Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 27, 2018

T. Pauly
Apple Inc.

October 24, 2017

Guidelines for Racing During Connection Establishment
draft-pauly-taps-guidelines-01

Abstract

Often, connections created across the Internet have multiple options of how to communicate: address families, specific IP addresses, network attachments, and application and transport protocols. This document describes how an implementation can race multiple options during connection establishment, and expose this functionality through an API.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 27, 2018.

Table of Contents

1. Introduction

Often, connections created across the Internet have multiple options of how to communicate: address families, specific IP addresses, network attachments, and application and transport protocols. If an application chooses to only attempt one of these options, it may fail to connect, or end up using a suboptimal path. If an application chooses to attempt one option after another, waiting for each to fail or time out, a user of the application may need to wait for a very long time before progress is made. And, if an application simultaneously attempts all options, it may unnecessarily consume significant local or network resources.

In order to solve this, applications can employ a method of racing their various connection establishment options. This approach is

commonly used for racing multiple IP address families, the algorithm for which is referred to as "Happy Eyeballs" [I-D.ietf-v6ops-rfc6555bis]. However, the approach can apply more generally.

This document describes how an implementation can race multiple options during connection establishment, and expose this functionality through an API.

2. Terminology

This document uses specific terminology when discussing connection establishment.

2.1. Endpoint

An identifier for a network service. Generally there is a concept of both a local and remote endpoint. Endpoints are the targets of network connections. If an endpoint of a given type cannot be directly used, it should be resolved into one or more endpoints of another type. Examples of endpoint types include:

- IP address + port
- Hostname + port
- Service name + type + domain
- URI

2.2. Derived Endpoint

A derived endpoint is an endpoint that is not the original target of an API client, but an endpoint created from the original endpoint through transformation or lookup. Derivation may take the form of hostname resolution into addresses, synthesis between address types, or changing to a different endpoint entirely based on a configuration requirement. For example, if a proxy server must be used for a connection, the endpoint that represents the proxy is a derived endpoint.

2.3. Path

A view of network properties that can be used to communicate to an endpoint from the current system. This is sometimes referred to as a Provisioning Domain (PvD) [RFC7556]. The path may include properties of the addresses and routes being used, the network interfaces being

used, and other metadata about the network learned from configuration or negotiation.

2.4. Connection

A flow of data between two endpoints. A connection is created with a target remote endpoint, and a set of parameters indicating client preferences for path selection and protocol options.

3. Connection Establishment Overview

The process of establishing a network connection begins when an application expresses intent to communicate with a remote endpoint (along with any constraints or requirements it may have on the connection). The process can be considered complete once there is at least one set of network protocols that have completed any required setup to the point that it can transmit and receive the application's data.

Looking more closely, connection establishment has three required steps that must be performed by some entity on a system:

1. Identifying the endpoint to which the connection should be established

2. Choosing which path or interface to use

3. Conducting the necessary set of protocol handshakes to establish the connection

The most simple example of this process might involve identifying the single IP address to which the application wishes to connect, using the system's current default interface or path, and starting a TCP handshake to establish a stream to the specified IP address. However, each step may also vary depending on the requirements of the connection: if the endpoint is defined as a hostname and port, then there may be multiple resolved addresses that are available; there may also be multiple interfaces or paths available, other than the default system interface; and some protocols may not need any transport handshake to be considered "established" (such as UDP), while other connections may utilize layered protocol handshakes, such as TLS over TCP.

Whenever an application has multiple options for connection establishment, it can view the set of all individual connection establishment options as a single, aggregate connection establishment. The aggregate set conceptually includes every valid combination of endpoints, paths, and protocols. As an example,

consider an application that initiates a TCP connection to a hostname + port endpoint, and has two valid interfaces available (Wi-Fi and LTE). The hostname resolves to a single IPv4 address on the Wi-Fi network, and resolves to the same IPv4 address on the LTE network, as well as a single IPv6 address. The aggregate set of connection establishment options can be viewed as follows:

