Introduction to Communication Networks and Distributed Systems

Unit 6 – Web Services -

Fachgebiet Telekommunikationsnetze
Prof. Dr. A. Zubow
zubow@tkn.tu-berlin.de

Acknowledgements

- We acknowledge the use of slides from: **Prof. Adam Wolisz**

- Moreover, some slides are based on the lectures from: Prof. Holger Karl, Paderborn; Prof. Ion Stoica, Berkeley, Prof. Ashay Parekh; Berkeley; Prof Lauer WPI; Prof. Baker, ACET, as well as slides form books by Tanenbaum, Kurose and Ross, Colouris at al.

Web Services

Web Services & Web Technology

- We already discussed the client-server approach as one of the core building blocks of today's Internet
- We also discussed the web as one of the core **services** using the Internet
- So far, we hove not discussed:
  - Scalability (think of Amazon, Google, Ebay, etc.)
  - Other services offered on top of web technologies
  - Recent trends in web technology
- **Today's topics**:
  - Scalable web architectures
  - Resource- and service-oriented web services
  - HTML5, WebRTC

What is a Web Service?

- A web service is a **collection of functions**
  - Packaged as a single entity and
  - Published to the network for use by other programs
- Web services
  - Building blocks for creating open distributed systems,
  - Allow companies and individuals to quickly and cheaply make their digital assets available worldwide
- A web service can **aggregate** other web services to provide a higher-level set of features
- Several popular sites provide Web services
  - Yahoo, Google, eBay, Amazon, …

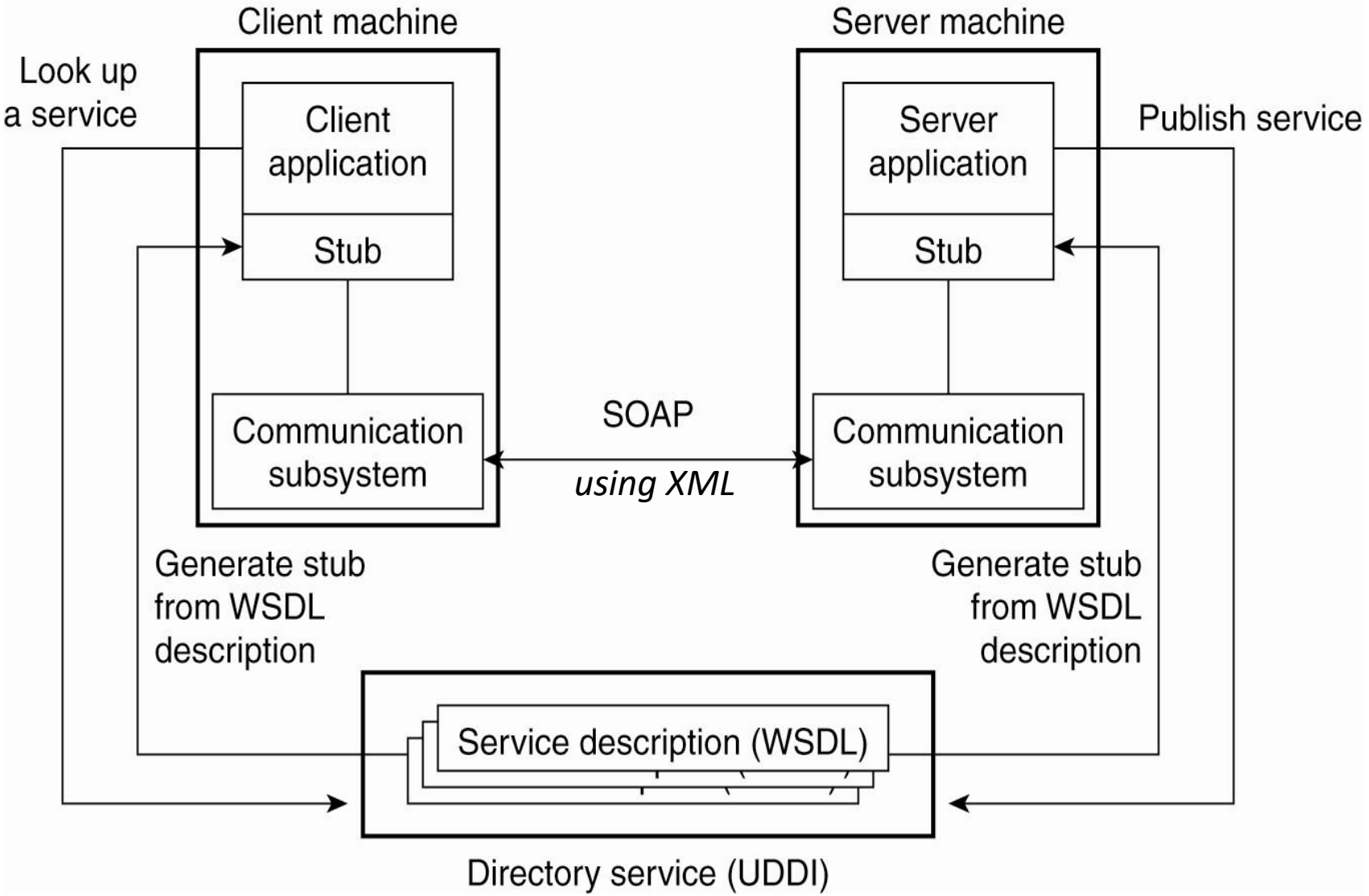
W3C Web Services

- A web service is a **software system** designed to support interoperable machine-to-machine interaction over a network. It has an **interface** described in a machine-processable format (specifically **WSDL**). Other systems interact with the web service in a manner prescribed by its description using **SOAP-messages**, typically conveyed using **HTTP** with an XML serialization in conjunction with other web-related standards.

W3C, Web Services Glossary

Web Services Architecture

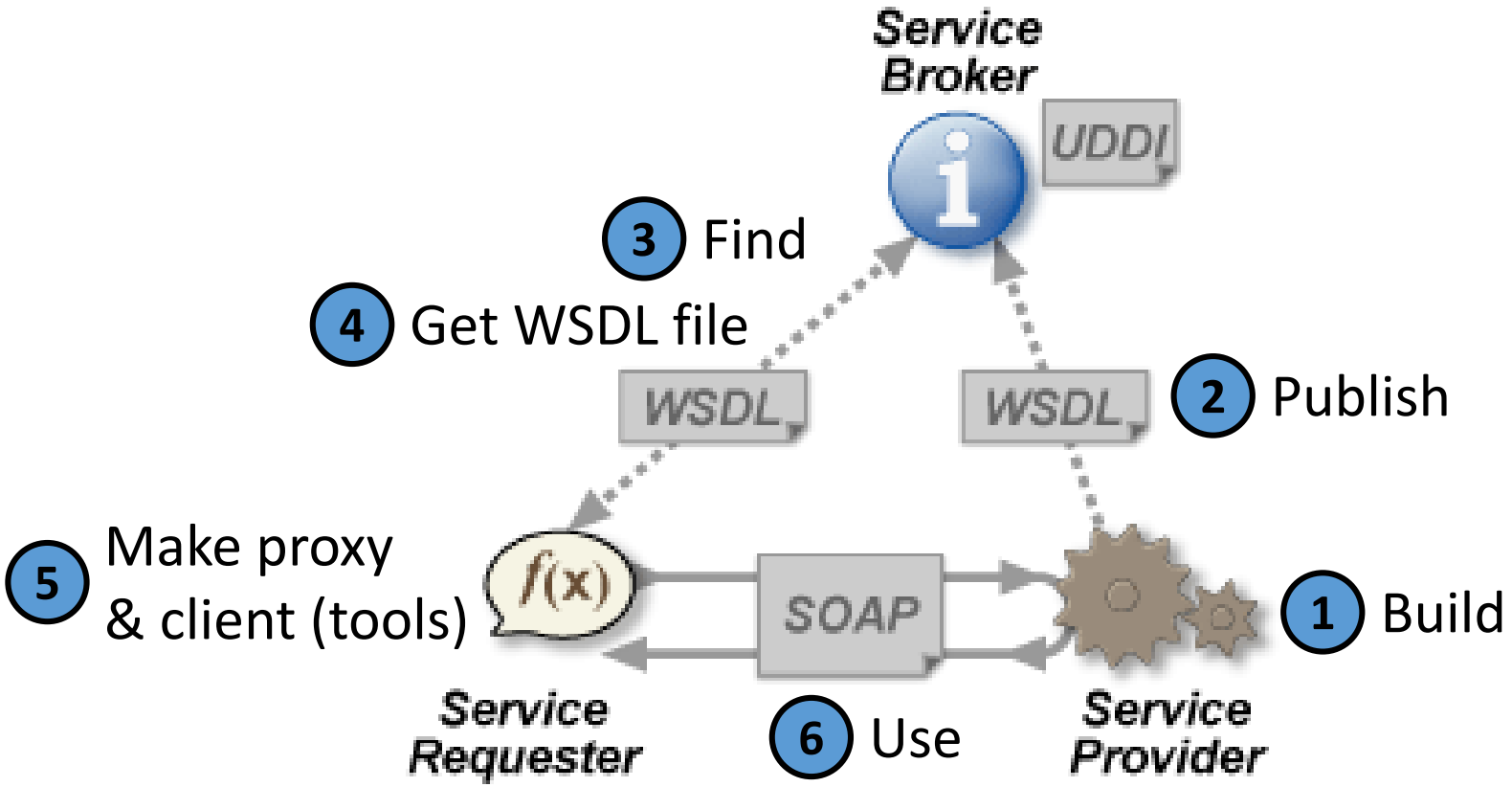
[Tanenbaum, op. cit.]



