a read buffer with a given size. If the amount of received data exceeds the provided buffer size, the *on_readable* callback will be triggered again until all received data has been handed over to the application. Internally, the application reads directly from the flow's underlying network socket without additional buffering by NEAT. Applications may close a NEAT flow at any time by calling *neat_close* or initiate a graceful connection shutdown by using *neat_shutdown*. The library will transmit all outstanding data to the remote peer, handle the connection closing procedure and trigger the *on_close* callback when all operations have been finished. After the *on_close* callback has been triggered, no flow specific callbacks will be triggered by the library and subsequent calls to read- or write-functions on the flow will result in an error.

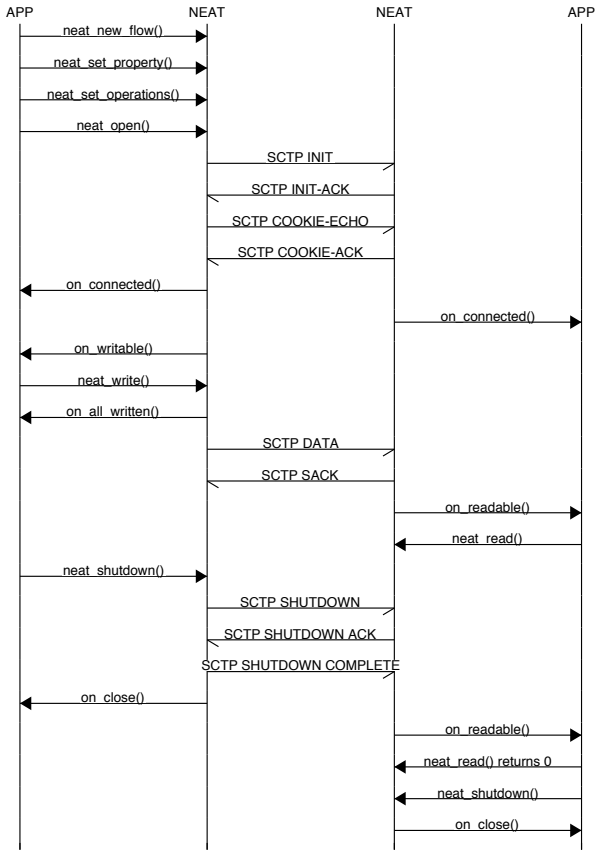


Fig. 2. NEAT message and function sequence example using SCTP

The usage of message oriented protocols like SCTP or UDP within NEAT differs internally from stream-based protocols like TCP, but operates transparently to the application. Figure 2 illustrates the usage and operation of a NEAT flow using the SCTP protocol. Once a SCTP based transport connection is established, NEAT will evaluate SCTP specific connection parameters and extensions before announcing the flow's *open* state to the application. The parameters and supported extensions are important for the further usage of the flow. They include the amount of available SCTP streams and support of explicit end of record (EOR) marking, which allows the transmission of arbitrary large user messages by the application. Once all SCTP notifications have been read, NEAT will trigger the *on_connected* callback to notify the application that the flow is ready for data transmission. When the application writes data to the flow, it will be sent to the network or buffered within a flow specific send buffer, similar to TCP, as explained before. In contrast to stream based protocols, NEAT buffers unsent data in a message preserving way by using a message queue. Every user message is buffered in a distinct entry within the queue. When incoming data is available at the network socket, NEAT will read the incoming message into the flow specific receive buffer. If the message is a fragment of a larger user message, NEAT receives and reassembles all fragments before announcing the complete message to the application via the *on_readable* callback. NEAT will only buffer a single user message, no further messages are read from the socket until the buffered message has been read by the application, in order to avoid bloating the incoming buffer on the receiver side. Closing SCTP based flows is similar to the procedure of flows using TCP. NEAT, like SCTP, does not support TCP's half-closed feature, in order to keep the promise of a unified API.

## IV. Transparent Flow Mapping

Transparent flow mapping hooks into NEAT's abstraction layer approach by multiplexing multiple NEAT flows to a single transport connection without additional actions of the application. If both endpoints support multiplexing and the applications have enabled the support for transparent mapping in their settings, NEAT will automatically use the transparent mapping. As shown in Figure 3, the flows still show up as they would when using a dedicated transport connection, providing the same API and functionality.

