Raising the Datagram API to Support Transport Protocol Evolution

Tom Jones, Gorry Fairhurst
University of Aberdeen, Aberdeen, U.K.
Email: {tom, gorry}@erg.abdn.ac.uk

Colin Perkins
University of Glasgow, Glasgow, U.K.
Email: csp@csperkins.org

***Abstract*—**
Some application developers can wield huge resources to build new transport protocols, for these developers the present UDP Socket API is perfectly fine. They have access to large test beds and sophisticated tools. Many developers do not have these resources. This paper presents a new high-level Datagram API that is for everyone else, this has an advantage of offering a clear evolutionary path to support new requirements. This new API is needed to move forward the base of the system, allowing developers with limited resources to evolve their applications while accessing new network services.

I. Introduction

The Berkeley Sockets Application Programming Interface (API) is the main interface to the network for developers. It has been hugely successful, with few changes to its core semantics over its 35 year history. The API has scaled to support applications running services over networks that could not have been envisaged during its inception, at a scale that could not have been imagined.

Application protocols using the socket API may choose to build on top of a stream protocol, running over TCP, or a datagram protocol using User Datagram Protocol (UDP). UDP is the simplest transport [1], offering a minimal protocol over IP, with service multiplexing with port numbers and optional checksums with best effort unreliable delivery. The UDP socket API provides a very simple interface for applications to send and receive datagrams, with the ability to control the options/parameters required to build applications [2].

While this model has been a success, the socket API is now showing its age. It is becoming clear that it does not offer a clear evolutionary path to support new requirements, as needs of applications change, and as the network changes beneath. There are two major areas where this has become visible. First, as more sophisticated applications are developed, and as the complexity of the network grows, we increasingly see that the Datagram socket API does not provide a sufficiently expressive interface. For some applications, the connection-less unreliable datagram service is a core feature. Others would prefer more transport support, but must use UDP because they require partial reliability, control of transmission timing, non-standard congestion control, multicast, or any of the other features that are only possible with UDP in the present API. The API is increasingly baroque (e.g., using obscure `setsockopt()` calls to control important semantics) but also overly simplistic, pushing applications to implement transport features themselves [3].

ISBN 978-3-901882-94-4

Secondly, there is an increasing trend to see UDP not as a transport protocol, but as a *demultiplexing substrate* layer that supports the deployment of new transport protocols [4] [3]. This is a reaction to ossification of the network: the intended transport demultiplexing point is the Protocol field in the IPv4 header, or the Next Header field in IPv6, but this in unusable in practice since use of values identifying transports other than TCP and UDP will result in firewalls dropping the packet. Accordingly, the transport demultiplex is moving up the stack, with dynamic binding of identifiers to transport protocols using UDP port numbers negotiated by out-of-band signalling [5].

We face new sophisticated applications [4] driving a growing volume of UDP traffic in the Internet, and the emergence of UDP as a core protocol for evolving datagram transport [6], an important question emerges: is the current network transport API fit for purpose?

This paper explores whether the simple UDP socket API is sufficient for the next step in evolution of Internet Transport, or whether applications can benefit from a higher-level API that builds upon thirty years experience of using the network. We examine whether this new API could open-up access to network functions (such as Quality of Service (QoS), Explicit Congestion Notifications (ECN), control of packet size) help enable session level functions (such as path selection for multi-homing, mobility and firewall punching), and greater support for fault reporting in increasingly complex network topologies.

This work is part of a larger effort designing, implementing and attempting to deploy new transport services and APIs. This includes the IETF Transport Services (TAPS) working group, defining and documenting the transport services available for applications; the IETF QUIC and RTCWEB working groups, defining new transports running over UDP; and research projects such as the EU NEAT [7] project that is building a system to enable transport evolution and ease deployment.

The next section identifies some issues applications have using the UDP Socket API. Section III examines how changes in the protocol stack above and below the current API can help applications evolve. Section IV presents an API to address the issues raised. Section V discusses how enabling evolution is only possible with a new API. The paper concludes in Sections VI and VII with a brief look to the future and discussion.

