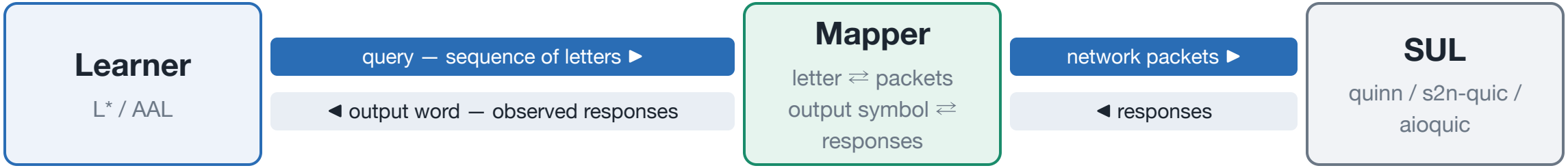
WIP on Building a Mapper for QUIC Protocol

Van Nam Pham, Olivier Levillain — Télécom SudParis

StANP Workshop, July 2026

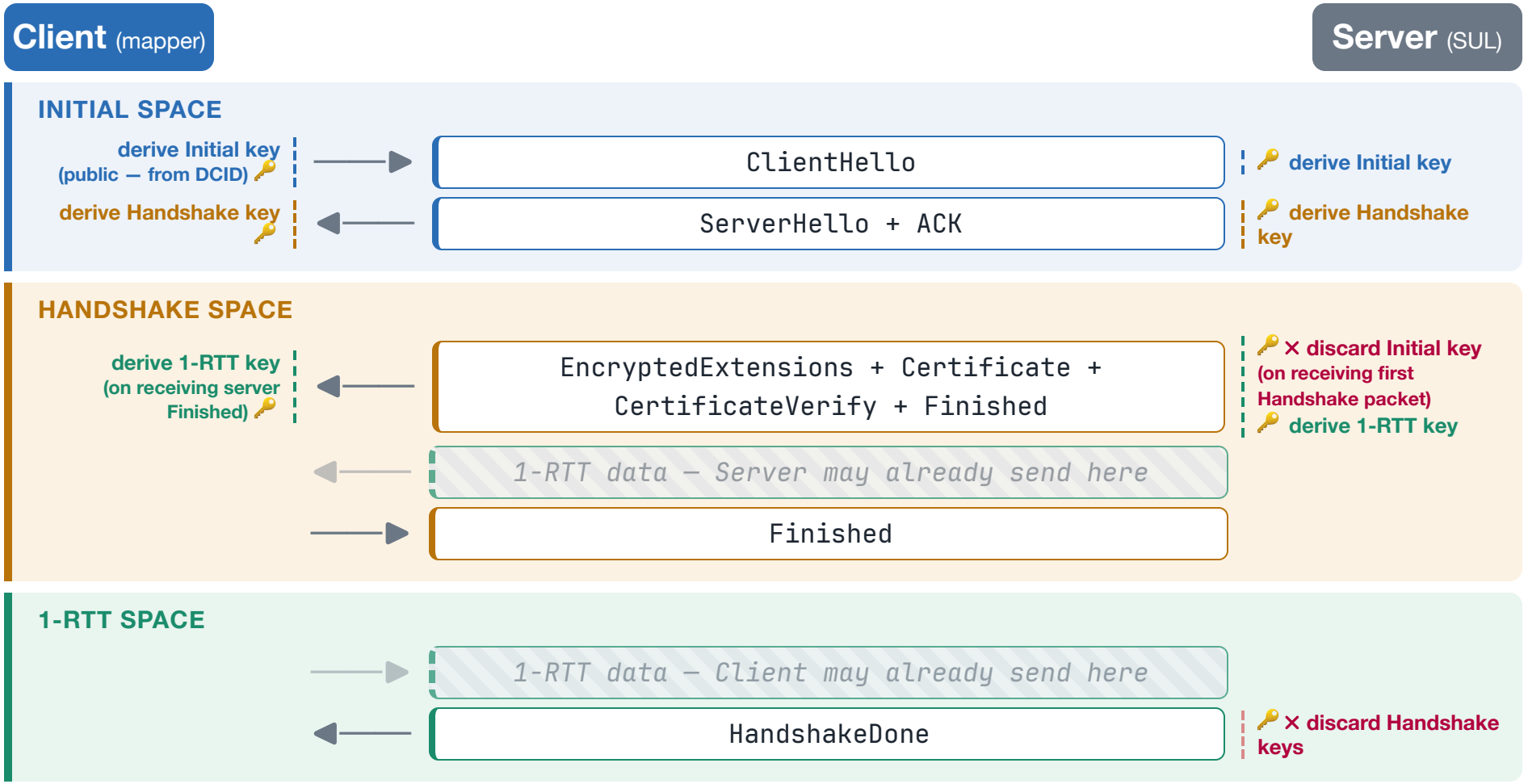
1. Context & Goal



- **Goal:** building a QUIC mapper to learn state machines of quinn (Rust) · s2n-quic (Rust) · aioquic (Python)
  - Compare behavioral differences
  - Uncover RFC non-compliance and logical bugs
- **Initial scope:** server side of the TLS 1.3 handshake over QUIC (mapper acts as the client)
- **Core challenge:** make QUIC deterministic for L* — same inputs ⇒ same outputs

2. A TLS 1.3 Handshake over QUIC

A **typical 1-RTT handshake** — spans 3 encryption spaces



- Each space uses a **different packet type** and has its own keys

3. Why QUIC ≠ TCP-based Protocols (SSH, TLS, …)

	TCP-based (SSH / TLS)	QUIC
Transport	TCP handles reliability → mapper handles only semantics of protocol messages	self-contained → mapper **also** runs the transport: packet numbers, retransmission, ACKing
Timeout Window	**longer, stable** timeout window	**PTO-driven & varying** (~25–50 ms): a few ms difference → changes the output
The end of a connection	TCP socket **EOF** = clear end signal	**no EOF** in UDP → mapper must define if a connection is ended

4. Handling Transport Reliability in the Mapper

- **Packet identity** — packet number (PN), only ever increases
  - a retransmit re-packs the old data under a new PN
  - sender retransmits if it gets no ACK within 1 PTO (*Probe Timeout*)
- **ACKing** — mapper ACKs the server's packet immediately (1-1) → less retransmit noise
  - implicit, *not yet* as a letter
- **As output** — per sent letter, track the server's ACK:
  - `ACKED` / `NOT_ACKED` → could the server decrypt it?
  - `ACK_INVALID` → server ACKed a PN never sent
- **Collection window** — ~1 PTO (30–40 ms) per letter *(longer = safer but slower)*

5. When Is a Connection "Over"? (No EOF)

