# SERVICE-ORIENTED ARCHITECTURES

Distributed Systems

4. Sem BSc Informatics

IMC FH Krems

# LECTURE OUTLINE

- Service Oriented Architecture (SOA)
  - Definition
  - REST
  - Web Services (SOAP, JSON-RPC, gRPC)
  - Microservices
- Message-Oriented Middleware (MOM)
  - Enterprise Bus (ESB)
  - Publish-subscribe
  - Queuing
  - Messaging

# LECTURE OUTLINE

- **Service Oriented Architecture (SOA)**
  - Definition
  - REST
  - Web Services (SOAP, JSON-RPC, gRPC)
  - Microservices
- Message-Oriented Middleware (MOM)
  - Enterprise Bus (ESB)
  - Publish-subscribe
  - Queuing
  - Messaging

# SERVICE ORIENTED ARCHITECTURE

Definition (W3C)

- Form of distributed systems architecture.
- Properties

  1. Logical view: abstracted view of applications **(services)** based on **what they do**, formally defined in terms of **messages.**
  2. Message-oriented: internal structure of services is abstracted away → loose coupling. Services must adhere to formal service definition.
  3. Description-oriented: metadata exposes publicly only details important for using the service. Semantics should be documented by this description.

# SERVICE ORIENTED ARCHITECTURE

Definition (W3C)

- Properties

  4. Granularity: only few operations, messages large.
  5. Network-orientation.
  6. Platform-neutral: standarized format for messages.

# SERVICE ORIENTED ARCHITECTURE

**Advantages**

- Loose coupling between services (w.r.t. hard coupling with remote objects).
- Support for heterogeneous implementations.
- Common protocols and technologies (HTTP, XML, JSON).
- Not only exchanging information → paradigm for programming, interacting with and integrating existing systems.

# REST

Representational State Transfer (REST)

- Key concept: **resource.**
- A resource is an abstract **data entity.**
- Resources are abstracted from their representation (documents, images, PDF, XML, JSON, etc).
- Resources are accessed via URIs.
- Communication via HTTP and standard HTTP CRUD verbs.

| HTTP Verb | Meaning |
|---|---|
| GET | Retrieve a resource |
| PUT/PATCH | Update a resource (total/partial) |
| DELETE | Eliminate a resource |
| POST | Create a resource |

# REST

Representational State Transfer (REST): four principles

1. Resources are identified with URIs of type URL.
2. Uniform Interface: using the HTTP standard.
3. Self-descriptive message: messages includes enough information to know how to interpret the message. Use of metadata (headers).
4. Stateless: State is always transferred between interactions.
   - Advantages? Disadvantages?

# REST

Representational State Transfer (REST): four principles

- Lightweight infrastructure.
- Services can be tested with an ordinary web-browser.
- Scalable (stateless).
- Some popular frameworks:
  - Spring Framework (Java)
  - ASP.NET (C#)
  - Flask, Django, Fast API (Python)

# REST

- Example: AWS Simple Storage Service (S3)
- Buckets contain objects (files)
- Create a bucket

```
POST /examplebucket HTTP/1.1
Host: s3.amazonaws.com
Date: Mon, 11 Apr 2016 12:00:00 GMT
x-amz-date: Mon, 11 Apr 2016 12:00:00 GMT
Authorization: authorization string
```

- Put an object on a bucket

```
POST /file.jpg HTTP/1.1
Host: examplebucket.s3.amazonaws.com
Date: Mon, 11 Apr 2016 12:00:00 GMT
x-amz-date: Mon, 11 Apr 2016 12:00:00 GMT
Authorization: authorization string
```

# REST

- Example: AWS Simple Storage Service (S3)
- Delete a bucket

```
DELETE /examplebucket HTTP/1.1
Host: s3.amazonaws.com
Date: Mon, 11 Apr 2016 12:00:00 GMT
x-amz-date: Mon, 11 Apr 2016 12:00:00 GMT
Authorization: authorization string
```

- We will see (and build) an example in the exercises!

# WEB SERVICES

- **Service:** application that is loosely coupled, reusable, coarse-grained, discoverable and self-contained. Interaction via messages.
- **Web service**: service which is used and accessed via the web (HTTP)
- **W3C definition:** "A software system designed to support interoperable machine-to-machine interaction over a network".
- Interface definition over standard format (WSDL – Web Services Description Language)
- Interaction via **Simple Object Access Protocol (SOAP)**: exchange of information in XML format.

