Preprint (accepted version)

To appear in IEEE Communications Magazine
http://www.comsoc.org/commag/

Until published, please cite as:

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

NEAT: A Platform- and Protocol-Independent Internet Transport API

Naeem Khademi, David Ros, Michael Welzl, Zdravko Bozakov, Anna Brunstrom, Gorry Fairhurst, Karl-Johan Grinnemo, David Hayes, Per Hurtig, Tom Jones, Simone Mangiante, Michael Tüxen, and Felix Weinrank

***Abstract*—The sockets Applications Programming Interface (API) has become the standard way that applications access the transport services offered by the Internet Protocol stack. This paper presents NEAT, a user-space library that can provide an alternate transport API. NEAT allows applications to request the service they need using a new design that is agnostic to the specific choice of transport protocol underneath. This not only allows applications to take advantage of common protocol machinery, but also eases introduction of new network mechanisms and transport protocols. The paper describes the components of the NEAT library and illustrates the important benefits that can be gained from this new approach. NEAT is a software platform for developing advanced network applications that was designed in accordance with the standardization efforts on Transport Services (TAPS) in the Internet Engineering Task Force (IETF), but its features exceed the envisioned functionality of a TAPS system.**

I. Introduction

For more than three decades, the Internet's transport layer has essentially supported just two protocols and the original design of the sockets API offered only two Transport Services to applications. One service provided stream-oriented in-order reliable delivery, manifested in TCP, and the other a message-based unordered unreliable delivery, manifested in UDP.

Today, more than three decades later, these are the only two transport protocols commonly offered by operating systems to applications. UDP-based applications are used for a wide variety of datagram services from service discovery to interactive multimedia, while TCP became the dominant protocol for Internet services from web browsing to file sharing and video content delivery. While their success has often been attributed to the robustness of these protocols, during the last decades new service requirements have emerged that are beyond what TCP can deliver or UDP can offer—examples include: an interactive multimedia application may prefer to prioritize low latency over strictly reliable delivery of data, but could use partially-reliable delivery to improve quality while ensuring timeliness, or an application may be designed to take advantage of multihoming when this is available. UDP has also emerged as a substrate upon which user-space transport protocols are being developed—many customized for specific applications (e.g., the QUIC protocol), where much effort can be expended re-implementing common transport functions.

A handful of protocols have been proposed to provide Transport Services beyond those of TCP and UDP; most notably, SCTP, DCCP and UDP-Lite. However none of these have seen widespread use or universal deployment. The reason behind this is often attributed to *ossification* of the Internet's transport layer, where further evolution has become close to impossible. This has two major aspects:

- **Inflexibility of the current socket API:** Application programmers need to specify *transport protocol-specific* configurations to request a desired service. This binding to protocols inevitably requires programmers to recode their applications to take advantage of any new transport protocol. It also introduces complexity when there is a need to customize for different network scenarios, and choose appropriate transport protocol-specific parameters.
- **Deployment vicious circle:** New protocols and mechanisms cannot be expected to work in unmodified networks. Some equipment may need to be reconfigured, updated or replaced to deploy a new protocol. Developers seeking to use new protocols simply find they cannot be relied upon to work across the Internet. Because the current socket API requires application developers to specifically choose a certain protocol, they therefore tend to avoid using a protocol other than TCP or UDP, knowing that any others are likely to be unsuccessful for many network paths. This chicken-and-egg situation has made it hard for unused transport protocols to become deployed in the Internet—even if they would provide a better service to some applications.

In this paper, we introduce the *NEAT Library*. This is a software library built above the socket API to provide networking applications with a new API offering platform- and protocol-independent access to Transport Services. NEAT is, to the best of our knowledge, the first prototype implementation of IETF standardization efforts on Transport Services (TAPS), which we will discuss in Section V. NEAT and its related standardization efforts in TAPS can re-enable the evolution of the Internet's transport layer because they break the deployment vicious circle; NEAT's flexible, customizable API makes it easy to define and use novel services on top of the socket API, seamlessly leveraging new transport protocols

Naeem Khademi and Michael Welzl are with the Department of Informatics, University of Oslo, Norway. E-mail: {naeemk, michawe}@ifi.uio.no.

David Ros and David Hayes are with Simula Research Laboratory, Norway. E-mail: {dros, davidh}@simula.no.

Zdravko Bozakov and Simone Mangiante are with Dell EMC, Ireland. E-mail: {Zdravko.Bozakov, Simone.Mangiante}@dell.com.

Anna Brunstrom, Karl-Johan Grinnemo and Per Hurtig are with Karlstad University, Sweden. E-mail: {anna.brunstrom, karl-johan.grinnemo, per.hurtig}@kau.se.

Gorry Fairhurst and Tom Jones are with the University of Aberdeen, Aberdeen, United Kingdom. E-mail: {tom, gorry}@erg.abdn.ac.uk.