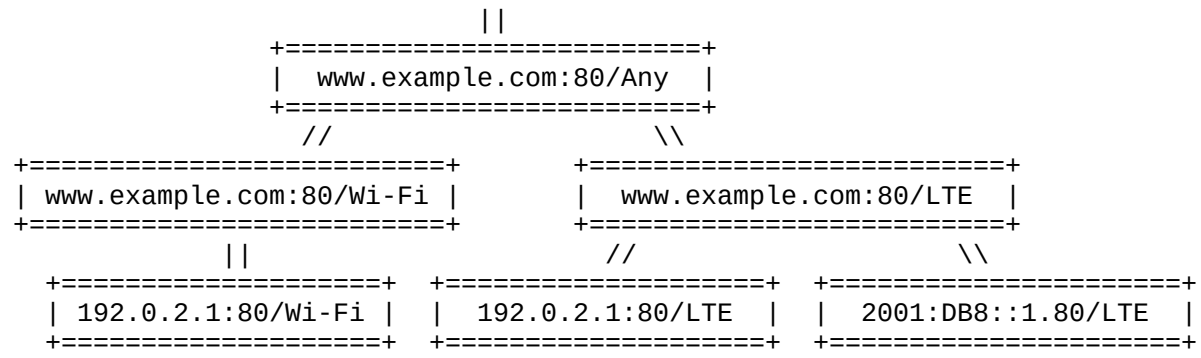
```
Aggregate [Endpoint: www.example.com:80] [Interface: Any]   [Protocol: TCP]
     |-> [Endpoint: 192.0.2.1:80]       [Interface: Wi-Fi] [Protocol: TCP]
     |-> [Endpoint: 192.0.2.1:80]       [Interface: LTE]   [Protocol: TCP]
     |-> [Endpoint: 2001:DB8::1.80]     [Interface: LTE]   [Protocol: TCP]
```

Any one of these sub-entries on the aggregate connection attempt would satisfy the original application intent. The concern of this document is the algorithm defining which of these options to try, when, and in what order.

4. Structuring Options as a Tree

When an implementation responsible for connection establishment needs to consider multiple options, it SHOULD logically structure these options as a hierarchical tree. Each leaf node of the tree represents a single, coherent connection attempt, with an Endpoint, a Path, and a set of protocols that can directly negotiate and send data on the network. Each node in the tree that is not a leaf represents a connection attempt that is either underspecified, or else includes multiple distinct options. For example. when connecting on an IP network, a connection attempt to a hostname and port is underspecified, because the connection attempt requires a resolved IP address as its remote endpoint. In this case, the node represented by the connection attempt to the hostname is a parent node, with child nodes for each IP address. Similarly, an application that is allowed to connect using multiple interfaces will have a parent node of the tree for the decision between the paths, with a branch for each interface.

The example aggregate connection attempt above can be drawn as a tree by grouping the addresses resolved on the same interface into branches:



The rest of this document will use a notation scheme to represent this tree. The parent (or trunk) node of the tree will be represented by a single integer, such as "1". Each child of that node will have an integer that identifies it, from 1 to the number of children. That child node will be uniquely identified by concatenating its integer to it’s parents identifier with a dot in between, such as "1.1" and "1.2". Each node will be summarized by a tuple of three elements: Endpoint, Path, and Protocol. The above example can now be written more succinctly as:

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    1.2.1 [192.0.2.1:80, LTE, TCP]
    1.2.2 [2001:DB8::1.80, LTE, TCP]
```

When an application views this aggregate set of connection attempts as a single connection establishment, it only will use one of the leaf nodes to transfer data. Thus, when a single leaf node becomes ready to use, then the entire connection attempt is ready to use by the application. Another way to represent this is that every leaf node updates the state of its parent node when it becomes ready, until the trunk node of the tree is ready, which then notifies the application that the connection as a whole is ready to use.

A connection establishment tree may be degenerate, and only have a single leaf node, such as a connection attempt to an IP address over a single interface with a single protocol.

```
1 [192.0.2.1:80, Wi-Fi, TCP]
```

A parent node may also only have one child (or leaf) node, such as a when a hostname resolves to only a single IP address.

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
```

4.1. Branch Types

There are three types of branching from a parent node into one or more child nodes. Any parent node of the tree MUST only use one type of branching.

4.1.1. Derived Endpoints

If a connection originally targets a single endpoint, there may be multiple endpoints of different types that can be derived from the original. The connection library should order the derived endpoints according to application preference and expected performance.