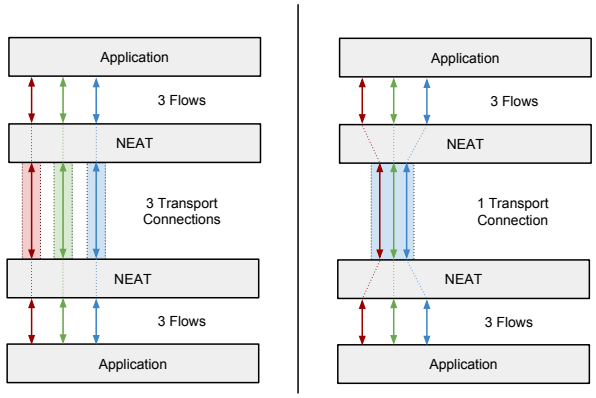


Fig. 3. NEAT flows - comparison of 1:1 and transparent flow mapping

### A. Requirements and Negotiation

Before the transparent mapping can be used, both peers have to fulfill some requirements and negotiate the support of the feature. NEAT requires some SCTP specific extensions to be supported by the network stacks on both sides, including the support for Stream Reconfiguration [10]. The user may require a flow to preserve data message boundaries. In this case NEAT requires the support for the SCTP User Message Interleaving (I-DATA) [11] extension, in order to prevent a sender side head-of-line blocking. If the local requirements are fulfilled, NEAT has to negotiate the multiplexing capabilities with its peer. This is achieved by using SCTP's adaptation layer indication. The NEAT specific adaptation layer indication value is exchanged within SCTP's connection setup procedure and provided as an SCTP notification on both sides, once the connection has been established. If all requirements are met, the transport connection is marked as usable for transparent flow mapping. Otherwise the NEAT library continues operating in regular mode and maps every flow to a separate transport connection. This approach has the advantage of being fully interoperable with peers not using the NEAT library.

### B. Flow creation

Creating a new flow triggers the NEAT library to search for an established SCTP association with a matching tuple of destination address, port, properties and support for transparent flow mapping. Only flows belonging to the same flow group are taken into this survey, allowing the application to prevent multistreaming for a flow by assigning it to an empty flow group.

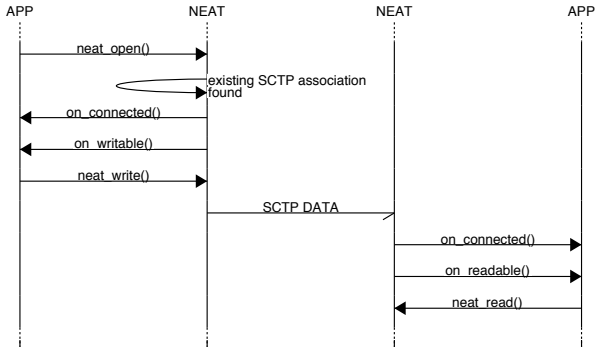


Fig. 4. Transparently mapped flow creation procedure

If NEAT discovers a matching SCTP association, the new flow is mapped to it instantly and all ongoing connection establishment procedures for other address-protocol-candidates are stopped. The mapping is realized by assigning the new flow to a dedicated SCTP stream of the established association. The amount of flows per transport connection is limited by the number of available incoming and outgoing SCTP streams per association. SCTP itself supports up to 65535 streams per association. As shown in Figure 4, the NEAT library will notify the application instantly by triggering the *on_connected* and *on_writable* callbacks. If multiple SCTP associations are available for a transparent mapping, NEAT takes the first one to bundle as many flows as possible.

The first flow, for which the SCTP association has initially been created, will always use stream id zero. All additional flows are assigned to unused stream ids. To avoid a glare situation, occurring when both endpoints map new flows simultaneously, the peer which initiated the transport connection will use even stream numbers whereas the remote side will map its flows to odd stream numbers. Both sides maintain a status map of the assigned stream numbers.