Michael Tüxen and Felix Weinrank are with Münster University of Applied Sciences, Germany. E-mail: {tuexen, weinrank}@fh-muenster.de.

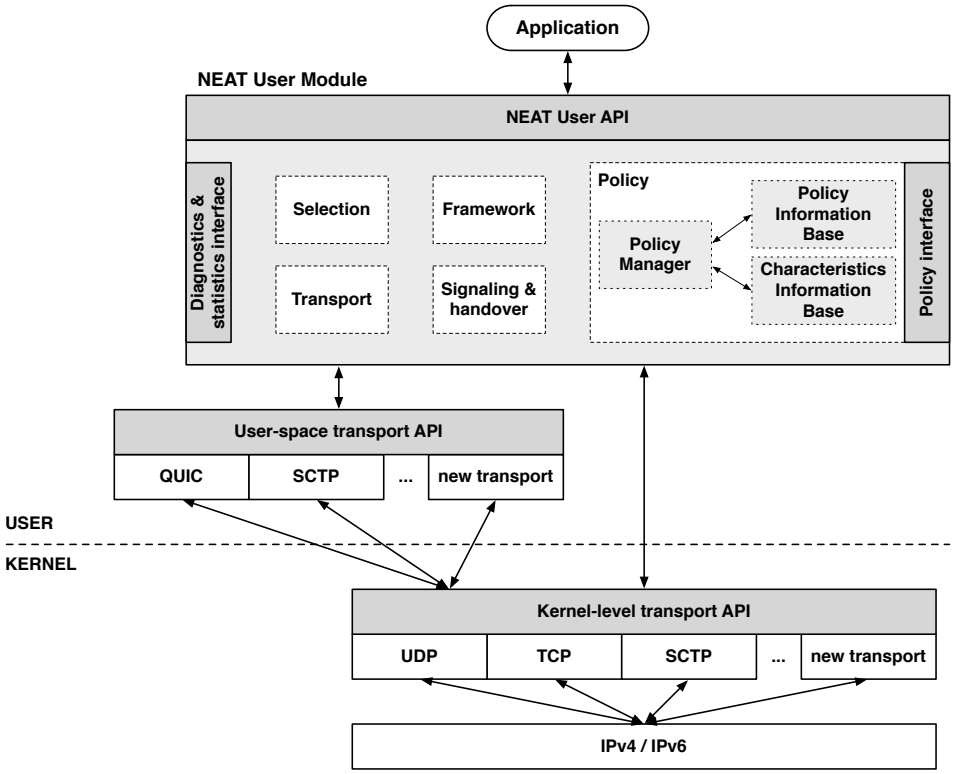


Fig. 1. The architecture of the NEAT System.

as they become available. This, in turn, may create a shift in the traffic pattern seen by vendors and administrators of middleboxes that could at some point lead them to support such traffic. Section IV presents several examples of benefits that NEAT offers to applications.

II. Background

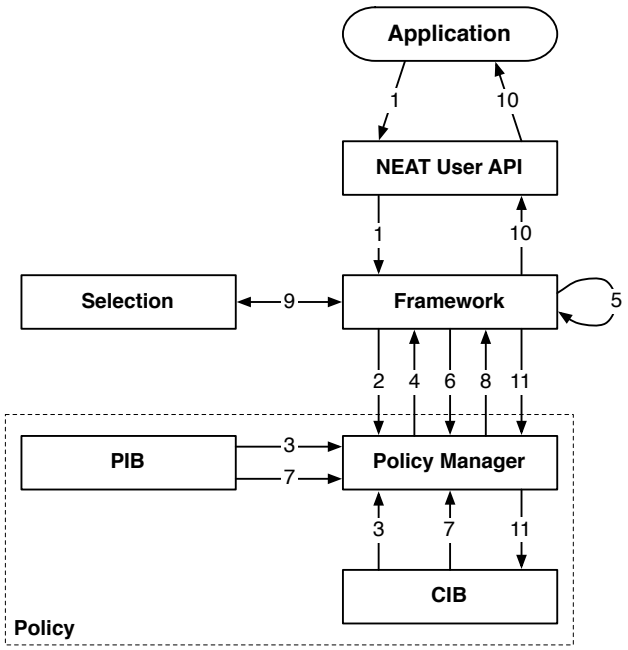
TCP and UDP are a part of the kernel of almost all operating systems and are also supported by nearly all middleboxes. During its lifetime, TCP has been substantially improved. However, the evolution of TCP has had to deal with constraints. Changes to packet format have to consider that middleboxes might block or limit communication. Therefore, a fallback mechanism to the old packet format has become a part of such protocol extensions. New protocol mechanisms (e.g., congestion control or loss recovery mechanisms) mostly focus on single-sided changes to allow faster deployment—but the speed of deployment is still limited by the software development cycle of operating systems.

In addition, having a feature available on an operating system does not imply that it is made available to an application running with user privileges; new features are often disabled by default and turning them on requires special privileges since it has host-wide consequences.

