

# Middleware Architectures 1

## Lecture 3: Communication Protocols

**doc. Ing. Tomáš Vitvar, Ph.D.**

tomas@vitvar.com • @TomasVitvar • <https://vitvar.com>



Czech Technical University in Prague

Faculty of Information Technologies • Software and Web Engineering • <https://vitvar.com/lectures>



Modified: Sun Nov 19 2023, 19:53:57  
Humla v1.0

## Overview

- Introduction to Application Protocols
- Introduction to HTTP
- Security

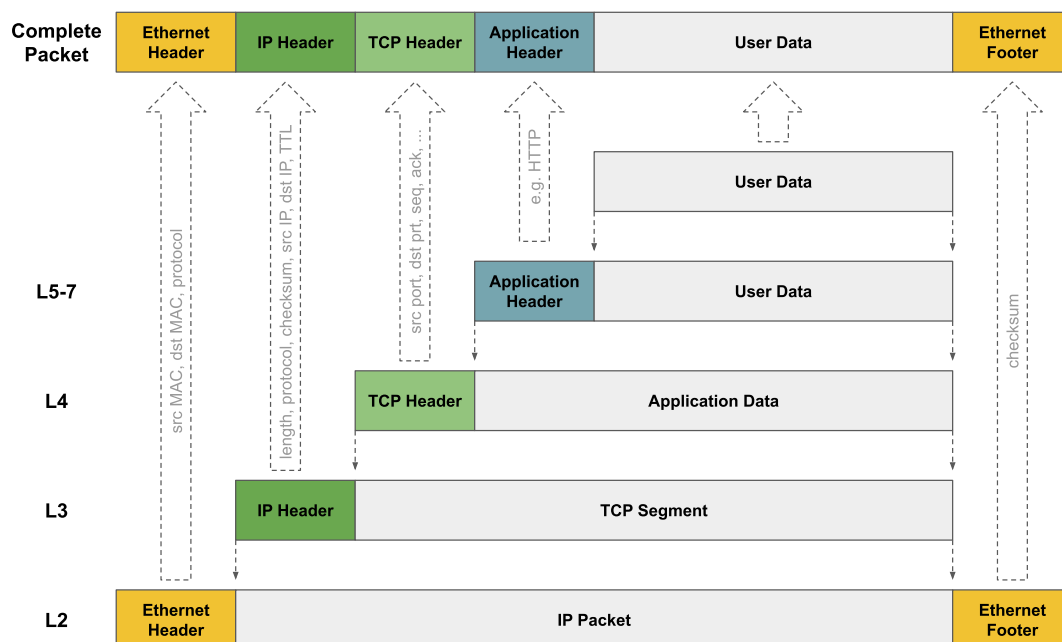
# Application Protocols

- Remember this

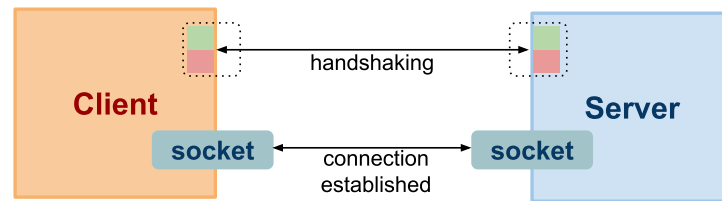


- App protocols mostly on top of the TCP Layer
  - use *TCP socket for communication*
- Major protocols
  - *HTTP – most of the app protocols layered on HTTP*
    - *widely spread*
  - *RMI – Remote Method Invocation*
    - *Java-specific; vendor-interoperability problem*
    - *may use HTTP underneath (among other things)*
  - *XML-RPC and SOAP – Remote Procedure Call and SOAP*
    - *HTTP-based*
  - *WebSocket – new protocol part of HTML5*