Basic building Blocks of Web Services

- **UDDI** (Universal Description, Discovery and Integration)
  - Services have to be discovered - http://uddi.xml.org/
- **WSDL** (Web Services Description Language)
  - Interfaces have to be described - http://www.w3.org/TR/wsdl
- **SOAP** (Simple Object Access Protocol)
  - (remote) objects access - http://www.w3.org/TR/soap/
- **XML** (Extensible Markup Language)
  - Data description format - http://www.w3.org/XML/
- **HTTP** (Hyper Text Transfer Protocol)
  - Communication layer - http://www.w3.org/Protocols/

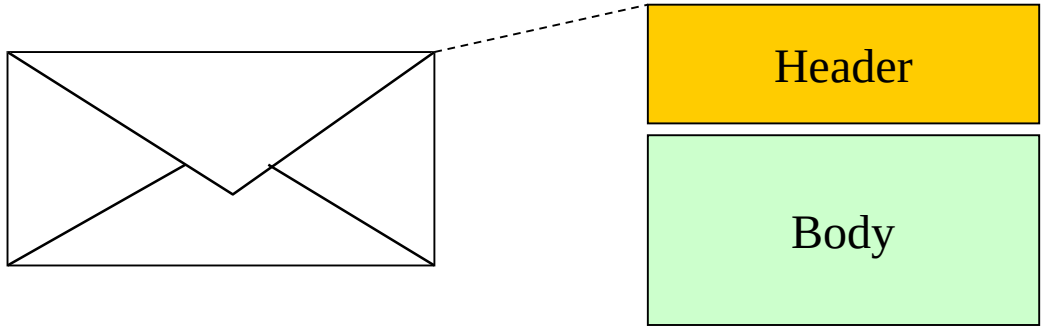
Web Service Interaction



Simple Object Access Protocol (SOAP)

- High-level communication protocol
  - Mostly: request/reply semantics (RPC style), also documents (message passing) …
  - SOAP defines message formats, not the protocol as such
  - Relies on the HTTP message for actually delivery
- Details
  - Using a common representation of data (XML)
  - Use a generally available transport protocol: mostly HTTP
    - E.g., to traverse firewalls
    - Implementations using other protocols (e.g. SMTP) exist!
    - Comment: W3C defines the use of SOAP with XML as payload and HTTP as transport

SOAP Elements



- **Envelope** (mandatory)
  - Top element of the XML document representing the message
- **Header** (optional)
  - Determines how a recipient of a SOAP message should process the message
  - Adds features to the SOAP message such as authentication, transaction management, payment, message routes, etc…
- **Body** (mandatory)
  - Exchanges information intended for the recipient of the message
  - Typical use is for RPC calls and error reporting

SOAP Example

- Usage of SOAP to query a database
  - Search for books with „SOAP“ in title
- Request sent by client:

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
        <s:Body>
                <m:TitleInDatabase xmlns:m="http://www.lecture-db.de/soap">
                        SOAP
                </m:TitleInDatabase>
        </s:Body>
</s:Envelope>
```

SOAP Example (II)

- Response sent by server:

```
<?xml version="1.0"?>
<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope">
        <s:Header>
                <m:RequestID xmlns:m="http://www.lecture-db.de/soap">a3f5c109b</m:RequestID>
        </s:Header>
        <s:Body>
                <m:DbResponse xmlns:m="http://www.lecture-db.de/soap">
                        <m:title value="SOAP">
                                <m:Choice value="1">Arbeitsbericht Informatik</m:Choice>
                                <m:Choice value="2">Seminar XML und Datenbanken</m:Choice>
                        </m:title>
                </m:DbResponse>
        </s:Body>
</s:Envelope>
```

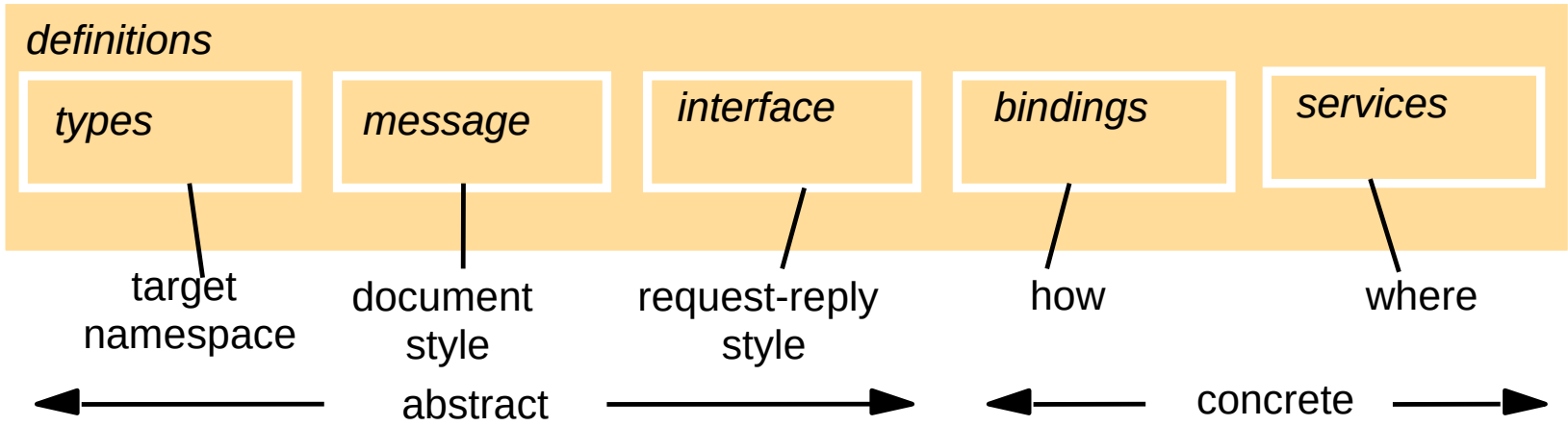
Web Service Description Language (WSDL)

- Main elements of WSDL description
  - Abstract: which compound types are used, combined into which messages
  - Concrete: **How** and **where** is the **service** to be contacted?
- Interface specification for web services
  - Written in XML to be programming-language-agnostic
  - Also includes how and where (URI) a service can be invoked
  - Define which kinds of messages can be sent between different entities (which data types are included in which message)

WSDL Definitions (1)

[Karl, op. cit.]

- **Types**: First define which data types are going to be exchanged between participants
  - Use existing XML-based type system
- **Message**: Define which kinds of messages can be sent between different entities
  - Which data types are included in which message
  - These are abstract messages, no reference to how these messages are represented on the wire



WSDL Definitions (2)

[Karl, op. cit.]

- **Port types**: A set of supported operations form the type of a port
  - Operation: An operation is a specification which abstract message type is sent and which one is received
  - Four kinds of operations exist:
    - One-way: entity only receives a message
    - Request/response: entity receives a messages and answers with a message
    - Solicit/response: entity sends a message and receives an answer
    - Notification: entity only sends a message
- **Binding**: As a port type is still an abstract concept, a mapping to a single, real protocol has to be specified
  - Message format, protocol details
  - Typical bindings: SOAP, HTTP GET/POST
  - Bindings must not include address information

WSDL Definitions (3)

[Karl, op. cit.]

- **Service**: Ports can be grouped into services
  - *Port*: A real port is then a binding with an address
    - Hence: an address where a number of operations can be invoked, along with the protocol information how to do so
  - Ports within a service do not communicate with each other
  - Service can contain several ports with the same port type, but different bindings -> alternative ways to access the same functionality using different protocols

Usage of WSDL

- Client
  - reads WSDL to find out the functionality provided by Web Service → all used data types are described in WSDL file
  - creates stub from WSDL file
  - uses SOAP to call a particular function listed in WSDL file

WSDL Example – Stock Quote Service