Because UDP provides only minimal services (port numbers and a checksum), it is possible to use it as substrate to implement transport protocols on top of it to introduce features; this approach has become increasingly common. This leads to every UDP-based application to some extent needing to implement the same core set of functions [1]. However, it also leads to per-application protocol stacks, where transport protocols cannot easily be moved between applications (and making this possible is often not in the interest of the application developer). Developing an efficient transport protocol is a difficult task which requires a number of features to be re-implemented again and again. UDP-based transport protocols have also done nothing to fix the general architectural problem: the socket API's protocol binding remains, typically with a choice between only TCP and UDP.

Specific applications can require services not provided by TCP. One example is the transport of signaling messages in telephony signaling networks. This is used to transfer mostly small messages and requires a high level of fault tolerance. When a protocol stack for this application was developed, a new transport protocol, SCTP [2], was created to fulfill these specific requirements. It was possible to deploy SCTP in these networks because there were no middleboxes, and kernel implementations for the operating systems used in telephony signaling networks were developed.

Currently, the IETF and the World Wide Web Consortium (W3C) are developing WebRTC, a technology for real-time multimedia communication directly between web browsers. Non-media communication using SCTP is also supported; to facilitate deployment across arbitrary Internet paths, SCTP runs over UDP. Google has developed QUIC, a UDP-based transport protocol with features including fast connection setup, cross-layer optimized security, and a modern congestion control and loss recovery mechanism. If QUIC fails to traverse a middlebox, the web browser can fall back to using TCP.



1. Request to open flow & pass application requirements
2. Query PM about feasible transport candidates based on destination domain name
3. PM determines available transport candidates that fulfil policy (PIB) and cached information (CIB)
4. Return ranked list of feasible transport candidates as pre-filter for address resolution
5. Resolve addresses
6. Query PM about feasible transport candidates for resolved destination address
7. PM builds candidates, assigning priorities based on PIB/CIB matches
8. Return ranked list of feasible transport candidates for flow establishment
9. Do Happy Eyeballs with candidates, according to specified priorities
10. Return handle to selected transport solution
11. Cache results from Happy Eyeballs in the CIB

Fig. 2. Simplified workflow showing how NEAT components interact when opening a flow.

By moving a transport protocol from the operating system to the application (e.g., WebRTC and QUIC integrated in web browsers), the release cycle can be substantially shortened, the implementation becomes independent from the operating system, and the protocol can be tailored to the specific application. Using UDP encapsulation is the only option to not using TCP and be able to traverse middleboxes.

This enables a larger variety of transport protocols to co-exist and change over time, but does not help with the issue of per-application protocol stacks mentioned before. An application programmer has to add complexity to benefit from advanced features in their application; this requires utilizing different APIs, figuring out which protocols are supported by the remote end-points, selecting protocol mechanisms, and providing fallback mechanisms when these happen to not work across the current network path. None of these are specific to a particular transport protocol, but are related to the need for the programmer to work with a variety of transport protocols. This general problem could be addressed by defining a new transport system, as outlined next.

III. Re-enabling evolution: introducing NEAT

As discussed above, using transport services beyond TCP and UDP today puts a high burden on the application developer. The NEAT Library addresses this problem by providing application developers with one enhanced API that is transport-protocol independent, with the library providing support for selecting the best available transport option *at run-time* and handling fallback between transport protocols as needed. Running as a user-space library, NEAT can make use of transports running both in user space and in the kernel, all transparent to the applications. Protocols like SCTP are already supported over many paths, but they cannot be easily used by application programmers unless they are supported over *all* paths. NEAT changes this by placing functions such as selecting a transport and handling fallback below the API. NEAT allows such functions to evolve with the network, rather than be bound to specific applications.

Figure 1 provides a schematic view of the NEAT architecture. Applications employ the NEAT User API to access transport services. This API is located in the NEAT User Module, which is the core of NEAT and comprises components that together deliver services tailored to application requirements at run-time. The components in the module are grouped in five categories: Framework, Policy, Selection, Transport, and Signaling & Handover.

Framework components provide basic functionality required to use NEAT. They define the structure of the NEAT User API and implement core library mechanisms. Applications provide information about their requirements for a desired transport service via this API.

Policy components comprise the Policy Information Base (PIB), the Characteristics Information Base (CIB), and the Policy Manager (PM). The function of the PM is to generate a ranked list of connection candidates that fulfill the application requirements while taking system and network constraints into account and adhering to configured policies. All policy components operate on so-called NEAT Properties, which express requirements and characteristics throughout the NEAT System. Each property is a key-value tuple with additional metadata indicating the priority (mandatory or optional) and weight of the associated attribute.

Policies and profiles—stored in the PIB—extend and modify the property set associated with each connection candidate.

In addition, the CIB repository maintains information about available interfaces, supported protocols towards previously-accessed destination endpoints, network properties and current/previous connections between endpoints. The content of the CIB is continuously updated by local and external CIB sources.