# Anatomy of a Packet



# Socket

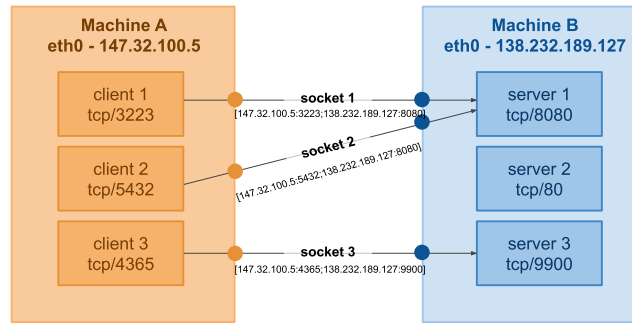


- Handshaking (connection establishment)
  - The server listens at `[dst_ip, dsp_port]`
  - Three-way handshake:
    - the client sends a connection request with TCP flags (SYN,  $x=\text{rand}$ )
    - the server responds with its own TCP flags (SYN ACK,  $x+1$   $y=\text{rand}$ )
    - the client acknowledges the response, can send data along (ACK,  $y+1$   $x+1$ )
  - Result is a socket (virtual communication channel) with unique identification:  
`socket=[src_ip,src_port;dst_ip,dst_port]`
- Data transfer (resource usage)
  - Client/server writes/reads data to/from the socket
  - TCP features: reliable delivery, correct order of packets, flow control
- Connection close

# New Connection Costs

- Creating a new TCP connection is expensive
  - It requires to complete a full roundtrip
  - It is limited by a network latency, not bandwidth
- Example
  - Distance from London to New York is approx. 5500 km
  - Communication over a fibre link will take at least 28ms one way
  - Three-way handshake will take a minimum of 56ms
- Connection reuse is critical for any app running over TCP
  - HTTP Keep-alive
  - HTTP pipelining
- TCP Fast Open (TFO)
  - TFO allows to speed up the opening of successive TCP connections
  - TCP cookie stored on the client that was established on initial connection
  - The client sends the TCP cookie with SYN packet
  - The server verifies the TCP cookie and can send the data without final ACK
  - Can reduce network transaction latency by 15%
  - TFO is supported by Linux in 3.7+ kernels

# Addressing in Application Protocol



- IP addressing: IP is an address of a machine interface
  - A machine can have multiple interfaces (*eth0, eth1, bond0, ...*)
- TCP addressing: TCP port is an address of an app running on a machine and listening on a machine interface
  - Multiple applications with different TCP ports may listen on a machine interface
- Application addressing
  - Additional mechanisms to address entities within an application
  - They are out of scope of IP/TCP, they are app specific
    - for example, Web apps served by a single Web server

## Overview

- Introduction to Application Protocols
- **Introduction to HTTP**
  - State Management
- Security

# Hypertext Transfer Protocol – HTTP

- Application protocol, basis of Web architecture
  - *Part of HTTP, URI, and HTML family*
  - *Request-response protocol*
- One socket for single request-response
  - *original specification*
  - *have changed due to performance issues*
    - *many concurrent requests*
    - *overhead when establishing same connections*
    - *HTTP 1.1 offers persistent connection and pipelining*
    - *Domain sharding*
- HTTP is stateless
  - *Multiple HTTP requests cannot be normally related at the server*
    - *"problems" with state management*
    - *REST goes back to the original HTTP idea*

## HTTP Request and Response

### ● Request Syntax

```
method uri http-version <crLf>
(header : value <crLf>)*
<crLf>
[ data ]
```

### ● Response Syntax

```
http-version response-code [ message ] <crLf>
(header : value <crLf>)*
<crLf>
[ data ]
```

### ● Semantics of terms

```
method      = "GET" | "POST" | "DELETE" | "PUT" | "HEAD" | "OPTIONS"
uri         = [ path ] [ ";" params ] [ "?" query ]
http-version = "HTTP/1.0" | "HTTP/1.1"
response-code = valid response code
header : value = valid HTTP header and its value
data        = resource state representation (hypertext)
```

## Persistent connections

- Persistent HTTP connection = HTTP keepalive
  - TCP established connection used for multiple requests/responses
  - Avoids TCP three-way handshake to be performed on every request
  - Reduces latency
  - FIFO queuing order on the client (request queuing)
    - dispatch first request, get response, dispatch next request
- Example: **GET /html**, **GET /css**
  - server processing time 40ms and 20ms respectively
- Without HTTP keepalive
  - three-way handshake 84ms before the data is received on the server
  - Response received at 152ms and 132ms respectively
  - The total time is 284ms
- HTTP keepalive
  - One TCP connection for both requests
  - In our example this will save one RTT, i.e. 56ms
  - The total time will be 228ms