Due to the lack of a connection setup procedure on the network, the creation of a new flow is signalized to the remote side by sending the first data message. Transparently mapped flows are instantly ready for data transmission without additional round-trips and, superior to the TCP fast open mechanism [12], the amount of outgoing data is not limited. When receiving an SCTP message on a previously unused stream id, the receiver creates a new incoming flow and triggers the same callbacks as if a new connection using a native transport connection had been opened. Using an implicit flow setup restricts the usage of transparently mapped flows for use cases where the server starts transmitting data to the client without receiving a request, for example a daytime-server. A possible approach to overcome this limitation is the explicit connection setup by sending a control message with a specific Payload Protocol Identifier (PPID) to trigger the incoming flow procedure on the receiver side.

### C. Data transmission

One of the most challenging parts of transparently mapped flows is the handling of incoming and outgoing data. Sharing a network socket between multiple flows requires the NEAT library to cope with scheduling and buffer management techniques. When a shared socket becomes writable, NEAT schedules over all assigned flows in a round-robin manner. Beginning with the first flow, the library transmits scheduled data from the outgoing flow buffer before triggering the flow's *on_writable* callback. When the flow neither has outstanding data in the buffer nor received new data from the application, the library will continue with the same procedure for the next flow. As mentioned in the negotiation section, applications may send arbitrary large messages and require message boundary preservation. To transmit user messages larger than the maximum segment size (MSS), SCTP supports fragmentation and reassembly. The sender fragments the user message in multiple DATA chunks for transmission which are reassembled by the receiver. If the sender starts transmitting a large user message, consisting of several data chunks, transmissions on all other streams are blocked until all fragments of the user message have been transmitted. To overcome this sender side head-of-line-blocking when transmitting large user messages, NEAT uses the SCTP I-DATA extension. I-DATA solves the sender side head-of-line-blocking issue by supporting message interleaving [11] and is also used in the WebRTC protocol for the same purpose [3]. Another major change for multistreaming affects the receiver side. Whereas a one-to-one style

mapped flow only buffers a single incoming user message, a socket used for multistreaming reads messages from the underlying SCTP socket until all assigned flows have at least one user message in their receive buffer. If the sender transmits data on two or more flows and the receiver does not read data from one particular flow, NEAT buffers this data to prevent other multistreamed flows from being blocked by this flow. Limiting the maximum amount of buffered data on the receiver side would either result in dropping data for the particular flow or in blocking all incoming messages for every transparently mapped flow on the affected SCTP socket, both cases are undesirable. A possible approach to overcome this limitation would be application based flow control per transparently mapped flow. Here the receiver signalizes the increasing flow buffer by sending a specific control message to the sender to prevent further transmissions on this particular SCTP stream.

### D. Teardown

Analogous to the creation of a transparently mapped flow, NEAT cannot make use of SCTP's native closing procedure for teardown. Instead, NEAT uses the SCTP Stream Reconfiguration extension for the closing procedure. When the application calls the *neat_shutdown* function for a flow to initiate a graceful shutdown, all outstanding data will be sent and the application may still receive data from its peer, shown in Figure 5. Internally, the flow is marked as closing by the library and once the outgoing flow buffer has been drained, NEAT will trigger the SCTP stream reset procedure for the outgoing stream. After calling the *neat_shutdown* method, the application cannot write any additional data to the flow, the *on_writable* event will not be triggered any more and calling *neat_write* will cause an error.

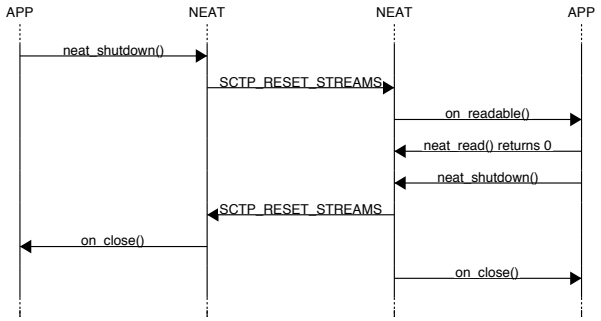


Fig. 5. Transparently mapped flow shutdown procedure

Upon receiving an SCTP Stream Request for an incoming stream, NEAT indicates the event by a return value of null when the application calls the *neat_read* function. The flow will not accept new data via the *neat_write* function for transmission. When the remote endpoint also responds with a Stream Reset Request for the incoming stream, the closing procedure of the flow has finished and all resources may be freed. This behavior reflects the connection teardown process for unmapped flows. An application may also use the *neat_close* function. In contrast to *neat_shutdown* the closing procedure resets the outgoing as well as the incoming SCTP streams. Once the closing procedure for a flow has been finished, the SCTP stream id may be reassigned to a new multistreamed flow. Both endpoints maintain an SCTP association assigned status map for every stream id.

