Formal Specification and Specification-based Testing of QUIC

Ken McMillan (kenmcm@cs.utexas.edu)

Lenore Zuck (zuck@uic.edu)

Specifications are the unicorns of the formal verification world.

We talk a lot about specifications, but few people have actually seen one.

We tend to assume specs will be provided to us by a “user” (another mythical creature).

Let’s make a foray into the mythic land of specifications. We’ll use QUIC as an example to look at some of the basic questions that arise in specifying complex systems.

Questions to ask about a specification

1 What is its function?

2 What is its form?

3 What is its content?

4 What is its process?

We have to answer 1 first.

Functions of a specification

Thinking tool

Contract between designers

Part of a formal proof

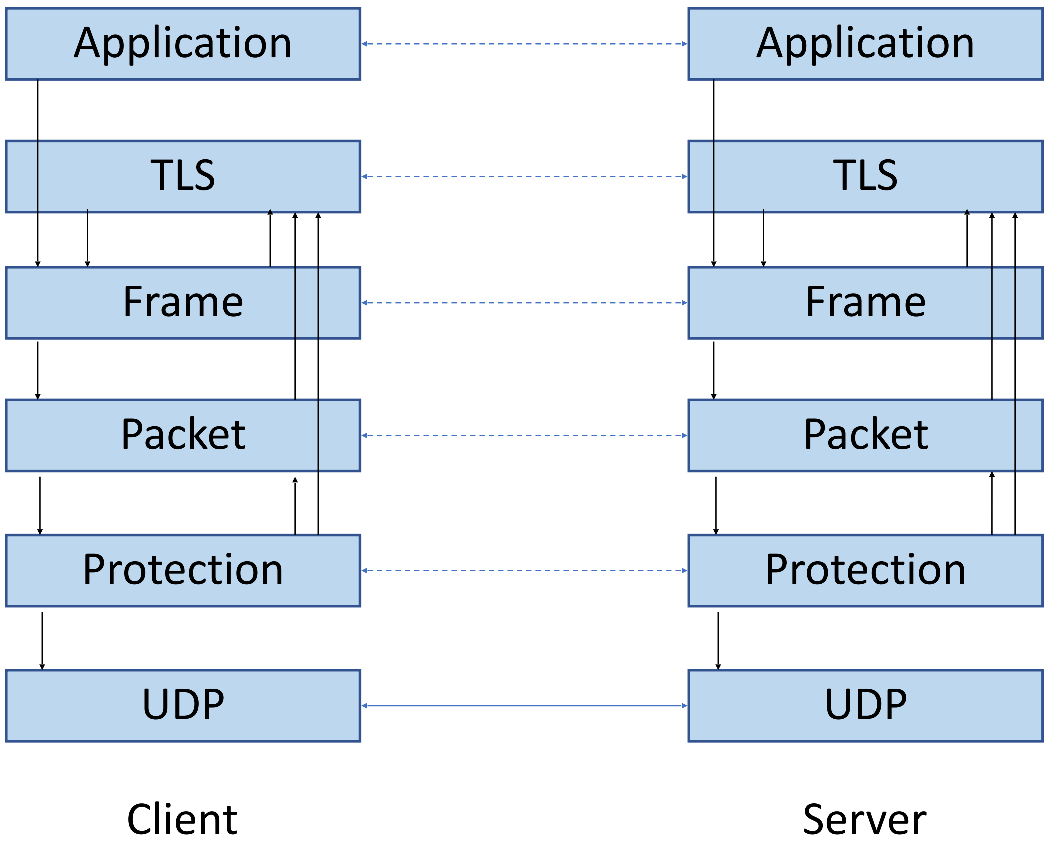
Test/simulation/evaluation artifact

What is QUIC?