Selection components choose an appropriate transport solution. The additional information provided by the NEAT User API enables the NEAT Library to move beyond the constraints of the traditional socket API, making the stack aware of what is actually *desired* or *required* by the application. On the basis of both the information provided by the NEAT User API and the PM, candidate transport solutions are identified. The candidate solutions are then tested by the Selection components, and the one deemed most appropriate is then used.

Transport components are responsible for providing functions to instantiate a transport service for a particular traffic flow. They provide a set of transport protocols and other necessary components to realize a transport service. While the choice of transport protocols is handled by the Selection components, the Transport components are responsible for configuring and managing the selected transport protocols.

Signaling & Handover components can provide advisory signaling to complement the functions of the Transport components. This could include communication with middleboxes, support for handover, failover and other mechanisms.

Figure 2 illustrates a simplified workflow, showing how the NEAT components interact when an application initiates a new flow. As follows from the above description, NEAT has an evolvable architecture that opens up for the introduction of new transport services and can enable interaction with network devices to improve such services. NEAT also enables the incremental introduction of new transport protocols, both in the kernel and in user space, as the API is independent from the underlying transport protocol.

IV. Benefits of NEAT

Next, we present four examples of key benefits of using the NEAT Library. First, NEAT provides an API that is simple to use. This allows existing applications to be easily ported to the NEAT Library, simplifying network communication and reducing code complexity.

NEAT also provides automatic fallback using a *Happy Eyeballs* (HE) mechanism. HE is a generic term for algorithms that test for end-to-end support of a protocol X simply by trying to *use* X, then falling back to a default choice Y known to work if X is found to not work (e.g., after a suitable timeout). This added functionality is lightweight and has negligible cost compared to other communication tasks. It allows applications to take advantage of the best available transport solution and in turn enables transport innovation (e.g., applications do not need to be recoded to use a new transport feature or protocol that becomes available).

NEAT not only facilitates evolution of the transport protocols and introduction of new transport mechanisms, it can also help enable innovation at the network layer. The higher-level of abstraction offered by the NEAT User API eases the path to utilizing Quality of Service (QoS) support for UDP-based applications, and could be used to access other network services should they become available (e.g., selection of the most cost-effective or secure path utilizing IPv6 provisioning-domain information). Applications and networks can also leverage the flexible control provided by the Policy components, for example to provide a generic interface for exchanging information between external SDN controllers and NEAT-enabled applications.

A. Porting applications to NEAT

The NEAT User API offers a uniform way to access networking functionality, independent from the underlying network protocol or operating system. Many common network programming tasks like address resolution, buffer management, encryption, connection establishment and handling are built into the NEAT Library and can be used by any application that uses NEAT.

Developers write applications using the asynchronous and non-blocking NEAT User API, implemented using the libuv [3] library which provides asynchronous I/O across multiple-platforms.

As shown in Listing 1, users can request the services that they expect from the network (e.g. low latency, reliable delivery, a specific TCP congestion control algorithm) by providing an optional set of properties to control the behavior of the library.

```
static neat_error_code
on_connected(struct neat_flow_operations *ops)
{
// set callbacks to write and read data
ops->on_writable    = on_writable;
ops->on_all_written = on_all_written;
ops->on_readable    = on_readable;
neat_set_operations(ops->ctx, ops->flow, ops);
return NEAT_OK;
}

int
main(int argc, char *argv[])
{
// initialization of basic NEAT structures
struct neat_ctx *ctx;
struct neat_flow *flow;
struct neat_flow_operations ops;
ctx  = neat_init_ctx();
flow = neat_new_flow(ctx);
memset(&ops, 0, sizeof(ops));

// callback when connection is established
ops.on_connected = on_connected;
neat_set_operations(ctx, flow, &ops);

// optional user requirements in JSON format
static char *properties = "{\"transport\":[\"SCTP\", \"TCP\"]}";
neat_set_property(ctx, flow, properties);

// connect
if (neat_open(ctx, flow, "127.0.0.1", 5000, NULL, 0)) {
fprintf(stderr, "neat_open failed\n");
return EXIT_FAILURE;
}

// start libuv loop
neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

neat_free_ctx(ctx);

return EXIT_SUCCESS;
}
```

Listing 1. Code example from a simple client using the NEAT API.

The NEAT Library then uses a set of internal components to establish a connection over the network. To make an

appropriate selection, the Policy Manager maps user properties to policies and computes a set of candidate transports that can satisfy the request. NEAT also can utilize policy information directly set by the user, system administrator or developer.

Connections to a peer endpoint are made by creating a new flow, which is a bidirectional link between two endpoints similar to a socket in the traditional Berkeley Socket API but not strictly tied to an underlying transport protocol.

The NEAT API executes callbacks in the application when an event from the underlying transport happens, creating a more natural and less error-prone way of network programming than with the traditional socket API. The three most important callbacks in the NEAT API are `on_connected`, called once the flow has connected to a remote endpoint; `on_readable` and `on_writable`, called once data may be written to or read from the flow.

