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

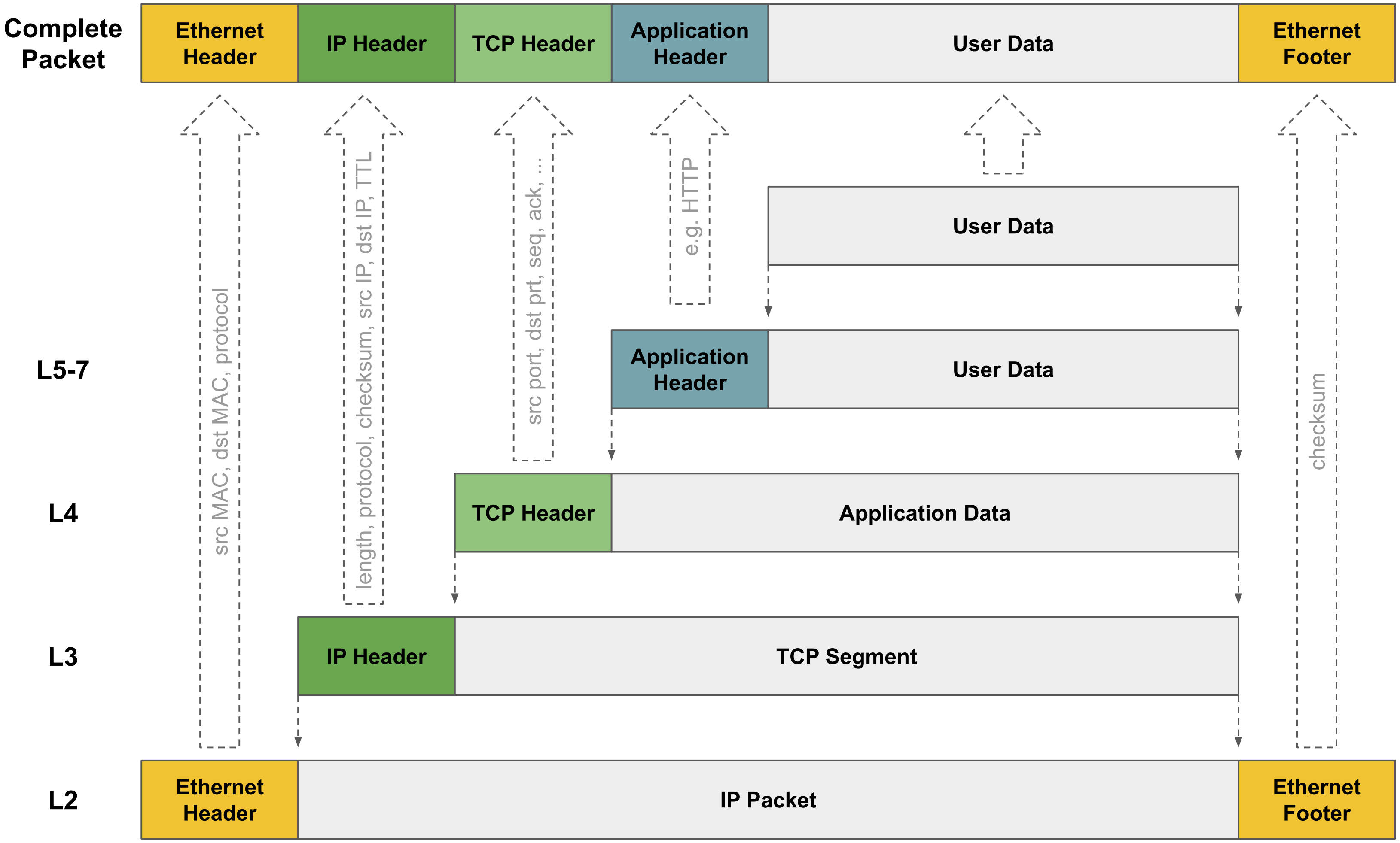
Application Protocols

- Remember this

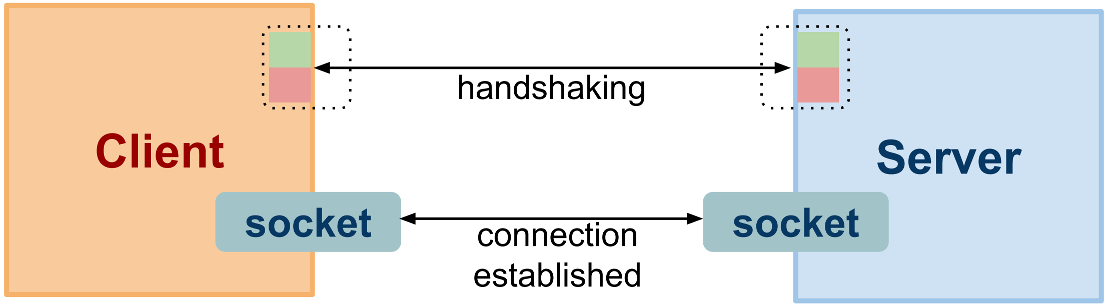
	All	People	Seem	To	Need	Data	Processing
OSI Model	Application	Presentation	Session	Transport	Network	Data Link	Physical
TCP/IP (services)	Application HTTP, XML-RPC, SOAP, RMI			Transport TCP	Network IP	Data Link	Physical

- 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

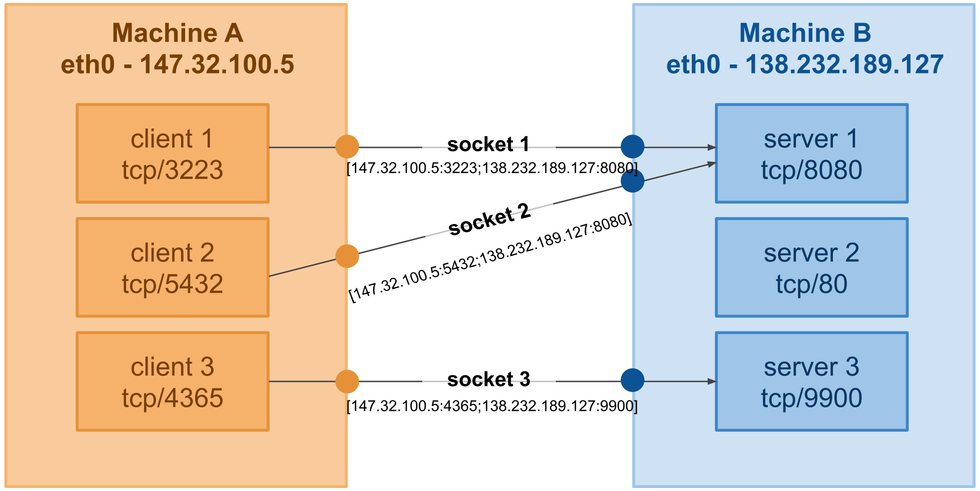


- Handshaking (connection establishment)
  - *The server listens at* `[dst_ip,dsp_port]`
  - *Three-way handshake:*
    - → *the client sends a connection request with TCP flags (SYN, x=rand)*
    - → *the server respons with its own TCP flags (SYN ACK, x+1 y=rand)*
    - → *the client acknowledges the 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 handskake 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 successfive 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 respectivelly*
- Without HTTP keepalive
  - *three-way handshake 84ms before the data is received on the server*
  - *Response received at 152ms and 132ms respectivelly*
  - *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) x 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 reuqests*
    - → *Images: 55 requests*
    - → *Javascript: 15 requests*
    - → *CSS: 5 requests*
    - → *Other: 5 requests*

HTTP pipelining

- Important optimization – response queuing
  - *Allows to relecote FIFO queue from the client to the server*
- Requests are pipelined one after another
  - *This allows the server to process requests immediatelly 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 (dupplicate processing)*
  - *Some intermediaries may not support pipelining and abort connection*
- HTTP pipelining support today is limited

