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

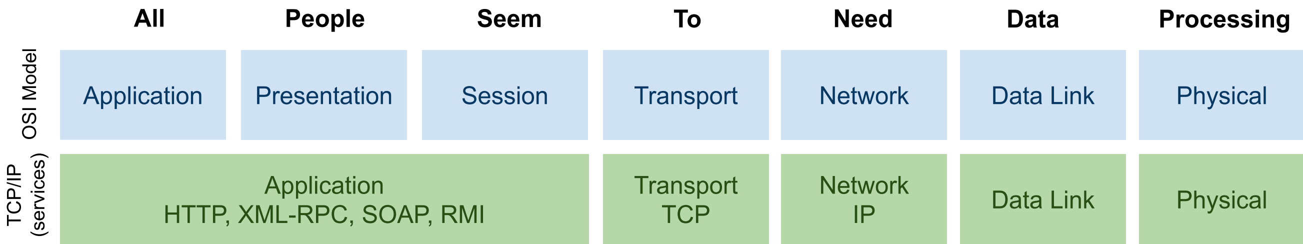
Modified: Mon Nov 01 2021, 07:23:09
Humla v1.0

Overview

- Introduction to Application Protocols
  – *Synchronous and Asynchronous Communication*
- Introduction to HTTP
- SOAP and WSDL

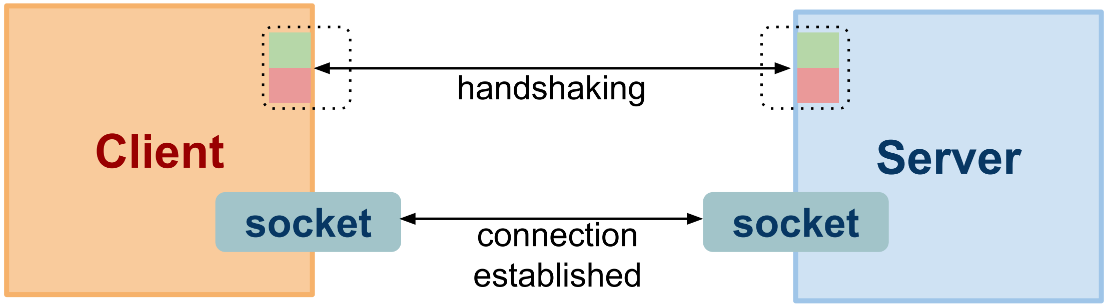
Application Protocols

- 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*

Socket

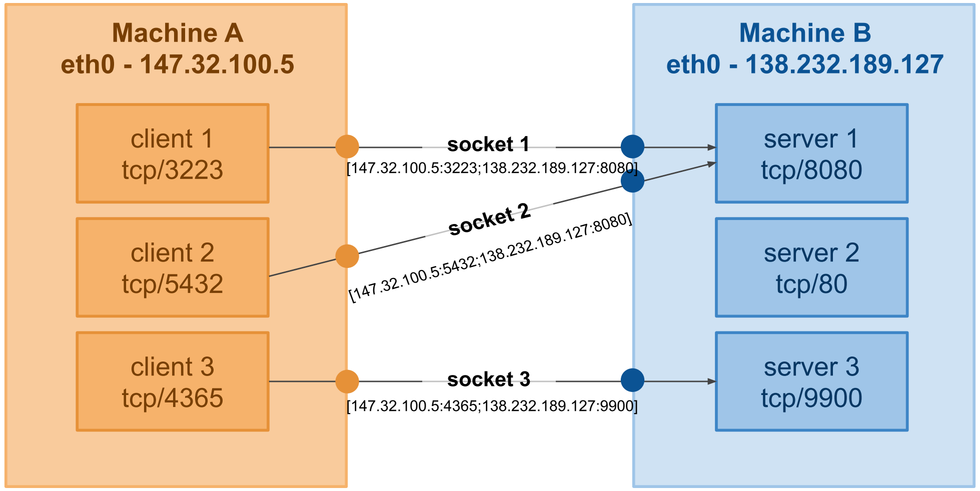


- Handshaking (connection establishment)
  - *The server listens at* `[dst_ip,dsp_port]`
  - *Three-way handshake:*
    - → *the client sends a connection request with TCP flags (SYN, x=rand)*
    - → *the server respons with its own TCP flags (SYN ACK, x+1 y=rand)*
    - → *the client acknowledges the 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