- Replacement for TLS/TCP stack
  - Introduce by Google in 2013
  - Implemented in user space using UDP
- Goals
  - Reduce connection latency
  - Better congestion control
  - More responsive web applications
- Standardization process
  - IETF working group, current draft = 20
  - Transport for HTTP/3

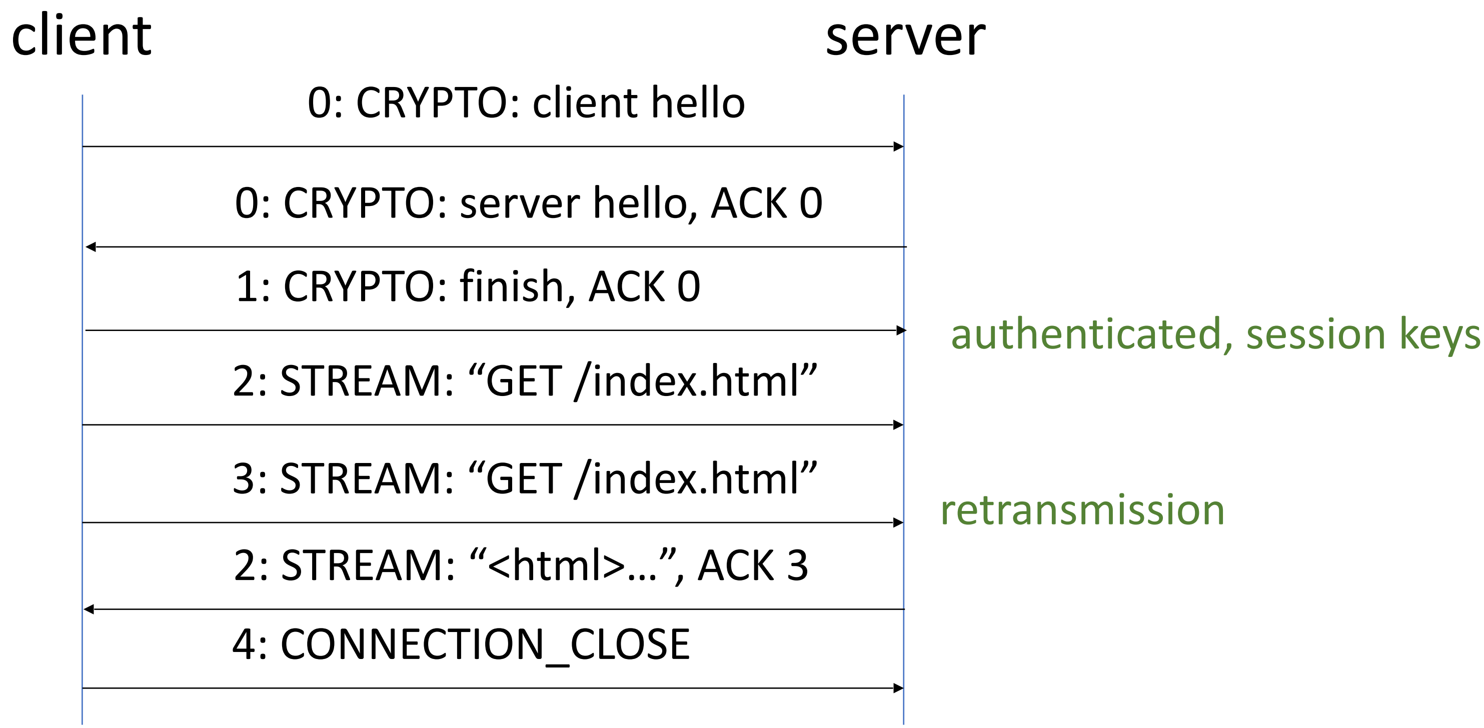
QUIC will likely carry a large fraction of traffic on the Internet. QUIC is *very complex*. We should be worrying about this.

How is it structured?



Protocol is not cleanly layered!

How does it work?



Functions: Multiple streams, retransmission, flow and congestion control, version negotiation, migration, path validation, connection ID management, pre-shared keys, etc. Describes in 224 pages of RFC.

The QUIC working group process

RFC: English-language standard document describing how to implement QUIC

Implementations: Each member implements and tests for performance, interoperability.

Why do we want a (formal) specification?

Test generality: Implementations testing each other are not sufficiently adversarial.

Compliance to a common standard: Assurance of compatibility with future implementations.

TLS story

Non-compliant implementations in the wild led to severe security issues.

Example: incorrect version negotiation led to ad-hoc downgrade strategies by browsers which led to downgrade attacks like POODLE.