Multiple TCP connections

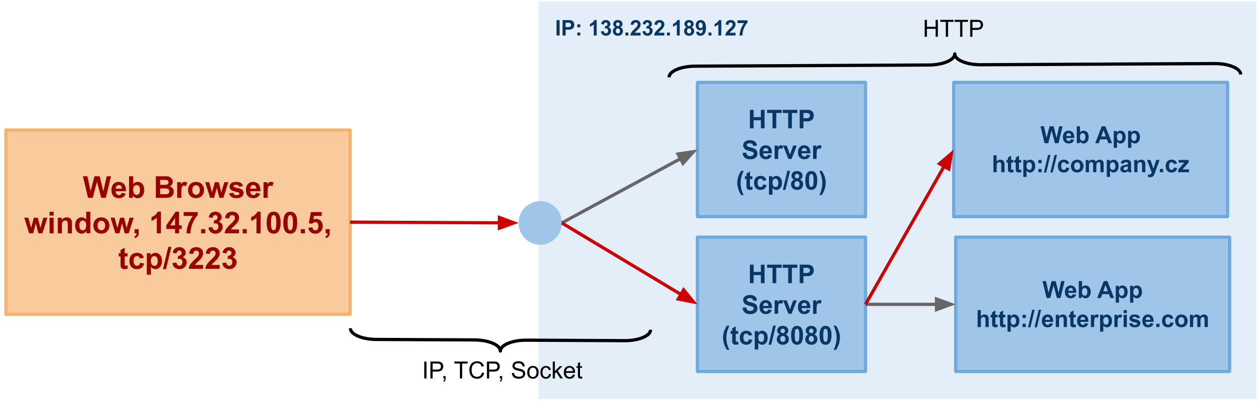
- Using only one TCP connection is slow
  - *Client must queue HTTP requests and process one after another*
- Multiple TCP connections work in parallel
- **There are 6 connections per host**
  - *The client can dispatch up to 6 requests in parallel*
  - *The server can process up to 6 requests in parallel*
  - *This is a trade-off between higher request paralellism 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` → `shard1.example.com`, `shard2.example.com`

    → *Each shard can resolve to the same IP or different IP, it does not matter*
  - *How many shards?*

Serving HTTP Request


IP: 138.232.189.127
HTTP
Web Browser window, 147.32.100.5, tcp/3223
HTTP Server (tcp/80)
HTTP Server (tcp/8080)
Web App http://company.cz
Web App http://enterprise.com
IP, TCP, Socket

- 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:*

     ```
     GET /orders HTTP/1.1
     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*

```
# all IP addresses will be used for named virtual hosts
NameVirtualHost *:80

<VirtualHost *:80>
        ServerName www.example.com
        ServerAlias shard1.example.com shard2.example.com
        ServerAdmin admin@example.com