# WEB SERVICES

Workflow for implementing a SOAP web service:

1. Write WS definition in WSDL (*hello.wsdl*)
   - Definitions.
   - Messages: Request and Response types.
   - Port type: define operations <Request, Response>.
   - Bindings: define input/output types.
   - Service: define address and binding.

# WEB SERVICES

Workflow for implementing a SOAP web service:

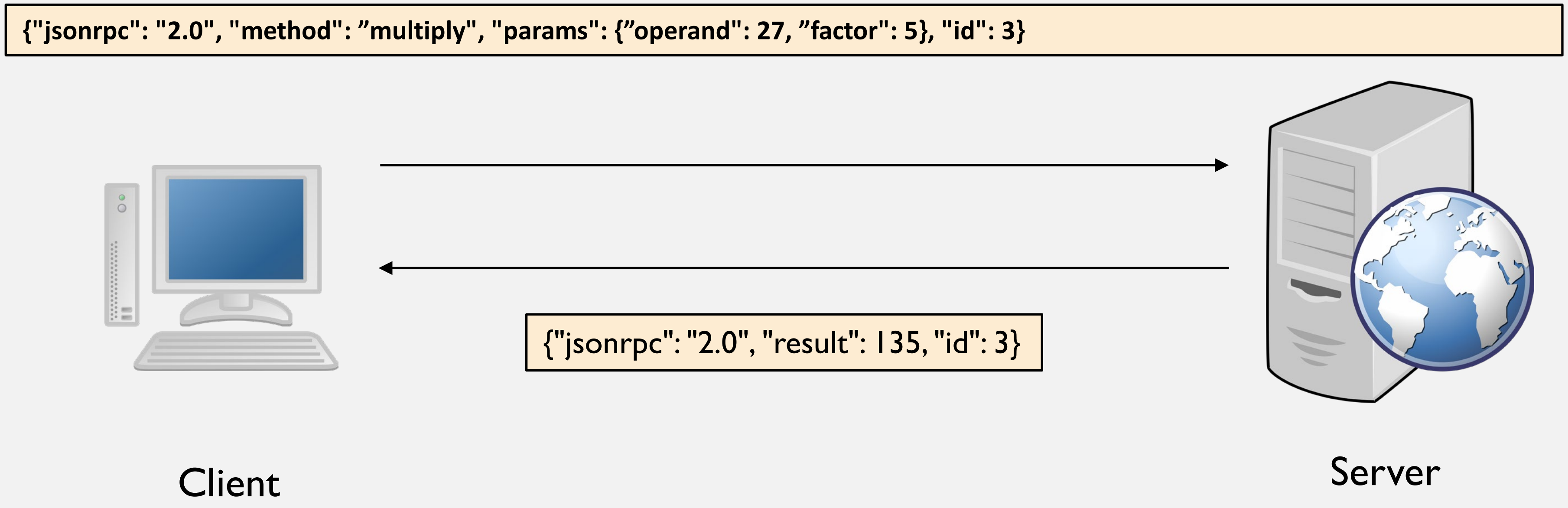
2. Implement server-side stubs.
3. Implement client-side stubs.
4. Set up server.
5. Done!

→ We will see (and build) an example in the exercises!

# WEB SERVICES

Other types of web services:

- JSON-RPC: used to call remote procedures via JSON objects



```
{"jsonrpc": "2.0", "method": ”multiply", "params": {”operand": 27, ”factor": 5}, "id": 3}
```

```
{"jsonrpc": "2.0", "result": 135, "id": 3}
```