This drives the specification function

Primary: test artifact

1) Generate adversarial tests
2) Check protocol compliance

Secondary: contract

1) Capture protocol knowledge implicit in implementations
2) Aid future implementers in compliance

Everything else follows function

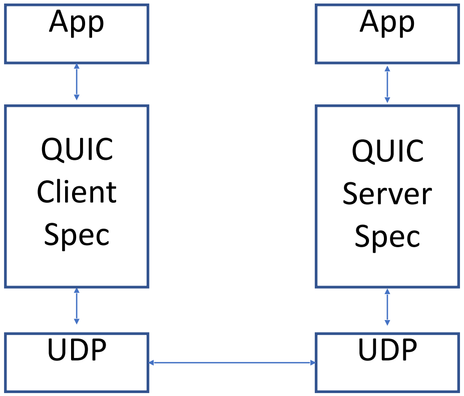
Form: Compositional, assume/guarantee, deterministic.

Content: *Safety* properties of events visible on the wire.

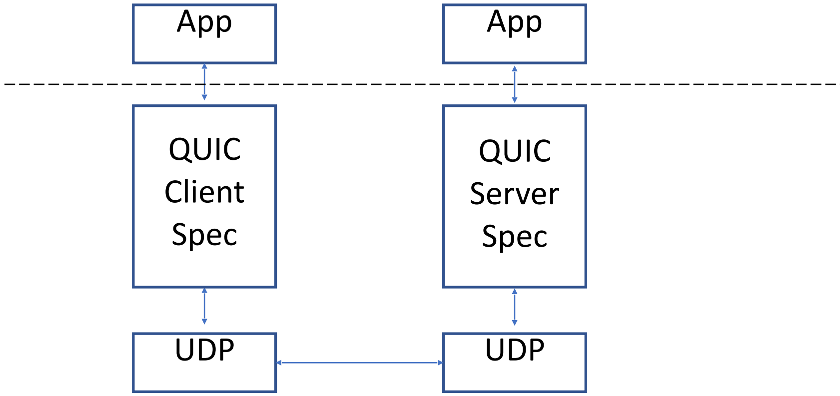
Process: Capture properties from testing actual implementation.

Formalize knowledge implicit in the implementations

Methodology (oversimplified)

