EN
English

Universidad de Oviedo

School of Computer Science

Distributed and big data systems

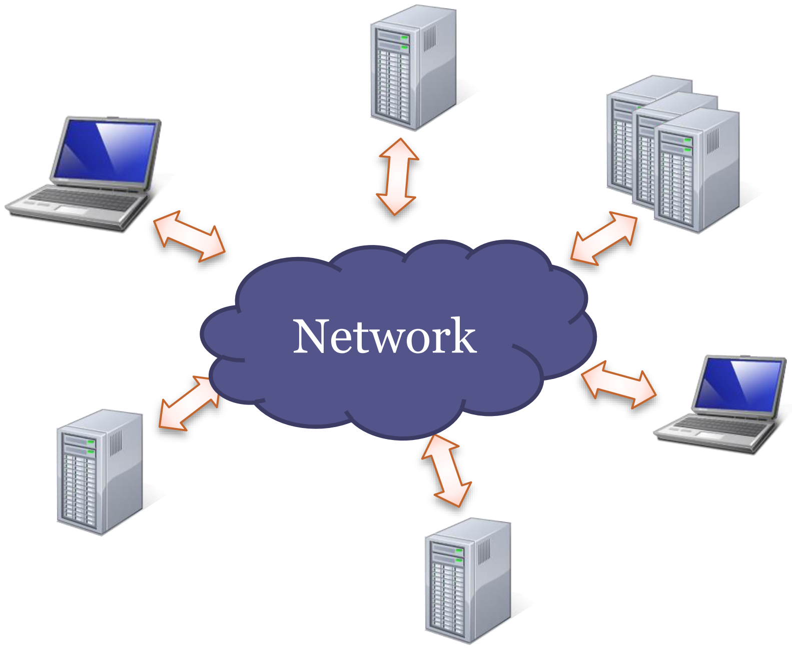
SOFTWARE
ARCHITECTURE

Course 2022/23

Jose E. Labra Gayo

Distributed systems

Integration styles
Topologies: Hub % Spoke, Bus
Broker pattern
Peer-to-peer
SOA
Microservices
Serverless



Integration styles

File transfer

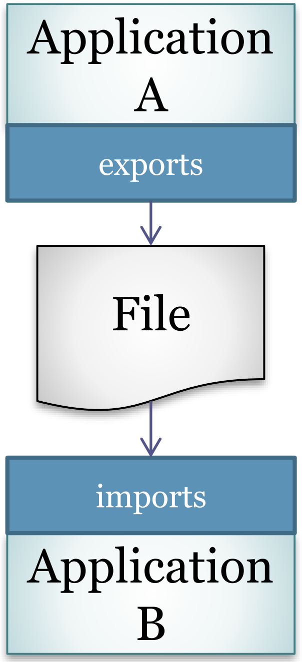
Shared database

Remote procedure call

Messaging

File transfer

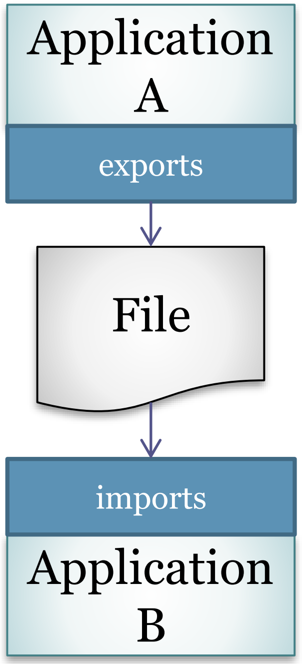
- An application generates a data file that is consumed by another
  - One of the most common solutions
- Advantages
  - Independence between A and B
  - Low coupling
  - Easier debugging
    - By checking intermediate files



File transfer

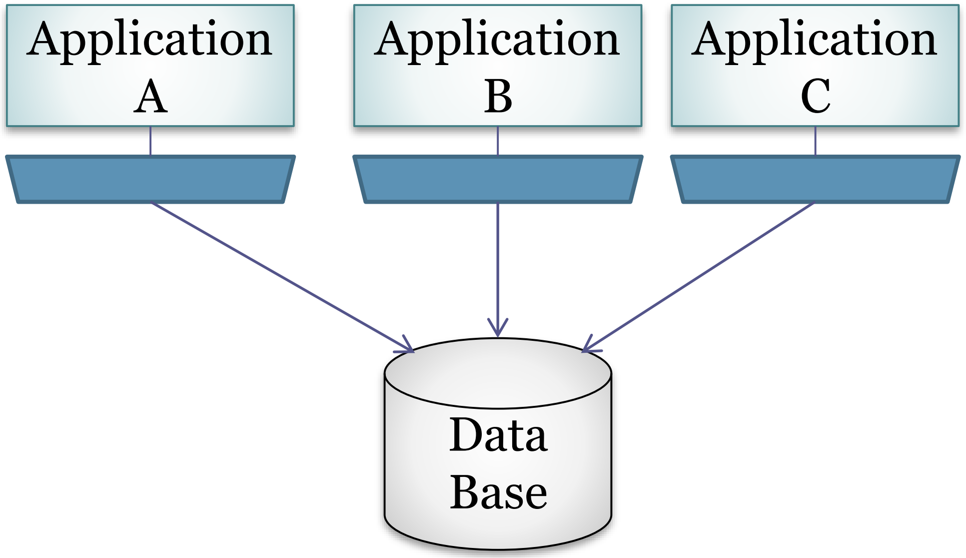
Challenges

- Both applications must agree a common file format
  - It can increase coupling
- Coordination
  - Once the file has been sent, the receiver could modify it ⇒ 2 files!
- It may require manual adjustments



Shared database

- Applications store their data in a shared database
- Advantage
  - Data are always available
  - Everyone has access to the same information
  - Consistency
  - Familiar format
    - SQL for everything



Shared database

Challenges

- Database schema can evolve
  - It requires a common schema for all applications
    - That can cause problems/conflicts
  - External packages are needed (common database)
- Performance and scalability
  - Database as a bottleneck
- Synchronization
  - Distributed databases can be problematic
  - Scalability
  - NoSQL ?

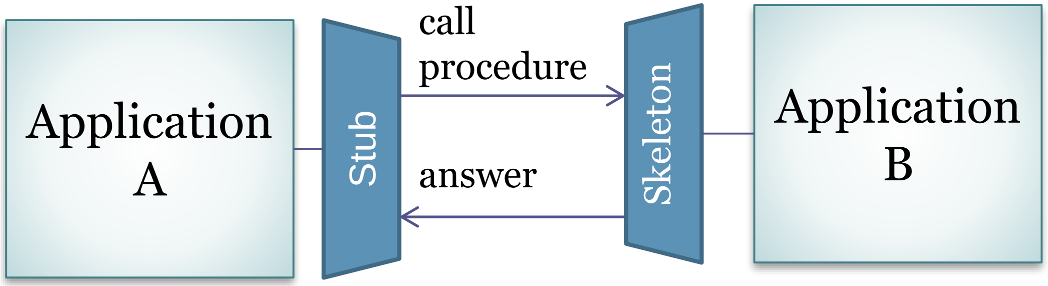
Shared database

Variants

- *Data warehousing*: Database used for data analysis and reports
- ETL: process based on 3 stages
  - Extraction: Get data from heterogeneous sources
  - Transform: Process data
  - Load: Store data in a shared database

Remote Procedure Call (RPC)

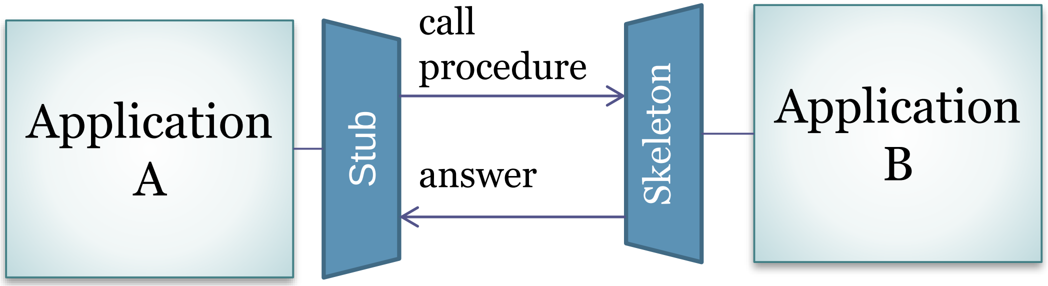
- An application calls a function from another application that could be in another machine
  - Invocation can pass parameters
  - Obtains an answer
- Lots of applications
  - RPC, RMI, CORBA, .Net Remoting, ...
  - Web services, ...



Remote Procedure Call (RPC)

Advantages

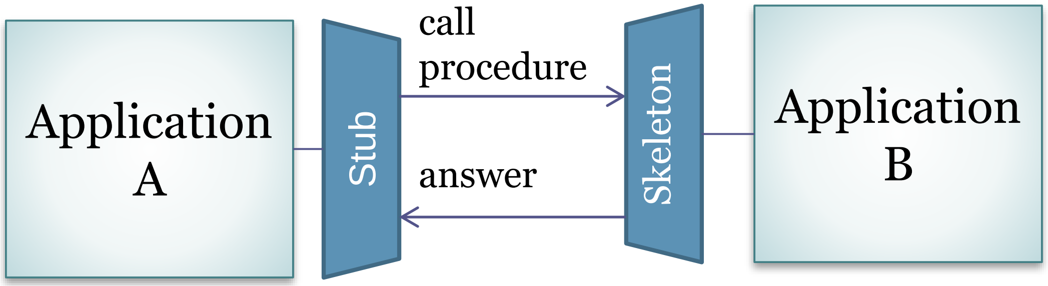
- Encapsulation of implementation
- Multiple interfaces for the same information
  - Different representations can be offered
- Model familiar for developers
  - It is similar to invoke a method



Remote Procedure Call (RPC)

Challenges

- False sense of simplicity
  - Remote procedure ≠ procedure
    - 8 fallacies of distributed computing
- Synchronous procedure calls
  - Increase application coupling



The network is reliable
Latency is zero
Bandwidth is infinite
The network is secure
Topology doesn't change
There is one administrator
Transport cost is zero
The network is homogeneous

8 fallacies of distributed computing
http://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

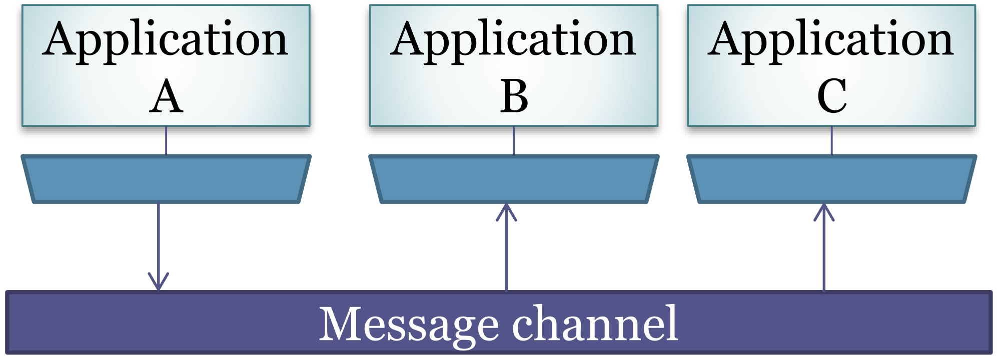
https://www.youtube.com/watch?v=UZxLYv5RFyI&t=54s

Remote procedure call

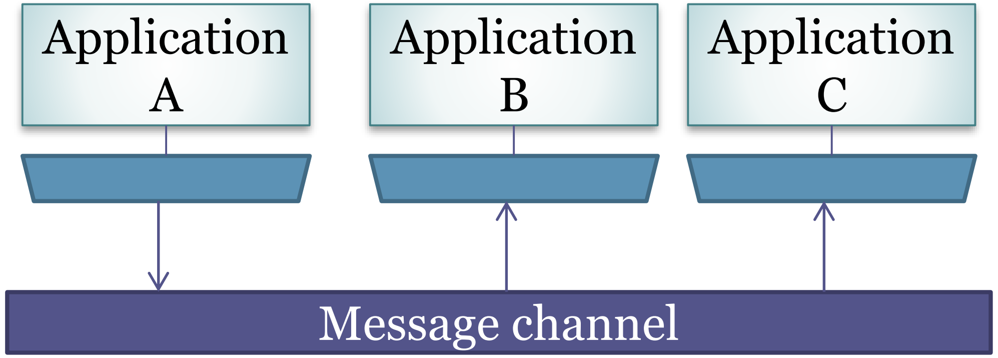
- New proposals: gRPC (https://grpc.io/)
- Google proposal
- High performance RPC framework
  - http/2 transport protocol

Messaging

- Multiple independent applications communicate sending messages through a channel
- Asynchronous communication
  - Applications send messages a continue their execution



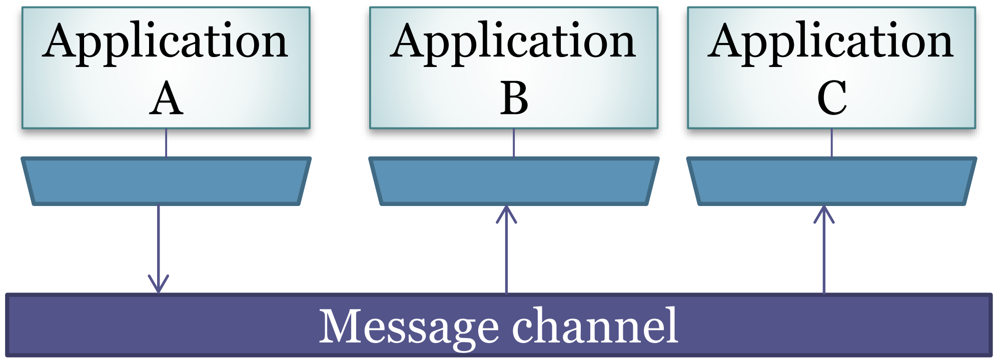
Messaging



Advantages

- Low coupling
  - Applications are independent between each other
- Asynchronous communication
  - Applications continue their execution
- Implementation encapsulation
  - The only thing exposed is the type of messages

Messaging



Challenges

- Implementation complexity
  - Asynchronous communication
- Data transfer
  - Adapt message formats
- Different topologies