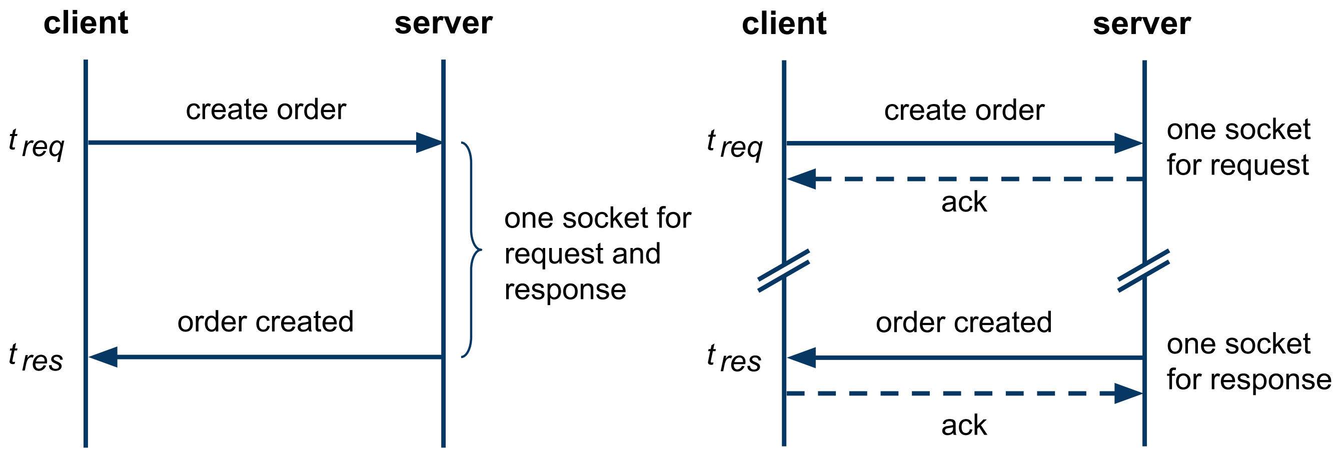


- IP addressing: IP is an address of a machine interface
  - *A machine can have multiple interfaces (eth0, eth1, bond0, ...)*
- TCP addressing: TCP port is an address of an app running on a machine and listening on a machine interface
  - *Multiple applications with different TCP ports may listen on 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

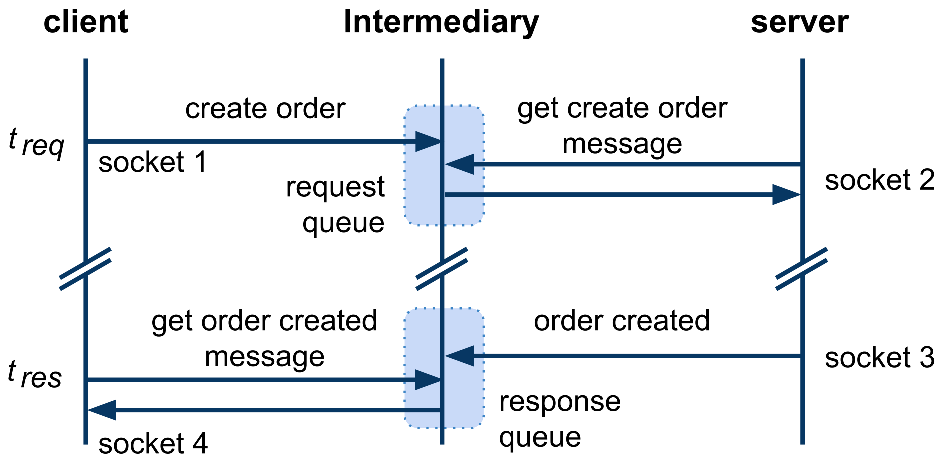
- Introduction to Application Protocols
  – *Synchronous and Asynchronous Communication*
- Introduction to HTTP
- SOAP and WSDL

