Towards a Flexible Internet Transport Layer Architecture

Karl-Johan Grinnemo[†], Tom Jones[⋆], Gorry Fairhurst[⋆], David Ros[‡], Anna Brunstrom[†] and Per Hurtig[†]
[†]Karlstad University, Karlstad, Sweden
{karl-johan.grinnemo, anna.brunstrom, per.hurtig}@kau.se
[⋆]University of Aberdeen, Aberdeen, U.K.
{tom, gorry}@erg.abdn.ac.uk
[‡]Simula Research Laboratory, Oslo, Norway
dros@simula.no

***Abstract*—There is a growing concern that the Internet transport layer has become less adaptive to the requirements of new applications, and that further evolution has become very difficult. This is because a fundamental assumption no longer holds: it can no longer be assumed that the transport layer is only in the scope of end-hosts. The success of TCP and UDP and the ubiquity of middleboxes have led to *ossification* of both the network infrastructure and the API presented to applications. This has led to the development of workarounds and point solutions that fail to cover many facets of the problem. To address this issue, this paper identifies requirements for a new transport layer and then proposes a conceptual architecture that we argue is both flexible and evolvable. This new architecture requires that applications interface to the transport at a higher abstraction level, where an application can express communication preferences via a new richer API. Protocol machinery can use this information to decide which of the available transport protocols is used. By placing the protocol machinery in the transport layer, the new architecture can allow for new protocols to be deployed and enable evolution of the transport layer.**

***Index Terms*—Transport layer, ossification, application-aware networking, transport API, Internet architecture.**

I. Introduction

By all measures, the current Internet is a huge success. One of the major reasons for its success is its evolvable architecture, and thus its ability to adapt to the requirements of new applications. During more than 40 years of existence, the Internet has undergone several fundamental changes. In the early 1980s the Network Control Protocol (NCP) was replaced with the Transmission Control Protocol (TCP [1]), the Domain Name System (DNS) was deployed, and link-state routing protocols came into existence. In the early 1990s, classful addressing was replaced by Classless Inter-Domain Routing (CIDR). However, in later years, very few changes to the Internet architecture have garnered global acceptance, and there is a growing concern that the transport layer has become ossified, where further developments are hard or impossible to deploy globally.

The opportunity to introduce new transports has disappeared. This seems to be the experience for all new transport protocols that have been designed. Even protocols implemented in networking stacks, such as UDP-Lite [2], SCTP [3], and DCCP [4], have still failed to be widely used across the Internet. This is largely due to NATs, firewalls, and various other middleboxes. The design of middleboxes has focused on widely deployed transport protocols and applications and as a result middleboxes tend to filter out all traffic that is not TCP or UDP. Some only pass port 80 (HTTP) or port 443 (HTTPS), or protocols that they have been directly configured to support.

The transport API offered by operating systems has also become ossified and unable to adapt to the needs of novel applications. The ubiquity of TCP and UDP has meant that APIs typically tie applications to a priori choices of protocol (either TCP or UDP), and so applications have become hard-coded to use a specific transport—decided at design time.

Even simple connectivity is sometimes hard, when it differs from what is considered the norm by network equipment designers and those who configure the equipment. Peer-to-peer connectivity (e.g., for voice or multi-party interaction) requires applications to introduce complicated workarounds, such as ICE [5], to determine what protocols actually work across a path. Although there may be common methods, the solutions are built into individual applications.

Facing this daunting world, transport protocol designers have more recently proposed development of application-layer transport protocols that use the current underlying transport layer as a communication substrate, e.g., QUIC [6] that uses UDP as a substrate, and the Minion suite of transport protocols [7] that use TCP or UDP as substrates. Still, these protocols are in many ways only point solutions: QUIC primarily targets web traffic and Minion datagram services. SPUD [8] is a new initiative that also tries to use a UDP substrate to enable faster transport evolution.

This paper argues that there needs to be a more complete solution to the ossification problem. It proposes a design that can decouple applications from the choice of actual transport protocol being used, and enable applications to explicitly communicate their service requirements via a new transport API, with the transport protocol and options being selected at *run-time*. This change opens up for a more adaptive transport solution that enables new network and transport functions and services to be added incrementally and transparently. Our proposal builds on previous work that has recognized the value

978-1-4673-9882-4/16/$31.00

of a higher layer transport interface (e.g., [9], [10]) where it can improve use of the network layer and take advantage of capabilities offered by different network paths. We extend this approach to a more complete solution that can enable transport protocol evolution across the Internet.