Integration topologies

Hub & Spoke

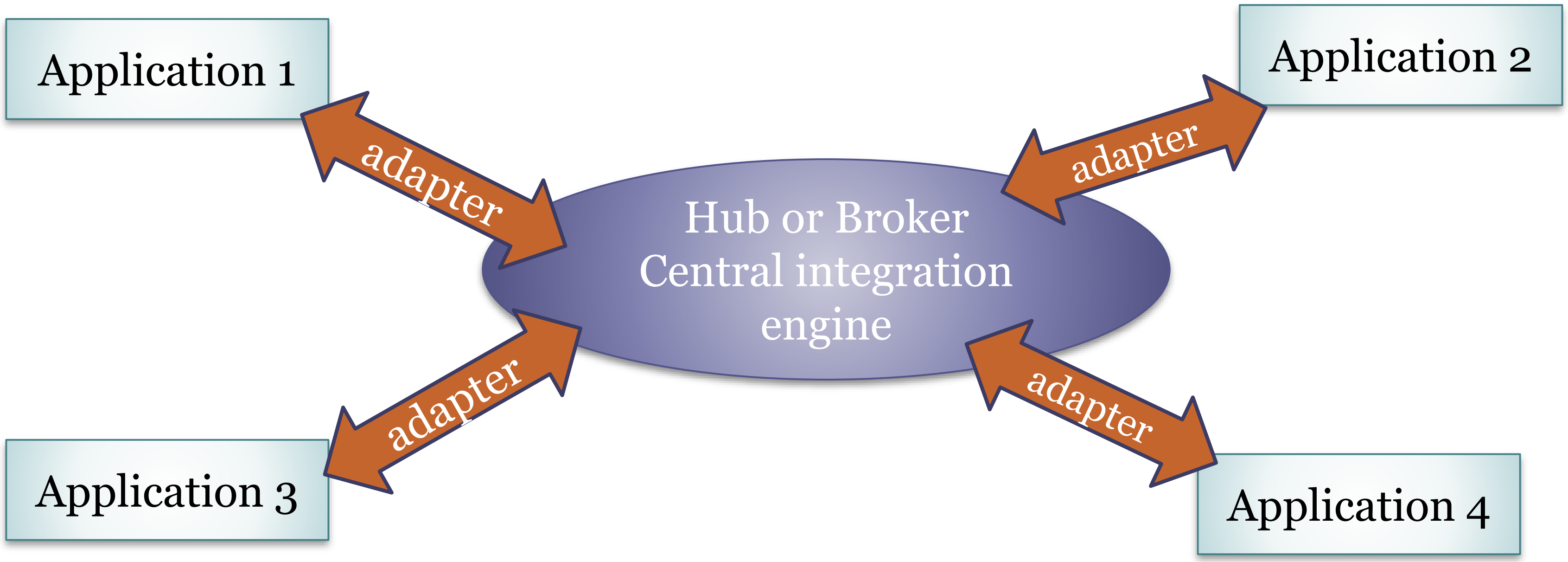
Bus

Hub & Spoke

Related with Broker pattern

Hub = Centralized message Broker

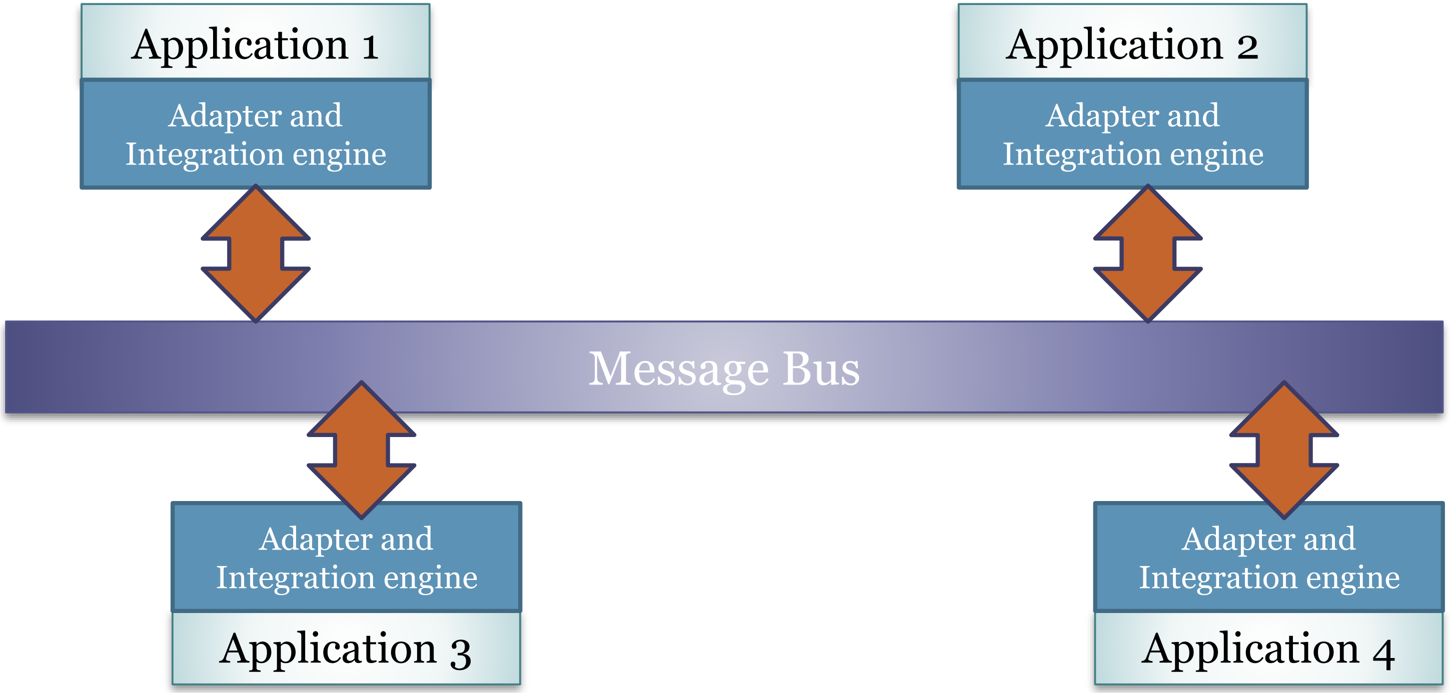
- It is in charge of integration



Bus

Each application contains its own integration machine

Publish/Subscribe style

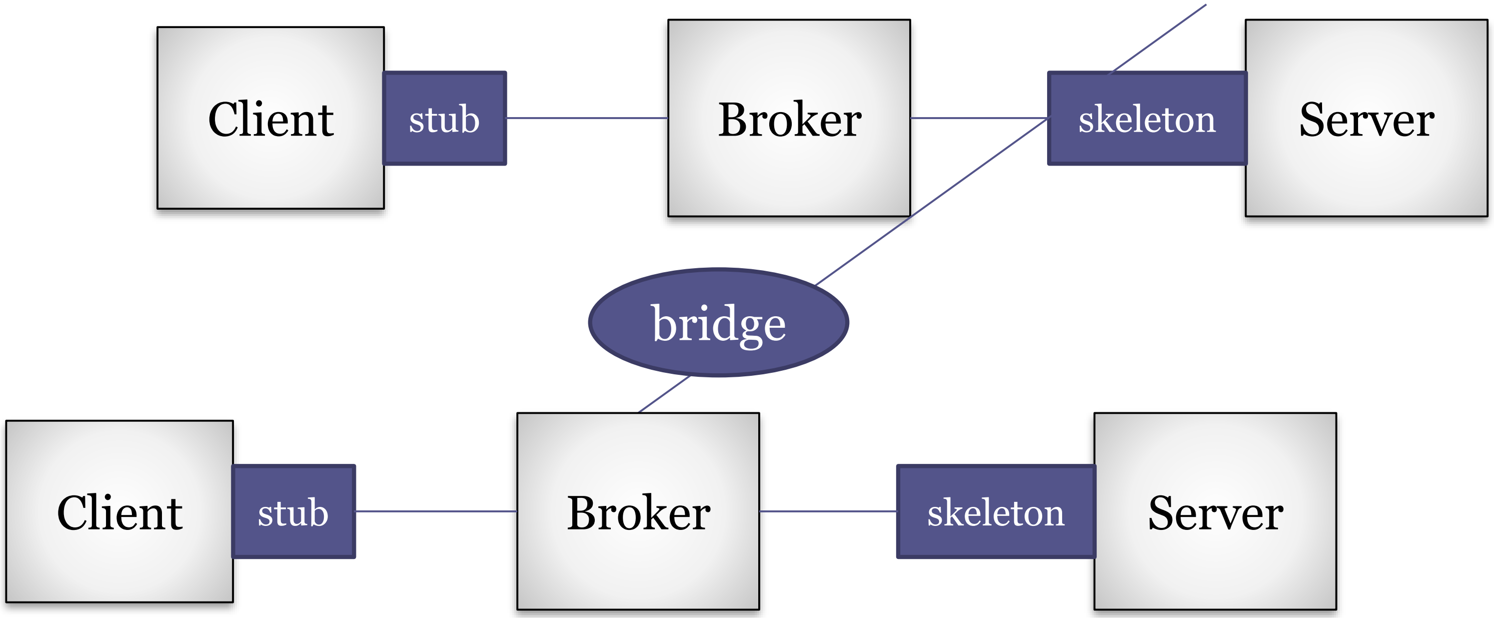


Bus

- ESB - Enterprise Service Bus
- Defines the messaging backbone
  - Some tasks
    - Protocol conversion
    - Data transformation
    - Routing
- Offers an API to develop services
  - MOM (Message Oriented Middleware)

Broker

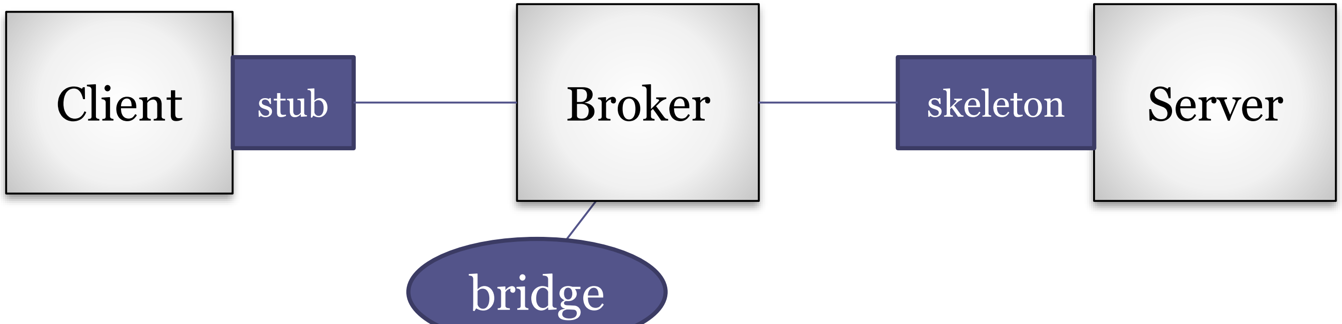
Intermediate node that manages communication between a client and a server



Broker

Elements

- Broker
  - Manages communication
- Client: Sends requests
  - Client Proxy: *stub*
- Server: Returns answers
  - Server Proxy: *skeleton*
- Bridge: Can connect brokers



Broker

Advantages

- Separation of concerns
  - Delegates low level communication aspects to the broker
  - Separate maintenance
  - Reusability
- Servers are independent from clients
- Portability
  - Broker = low level aspects
- Interoperability
  - Using *bridges*

Challenges

- Performance
  - Adds an indirection layer
- Can increase coupling between components
- Broker = single point of failure

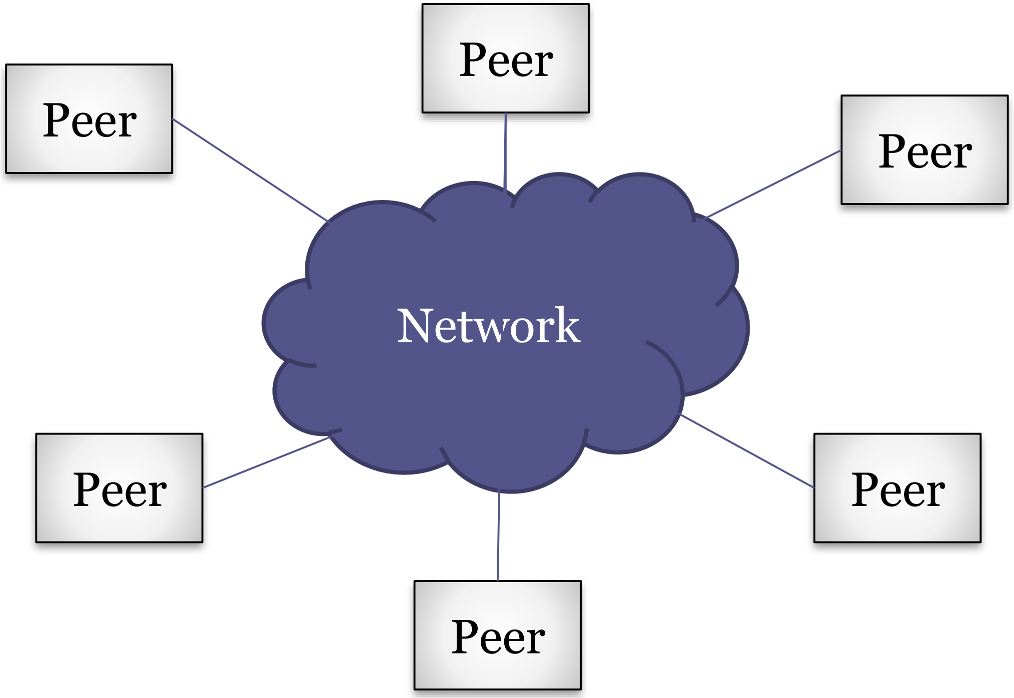
Broker

Applications

- CORBA and distributed systems
- Android uses a variation of Broker pattern

Peer-to-Peer

Equal and autonomous nodes (*peers*) that communicate between them.



