CS434/534: Topics in Network Systems

Reliable, High-Performance, Secure Network Systems Transport: Secure Transport Handshake, Substream Transport

Oct. 28, 2021

Y. Richard Yang
Computer Science Department, Yale University
Email: yry@cs.yale.edu

http://zoo.cs.yale.edu/classes/cs434/
### Programming project 2 design meetings

**proj-sol:**

- FishThread.java: 129, 400, 3045
- Node.java: 83, 250, 1145
- PingRequest.java: 51, 167, 1145
- SimpleTCPSockSpace.java: 81, 605, 2106
- TCPManager.java: 889, 3088, 26381
- TCPSock.java: 60, 149, 1316
- TCPSockID.java: 123, 382, 3866
- TransferClient.java: 147, 500, 5059
- TCPSock.java: 181, 605, 5248
- Node.java: 51, 167, 1145
- PingRequest.java: 50, 128, 909
- TCPSock.java: 132, 460, 3146
- TransferClient.java: 123, 382, 3866
- TransferServer.java: 147, 500, 5059

**proj:**

- FishThread.java: 129, 400, 3045
- Node.java: 341, 1301, 11313
- PingRequest.java: 51, 167, 1145
- TCPManager.java: 50, 128, 909
- TCPSock.java: 132, 460, 3146
- TransferClient.java: 123, 382, 3866
- TransferServer.java: 147, 500, 5059

**total:**

- proj-sol: 2051, 6998, 61039
- proj: 973, 3338, 28483

Part 1 (reliable transport)+Part 2 (congestion control/AIMD code size)
Recap: MPTCP

- **Goals (informal)**
  - Improve throughput and robustness
  - Do no harm

- **Design**
  - Each ACK on subflow $r$, increase the window $w_r$ by $\min(a/w_{total}, 1/w_r)$
  - Each loss on subflow $r$, decrease the window $w_r$ by $w_r/2$
Recap: Subflow Tput vs TCP Tput

\[ \sum_{r \in R} \frac{w_r}{RTT_r} \geq \max_{r \in R} \frac{w_r^{TCP}}{RTT_r} \]

\[
(1 - p_r) \min\left( \frac{a}{\hat{w}_{total}}, \frac{1}{\hat{w}_r} \right) = pr\frac{\hat{w}_r}{2}
\]

\[
\min\left( \frac{a}{\hat{w}_{total}}, \frac{1}{\hat{w}_r^2} \right) = pr\frac{1}{2(1-pr)}
\]

\[
\min\left( \frac{a}{\hat{w}_{total}}, \frac{1}{\hat{w}_r^2} \right) \approx \frac{p_r}{2} \quad \hat{w}_r^{tcp} = \sqrt{2/pr}
\]

\[
\min\left( \frac{a}{\hat{w}_{total}}, \frac{1}{\hat{w}_r^2} \right) = \frac{1}{\hat{w}_r^{tcpz}}
\]

\[
\max\left( \sqrt{\frac{a}{\hat{w}_{total}}}, \hat{w}_r \right) = \hat{w}_r^{tcp}
\]

\[
\max\left( \sqrt{\frac{\hat{w}_{total}}{a RTT^2}}, \frac{\hat{w}_r}{RTT_r} \right) = \frac{\hat{w}_r^{tcp}}{RTT_r}
\]
Recap: Achieving Improve Tput

\[ \sum_{r \in R} \frac{\hat{w}_r}{RTT_r} \geq \max_{r \in R} \frac{\hat{w}_r^{TCP}}{RTT_r} \]

\[
\max \left( \frac{\sqrt{\hat{w}_{total} \hat{w}_r}}{a \ RTT_r^2}, \frac{\hat{w}_r}{RTT_r} \right) = \frac{\hat{w}_r^{tcp}}{RTT_r}
\]

\[
\sum_{r \in R} \frac{\hat{w}_r}{RTT_r} \geq \max_{r} \sqrt{\frac{1}{a} \frac{\hat{w}_{total}}{RTT_r^2}}
\]

\[
1 \geq \max_{r} \sqrt{\frac{1}{a} \frac{\hat{w}_{total}}{RTT_r^2}} (\sum_{r \in R} \frac{\hat{w}_r}{RTT_r})^2
\]

\[
a = \max_{r} \frac{\hat{w}_{total} \hat{w}_r^2}{(\sum_{r \in R} \frac{\hat{w}_r}{RTT_r})^2}
\]
Recap: Transport Considering Security

- **Confidentiality** (The connection is private): only sender, intended receiver can “understand” msg
  - Typical solution: sender encrypts data using symmetric cryptography
  - => Need key agreement