The remainder of the paper is organized as follows. Section II reviews previous and ongoing efforts to revitalize the Internet transport architecture. Section III presents an example of a point solution as a way of highlighting possible improvements, but also deficiencies of a piecemeal approach to ossification. Next, in Section IV, we discuss the requirements that need to be fulfilled by a new transport system. A conceptual architecture addressing these issues is detailed in Section V. The paper concludes in Section VI with a brief discussion of how we intend to continue our efforts in designing and developing this system.

II. Point Solutions to Transport Ossification

The slow pace at which new transport services can be introduced in the Internet has become a major concern in the research community for at least the past decade [11]–[15]. Early on, virtual network overlays were seen as a promising solution to the ossification of the Internet transport architecture. For example, Peterson et al. [16] suggested using an overlay-based testbed in which legacy and novel services run concurrently in separate so-called slices. An even bolder proposal was put forward by Turner and Taylor [13], who proposed using virtual overlay techniques to transform the Internet into a network of meta-networks, with each meta-network using the existing Internet architecture as a substrate on top of which meta-links and routers could be created.

Although virtual network overlays are not without virtues, they hide the underlying network from the transport. They also deny the opportunity to interact with the network (e.g., to take advantage of differences in the offered network services, or infer up-to-date overlay link information), which adds to ossification. These methods typically also need to send probing messages to neighboring nodes. The overhead incurred depends on both the topology and the employed routing protocol, but can be substantial [17]. The processing overhead for packet encapsulation, decapsulation, and demultiplexing adds to total overhead. Virtual overlays basically relegate the current Internet to become a communication substrate, imposing homogeneity on a diverse network service—providing an obstacle to evolution of different network-transport interactions.

A less drastic solution to the problem of ossification is to use application-layer transport enhancements, such as middleware. This can enable distributed applications to communicate with little or no knowledge of the underlying network infrastructure and protocols. A large and varied selection of communication middleware has been proposed. Early examples include CORBA [18] and traditional RPC toolkits. These can be fairly complex to use and incur a considerable overhead, especially when used for performance-sensitive traffic. Two more recent examples are the Windows Communication Foundation (WCF) [19] of the .NET Framework [20], and the communication components of the Java Enterprise Edition (J2EE) [21]. Both WCF and the J2EE communication parts have limited scope, and primarily target web and transactional services.

ZeroMQ [22] is an example of a more general middleware. This appears to an application as an extension of the traditional socket API adding support for several communication patterns such as publish-subscribe and pipeline. ZeroMQ has support for message framing and queueing, automatic error recovery, and other common middleware features. Still, although ZeroMQ has broad support for transport protocols its focus is largely on TCP.

The last decade has seen a renewed interest in application-level transport protocols, another category of application-level transport enhancements. One early example is the uTorrent transport protocol (uTP) [23], motivated by allowing BitTorrent clients to better share capacity with standard TCP applications, while at the same time enabling utilization of unused capacity. More recent examples include SPDY [24] and QUIC (Quick UDP Internet Connections) [6], both emanating from Google's research to make the Web faster. SPDY was designed to be a low-latency replacement for HTTP. It addressed the poor support for pipelining and prioritization in HTTP, its inability to send compressed headers, and its lack of resource push capabilities, becoming the foundation for HTTP/2 [25]. SPDY runs over TCP, and thus has to live with TCP's relatively long connection-establishment times, vulnerabilities to head-of-line blocking, and less than ideal congestion control, with mixed performance – sometimes it is able to reduce page load times, however, sometimes it has the opposite effect [26]. QUIC was designed in response to the deficiencies of SPDY, and is essentially a UDP-based transport protocol that incorporates the functionality of both SPDY and TCP, as well as support for Transport Layer Security (TLS) [27]. The main argument against uTP, SPDY, QUIC, and other application-level transport protocols is that they primarily target one particular application or category of applications, and that they are less useful, or even impossible to use, by other applications: uTP works well for delay-insensitive but not for -sensitive traffic, and SPDY and QUIC are not intended for use by streaming-media applications.

The Minion suite of transport protocols [7] was designed as a framework for the development of TCP wire-compatible transport services that offer unordered message delivery. The uTCP extension adds unordered delivery primitives to standard TCP. At present, two application-level transport protocols have been designed on top of uTCP: uCOBS, a lightweight datagram delivery service, and uTLS, an encrypted version. These protocols still function properly over standard TCP, and thus support incremental deployment. Despite its attractive features, three years after conception, the Minion suite has yet to see any wide-scale use. One reason could be that uTCP, arguably crucial to the success of Minion, is OS dependent.