Peer-to-Peer

- Elements
  - Computational nodes: *peers*
    - They contain their own state and control thread
  - Network protocol
- Constraints
  - There is no main node
  - All peers are equal

Peer-to-Peer

Advantages

- Decentralized information and control
- Fault tolerance
  - There is no single point of failure
  - A failure in one peer does not compromise the whole system

Challenges

- Keeping the state of the system
  - Complexity of the protocol
- Bandwidth Limitations
  - Network and protocol latency
- Security
  - Detect malicious *peers*

Peer-to-Peer

- Popular applications
  - Napster, BitTorrent, Gnutella, ...
  - This architecture style is not only to share files
    - e-Commerce (B2B)
    - Collaborative systems
    - Sensor networks
    - Blockchain
    - ...
- Variants
  - Super-peers

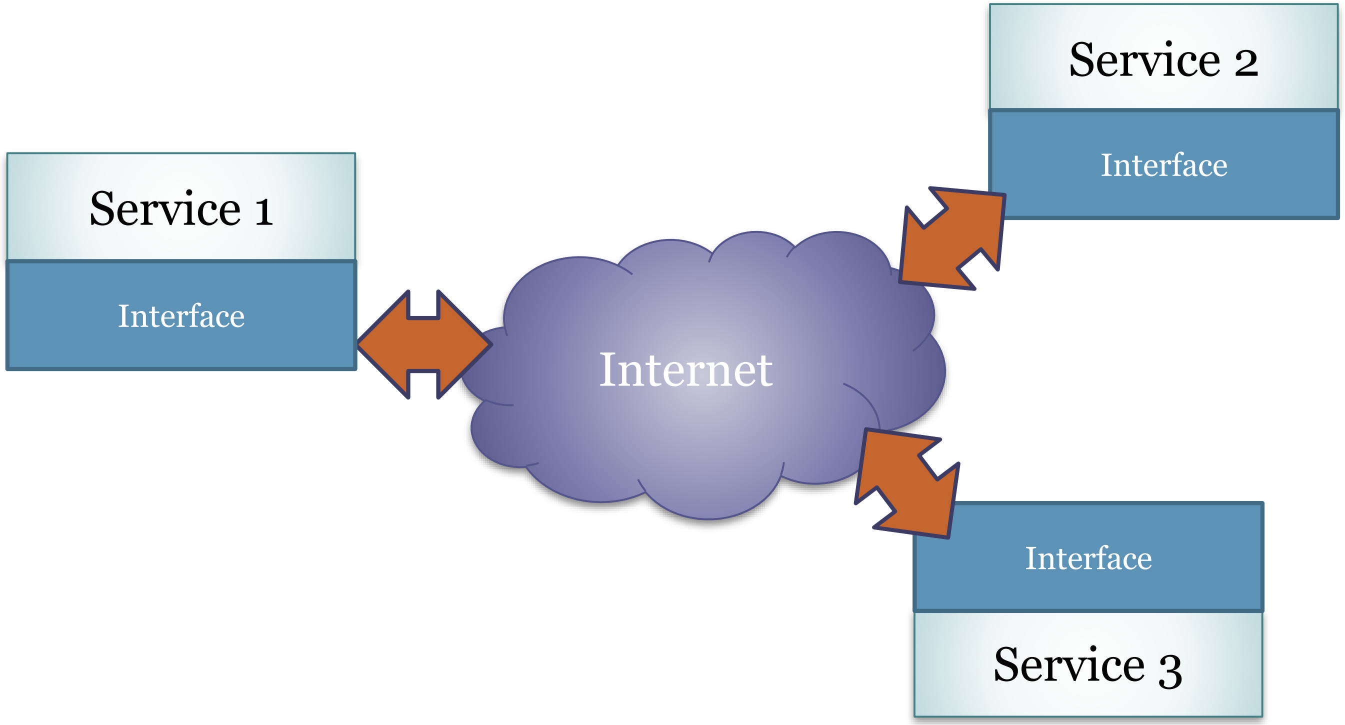
Service Oriented Architectures

SOA
WS-*
REST

SOA

SOA = Service Oriented Architecture

Services are defined by an interface



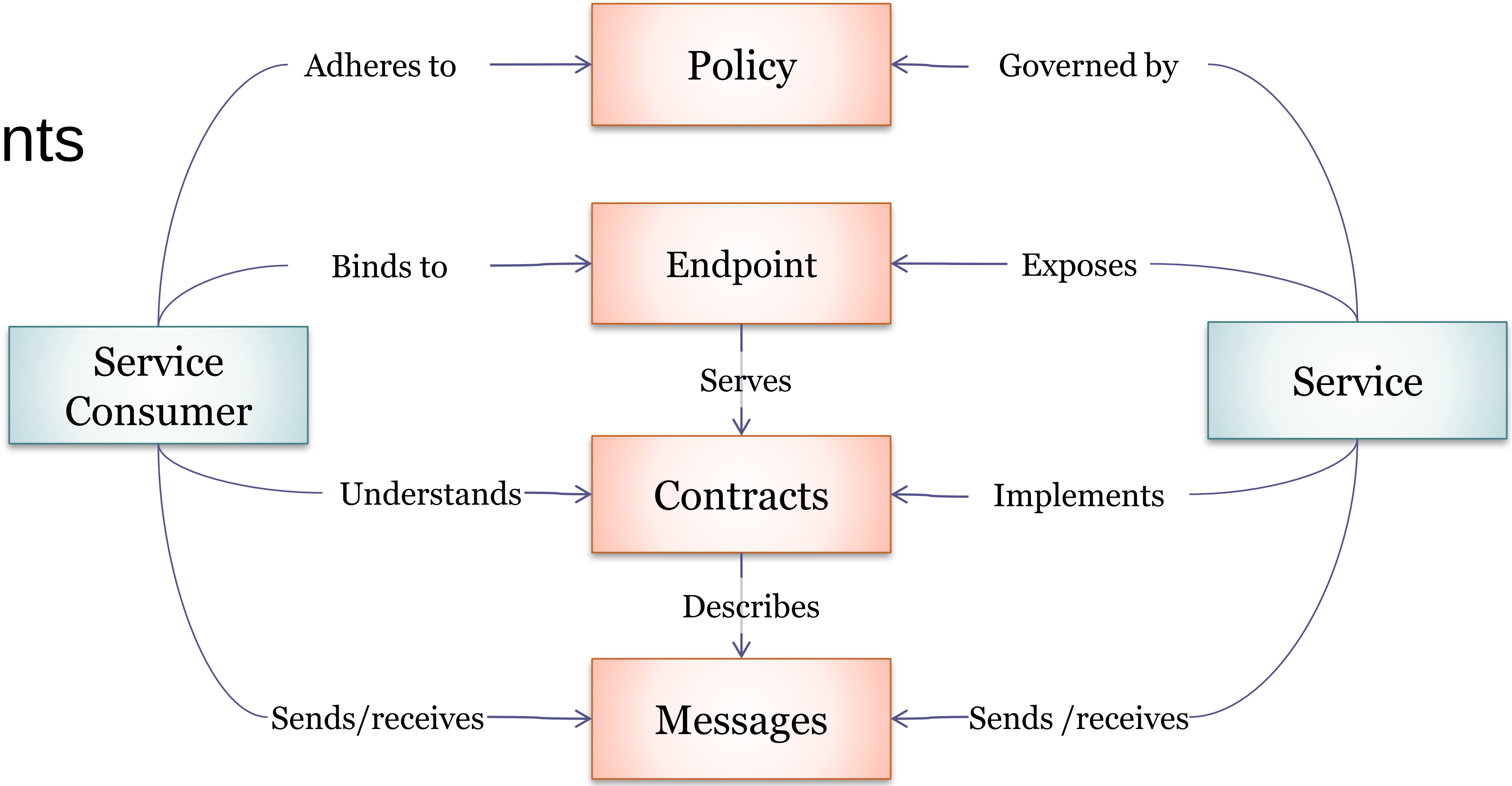
SOA

Elements

- Provider: Provides service
- Consumer: Does requests to the service
- Messages: Exchanged information
- Contract: Description of the functionality provided by the service
- Endpoint: Service location
- Policy: Service level agreements
  - Security, performance, etc.

SOA

Constraints



Fuente: SOA Patterns, A. Rottem Gal Oz

SOA

Advantages

- Independent of language and platform
- Interoperability
  - Use of standards
- Low coupling
- Decentralized
- Reusability
- Scalability
  - one-to-many vs one-to-one
- Partial solution for legacy systems
  - Adding a web services layer

Challenges

- Performance
  - E.g. real time systems
- Overkill in very homogeneous environments
- Security
  - Risk of public exhibition of API to external parties
  - DoS attacks
- Service composition and coordination

SOA

Variants:

- WS-*
- REST

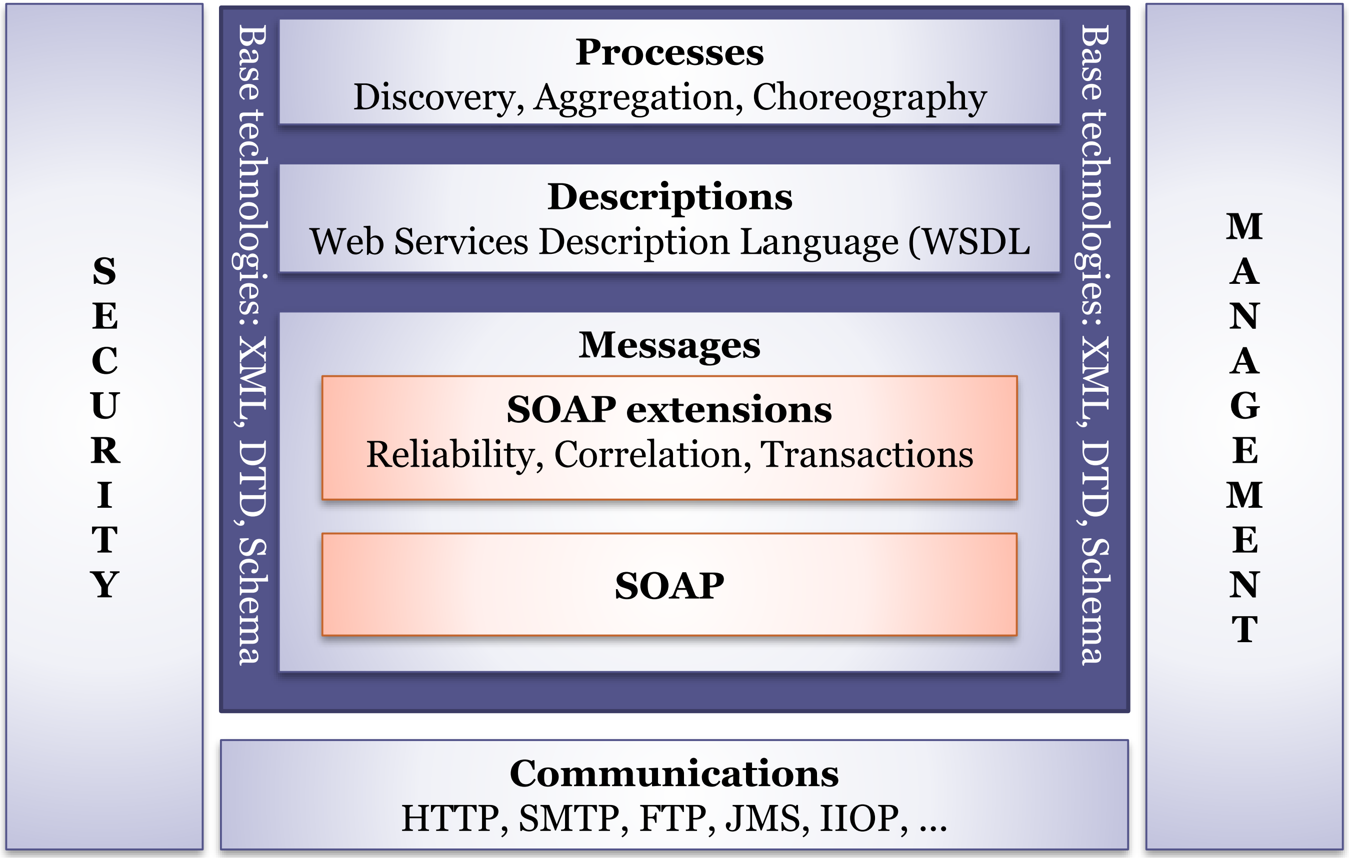
WS-*

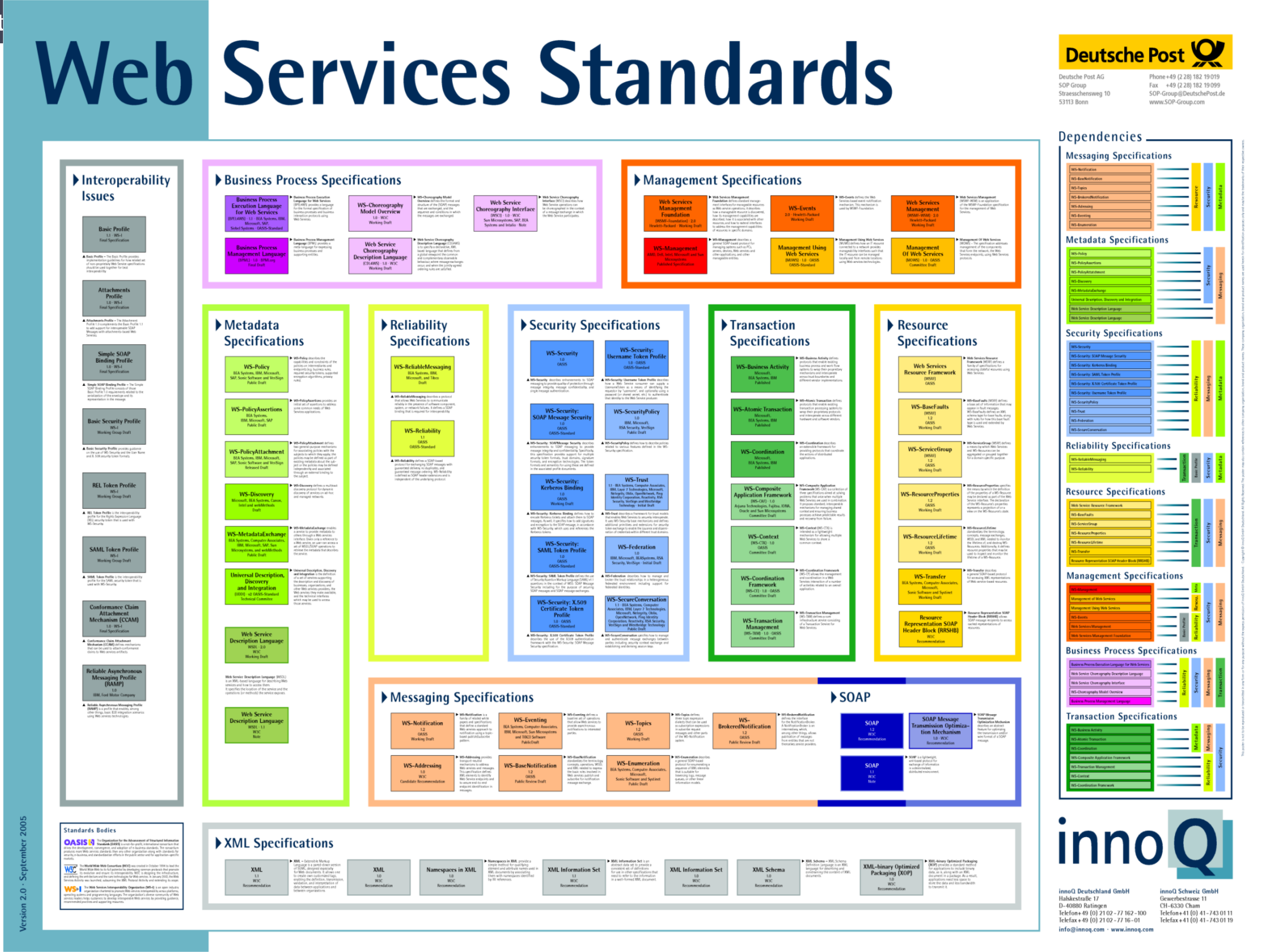
- WS-* model = Set of specifications
  - SOAP, WSDL, UDDI, etc....
  - Proposed by W3C, OASIS, WS-I, etc.
  - Goal: Reference SOA implementation