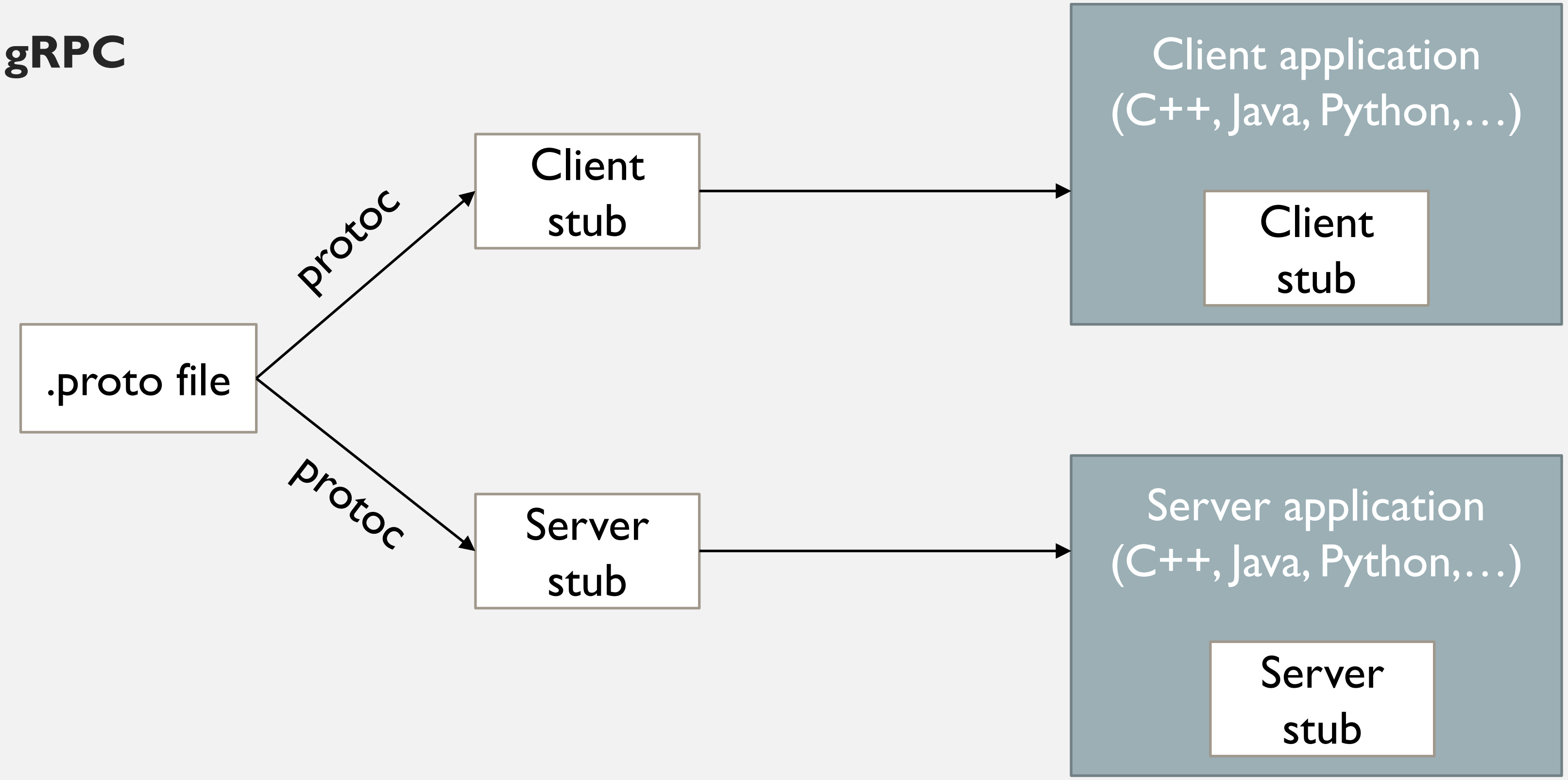
# WEB SERVICES

**gRPC**

- Proposed by Google (2015).
- Evolution of internal general-purpose RPC infrastructure called **Stubby.**
  - Made open-source as general API framework for microservices.
- Lightweight, efficient and representation-agnostic.
- Uses **ProtocolBuffers** as Interface Definition Language (IDL) and message payload.
- Google APIs have gRPC versions too.
- **HTTP/2.**
- Client/Server implementations for many languages like Java, C++, Python, Rust, etc. and OS (Windows, Linux, Mac).

# WEB SERVICES

**gRPC**


.proto file
protoc
protoc
Client stub
Server stub
Client application (C++, Java, Python,…)
Client stub
Server application (C++, Java, Python,…)
Server stub