II. Background

The Socket API closely models the file system API. Calls to `send` and `receive` are mapped to performing `read` and `write` calls on the socket for the network connection. Datagram-orientated protocols are modeled as atomic `read` and `write` socket operations that either succeed or fail depending on the buffer size. UDP is offered in this API as either a connected or unconnected transport, the default unconnected state allows a sender to send datagrams to an IP address. Connecting a UDP socket causes the socket to pass ICMP errors up to the application. Connections have no side effects on the wire, offering only a shortcut to applications by using the explicit connected address for datagrams [3].

The UDP API offers only a few methods to access its minimal services. Applications can create a socket, look up a host, connect, set options and send and receive data, represented by the pseudo-code for a typical client in Listing 1.

```
int main()
{
  int sockfd, rv, numbytes;
  struct addrinfo hints, *servinfo, *p;

  hints.ai_family = AF_UNSPEC;
  hints.ai_socktype = SOCK_DGRAM;
  if ((rv = getaddrinfo(argv[1], SPORT, &hints, &
      servinfo)) != 0) {
    fprintf(stderr, "getaddrinfo: %s\n",
        gai_strerror(rv));
    return 1;
  }
  while (true) {
    if ((numbytes = sendto(sockfd, "hello",
        strlen("hello"), 0,
      p->ai_addr, p->ai_addrlen)) == -1) {
      perror("talker: sendto");
      exit(1);
    }
    if ((numbytes = recvfrom(sockfd, buf,
        MAXBUFLEN-1 , 0,
              (struct sockaddr *)&their_addr, &
                  addr_len)) == -1) {
      perror("recvfrom");
      exit(1);
    }
  }
  close(sockfd);
  return 0;
}
```

Listing 1. Example of a client application using the UDP Socket API. (The example client, looks up the remote host, chooses an IP address and settles into a loop of sending and receiving data until the application completes.)

An application may modify protocol options via the `setsockopt`/`getsockopt` API calls. These provide the only way to interact with the lower networking stack. Commonly used options allow control of the differentiated services code point (DSCP) used, setting the ECN field, setting the hop count for the IP datagram, the link maximum transmission unit (MTU), and the "don't fragment" (DF) bit in the IP header.

The `setsockopt` API allows applications to set options, but provides no mechanism for discovering whether they will work and no path for falling back to options known to always work. This can make it dangerous to use QoS or ECN: if the application has to provide fallback code it is more likely it will stick to a safe set of values. Further, the set of options has evolved over time, and is inconsistent between platforms and often presents variants of the same function. This complicates application portability between platforms. Furthermore, the same `setsockopt` API is used to control features that are semantically *not* socket options [2]. For example, the `IP_ADD_MEMBERSHIP` option triggers an IGMP join of a multicast group, with semantic closer to that of `connect()`.

A. What features are missing from the Sockets API?

Establishing Connectivity: TCP-based applications tend to use one of a small number of protocols, e.g, HTTP, SSH, FTP, and typically run in a client-server manner. The `connect()`, `listen()`, and `accept()` API fits this use case cleanly, and is straightforward for Network Address Port Translation (NAPT) or firewall traversal: ports are opened in response to outgoing connection establishment packets, for the 5-tuple representing the connection; the traffic is inspected to ensure it looks like the corresponding protocol; and the connection is closed when a FIN is seen (or after a timeout).

There are a diverse set of Applications built on UDP that need themeselves to perform some form of connection establishment. Many more protocols are in use, communication patterns are more varied and often peer-to-peer, and the stateless nature of the transport protocol means that middleboxes that track transport protocol state to maintain holes must resort to using timers to keep the firewall open. This environment makes it likely that UDP-based applications will encounter connectivity issues. This is especially true when the remote endpoint is a peer that is also behind a NAPT. One solution to this problem uses the combination of STUN [8] to determine NAPT binding and probe connectivity, TURN relays as dynamically configured proxies for UDP [9] or TCP [10] flows, the Interactive Connectivity Establishment (ICE) algorithm [11] to categorise network impediments and systematically probe connectivity, and a relayed signalling protocol to rendezvous with the remote host and exchange candidate addresses for connectivity. While the signalling is likely inherently application specific, there is scope to implement the other functions generically, as a *path layer* below the socket API, rather than have each application implement the entire complex NAPT traversal stack.[1]