University of Oviedo
School of Computer Science

WS-*

Web Services Architecture

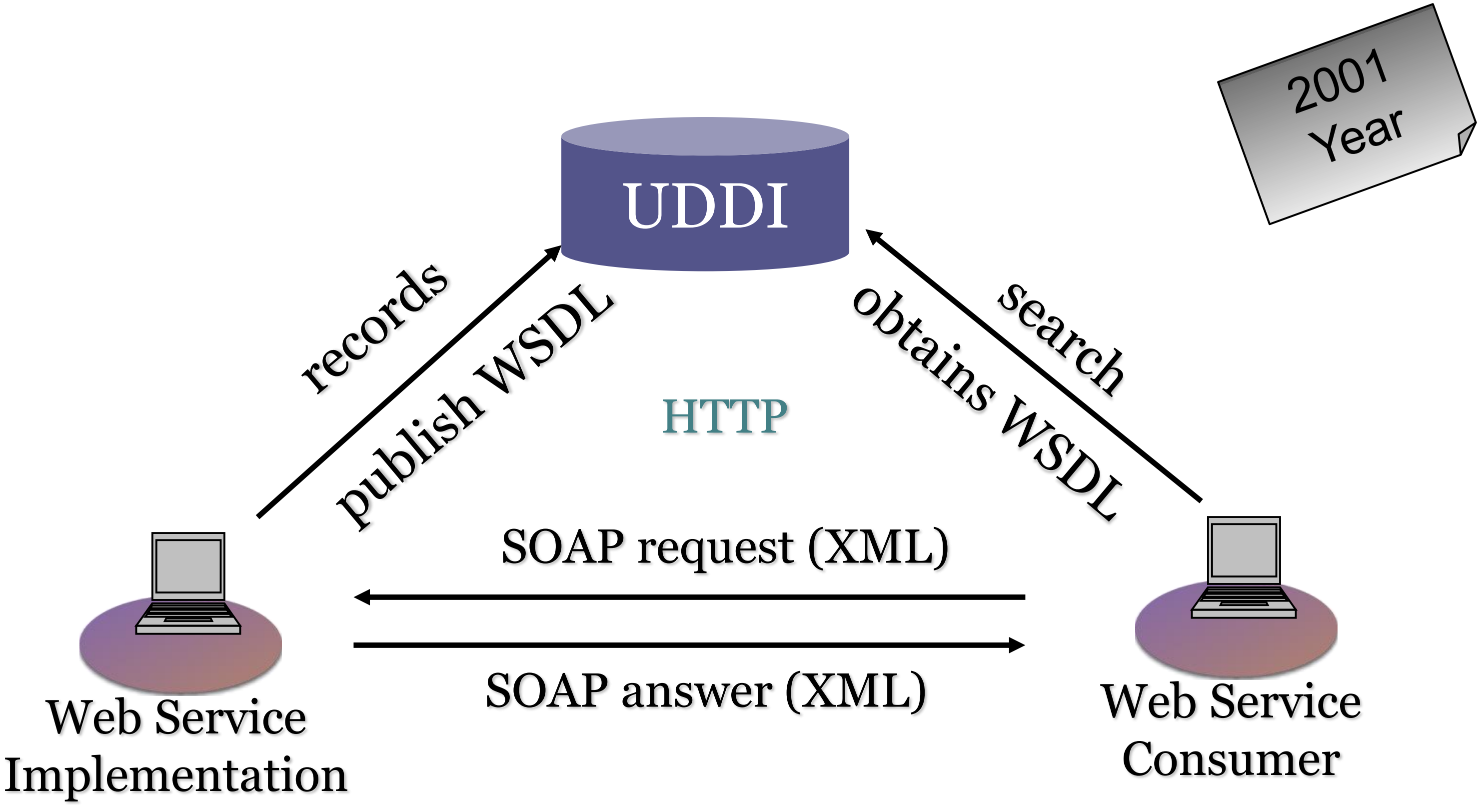




University of Oviedo

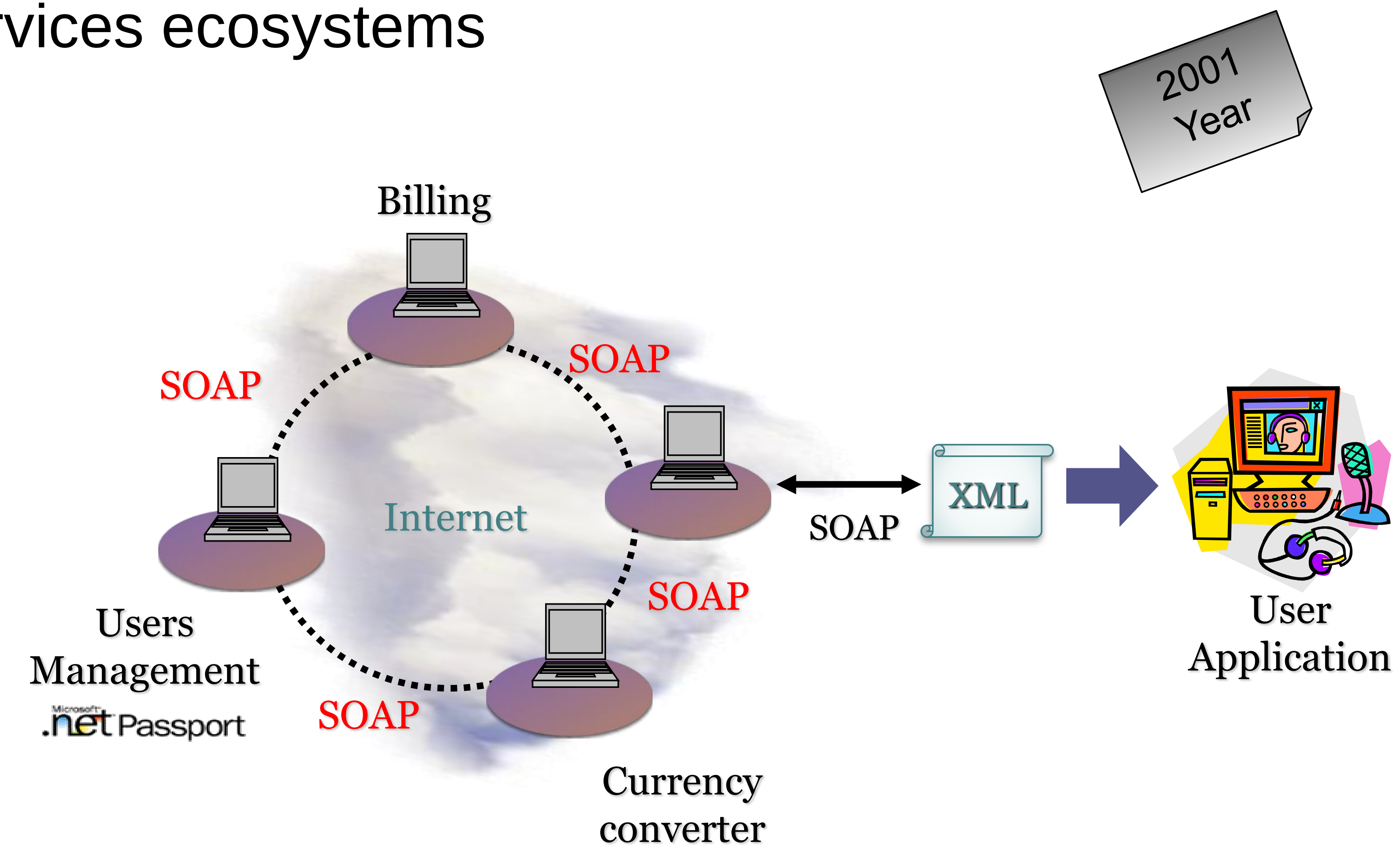
School of Computer Science

WS-*


2001
Year
UDDI
records
publish WSDL
HTTP
search
obtains WSDL
SOAP request (XML)
SOAP answer (XML)
Web Service
Implementation
Web Service
Consumer

WS-*

Web Services ecosystems



WS-*

SOAP

Defines messages format and bindings with several protocols

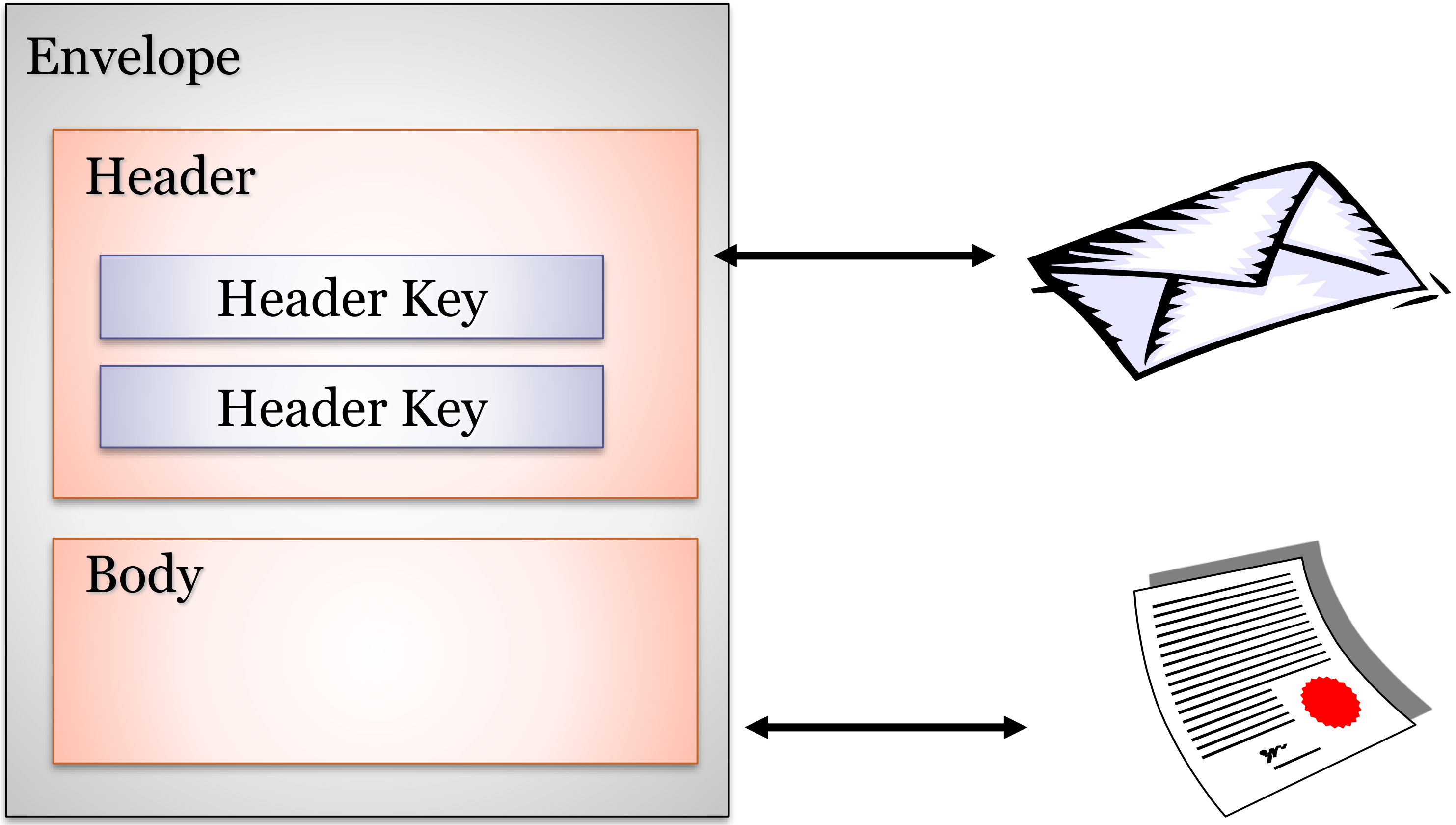
- Initially *Simple Object Access Protocol*

Evolution

- Developed from XML-RPC
- SOAP 1.0 (1999), 1.1 (2000), 1.2 (2007)
- Initial development by Microsoft
- Posterior adoption by IBM, Sun, etc.
- Good Industrial adoption

WS-*

Message format in SOAP



WS-*

Example of SOAP over HTTP

2001
Year

POST ?

```
POST /Suma/Service1.asmx HTTP/1.1
Host: localhost
Content-Type: text/xml; charset=utf-8
Content-Length: longitod del mensaje
SOAPAction: "http://tempuri.org/suma"
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope
     xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
  <sum xmlns="http://tempuri.org/">
     <a>3</a>
     <b>2</b>
  </sum>
 </soap:Body>
</soap:Envelope>
```

WS-*

Advantages

- Specifications developed by community
  - W3c, OASIS, etc.
- Industrial adoption
  - Implementations
- Integral view of web services
- Numerous extensions
  - Security, orchestration, choreography, etc.

Challenges

- Not all specifications were mature
  - Over-specification
  - Lack of implementations
- RPC style abuse
  - Uniform interface
  - Sometimes, bad use of HTTP architecture
    - Overload of GET/POST methods

WS-*

Applications

- Lots of applications have been using SOAP
  - Example: eBay (50mill. SOAP transactions/day)
- But…some popular web services ceased to offer SOAP support
  - Examples: Amazon, Google, etc.