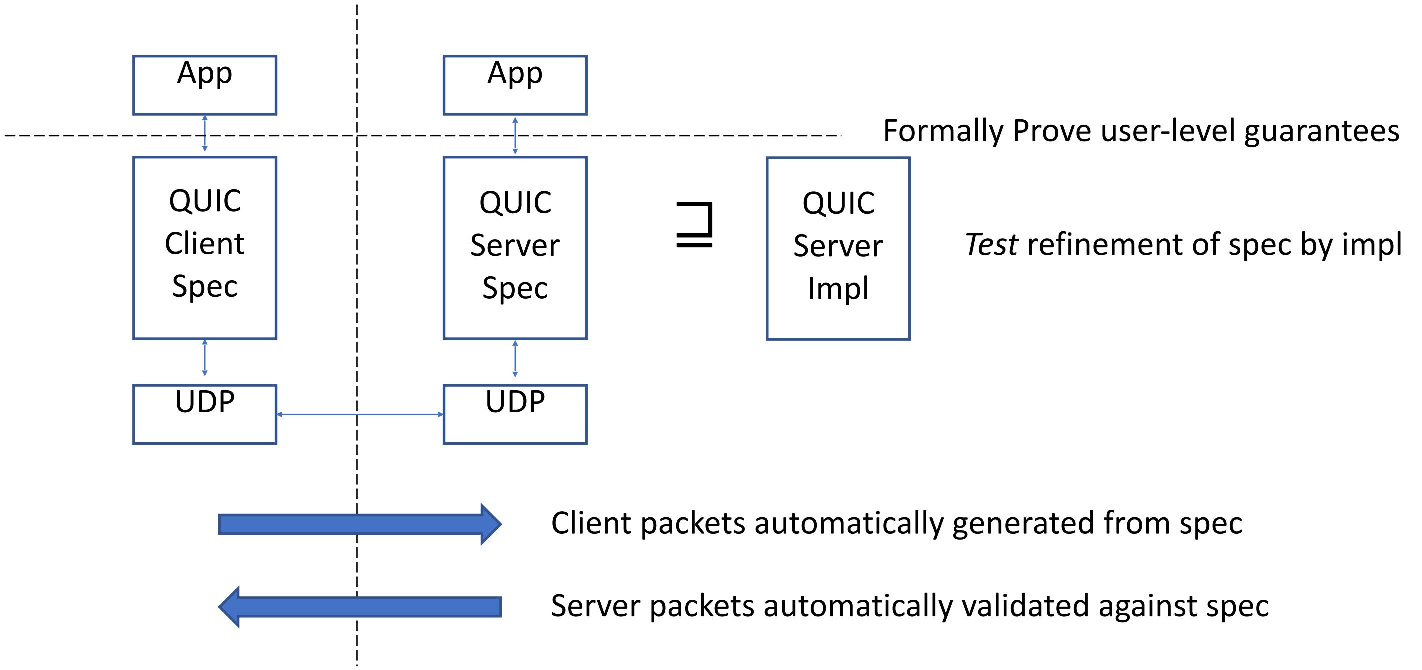


Methodology (oversimplified)



Formally Prove user-level guarantees

Methodology (oversimplified)



Capturing QUIC spec by testing

SIGCOMM 2019

Began specification work with draft RFC.

Specified only safety properties – no liveness or timing properties

Refined by testing four implementations of QUIC using specification-based testing infrastructure in Ivy.

Resulting specification in about 3KLOC of Ivy is highly incomplete. However, it can interact with real servers and clients to transfer web pages, without the servers and clients detecting protocol errors. This process revealed many errors in implementations and some problems in the draft standard.

Errors discovered

27 errors detected: 13 crashes, 12 compliance violations, 2 progress failures (no data transferred).

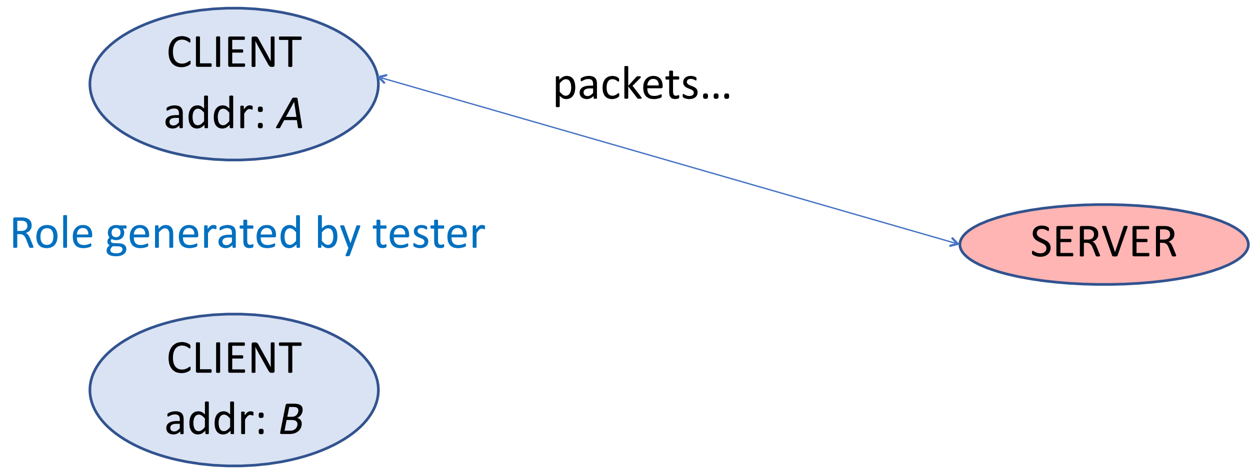
4 errors (apart from crashes) considered to be exploitable.

4 errors resulted from ambiguities in the standard.