Our experience with NEAT shows a reduction of the code size by ≈ 20% for each application, as the library streamlines a number of connection establishment steps. For example, the single function call `neat_open` requests name resolution and all other functions required before communication can start, hiding complex boilerplate code. Ported applications remain fully interoperable with regular TCP/IP-based implementations, while being able to take advantage of NEAT functions. Besides, they can benefit from support for alternative transports, when available, relieving programmers from dealing with fallbacks between protocols. Finally, a traditional socket-based shim layer has been implemented on top of NEAT to allow legacy applications to make use of NEAT functionalities through policies without requiring direct porting to the NEAT API.

B. Happy Eyeballs: A Lightweight Transport Selection Mechanism

Selection components employ a HE mechanism to enable a source host to determine whether a transport protocol is supported along the current network path. This allows applications to benefit from advances in transports that may be only partially deployed in the Internet. The HE mechanism used by NEAT is similar to that introduced to facilitate IPv6 adoption [4], but works at the transport layer to select one of a set of connection-oriented transport solutions. The Selection components receive a ranked list of potential candidates generated by the PM, where a higher ranking indicates a better match with application and policy requirements. The HE mechanism then concurrently tries each transport solution from the list, delaying initiation of lower-priority transport solutions.

Figure 3 shows the HE mechanism in a scenario where the best transport to the destination is unknown and current policy dictates that the HE process is used to select between TCP and SCTP, but preferring SCTP. The initiation of the TCP connection is delayed for a time interval governed by policy, specifying a difference in priority between candidate protocols. If the SCTP connection does not complete within the time interval, a TCP connection is also started. The first transport to complete a connection is selected and becomes the transport

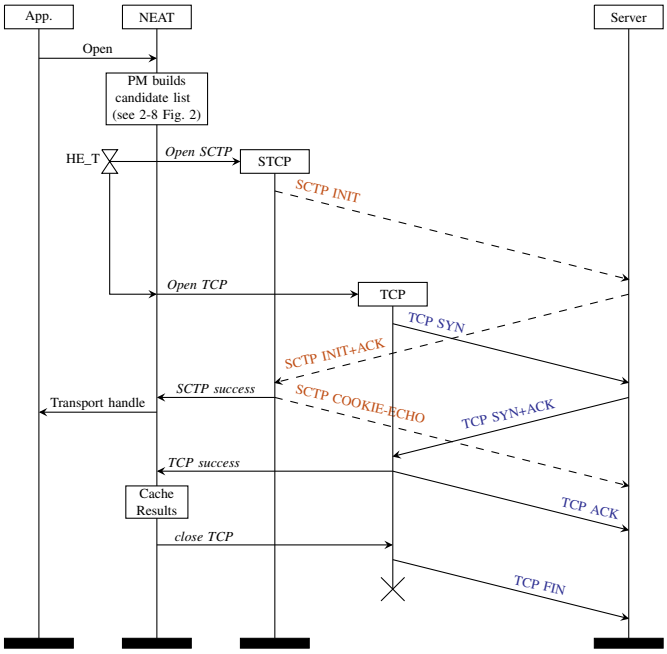


Fig. 3. Message Sequence Chart (MSC) illustrating the NEAT Happy Eyeballs (HE) transport selection process when selecting between TCP and SCTP, SCTP preferred.

of choice. Once connectivity is established, other methods are abandoned, and their connections closed.

To avoid wasting network resources by routinely attempting concurrent connections, HE instructs the Policy components to cache the outcome of each selection result in the CIB for a configurable amount of time. After expiry of the time, the selection is removed from the cache, re-enabling HE.

Consider the scenario in Figure 3. Attempting selection when there is no existing cache entry requires extra resources, potentially resulting in opening connections for each candidate transport protocol. In this example, SCTP completes first and the TCP connection is closed having sent no data. With typical web traffic and worst-case packet sizes, byte overhead is as small as ≈ 1%. For a cache hit rate of 80%, this reduces further to ≈ 0.2%. A detailed evaluation of the impact of HE in terms of memory and CPU utilization can be found in [5], where it is shown that CPU costs are relatively small (especially when considering the cost of TLS encryption), and that HE has only a minor impact on memory consumption.

C. Deployable QoS with NEAT

Network QoS is often used for traffic engineering, but few applications have managed to exploit this technology beyond a controlled network environment. One major obstacle is the lack of a consistent high-level API.

There have been attempts to add methods that directly associate QoS with IP traffic (e.g., [6], [7]), but they have seen little to no adoption. A key challenge is how to express the service requirements, while still enabling policy to influence choice and providing flexibility when the network is unable to directly satisfy the requirements.