DNS hostname-to-address resolution is the most common method of endpoint derivation. When trying to connect to a hostname endpoint on a traditional IP network, the implementation SHOULD send DNS queries for both A (IPv4) and AAAA (IPv6) records if both are supported on the local link. The algorithm for ordering and racing these addresses SHOULD follow the recommendations in Happy Eyeballs [I-D.ietf-v6ops-rfc6555bis].

```
1 [www.example.com:80, Wi-Fi, TCP]
  1.1 [2001:DB8::1.80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, Wi-Fi, TCP]
  1.3 [2001:DB8::2.80, Wi-Fi, TCP]
  1.4 [2001:DB8::3.80, Wi-Fi, TCP]
```

DNS-Based Service Discovery can also provide an endpoint derivation step. When trying to connect to a named service, the client may discover one or more hostname and port pairs on the local network using multicast DNS. These hostnames should each be treated as a branch which can be attempted independently from other hostnames. Each of these hostnames may also resolve to one or more addresses, thus creating multiple layers of branching.

```
1 [term-printer._ipp._tcp.meeting.ietf.org, Wi-Fi, TCP]
  1.1 [term-printer.meeting.ietf.org:631, Wi-Fi, TCP]
    1.1.1 [31.133.160.18.631, Wi-Fi, TCP]
```

4.1.2. Alternate Paths

If a client has multiple network interfaces available to it, such as mobile client with both Wi-Fi and Cellular connectivity, it can attempt a connection over either interface. This represents a branch point in the connection establishment. Like with derived endpoints,

the interfaces should be ranked based on preference, system policy, and performance. Attempts should be started on one interface, and then on other interfaces successively after delays based on expected round-trip-time or other available metrics.

```
1 [192.0.2.1:80, Any, TCP]
  1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [192.0.2.1:80, LTE, TCP]
```

This same approach applies to any situation in which the client is aware of multiple links or views of the network. Multiple Paths, each with a coherent set of addresses, routes, DNS server, and more, may share a single interface. A path may also represent a virtual interface service such as a Virtual Private Network (VPN).

The list of available paths should be constrained by any requirements or prohibitions the application sets, as well as system policy.

4.1.3. Protocol Options

Differences in possible protocol compositions and options can also provide a branching point in connection establishment. This allows clients to be resilient to situations in which a certain protocol is not functioning on a server or network.

This approach is commonly used for connections with optional proxy server configurations. A single connection may be allowed to use an HTTP-based proxy, a SOCKS-based proxy, or connect directly. These options should be ranked and attempted in succession.

```
1 [www.example.com:80, Any, HTTP/TCP]
  1.1 [192.0.2.8:80, Any, HTTP/HTTP Proxy/TCP]
  1.2 [192.0.2.7:10234, Any, HTTP/SOCKS/TCP]
  1.3 [www.example.com:80, Any, HTTP/TCP]
    1.3.1 [192.0.2.1:80, Any, HTTP/TCP]
```

This approach also allows a client to attempt different sets of application and transport protocols that may provide preferable characteristics when available. For example, the protocol options could involve QUIC [I-D.ietf-quic-transport] over UDP on one branch, and HTTP/2 [RFC7540] over TLS over TCP on the other:

```
1 [www.example.com:443, Any, Any HTTP]
  1.1 [www.example.com:443, Any, QUIC/UDP]
    1.1.1 [192.0.2.1:443, Any, QUIC/UDP]
  1.2 [www.example.com:443, Any, HTTP2/TLS/TCP]
    1.2.1 [192.0.2.1:443, Any, HTTP2/TLS/TCP]
```

Another example is racing SCTP with TCP:

```
1 [www.example.com:80, Any, Any Stream]
  1.1 [www.example.com:80, Any, SCTP]
        1.1.1 [192.0.2.1:80, Any, SCTP]
  1.2 [www.example.com:80, Any, TCP]
    1.2.1 [192.0.2.1:80, Any, TCP]
```

Implementations that support racing protocols and protocol options SHOULD maintain a history of which protocols and protocol options successfully established, on a per-network basis. This information can influence future racing decisions to prioritize or prune branches.

4.2. Branching Order-of-Operations

Branch types must occur in a specific order relative to one another to avoid creating leaf nodes with invalid or incompatible settings. In the example above, it would be invalid to branch for derived endpoints (the DNS results for www.example.com) before branching between interface paths, since usable DNS results on one network may not necessarily be the same as DNS results on another network due to local network entities, supported address families, or enterprise network configurations. Implementations must be careful to branch in an order that results in usable leaf nodes whenever there are multiple branch types that could be used from a single node.