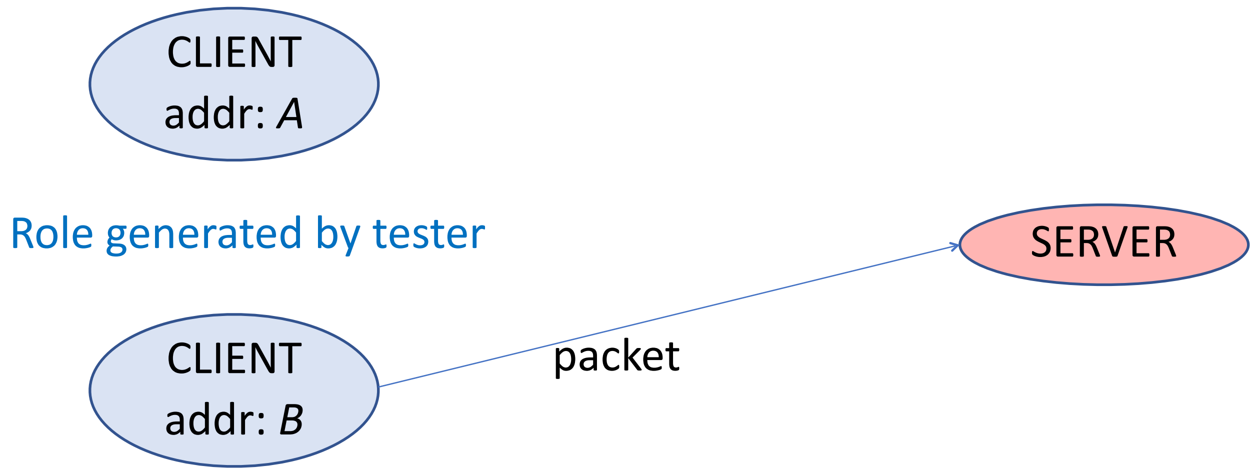
18-22 results from adversarial stimulus (e.g., unusual message order)

In every case where we could assign a root cause to the error, we found that detection was due to either adversarial stimulus or compliance checking. This validates our intuition that these elements were missing from prior testing efforts.