## V. Measurements

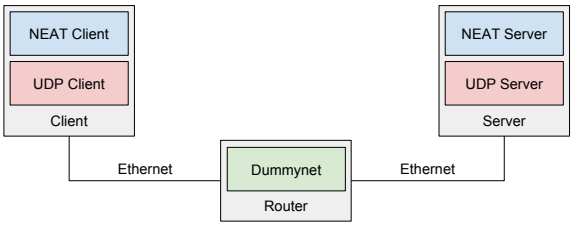


Fig. 6. NEAT flow mapping - 1:1 mapping and transparent flow mapping

To examine the advantages and disadvantages of a transparent mapping, we used a client-router-server scenario. All machines are physical nodes running FreeBSD 12 with a *GENERIC-NODEBUG* kernel. As shown in Figure 6, the NEAT Client and the NEAT Server are connected via the router which emulates various network conditions between the two peers by using FreeBSD's builtin dummynet [13] network emulation tool. The router emulates different network conditions by adding delay and packet loss to the path between the server and the client. To achieve some randomness during the measurements, the client transmits a low amount of random UDP messages to the server. Our benchmarking tool, using the NEAT library, is designed to measure a variety of parameters, including the application-to-application-delay between both applications for every flow.

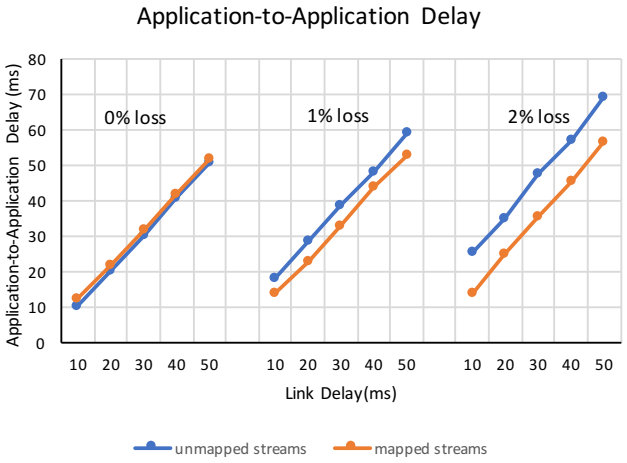


Fig. 7. Measurement results comparison of mapped and unmapped flows

The scenario compares the impact of packet loss and link delay for mapped and unmapped streams concerning application-to-application delay. The NEAT Client opens two SCTP based flows to the NEAT Server and sends small messages between 100-200 bytes periodically with a low

rate on each flow to simulate an application using multiple flows for data transmission. This behavior is typical for many use-cases like a browser scheduling requests over multiple connections or control systems reporting data to a central instance.

We varied the link delay, starting with 10 milliseconds in steps of 10 milliseconds to a delay of 50 milliseconds and used loss rates of zero, one and two percent on the link. Every measurement ran for 60 seconds and was repeated ten times. As shown in Figure 7, the results show a slightly higher application delay for multiplexed flows on connections without loss, resulting from internal data handling within the NEAT library. In case of packet loss, the transparently mapped flows show a lower delay compared to regular flows. This is a result of better utilizing the transport protocols loss detection algorithms.

Our results show a significant application-to-application delay improvement for transparently mapped flows in comparison to regular flows, fulfilling our expectations.

## VI. Alternative transport protocol considerations

As mentioned in the previous sections, transparent flow mapping is not tied to the SCTP protocol. In addition to the SCTP protocol, Google's QUIC protocol also covers many requirements for the transparent mapping of multiple flows and, since it is layered on top of UDP, it can seamlessly be integrated into NEAT's abstraction layer approach. Mainly developed to replace TCP as the underlying transport protocol for HTTP2, QUIC is not tied to this use-case and may be used by any other application for generic purposes. Similar to SCTP, QUIC uses multiplexed streams and does not suffer from head-of-line blocking. In contrast to SCTP, QUIC supports zero-RTT connection setup and uses encryption by default. Due to QUIC's early stage of development and the lack of a specification, QUIC is a candidate for future work. Another candidate is Adobe's RTMFP protocol which is also UDP based and multiplexes multiple flows over a single transport connection. Although specified in by an RFC [1], no official RTMFP library is available and the development has been discontinued.