Synchronous and Asynchronous Communication



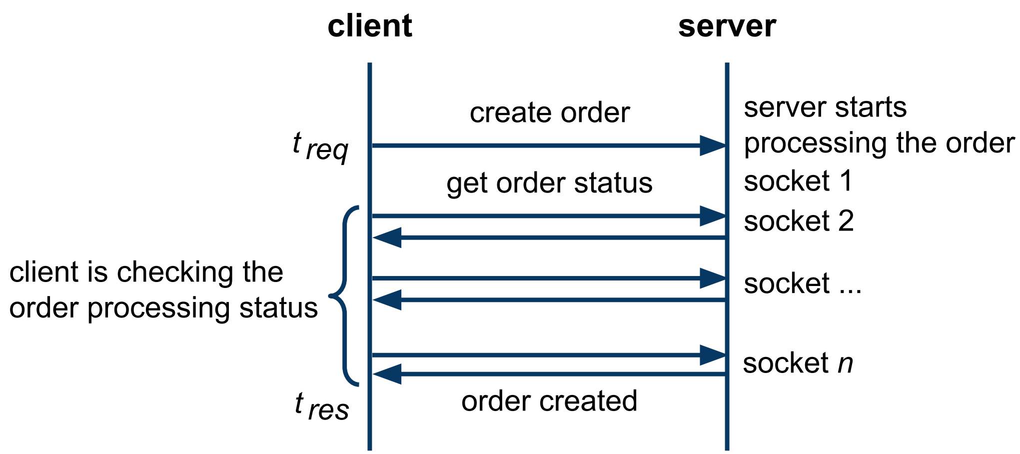
- Synchronous
  - *one socket, $|t_{req} - t_{res}|$ is small*
  - *easy to implement and deploy, only standard firewall config*
  - *only the server defines endpoint*
- Asynchronous
  - *request, response each has socket, client and server define endpoints*
  - *$|t_{req} - t_{res}|$ can be large (hours, even days)*
  - *harder to do across network elements (private/public networks issue)*

Asynchronous via Intermediary



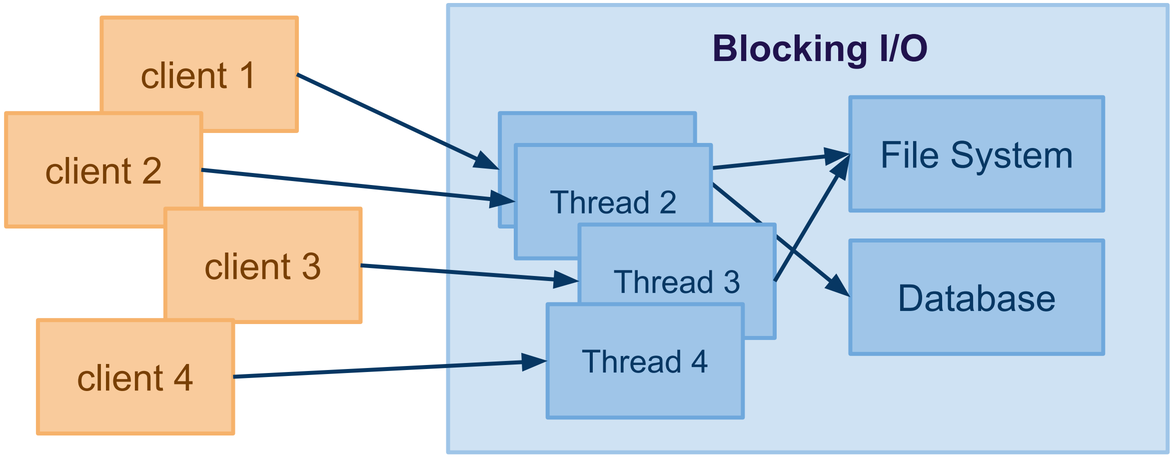
- Intermediary
  - *A component that decouples a client-server communication*
  - *It increases reliability and performance*
    - → *The server may not be available when a client sends a request*
    - → *There can be multiple servers that can handle the request*
- Further Concepts
  - *Message Queues (MQ) – queue-based communication*
  - *Publish/Subscribe (P/S) – event-driven communication*