Multipath TCP (MPTCP [28]) is an extension to standard TCP that enables multiple network paths to be simultaneously

used by a single transport connection. Applications can employ MPTCP without modifications to the socket API calls. The design of MPTCP had to work around the ossification of the Internet transport architecture [12], and MPTCP is therefore designed to look like standard TCP on the wire. In recent years, MPTCP has earned quite a bit of interest, not least due to Apple's use of MPTCP for its Siri personal assistant service. Still, it is rather obvious from the huge effort necessary to design and implement MPTCP as part of standard TCP that MPTCP is not demonstrating a way forward to a more flexible and adaptive transport layer.

SCTP, UDP-Lite and DCCP are three protocols that struggle with deployment, since to achieve broad adoption they would require support within middleboxes in the network, and such support is unlikely to emerge in general equipment unless there is wide-scale use. This vicious circle of dependency prevents any evolution path. One approach could be to encapsulate the protocols over UDP, and such methods have been standardized for SCTP and DCCP [29], [30].

A different approach to ossification is to extend the socket API. Over the years, several socket API extensions have been proposed. For example, msocket [31] enables an application to access multiple distinct protocol stacks, something not possible via the socket API. Another, more comprehensive, extension is Sockets++ [32], which addresses a range of limitations, including support for multipoint connections and multi-streaming, i.e., connections that comprise several concurrent streams.

Some proposed extensions to the socket API seek to enable applications to express their service requirements to the network stack. One example is multi-sockets [9], which introduces the idea of intentional networking. Applications use labels to communicate their intent to the multi-sockets library, which maps these intents to networks with particular characteristics. The application intents are qualitative rather than quantitative, e.g., they could inform the multi-sockets library whether a transmission is to be considered background or foreground traffic. Another example of a more expressive socket API is Socket Intents [10]. In the same way as multi-sockets, Socket Intents makes it possible for applications to supply information about their traffic that can enable better selection of the network service. However, in contrast to multi-sockets, Socket Intents uses socket options instead of labels. It also takes a broader scope on intentional networking, and lets socket options not only determine the choice of network interface, but also how socket parameters should be tuned. When a host is multihomed, Socket Intents relies upon the DNS being consistent between network interfaces.

Our proposed transport system builds to a large extent on the idea of intentional networking and Socket Intents, however, takes a much broader scope. In the same way as Socket Intents, our transport system offers a user API that enables applications to express their communication preferences, however, unlike Socket Intents, our transport system explicitly considers additional requirements deemed necessary for a future-proof transport system, including: deployability, flexibility, evolvability, and portability. Notably, our transport system provides mechanisms that handle middlebox traversal, and allow the incremental introduction of new transport protocols and services. Also, our transport system is designed in such a way that it is portable across different platforms and operating systems.

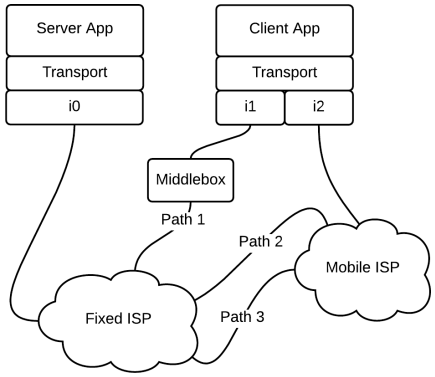

Fig. 1. Emulated network for web browsing over multiple interfaces.

III. Point Solution: An Example of Issues

This section provides an example of how a more expressive API can be used to enable better network decisions, and why it is an important prerequisite to releasing the protocol stack from the limitations presented by ossification. However, a more expressive API is a necessary but not a sufficient solution to the ossification problem.

We consider two simple experiments using a framework based on Socket Intents. These experiments show the advantage of such a system when using a device with multiple network interfaces.

Our experiments used settings similar to those in [10]. Two gigabit Ethernet interfaces (i1, i2) are used to make requests to a common web server (i0), each connected via a traffic shaper running dummynet (represented by network clouds in Fig. 1). The client, web server and traffic shaper ran FreeBSD. Interface i1 had a low latency and low capacity (representative of a typical DSL line with 10ms RTT, 6 Mbit/s downlink, 768 Kbit/s uplink) and i2 had high latency, but high capacity (representing a lightly loaded LTE link with 70ms RTT, 12 Mbit/s downlink, 6 Mbit/s uplink). Competing traffic was created using a Python program to download a large file on the same interface for the i1 and i2 single interface tests and on the bulk interface (i2) in the multi-interface test.