Support for multiple interfaces: An application running on a multihomed host has to account for the presence of multiple interfaces and that those interfaces will vary in properties and connectivity over time (mobility). The present API does not make it easy for applications to discover the local interfaces or their properties, it requires application level methodologies to discover what is working. Applications must determine whether network information gathered on one interface is valid on another interface. Issues can arise with locality if name servers are used across domains for accessing resources. DNS resolution on multihomed systems can also be problematic [13]. The interface used for name resolution (`getaddrinfo()`) does not support multiple interfaces, and

[1]This can be viewed as a generalisation of the happy eyeballs connection racing technique [12] used by TCP applications to probe IPv4 and IPv6 connectivity. That, too, would benefit from a consistent implementation in the socket layer.

complications can result with geographic load balancing and applications that require mobility.

Control over quality of service and reliability: TCP provides a reliable, ordered, byte stream service, that is subject to head-of-line blocking while waiting for retransmissions of lost packets. There is no portable way to inspect the receive buffer, or access data out of order [14]. When needed, UDP-based applications must implement (partial) reliability above the API. The developer has to take responsibility for building a solid network system [3].

Congestion control: For TCP, congestion control is assumed to take place below the Sockets API, and there is no interface to select the algorithm, or query congeston state. For UDP-based application, congestion control must be implemented above the API, with no support from the socket.

In summary, the socket API has been a companion for developers writing application protocols for decades, but the interface is starting to show its age. It provides a poor API for many important features, and requires applications to implement other features in their entirity. To address these issues libraries can be integrated into an application, but integrating such a library requires modifications to the code base, and the libraries event model has to be made compatible with the application. Supporting each and any new feature means the application has to integrate more with libraries that help support them, with a corresponding increase in complexity and maintenance costs. A new, standard, API is needed.

III. Raise the Datagram API

The current socket API is too low level. Even applications that need direct access to the network can benefit from a higher level Datagram API. By placing commonly needed functions below this API, applications can specify what they want from the stack, but allow the system below the API to perform the actions needed to realise a service. However, it is not immediately obvious what of the set of functions identified in Section II-A should lie below a new Datagram API, and which should remain in the application. On the one hand, part of the success of UDP was an API that enables application choice of how to interact with the network. On the other hand, applications increasingly require solutions to the same set of problems and implementing these below the API can benefit from wider context of the network paths and interfaces, and allow mechanisms to evolve independently of applications.

We use the following principles to guide our choice of which functions should be placed below the Datagram API:

1) An application using the new API that does nothing new, should be able to at least receive similar service to that of the socket API.
2) Commonly needed functions should be placed below the API when these are automatable (do not require application decisions).
3) Functions where the preference can be expressed as a policy can also be placed below the API.
4) Functions that rely on application algorithms or detailed knowledge of trade-offs relating to data should be implemented above the API [3] (e.g., choice of codec to meet congestion control constraints in a conferencing application, or how to trade loss vs. capacity constraints).

A richer API should allow an application to request a set of abstract properties for the transport service it desires (e.g., requiring a datagram service, whether high capacity is needed, whether there is benefit from low latency, whether low cost is preferred, etc). Understanding application needs can help a Datagram API because it can then automate functions that are hard for an application to optimise.

A. Below the Datagram API

A higher level API can reduce the volume of code required to build an Internet application, it can also significantly reduce the complexity the application has to manage, providing a starting point to automate appropriate choices below the API.