Asynchronous via Polling



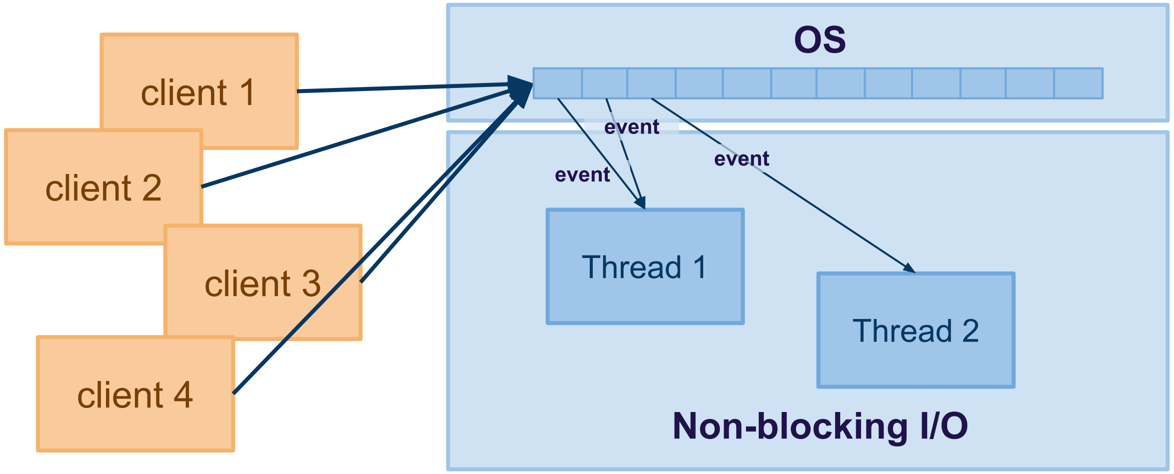
- Polling – only clients open sockets
  – *A client performs multiple request-response interactions*
    → *The first interaction initiates a process on the server*
    → *Subsequent interactions check for the processing status*
    → *The last interaction retrieves the processing result*
- Properties of environments
  – *A server cannot open a socket with the client (network restrictions)*
  – *Typically on the Web (a client runs in a browser)*

Blocking (Synchronous) I/O



- Inbound connection
  - *A server creates a thread for every inbound connection*
  - *For example, 1K connections = 1K threads, big overhead*
  - *A thread is reserved for the entire duration of the request processing*
- Outbound connection
  - *A thread is blocked when outbound connection is made*
  - *When outbound connection is slow, the scalability is poor*

Non-Blocking (Asynchrnous) I/O



- Inbound connections
  - *The connection is maintained by the OS, not the server app*
  - *The Web app registers events, OS triggers events when they occur*
  - *The app may create working threads and controls their number*
- Outound connections
  - *The app registers a callback that is called when the data is available*
  - *Event loop*

Overview

- Introduction to Application Protocols
- Introduction to HTTP
  – *State Management*
- SOAP and WSDL