```
<definitions name="StockQuote" targetNamespace="http://example.com/stockquote.wsdl"
xmlns:tns="http://example.com/stockquote.wsdl"
xmlns:xsd1="http://example.com/stockquote.xsd"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/">
<types>
        <schema targetNamespace="http://example.com/stockquote.xsd"
xmlns="http://www.w3.org/2001/XMLSchema">
        <element name="TradePriceRequest">
                <complexType>
                        <all> <element name="tickerSymbol" type="string"/> </all>
                </complexType>
        </element>
        <element name="TradePrice">
                <complexType>
                        <all> <element name="price" type="float"/> </all>
                </complexType>
        </element>
        </schema>
</types>
```

WSDL Example – Stock Quote Service (II)

```
<message name="GetLastTradePriceInput">
       <part name="body" element="xsd1:TradePriceRequest"/>
</message>
<message name="GetLastTradePriceOutput">
       <part name="body" element="xsd1:TradePrice"/>
</message>

<portType name="StockQuotePortType">
       <operation name="GetLastTradePrice">
              <input message="tns:GetLastTradePriceInput"/>
              <output message="tns:GetLastTradePriceOutput"/>
       </operation>
</portType>
```

WSDL Example – Stock Quote Service (III)

```
<binding name="StockQuoteSoapBinding" type="tns:StockQuotePortType">
        <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetLastTradePrice">
                <soap:operation soapAction="http://example.com/GetLastTradePrice"/>
                <input>
                        <soap:body use="literal"/>
                </input>
                <output>
                        <soap:body use="literal"/>
                </output>
        </operation>
</binding>
<service name="StockQuoteService">
        <documentation>My first service</documentation>
        <port name="StockQuotePort" binding="tns:StockQuoteSoapBinding">
                <soap:address location="http://example.com/stockquote"/>
        </port>
</service>
</definitions>
```

UDDI

- UDDI is used to register and look up services with a central registry
  - Service Providers advertise their business services
  - Service consumers can look up UDDI-entries
  - UDDI parts:
    - **White** pages: Business information (Name, contact, description,...)
    - **Yellow** pages: Service information
    - **Green** pages: Technical information (Access point, WSDL reference)
- UDDI registry:
  - Distributed system(!) of individual UDDI servers
  - XML-based; Stores descriptions, provides WSDL

UDDI (2)

- Initial vision: “[…] help companies conduct business with each other in an automated fashion [...]” [sys-con.com]
- Reality today: Human element stays important ->UDDI not very widespread
- Followed by: Business Process Execution Language (BPEL)
  - Specification of business processes based on Web Services

Representational State Transfer (REST)

REST

[Baker, op. cit.]

- REST
  - REST is an **architectural style** for distributed systems.
- An architectural style is:
  - … an abstraction, a design pattern, a way of discussing an architecture without concern for its implementation.
- REST defines a series of constraints for distributed systems that together achieve the **properties** of:
  - Simplicity, scalability, modifiable, performance, visibility (to monitoring), portability and reliability.
- A system that exhibits all defined constraints is **RESTful**!

Understanding REST

[Baker, op. cit.]

- Representational State Transfer:
  - The **Resource**:
    - A resource is any information that can be named: documents, images, services, people, and collections.
  - Resources have state:
    - State may change over time.
  - Resources have identifiers:
    - A resource is anything important enough to be referenced.
  - Resources expose a uniform interface:
    - System architecture simplified, visibility improved,
    - Encourages independent evolution of implementations.

Understanding REST (2)

[Baker, op. cit.]

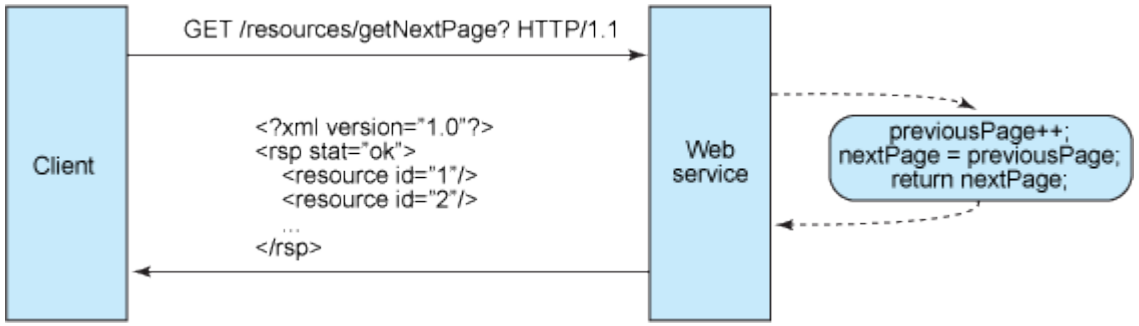
- Representational State Transfer:
  - On request, a resource may **transfer** a **representation** of its **state** to a client:
    - Necessitates a client-server architecture.