The system below the API needs to interpret properties from the application together with system wide properties. Turning these into concrete actions requires a policy system to select protocol mechanisms, help discover interfaces and inform parameter choices. For example, a video streaming application could request properties that indicate a minimum capacity required for the datagram service and QoS preferences to minimise latency while constraining cost. Listing 2 shows an example JSON policy file that indicates a QoS Live Video precedence.

```
{
    "transport": [
    {
      "value": "Datagram", "precedence": 1
    },
  ],
  "qos": [
    {
      "value": "Interactive Video", "precedence":
          1
    },
    {
      "value": "Live Video", "precedence": 2
    }
  ]
  "network": [
    {
      "value": "cost", "precedence": 1
    },
    {
      "value": "capacity", "precedence": 2
    }
  ]
}
```

Listing 2. Example JSON file describing a NEAT Abstract Policy

To provide network context for functions below the API, information needs to be gathered about the properties of network paths, and network service interfaces. This knowledge base can be related to policy and application requirements to enable the application to rely on the system making good choices about how to use the network. At the simplest level, this implies understanding of available network service interfaces – by gathering information (e.g., MTU, line rate, address) about local physical and virtual interfaces (e.g., across tunnels or source addresses that bind to provisioning domains).

TCP maintains information about the paths that have been used from an endpoint, and similar data may be collected for use by UDP – such as the path MTU, capacity recently used,

etc.). This can also help eliminate transport candidates that include protocols that are known not to be supported on a specific path. Further information may be gleaned from the experience of protocols using a path, including experienced round-trip time (RTT) and capacity insight from coupled congestion control [15]. Other functions could also be automated here, such as NAPT keep-alive and black-hole detection, easing the tasks of finding a candidate path, failover between paths, concurrent use of multiple paths.

We also note that application developers and users need to be able to understand the decisions made on behalf of the application. While most of the time it is expected that good decisions will be taken, there is a need to understand why a particular policy or application property resulted in a particular choice. This supports troubleshooting and allows polices to be refined when needed – this in itself is valuable compared to the current information made available by the UDP socket API.

B. Above the Datagram API

We recognise that some functions cannot be easily migrated below the API. While datagram congestion control can benefit from standard mechanisms/algorithms, the details are often linked to application design; applications have to provide their own congestion control. This function is expected to remain above the API. In contrast, the system below the API could offer circuit breaker functions when required to control the envelope of the capacity consumed by an application [3].

NAPT traversal could be automated for simple cases, but many applications need complex processing to finally select amongst a set of transport candidates. This is often complicated by the need to interact with rendezvous points, signalling intermediaries and to understand session-level negotiation dialogues. For these reasons more sophisticated applications are likely to continue to utilise ICE libraries to perform the NAPT traversal. None-the-less the availability of information from below the API (such as speed, cost, reliability) can help select candidates. The automation of path-related functions such as keep-alive and path MTU discovery can eliminate features that otherwise would need to be implemented above the API by an application.

The transport 5-tuple of source IP, port, destination IP, port and transport protocol is used to identify datagrams forming a flow. If an application is multi-homed or mobile between multiple network interfaces the 5-tuple cannot be used to identify the endpoint. Mobility between interfaces requires context (including a connection_ID) beyond individual flows and can outlive transport usage, and as such is primarily an application function, although such mechanisms may take advantage of context information gathered below the API.

C. Traversing the Datagram API

Some functions require cooperation between the application and transport to be effective, and straddle the Datagram API.

An example might be congestion control for an interactive video conference. This has strict timing constraints: audio frames *must* be sent every 20ms and video frames every 1/60th of a second, but there is some flexibility to change *what* is being sent, if not *when*, but this requires cooperation of the media codecs. Real-time performance offers the application to be tightly coupled with the congestion controller, and for *both* the application to respect the congestion constraints and the congestion control to respect application limitations.

IV. The Datagram API for the NEAT System

This section provides a concrete example of some of the API aspects discussed in the paper, based on the open source NEAT System [16], developed as part of the EU NEAT project [17]. Designed as a replacement for the socket API, this provides a one-sided change to the transport API at the sender.

The new API offers applications access to abstract transport services. This allows selection between the available transport protocols including TCP, SCTP, SCTP/UDP, UDP and UDP-Lite via a single unified API. Mechanisms beneath the API, provide many functions including help to discover the set of protocols that may work across an Internet path.