## Persistent connections savings

- Each request needs
  - Without keepalive, 2 RTT of latency
  - With keepalive, the first request needs 2 RTT, a following request needs 1 RTT
- Savings for **N** requests: **(N-1) × RTT**
- Average value of **N** is 90 requests for a Web app
  - Measured by HTTP Archive (<http://httparchive.org>) as of 2013
  - Average Web application is composed of 90 requests fetched from 15 hosts
    - HTML: 10 requests
    - Images: 55 requests
    - Javascript: 15 requests
    - CSS: 5 requests
    - Other: 5 requests

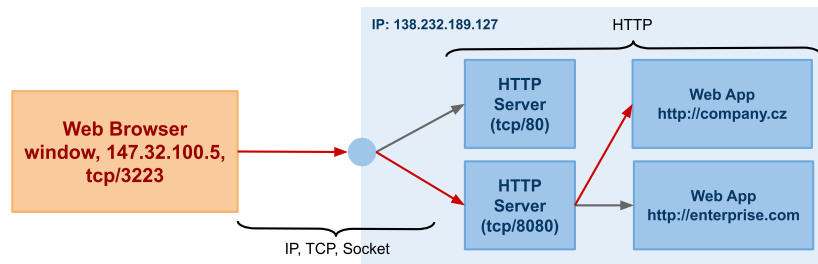
## HTTP pipelining

- Important optimization – response queuing
  - Allows to relegate FIFO queue from the client to the server
- Requests are pipelined one after another
  - This allows the server to process requests immediately one after another
  - This saves one request and response propagation latency
  - In our example, the total time will be 172ms
- Parallel processing of requests
  - In our example this saves another 20ms of latency
  - **Head of line blocking**
    - Slower response (css with processing time 20ms) must be buffered until the first response is generated and sent (no interleaving of responses)
- Issues
  - A single slow response blocks all requests behind it
  - Buffered (large or many) responses may exhaust server resources
  - A failed response may terminate TCP connection
    - A client must request all sub-sequent resources again (duplicate processing)
  - Some intermediaries may not support pipelining and abort connection
- HTTP pipelining support today is limited

## Multiple TCP connections

- Using only one TCP connection is slow
  - Client must queue HTTP requests and process one after another
- Multiple TCP connections work in parallel
- **There are 6 connections per host**
  - The client can dispatch up to 6 requests in parallel
  - The server can process up to 6 requests in parallel
  - This is a trade-off between higher request parallelism and the client and server overhead
- The maximum number of connections prevents from DoS attacks
  - The client could exhaust server resources
- Domain sharding
  - The connection limit as per host (origin)
  - There can be multiple origins used in a page
    - Each origin has 6 maximum connection limit
  - A domain can be sharded
    - [www.example.com](http://www.example.com) → [shard1.example.com](http://shard1.example.com), [shard2.example.com](http://shard2.example.com)
    - Each shard can resolve to the same IP or different IP, it does not matter
  - How many shards?

# Serving HTTP Request



- Serving HTTP request

1. User enters URL **http://shard1.example.com/orders** to the browser
2. DNS resolution: browser gets an IP address for **shard1.example.com**
3. Three-way handshake: browser and Web Server creates a socket
4. Browser sends ACK and HTTP request:

```
1 | GET /orders HTTP/1.1
2 | Host: shard1.example.com
```
5. Web server passes the request to the web application **shard1.example.com** which serves **GET orders** and that writes a response back to the socket.

# Virtual Host

- Virtual host
  - Configuration of a named virtual host in a Web server
  - Web server uses host request header to distinguish among multiple virtual hosts on a single physical host.
- Apache virtual host configuration
  - Two virtual hosts in a single Web server

```
1 | # all IP addresses will be used for named virtual hosts
2 | NameVirtualHost *:80
3 |
4 | <VirtualHost *:80>
5 |     ServerName www.example.com
6 |     ServerAlias shard1.example.com shard2.example.com
7 |     ServerAdmin admin@example.com
8 |     DocumentRoot /var/www/apache/example.com
9 | </VirtualHost>
10 |
11 | <VirtualHost *:80>
12 |     ServerName company.cz
13 |     ServerAdmin admin@firm.cz
14 |     DocumentRoot /var/www/apache/company.cz
15 | </VirtualHost>
```