REST

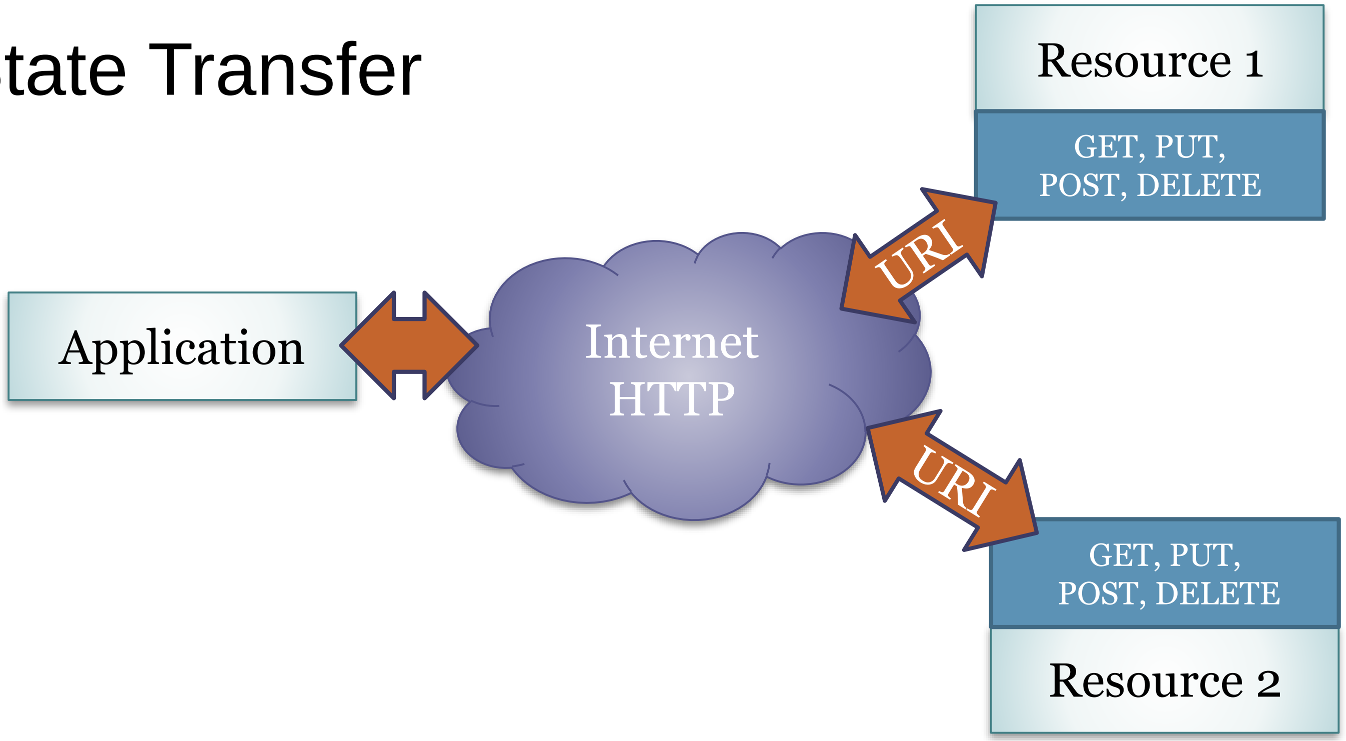
REST = REpresentational State Transfer

- Architectural style
- Source: Roy T Fielding PhD dissertation (2000)
- Inspired by Web architecture (HTTP/1.1)

REST

REST - Representational State Transfer

Diagram



REST

- Set of constraints
  - Resources with uniform interface
    - Identified by URIs
    - Fixed set of actions: GET, PUT, POST, DELETE
  - Resource representations are returned
  - Stateless
- REST = Architectural style
  - Some levels of adoption:
    - RESTful
    - REST-RPC hybrid

REST as a composed style

- Layers
- Client-Server
  - Stateless
  - Cached
  - Replicated server
- Uniform interface
  - Resource identifiers (URIs)
  - Auto-descriptive messages (MIME types)
  - Links to other resources (HATEOAS)
- Code on demand (optional)

REST uniform interface

Fixed set of operations

GET, PUT, POST, DELETE

Method	In databases	Function	Safe?	Idempotent?
PUT	≈Create/Update	Create/update	No	Yes
POST	≈Update	Create/ Update children	No	No
GET	Retrieve	Query resource info	Yes	Yes
DELETE	Delete	Delete resource	No	Yes

Safe = Does not modify server data
Idempotent = The effect of executing N-times is the same as executing it once

REST

- Stateless client/server protocol
- State handled by client
  - HATEOAS *(Hypermedia As The Engine of Application State)*
    - Representations return URIs to available options
    - Chaining of resource requests

Example: Student management

1.- Get list of students

```
GET http://example.org/student
```

Returns list of students with each student URI

2.- Get information about an specific student

```
GET http://example.org/student/id2324
```

3.- Update information of an specific student

```
PUT http://example.org/student/id2324
```

REST

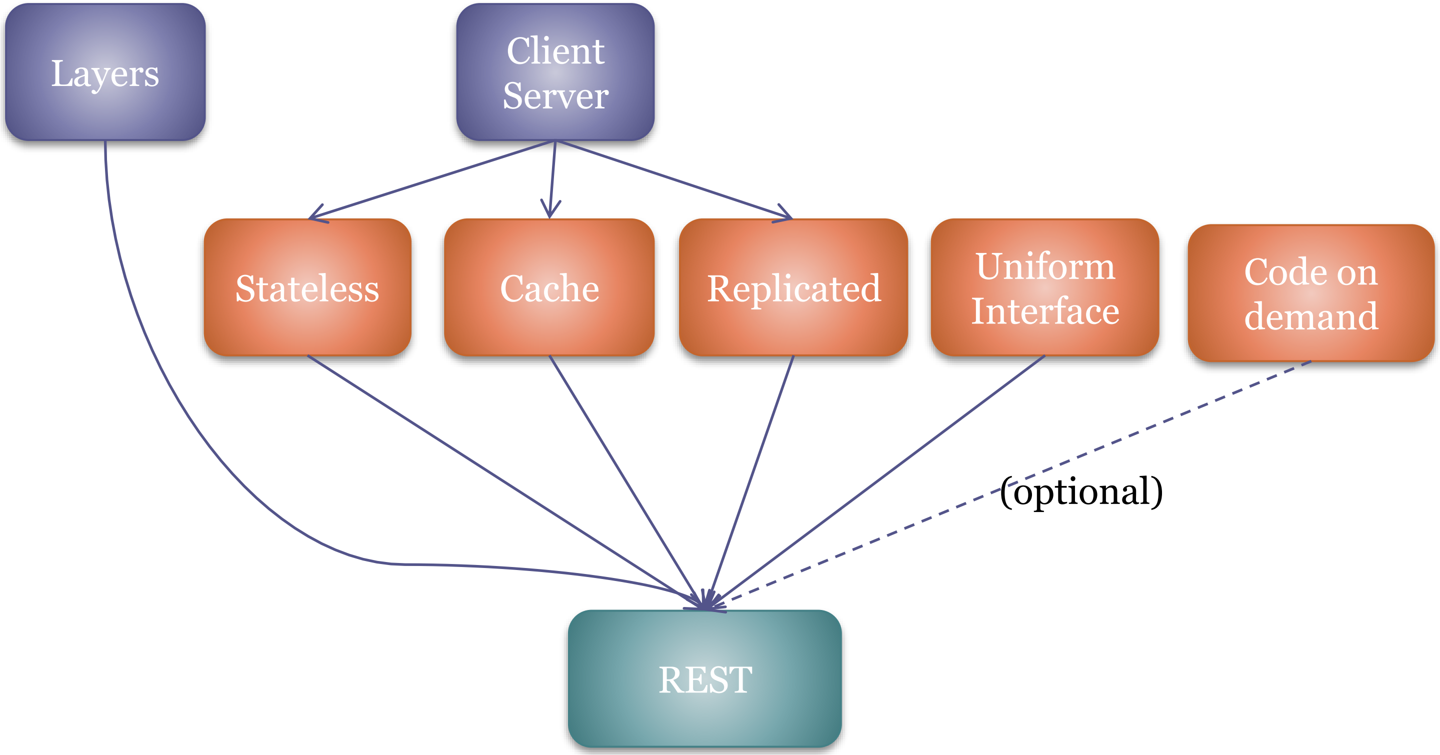
Advantages

- Client/Server
  - Separation of concerns
  - Low coupling
- Uniform interface
  - Facilitates comprehension
  - Independent development
- Scalability
- Improves answer times
- Less network load (cached)
  - Less bandwidth

Challenges

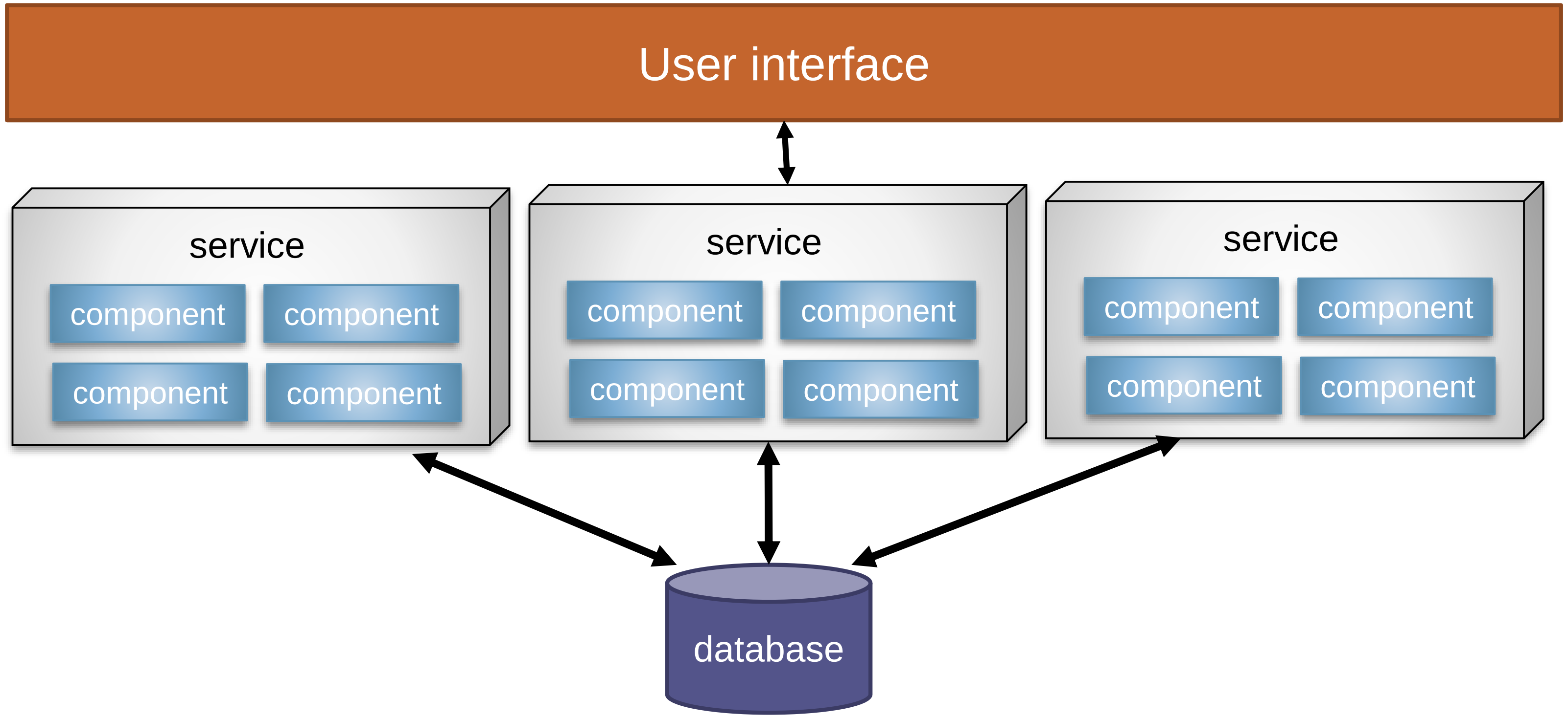
- REST partially adopted
  - Just using JSON or XML
    - Web services without contract or description
  - RPC style REST
- Difficult to incorporate other requirements
  - Security, transaction, composition, etc.

REST as a composed style



Service based architecture

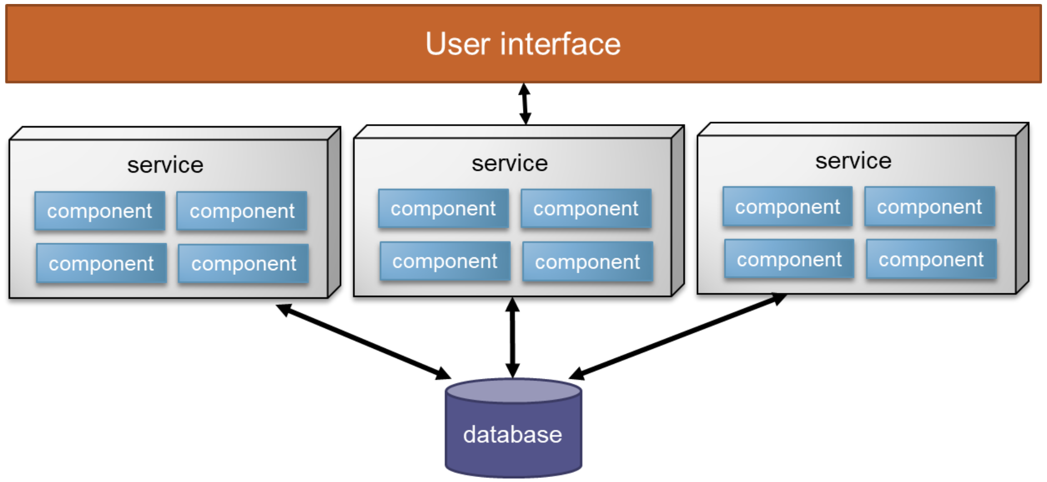
Pragmatic architectural style based on SOA



Service based architecture

Elements

- Services = independently deployed units
  - Usually composed of different components
- User interface accesses services remotely (Internet)
- Database shared by those services

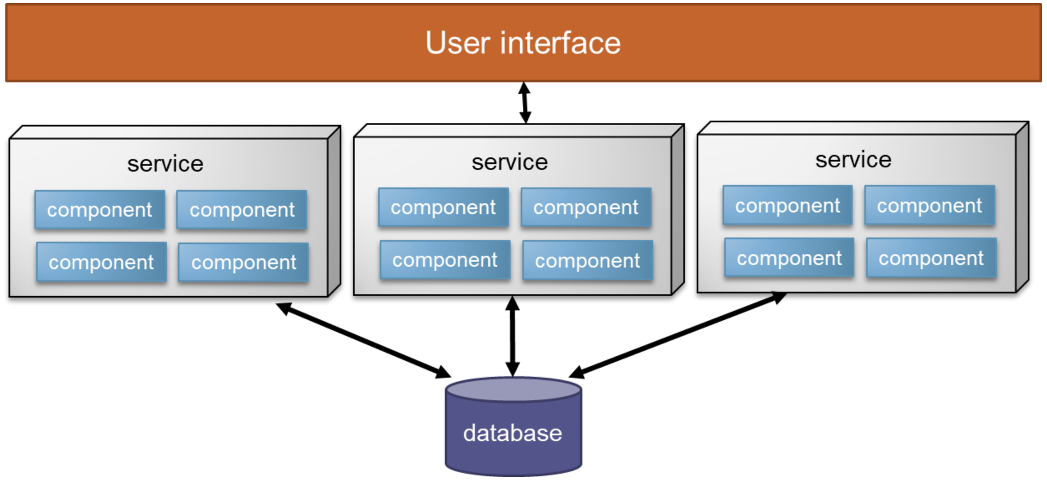


University of Oviedo

Service based architecture

Constraints

- Each service is independently deployed
- Services are usually coarse grained
- User interface can be divided (different topologies)
- Database is usually shared by each service