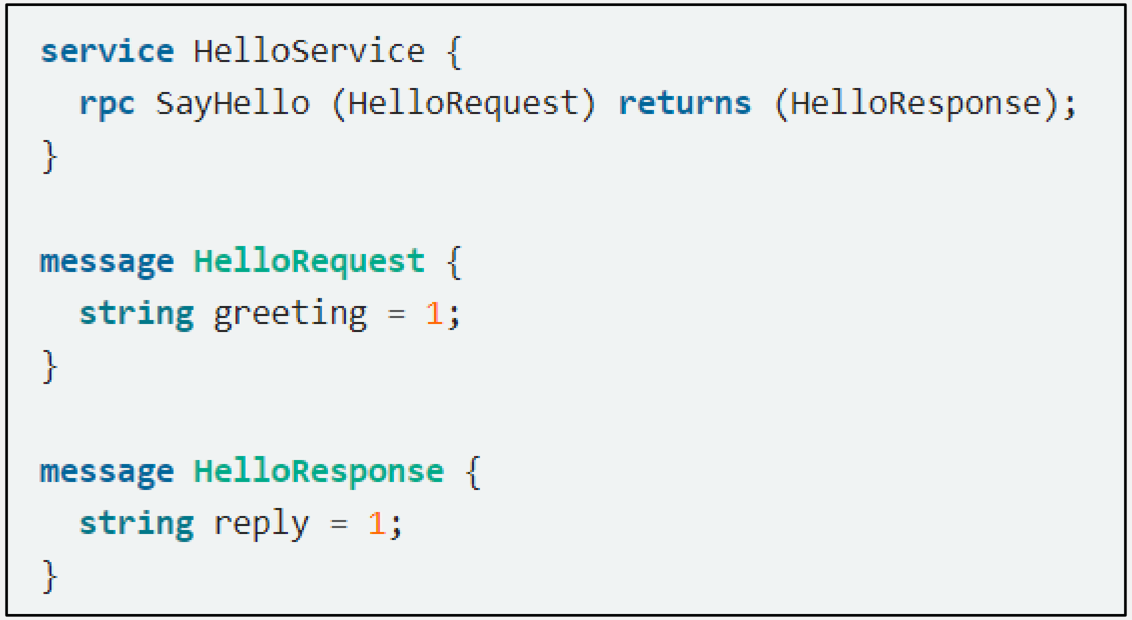
protoc generates

- Classes
- Methods
- Parsing
- Serializing

# WEB SERVICES

## gRPC

Example .proto file



```
service HelloService {
  rpc SayHello (HelloRequest) returns (HelloResponse);
}

message HelloRequest {
  string greeting = 1;
}

message HelloResponse {
  string reply = 1;
}
```

# WEB SERVICES

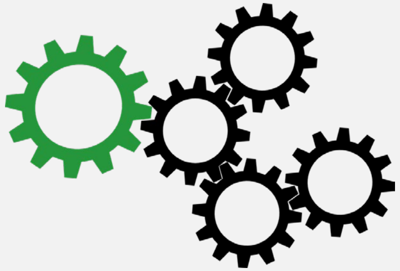
**gRPC**

- Types of RPC calls:
  - Synchronous/asynchronous.
  - Unary → single request/single response.
  - Server streaming → server streams messages as response of single request.
  - Client streaming → stream of requests from client / single response from server.
  - Bidirectional → (possibly) independent streams of requests / responses.
- Advantages/disadvantages when compared to REST?

# MICROSERVICES

- Increasingly popular software architectural style for distributed applications.
- No generally accepted definition
  - Term “microservice” first proposed in a workshop in 2011
- “Fine-grained SOA” (A. Cockcroft, Netflix).
- Can be regarded as „extreme“ SOA + organizational properties.
- Additional characteristics not found in the SOA paradigm.

# MICROSERVICES



1. Modularization by means of **services**
   - As opposed to libraries in traditional modular applications.
   - Services can be indenpendently upgraded and replaced.
   - How "micro" a service is depends strongly on context.
   - Service interface definition and coordination with clients (frequently inside an organization).
     - **Tolerant Reader** → Clients should be as tolerant as possible.
       - Code should not break if a new version is deployed.
       - Robustness principle: „*Be conservative in what you do, be liberal in what you accept from others.*"
   - Calls to service are more costly than library calls → service calls usually coarse grained.
   - A service may consist of multiple processes (i.e. backend + DB).

# MICROSERVICES

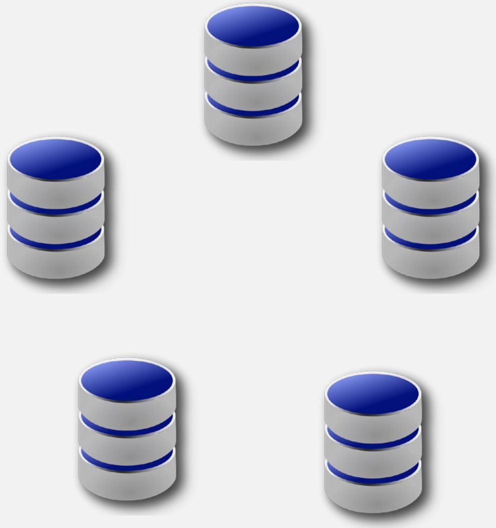
2. Service **ownership**
   - Popularized by Amazon's "you build it, you run it".
   - Development team responsible for build and operations.
   - On-going relationship with the product.
   - Dezentralized governance.
     - Each team responsible for their technical choices (languages, frameworks, storage).
   - Automated build and deployment (CD/CI).
     - Probably with own tools (frequently open-sourced).
   - Service contract definitions published for other services.