## Better Support for HTTP Testing

- Use **curl** to test HTTP protocol

```
1 Usage: curl [options...] <url>
2
3 -X/--request <command>      Specify request command to use
4 -H/--header <line>          Custom header to pass to server
5 -d/--data <data>            HTTP POST data
6 -b/--cookie <name=string/file> Cookie string or file to read cookies from
7 -v/--verbose                Make the operation more talkative
```

- Example

```
1 curl -v -H "Host: company.cz" 127.0.0.1:8080
2
3 * About to connect() to 127.0.0.1 port 8080
4 * Trying 127.0.0.1... connected
5 * Connected to 127.0.0.1 port 8080
6 > GET / HTTP/1.1
7 > User-Agent: curl/7.20.0 (i386-apple-darwin10.3.2) libcurl/7.20.0 OpenSSL/0.9.8
8 > Accept: */*
9 > Host: company.cz
10 >
11 < HTTP/1.1 201 OK
12 < Connection: keep-alive
13 < Content-Type: plain/text
14 <
15 < This is the response...
```

## Overview

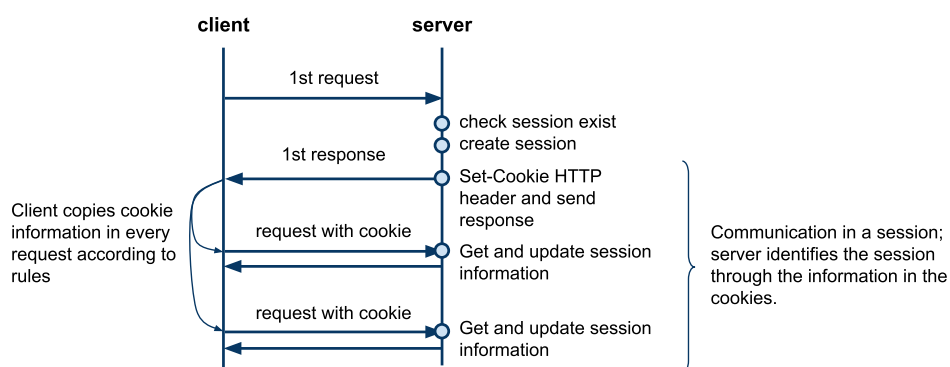
- Introduction to Application Protocols
- Introduction to HTTP
  - *State Management*
- Security

## State Management

- HTTP is a stateless protocol – original design
  - No information to relate multiple interactions at server-side
    - Except **Authorization** header is copied in every request
    - IP addresses do not work, one public IP can be shared by multiple clients
- Solutions to check for a valid state at server-side
  - **Cookies** – obvious and the most common workaround
    - RFC 2109 – HTTP State Management Mechanism
    - Allow clients and servers to talk in a context called **sessions**
  - **Hypertext** – original HTTP design principle
    - App states represented by resources (hypermedia), links define transitions between states
    - Adopted by the REST principle **statelessness**

## Interaction with Cookies

- Request-response interaction with cookies
  - Session is a logical channel maintained by the server



- Stateful Server
  - Server remembers the session information in a server memory
  - Server memory is a non-persistent storage, when server restarts the memory content is lost!

# Set-Cookie and Cookie Headers

- **Set-Cookie** response header

```
1 set-cookie = "Set-Cookie:" cookie ("," cookie)*
2 cookie     = NAME "=" VALUE ("," cookie-av)*
3 cookie-av  = "Comment" "=" value
4             "Domain"  "=" value
5             "Max-Age"  "=" value
6             "Path"    "=" value
```

- **domain** – a domain for which the cookie is applied
- **Max-Age** – number of seconds the cookie is valid
- **Path** – URL path for which the cookie is applied

- **Cookie** request header. A client sends the cookie in a request if:

- **domain** matches the origin server's fully-qualified host name
- **path** matches a prefix of the request-URI
- **Max-Age** has not expired

```
1 cookie = "Cookie:" cookie-value ("," cookie-value)*
2 cookie-value = NAME "=" VALUE ["," path] ["," domain]
3 path        = "$Path" "=" value
4 domain      = "$Domain" "=" value
```