School of Computer Science

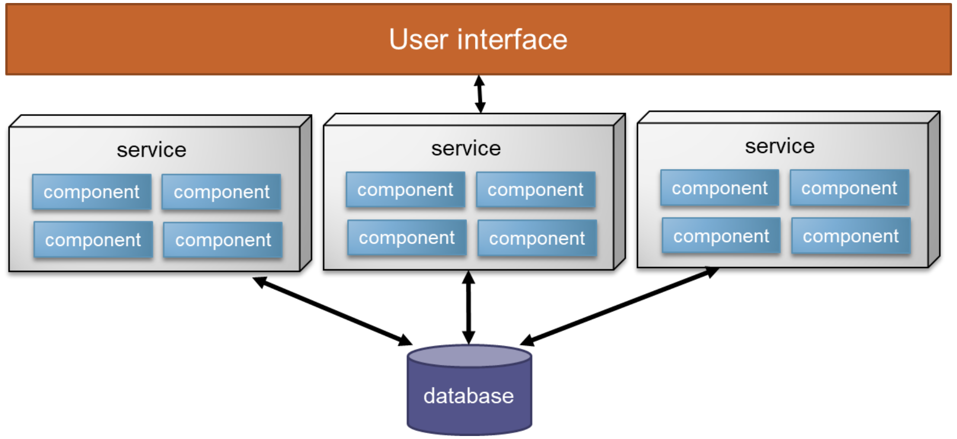
Service based architecture

Advantages

- Modularity of development
  - Services can be independently developed
- Technology diversity
  - Each service can be developed using a different programming language & technology
- Time to market
  - Several frameworks
- Availability
- Reliability

Challenges

- Scalability (database partitioning)
- Evolution of services
  - Adaption to change is usually difficult
  - Services can be monoliths
- Conway's law
  - Database team
  - User interface team
  - Programmers



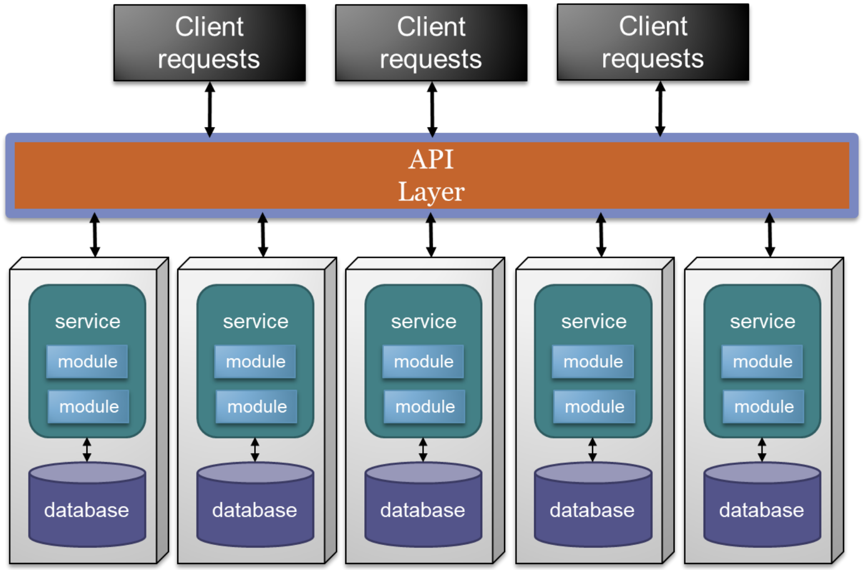
Microservices

Applications decomposed in microservices

Microservice = small, autonomous services that work together

Each microservice = independent building and deployment block

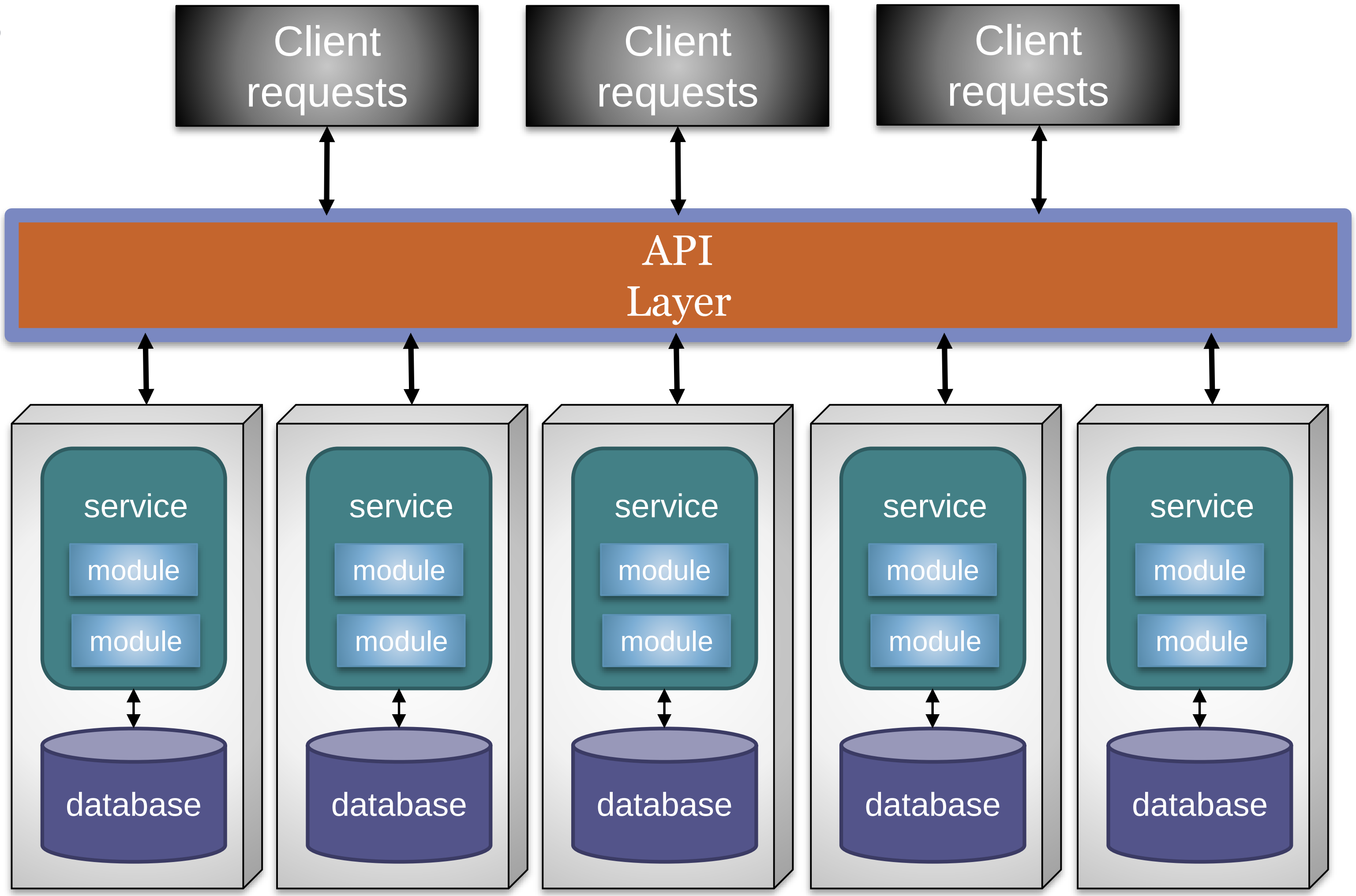
- Highly uncoupled
- Focus on a specific task
- Manage their own data



http://martinfowler.com/articles/microservices.html

Microservices

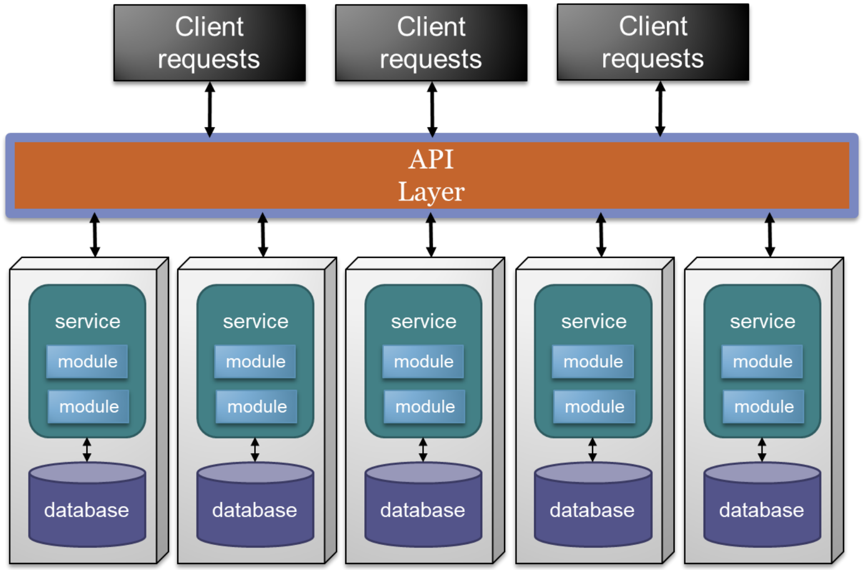
Diagram



Microservices

Elements

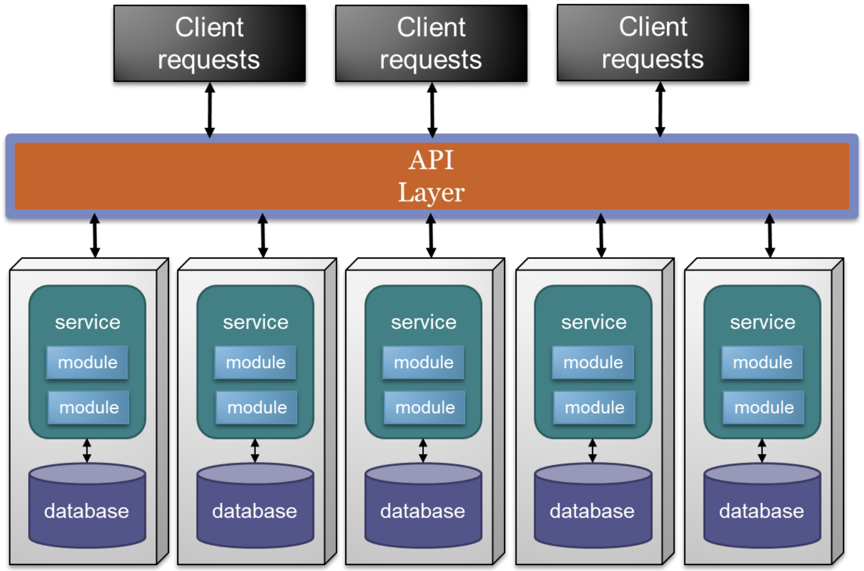
- A service + database form a deployed component
- A service contains several modules and its own database
- API layer (optional) offers a proxy or naming service



Microservices

Constraints

- Distributed
- Bounded context:
  - Each service models a domain or workflow
- Data isolation
- Independency:
  - No mediator or orchestrator



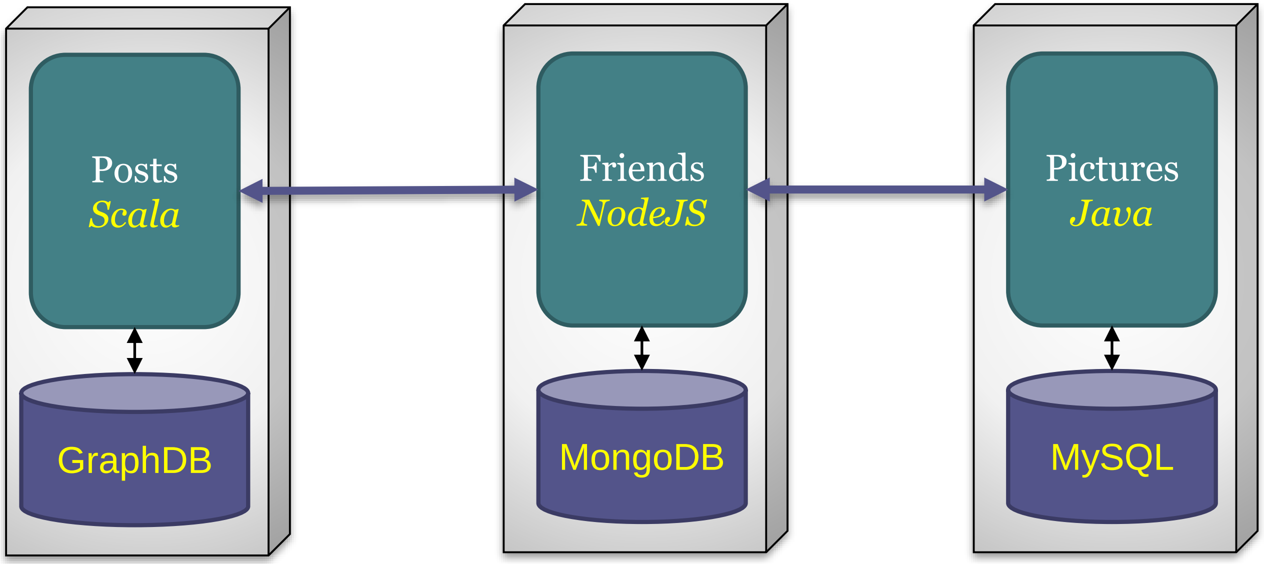
Features/advantages

- Technology heterogeneity
- Resilience
- Scalability
- Deployability
- Organizational alignment
- Decentralized data management
- Optimizing for replaceability

Microservices and heterogeneity

Each microservice can be implemented in its own programming language and technology stack

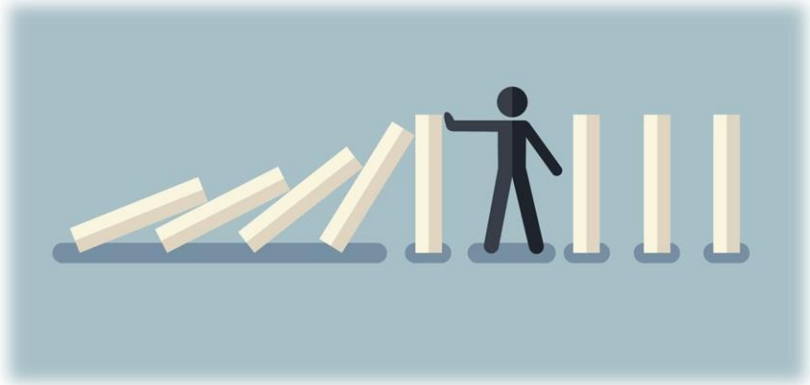
- Facilitates experimentation with new technologies
- Flexibility



Resilience

If a component of a system fails and the failure doesn't scale, the system can carry on working

