An Abstract Application Layer Interface to Transport Services
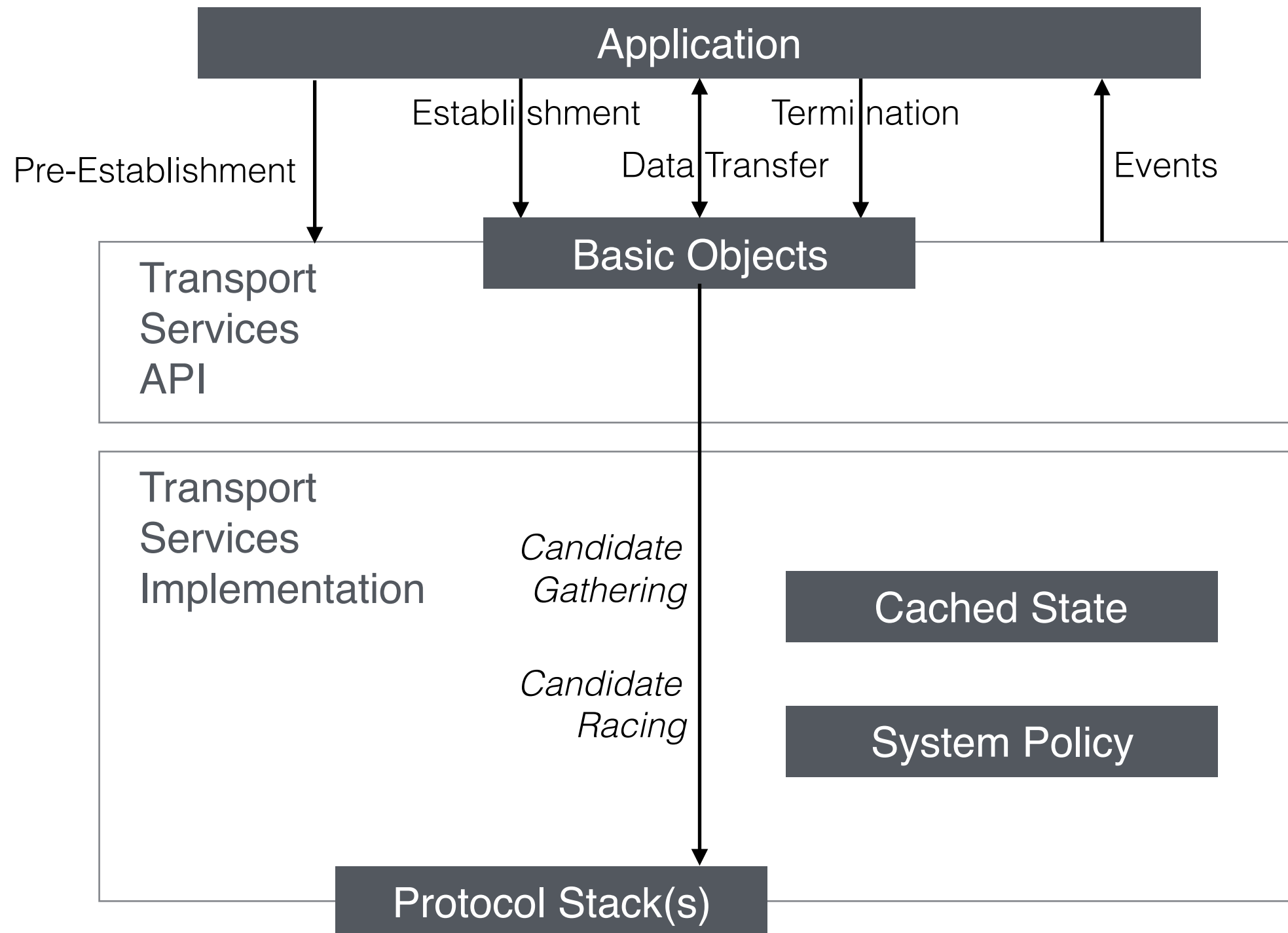# draft-trammell-taps-interface-00

Brian Trammell

TAPS — IETF 101 — London — 21 March 2018

Architecture Diagram

Interface Design Principles (§3)

We set out to define a **single interface** to a variety of transport protocols to be used in a variety of application design patterns, to enable applications written to a single API to make use of multiple transport protocols in terms of the features they provide, providing:

- explicit support for **security properties** as first-order transport features;

- **asynchronous** connection, transmission, and reception;

- support for **multistreaming and multipath** transport protocols; and

- **atomic transmission of data**, using application-assisted framing and deframing where necessary.