The order of operations for branching, where lower numbers are acted upon first, SHOULD be:

1. Alternate Paths

2. Protocol Options

3. Derived Endpoints

Branching between paths is the first in the list because results across multiple interfaces are likely not related to one another: endpoint resolution may return different results, especially when using locally resolved host and service names, and which protocols are supported and preferred may differ across interfaces. Thus, if multiple paths are attempted, the overall connection can be seen as a race between the available paths or interfaces.

Protocol options are checked next in order. Whether or not a set of protocol, or protocol-specific options, can successfully connect is generally not dependent on which specific IP address is used. Furthermore, the protocol stacks being attempted may influence or

altogether change the endpoints being used. Adding a proxy to a connection's branch will change the endpoint to the proxy's IP address or hostname. Choosing an alternate protocol may also modify the ports that should be selected.

Branching for derived endpoints is the final step, and may have multiple layers of derivation or resolution, such as DNS service resolution and DNS hostname resolution.

5. Connection Establishment Dynamics

The primary goal of the connection establishment process is to successfully negotiate a protocol stack to an endpoint over an interface--to connect a single leaf node of the tree--with as little delay and as few unnecessary connections attempts as possible. Optimizing these two factors improves the user experience, while minimizing network load.

This section covers the dynamic aspect of connection establishment. While the tree described above is a useful conceptual and architectural model, an implementation does not know what the full tree may become up front, nor will many of the possible branches be used in the common case.

5.1. Building the Tree

The tree of options is built dynamically, out from the original trunk node. Any time that a connection attempt may be made directly to an endpoint without further derivation, and without needing to try alternate paths or protocol options that have not yet been covered by previous branches, the implementation SHOULD treat this as a leaf node and connect directly. Any time that an implementation chooses to branch between multiple options, it SHOULD determine a preferred order between the child nodes based on system policy, expected or historical performance, and application preference.

When multiple paths are available, and permitted by the system's policy, the implementation SHOULD branch between the various paths. The list SHOULD be sorted based on the system policies and routes (which often determine a "default" interface), preferences expressed by the application, and expected performance based on measured or advertised properties of each path.

When multiple protocol options are allowed by an application, and the system and implementation identify valid sets of protocols and protocol options, the implementation SHOULD branch between these sets. This list SHOULD be sorted based on application preference and

expected performance, generally measured in terms of latency and bandwidth.

An implementation will only branch to derive endpoints when necessary. This step involves the most external information, as endpoint derivation is often a process that requires fetching information from the network. Before branching, an implementation must first generate the list of derived endpoints. Once this list is sufficiently populated to continue, the implementation SHOULD sort the list based on preference and expected performance. When these derived endpoints are IP addresses, implementations SHOULD use the algorithm in [RFC6724] to sort the addresses. In cases where additional information can become available after the initial tree has been constructed, the implementation SHOULD update the tree to reflect new information and orderings if none of the leaf nodes are fully established.

5.2. Racing Methods

There are three different approaches to racing the attempts for different nodes of the connection establishment tree:

1. Immediate

2. Delayed

3. Failover

Each approach is appropriate in different use-cases and branch types. However, to avoid consuming unnecessary network resources, implementations SHOULD NOT use immediate racing as a default approach.

The timing algorithms for racing SHOULD remain independent across branches of the tree. Any timers or racing logic is isolated to a given parent node, and is not ordered precisely with regards to other children of other nodes.

5.2.1. Delayed Racing

Delayed racing can be used whenever a single node of the tree has multiple child nodes. Based on the order determined when building the tree, the first child node will be initiated immediately, followed by the next child node after some delay. Once that second child node is initiated, the third child node (if present) will begin after another delay, and so on until all child nodes have been initiated, or one of the child nodes successfully completes its negotiation.

Delayed racing attempts occur in parallel. Implementations SHOULD NOT terminate an earlier child connection attempt upon starting a secondary child.

The delay between starting child nodes SHOULD be based on the properties of the previously started child node. For example, if the first child represents an IP address with a known route, and the second child represents another IP address, the delay between starting the first and second IP addresses can be based on the expected retransmission cadence for the first child's connection (derived from historical round-trip-time). Alternatively, if the first child represents a branch on a Wi-Fi interface, and the second child represents a branch on an LTE interface, the delay should be based on the expected time in which the branch for the first interface would be able to establish a connection, based on link quality and historical round-trip-time.