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*

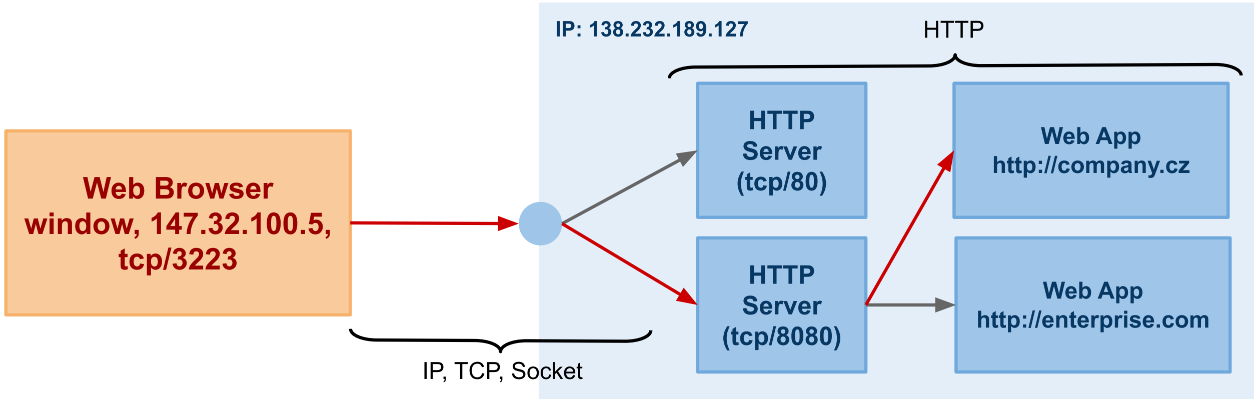
HTTP pipelining

- 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*
    - $\longrightarrow$ *Each origin has 6 maximum connection limit*
  - *A domain can be sharded*
    - $\longrightarrow$ `www.example.com` $\longrightarrow$ `shard1.example.com`, `shard2.example.com`
    - $\longrightarrow$ *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:*

     ```
     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

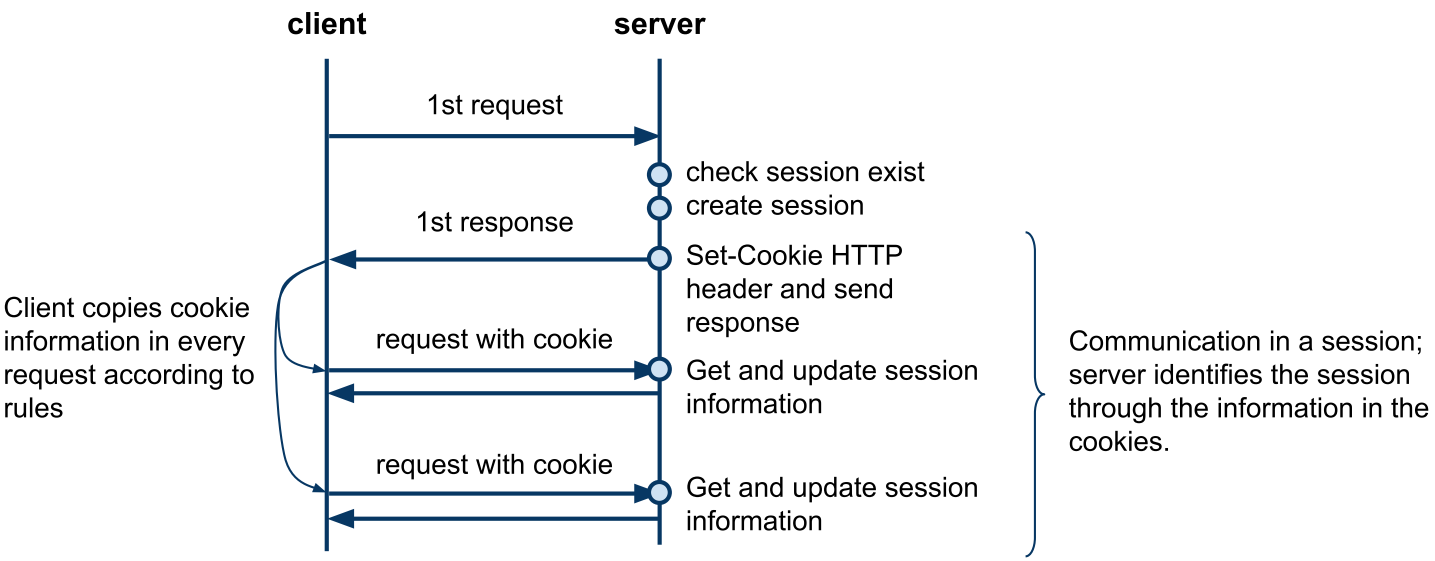
- Introduction to Application Protocols
- Introduction to HTTP
  - *State Management*
- SOAP and WSDL