Interface Diagram
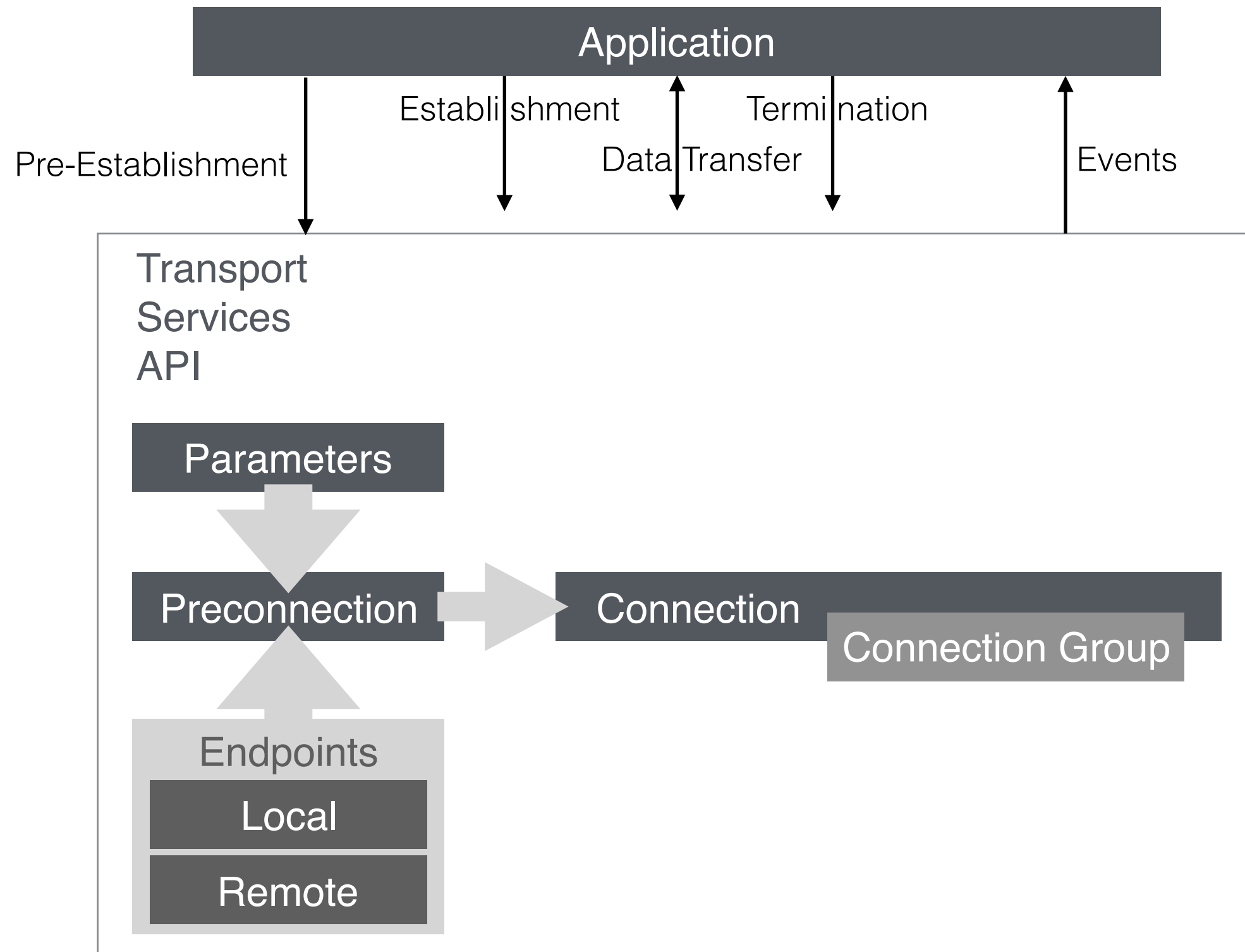
Endpoints (§5.1)

- Remote and local endpoints can be specified at a variety of resolutions (e.g. hostname / service name, address / port, interface).

- Resolution is under transport services control, not application control.

  - May depend on PvD / selected protocol stack.

  - Open issue: resolution can leak interest when DNS is not private.

Transport Parameters (§5.2): Protocol and Path Selection Properties

- Protocol and path selection properties used to select/eliminate candidates during connection establishment.

- Five levels of preference: require, prefer, ignore, avoid, prohibit.

- Properties (mostly) derived from minset:

- Reliable Data Transfer

- Preservation of Ordering

- Per-Message Reliability

- 0-RTT Session Establishment

- Multiplexing (multistreaming)

- RTX and ICMP notification

- Checksum coverage control

- Capacity profile

- (path-only) Interface Type

# Transport Parameters: Protocol Properties (§9.1)

- Generic protocol properties allow configuration and querying of protocol stacks in a transport-independent way:

- Relative Niceness within group

- Group TX scheduler

- Connection Abort timeout

- RTX notification threshold

- Minimum checksum coverage

- Maximum 0RTT message size

- Maximum non-fragmented message size

- Maximum non-partial message size on send

- Maximum non-partial message size on receive

- Specific protocol properties allow specific stacks to be configured in detail, should they be selected.