# MICROSERVICES

3. **Lightweight** middleware
   - In contrast to large, complex, proprietary enterprise middleware.
     - Specially lightweight message-oriented middleware (MOM).
   - Often open-source solutions:
     - RabbitMQ.
     - Kafka.
   - Use of web protocols (REST, gRPC, JSON-RPC).

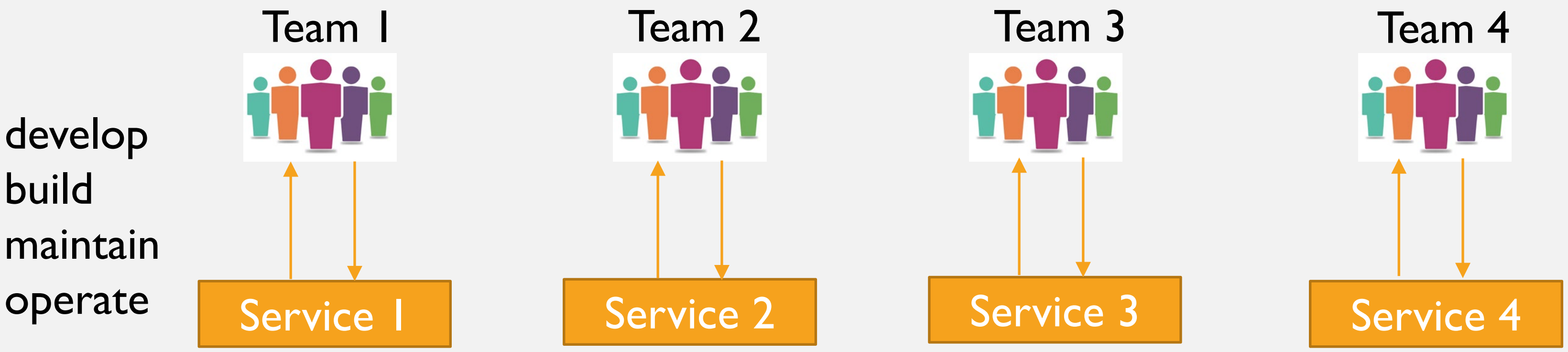
# MICROSERVICES

4. **Decentralized** data management
   - Domains differ between services.
   - Often overlapping domain entities.
   - Context boundaries mapping to service boundaries.
     - Most important: where to draw the service boundary.
   - Transactionless updates between services.
     - Can not *always* guarantee consistency → eventual consistency.
     - Ad-hoc mechanisms to restore consistency necessary.



# MICROSERVICES

5. Coupled to **organization** structure
   - Service ownership → organizational unit needed.
   - Services structure maps to organization's structure (Conway's Law).