- **Integrity** (The connection is reliable): sender, receiver want to ensure message not altered without detection
  - Typical solution: sender includes keyed message digest
  - => Need key agreement

- **Authenticity** (The peer's identity can be authenticated): sender, receiver can confirm origin of data
  - Typical solution: public key or pre-shared key
  - => Need handshake authentication protocol
Recap: RSA Public Key

1. Choose two large prime numbers \( p, q \) (e.g., 2048 bits each)

2. Compute \( n = pq, \quad z = (p-1)(q-1) \)

3. Choose \( e \) (with \( e < n \)) that has no common factors with \( z \).

4. Choose \( d \) such that \( ed - 1 \) is divisible by \( z \) (in other words: \( ed \mod z = 1 \)).

5. Public key is \( (n,e) \). Private key is \( (n,d) \).

A MUST CS read: https://people.csail.mit.edu/rivest/Rsapaper.pdf
RSA Uses

- **Encryption**: give message, \( m \), compute \((m < n; \text{otherwise, divide the message into blocks})\)
  - \( m^e \mod n \)

- **Authentication through challenge**
  - Alice chooses a secret random \( x \), computes \( x^e \), and asks Bob what \( x \) is
    - \((x^e)^d = x\)

- **Authentication through signature**
  - Alice chooses a random number \( x \), and asks Bob to compute \( x^d \), Alice verifies that \((x^d)^e = x\)

- **RSA security based on the hardness of discrete log**, 
  - Assume \( y = x^d \mod n \)
  - Given \( y \) and \( x \), it is difficult to compute \( d \)
Recap: Basic Authentication Protocol

- Basic requirements
  - [p1] What Alice receives is fresh
  - [p2] What Alice receives is only Bob can say

- Exercise:
  - chrome://settings/security
  - Manage Certificates
Building Block: Key Agreement using Public Key

- One more type of public key system: Diffie-Hellman (DH) => Key Agreement

- DH is also based on the hardness of discrete log:
  - given $q$, and $\alpha$, it is easy to calculate $\alpha^n \mod q$;
  - on the other hand, given $\alpha^n \mod q$, it is hard to get back $n$.

- Scheme
  - Public domain: $q$, $\alpha$
  - Private key: $n$
  - Public key: $\alpha^n \mod q$
  - Key agreement:
    - Alice wants to talk to Bob, Alice takes Bob’s public key $\alpha^{nb} \mod q$, raises it to her private key $na$’s power, $(\alpha^{nb})^n a \mod q$
Recap: TCP Transport => Secure Transport

Goal: Design transport (handshake and transport) on top of TCP so that

- [R1] Client ensures that the reply comes from the intended server (e.g., www.yale.edu)
- [R2] Key agreement, to use for encryption and integrity
- [R3] Server ensures that the client request is fresh
Exercise: R1 (Client ensures that the reply comes from the intended server (e.g., www.yale.edu))

- **Client**
  - Nonceₐ
  - Public Pₛ, Sign (Nonceₐ)
  - Certificate indicate Pₛ for server

- **Server**
Exercise R2: (Key Agreement)

\[ z = \alpha^{n_C n_S} \]

Data encrypt/mac using key \( z \)

Diffie-Hellman (DH)

\[ c = \text{Enc}(\text{key}, \text{pub}_S) \]

Shared key

\[ \text{key} = \text{decr} c \]

Data encrypt/mac using key

RSA
Exercise R3 (Server ensures that the client request is fresh)
Outline

- Admin and recap
- Reliable, High-Performance, Secure Network Transport
  - Overview
  - Basic reliable transport structures
  - Transport (congestion) rate control
  - Secure transport
    - Motivation
    - Background
    - Transport layer security (TLS)
Putting it Together

- Using `curl -v https://www.google.com`
- Capture traffic using `wireshark`

- See the high level messages from `curl`
- See details from `wireshark` messages

- A good online page: `https://tls.ulfheim.net/`
client_hello: version, session_id, supported cipher suites, nonce \text{Nc}

server_hello: version, session_id chosen cipher suite, nonce \text{Ns}

certificate: chain of X509v3 certificates

server_key_exchange: parameters, signature

certificate_request: type, authorities

server_hello_done: now is client’s turn

certificate: chain of X509v3 certificates

Client key_exchange;

master_key is generated

Exercise: Which authenthtication do we see?

Exercise: Which key agreement do we see?

Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

Change cipher suite and finish handshake protocol.