Transport Parameters: Security Parameters (§5.3)

- Generic security properties allow configuration and querying of security features in a protocol-independent way:

- Identity

- Private Key

- Groups

- Algorithms

- Ciphersuites

- Session Cache configuration

- Pre-shared keys

- Trust verification and identity challenge callbacks

Preconnection (§5)
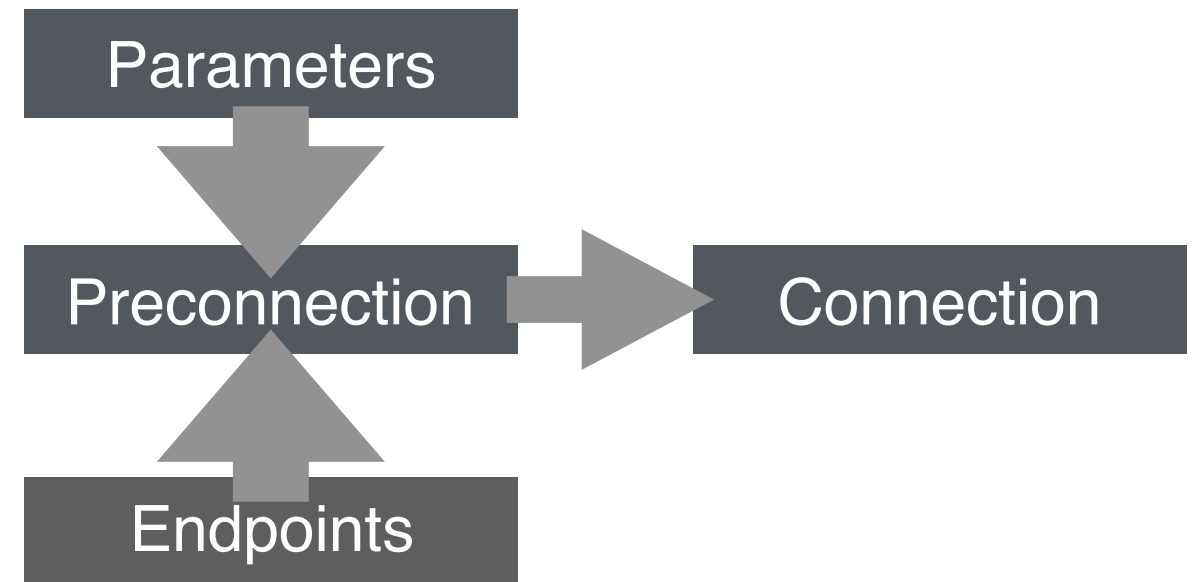
- A **preconnection** describes the state of a connection that might exist in the future, including parameters and endpoint specifiers.

- This design allows the system to prepare and cache information based on application requirements before establishment

- Preconnections can also be used to *group* connections before establishment.

- Implementations of the interface may provide convenience calls to connect via an implicit preconnection.

Establishing Connections (§6)

- Three ways to establish a connection:

  - Active (`Initiate()`): application notified that the connection is up by a `Ready<>` event.

  - Passive (`Listen()`): application notified of each incoming connection by a `ConnectionReceived<>` event.

  - Simultaneous/Peer (`Rendezvous()`): application notified connection is up by a `RendezvousDone<>` event

- Data can be sent on an initiating connection immediately.

  - Details of 0RTT still an open issue.

Connection Groups (§6.4)

Connection

Connection Group

- Connections can be *entangled* into groups

- All connections in a group share protocol properties and may share connection state.

- Connections in a connection group are implemented as streams in a multistreaming protocol when available.

- `Preconnection.Clone()` creates preconnections whose eventual connections will be entangled.

- `Connection.Clone()` creates a new connection entangled with an existing one.

  - New streams yield a `ConnectionReceived<>` event

Sending Data (§7)

- Data (as a Message) sent with `Connection.Send()`

    - Sender-side framing allows for arbitrarily-typed application objects to be converted to octet streams.

- Send parameters control per-Send behavior:

- Lifetime

- Niceness

- Ordered

- Idempotent

- Checksum Coverage

- Immediate Acknowledgment

- Instantaneous Capacity Profile

- Sending may yield `Sent<>` or `Expired<>` events

Receiving Data (§8)

- Application indicates readiness to receive via `Connection.Receive()`, message sent to application via supplied callback.

- Message contains an octet array, as well as transport metadata

- Messages are split from octet via application-provided receiver-side deframing when the transport doesn't provide its own framing

- Very large messages or lack of deframing may result in partial reception

Connection Termination (§10)

- `Connection.Close()`: orderly connection shutdown after pending send and receive, results in `Connection.Closed<>` event

  - Underlying stack closes after last Connection in a Group closes.

- `Connection.Abort()`: immediate connection shutdown, results in `Connection.Aborted<>` event

  - All Connections in a Group abort simultaneously.

Interface Diagram

Parameters
Require()   Prefer()   Ignore()   Avoid()   Prohibit()
Security parameters (Identity, PrivateKey, Algorithm, Group, Ciphersuite)

Preconnection

Clone()

Initiate() → Ready<>
Listen() → CReceived<>
Rendezvous() → RDone<>

Connection

Clone()   → Connection Group

Send()   → Sent<>, Expired<>
Receive() → Received<>

Close() → Closed<>
Abort() → Aborted<>

Endpoints

Local

Remote

Next steps

There are still some open issues:
github.com/taps-api/drafts/issues

Adopt?