Guidelines for Racing During Connection Establishment
—or—
What Are We Connecting to, Anyway?

draft-pauly-taps-guidelines-01

Tommy Pauly
TAPS

IETF 100, November 2017, Singapore

Draft Updates

Now focused solely on the use of Racing during Connection Establishment

Generalizes Happy Eyeballs (RFC6555 and draft-ietf-v6ops-rfc6555bis) beyond addresses

Attempt at making the racing algorithm more normative and deterministic via SHOULDs

Connection Tree

Draft recommends viewing connection establishment racing as a tree.

Defines notation for summarizing connection attempts as [Endpoint, Path, Protocol]:

```
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]
```

Branching Types

Types of racing are strictly ordered:

1. **Path** racing (e.g. Wi-Fi, then LTE)
2. **Protocol Stack** racing (e.g. QUIC then HTTP/2)
3. **Derived Endpoint** racing (e.g. IPv6 then IPv4)

Other orderings are liable to cause errors:

- Resolved endpoints and supported protocols may be specific to a network path
- Endpoint attributes (ports, proxies) are specific to a protocol stack

Racing Methods

There are three approaches to racing:

1. **Simultaneous** (don't do this one!)
2. **Timer Delayed** (based on historical data)
3. **Failover** (used when there is a strong policy preference)

Determining Establishment

Once one attempt has successfully established, other attempts are not used (may be cancelled)

Establishment can have several interpretations:

- Transport handshake complete (TCP)
- Security handshake complete (TLS)
- Useful application data received (HTTP Response)

Draft Commonalities

draft-pauly-taps-guidelines & draft-grinnemo-taps-he

Both describe *Happy Eyeballs* algorithm for protocol stacks (SCTP v TCP, QUIC v HTTP/2, Proxy v Direct)

Both recommend historical databases of which protocols work on different paths to order options

Both determine attempts as a combination of:

- Application preferences via API
- System policy
- Historical data

Draft Differences

draft-pauly-taps-guidelines & draft-grinnemo-taps-he

Should the system try protocols the app didn't explicitly ask for?

- ✓ More likely to try non-"default" protocol
- ✖ No reason to believe the server will support the protocol, incurring many failures

Should the racing be flat or less structured than a tree?

- ✓ More combinations allowed, and more orderings of attempts
- ✖ Easier to try "invalid" connection attempts that will fail or connect to the wrong peer

What Are We Connecting to, Anyway?

Connection options **must** be determined by the application or the network, **not** the library

- Happy Eyeballs employs racing between addresses received via DNS
- TLS can indicate QUIC & HTTP/2 support via ALPN
- Application knows expected server configuration (ports, protocols, options) beforehand

What Are We Connecting to, Anyway?

Vague application preferences:

Low latency

Allow unreliable or out-of-order

High priority

Cheapest interface

Strict application preferences:

Prohibit cellular interfaces

Server supports both SCTP and TCP

What Are We Connecting to, Anyway?

Vague application preferences:

- *Low latency*
- *Allow unreliable or out-of-order*
- *High priority*
- *Cheapest interface*

Implementation can interpret

Strict application preferences:

- *Prohibit cellular interfaces*
- *Server supports both SCTP and TCP*

Implementation must follow

One of the main points of TAPS is to let an implementation of the transport library be flexible and not ossify on just one protocol stack

But what has caused ossification so far?

Avoiding Ossification

Possible Culprit 1: Applications using code only compatible with one protocol

- True for IPv6 transitions, where applications only handle AF_INET
- Is this true for transport protocols?
    - Often for socket options
    - …but isn't SOCK_STREAM "generic"?

Avoiding Ossification

Possible Reason 2: Libraries inferring acceptable protocols based on “generic” options

- The generic stream socket API, SOCK_STREAM, has come to mean “use TCP”, tying a mode of data transport to the protocol underneath
    - Would have been better to separate “streams” from TCP, so we could use streams over QUIC, etc
- A TAPS solution that always tries the same set of protocols for certain options will lead to the same ossification

Avoiding Ossification

If solutions deploy new protocols or new protocol options, they likely will rev their applications *and* servers at the same time

The TAPS API *should*:

- Allow applications to tune a “protocol options configuration” without changing any code that managers or uses the transports.
- Prune or sort options based on which protocols best meet application preferences.
- Race/Happy Eyeballs between options automatically.

The TAPS API *should not*:

- Attempt protocols not explicitly requested by the application or some other system authority that knows what protocols might work.
- Create new “mappings” that always mean the same thing: “streams” → TCP, or “Partial Reliability” → SCTP