Content was replicated on the server from the front page of the New York Times (NYT, nytimes.com) and an internal discovery page on Flickr (flickr.com/explore). The Flickr page contained 31 objects, with sizes in the range from 13 Kbytes to 589 Kbytes and a mean of 134 Kbytes. The NYT Page contained 135 objects with sizes in the range from 1 Kbyte to 211 Kbytes and a mean of 30 Kbytes.

Ideally, time-dependent web requests would be sent using the lowest latency interface (providing a smooth browsing experience), but any media files would be requested over the higher bandwidth bulk interface (reducing download time).

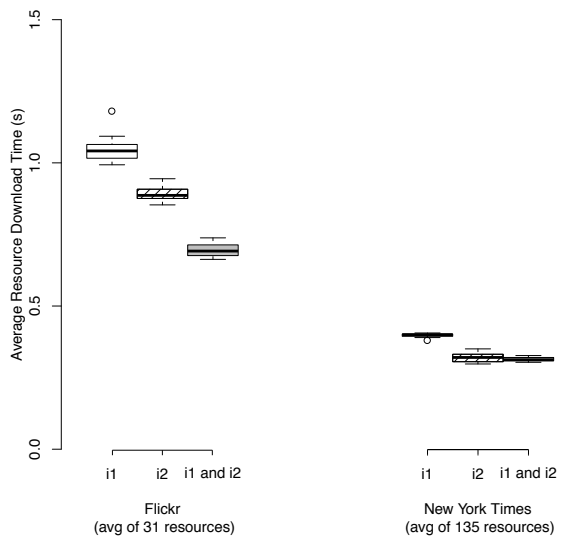

Fig. 2. Box plot showing average resource download times for different strategies.

We tried to verify the results in [10] by independently implementing the Socket Intents interface selection mechanism which allows traffic to be identified as time-dependent or bulk. A Python program was used to evaluate each URL retrieved by the client: if the resource was of type png or jpeg, it was considered a media file and directed over the high capacity interface i2. If not, it was sent over interface i1. This directed traffic to be sent using the most appropriate interface. Socket Intents and our Python program are configured with the interface properties to aide with selecting the interface to use for each request.

We measured the total time consumed to download all the resources on the page using a single http connection. For simplicity we present the time per object. Browsers using HTTP/1.1 typically use parallel multiple connections to a server, and content providers often use methods such as sharding to distribute content across multiple servers. This makes the ideal minimum page download time a function of the number of parallel connections, but the actual download time will be impacted by the design of the browser, the distribution of object sizes and server layout.

Fig. 2 is a box plot of the average download times for each of the resources on each site over each of the interfaces, also considering taking advantage of both interfaces. The experiments show that even small changes in the transport system can significantly improve download time, compared to a design that could only take advantage of one interface at a time. We can see that the simple multiple interface binding shows an average improvement of 20% over the i2 only case for the Flickr page and similar performance against just with i2 for the NYT page. This difference can be explained by the larger variation in range between the web requests on the Flickr page than on the NYT page.

While we do expect to see a growing number of devices with multiple interfaces (each with local link characteristics, like i1 and i2 in Fig. 1), we also expect to see the emergence of multiple *services* across an interface—such as QoS differentiation, and the use of multiple IPv6 prefixes linked to different operator SLAs, e.g., allowing traffic from i2 to be routed on different network paths (paths 2 and 3 in Fig. 1). However, this is only a part of the problem that needs to be addressed in a real-world setting. The practice of using DNS redirection to route to local content means that resources that have been looked up on one interface may not be local to the network offered by another interface. Unless the system is able to understand the actual context of each interface, this can lead to suboptimal routing, adding to latency and network load. The actual problem is more than simply making the best choice based on the interface properties.

Furthermore, there are opportunities to significantly improve performance by using a transport protocol that is more appropriate for web page downloads than the default UDP or TCP transports, e.g., QUIC, SCTP, or TCP Minion. However, this would require a method to determine the set of transports supported at both the local and the destination endpoints. This could be found by probing the remote endpoint and seeing if it supports the alternate transport, but what actually matters is that the path over which the packets are sent can support the new transport. A middlebox on the path (e.g., path 1 in Fig. 1) could pass all these protocols, or may block any of them. If applications had to change to use these protocols, they would need to implement probing and discovery mechanisms—but we argue that such methods are better placed below the transport interface, where they can also take advantage of the evolution of the transport stack, and of shared information about the success/failure of previous attempts by this and other applications to use each known path. Choosing the best network path and the most appropriate transport therefore becomes a multi-dimensional selection problem.