In a monolithic system if a component fails, the whole system stops working



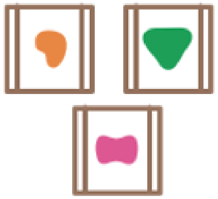
Scalability

- It is possible to scale on demand specific services
  - Monolithic systems require to scale the whole system
  - Not all components have the same needs
    - Microservices can be replicated as needed

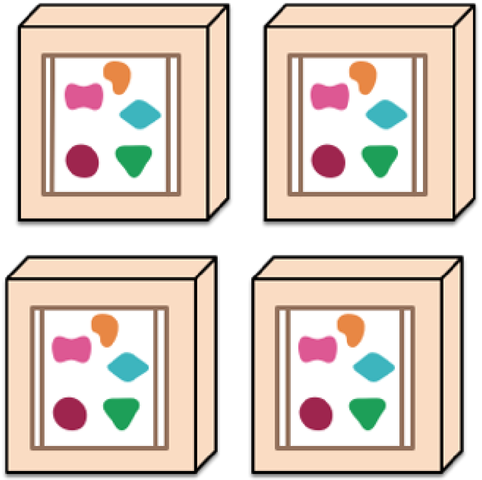
Scalability

Monolithic: all functionality in a single process

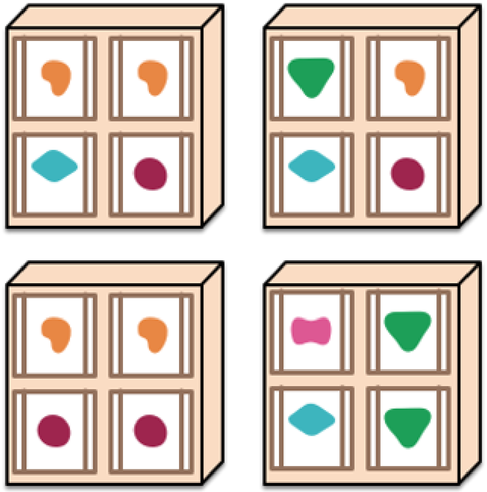
Microservices: each element of functionality into a separate service



...scales replicating the monolith on multiple services



... scales distributing these services and replicating as needed



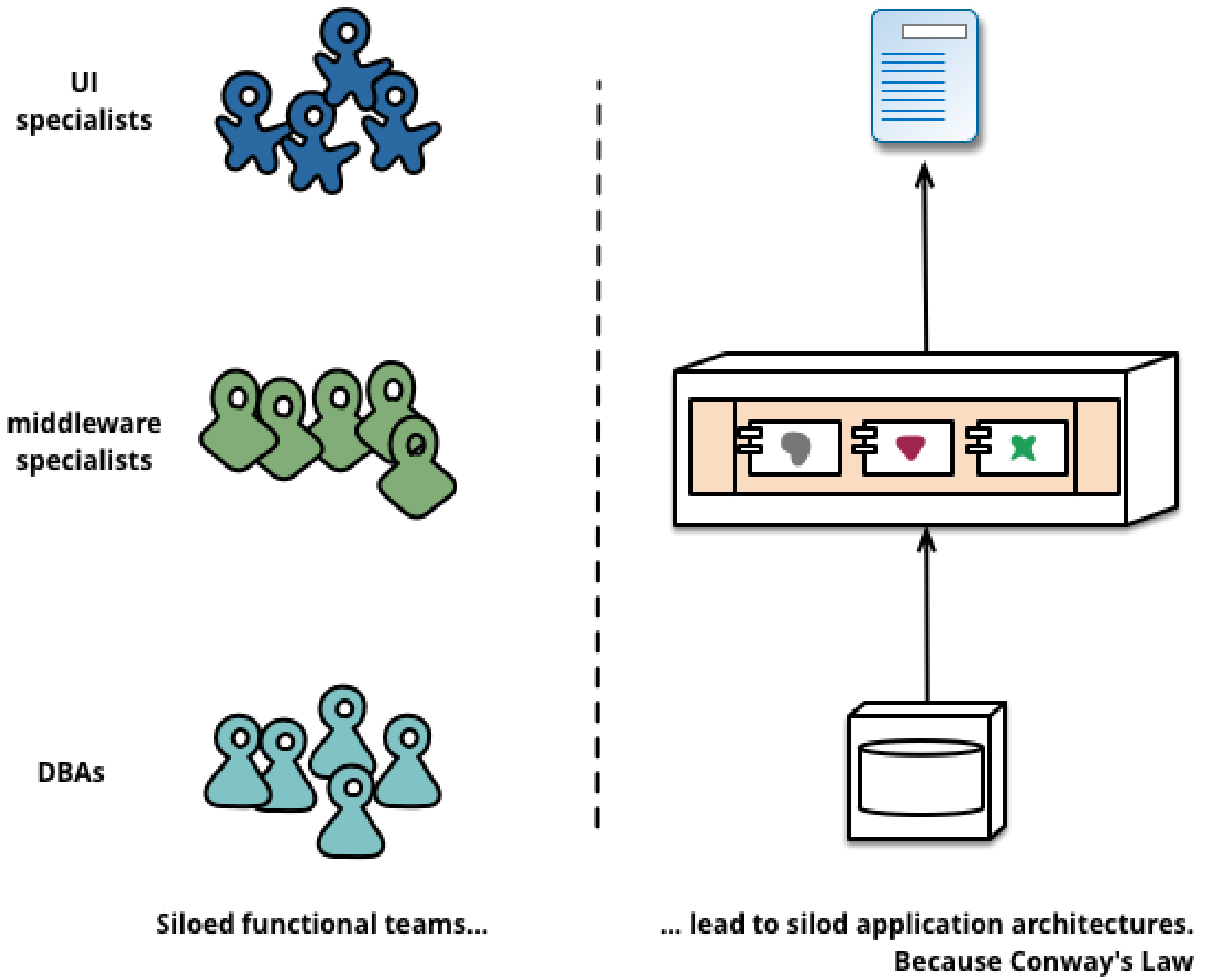
Deployability

- Deploy each service independently
  - Enables to do a change in a service and deploy it inmediately
  - Towards continuous deployment

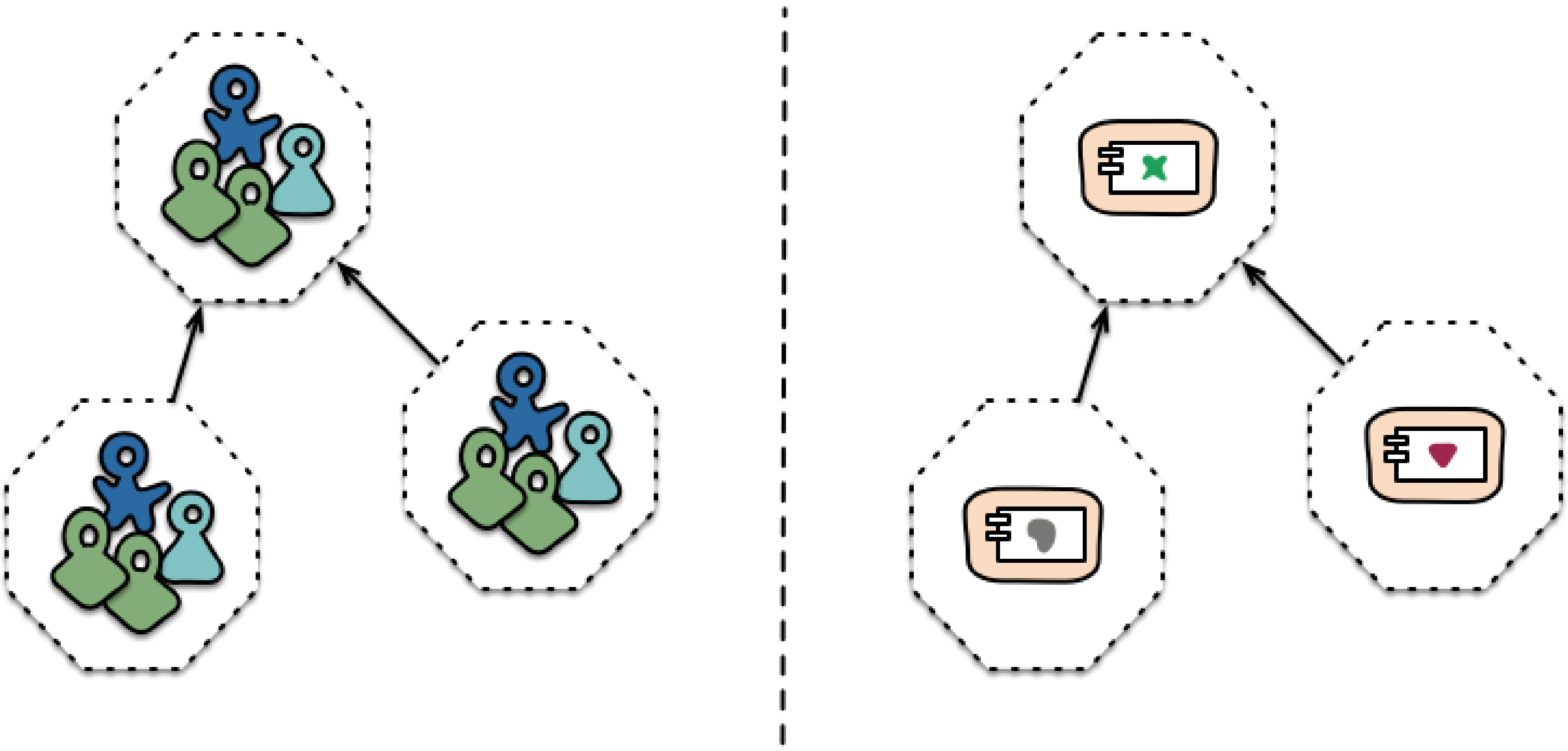
Organizational alignment

- Inverse Conway Law maneuver
  - Evolve teams and organizational structure to promote the desired architecture
- Create teams following the modular decomposition
  - Cross-functional teams
- Service ownership: the team owning a service is responsible for making changes and deploying it
  - "You build it, you run it" (Amazon)
- Goal: increased autonomy and speed of delivery

Traditional applications



With microservices

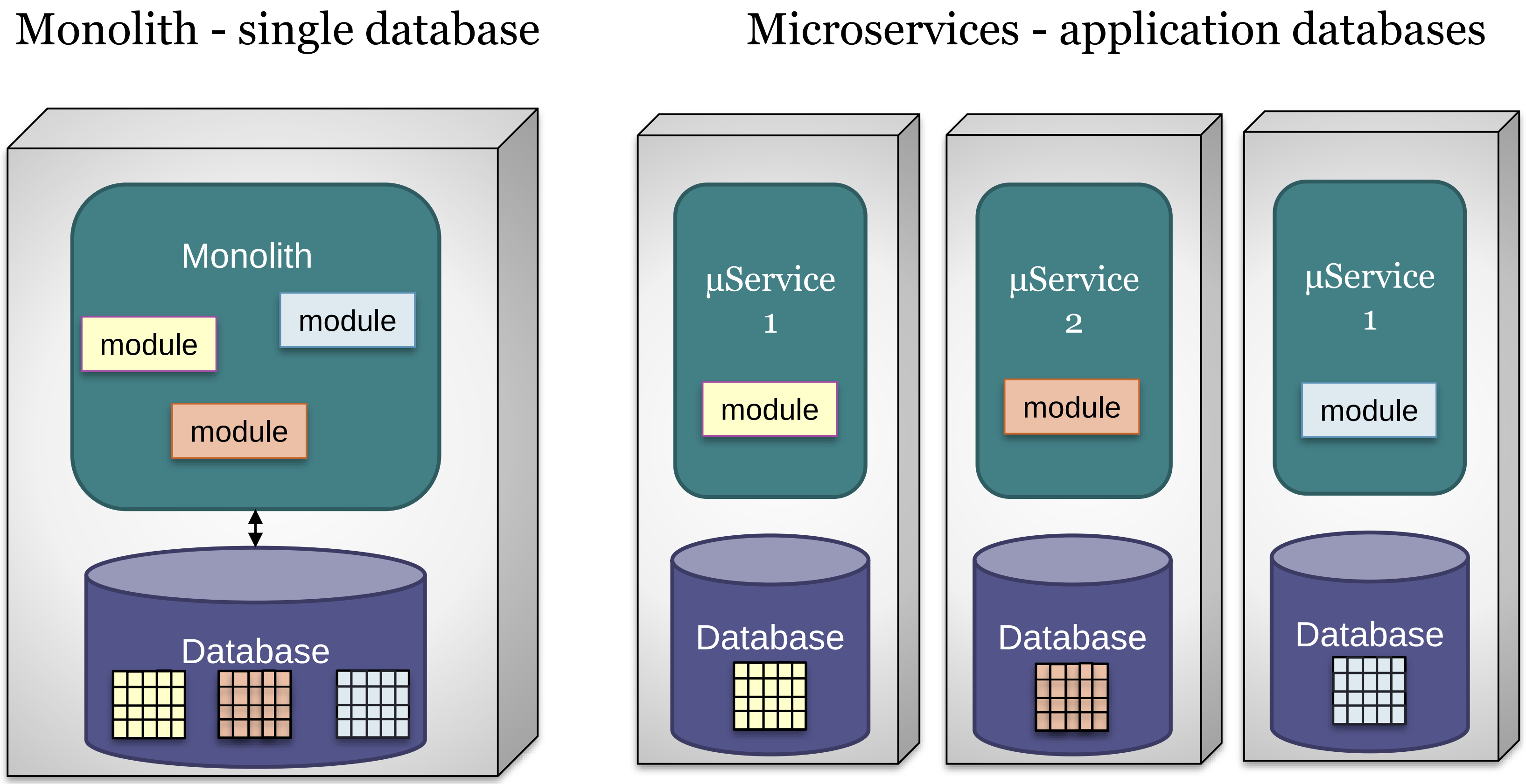


Cross-functional teams...