Team 1

Team 2

Team 3

Team 4

develop
build
maintain
operate

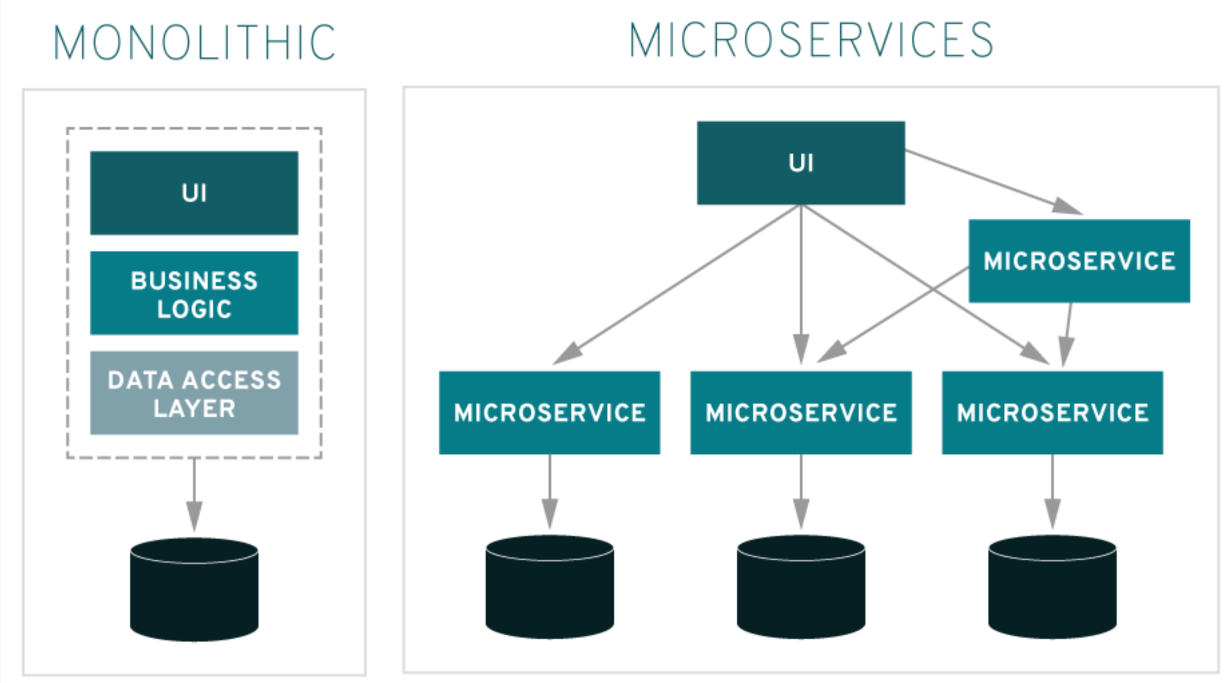
Service 1

Service 2

Service 3

Service 4

# MICROSERVICES



Source: Mert Gültekin, published on Medium

# MICROSERVICES

**Amazon**

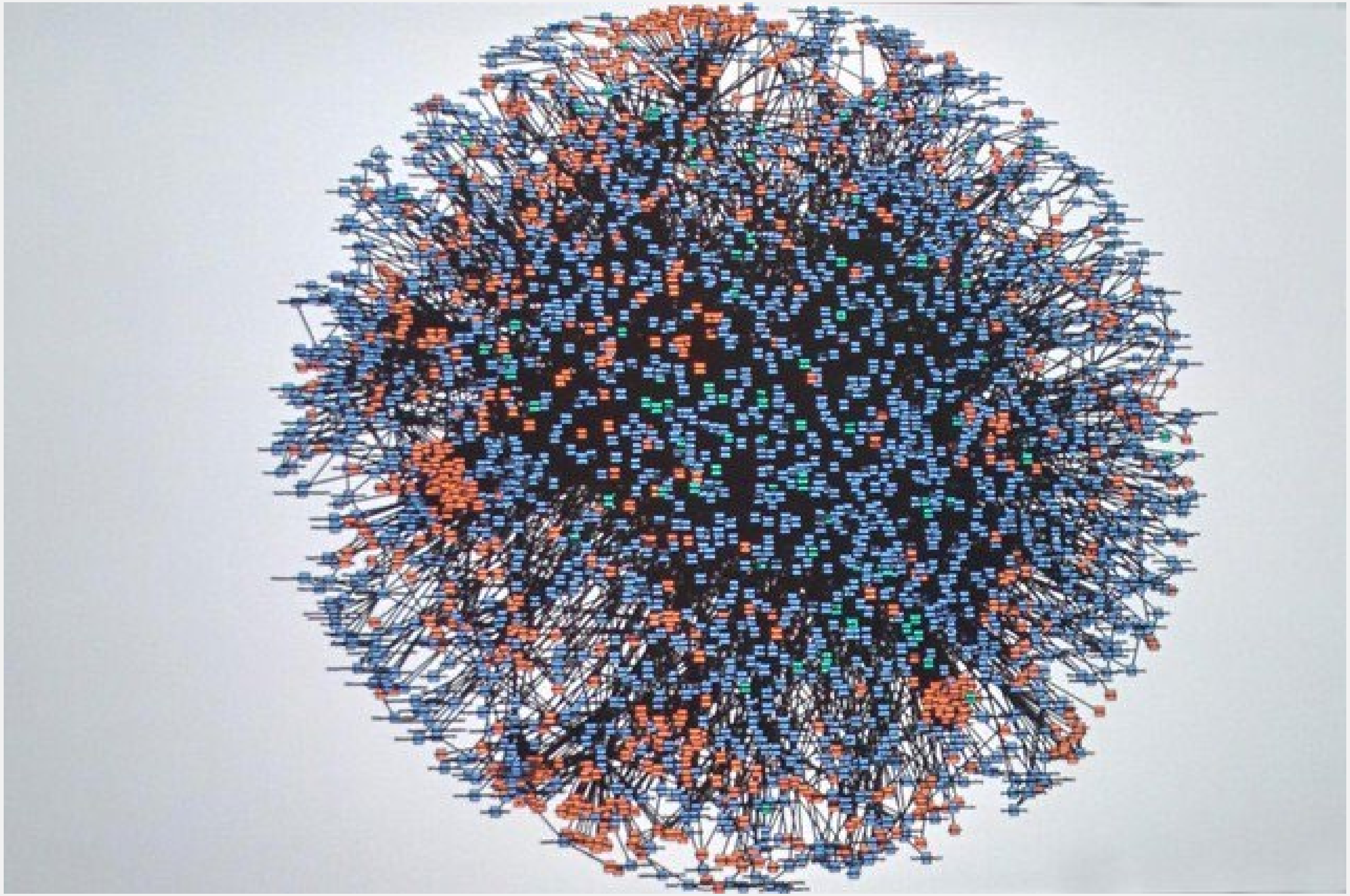
- Back in 2001, Amazon’s retail website was a monolithic application.
  - Multi-tiered but highly coupled.
- As developer based grew, monolithic architecture caused process overhead.
  - Updates on a single part may disrupt the whole application.
  - Frequent down-times.
- Single-purpose functional parts were wrapped into a web service.
  - Rendering “Buy” button.
  - Correct tax calculation on checkout.
- Functions **only** communicate with the rest of the world by means of their web service interface.