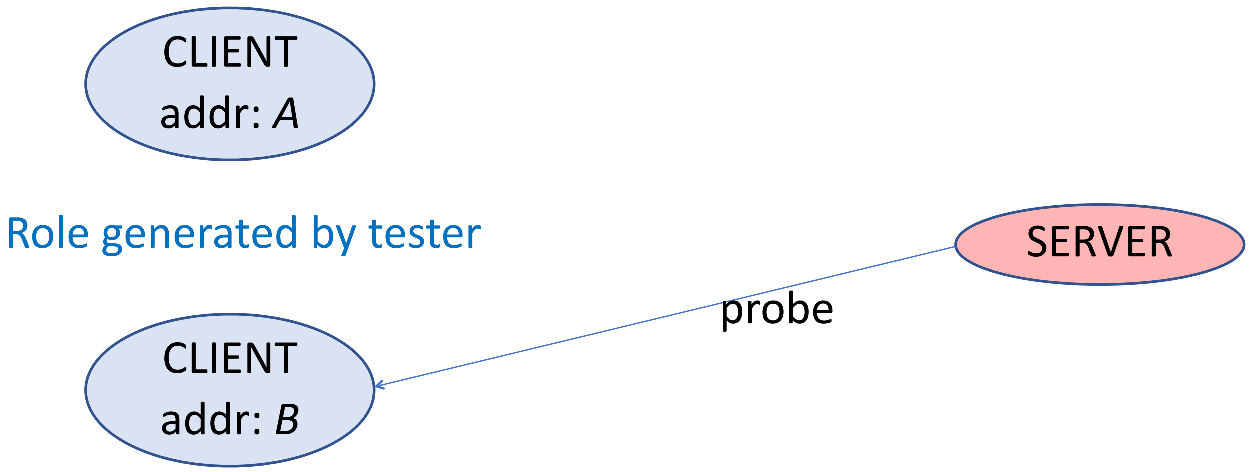
A generated DoS scenario



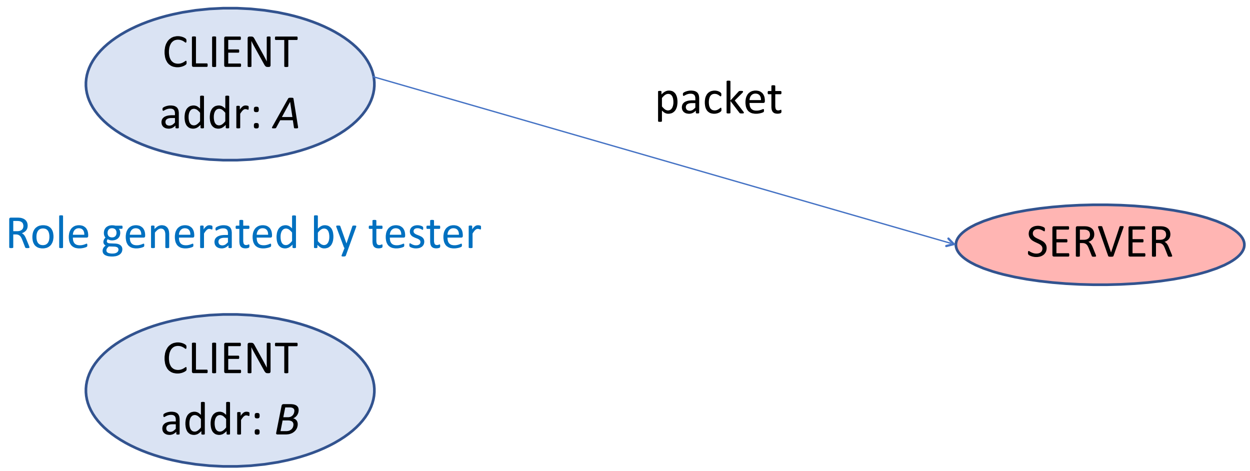
A generated DoS scenario



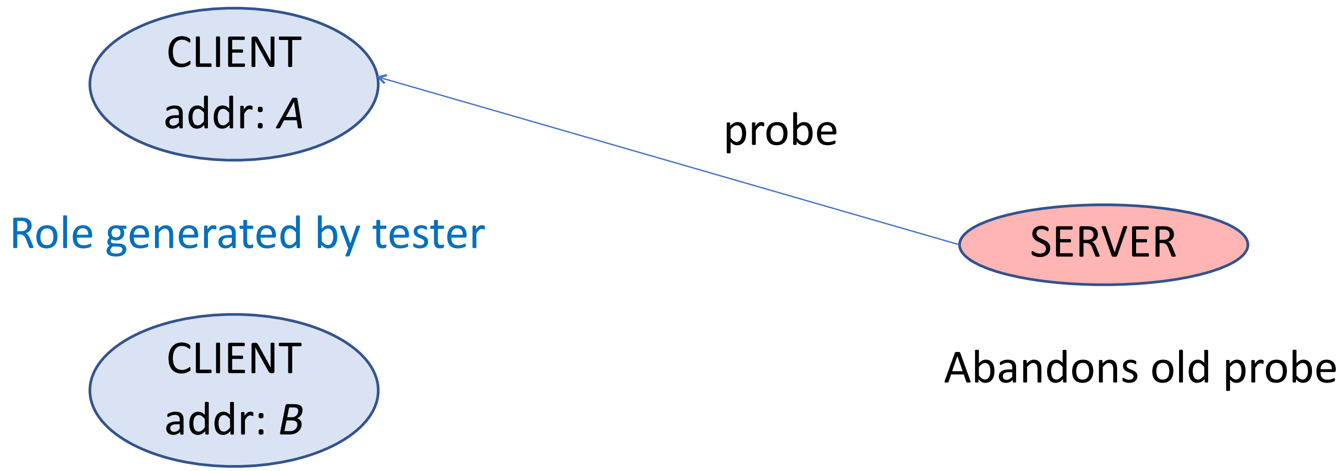
A generated DoS scenario



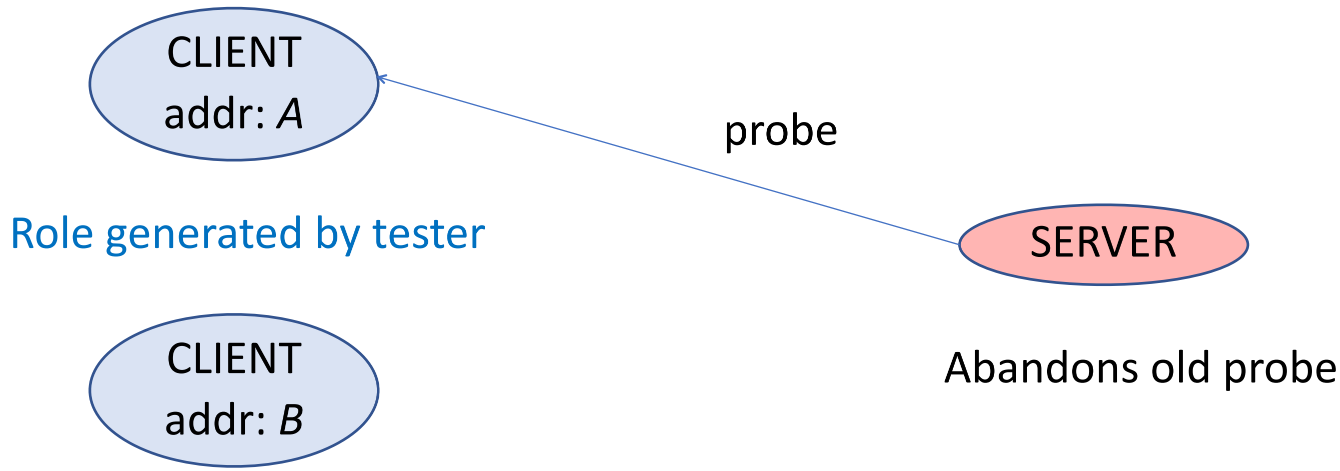
A generated DoS scenario



A generated DoS scenario



A generated DoS scenario



- Packets alternate from different address
  - Probe never finishes
  - Data stream is blocked indefinitely
- Attacker replaying packets from addr *B* can effect DoS

A heartbleed-style data leak

A server sent a STREAM data frame containing bytes read beyond the end of a buffer.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
<TITLE>PicoQuic HTTP 0.9 service</TITLE>
</HEAD><BODY>
<h1>Simple HTTP 0.9 Responder</h1>
<p>GET /, and GET index.html returns this text</p>
<p>Get /doc-NNNNN.html returns html document of length NNNNN bytes(decimal)</p>
<p>Get /doc-NNNNN also returns html document of length NNNNN bytes(decimal)</p>
<p>Get /doc-NNNNN.txt returns txt document of length NNNNN bytes(decimal)</p>
<p>Get /NNNNN returns html document of length NNNNN bytes(decimal)</p>
<p>Any other command will result in an error, and an empty response.</p>
<h1>Enjoy!</h1>
</BODY></HTML>
```

Adversarial stimulus (in flow control) resulted in finding a security flaw, even though security properties not specified.

A heartbleed-style data leak

A server sent a STREAM data frame containing bytes read beyond the end of a buffer.

```
<!DOCTYPE HTML PUBLIC "-//IETF//DTD HTML 2.0//EN">
<HTML>
<HEAD>
<TITLE>PicoQuic HTTP 0.9 service</TITLE>