A simple example in Listing 3 illustrates the lifetime of an application using the NEAT System. The application creates a *NEAT context*, within which it then creates a *NEAT flow*, using application policies passed in JSON to describe the abstract properties it requires or desires. The Policy Manager combines these policies with a global configured policy to inform its decisions, e.g., to generate a list of transport candidates. The NEAT Characteristic Information Base (CIB) is populated with information about the network interfaces and paths allowing decisions to also consider network, path and transport statistics.

```
static struct neat_flow_operations ops;
static struct neat_ctx *ctx = NULL;
static struct neat_flow *flow = NULL;

ctx = neat_init_ctx()
flow = neat_new_flow(ctx)
prop = "(see Listing 2)";
neat_set_property(ctx, flow, &prop)
ops.on_writable = on_writable;
ops.on_readable = on_readable;
ops.on_error = on_error;

neat_set_operations(ctx, flow, &ops)
neat_open(ctx, flow, hostname, port)
neat_start_event_loop(ctx, NEAT_RUN_DEFAULT);

static neat_error_code
on_writable( struct neat_flow_operations *opCB)
{
    neat_write(opCB->ctx, opCB->flow, buf)
  return NEAT_OK;
}

static neat_error_code
on_readable( struct neat_flow_operations *opCB)
{
    neat_read(opCB->ctx, opCB->flow, buf)
  return NEAT_OK;
}
```

Listing 3. NEAT Example Application listing

Rather than an imperative polling-based socket API, an application uses a callback-based API to access the NEAT System. It therefore needs to provide a set of callback handlers for each NEAT Flow. In Listing 3, the application sets up the `on_writeable` callback. The application calls `neat_connect` with the name and port of a listening server.

The key difference between Listing 3 and Listing 1, is that the NEAT System performs common network actions automatically on behalf of the application. The example policy, Listing 2, illustrates a high level abstract transport request, that can inform selection of an appropriate DSCP, and help identity transport candidates when multiple network interfaces are active. Once created, a NEAT flow is comparable to a socket, but offers much more utility, using the callback-based API to call an application on network events or data.

For connection-oriented protocols the NEAT System can select an appropriate transport configuration for a flow using Happy-Eyeballs selection logic [18] to choose between transport candidates and instantiate a concrete Operating System socket. The process of selecting a transport is different for a Datagram services that do not have a connection-setup (e.g. UDP or UDP-Lite). It is not possible for the NEAT System to know whether a datagram flow has suffered a connectivity failure (e.g., by expiry of NAPT state, routing changes, or from choice of a DiffServ Code Point that is not available). Such information is only known at the application layer through the reception of receiver-generated feedback messages.

Datagram applications can use the *Happy Applications* mechanism to register a periodic callback. This allows mechanisms below the API to query the application, asking whether it is 'happy' with the progress of a NEAT flow. This allows datagram applications to perform automatic selection and fallback, handled by the NEAT System Because the application decides what makes it 'happy', it can trigger selection mechanisms based on application level criteria. If a certain capacity were requested and latency and the chosen transport candidate was not able to satisfy this, the system now has the correct signals to choose a second transport candidate. On completion, the application may retrieve NEAT Flow parameters to determine the transport state (e.g., addresses, port numbers, DSCP, etc).

The NEAT System leaves implementation of support for mobility and/or ICE to datagram applications. Similarly, libraries to support application-oriented functions, such as the Real Time Protocol (RTP), can be used over the NEAT API.

V. Supporting Protocol Evolution

One key advantage of raising the API is the ability to enable protocol evolution [16] in support of new application requirements. Datagram transport protocols can be developed independently of the kernel, and our more expressive API eases this evolution. It also enables innovation in the use of the network, by easing the introduction of new protocols and mechanisms, and by taking advantage of these when the network becomes available. A particular supporting protocol may only be available in a few networks or supported by limited equipment, which might be insufficient to justify inclusion in every application, but the features it offers may be attractive across a range of application when it happens to be supported, which could justify inclusion below the API.

We consider three examples of mechanisms below the Datagram API: Protocols to supply provisioning domain (PvD [19]) information, protocols to communicate path information (PLUS [20]), and an extension to UDP to permit transport of options. All are work-in-progress in the IETF, available as Internet Drafts, but none are as yet fully specified nor implemented.