- UDP has **no EOF** → socket never tells us if the peer is dead or alive
- QUIC's end signal is the `ConnectionClose` frame → on **sending or receiving** it, the server enters a **draining** state:

After `ConnectionClose`	Server reaction
within **3×PTO**	still draining → another `ConnectionClose` or silence
after **3×PTO**	`StatelessReset` / accepts a new connection

- **Now:** unlike TCP (stop after EOF), we keep sending after `ConnectionClose` — with a long timeout (~4× client PTO) to stay deterministic
- **Future:** *parameterize* the timeout (short / long) → learn behavior inside vs outside draining

6. Current Vocabulary — the Letters

- **Handshake:** `TLSClientHello`, `TLSClientFinished`
  - `ClientHello` variants: + RetryToken, + PSK (for 0-RTT resumption)
- **Control frames:** `Ping`, `PathChallenge`, `ConnectionClose`
- **Internal:** control the mapper only — e.g. `ResetCryptoState` (reset CRYPTO send offset)
- **Packet type is part of the letter** — mapper keeps lower keys → can send in a mismatched space:

e.g. packet type × letter	
`HandshakePacket(TLSClientFinished)`	correct
`OneRTTPacket(TLSClientFinished)`	wrong space

7. Parsing the Output

- **Frame-level semantics**: `HandshakeDone`, `PathResponse`, `NewToken`, `NewCID`, ...
- **Exception** — `Crypto` frames are reassembled into TLS handshake messages: `ServerHello`, `Certificate`, `Finished`, ...
- A `Crypto` frame at an already-seen offset = a duplicate / retransmit
  → `CryptoOverlapDuplicate` / `CryptoOverlapConflict`

Discussion & Open Questions

Behaviors we'd like to model next — and how to fit them into the mapper

8. Connection Identity & Migration

- Connection identity: **Connection ID pair (SCID / DCID)**
  → client may change `IP:port` and keep the connection: **connection migration**
- **RFC 9000 §9:** only allowed after the handshake is confirmed

Open question — what if a peer migrates *too early*?