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

```
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
```

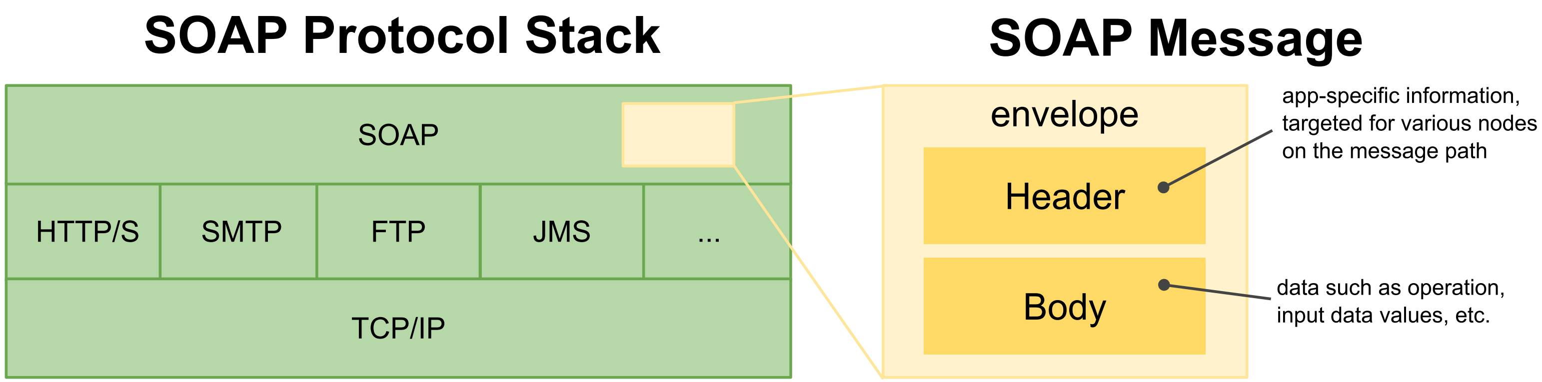
  - `domain`, *and* `path` *are values from corresponding attributes of the* `Set-Cookie` *header*