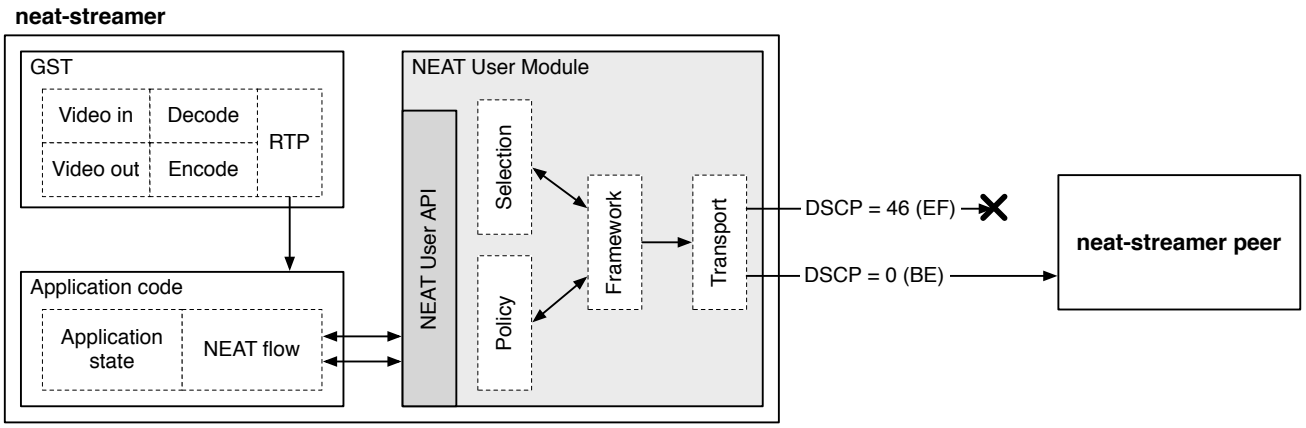


Fig. 4. Example of neat-streamer using QoS fallback with NEAT. The application sets up the media pipeline and uses NEAT to transfer data across the network, according to the requested service. The NEAT Library could try to send UDP datagrams with a DSCP set to High Priority Expedited Forwarding (EF). A timer triggers the NEAT Library to query the application status, which then reveals the application failed to use this DSCP, so NEAT can now try the next DSCP value, Default (BE). When the timer again triggers, the application reports success and this code point continues to be used.

The NEAT API can allow applications to specify QoS requirements. This can, for example, utilize policy information to drive an appropriate Differentiated Service Code Point (DSCP). The finally chosen DSCP can be based on both static policy and dynamic information collected from connections using NEAT.

The NEAT fallback mechanism can be used with any datagram services to enable the NEAT Library to select between a list of candidate datagram transports, network encapsulations and interfaces. This can assist an application to robustly find desirable connection parameters for any path by transparently falling back to alternative services when required (resembling, but different to the NEAT HE function for connection-oriented transports).

Neat-streamer [8] is a demo application that utilizes the NEAT Library for live streaming video over connectionless transports using the GStreamer (GST) media libraries. GST is a pipeline-based media system that supports a wide range of audio and video formats and other functions via a plugin system.

Figure 4 shows the interactions between NEAT and neat-streamer running on a network that drops traffic with certain DSCP values set.

Because neat-streamer uses NEAT, it can indicate the QoS treatment that it requires for each media flow, and the endpoint to which it wishes to stream. NEAT provides the required QoS marking and may determine which transport service to use (e.g., choosing between UDP-Lite, UDP, or use of Traversal Using Relays around NAT, TURN), and whether security functions are required.

NEAT also provides the protocol machinery to update the selected flow parameters should network connectivity problems be reported by the application. A timer triggers a callback function within the application to determine whether the application believes the network is delivering the service it requires (in many cases, only the application is aware of the performance reported by a remote datagram receiver). When an application reports failure it can allow NEAT to use the list of candidates, and potentially other information (e.g., held within the CIB) to search for alternate parameters.

D. SDN Integration

The ability of enabling external sources to query and augment the state of the Policy Manager is a key design choice of the NEAT architecture. As a consequence, NEAT-enabled end-hosts can be seamlessly integrated in centrally controlled environments, such as Software-Defined Networks (SDNs). In such environments, logically centralized controllers aim to maintain a global view of the network and optimize its utilization. To achieve this, controllers ideally require detailed and up-to-date knowledge of available resources, in addition to the requirements and characteristics of deployed applications. Today, controllers rely on time-consuming and error-prone heuristics to infer the association between applications, their requirements, and observed flows.

In this context, the benefit of the NEAT approach is threefold. Firstly, NEAT applications may inform controllers directly about their particular requirements towards the network. In NEAT, such requirements are defined either explicitly by application developers, or through suitable system policies. This strategy can reduce the need for network controllers to guess how to treat individual flows. Secondly, through the Policy Manager CIB, NEAT enables controllers to supply applications with detailed information about the state of paths available to the host. In the absence of this feedback, metrics such as available bandwidth or latency may need to be inferred individually by each application through measurements. Finally, the controller gains the ability to deploy policies at the host level which influence the transport protocols, interfaces and associated parameters used in NEAT applications.