  - A client may transfer a proposed representation to a resource:
    - Manipulation of resources through representations.
  - Representations returned from the server should link to additional application state:
    - Clients may follow a proposed link and assume a new state → Hypermedia as the engine of application state.

Understanding REST (3)

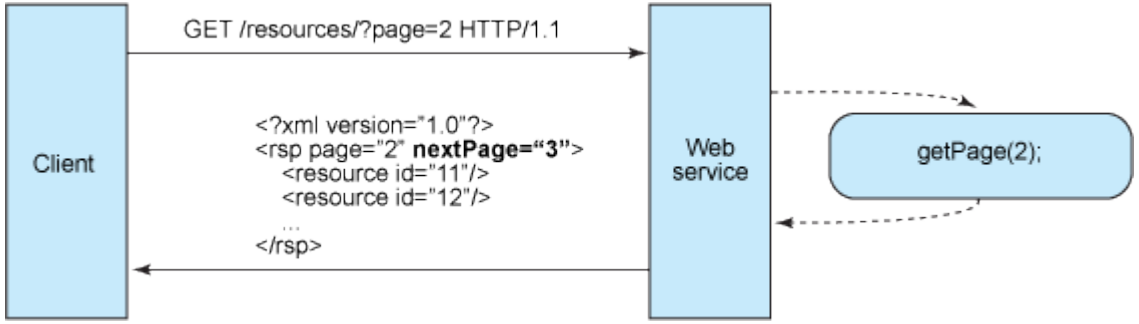
[Baker, op. cit.]

- Representational State Transfer:
  - **Stateless** interactions:
    - Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server.
  - Statelessness necessitates **self-descriptive** messages:
    - Standard media types,
    - Meta-data and control-data.
  - Uniform interface + Stateless + Self-descriptive = **Cacheable**:
    - Cacheable necessitates a layered-system.

Stateful vs. Stateless Design



Stateful design: Server keeps Client-related state (previousPage)



Stateless design: Client has to include explicit state information in the call, the Server does not keep any client-related state

REST for Web Services

- An alternative to SOAP/WSDL-based Web services
- Set of architectural principles by which one can design Web services that focus on a system's **resources**, including how **resource states** are addressed and transferred over **HTTP**
- Service access by **direct use of HTTP methods** in accordance to the semantics defined in RFC 2616.
- One-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods
  - To create a resource on the server (append), use HTTP POST method
  - To retrieve a resource, use HTTP GET method
  - To change the state of a resource or to replace, use HTTP PUT method
  - To remove or delete a resource, use HTTP DELETE method

Uniform Interface

- The semantics of the methods is constant and independent from the particular resource that is being addressed
- Evolution of services goes via new resources rather than calls
- Simplifies integration of services from different providers (service mashing)
- Simplifies error recovery by end systems and intermediate systems (caches) who can repeat idempotent calls automatically without having access / understanding of the “payload”
- Accompanied by a set of “Unified Status Codes”

Uniform Interface (2)

	GET	PUT	DELETE	POST
Resource URL e.g. *http://shop.oreilly.com/product/9780596529260.do*	**Retrieve**	**Create / Replace**	**Delete**	**Append / Modify**
	Safe Idempotent Cacheable	Idempotent	Idempotent	Not safe Not idempotent

Unified Status Codes

GET /product/9780596529260.do HTTP/1.1
Host: shop.oreilly.com

2xx Success	HTTP/1.1 200 OK Content-Type: text/html;charset=UTF-8 ...
3xx Redirect	HTTP/1.1 304 Not Modified ETag: “1234567890”
4xx Client Error	HTTP/1.1 401 Unauthorized WWW-Authenticate: Basic ...
5xx Server Error	HTTP/1.1 500 Internal Server Error ...

Narrow vs. Wide Interfaces

- The usage of just the HTTP methods in REST is an example of a more general class of service designs based on “narrow” interfaces
- Narrow interface service design
  - Limited set of common methods / procedures / functions
  - Services defined through the resources / arguments passed to the narrow method interface
  - Service design is focused on the resources, not on the functions

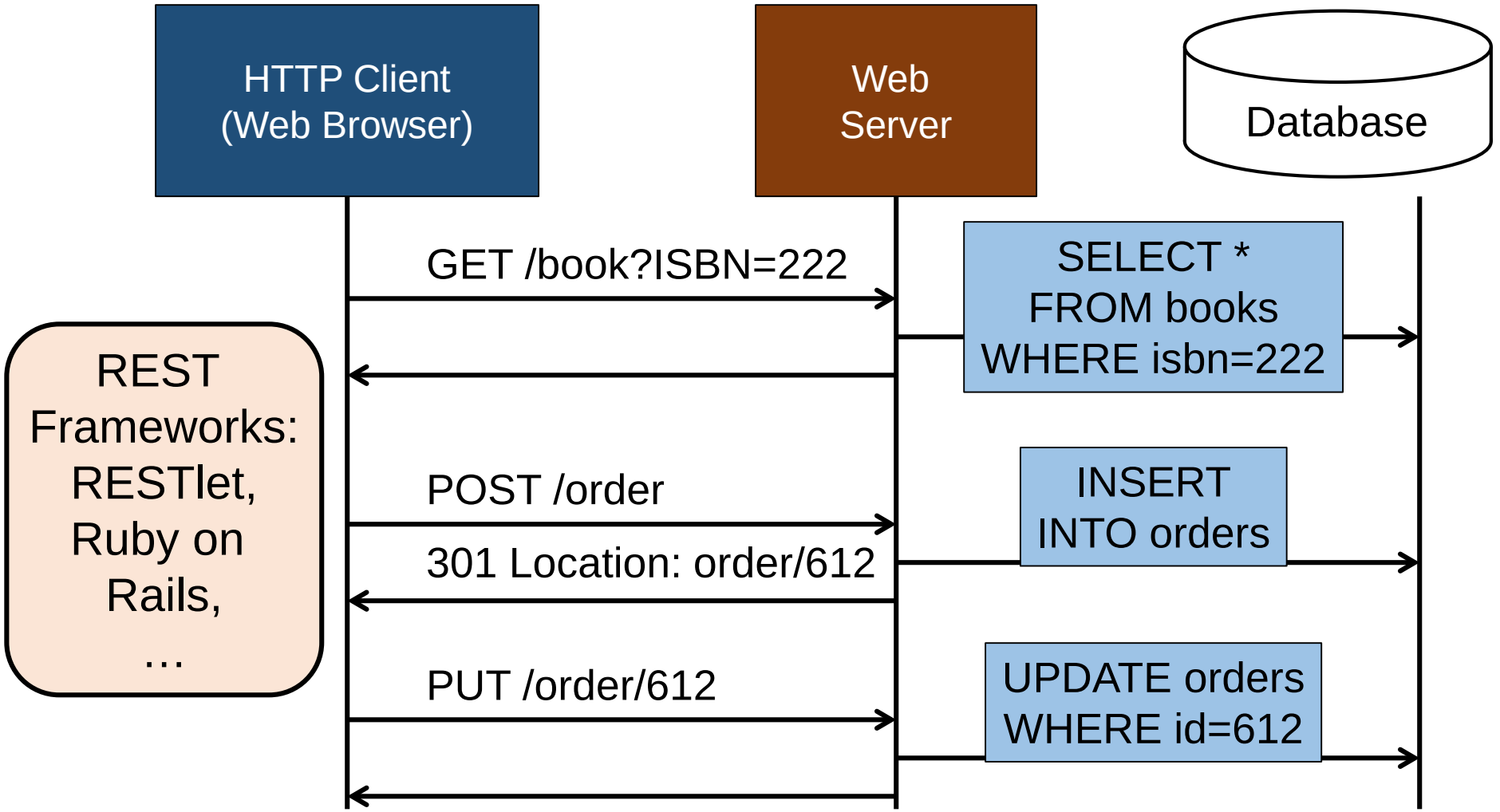
Narrow vs. Wide Interfaces (2)

- Examples:
  - UNIX: “Everything is a file descriptor”
    - Methods: create, open, read, write, sync, seek, close
    - Resource: a “file”, device, socket, etc. , addressed by a file descriptor
  - SNMP: Simple Network Management Protocol
    - Methods: snmpGet, snmpGetBulk, snmpGetList, snmpSet, etc.
    - Resource: managed “objects” identified by object identifiers (OIDs), organized in hierarchical Management Information Base (MIB)

Summary: Comparison of SOAP and REST

Features	SOAP-Style	REST-Style
Interaction	Stateful	Stateless
Interface	Specified by description language (WSDL, IDL)	Uniform interface
Data Format	Specified by description language (WSDL, IDL)	MIME Types (negotiation)
URI	Service	Resource
Payload	Opaque message	Self-described message
Performance	Hard to cache	Easy caching

REST - Example



REST Example using JAX-RS

- JAX-RS = Java API for RESTful Web Services
- https://javaee.github.io/tutorial/jaxrs003.html

HelloWorld.java running in webserver:

HTTP method

MIME type