\text{TLS1.2 Handshake Protocol}

Note: Shaded transfers are optional or situation-dependent messages that are not always sent.
master_secret = PRF(pre_master_secret, "master secret",
ClientHello.random + ServerHello.random)
[0..47];

8.1.1. RSA

When RSA is used for server authentication and key exchange, a 48-
byte pre_master_secret is generated by the client, encrypted under
the server's public key, and sent to the server. The server uses its
private key to decrypt the pre_master_secret. Both parties then
convert the pre_master_secret into the master_secret, as specified
above.

8.1.2. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The
negotiated key (Z) is used as the pre_master_secret, and is converted
into the master_secret, as specified above. Leading bytes of Z that
contain all zero bits are stripped before it is used as the
pre_master_secret.

Note: Diffie-Hellman parameters are specified by the server and may
be either ephemeral or contained within the server's certificate.
client_hello: version, session_id, supported cipher suites, nonce Nc

server_hello: version, session_id
chosen cipher suite, nonce Ns

certificate: chain of X509v3 certificates

server_key_exchange: parameters, signature

certificate_request: type, authorities

server_hello_done: now is client’s turn

client_key_exchange;
master_key is generated

Exercise: where is R1 (client authenticates server)?
Exercise: where is R2 (key agreement)?
Exercise: where is R3 (server makes sure req fresh)?
TLS Layering


TLS/SSL Record Protocol

TCP

IP
TLS Record-Layer Packet Format

20: change_cipher
21: alert
22: handshake
23: application
TLS/SSL Record-Layer Packet Format: Content Types

(a) Change Cipher Spec Protocol

(b) Alert Protocol

(c) Handshake Protocol

(d) Other Upper-Layer Protocol (e.g., HTTP)
TLS/SSL Record Operation

- Application Data
- Fragment
- Compress
- Add MAC
- Encrypt
- Append SSL Record Header
Exercise: Overall (TLS1.2 + TCP) Message Flow before Data Can Be Sent?

Host A
SYN(seq=x)
ACK(seq=x), SYN(seq=y)
ACK(seq=y)
DATA(seq=x+1)
DATA(seq=y+1)

Host B
TLS1.2: Reuse Handshake

- TLS1.2 allows reuse the result of a handshake (benefits?)

**Handshake Protocol: Client Hello**
- Handshake Type: Client Hello (1)
- Length: 224
- Version: TLS 1.2 (0x0303)
- Random: 402dee0792b8d4b4e38a72f88d3a8566c3c4f7cf87480ff73e99c4d283b7
- Session ID Length: 0

**Handshake Protocol: Server Hello**
- Handshake Type: Server Hello (2)
- Length: 92
- Version: TLS 1.2 (0x0303)
- Random: 606a19740984e0f31327304acb8a4fc96ae96d223f3eb5e6444f574e4752
- Session ID Length: 32
- Session ID: a9368ca035307d7384f140804ed6fb5911608e533b00c6e43a1e4621
- Cipher Suite: TLS_ECDHE_ECDSA_WITH_CHACHA20_POLY1305_SHA256 (0xcca9)

“If the ClientHello.session_id was non-empty, the server will look in its session cache for a match. If a match is found and the server is willing to establish the new connection using the specified session state, the server will respond with the same value as was supplied by the client and the parties must proceed directly to the Finished messages.”
TLS1.2 Reuse + TCP
TLS1.2 -> TLS1.3

- A good read is RFC8446
- TLS1.3 still is not as widely deployed as TLS1.2
- Main changes to TLS1.2
  - Simpler, stronger cipher suites
  - Streamlining handshake, e.g.,
    - 0-RTT handshake

https://www.ssllabs.com/ssl-pulse/
TLS1.3 0-RTT

ClientHello
+ early_data
+ key_share*
+ psk_key_exchange_modes
+ pre_shared_key
(Application Data*)  """

ServerHello
+ pre_shared_key
+ key_share*
{EncryptedExtensions}
+ early_data*
{Finished}
"""

<-------- [Application Data*]

(EndOfEarlyData)
{Finished}  """

(Application Data)  """

[Application Data]  """

+ Indicates noteworthy extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages/extensions that are not always sent.

() Indicates messages protected using keys derived from a client_early_traffic_secret.

{} Indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

[] Indicates messages protected using keys derived from [sender]_application_traffic_secret_N.