All mechanisms necessary for exchanging information between the controller and NEAT-enabled applications are implemented in Policy components. Specifically, the Policy Interface is exposed through a REST API, enabling external entities to push information to the PIB and CIB and query their contents. As a result, for each flow request created by a NEAT application, the Policy Manager will utilize the latest policies

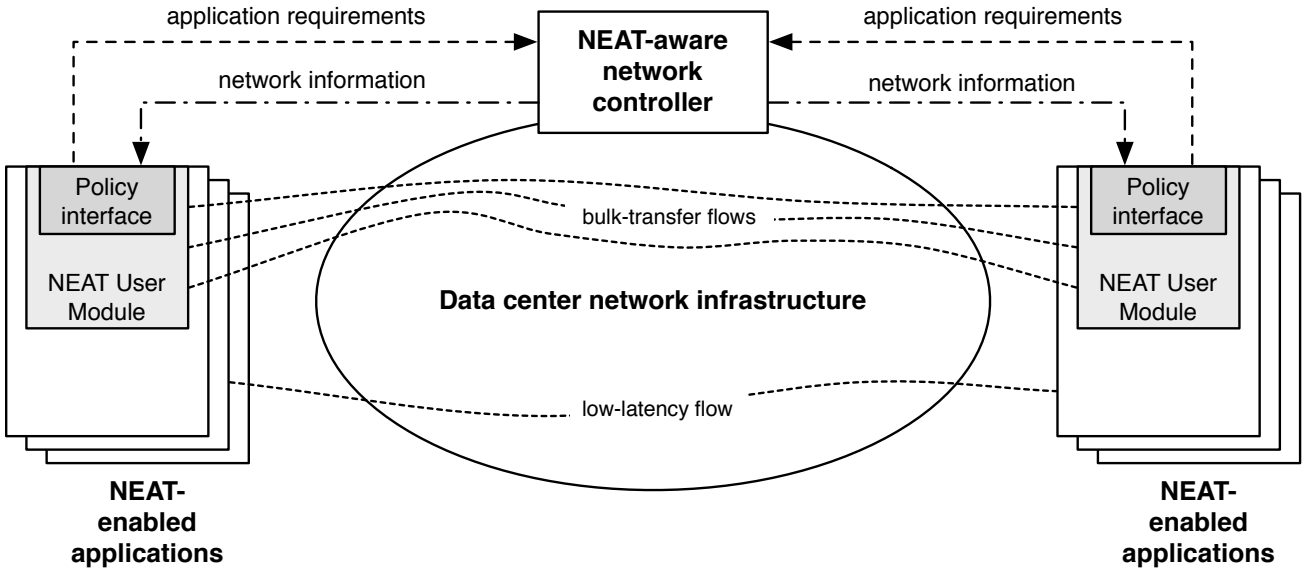


Fig. 5. SDN architecture in which a controller uses the NEAT Library to supply end-hosts with information about the available network resources, and to collect information about the application requirements.

and network attributes supplied by a controller to select the most suitable connection option. Similarly, the API allows the controller to query the CIB to identify the requirements associated with specific application flows or relevant policies configured in the PIB.

To demonstrate the feasibility of the aforementioned controller integration we have implemented a scenario comprised of NEAT-enabled hosts deployed in an OpenFlow SDN network. The aim of the scenario, depicted in Figure 5, is to enable a controller to steer the handling of bulk traffic flows. Each host runs a NEAT-enabled data replication application which provides the estimated flow size as part of the NEAT API call. We used the OpenDaylight framework to implement a controller which monitors the network utilization and calculates a data volume threshold above which flows are considered as bulk flows. We implemented a northbound API and periodically publish a policy to the NEAT end hosts. The policy is triggered when the flow size exceeds the threshold and forces the flows to be tagged with a predefined DSCP marking. As a result, flows affected by the policy are routed through a pre-provisioned network path.

V. Standardization

Recognizing the need for the transport layer (socket) interface to become protocol-independent, the IETF chartered a working group called "Transport Services" (TAPS) in September 2014. A common approach in prior work was to start analysis based on the needs of applications. Instead, TAPS used a methodology that started from a survey of the services offered by available IETF transport protocols [9]. It is currently documenting the primitives and parameters used to access features of a subset of these protocols [10] to form a basis for the design of a protocol-independent API. NEAT developers have been actively contributing to this initiative based on experience of using the NEAT API, which shares many of the goals behind development of TAPS.

The working group is now shortening the list of transport features. Examples of features include "Specify ECN field" or "Choice between unordered (potentially faster) or ordered delivery of messages". A recent contribution by NEAT developers [11] recommends against exposing a transport feature in the API when either choosing or configuring it requires knowledge specific to the network path or the operating system, but not the application. A final step will eliminate features specific to a particular protocol that cannot reasonably be implemented using a different protocol—such features contradict the main purpose of TAPS, to be protocol-independent. At the end of this process, this will result in a subset of transport features that end systems supporting TAPS need to provide. NEAT implements all services specified in the current TAPS documents and may therefore be regarded as a prototype implementation of TAPS.