        DocumentRoot /var/www/apache/example.com
</VirtualHost>

<VirtualHost *:80>
        ServerName company.cz
        ServerAdmin admin@firm.cz
        DocumentRoot /var/www/apache/company.cz
</VirtualHost>
```

Better Support for HTTP Testing

- Use `curl` to test HTTP protocol

```
Usage: curl [options...] <url>

-X/--request <command>          Specify request command to use
-H/--header <line>              Custom header to pass to server
-d/--data <data>                HTTP POST data
-b/--cookie <name=string/file>  Cookie string or file to read cookies from
-v/--verbose                    Make the operation more talkative
```

- Example

```
curl -v -H "Host: company.cz"  127.0.0.1:8080

* About to connect() to 127.0.0.1 port 8080
*   Trying 127.0.0.1... connected
* Connected to 127.0.0.1 port 8080
> GET / HTTP/1.1
> User-Agent: curl/7.20.0 (i386-apple-darwin10.3.2) libcurl/7.20.0 OpenSSL/0.9.8n
> Accept: */*
> Host: company.cz
>
< HTTP/1.1 201 OK
< Connection: keep-alive
< Content-Type: plain/text
<
< This is the response...
```

Overview

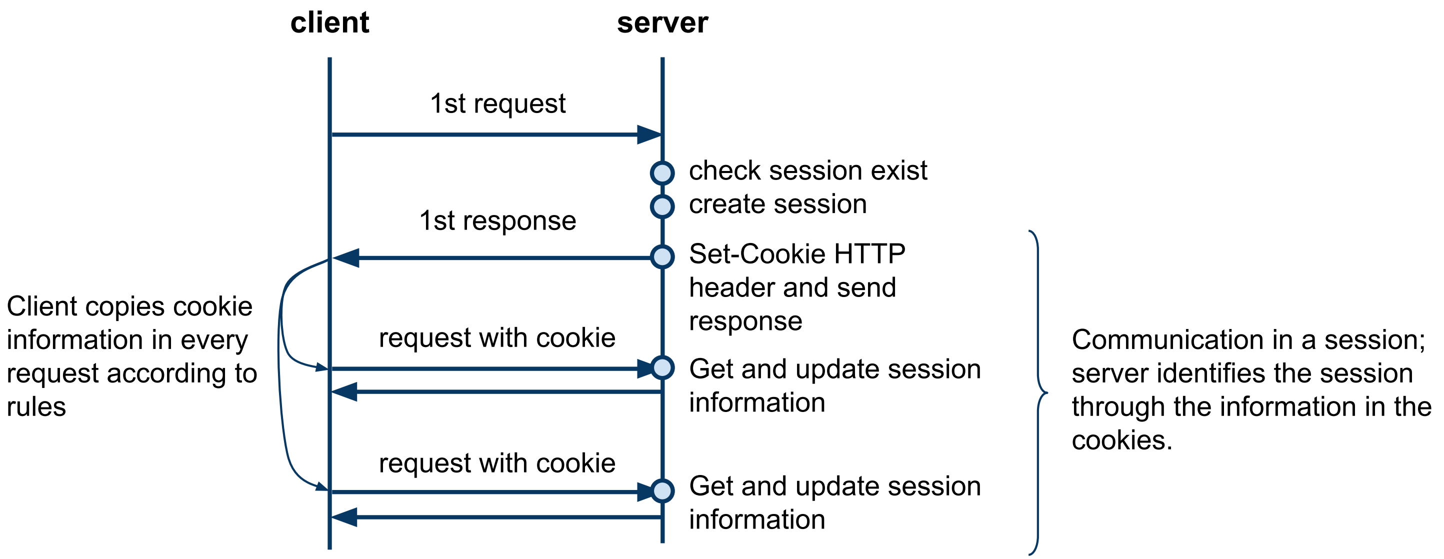
- Introduction to Application Protocols
- Introduction to HTTP
  - *State Management*
- Security