Figure 4: Message Flow for a 0-RTT Handshake
TLS1.3 (Generic Pre-share Key)

```
struct {
    uint32 ticket_lifetime;
    uint32 ticket_age_add;
    opaque ticket_nonce<0..255>;
    opaque ticket<1..2^16-1>;
    Extension extensions<0..2^16-2>;
} NewSessionTicket;
```

extensions: A set of extension values for the ticket. The "Extension" format is defined in Section 4.2. Clients MUST ignore unrecognized extensions.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.18). It contains the following value:

max_early_data_size: The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only

---

**Figure 3: Message Flow for Resumption and PSK**
TLS1.3 0-RTT Handshake (Offline)

Figure 4: Timeline of QUIC’s initial 1-RTT handshake, a subsequent successful 0-RTT handshake, and a failed 0-RTT handshake.
Outline

- Admin and recap
- Reliable, High-Performance, Secure Network Transport
  - Overview
  - Basic reliable transport structures (TCP)
  - Transport (congestion) rate control
  - Secure transport
    - Motivation
    - Background
    - Transport layer security (TLS)
  - HTTP + TLS + TCP and optimizations
Exercise: HTTP + TLS + TCP

- Vast majority of Internet traffic is HTTP + TLS + TCP

- How much time to download an average Web page if download one object at a time?
  - Average Web requests see https://httparchive.org/reports/page-weight#reqTotal
HTTP/1.0 => HTTP/1.1 Keep-Alive

Per-request

Keep alive (Share handshake)

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
HTTP/1.0 => HTTP/1.1 Keep-Alive + Pipelining

Per-request

Keep alive (Share Handshake)

Pipelining (Parallel Request)

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
HTTP/1.1 => HTTP/2 Basic Idea: Sequential (HTTP/1.1) => Concurrent Response (HTTP/2)

Data flows from sequential to parallel: two requests can be served concurrently.

Source: http://chimera.labs.oreilly.com/books/1230000000545/ch11.html
HTTP/2 Demo

- Demo: https://http2.akamai.com/demo
HTTP/2 vs HTTP/1.1 Demo

HTTP/2 is the future of the Web, and it is here!

Your browser supports HTTP/2!

This is a demo of HTTP/2's impact on your download of many small tiles making up the Akamai Spinning Globe.

HTTP/1.1

Latency: ms
Load time: 4.06s

HTTP/2

Latency: ms
Load time: 1.10s
HTTP/1.1 => HTTP/2 Basic Idea: Sequential (HTTP/1.1) => Concurrent Response (HTTP/2)

Exercise: Major issue with serving concurrent requests in a single HTTP connection?
HTTP/2 High-Level Structure

- Application-layer (HTTP) multiplexing of concurrent reqs
  - Each client request and its response is assigned an individual stream
HTTP/2 High-Level Structure

- Application-layer (HTTP) multiplexing of concurrent requests
  - Messages of a single HTTP/2 connection divided into frames (chunks), with each frame belongs to one stream

HTTP/2 Binary Framing

<table>
<thead>
<tr>
<th>Bit</th>
<th>0..7</th>
<th>8..15</th>
<th>16..23</th>
<th>24..31</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>Flags</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>R</td>
<td>Stream Identifier</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>Frame Payload</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

HTTP/1.1 Request/Response => HTTP/2 Frames

GET /resource HTTP/1.1. HEADERS
Host: example.org
Accept: image/jpeg

+ END_STREAM
+ END_HEADERS
:method = GET
:scheme = https
:path = /resource
:host = example.org
:accept = image/jpeg

HTTP/1.1 304 Not Modified
ETag: "xyzzy"
Expires: Thu, 23 Jan ...

==> + END_STREAM
+ END_HEADERS
:status = 304
:etag = "xyzzy"
:expires = Thu, 23 Jan ...
### HTTP/1.1 Request/Response

<table>
<thead>
<tr>
<th>POST /resource HTTP/1.1</th>
<th>HEADERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Host: example.org</td>
<td>=&gt; END_STREAM</td>
</tr>
<tr>
<td>Content-Type: image/jpeg</td>
<td>=&gt; END_HEADERS</td>
</tr>
<tr>
<td>Content-Length: 123</td>
<td>:method = POST</td>
</tr>
<tr>
<td></td>
<td>:path = /resource</td>
</tr>
<tr>
<td></td>
<td>:scheme = https</td>
</tr>
<tr>
<td>{binary data}</td>
<td>CONTINUATION</td>
</tr>
</tbody>
</table>

| DATA |
| + END_HEADERS |
| content-type = image/jpeg |
| host = example.org |
| content-length = 123 |

### HTTP/1.1 200 OK

<table>
<thead>
<tr>
<th>HEADERS</th>
</tr>
</thead>
<tbody>
<tr>
<td>=&gt; END_STREAM</td>
</tr>
<tr>
<td>+ END_HEADERS</td>
</tr>
</tbody>
</table>

| :status = 200 |
| content-type = image/jpeg |
| content-length = 123 |

| DATA |
| + END_STREAM |
| {binary data} |
HTTP/2 Server Push

- Goal: server has access to content and hence can predict before the client that some content will be needed and can push to client before client request