```
@GET
@Produces("text/html")
public String getHtml() {
    return "<html lang=\"en\"><body><h1>Hello, World!!</body></h1></html>";
}
```

Client call from browser:

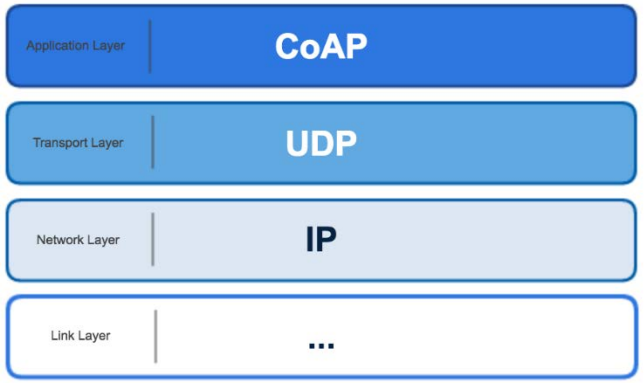
```
http://localhost:8080/HelloWorldApplication/HelloWorldApplication
```

A browser window opens and displays the return value of Hello, World!!

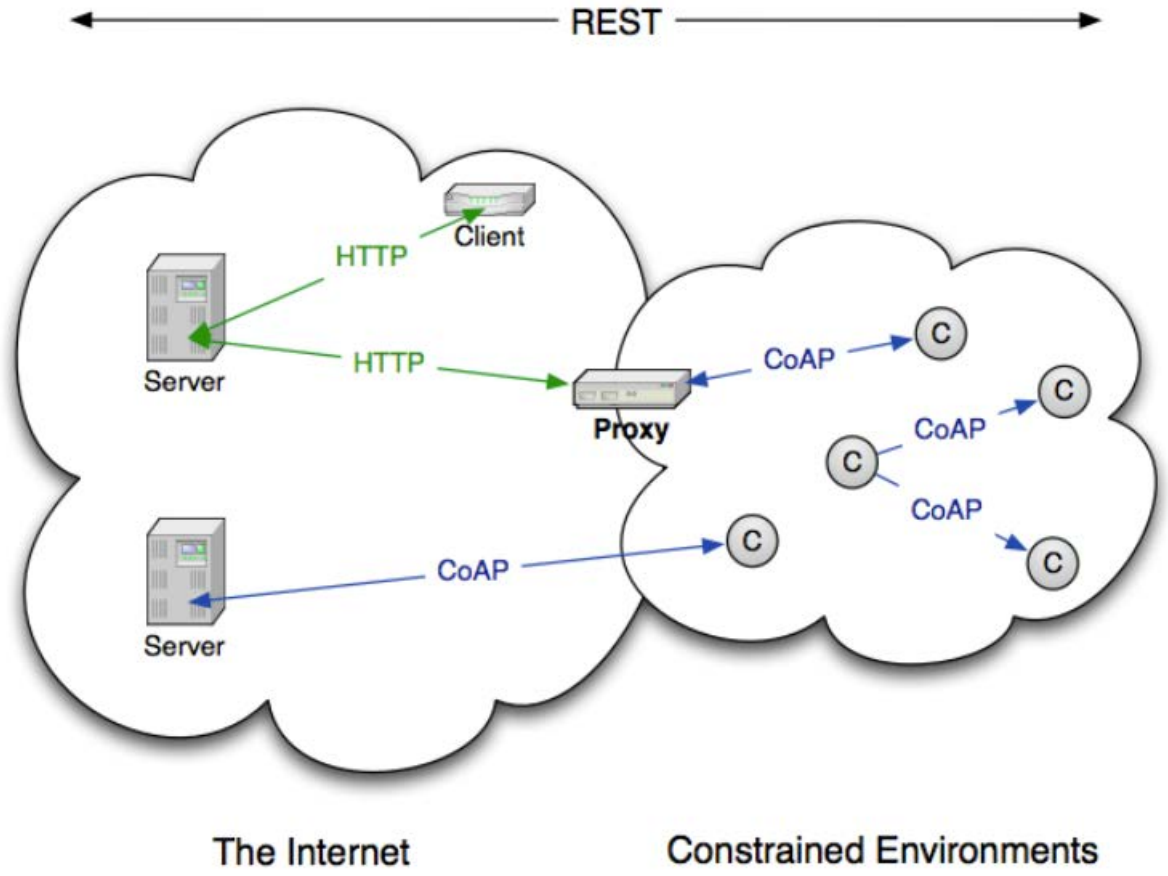
Constrained Application Protocol (CoAP)

Constrained Application Protocol (CoAP)

- Adapts the HTTP's REST model to resource constrained devices (sensors and actuators)
  - Keeps the uniform method set (GET, PUT, DELETE, POST) thus easy automatic translation between HTTP to CoAP
  - Simple transport protocol - datagram only (UDP), ?recovery?
  - Different options for data formats (serialization), negotiated
  - Integrated discovery, stronger structuring of the names
- Example:
  - GET *coap://temp1.25b006.floor1.example.com/temperature*
    - ASCII string: 22.5
  - PUT *coap://bluelights.bu036.floor1.example.com/intensity*
    - ASCII string: 70 %

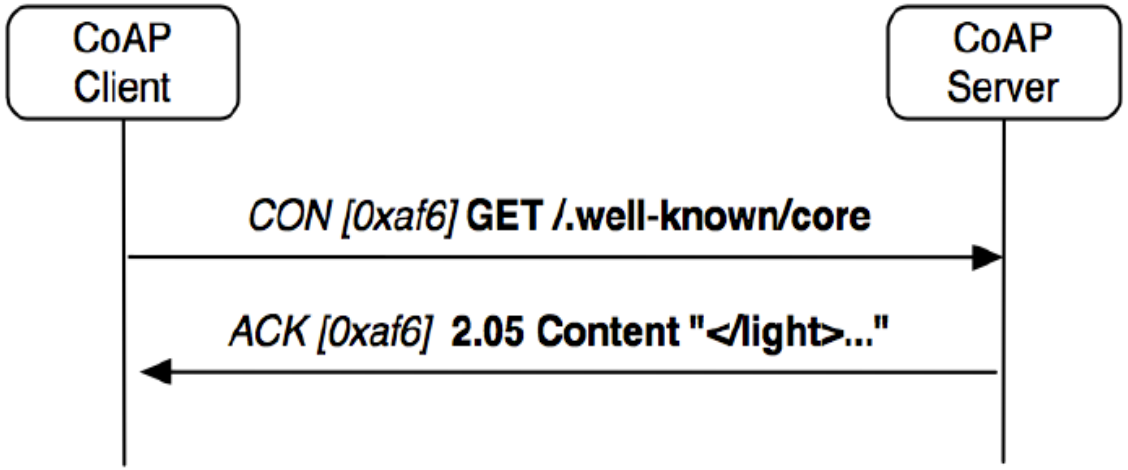


CoAP Network Architecture



CoAP Resource Discovery (within a Server)

- GET *coap://25b006.floor1.example.com/.well-known/core*
  - </temp>; n=”TemperatureC”,
  - </light>; ct=41; n=”LightLux”
- Well known service generates a list of all services available on the server of interest (including content format)



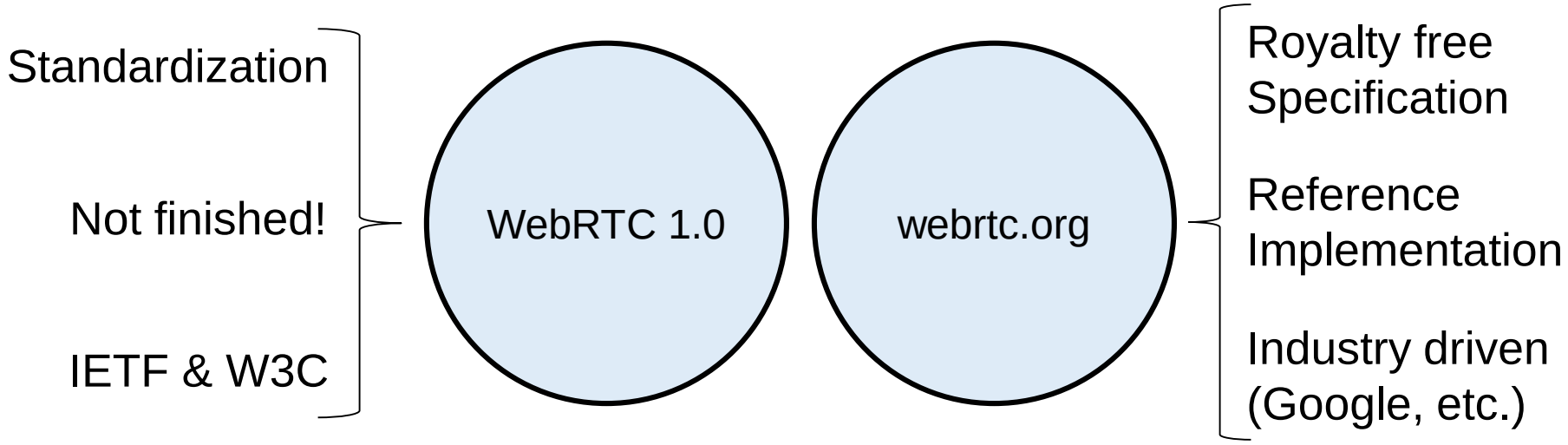
Web Real-Time Communication (WebRTC)

Web Real-Time Communication (WebRTC)

- Goals of WebRTC:
  - Enable Real-Time Communications (RTC), such as multimedia calls (Skype, Hangouts, Facetime, etc.) within web applications,
  - Without third party plugins,
  - Peer-to-peer,
  - Industry-standard end-to-end encryption for all calls,
  - Sending arbitrary data,
  - Without sophisticated Telco equipment as backend.

WebRTC

- WebRTC is both the open-source reference implementation and a standardization effort:
  - webrtc.org

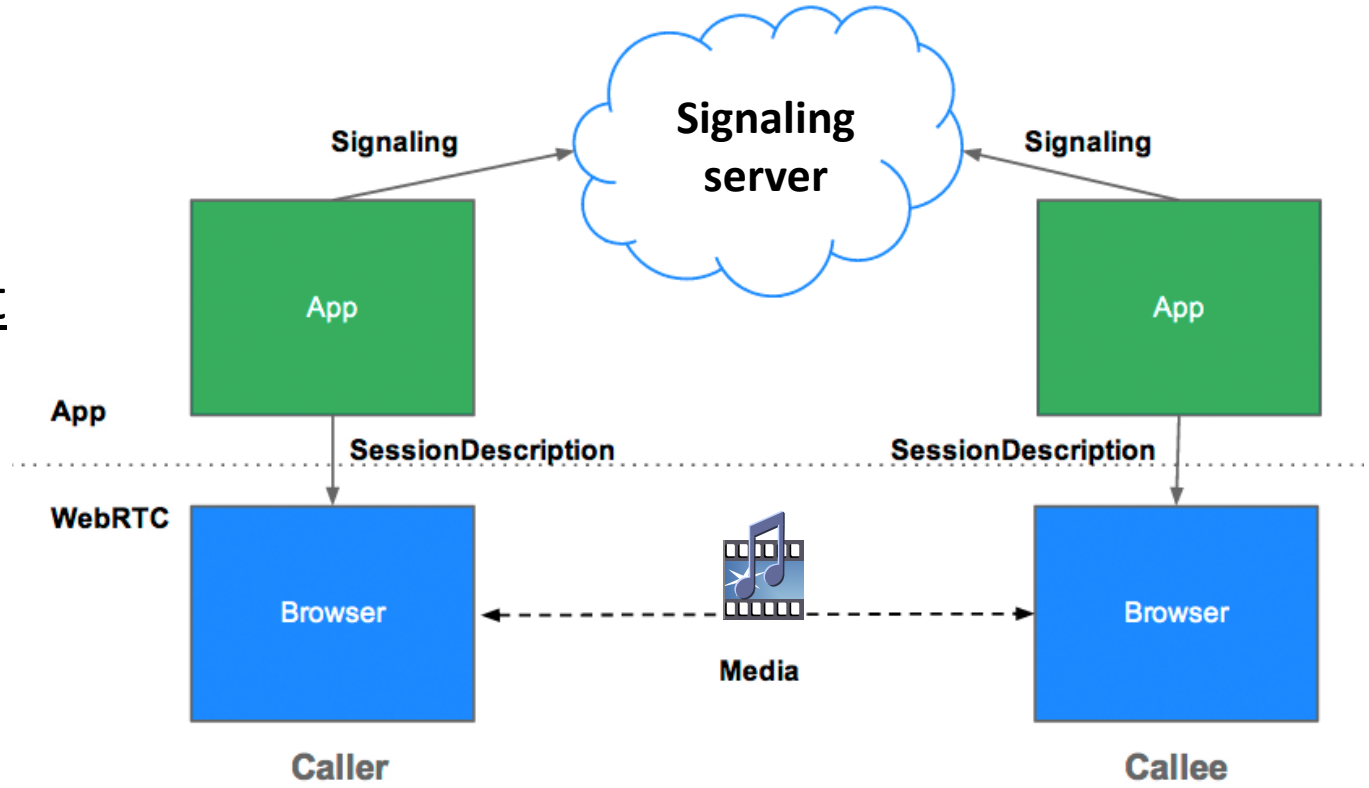


The three WebRTC APIs

- WebRTC implements three APIs that are offered by the browser and can be called using (client-side) JavaScript
- *MediaStream*
  - Easy access to cameras, microphones, as well as screen capturing streams from web application
  - Automatically synchronized (e.g. audio & video)
- *RTCPeerConnection*
  - Enables establishing peer-to-peer bi-directional connections to transfer MediaStreams
  - Automatic encoding/decoding, buffering, bandwidth management
- *RTCDataConnection*
  - Enables sending arbitrary data to other peers

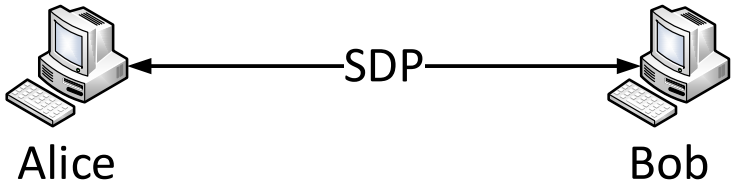
Abstract functions needed for making a call

1. Discover the callee (not part of WebRTC!)
2. Exchange stream metadata and agree on format (signaling format SDP part of, transport not part of WebRTC – could be XMPP, Websockets, etc.)