IV. Transport Requirements

This section presents the requirements for a new approach to the design of an Internet transport system that is flexible and evolvable. From the example in the previous section it is clear that a seemingly easy task such as choosing the best network path, in reality is a very hard problem. The requirements for a new transport system must therefore be well designed to offer a remedy to the present ossification problem.

a) Deployable: A key requirement is that any new transport system is itself deployable. Its design must be independent of the operating system and particular network technologies. As an implication of this, it should not rely on protocols or features that are only available on certain host operating systems, or expect applications to include special mechanisms. Instead, these should be an integral part of the transport system itself.

b) Evolvable: A transport system must be evolvable and permit independent evolution of its components, allowing new

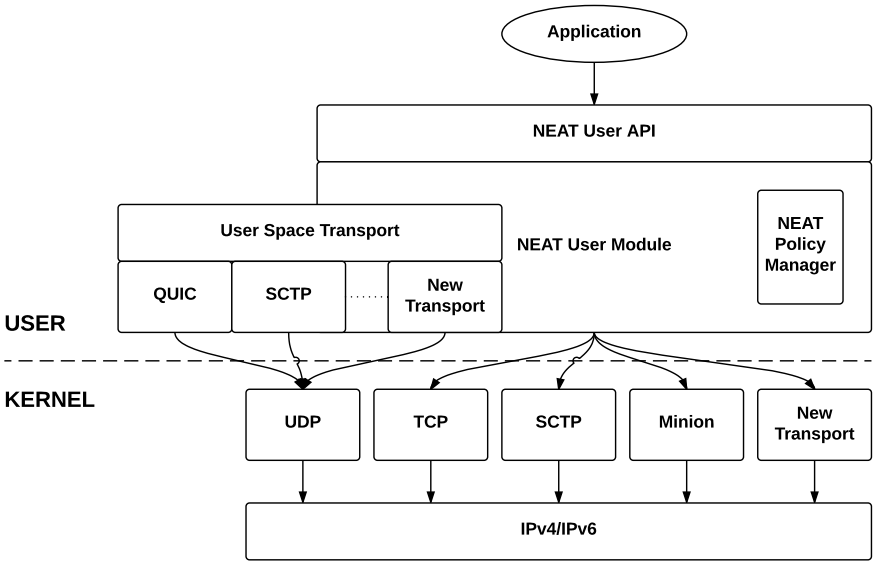

Fig. 3. The NEAT Architecture.

mechanisms and protocols to be added as required, and for applications to benefit from new features as infrastructure evolves. We recognize that middleboxes have become an integral part of the Internet [33], [34], and a transport system has to be able to discover whether any new solution works across a middlebox-encumbered path, and if necessary fall back to a protocol that can work across this path, allowing incremental deployment.

c) Present a new API: In analyzing the protocol architecture, it has become clear that the goals can not be met unless the API to the transport layer evolves. A new API must to offer a higher abstraction than the present sockets API, ideally allowing applications to specify their *transport service* needs, rather than configuring a specific transport protocol instance, e.g., if the instance is stream-oriented or transactional, if it is bursty or generates data at a fairly constant rate, etc. This allows the transport system to have the option of deciding which transport protocol to use and how it should be configured.

V. The NEAT System

Currently available solutions are essentially point or partway solutions aimed at fixing specific issues (addressing one or several of the requirements). We do not know of any existing single architecture that encompasses all the requirements to fix the current "transport logjam" [7]. In isolation, these solutions do not provide a way to revitalize the Internet architecture so that it again can become *flexible* and *evolvable*.

We believe that the correct approach is to provide a comprehensive *transport architecture*, which we have called a New, Evolutive API and Transport-layer architecture for the Internet (NEAT). An overview of the NEAT architecture is given in Fig. 3. NEAT offers an enhanced API for applications to access *transport services*. NEAT includes a set of protocol mechanisms that takes care of middlebox traversal and protocol selection, accompanied with a fallback service when paths are incapable of supporting the chosen protocol. NEAT has an evolvable architecture that opens up for new transport services and can enable interaction with network devices to improve the transport service. NEAT also enables the incremental introduction of new transport protocols: both in the kernel and in user space.

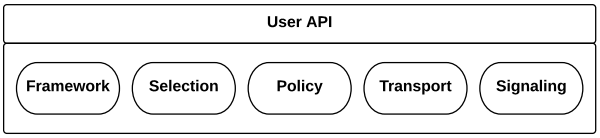

Fig. 4. The NEAT User Module.

The User Module is designed to be portable across different operating systems and network stacks. It comprises five groups of components (see Fig. 4): *Framework*, *Selection*, *Policy*, *Transport*, and *Signaling*.