</HEAD><BODY>
<h1>Simple HTTP 0.9 Responder</h1>
<p>GET /, and GET index.html returns this text</p>
<p>Get /doc-NNNNN.html returns html document of length NNNNN bytes(decimal)</p>
<p>Get /doc-NNNNN also returns html document of length NNNNN bytes(decimal)</p>
<p>Get /doc-NNNNN.txt returns txt document of length NNNNN bytes(decimal)</p>
<p>Get /NNNNN returns html document of length NNN^Z^X\200^Cl\357^@^D^@
```

Adversarial stimulus (in flow control) resulted in finding a security flaw, even though security properties not specified.

Conclusions

- A specification is a tool with a use. For QUIC:
  - Generate adversarial tests
  - Check compliance of implementations
  - Capture protocol knowledge from implementations
- This dictated form, content, process:
  - Compositional Assume/guarantee style
  - Deterministic monitor form
  - Iteration by testing implementations

A spec with a different purpose (e.g. proof lemma) may be very different

Conclusions

- A specification need not be complete or perfect to serve its function well
  - A modest specification effort pays significant dividends
- Specification is an iterative process
  - A spec must have corrective forces pushing on *both sides*
- Specifications can address significant pain points in protocol development
  - Capture knowledge in precise and actionable form
  - Faster approach to unit test development
  - Avoid release of non-compliant implementations.
- Ultimately, spec can be used for formal proof
  - Testing provides connection to informal implementations