Fizz: a C++14 implementation of TLS 1.3

Subodh Iyengar (subodh@fb.com), Kyle Nekritz (knekritz@fb.com)

Fizz

Fizz is a new library that implements the TLS 1.3 standard in C++14. Fizz has been deployed at Facebook to our load balancers, mobile apps, and backend services.

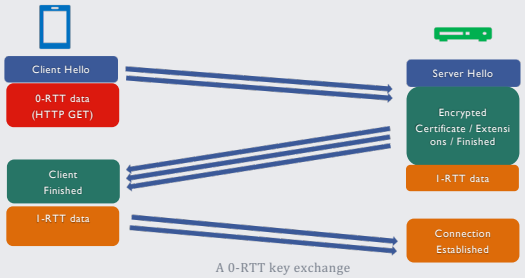
More than 50% of Facebook's internet traffic uses TLS 1.3 and Fizz. We believe Fizz is the largest current deployment of TLS 1.3 on the internet. Fizz has improved the CPU utilization, memory, and latency for several of Facebook's services. Fizz provides tight integrations with folly's AsyncTransport interface and several features needed by modern servers and clients.

Transport Layer Security 1.3

Transport Layer Security (TLS, previously known by SSL) is a widely used protocol that protects data in transport. The next generation of TLS, TLS 1.3, was standardized in August as RFC 8446. It re-designs TLS to support new features like:

- 0-RTT connection setup
- A simpler and more secure TLS handshake
- Additional privacy by encrypting client and server certificates

Deploying TLS 1.3 is important to the security, performance and privacy of the internet.



A 0-RTT key exchange

Facebook for Android	Latency Reduction at 50th percentile	Latency Reduction at 75th percentile
Establishing connection	46%	45%
HTTP request with new connection	10%	9%

Results deploying Fizz to our mobile apps

Compile time state machine verification

The TLS state machine is complex. State machine bugs can cause fatal vulnerabilities in an implementation, for example, the CCS vulnerability in OpenSSL. Fizz uses an explicit state machine for TLS 1.3. New state machine handlers are defined as:

```
FIZZ_DECLARE_EVENT_HANDLER(
    StateEnum::ExpectingClientHello,  // Current state
    Event::ClientHello,               // The message event
    // Allowed state transitions from the current state
    StateEnum::ExpectingClientHello,  StateEnum::ExpectingCertificate,
    StateEnum::ExpectingFinished,  StateEnum::AcceptingEarlyData);
```

As shown, receiving a message can cause transitions to different states. For example, shown in pseudo-code:

```
(ExpectingClientHello, ClientHello) → Handler
  if (isAcceptable(ClientHello)):
    transition<ExpectingFinished>();
  else:
    sendRetry();
    transition<ExpectingClientHello>();
```

This can cause a mismatch between the explicit state machine and the code which could cause a bug. To prevent mismatches, we implement a mechanism to verify that the code does not violate the explicit state machine. FIZZ_DECLARE_EVENT_HANDLER creates a unique EventHandler type for each handler.

```
template <typename SM,
          typename SM::StateEnum state, typename SM::Event event,
          typename SM::StateEnum... AllowedStates>
class EventHandler {
    template <typename SM::StateEnum to>
    static void transition(typename SM::State& stateStruct) {
        static_assert(
            Or<StateSame<SM, to, AllowedStates>...>::value, "Transition invalid");
        stateStruct.state() = to;
    }};
```

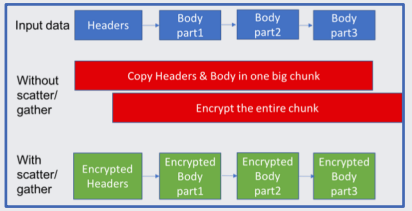
The EventHandler is templated on the current state, Event, and the AllowedStates. When a handler needs to transition to another state, the Transition function is invoked. Because of this construction, instantiation of the Transition function for all the States not in AllowedStates will cause a compile error. For example, the following code will not compile, because ExpectingCertVerify is not in the Allowed states:

```
(ExpectingClientHello, ClientHello) → Handler
    transition<ExpectingCertVerify>(); // <---- compile error
```

Scatter-gather I/O support

TLS APIs in several libraries require a user to supply a contiguous chunk of memory. TLS libraries encrypt this data and write it to a socket. This requires an additional copy, which hurts performance.

Fizz has first-class support for scatter/gather I/O APIs, as all its APIs accept the scatter/gather abstraction of folly IOBufs. It will also encrypt data in-place. In synthetic benchmarks with our load balancer we see that Fizz is 10% faster than our previous TLS libraries, which we've done a lot of work to optimize. Fizz also stores less state per connection, and we see this reduces the RSS of services.



Asynchronous APIs

In large scale TLS deployments, servers are all over the world. It's common for a TLS server to be in one place while the TLS certificate signing keys are provided by a secure service all the way across the world. To allow evented servers to RPC other services in The middle of a handshake, Fizz provides a future based API using folly futures so that callbacks can return futures without blocking the event thread.

```
server->setCertificate(AsyncCert cert);

onClientHello(...):
  auto asyncCert = getCertificate(...);
  asyncCert->signFuture(...).then([](auto& signature) {
      sendCertificateVerify(signature);
  });
```

Open source

- BSD licensed
- Star us on github: https://github.com/facebookincubator/fizz
- Builds and tested on android, iOS, Mac, and x86.
- Supports other features like token binding, and soon certificate compression.
- Reach us over email or twitter (@subiye, @kylenekritz)