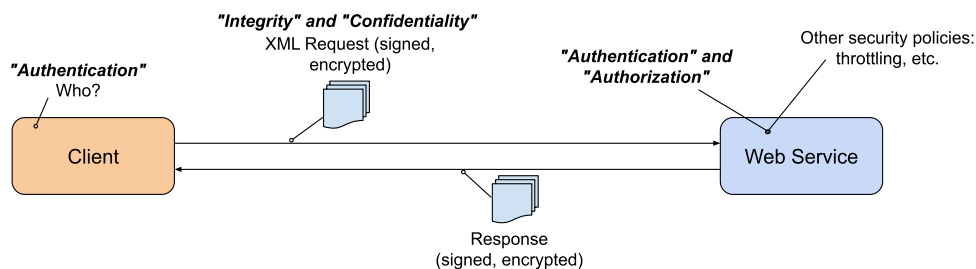
- **domain**, and **path** are values from corresponding attributes of the **Set-Cookie** header

## Overview

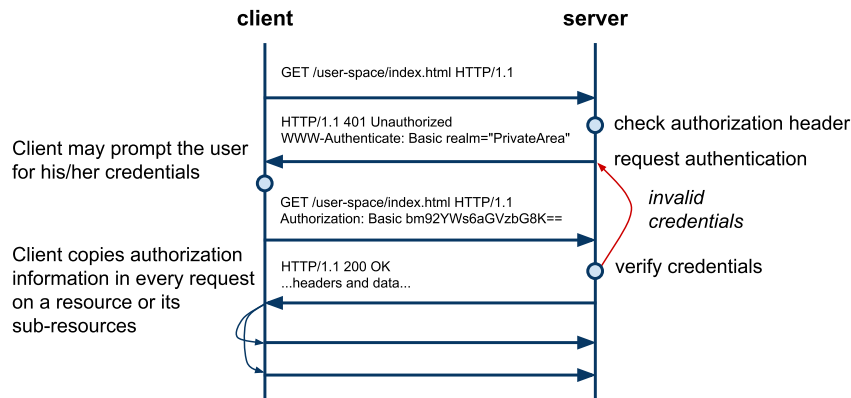
- Introduction to Application Protocols
- Introduction to HTTP
- **Security**

## Web Service Security Concepts

- Securing the client-server communication
  - Message-level security
  - Transport-level security
- Ensure
  - Authentication – verify a client's identity
  - Authorizaton – rights to access resources
  - Message Confidentiality – keep message content secret
  - Message Integrity – message content does not change during transmission
  - Non-repudiation – proof of integrity and origin of data



## Basic Access Authentication



- Realm
  - an identifier of the space on the server (~ a collection of resources and their sub-resources)
  - A client may associate a valid credentials with realms such that it copies authorization information in requests for which server requires authentication (by **WWW-Authenticate** header)

## Basic Access Authentication – Credentials

- Credentials

- *credentials are base64 encoded*
- *the format is: **username:password***

```
1 # to encode in linux
2 echo "novak:heslo" | base64
3 > bm92YWs6aGVzbG8K
4
5 # and to decode
6 echo "bm92YWs6aGVzbG8K" | base64 -d # use capital "D" in OS X
7 > novak:heslo
```

- Comments

- *When TLS is not used, the password can be read*
- *An attacker can repeat interactions*

## Digest Access Authentication

- RFC 2617 – Basic and Digest Access Authentication

- *No password between a client and a server but a hash value*
- *Simple and advanced mechanisms (only server-generated nonce value – replay-attacks or with client-generated nonce value)*

- Basic Steps

1. Client accesses a protected area

```
1 | > GET / HTTP/1.1
```

2. Server requests authentication with **WWW-Authenticate**

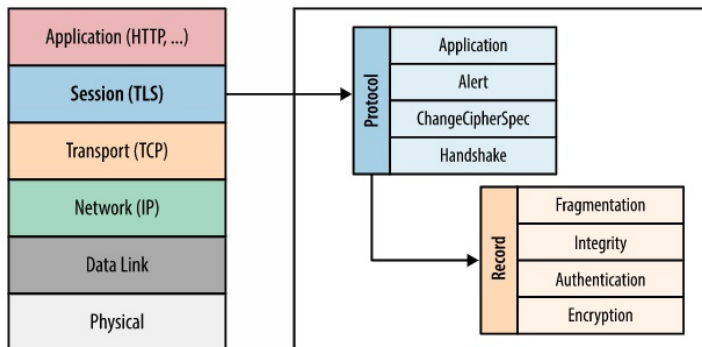
```
1 < HTTP/1.1 401 Unauthorized
2 < WWW-Authenticate: Digest realm="ProtectedArea",
3   nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045",
4   algorithm=MD5, domain="/", qop="auth"
```