- SCID/DCID (plaintext) + Initial keys are public (derived from DCID) → spoof from another `IP:port` and make the server respond (amplify / redirect)
- **Idea:** mapper holds multiple `IP:port`s, switches between them → does the server reply?

9. Fragmentation of TLS Handshake Messages

- A TLS message = 1-byte type + 3-byte length + body, sliced into `CRYPTO(offset, data)` frames → the receiver reassembles by offset
- **Idea:** fragment the messages, then **fuzz the reassembly**:
  - reorder the fragments
  - overlap — same offset, different data

A curious case (aioquic/quinn): RFC keeps CRYPTO offsets separate per space, yet

`InitialPacket(TLSClientHello + 0x14)` + `HandshakePacket(TLSClientFinished[1:])` → **accepted**

→ the server **merges the CRYPTO buffer across two different spaces** — worth exploring more

10. Extending the Letters — Handshake Authentication

- Model **client authentication** (mutual TLS): the server asks for a client cert
  - server → `CertificateRequest`
  - client → `Certificate` + `CertificateVerify`
- Possible letters for **invalid certs**:
  - empty, oversized
  - expired, untrusted, revoked
  - bad signature, bad transcript in `CertificateVerify`

11. Minor Findings — RFC Non-compliance (aioquic server)

Real deviations — still need more exploring to turn any into a vulnerability.

- **Retransmits 1-RTT data before `ClientFinished`**
  - has the 1-RTT *encrypt* key but not the *decrypt* key → can't read the client's ACK → data is un-ACK-able
  - RFC: no 1-RTT retransmit before the handshake completes
- **The client sends an Initial with an empty DCID, and the server accepts it** — RFC 9000 §7.2: the DCID *SHOULD* be ≥ 8 bytes
- **Migrates before the handshake completes** — client changes `IP:port` right after `ClientHello`, aioquic responds (RFC: only after handshake completed)
- **Replies to a reused PN** — a PN must never repeat → *SHOULD* be dropped

Thank you for listening!

Questions?

Appendix

Appendix — The Mapper's ACKing

- QUIC is reliable: a receiver
  - must **ACK** an *ack-eliciting* packet within `max_ack_delay` (default 25 ms)
  - if it doesn't → the sender's PTO (*Probe Timeout*) fires → it retransmits

Mapper ACK strategy	Effect on the output word
ACK late / not at all	server retransmits → duplicate responses → noise, non-determinism
ACK 1-1 *(our choice)*	every ack-eliciting packet ACKed immediately → reduce retransmit noise

> **ACK 1-1:** one ACK per received ack-eliciting packet

Appendix — ACKing the *Wrong* PN

- QUIC packet numbers only ever increase; an ACK tells the sender which PNs it has received
- RFC 9000 §13.2.1: a peer **MUST NOT** ACK a PN it never received — but the RFC doesn't say how a server reacts to one → implementation choice

quinn and aioquic servers react **differently** to such an invalid ACK:

Server	Reaction to an invalid ACK
quinn	closes — `CONNECTION_CLOSE(10)` PROTOCOL_VIOLATION
aioquic	accepts it — handshake continues

- **Our choice:** ACK exactly what was received, implicitly 1-1 — not yet added ACK as a letter

Appendix — ACK as Output Symbols

- Ack-eliciting frames **must** be ACKed → mapper checks: **did the server ACK what I sent?**

Output symbol	Meaning
`ACKED`	sent letter **was** ACKed
`NOT_ACKED`	sent letter was **not** ACKed
`ACK_INVALID`	server ACKed a PN **never sent**

- Send a letter at a **discarded** level (e.g. `HandshakePacket(Ping)` after the handshake): an `ACKED` means the server **still decrypted it** — `NOT_ACKED` means the key is gone