The Framework components provide the functionality required to use the NEAT System. They define the structure of the User API and interfaces to the logic that implements the core mechanisms. Applications provide information about the requirements for a desired transport service via the API. The framework also includes components for diagnostic, debugging and measurement. The additional information reported by NEAT can be used to identify which components in NEAT are in use, and how components have been configured.

The Selection components choose an appropriate transport endpoint and a set of candidate transport components. The additional API information enables the NEAT System to move beyond the constraints of the traditional socket API, making the stack aware of what is actually desired or required for each traffic flow. This is combined with inputs from a

Policy Manager (see below). After identification of candidate services, it will test the suitability of the candidates, utilizing information known about the path, and by attempting to make endpoint connections. Parts of this algorithm will be performed in parallel to avoid unnecessary delay.

The Policy components comprise the Policy Information Base (PIB), the Characteristics Information Base (CIB), and the Policy Manager. The PIB is a repository that contains a collection of policies, where each policy consists of a set of rules linking a set of matching requirements to a set of preferred or mandatory transport characteristics. Policies can be added by the system administrator, external entities or applications, and have different priorities. The CIB is a repository storing information about available interfaces, supported protocols, network properties and current/previous connections between endpoints. Some mechanisms to populate the CIB are already implemented in operating systems as statistics/measurement tools and will be made available as default CIB sources. Another class of CIB sources will be provided by the NEAT Selection components which will store discovered transport protocols and parameters supported along paths in the CIB for future reuse. External CIB sources may be provided by device and operating system vendors or third parties developing modules for active or passive measurements, statistics and metadata collection.

The Policy Manager enables a set of rules to specify the transport protocol to use for a particular transport service, the configuration of the selected transport protocol, etc., implemented in the PIB. In contrast, acquired knowledge about endpoints and the network, evolves over time, and is stored in the CIB. This enables the Policy Manager to cache characteristics allowing future flows to benefit from previous flow experience. Tests have already been made with the Mozilla Firefox web browser that clearly suggest that caching flow characteristics would be useful. Particularly, in a field trial by Mozilla, they observed a hit rate in their internal 'route' cache of more than 80%, suggesting frequent requests for the same web objects.

Together, Selection and Policy enable applications to be designed and implemented to be oblivious to the transport protocols of a particular platform. They facilitate NEAT to make an appropriate decision based on application requirements and what is made available for a network endpoint. Making these decisions at run time rather than at application-design time, ensures that an appropriate choice is made, provides opportunities to consider multiple, possibly conflicting, constraints, and avoids each application having to code for the possibility that a path does not support a particular mechanism or combination of mechanisms.

The Transport components are responsible for providing functions to instantiate the transport service for a particular flow in NEAT. Transport provides a set of transport protocols, e.g., TCP, UDP, SCTP, TLS, DTLS, etc., and other components to realize a transport service (such as priority handling). While the selection of transport protocols are handled by the Selection components, the Transport components are responsible for configuring and managing the transport protocols.

The Signaling components can provide advisory signaling to complement the functions of Transport. This could include communication with middleboxes, support for failover and handover and other mechanisms. There are two types of signaling: transport and network. Transport signaling enables the exchange of capability information between NEAT endpoints, and provides input to the Policy Manager concerning what capabilities a peer endpoint supports. Network signaling communicates with devices along the network path. This could be as simple as the distribution of Differentiated Services Code Points, signals that help up-speed or down-speed for transports, through to something as complex as QoS negotiation.

Placing optional network signaling below the transport API can allow applications to indicate how/whether they would like to use network signaling, without requiring each application to be updated each time a new signaling protocol is introduced. The NEAT system can determine the appropriate signaling mechanisms to use on a path to a particular endpoint. Freeing applications from choosing and supporting signaling protocols is expected to reduce the barriers to introducing new mechanisms in the network, allowing signaling messages to be exchanged with network devices as such new signaling protocols emerge and become supported across the network.

Just as a higher-level transport API decouples transport and interface selection (e.g., to offer potential for services to evolve independently of applications), so also there can be benefits in the way that transports interact with the network. For example, when UDP is used to encapsulate a transport that does not have widespread network support (e.g., SCTP, or any new transport), the NEAT system can select one of a number of signaling protocols (e.g., PCP [35]) to coordinate with local mappings in a firewall/NAT ensuring that the local network device keeps state while a transport protocol connection remains active. Similarly, a future transport protocol could benefit from signaling received about a path's congestion, disruption, or some other characteristic—all of which can be interpreted within the context of the transport connection. The higher abstraction of the transport API therefore not only enables flexibility, it can provide a path to future evolution of the transport layer.