Overview

- Introduction to Application Protocols
- Introduction to HTTP
- SOAP and WSDL

SOAP Protocol

- SOAP defines a messaging framework



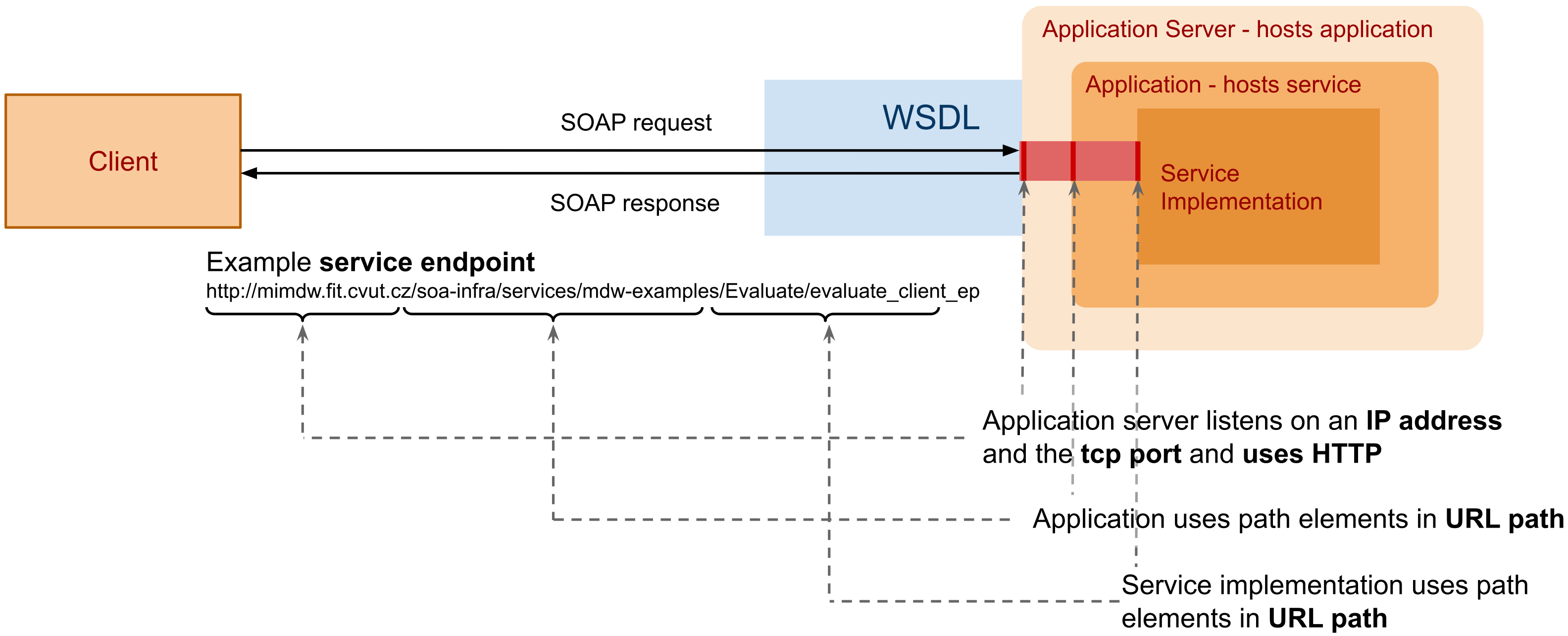
  – *XML-based protocol*
  – *a layer over transport protocols*
    → binding to HTTP, SMTP, JMS, ...
  – *involves multiple nodes (message path)*
    → sender, receiver, intermediary

SOAP Message

- Envelope
  – *A container of a message*
- Header
  – *Metadata – describe a message, organized in header blocks*
    → *routing information*
    → *security measures implemented in the message*
    → *reliability rules related to delivery of the message*
    → *context and transaction management*
    → *correlation information (request and response message relation)*
  – *WS extensions (WS-*) utilize the message header*
- Body (payload)
  – *Actual contents of the message, XML formatted*
  – *Contains also faults for exception handling*
- Attachment
  – *Data that cannot be serialized into XML such as binary data*

Endpoint

- SOAP service endpoint definition



  - *Endpoint – a network address used for communication*
  - *Communication – request-response, SOAP messages over a communication (application) protocol*
  - *Synchronous communication – only service defines endpoint*
  - *Asynchronous communication – service and client define endpoints*

WSDL Overview and WSDL 1.1 Syntax

- Components of WSDL
  - *Information model (`types`)*
    → *Element types, message declarations (XML Schema)*
  - *Set of operations (`portType`)*
    → *A set of operations is "interface" in the WSDL terminology*
    → *operation name, input, output, fault*
  - *Binding (`binding`)*
    → *How messages are transfered over the network using a concrete transport protocol*
    → *Transport protocols: HTTP, SMTP, FTP, JMS, ...*
  - *Endpoint (`service`)*
    → *Where the service is physically present on the network*
- Types of WSDL documents
  - ***Abstract WSDL** – only information model and a set of operations*
  - ***Concrete WSDL** – everything, a concrete service available in the environment*

WSDL Components and Dependencies