3. Client calculates a response hash by using the realm, his/her username, the password, and the quality of protection (QoP) and requests the resource with **authorization** header

```
1 > GET / HTTP/1.1
2 > Authorization: Digest username="novak", realm="ProtectedArea",
3   nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045", uri="/",
4   algorithm=MD5, response="c4ea2293aeb318826d1e533f363efd90", qop=auth,
5   nc=00000001, cnonce="531ee8ba7f2a8fd1"
```

# Transport Level Security

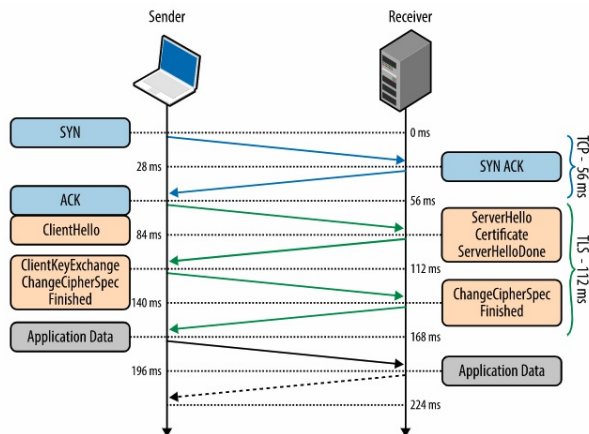
- SSL and TLS
  - *SSL and TLS is used interchangeably*
  - *SSL 3.0 developed by Netscape*
  - *IETF standardization of SSL 3.0 is TLS 1.0*
    - *TLS 1.0 is upgrade of SSL 3.0*
  - *Due to security flaws in TLS 1.0, TLS 1.1 and TLS 1.2 were created*
- TLS layer



# TLS Services

- Encryption
  - *Peers must agree on ciphersuite and keys*
  - *This is achieved by **TLS handshake***
- Authentication
  - *Peers can authenticate their identity*
    - *The client can verify that the server is who it is claimed to be*
    - *Achieved by "Chain of Trust and Certificate Authorities"*
    - *The server can also verify the client*
- Integrity
  - *TLS provides message framing mechanism*
  - *Every message is signed with Message Authentication Code (MAC)*
    - *MAC hashes data in a message and combines the resulting hash with a key (negotiated during the TLS handshake)*
    - *The result is a message authentication code sent with the message*

# TLS Handshake Protocol



- TLS Handshake

*56 ms: ClientHello, TLS protocol version, list of ciphersuites, TLS options*

*84 ms: ServerHello, TLS protocol version, ciphersuite, certificate*

*112 ms: RSA or Diffie-Hellman key exchange*

*140 ms: Message integrity checks, sends encrypted "Finished" message*

*168 ms: Decrypts the message, app data can be sent*

## Key Exchange

- RSA key exchange(Rivest–Shamir–Adleman)
  - The client generates a symmetric key
  - The client encrypts the key with the server's public key
  - The client sends the encrypted key to the server
  - The server uses its private key to decrypt the symmetric key
- RSA critical weakness
  - The same public-private key pair is used to:
    - authenticate the server (the server's private key is used to sign and verify the handshake)
    - encrypt the symmetric key
  - When an attacker gets hold of the server private key
    - It can decrypt the entire session
- Diffie-Hellman key exchange
  - Client and server can negotiate shared secret without its explicit communication
    - Attacker cannot get the key
  - Reduction of risk of compromising of the past communications
    - New key can be generated as part of every key exchange
    - Old keys can be discarded

# TLS and Proxy Servers

- TLS Offloading
  - *Inbound TLS connection, plain outbound connection*
  - *Proxy can inspect messages*
- TLS Bridging
  - *Inbound TLS connection, new outbound TLS connection*
  - *Proxy can inspect messages*
- End-to-End TLS (TLS pass-through)
  - *TLS connection is passed-through the proxy*
  - *Proxy cannot inspect messages*
- Load balancer
  - *Can use TLS offloading or TLS bridging*
  - *Can use TLS pass-through with help of Server Name Indication (SNI)*