Any delay SHOULD have a defined minimum and maximum value based on the branch type. Generally, branches between paths and protocols should have longer delays than branches between derived endpoints. The maximum delay should be considered with regards to how long a user is expected to wait for the connection to complete.

If a child node fails to connect before the delay timer has fired for the next child, the next child SHOULD be started immediately.

5.2.2. Failover

If an implementation or application has a strong preference for one branch over another, the branching node may choose to wait until one child has failed before starting the next. Failure of a leaf node is determined by its protocol negotiation failing or timing out; failure of a parent branching node is determined by all of its children failing.

An example in which failover is recommended is a race between a protocol stack that uses a proxy and a protocol stack that bypasses the proxy. Failover is useful in case the proxy is down or misconfigured, but any more aggressive type of racing may end up unnecessarily avoiding a proxy that was preferred by policy.

5.3. Completing Establishment

The process of connection establishment completes when one leaf node of the tree has completed negotiation with the remote endpoint successfully, or else all nodes of the tree have failed to connect. The first leaf node to complete its connection is then used by the application to send and receive data.

It is useful to process success and failure throughout the tree by child nodes reporting to their parent nodes (towards the trunk of the tree). For example, in the following case, if 1.1.1 fails to connect, it reports the failure to 1.1. Since 1.1 has no other child nodes, it also has failed and reports that failure to 1. Because 1.2 has not yet failed, 1 is not considered to have failed. Since 1.2 has not yet started, it is started and the process continues. Similarly, if 1.1.1 successfully connects, then it marks 1.1 as connected, which propagates to the trunk node 1. At this point, the connection as a whole is considered to be successfully connected and ready to process application data

```
1 [www.example.com:80, Any, TCP]
  1.1 [www.example.com:80, Wi-Fi, TCP]
    1.1.1 [192.0.2.1:80, Wi-Fi, TCP]
  1.2 [www.example.com:80, LTE, TCP]
    ...
```

If a leaf node has successfully completed its connection, all other attempts SHOULD be made ineligible for use by the application for the original request. New connection attempts that involve transmitting data on the network SHOULD NOT be started after another leaf node has completed successfully, as the connection as a whole has been established. An implementation MAY choose to let certain handshakes and negotiations complete in order to gather metrics to influence future connections. Similarly, an implementation MAY choose to hold onto fully established leaf nodes that were not the first to establish for use in future connections, but this approach is not recommended since those attempts were slower to connect and may exhibit less desirable properties.

5.3.1. Determining Successful Establishment

Implementations may select the criteria by which a leaf node is considered to be successfully connected differently on a per-protocol basis. If the only protocol being used is a transport protocol with a clear handshake, like TCP, then the obvious choice is to declare that node "connected" when the last packet of the three-way handshake has been received. If the only protocol being used is an "unconnected" protocol, like UDP, the implementation may consider the node fully "connected" the moment it determines a route is present, before sending any packets on the network.

For protocol stacks with multiple handshakes, the decision becomes more nuanced. If the protocol stack involves both TLS and TCP, an implementation MAY determine that a leaf node is connected after the TCP handshake is complete, or it MAY wait for the TLS handshake to complete as well. The benefit of declaring completion when the TCP

handshake finishes, and thus stopping the race for other branches of the tree, is that there will be less burden on the network from other connection attempts. On the other hand, by waiting until the TLS handshake is complete, an implementation avoids the scenario in which a TCP handshake completes quickly, but TLS negotiation is either very slow or fails altogether in particular network conditions or to a particular endpoint.

6. API Considerations

In general, the internal states and nodes of racing connection establishment do not need to be exposed to applications. Instead, this process SHOULD be treated as an abstraction of a single, aggregate connection establishment behind an API. This places some requirements on the API, including:

- The API must allow the application to specify an un-resolved endpoint as the remote side of the connection, such as a URI or hostname + port. The application also should be able to provide constraints on path selection and protocol features.

- Any read or write operations cannot take effect until one leaf node has been chosen as the connected node. The API needs to either expose asynchronous reads and writes, or else prohibit reads and writes until the connection is established.

- The action of starting or initiating the connection may involve many network-bound operations, so this operation SHOULD be asynchronous.

- Properties of the connection, such as the remote and local addresses, the interface used, and the protocols used, may not be queryable until the connection is established.