VI. Conclusion

This paper has examined the requirements for a new transport system and proposed a conceptual architecture, NEAT, which we suggest can break the current ossification of the Internet transport architecture, enabling incremental flexible deployment and use of new transport services and features. In this architecture, applications will interface through an enhanced API that decouples them from the choice of transport protocol and network features to be used. The enhanced API provides information about traffic requirements, and NEAT combines this with pre-specified policies, and measured network conditions; the transport system then chooses appropriate transport protocols and features. Protocol machinery within the transport stack manages functions such as middlebox traversal, and can be extended to interact with network devices to

enable specific transport services, and provide a pathway to creation of new services. Still, NEAT is not without challenges. A key challenge for any new service-oriented interface to reach significant impact is the need for industry uptake and standardization of the interface. The ongoing IETF work within the Transport Services working group (TAPS) shows the current interest in this regard [36]–[38].

We are currently implementing the main parts of the NEAT system, and use this to explore performance in actual networks. For example, feature prototypes for the discovery of available transport services have been developed, and work is ongoing to design a NEAT policy management component. The source code to our NEAT system implementation is available on GitHub [39].

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] J. Postel, "Transmission Control Protocol," RFC 793 (Internet Standard), Internet Engineering Task Force, Sep. 1981, Updated by RFCs 1122, 3168, 6093, 6528. [Online]. Available: http://www.ietf.org/rfc/rfc793.txt

[2] L.-A. Larzon, M. Degermark, S. Pink, L.-E. Jonsson, and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)," RFC 3828 (Proposed Standard), Internet Engineering Task Force, Jul. 2004, Updated by RFC 6335. [Online]. Available: http://www.ietf.org/rfc/rfc3828.txt

[3] R. Stewart, "Stream Control Transmission Protocol," RFC 4960 (Proposed Standard), Internet Engineering Task Force, Sep. 2007, Updated by RFCs 6096, 6335, 7053. [Online]. Available: http://www.ietf.org/rfc/rfc4960.txt

[4] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340 (Proposed Standard), Internet Engineering Task Force, Mar. 2006, Updated by RFCs 5595, 5596, 6335, 6773. [Online]. Available: http://www.ietf.org/rfc/rfc4340.txt

[5] J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal for Offer/Answer Protocols," RFC 5245 (Proposed Standard), Internet Engineering Task Force, Apr. 2010, Updated by RFC 6336. [Online]. Available: http://www.ietf.org/rfc/rfc5245.txt

[6] J. Roskind, "Quick UDP Internet Connections," Google, Tech. Rep., Apr. 2012.

[7] M. F. Nolan, N. Tiwari, J. Iyengar, S. O. Amin, and B. Ford, "Fitting Square Pegs Through Round Pipes: Unordered Delivery Wire-Compatible with TCP and TLS," in *USENIX NSDI*, San Jose, CA, U.S., Apr. 2012.

[8] M. Kuehlewind and B. Trammell, "SPUD Use Cases," Jul. 2015.

[9] B. D. Higgins, A. Reda, T. Alperovich, and J. Flinn, "Intentional Networking: Opportunistic Exploitation of Mobile Network Diversity," in *The* 16^{th} *Annual International Conference on Mobile Computing and Networking*, Chicago, IL, U.S., Sep. 2010.

[10] P. S. Schmidt, T. Enghardt, R. Khalili, and A. Feldmann, "Socket Intents: Leveraging Application Awareness for Multi-Access Connectivity," in *ACM CoNEXT*, Santa Barbara, CA, U.S., Dec. 2013.

[11] M. Handley, "Why the Internet only just works," *BT Technology Journal*, vol. 24, pp. 119–129, Jul. 2006.

[12] M. Honda, Y. Nishida, C. Raiciu, M. Greenhalgh, M. Handley, and H. Tokuda, "Is it still possible to extend TCP?" in *ACM IMC*, Berlin, Germany, Nov. 2011.

[13] J. S. Turner and D. E. Taylor, "Diversifying the Internet," in *IEEE Global Telecommunications Conference*, St. Louis, MO, U.S., Nov. 2005.

[14] L. M. Correia, H. Abramowicz, and K. Wünstel, Eds., *Architecture and Design for the Future Internet*. Dordrecht, Heidelberg, London, New York: Springer, 2011.

[15] M. Welzl, S. Jörer, and S. Gjessing, "Towards a Protocol-Independent Transport API," in *IEEE ICC*, Kyoto, Japan, Jun. 2011.

[16] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology in the Internet," *ACM SIGCOMM Comp. Comm. Review*, vol. 33, pp. 59–64, Jan. 2003.

[17] Z. Li and P. Mohapatra, "The Impact of Topology on Overlay Routing Service," in *IEEE INFOCOM*, Hong Kong, Mar. 2004.