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*


client
server
1st request
check session exist
create session
1st response
Set-Cookie HTTP header and send response
Client copies cookie information in every request according to rules
request with cookie
Get and update session information
request with cookie
Get and update session information
Communication in a session; server identifies the session through the information in the cookies.

- 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

```
set-cookie = "Set-Cookie:" cookie ("," cookie)*
  cookie     = NAME "=" VALUE (";" cookie-av)*
  cookie-av = "Comment" "=" value
            | "Domain" "=" value
            | "Max-Age" "=" value
            | "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*

```
cookie  = "Cookie:" cookie-value (";" cookie-value)*
  cookie-value    = NAME "=" VALUE [";" path] [";" domain]
  path            = "$Path" "=" value
  domain          = "$Domain" "=" value
```

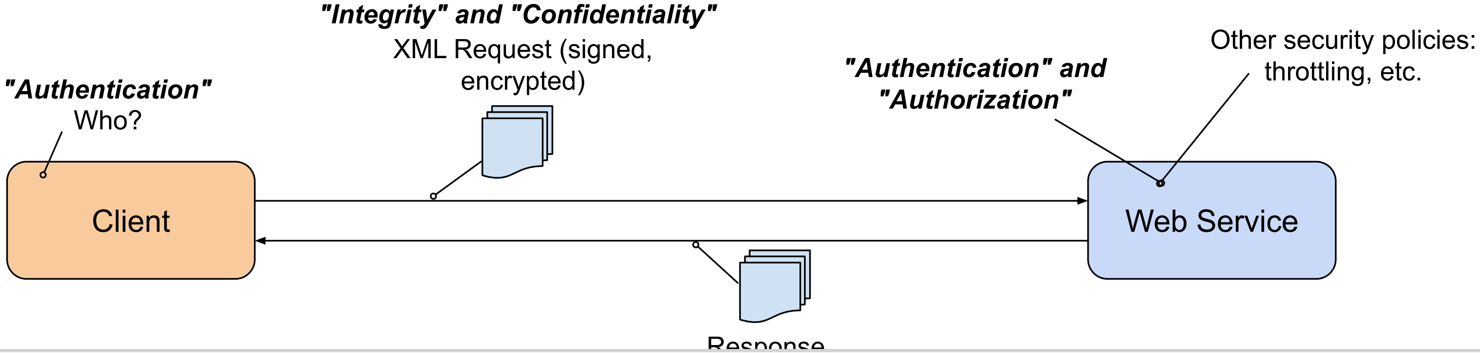
  - `domain`, *and* `path` *are values from corresponding attributes of the* `Set-Cookie` *header*

Overview

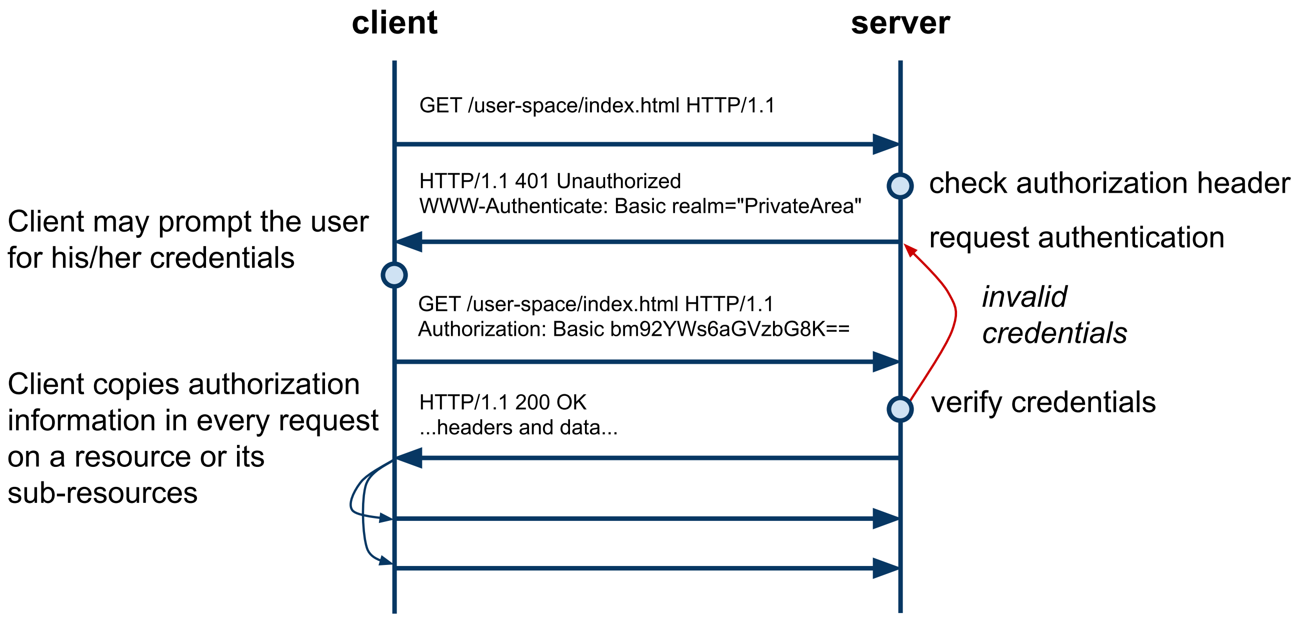
- Introduction to Application Protocols
- Introduction to HTTP
- Security

Web Service Security Concepts

- Securing the client-server communcation
  - *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*


"Integrity" and "Confidentiality"
XML Request (signed, encrypted)
"Authentication"
Who?
"Authentication" and "Authorization"
Other security policies: throttling, etc.
Client
Web Service
Response

Basic Access Authentication


client
server
GET /user-space/index.html HTTP/1.1
HTTP/1.1 401 Unauthorized
WWW-Authenticate: Basic realm="PrivateArea"
check authorization header
request authentication
Client may prompt the user for his/her credentials
GET /user-space/index.html HTTP/1.1
Authorization: Basic bm92YWs6aGVzbG8K==
invalid credentials
Client copies authorization information in every request on a resource or its sub-resources
HTTP/1.1 200 OK
...headers and data...
verify credentials

- 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`

```
# to encode in linux
echo "novak:heslo" | base64
> bm92YWs6aGVzbG8K

# and to decode
echo "bm92YWs6aGVzbG8K" | base64 -d # use capital "D" in OS X
> 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*

```
> GET / HTTP/1.1
```