## VII. Conclusion and outlook

While multiplexing of several data streams on a single transport connection has become a feature more and more popular due to its usage within new protocols, it still requires additional effort for application developers. Especially when the application has no knowledge about its peer. Even if the developer uses a userland implementation of a transport protocol that supports multiplexing, it still remains an additional coding effort, especially when a fallback solution is desired. With NEAT's approach of creating an abstraction layer on top of the different network protocol APIs to give developers a unified way of accessing transport function. We were able to seamlessly integrate a transparent flow mapping feature which gives application developers the benefit of multiplexing without additional coding effort and still being fully compatible. We introduced our approach for multiplexing using SCTP, the integration in the NEAT library and the techniques for feature negotiation, flow handling and data transmission. Our measurements show advantages of transparently mapped flows over regular flows in usual use-cases. In our ongoing work, we are focusing on improving the buffer management and scheduling of concurrent multiplexed flows. Additionally, we will add support for WebRTC Data-Channels [3] to the NEAT library. This allows developers to use NEAT not only for client-server communication but also for building peer-to-peer applications.

## Acknowledgments

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 authors.

## References

[1] M. Thornburgh, "Adobe's Secure Real-Time Media Flow Protocol," RFC 7016 (Informational), Internet Engineering Task Force, Nov. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc7016.txt

[2] R. Hamilton, J. Iyengar, I. Swett, and A. Wilk, "Quic: A udp-based secure and reliable transport for http/2," Working Draft, IETF Secretariat, Internet-Draft draft-hamilton-early-deployment-quic-00, July 2016, http://www.ietf.org/internet-drafts/draft-hamilton-early-deployment-quic-00.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-hamilton-early-deployment-quic-00.txt

[3] R. Jesup, S. Loreto, and M. Tuexen, "Webrtc data channels," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-rtcweb-data-channel-13, January 2015, http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-rtcweb-data-channel-13.txt

[4] NEAT Project, "A New, Evolutive API and Transport-Layer Architecture for the Internet,"
*Available at https://www.neat-project.org/*, 2017.

[5] "usrsctp - a portable SCTP userland stack,"
*Available at https://github.com/sctplab/usrsctp*, 2017.

[6] M. Welzl, F. Niederbacher, and S. Gjessing, "Beneficial Transparent Deployment of SCTP: the Missing Pieces," *IEEE Globecom 2011 proceedings*, 2011.

[7] libuv — Cross-platform Asynchronous I/O. [Online]. Available: https://libuv.org/

[8] N. Khademi, D. Ros, M. Welzl, Z. Bozakov, A. Brunstrom, G. Fairhurst, K.-J. Grinnemo, D. Hayes, P. Hurtig, T. Jones, S. Mangiante, M. Tuexen, and F. Weinrank, "NEAT: A Platform- and Protocol-Independent Internet Transport API," *IEEE Communications Magazine*, 2017.

[9] D. Wing and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts," RFC 6555 (Proposed Standard), Internet Engineering Task Force, Apr. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6555.txt

[10] R. Stewart, M. Tuexen, and P. Lei, "Stream Control Transmission Protocol (SCTP) Stream Reconfiguration," RFC 6525 (Proposed Standard), Internet Engineering Task Force, Feb. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6525.txt

[11] R. Stewart, M. Tuexen, S. Loreto, and R. Seggelmann, "Stream schedulers and user message interleaving for the stream control transmission protocol," Working Draft, IETF Secretariat, Internet-Draft draft-ietf-tsvwg-sctp-ndata-08, October 2016, http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctp-ndata-08.txt. [Online]. Available: http://www.ietf.org/internet-drafts/draft-ietf-tsvwg-sctp-ndata-08.txt

[12] Y. Cheng, J. Chu, S. Radhakrishnan, and A. Jain, "TCP Fast Open," RFC 7413 (Experimental), Internet Engineering Task Force, Dec. 2014. [Online]. Available: http://www.ietf.org/rfc/rfc7413.txt

[13] "The dummynet project,"
*Available at http://info.iet.unipi.it/~luigi/dummynet/*, 2017.