[18] Object Management Group (OMG), "Common Object Request Broker Architecture (CORBA) Specification, Version 3.3," Nov. 2012.

[19] Microsoft Inc. Windows Communication Foundation (WCF). https://msdn.microsoft.com/en-us/library/hh128109.aspx.

[20] ——. Overview of the .NET Framework. https://msdn.microsoft.com/en-us/library/zw4w595w.aspx.

[21] Oracle Inc. Java EE at a Glance. http://www.oracle.com/technetwork/java/javaee/overview/index.html.

[22] iMatix. ZeroMQ – Distributed Messaging. http://zeromq.org.

[23] A. Norberg. uTorrent Transport Protocol. http://www.bittorrent.org/beps/bep_0029.html.

[24] M. Belshe and R. Peon, "SPDY Protocol," Feb. 2012.

[25] M. Belshe, R. Peon, and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)," RFC 7540 (Proposed Standard), Internet Engineering Task Force, May 2015. [Online]. Available: http://www.ietf.org/rfc/rfc7540.txt

[26] Y. Elkhatib, G. Tyson, and M. Welzl, "Can SPDY really make the web faster?" in *IFIP Networking*, vol. 41, no. 4, 2014, pp. 1–9.

[27] T. Dierks and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2," RFC 5246 (Proposed Standard), Internet Engineering Task Force, Aug. 2008, Updated by RFCs 5746, 5878, 6176, 7465, 7507, 7568, 7627, 7685. [Online]. Available: http://www.ietf.org/rfc/rfc5246.txt

[28] A. Ford, C. Raiciu, M. Handley, and O. Bonaventure, "TCP Extensions for Multipath Operation with Multiple Addresses," RFC 6824 (Experimental), Internet Engineering Task Force, Jan. 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6824.txt

[29] M. Tuexen and R. Stewart, "UDP Encapsulation of Stream Control Transmission Protocol (SCTP) Packets for End-Host to End-Host Communication," RFC 6951 (Proposed Standard), Internet Engineering Task Force, May 2013. [Online]. Available: http://www.ietf.org/rfc/rfc6951.txt

[30] T. Phelan, G. Fairhurst, and C. Perkins, "DCCP-UDP: A Datagram Congestion Control Protocol UDP Encapsulation for NAT Traversal," RFC 6773 (Proposed Standard), Internet Engineering Task Force, Nov. 2012. [Online]. Available: http://www.ietf.org/rfc/rfc6773.txt

[31] R. Davoli and M. Goldweber, "msocket: Multiple Stack Support for the Berkeley Socket API," in *The* 27^{th} *Symposium On Applied Computing*, Riva Del Garda, Trento, Italy, Mar. 2012.

[32] S. Böcking, "Sockets++: A Uniform Application Programming Interface for Basic-Level Communication Services," *IEEE Communications Magazine*, vol. 34, pp. 114–123, Dec. 1996.

[33] J. Sherry, S. Hasan, C. Scott, A. Krishnamurthy, S. Ratnasamy, and V. Sekar, "Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service," *ACM SIGCOMM Comp. Comm. Review*, vol. 42, no. 4, pp. 13–24, 2012.

[34] Z. Wang, Z. Qian, Q. Xu, Z. Mao, and M. Zhang, "An Untold Story of Middleboxes in Cellular Networks," in *ACM SIGCOMM Comp. Comm. Review*, vol. 41, no. 4. ACM, 2011, pp. 374–385.

[35] D. Wing, S. Cheshire, M. Boucadair, R. Penno, and P. Selkirk, "Port Control Protocol (PCP)," RFC 6887 (Proposed Standard), Internet Engineering Task Force, Apr. 2013, Updated by RFCs 7488, 7652. [Online]. Available: http://www.ietf.org/rfc/rfc6887.txt

[36] IETF. Transport Services (TAPS) Charter. https://datatracker.ietf.org/wg/taps/charter/.

[37] G. Fairhurst (ed.), B. Trammell (ed.), and M. Kuehlewind (ed.), "Services provided by IETF transport protocols and congestion control mechanisms," Internet Draft draft-ietf-taps-transports, work in progress, Jan. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports

[38] M. Welzl, M. Tüxen, and N. Khademi, "On the Usage of Transport Service Features Provided by IETF Transport Protocols," Internet Draft draft-ietf-taps-transports-usage, Work in Progress, Jan. 2016. [Online]. Available: https://tools.ietf.org/html/draft-ietf-taps-transports-usage

[39] NEAT, "NEAT Project," https://github.com/NEAT-project, 2016.