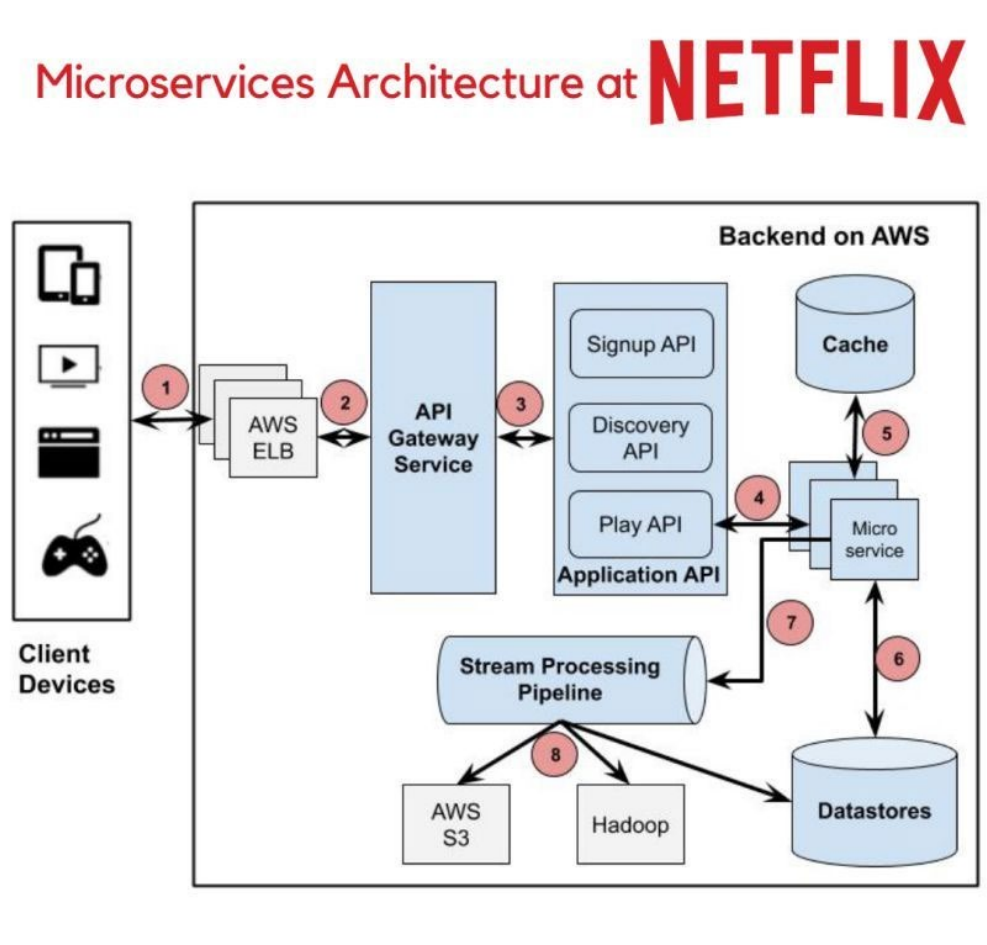
# MICROSERVICES

**Amazon**

- A team was assigned to each service.
- Goal: mostly independent services/teams adhering to standard rules and fully responsible for their service.
- Scaling much easier, probability of outages smaller.