HTTP/2 connection

- stream 1: /page.html (client request)
- stream 2: /script.js (push promise)
- stream 4: /style.css (push promise)
HTTP/2 Server Push (Details)

- Server push is semantically equivalent to a server responding to a request
  - HTTP/2: request is sent by the server, as a PUSH_PROMISE frame
- Promised requests MUST be cacheable (see [RFC7231], Section 4.2.3), MUST be safe (see [RFC7231], Section 4.2.1), and MUST NOT include a request body.
- Client determines, for any reason, that it does not wish to receive the pushed response from the server or if the server takes too long to begin sending the promised response, the client can send a RST_STREAM frame, using either the CANCEL or REFUSED_STREAM code and referencing the pushed stream's identifier.
Figure 2: Stream States
HTTP/2 Additional Details: Header Compression (Offline)

- Details see HPACK [https://tools.ietf.org/html/rfc7541]
HTTP/2 Additional Details: Stream Dependency and Weights (Offline)

- Details see HTTP/2 RFC
HTTP/2 -> QUIC/HTTP/3

- Integrate TLS directly into transport (combine TCP handshake + TLS handshake)
- Remove head of line blocking (sequential)

TCP segment 3 is lost in transit.

OS will not deliver data in later segments to application until after segment 3 is retransmitted.

(eg., three different web images)
The Network Transport Journey

- Packet can be lost/corrupted
  - Retransmission, proper RTO
- Single packet stop-and-wait is inefficient
  - Sliding window protocol, fast retransmit, flow control
- Transport (connection) state need proper management
  - Three-Way-Handshake, Four Way Teardown
- Transport need efficient, fair sharing network-wide resources
  - MI/AI+MD model => slow start (MI), AIMD algorithm
  - Improve efficiency and robustness of resource use => multi-path TCP
  - Improve efficiency for fast, long-distance links w/ limited buffer => Cubic+friendly
  - Achieve both bandwidth efficiency and low latency (+headroom) => high rate low delay control
- Transport packets can be intercepted, modified or fabricated
  - Secure handshake, encryption/integrity protection using TLS; using freshness to reason about authentication
- A single Web transaction (page) includes many requests, request per connection is inefficient
  - Sequential HTTP request transport => concurrent HTTP/2 request transport (multi-streams)
- Redundant (TLS+TCP) handshake, single TCP connection can have head of line blocking when serving multiple packets of multiple streams
  - QUIC combining TLS+TCP handshake, independent streams not embedded in a single TCP stream
Backup Slides
**Head-of-line blocking** *(in general)*

- HOL blocking is when an item at the head a queue unnecessarily blocks items behind it.
  - At right, two input ports are trying to use Output 4.
  - If Input 3 is chosen to proceed, then Input 1 will have to wait.
  - Thus, the next item in Input 1's queue (destined to the idle Output 3) is stuck waiting for no good reason.

- The fundamental problem is that we're using a FIFO queue for items with no ordering dependence. We should pop whichever item is ready.
QUIC Protocol

- Run on top of UDP
- Connection ID on top of UDP
- Sequential transport at stream level, not connection level
- Details see QUIC draft

19. Frame Types and Formats
   19.1. PADDING Frames
   19.2. PING Frames
   19.3. ACK Frames
   19.3.1. ACK Ranges
   19.3.2. ECN Counts
   19.4. RESET_STREAM Frames
   19.5. STOP_SENDING Frames
   19.6. CRYPTO Frames
   19.7. NEW_TOKEN Frames
   19.8. STREAM Frames
   19.9. MAX_DATA Frames
   19.10. MAX_STREAM_DATA Frames
   19.11. MAX_STREAMS Frames
   19.12. DATA_BLOCKED Frames
   19.13. STREAM_DATA_BLOCKED Frames
   19.14. STREAMS_BLOCKED Frames
   19.15. NEW_CONNECTION_ID Frames
   19.16. RETIRE_CONNECTION_ID Frames
   19.17. PATH_CHALLENGE Frames
   19.18. PATH_RESPONSE Frames
   19.19. CONNECTION_CLOSE Frames
   19.20. HANDSHAKE_DONE Frames
   19.21. Extension Frames
HTTPS Exercises

- Why does ClientHello has server name (analogy in HTTP)?
- How does the connection use http/2 instead of http/1.1?