TAPS is also chartered to define experimental support mechanisms, for example to select and engage an appropriate protocol and discover the set of protocols available for a selected service between a given pair of endpoints, to allow the operating system to choose between protocols (e.g., HE and application-level feedback mechanisms). This approach of breaking the binding between applications and transport protocols is an important final step for TAPS.

VI. Conclusion

The service needs of today's Internet applications range well beyond the basic ones provided by TCP and UDP. Yet, the Internet's transport layer, as it presents itself to a developer via the socket API, has remained unchanged. This has led to per-application (and per-company) developments in user space, over UDP, such as QUIC for Google Chrome. While these new UDP-based transport protocols have recently

pushed the transport layer into the spotlight, they are also only silo solutions which do nothing to solve the architectural ossification problem: the socket API binds applications to protocols at design time—therefore, transport protocols cannot be replaced without changing applications.

In this paper we presented the NEAT Library, which lets application developers access features of transport protocols in a simple and uniform way. NEAT helps freeing developers from platform or protocol dependencies; they do not have to worry about the specifics of each protocol or operating system; they also do not need to worry about whether a protocol works on a given path. Underneath the NEAT User API, new protocols can seamlessly be inserted, automatically yielding benefits to the application on top. With NEAT's clear layer separation, the Internet's transport layer can finally evolve again.

At the time of writing, prototype code for all component types has been developed for several Unix-like OSs. Besides neat-streamer, the NEAT development team has ported example applications to NEAT for early testing, including the Nghttp2 [12] web server and client, several smaller applications like HTTP/HTTPS clients and performance measurement tools; also, a NEAT-supported Firefox implementation is currently under development by Mozilla. NEAT is an open-source project that welcomes contributions. Source code, documentation and implementation status can be found on GitHub [13].

ACKNOWLEDGMENT

The authors would like to thank the anonymous reviewers for their useful remarks.

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).

REFERENCES

[1] L. Eggert and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers," RFC 5405 (Best Current Practice), Internet Engineering Task Force, Nov. 2008, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc5405.txt

[2] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[3] libuv — Cross-platform Asynchronous I/O. Accessed on February 23, 2017. [Online]. Available: https://libuv.org/

[4] D. Wing and A. Yourtchenko, "Happy Eyeballs: Success with Dual-Stack Hosts," RFC 6555 (Proposed Standard), Internet Engineering Task Force, Apr. 2012, accessed on February 23, 2017. [Online]. Available: http://www.ietf.org/rfc/rfc6555.txt

[5] G. Papastergiou, K.-J. Grinnemo, A. Brunstrom, D. Ros, M. Tüxen, N. Khademi, and P. Hurtig, "On the Cost of Using Happy Eyeballs for Transport Protocol Selection," in *Proceedings of the 2016 Applied Networking Research Workshop (ANRW)*. Berlin: ACM, Jul. 2016, pp. 45–51.

[6] H. Abbasi, C. Poellabauer, K. Schwan, G. Losik, and R. West, "A Quality-of-service Enhanced Socket API in GNU/Linux," in *4th Real-Time Linux Workshop*, 2002, accessed on February 23, 2017. [Online]. Available: https://www.osadl.org/fileadmin/events/rtlws-2002/proc/g08_abbasi.pdf

[7] P. Gomes Soares, Y. Yemini, and D. Florissi, "QoSockets: A New Extension to the Sockets API for End-to-end Application QoS Management," *Computer Networks*, vol. 35, no. 1, pp. 57–76, 2001.

[8] Neat-streamer Video Workload Tool. Accessed on February 23, 2017. [Online]. Available: https://github.com/uoaerg/neat-streamer

[9] G. Fairhurst, B. Trammell, and M. Kühlewind, "Services provided by IETF transport protocols and congestion control mechanisms," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-transports-11, Sep. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-11

[10] M. Welzl, M. Tüxen, and N. Khademi, "On the Usage of Transport Service Features Provided by IETF Transport Protocols," Internet Engineering Task Force, Internet-Draft draft-ietf-taps-transports-usage-01, Jul. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage-01

[11] S. Gjessing and M. Welzl, "A Minimal Set of Transport Services for TAPS Systems," Internet Engineering Task Force, Internet-Draft draft-gjessing-taps-minset-03, Oct. 2016, work in Progress. Accessed on February 23, 2017. [Online]. Available: https://tools.ietf.org/html/draft-gjessing-taps-minset-03

[12] T. Tsujikawa. Nghttp2: HTTP/2 C Library. https://nghttp2.org/. Accessed on February 23, 2017.

[13] NEAT GitHub public repository. Accessed on February 23, 2017. [Online]. Available: https://github.com/NEAT-project/neat