PANAPI, a Path-Aware Networking API

Thorben Krüger

Networks and Distributed Systems Lab
Otto-von-Guericke-Universität
Magdeburg

March 23, 2022

Outline

Who we are

- Networks and Distributed Systems (NetSys) Lab at Otto von Guericke University (OVGU)
  `https://netsys.ovgu.de`
- Research focus on SCION Internet architecture
  `https://scion-architecture.net`
- My particular topic: Host-based traffic optimization under path-awareness

`thorben.krueger@ovgu.de`

About SCION

- Proposed new Internet architecture, designed to supersede BGP
- Provides practical path-awareness at the end host and multipath at the inter-domain level
- Already deployed in the wild
- Developed at ETH Zürich

https://scion-architecture.net

Path-Awareness at the End Hosts?

- General Insight: Adoption of new network technology and features tends to happen behind the scenes. Users and App developers shouldn't have to be directly involved
- New functionality must (first) be added at deeper levels of the network stack
- Network programming needs modern high-level abstractions
- The TAPS WG seems to hold similar views

PANAPI[1] - A Transport Services System (not only) for SCION

- Adopts existing sensible modern networking abstractions from IETF community
- Adds support for path-awareness (specifically SCION) to TAPS
- Scriptable back end to test out new networking features and adjust behavior
- Open source and funded via NGI Pointer[2]

[1] https://dl.acm.org/doi/pdf/10.1145/3472727.3472808
[2] https://www.ngi.eu/funded_solution/ngi-pointer-project-33/

PANAPI - Implementation

- Written in Go
- Backend scriptable in Lua
- Support for TCP/UDP/QUIC as well as SCION with (UDP/QUIC)
- Frontend API follows the TAPS architecture as much as possible

PANAPI - API Design Decisions

SCION Support

- Added "in the spirit of" the TAPS spec

No Event System

- Go comes with powerful native primitives for concurrency
- All calls are blocking, instead of "try-and-fail"
- (Blocking calls can be dispatched in asynchronous "goroutines" as required)
- Any errors are returned early

Strongly typed properties

- Property collections are struct types
- Properties are named struct fields
- Property values are pre-defined as constants
- (Catches typos at compile time, avoids errors at runtime)

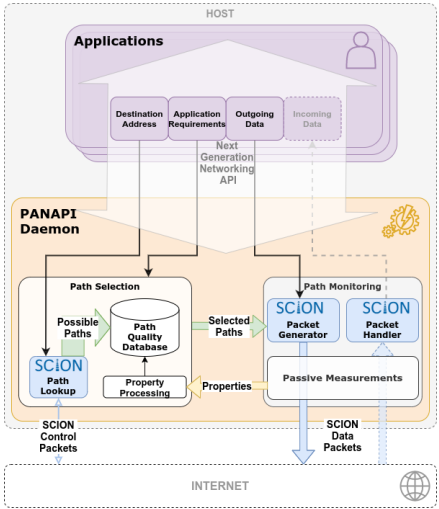
Does this sound reasonable? Feedback welcome!

PANAPI - Code Example

```
r := taps.RemoteEndpoint{
     Address: "19-ffaa:0:1303,203.0.113.42:1337",
     Protocols: []taps.Protocol{quic.Protocol()},
}

tp := taps.NewTransportProperties()
tp.Multipath = taps.ACTIVE

p := taps.NewPreconnection(r, tp, taps.NewSecurityParameters())
p.ConnectionProperties.MultipathPolicy = taps.AGGREGATE
p.ConnectionProperties.CapacityProfle = taps.CAPACITY_SEEKING

Connection, err := p.Initiate()
if err != nil { ... }

err = Connection.Send(Request)
if err != nil { ... }

Response, err := Connection.Receive()
if err != nil { ... }

Connection.Close()
```

PANAPI - Backend

- SCION-specific (currently)
- Collects and caches network performance information per Path
- Scriptable path quality estimation
- Scriptable path choice per Message/Packet
- Scriptable active latency measurements per Path (optional)



Applying TAPS concepts to SCION

Many Generic Connection Properties from TAPS can directly be applied to SCION in a pretty straight-forward manner

`connCapacityProfile`

- Consult cached performance information to compare path options
- *Translating the cached information to path decisions respecting the different profiles is part of my research*

`multipathPolicy`

- Duplicate/Distribute Messages/Packets according to policy

`isolateSession`

- Select paths without consulting cached metrics

Some mechanisms that are used in the SCION world do not appear to have a TAPS equivalent yet:

`activeProbe`

- Perform active (latency) measurements on alternative Paths to a destination

Feedback and Opinions Welcome!

- Does PANAPI honor the Spirit of TAPS?
- Any general pitfalls to avoid?
- Any SCION-specific issues you perceive?
- General remarks?

(I also have some feedback of my own for TAPS on the next slides)

Feedback: Property Types

In a statically typed language, the various possible property types are a challenge to implement properly

3.1. Provide Common APIs for Common Features

Any functionality that is common across multiple transport protocols SHOULD be made accessible through a unified set of calls using the Transport Services API. As a baseline, any Transport Services API SHOULD allow access to the minimal set of features offered by transport protocols [RFC8923].

An application can specify constraints and preferences for the protocols, features, and network interfaces it will use via Properties. Properties are used by an application to declare its preferences for how the transport service should operate at each stage in the lifetime of a connection. Transport Properties are subdivided into Selection Properties, which specify which paths and protocol stacks can be used and are preferred by the application; Connection Properties, which inform decisions made during connection establishment and fine-tune the established connection; and Message Properties, set on individual Messages.¶

Section 8.1 provides a list of Connection Properties, while Selection Properties are listed in the subsections below. Many properties are only considered during establishment, and can not be changed after a Connection is established; however, they can still be queried. The return type of a queried Selection Property is Boolean, where `true` means that the Selection Property has been applied and `false` means that the Selection Property has not been applied. Note that `true` does not mean that a request has been honored. For example, if `Congestion control` was requested with preference level `Prefer`, but congestion control could not be supported, querying the `congestionControl` property yields the value `false`. If the preference level `Avoid` was used for `Congestion control`, and, as requested, the Connection is not congestion controlled, querying the `congestionControl` property also yields the value `false`.¶

Suggestion: Why not distinguish

- "Preferences", i.e., selection properties that can (still) be modified (e.g., "Prefer", "Prohibit")
- "Properties" that actually reflect the settled result of transport selection process (e.g., "true", "false")

Nit pick: Event System

- Language in the drafts is biased in favor of event-system based implementations
- Validity of other approaches is not obvious or only mentioned in passing

> ## 2.1. Event-Driven API
>
> Originally, the Socket API presented a blocking interface for establishing connections and transferring data. However, most modern applications interact with the network asynchronously. Emulation of an asynchronous interface using the Socket API generally uses a try-and-fail model. If the application wants to read, but data has not yet been received from the peer, the call to read will fail. The application then waits and can try again later. In contrast to the Socket API, all interaction using the Transport Services API is expected to be asynchronous and use an event-driven model (see Section 4.1.6). For example, an application first issues a call to receive new data from the connection. When delivered data becomes

Suggestion

- Add a clarifying introductory sentence and perhaps some minor rewording

Conclusion

- TAPS has already been incredibly useful for us
- We hope to continue to provide valuable feedback