3. Exchange network information and establish connection (direct, NAT-traversal, or by relay server)



Signaling with Session Description Protocol (SDP)

- Used for negotiating session capabilities between peers
  - … based on SDP Offer/Answer exchange mechanism [RFC 3264]



- Session description includes information about media streams,
- Media codecs and their parameters, and more
- Procedure:
  1. Caller application requests SDP string (representing media metadata) from browser and sends offer with SDP string to callee
  2. Callee saves remote SDP string and requests own SDP string; answers to caller with own SDP string
  3. Caller saves remote SDP string

[Offer]
v=0
o=alice 2890844526 2890844526 \
IN IP4 host.atlanta.example.com
s=
c=IN IP4 host.atlanta.example.com
t=0 0
m=**audio** 49170 RTP/AVP 0 8 97
a=rtpmap:0 PCMU/8000
a=rtpmap:8 PCMA/8000
a=rtpmap:97 **iLBC**/8000
m=**video** 51372 RTP/AVP 31 32
a=rtpmap:31 **H261**/90000
a=rtpmap:32 MPV/90000

session originator

connection address

port

2 media streams

[Answer]
v=0
o=bob 2808844564 2808844564 \
IN IP4 host.biloxi.example.com
s=
c=IN IP4 host.biloxi.example.com
t=0 0
m=audio 49174 RTP/AVP 0
a=rtpmap:0 PCMU/8000
m=video 49170 RTP/AVP 32

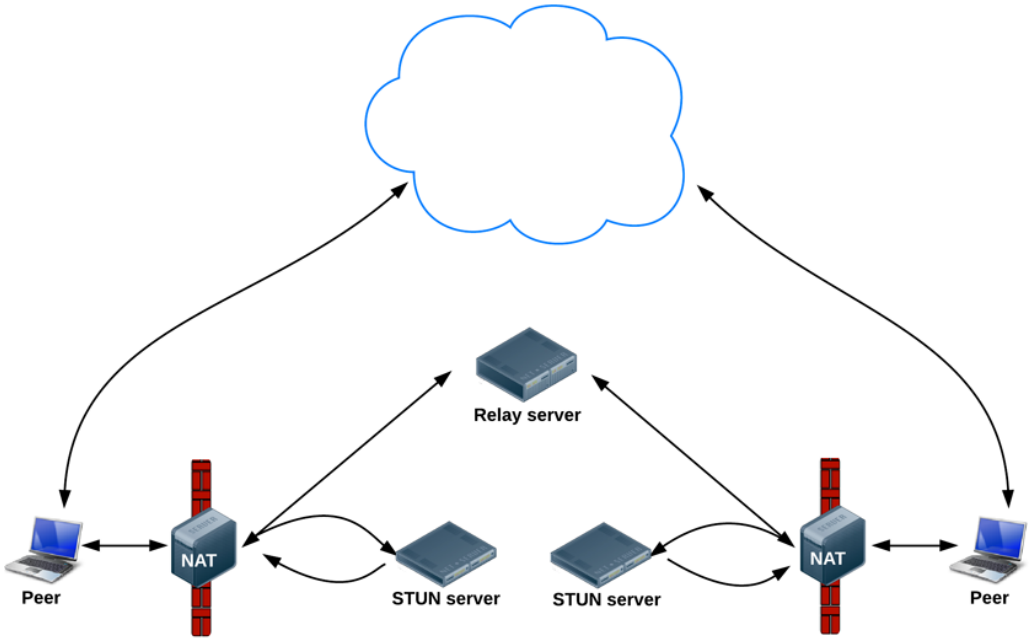
session duration

Extension OTRC (also referred to as WebRTC 1.1)

- Shortcomings of SDP: no control by application, browser decides on formats etc.
- OTRC (Object RTC) exposes media and transport parameters via mutable objects – parameters can be changed on the fly
- This enables the Javascript application to:
  - “Warm up” media path while “ringing”
  - Change send codec on the fly
  - Change the camera source instantly (e.g. front to back on mobile)
  - Enable/disable sending of media instantly (without signaling)
  - Set a maximum bitrate or maximum framerate
  - Send simulcast
- Functionality is gradually added to WebRTC reference implementations and ongoing standardization effort

NAT Traversal using STUN and TURN

- **Problem**: your internal IP may be 192.168.1.100 and external IP may be 147.232.159.135
- ICE (Interactive Connectivity Establishment) allows for discovering all those IPs to find a way for the remote peer to reach your host
- Peers try to connect by:
  1. Obtaining own public IP using (Session Traversal Utilities for NAT (STUN) servers (i.e. like WhatIsMyIP.com for WebRTC)
  2. Connecting to directly using UDP, fallback to TCP on HTTP(S) ports
  3. Use relay (Traversal Using Relays around NAT (TURN)) servers to relay streams to other peer (if a peer-to-peer connection cannot be established due to blocked ports)



WebRTC - Example

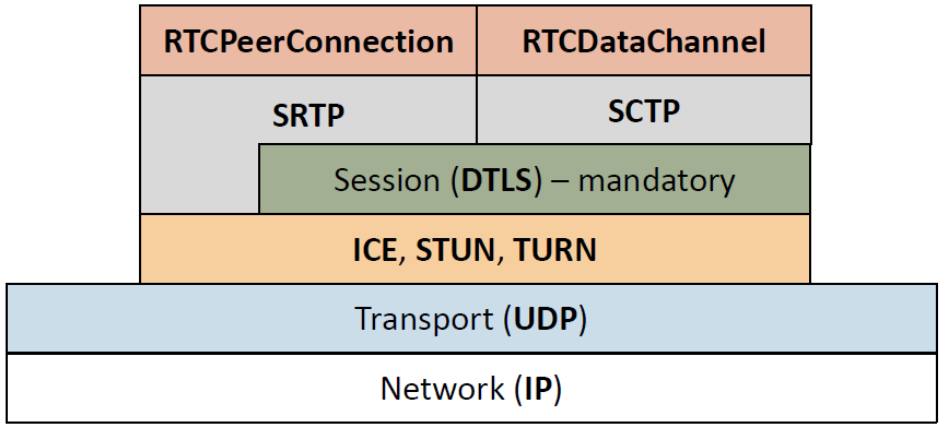
- Initialization of a WebRTC call in a nutshell:

```
var pc = new RTCPeerConnection();
navigator.getUserMedia({video: true}, function(stream) {
        pc.addStream(stream);
        pc.createOffer(function(offer) {
                pc.setLocalDescription(new RTCSessionDescription(offer),
                function() {
                        // Send the offer to a signaling server to be forwarded to the peer.
                }, errorCb);
        }, errorCb);
});
```

WebRTC Protocol Stack

[Matsak, Grigorik, op. cit.]

- Peer-to-peer connection is established over UDP using ICE, STUN and TURN
  - We can live without a few lost frames; low latency is more important
- DTLS (Datagram Transport Layer Security) is used to secure all data transfers between peers
  - Unlike TLS, DTLS can be used over UDP
  - Encryption is WebRTC's mandatory feature

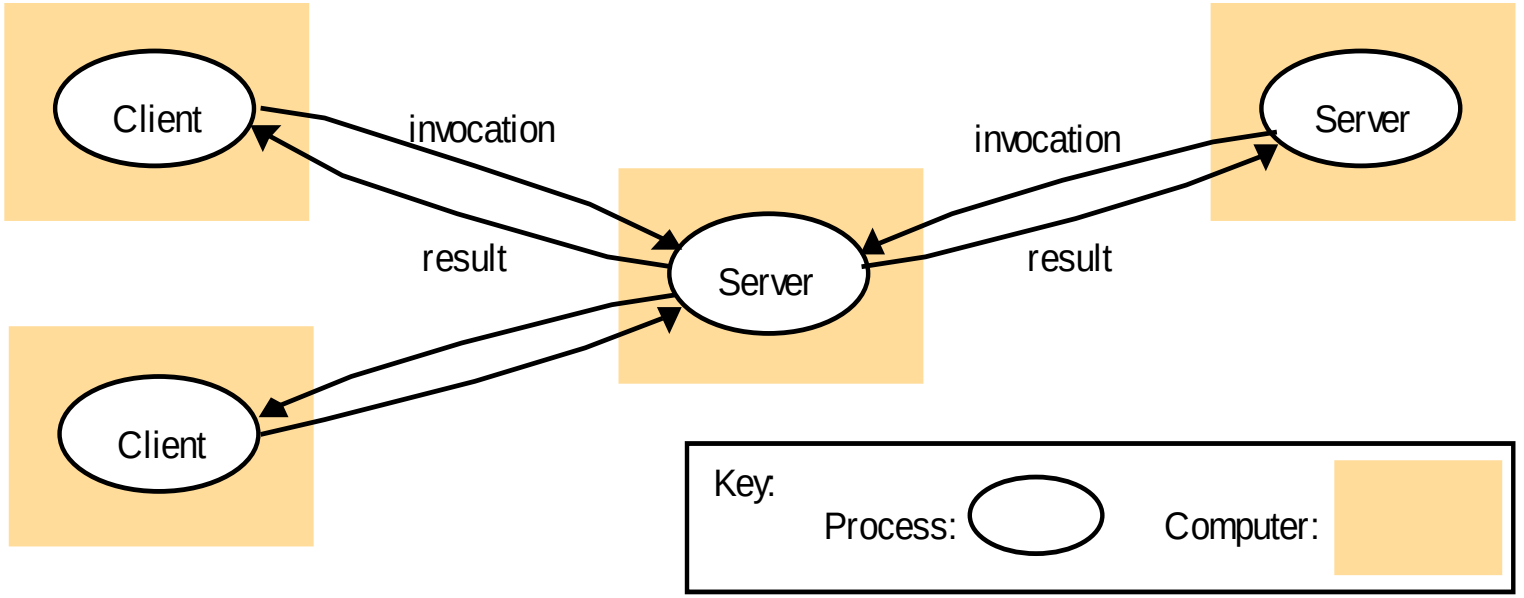


- SRTP (Secure Real-Time Protocol) is used to transport audio and video streams
- SCTP (Stream Control Transport Protocol) is used to transport application data

Service Environment – Implementation Issues

Client Server Architecture

- Approach: server does not have to do all the work himself!
- Web application is divided into **components:**
  - Software units that are independently replaceable, upgradable & deployable
- Can now use different servers for different tasks in one request (or just balance the load between identical servers)



Microservices

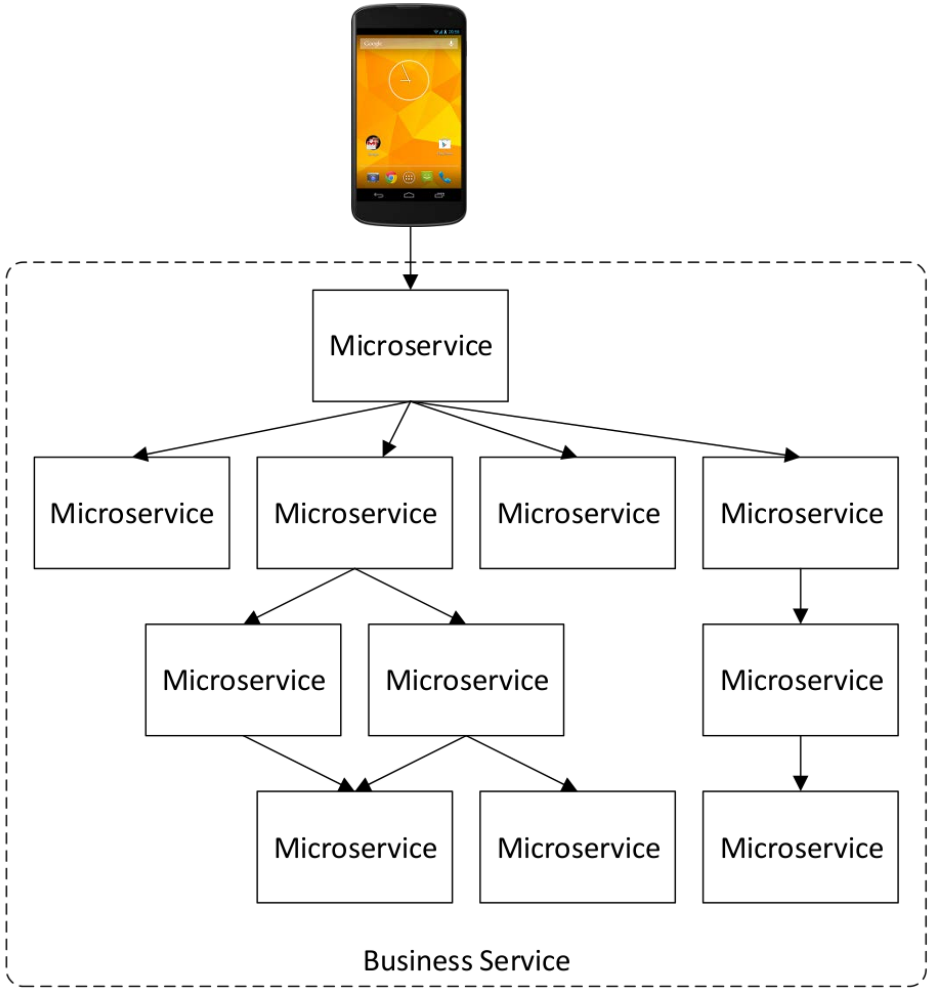
[Newman, op. cit.]

- Pushing the SOA “separation of concerns” approach to the extreme, by applying the same design principles at a very small level of granularity
- Design of complex systems as a collection of small, autonomous services that work together
  - Focused on doing one thing well
  - Each service exposes an application programming interface (API), and collaborating services communicate only via those APIs
  - All communication between services is via network calls, to enforce separation between and avoid perils of tight coupling
  - The services need to be able to change independently of each other, and be deployed by themselves without requiring consumers to change

Microservices

[NICTA, op. cit.]

- This leads to a **Microservice** architecture
- Each user request is satisfied by some sequence of service invocations
- Service depth may be up to 70! (example: LinkedIn)
- Micro-services ownership:
  - In the case of big companies mostly internal services (not externally available)
  - But: Parts of the services can be provided by other parties:
    - Amazon Simple Storage Service (S3)
    - Google BigQuery (Data Analysis)



Characteristics of Microservices

- The main reason for this architecture is **decoupling**:
  - Independently developed (e.g. different programming languages)
  - Independently deployable
    (only parts of the system are changed at a time)
  - Independently scalable
- That means that services can be deployed and scaled independently on distributed computing resources
  (today mostly using cloud computing)
- Main downside:
  - Remote call always “more expensive” than local library call

Microservices Benefits

[Newman, op. cit.]

- Heterogeneity
  - With a system composed of multiple, collaborating services, different and optimal technologies can be used inside each one
- Resilience
  - If one component of a system fails, but that failure doesn’t cascade, the problem can be isolated and the rest of the system can continue working
