Write your own QUIC application

To write an application that uses QUIC, you will have to subclass some of the QUIC classes as well as creating some new classes. This document tries to provide an introduction on how to use QUIC in your own application, by using the QUIC performance test client and server as example.

In the example, epoll is used to get notified on available incoming packets or when new data can be written to the connection. This is because performance test client and server are based on the example in the Chromium project in the net/tools/quic folder. It is not necessary to use epoll when writing an application that uses QUIC.

New classes

To facilitate the handling of all the QUIC classes, you will probably want to create classes that wrap the client and the server functionality. Also, you need to create at least the QuicPacketWriter helper class.

QUIC client application

You will need a class that will create a session (e.g. QuicClient), connection and a packet writer and connects all those classes. It must also read the network packages and dispatches them to the session.

Creating a QUIC client an be divided into several steps. First you have to determine the IP address to connect to, the server id and the QUIC versions you want to use. The server id identifies the server to connect to and consists of a scheme (http or https), a host and a port. The example client receives those in the constructor and stores them in his attributes (including the epoll server).

```
QuicClient::QuicClient(IPEndPoint server_address,
                       const QuicServerId& server_id,
                       const QuicVersionVector& supported_versions,
                       EpollServer* epoll_server)
    : server_address_(server_address),
      server_id_(server_id),
      epoll_server_(epoll_server),
      ...
      supported_versions_(supported_versions) {
}
```

Then you need to open the UDP socket an register it with the epoll server.

```
bool QuicClient::Initialize() {

  ...

  if (!CreateUDPSocket()) {
    return false;
  }

  epoll_server_->RegisterFD(fd_, this, kEpollFlags);
  initialized_ = true;
  return true;
}
```

With the UDP socket open, we can open the QUIC connection. We need to first create a QuickPacketWriter and a packet writer factory and initialize a new QuicConnection with it. Then we can create a new QuicClientSession using the connection. The session must then be initialized and finally we can start the crypto handshake and wait for it to finish.

```
bool QuicClient::Connect() {
  QuicPacketWriter* writer = new QuicDefaultPacketWriter(fd_);

  DummyPacketWriterFactory factory(writer);

  session_.reset(new QuicClientSession(
     config_,
     new QuicConnection((QuicConnectionId) QuicRandom::GetInstance()->RandUint64(),
                     server_address_, helper_.get(),
                     factory,
                     /* owns_writer= */ false, Perspective::IS_CLIENT,
                     server_id_.is_https(), supported_versions_)));

  // Reset |writer_| after |session_| so that the old writer outlives the old
  // session.
  if (writer_.get() != writer) {
    writer_.reset(writer);
  }
  // Initialize the session
  session_->InitializeSession(server_id_, &crypto_config_);
  // Start the crypto handshake
  session_->CryptoConnect();

  // Wait for the crypto handshake to finish
  while (!session_->IsEncryptionEstablished() &&
        session_->connection()->connected()) {
    epoll_server_->WaitForEventsAndExecuteCallbacks();
  }
  return session_->connection()->connected();
}
```

For this to work, the client must also read the UDP packets when it is notified by the epoll server. When an QUIC packet is read successfully it has to be dispatched to the QuicConnection using the ProcessUdpPacket method.

```
void QuicClient::OnEvent(int fd, EpollEvent* event) {
 if (event->in_events & EPOLLIN) {
   while (connected() && ReadAndProcessPacket()) {
   }
 }
 ...
}

bool QuicClient::ReadAndProcessPacket() {
 // Allocate some extra space so we can send an error if the server goes over
 // the limit.
 char buf[2 * kMaxPacketSize];

 IPEndPoint server_address;
 IPAddressNumber client_ip;

 // Read one UDP packet and the IP addresses
 int bytes_read = QuicSocketUtils::ReadPacket(
      fd_, buf, arraysize(buf),
      overflow_supported_ ? &packets_dropped_ : nullptr, &client_ip,
      &server_address);

 if (bytes_read < 0) {
   return false;
 }

 // Each UDP packet contains one encrypted QUIC packet,
 // so we initialize an instance of QuicEncryptedPacket with the content of the UDP packet
 QuicEncryptedPacket packet(buf, bytes_read, false);

 IPEndPoint client_address(client_ip, client_address_.port());
 // Pass the encrypted QUIC packet to the QuicConnection
 session_->connection()->ProcessUdpPacket(client_address, server_address, packet);
 return true;
}
```

To send or receive data, a QUIC stream has to be created. The new stream has to be created in the session, as the protected ActivateStream method has to be called with the new stream.

```
QuicClientStream* QuicClient::CreateClientStream() {
 if (!connected()) {
   return nullptr;
 }
 return session_->CreateClientStream();
}

QuicClientStream* QuicClientSession::CreateClientStream() {
 QuicDataStream* stream = new QuicClientStream(GetNextStreamId(), this);
 ActivateStream(stream);
 return (QuicClientStream*) stream;
}
```

Data can then be written to the stream using the WriteOrBufferData method.