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

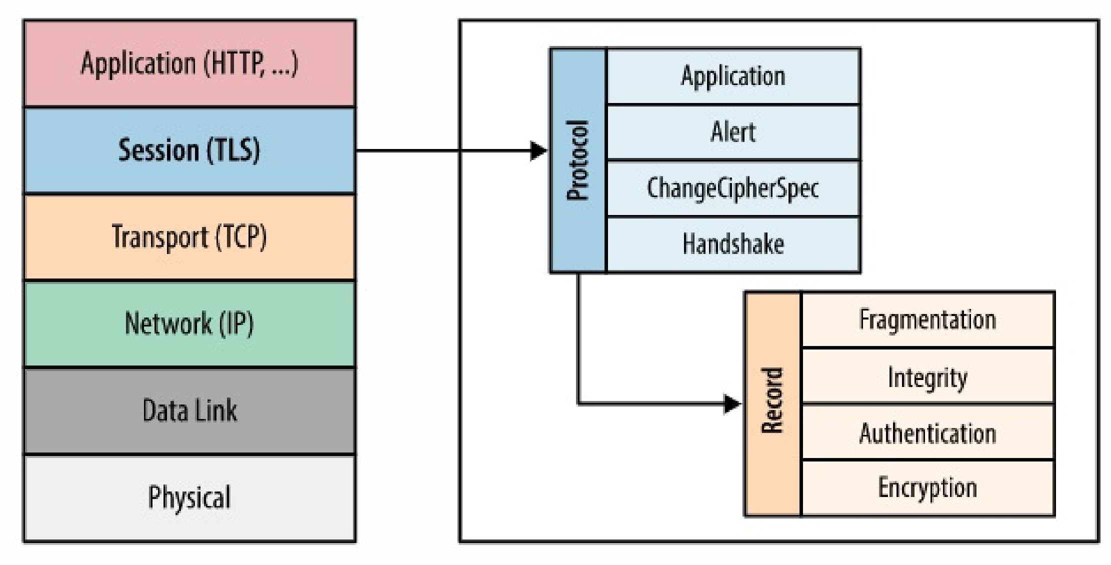
```
< HTTP/1.1 401 Unauthorized
< WWW-Authenticate: Digest realm="ProtectedArea",
  nonce="BbdQof3DBAA=a293ff3d724989371610f03015f2d23f3cd2c045",
  algorithm=MD5, domain="/", qop="auth"
```

  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*

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

Transport Level Security

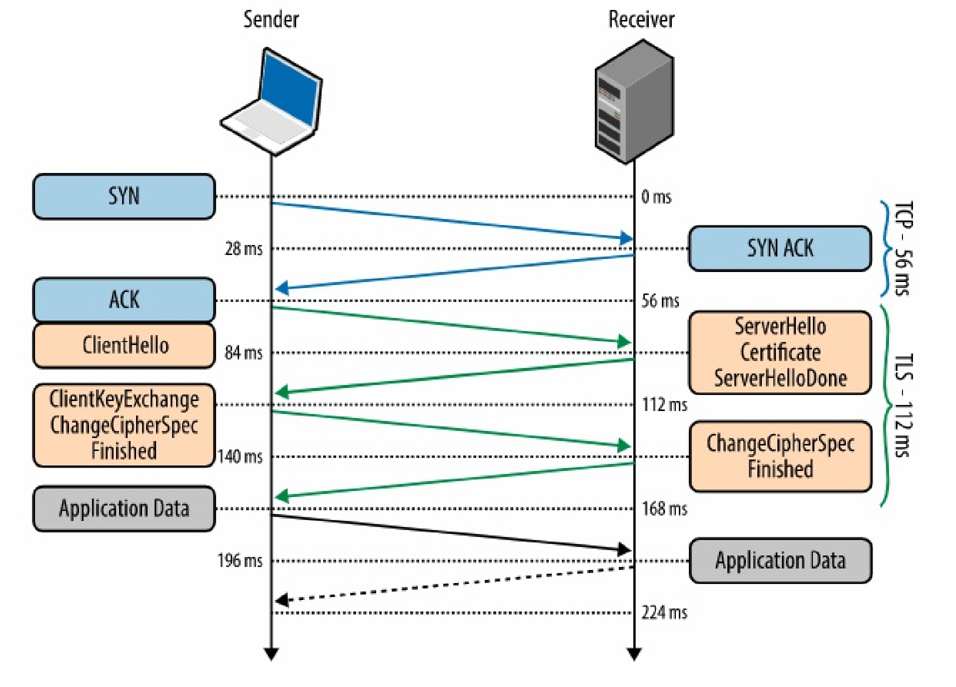
- SSL and TLS
  - *SSL and TLS is used interchangeably*
  - *SSL 3.0 developed by Netscape*
  - *IETF standardization of SSL 3.0 is TLS 1.0*
    → *TLS 1.0 is upgrade of SSL 3.0*
  - *Due to security flaws in TLS 1.0, TLS 1.1 and TLS 1.2 were created*
- TLS layer



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 Authoritites"*
    - → *The server can also verify the client*
- Integrity
  - *TLS provives 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 exhange(Rivest–Shamir–Adleman)
  - *The client generates a symetric key*
  - *The client encrypts the key with the server's public key*
  - *The client sends the encrpyed key to the server*
  - *The server uses its private key to decrypt the symetric key*
- RSA critical weekness
  - *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 symetric key*
  - *When an attacker gets hold of the server private key*
    - → *It can decrypt the entire session*
- Diffie-Hellman key exhange
  - *Client and server can negoriate 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)*