- Scaling
  - Instead of scaling everything together, scale only services that need scaling, allowing to run other parts on less powerful hardware

Microservices Benefits (2)

[Newman, op. cit.]

- Ease of deployment
  - Changes can be made to individual services which can be deployed independently of the rest of the system
- Evolvability
  - Due to the smaller size, the barriers to rewriting or removing services entirely are very low
- Composability
  - Enables functionality to be combined in different ways for different purposes, and increases the number of internal interfaces that are addressable by outside parties.

Cloud Computing

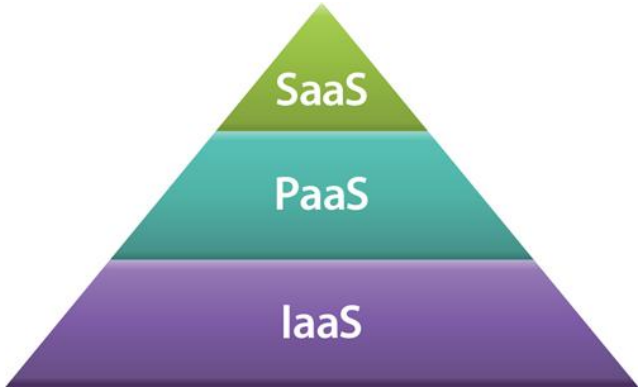
- Common definition by NIST:
  - “Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”
- Main characteristics:
  - On-demand self-service (without human intervention at provider)
  - Broad network access (available form anywhere)
  - Resource pooling (dynamically assigned according to demand)
  - Rapid elasticity (automatic, rapid scaling)
  - Measured service (resource usage monitored, controlled & reported)

Cloud Deployment Models

- The definition of cloud does not specify ownership, size, or access of a cloud
- Based on those characteristics, three main deployment models are usually distinguished:
  - **Public cloud**:
    - A cloud platform run by a service provider made available to many end-user organizations
  - **Private cloud**:
    - A cloud platform run solely for a single end-user organization, such as a bank or retailer
  - **Hybrid cloud:**
    - Most organizations will probably use some hybrid of both

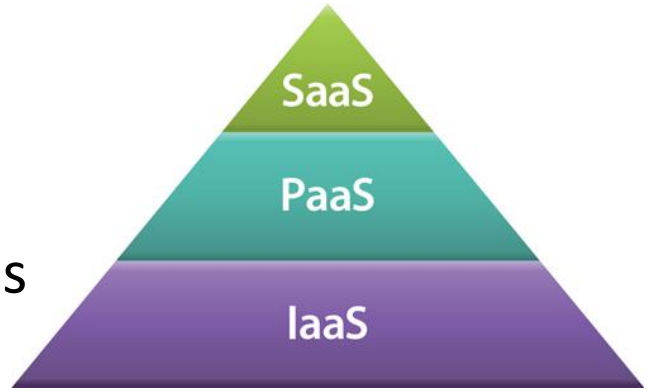
Cloud Service Models

- There are different levels of services cloud providers can offer:
  - **Infrastructure as a service** (IaaS)
    - Provide physical or virtual machines
    - Access to operating system & internal services like distributed persistence, messaging, firewall, VLAN, …
  - **Platform as a service** (PaaS)
    - Provide development environment
    - Manage the underlying hardware and software layers
    - User is programming using some API
    - Example: Google App Engine -> Java Servlets (Java software component running in web container)



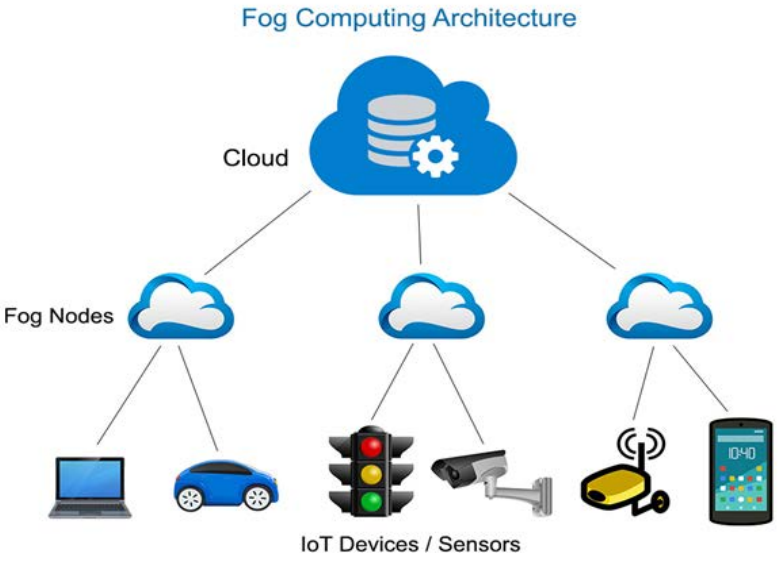
Cloud Service Models (II)

- There are different levels of services cloud providers can offer:
  - **Software as a service** (SaaS)
    - Most abstract view on a cloud service
    - Provide an application or service
    - Manage infrastructure, platforms & runs the applications
    - Example: Google Drive, ownCloud
  - Also: **Backend as a service** (BaaS)
    - Provide a “complete” service set for apps and mobile web sites
    - Like: Push notifications, cloud storage etc.



Fog Computing

- What are the main downsides of (public) cloud computing?
  - A lot of data is (unnecessarily) sent to the cloud, while local processing would be possible
  - Resources typically in a datacenter not necessarily close to client
  - This inevitably leads to **delay** and a lot of **traffic** in the core network
- Possible solution:
  - **Fog Computing** (because the fog is a cloud close to the ground)
  - Cloud close to the edge of the network (close to the user):
    - Geographical distribution
    - Lower latency
    - Only subset of data is sent to traditional cloud computing data centers
    - Location awareness
    - Even imaginable: mobility



Source: https://bit.ly/2F1CosM