QUIC server application

The server side is more complex, as it has to handle multiple sessions with multiple connections. Therefore, the server class (QuicServer) usually uses a dispatcher class (QuicDispatcher). It will handle the reading of network packets similar to the client (using epoll) but instead of handing it to the connection directly it will hand it to the dispatcher.

First you need a QuicConfig that will late be passed to the sessions, a QuicCryptoServerConfig that must be initialized with a default config and a vector of QUIC versions that should be supported.

```
QuicServer::QuicServer(const QuicConfig& config,
                       const QuicVersionVector& supported_versions)
   : ...,
     config_(config),
     crypto_config_(kSourceAddressTokenSecret, QuicRandom::GetInstance()),
     supported_versions_(supported_versions) {

 ...

 QuicEpollClock clock(&epoll_server_);

 scoped_ptr<CryptoHandshakeMessage> scfg(
     crypto_config_.AddDefaultConfig(
         QuicRandom::GetInstance(), &clock,
         QuicCryptoServerConfig::ConfigOptions()));
}
```

Then you have to start to listen on a UPD socket and bind to the requested address. With the socket you can then initialize a new dispatcher.

```
bool QuicServer::Listen(const IPEndPoint& address) {
 port_ = address.port();
 int address_family = address.GetSockAddrFamily();
 fd_ = socket(address_family, SOCK_DGRAM | SOCK_NONBLOCK, IPPROTO_UDP);

 ...

 sockaddr_storage raw_addr;
 socklen_t raw_addr_len = sizeof(raw_addr);
 address.ToSockAddr(reinterpret_cast<sockaddr*>(&raw_addr), &raw_addr_len);

 rc = bind(fd_,reinterpret_cast<const sockaddr*>(&raw_addr), sizeof(raw_addr));
```

```
  ...

  epoll_server_.RegisterFD(fd_, this, kEpollFlags);
  dispatcher_.reset(new QuicDispatcher(
      config_,
      &crypto_config_,
      supported_versions_,
      new QuicDispatcher::DefaultPacketWriterFactory(),
      new QuicEpollConnectionHelper(&epoll_server_)));
  dispatcher_->InitializeWithWriter(new QuicDefaultPacketWriter(fd_));

  return true;
}
```

Similar to the client, the server gets called by the epoll server when the socket is ready to read or write. The server can then read an UDP packet, create a QUIC packet from it and hand it to the dispatcher.

```
bool QuicServer::ReadAndDispatchSinglePacket(int fd,
                                             int port,
                                             ProcessPacketInterface* dispatcher,
                                             QuicPacketCount* packets_dropped) {
  // Allocate some extra space so we can send an error if the client goes over
  // the limit.
  char buf[2 * kMaxPacketSize];

  IPEndPoint client_address;
  IPAddressNumber server_ip;
  int bytes_read =
      QuicSocketUtils::ReadPacket(fd, buf, arraysize(buf),
                                  packets_dropped,
                                  &server_ip, &client_address);

  if (bytes_read < 0) {
    return false;  // We failed to read.
  }

  QuicEncryptedPacket packet(buf, bytes_read, false);

  IPEndPoint server_address(server_ip, port);
  dispatcher->ProcessPacket(server_address, client_address, packet);

  return true;
}
```

The dispatcher uses a framer to parse the QUIC packets which in turn will call the OnUnauthenticatedPublicHeader method of the dispatcher on reading frames.

```
void QuicDispatcher::ProcessPacket(const IPEndPoint& server_address,
                                   const IPEndPoint& client_address,
                                   const QuicEncryptedPacket& packet) {
  current_server_address_ = server_address;
  current_client_address_ = client_address;
  current_packet_ = &packet;
  // ProcessPacket will cause the packet to be dispatched in
  // OnUnauthenticatedPublicHeader, or sent to the time wait list manager
  // in OnAuthenticatedHeader.
  framer_.ProcessPacket(packet);
}
```

The dispatcher looks at the connection id in the public header to find an existing session (with the given connection id). When the dispatcher has determined the connection to which the QUIC packet belongs it will call the ProcessUdpPacket of the sessions connection. If there is no such session, the dispatcher will first create it.

```
bool QuicDispatcher::OnUnauthenticatedPublicHeader(
    const QuicPacketPublicHeader& header) {
  // The session that we have identified as the one to which this packet belongs.
  QuicServerSession* session = nullptr;
  QuicConnectionId connection_id = header.connection_id;
  SessionMap::iterator it = session_map_.find(connection_id);
  if (it == session_map_.end()) {

    ...

    // Create a new session if there are no errors (like unsupported QUIC version)
    session = AdditionalValidityChecksThenCreateSession(header, connection_id);
    if (session == nullptr) {
      return false;
    }
  } else {
    session = it->second;
  }

  session->connection()->ProcessUdpPacket(current_server_address_, current_client_address_, *current_packet_);

  // Do not parse the packet further.  The session methods called above have processed it completely.
  return false;
}
```

From there on, the process is the same as in the client.