Real-time graph of microservice dependencies at Amazon, 2008





Credits: Cao Duc Nguyen

1. Request arrives at **load balancer** (AWS ELB).
2. **API gateway** forwards and monitors API calls (typically terminates HTTPS).
3. **Application API** as core business logic API with different functionalities (signup, discovery, play).
4. **Play API** calls microservices to fulfill request.
5. Microservices execute in an isolated environment.
6. Microservices read/write from different data stores.
7. Microservices can produce events (e.g. user tracking) that are sent to the stream processing pipeline.
8. Data from stream processing pipeline can be persisted (AWS S3 / Hadoop).

# LECTURE OUTLINE

- Service Oriented Architecture (SOA)
  - Definition
  - REST
  - Web Services (SOAP, JSON-RPC, gRPC)
  - Microservices
- **Message-Oriented Middleware (MOM)**
  - Enterprise Bus (ESB)
  - Publish-subscribe
  - Queuing
  - Messaging

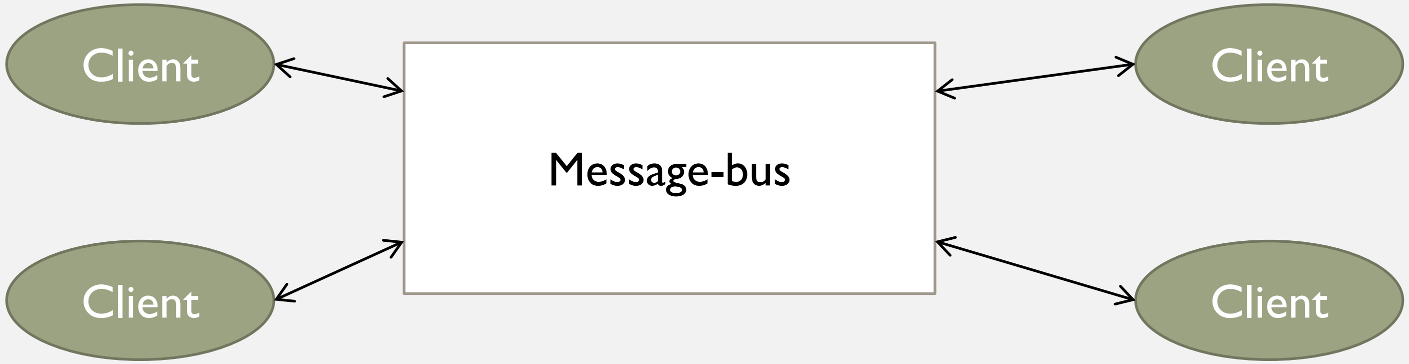
# MESSAGE-ORIENTED MIDDLEWARE

- Provides the “glue” for service architectures.
- Services interact with different formats, protocols, etc.
- Instead of “direct” communication, services stay independent of each other
  - **Loose** coupling.
  - Services don‘t need to know each other.
- Paradigms:
  - Enterprise Service Bus (ESB).
  - Publish-Subscribe.
  - Queuing and messaging Systems.

# MESSAGE-ORIENTED MIDDLEWARE

**Enterprise Service Bus (ESB)**

- Central bus or distributed brokers receive/send messages to clients.
- Client sends a message to the bus with metadata, so message can be delivered.
- Examples: WebSphereMQ.



# MESSAGE-ORIENTED MIDDLEWARE

**Publish-Subscribe Model**

- Publisher labels messages by "topic".
- Subscriber listens to messages with a certain topic.
- Middleware sends message to subscribers.
- Also message filtering possible by SQL-like syntax.
- Notification or event-based programming models .

# MESSAGE-ORIENTED MIDDLEWARE

**Queuing and Messaging**

- Several competing standards.
- Java Message Service (JMS).
- Advanced Message Queuing Protocol (AMQP).
  - E.g. RabbitMQ.
- Cloud queuing.
  - Amazon Simple Queue Service (SQS).
  - Azure Queue.