There are a growing number of devices that are capable of connecting to multiple network services. These devices may have multiple physical interfaces, and additional could support virtual interfaces (e.g., able to send using multiple IPv6 address prefixes). PvD is an architecture for endpoints in a multiple network interface environment to discover network configuration information. A PvD-aware endpoint can use a protocol to discover authoritative information such as; source address prefixes, DNS server locations, HTTP Proxy location, default gateway address, and could be extended to include characteristics of the service (e.g., maximum capacity available, existence of supported QoS services, cost of using an interface, etc). The new PvD protocols provide a way for an endpoint stack to select transport candidates and also to assist in configuring the protocols for the local network service. In addition, applications could benefit from an interface to the PvD information – in the NEAT System this type of function is provided by the *Policy Manager*.

Path Layer UDP Substrate (PLUS) is a proposed encapsulation header and protocol that provides bi-directional communication over UDP. This is intended to convey selected transport information to middleboxes on an Internet path, even in the face of pervaisive application encryption. The transport-agnostic method assists state management for pinholes through NAPTs, firewalls and other boxes on the network path. Extensions to PLUS can provide path information (e.g. advice on MTU or available capacity) that may help a transport protocol and can inform selection of a suitable transport candidate. PLUS could also evolve to support non-data-related diagnostics, e.g. measure progress of flow, duplication/loss, relative fairness, etc.

Another proposal suggests adding options to UDP [21]. This utilises the UDP length field, in a way resembling its use in UDP-lite, but in this case to provide a field in which options can be attached to any datagram. UDP Options could be utilised by mechanisms below the Datagram API to provide a standard way to communicate control information. This receiver demultiplexing lets the transport extract control information not intended for the application. This would therefore allow the stack to send probes/measurements on behalf of the application, such as using an "echo this data" message for RTT measurement, keep-alive probes, Path MTU probes, etc.

It is too early to tell where there is merit for the community at large in using new service discovery methods (such as PvD), a new encapsulation (such as Plus) or an update to UDP (such as UDP Options). However, we do note that transition to support such transport evolution would be greatly eased by the presence of a higher-level Datagram API.

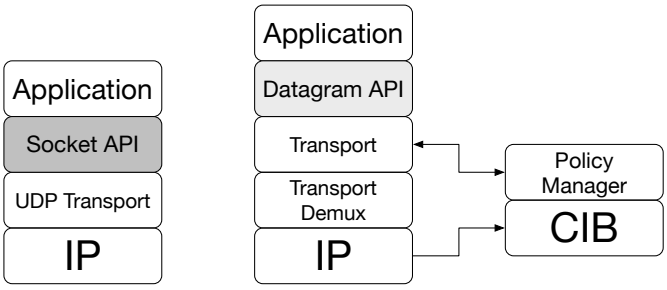

Fig. 1. Stack Evolution: Left, The traditional socket API. Right, A new Datagram API utilising Policy and Transport-layer Demultiplexing

VI. Looking to the Future

The network has become ossified and experience has shown it has been virtually impossible to deploy transport protocols with different IP protocol numbers. TCP development requires modifications in operation system kernels, needing a large effort for the developer to deploy enhancements. The time required to have enough hosts running an enhancement to see a benefit impedes iteration times.

Accordingly, new protocol development is happening on top of UDP (Figure 1, left stack). This has several advantages. First, and most critically, it enables the permissionless end-to-end deployment of new transports: UDP has wide enough deployment that it can be expected to work in most networks. Secondly, a UDP demultiplexing substrate introduces minimal bandwidth and processing overhead. In addition, there is already at least some support in middlebox devices (NAPT, Firewalls) that can be used as a starting point for deployment. Finally, the UDP API is widely supported allowing user-space stacks to directly access the network without requiring special privileges. The latter overcomes the time and effort required to integrate a new transport across a range of operation systems.

The history of the Stream Control Transmission Protocol (SCTP) illustrates the benefits of using UDP as a demultiplexing substrate. SCTP has an assigned IP protocol number (132), and is moderately widely implemented as a native transport, but has seen only limited deployment because it does not pass residential NATs/firewalls. When running over UDP, as the WebRTC data channel [22], however, SCTP has seen worldwide, deployment in web browsers, in part because of ease of implementation in user-space, and in part because it is not blocked by most firewalls/NATs.