When the server receives a new connection it will create a new session as shown above. When a server session receives a new stream (a stream opened by the client), the CreateIncomingStream method of the session will be called.

```
QuicDataStream* QuicServerSession::CreateIncomingDataStream(QuicStreamId id) {
  QuicServerStream* stream = new QuicServerStream(id, this, helper_);
  return stream;
}
```

Here you can create an instance of your own stream class (a subclass of QuicDataStream). The stream class then implements the ProcessRawData method which is where you can determine how incoming data is handled.

```
uint32 QuicServerStream::ProcessRawData(const char* data, uint32 data_len) {
  // Return the number of bytes that where consumed
  return data_len;
}
```

QuicPacketWriter

To handle packet writing you will have to implement the QuicPacketWriter interface. The important method to implement is the WritePacket method. It takes a buffer of raw data, the length of the buffer, the own IP address and the IP address of the peer. It is responsible for writing the data to a network socket.

```
WriteResult QuicDefaultPacketWriter::WritePacket(
    const char* buffer,
    size_t buf_len,
    const IPAddressNumber& self_address,
    const IPEndPoint& peer_address) {
  WriteResult result = QuicSocketUtils::WritePacket(
      fd_, buffer, buf_len, self_address, peer_address);
  if (result.status == WRITE_STATUS_BLOCKED) {
    write_blocked_ = true;
  }
  return result;
}
```

The packet writer is usually created in the QuicClient and QuicServer. It is then passed to and used by the QuicConnection to write QUIC packets to the network socket.

Epoll server

To asynchroniously handle the network connection, epoll can be used. An implementation of an epoll server is already present in the chromium source code in the net/tools/quic folder. This example contains a copy of this epoll server.

Sub classes

For each the server and the client, you need at least to subclass the QuicSession class and likely the QuicDataStream class.

Session sub classes

All QuicSession sub classes have to implement the following methods as they are purely virtual in the base class.

- *GetCryptoStream*

  Returns the reserved crypto stream. This stream has to be created in your class.

- *CreateIncomingDataStream*

  As explained in the QUIC server application section, this method will be called when a new stream should be created because the peer opened a new stream. If you don't want incoming streams to be created, you can just return nullptr.

- *CreateOutgoingDataStream*

  Same as CreateIncominDataStream but for streams you open yourself (and not the peer). Beware that if you create an instance of a stream you have to call the ActivateStream method of the session with the stream or else the stream will never receive any packets!

The last two methods are called when a new QuicDataStream (sub) class must be created. By implementing these methods the session can be made to use a custom QuicDataStream sub class. You can look at the QUIC server application section for an example.

The client session class inherits from QuicClientBaseSession. It has to additionally implement *OnProofValid* and *OnProofVerifyDetailsAvailable* as they are purely virtual in the base class. These have be implemented if you want to have secure connections but can be no-ops otherwise. You also have to implement the creation of the crypto stream and start the crypto negotiation.

```
void QuicClientSession::InitializeSession(
    const QuicServerId& server_id,
    QuicCryptoClientConfig* crypto_config) {
  crypto_stream_.reset(
      new QuicCryptoClientStream(server_id, this, nullptr, crypto_config));
  QuicClientSessionBase::InitializeSession();
}
```

The server session class inherits directly from QuicSession. Here too, you have to implement handling of the crypto stream. Most likely you want to override CreateIncomingDataStream and make it return an instance of your own stream class.

```
QuicDataStream* QuicServerSession::CreateIncomingDataStream(QuicStreamId id) {
  return new QuicServerStream(id, this);
}
```

Stream sub classes

In the client, you can probably use the existing QuicDataStream.

In the server you want to inherit from QuicDataStream. On creation of your stream class you **have to** call sequencer()->FlushBufferedFrames() to unblock the sequencer. Otherwise it will not pass new data to the stream until the headers of a SPDY request have been received. You can then override ProcessRawData which will receive all data for this stream.

Open Questions

- Do we need the TimeWaitListManager to have a valid QUIC implementation?
- Why do we have to create the CryptoStream our self?
- Where can you set the certificates you accept?