6.1. Handling 0-RTT Data

Several protocols allow sending higher-level protocol or application data within the first packet of their protocol establishment, such as TCP Fast Open [RFC7413] and TLS 1.3 [I-D.ietf-tls-tls13]. This approach is referred to as sending Zero-RTT (0-RTT) data. This is a desirable property, but poses challenges to an implementation that uses racing during connection establishment.

If the application has 0-RTT data to send in any protocol handshakes, it needs to provide this data before the handshakes have begun. When racing, this means that the data SHOULD be provided before the process of connection establishment has begun. If the API allows the application to send 0-RTT data, it MUST provide an interface that

identifies this data as idempotent data. In general, 0-RTT data may be replayed (for example, if a TCP SYN contains data, and the SYN is retransmitted, the data will be retransmitted as well), but racing means that different leaf nodes have the opportunity to send the same data independently. If data is truly idempotent, this should be permissible.

Once the application has provided its 0-RTT data, an implementation SHOULD keep a copy of this data and provide it to each new leaf node that is started and for which a 0-RTT protocol is being used.

It is also possible that protocol stacks within a particular leaf node use 0-RTT handshakes without any idempotent application data. For example, TCP Fast Open could use a Client Hello from a TLS as its 0-RTT data, shortening the cumulative handshake time.

0-RTT handshakes often rely on previous state, such as TCP Fast Open cookies, previously established TLS tickets, or out-of-band distributed pre-shared keys (PSKs). Implementations should be aware of security concerns around using these tokens across multiple addresses or paths when racing. In the case of TLS, any given ticket or PSK SHOULD only be used on one leaf node. If implementations have multiple tickets available from a previous connection, each leaf node attempt MUST use a different ticket. In effect, each leaf node will send the same early application data, yet encoded (encrypted) differently on the wire.

7. Security Considerations

See Section 6.1 for security considerations around racing with 0-RTT data.

An attacker that knows a particular device is racing several options during connection establishment may be able to block packets for the first connection attempt, thus inducing the device to fall back to a secondary attempt. This is a problem if the secondary attempts have worse security properties that enable further attacks. Implementations should ensure that all options have equivalent security properties to avoid incentivizing attacks.

Since results from the network can determine how a connection attempt tree is built, such as when DNS returns a list of resolved endpoints, it is possible for the network to cause an implementation to consume significant on-device resources. Implementations SHOULD limit the maximum amount of state allowed for any given node, including the number of child nodes, especially when the state is based on results from the network.

8. IANA Considerations

This document has no request to IANA.

9. Acknowledgments

Thanks to Josh Graessley and Stuart Cheshire for their help in the design of the original implementation of Happy Eyeballs for Apple that began this work.

10. Informative References

[I-D.ietf-quic-transport]
Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", draft-ietf-quic-transport-07 (work in progress), October 2017.

[I-D.ietf-tls-tls13]
Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", draft-ietf-tls-tls13-21 (work in progress), July 2017.

[I-D.ietf-v6ops-rfc6555bis]
Schinazi, D. and T. Pauly, "Happy Eyeballs Version 2: Better Connectivity Using Concurrency", draft-ietf-v6ops-rfc6555bis-06 (work in progress), October 2017.

[RFC6724] Thaler, D., Ed., Draves, R., Matsumoto, A., and T. Chown, "Default Address Selection for Internet Protocol Version 6 (IPv6)", RFC 6724, DOI 10.17487/RFC6724, September 2012, <https://www.rfc-editor.org/info/rfc6724>.

[RFC7413] Cheng, Y., Chu, J., Radhakrishnan, S., and A. Jain, "TCP Fast Open", RFC 7413, DOI 10.17487/RFC7413, December 2014, <https://www.rfc-editor.org/info/rfc7413>.

[RFC7540] Belshe, M., Peon, R., and M. Thomson, Ed., "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, DOI 10.17487/RFC7540, May 2015, <https://www.rfc-editor.org/info/rfc7540>.

[RFC7556] Anipko, D., Ed., "Multiple Provisioning Domain Architecture", RFC 7556, DOI 10.17487/RFC7556, June 2015, <https://www.rfc-editor.org/info/rfc7556>.

Author's Address

Tommy Pauly
Apple Inc.
1 Infinite Loop
Cupertino, California 95014
United States of America

Email: tpauly@apple.com