Large developers are evolving their applications, we see efforts from Facebook, Google, Apple and others to develop new protocols on top of UDP. The new protocols offer a higher level API to the applications, this API is locked away under layers of application state. If you are not the browser vendor with the new HTTP transport protocol you are playing catch up to remain on a level with their networking stack. Developers that do not have the same wide scale of resources have access only to the socket API, this is not sufficient to continue to evolve on the Internet.

VII. Conclusion

UDP is increasingly playing the role of a demultiplexing substrate layer, dynamically binding the transport protocol to a signalled "port" number. The UDP Sockets API needs to evolve to be less an *application* programming interface, and more a *transport* protocol interface. The usefulness of the present UDP Sockets API has passed. It is time to raise the Datagram API to support transport protocol evolution.

The application interface must migrate up the stack, to provide a higher level of abstraction for applications, while allowing transport flexibility to meet their needs. This provides a substrate for new low-level transport protocol development, while providing the transport services needed by the next generation applications.

Acknowledgment

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] J. Postel, "User datagram protocol," IETF, RFC 768, August 1980.
[2] G. Fairhurst and T. Jones, "Features of the user datagram protocol (UDP) and lightweight UDP (UDP-lite) transport protocols," IETF, Internet-Draft, October 2016.
[3] L. Eggert, G. Fairhurst, and G. Shepherd, "UDP usage guidelines," RFC Editor, BCP 145, March 2017.
[4] J. Iyengar and M. Thomson, "QUIC: A UDP-based multiplexed and secure transport," IETF, Internet-Draft, January 2017.
[5] S. McQuistin and C. S. Perkins, "Reinterpreting the transport protocol stack to embrace ossification," in *Proc. Workshop on Stack Evolution in a Middlebox Internet*. Zürich, Switzerland: IAB, January 2015.
[6] T. Herbert, L. Yong, and O. Zia, "Generic UDP encapsulation," IETF, Internet-Draft, October 2016.
[7] NEAT, "NEAT Project," https://www.neat-project.org, 2017.
[8] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing, "Session traversal utilities for NAT (STUN)," IETF, RFC 5389, October 2008.
[9] R. Mahy, P. Matthews, and J. Rosenberg, "Traversal using relays around NAT (TURN): Relay extensions to session traversal utilities for NAT (STUN)," IETF, RFC 5766, April 2010.
[10] S. Perreault and J. Rosenberg, "Traversal using relays around nat (turn) extensions for tcp allocations," IETF, RFC 6062, November 2010.
[11] J. Rosenberg, "ICE: A protocol for NAT traversal for offer/answer protocols," IETF, RFC 5245, April 2010.
[12] D. Wing and A. Yourchenko, "Happy eyeballs: Success with dual-stack hosts," IETF, April 2012, RFC 6555.
[13] T. Savolainen, J. Kato, and T. Lemon, "Improved recursive dns server selection for multi-interfaced nodes," IETF, RFC 6731, December 2012.
[14] S. McQuistin, C. S. Perkins, and M. Fayed, "TCP Hollywood: An unordered, time-lined, TCP for networked multimedia applications," in *Proc. Networking Conference*. Vienna, Austria: IFIP, May 2016.
[15] S. Islam and M. Welzl, "Start me up: Determining and sharing TCP's initial congestion window," in *Proc. IRTF ANRW*. ACM, 2016.
[16] K.-J. Grinnemo, T. Jones, G. Fairhurst, D. Ros, A. Brunstrom, and P. Hurtig, "Towards a flexible Internet transport layer architecture," in *Proc. LANMAN*. IEEE, jun 2016.
[17] NEAT, "NEAT Source Code," https://github.com/NEAT-project, 2017.
[18] 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 *Proc. IRTF ANRW*. ACM, 2016.
[19] D. Anipko, "Multiple provisioning domain architecture," IETF, RFC 7556, June 2015.
[20] B. Trammell and M. Kuehlewind, "Path layer UDP substrate specification," IETF, Internet-Draft, December 2016.
[21] J. Touch, "Transport options for UDP," IETF, Work in progress, February 2017.
[22] H. Alvestrand, "Transports for WebRTC," IETF, Work in progress, October 2016.