... organised around capabilities
Because Conway's Law

Decentralized data management

Each team/service handles its own data



Optimizing for replaceability

- Traditional systems usually contained old legacy systems which no one wants to touch
- With microservices
  - Less cost to replace a microservice with a better implementation
  - Or even delete it

Challenges of microservices

- Managing lots of microservices
  - Too much microservices = antipattern (nanoservices)
  - Ensure application consistency
- Complexity of distributed system management
  - New challenges: latency, message format, load balance, fault tolerance, etc.
- Testing & deployment
  - Operational complexity
- Antipattern: distributed monolith
  - Microservices tangled that are not independently deployed
- Structural decay (*see next slide*)

http://martinfowler.com/articles/microservice-trade-offs.html
https://www.ufried.com/blog/microservices_fallacy_1/

Microservices structural decay

- Code dependencies between services
  - Too much shared libraries
- Too much inter-service communication
- Too many orchestration requests
- Database coupling

Analyzing architecture (microservices)
https://www.youtube.com/watch?v=U7s7Hb6GZCU

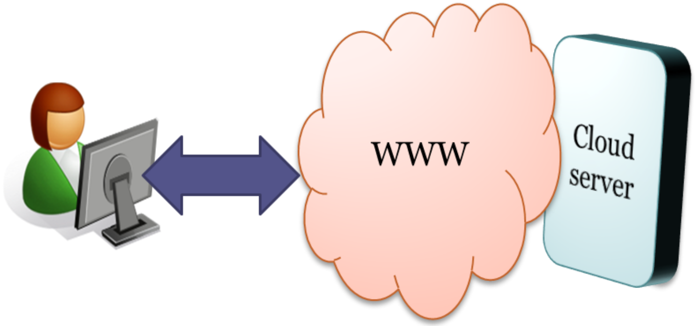
Microservices

Variants

Self contained Systems (SCS) Architecture

- Separation of functionality into many independent systems
- https://scs-architecture.org/
- Each SCS contains logic and data

Serverless

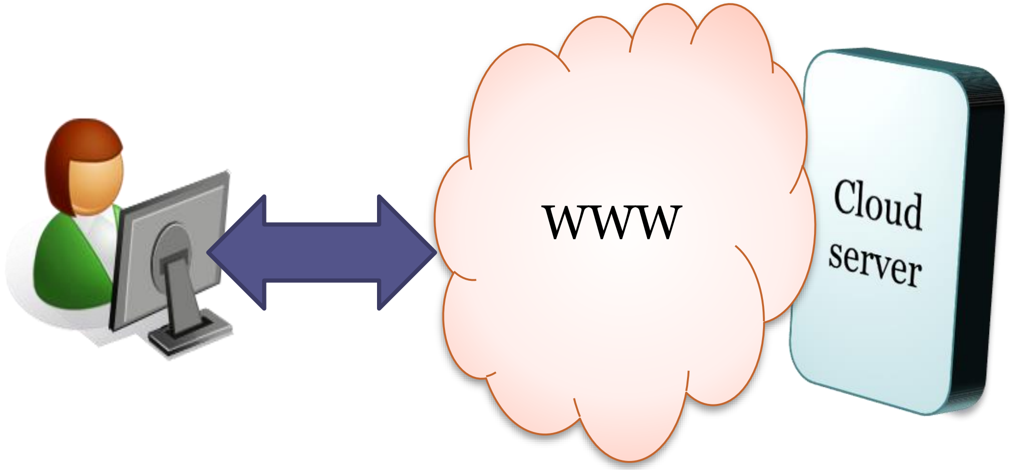


- Also known as:
  - Function as a service (FaaS)
  - Backend as a service (BaaS)
- Applications depend on third-party services
- Developers don't need to care about servers
  - Automatic scalability
- Rich clients
  - *Single Page Applications, Mobile apps*
- Examples:
  - AWS Lambda, Google Cloud Functions, Ms Azure Functions

https://en.wikipedia.org/wiki/Serverless_computing
https://martinfowler.com/articles/serverless.html

Serverless

- Elements
  - Client that runs functions as a services
  - Cloud server which provides backend as a service
- Constraints
  - No management of server hosts
  - Automatic scalability and provisioning based on load
  - Costs based on precise usage



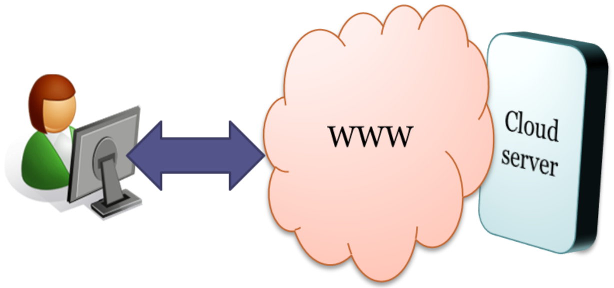
Serverless

Advantages

- Automatic scalability
- Implicit high availability
- Performance not defined in terms of host size/cost
- Costs based on precise usage
    - Only pay for the compute you need
- Time to market

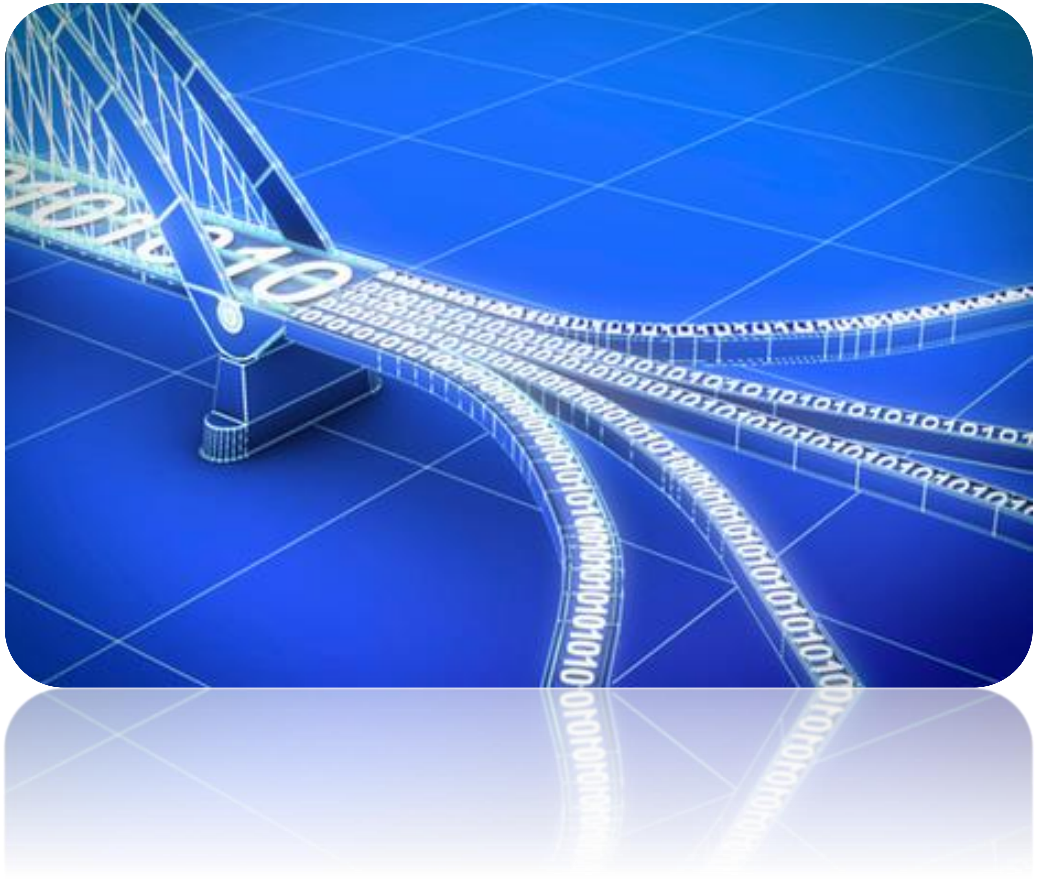
Challenges

- Vendor control
- Vendor lock-in
    - Incompatibility between vendors
- Security
- Startup latency
- Integration testing
- Monitoring/debugging



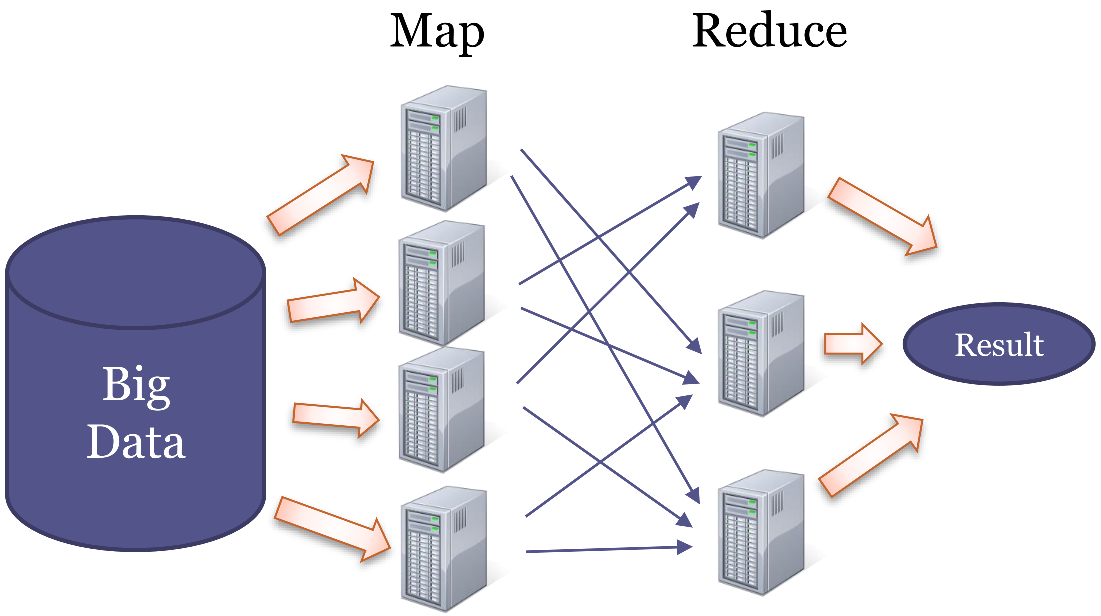
Big data and scalable systems

MapReduce
Lambda architecture
Kappa architecture



MapReduce

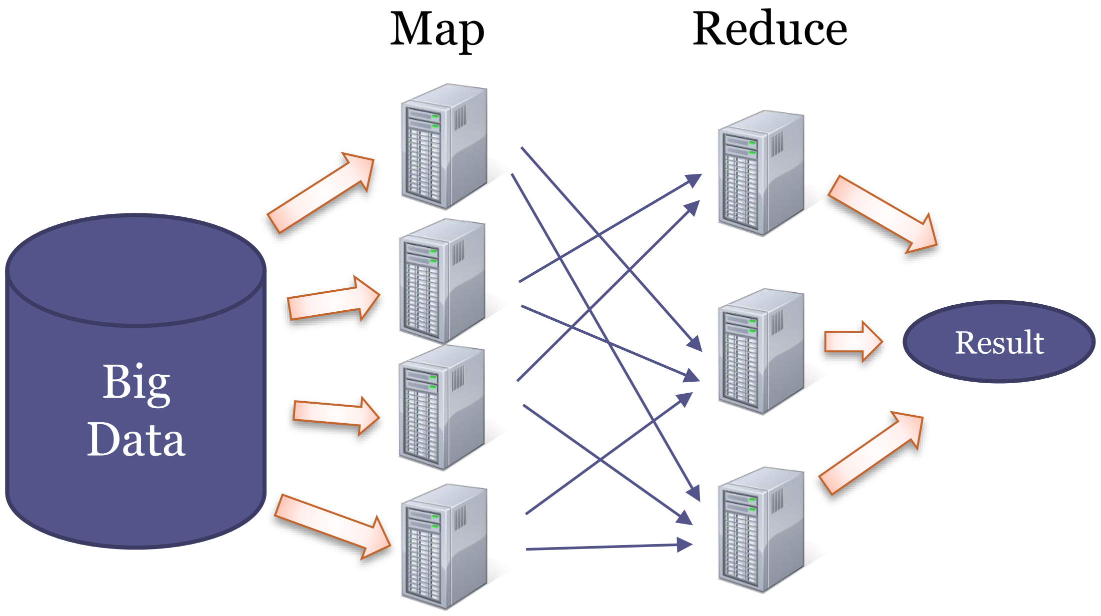
- Proposed by Google
  - Published in 2004
  - Internal implementation by Google
- Goal: big amounts of data
  - Lots of computational nodes
  - Fault tolerance
  - Write-once, read-many
- Style composed of:
  - Master-slave
  - Batch



MapReduce

Elements

- *Master* node: Controls execution
  - Node table
  - It manages replicated file system
- *Slave nodes*
  - Execute mappers, reducers
  - Contain replicated data blocks



MapReduce - Scheme

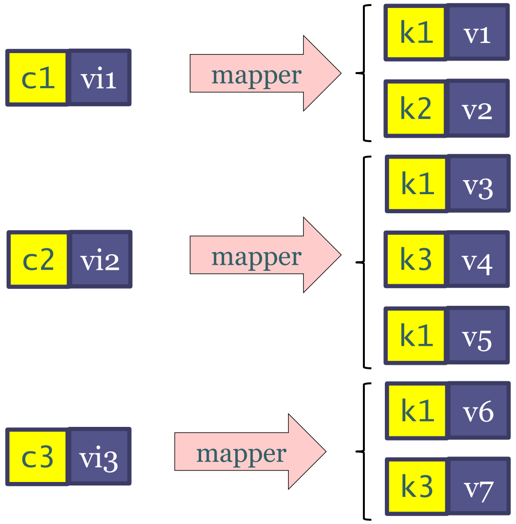
- Inspired by functional programming
  - 2 components: mapper and reducer
- Data are divided for their processing
- Each data is associated with a key
  - Transforms `[(key1,value1)]` to `[(key2,value2)]`

Input:
`[(key1,value1)]`

Output:
`[(key2,value2)]`

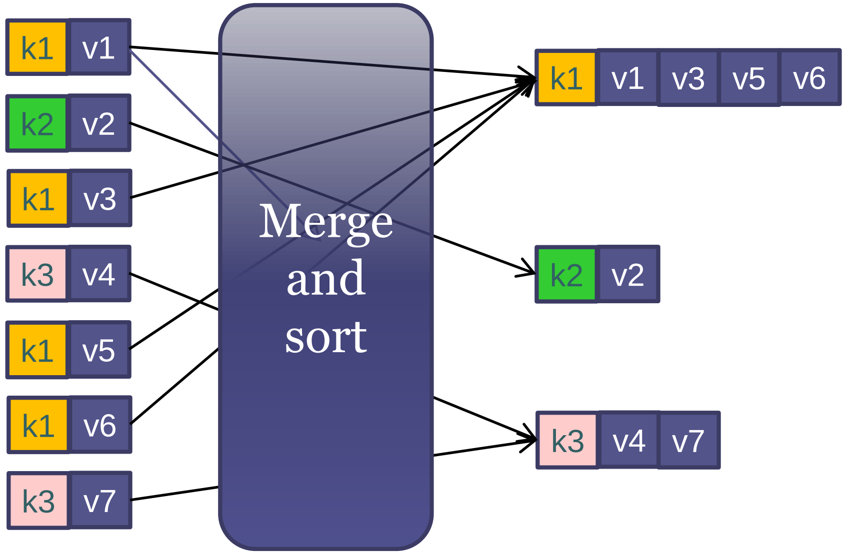
Step 1: mapper

mapper: `(Key1, Value1)` → `[(Key2,Value2)]`



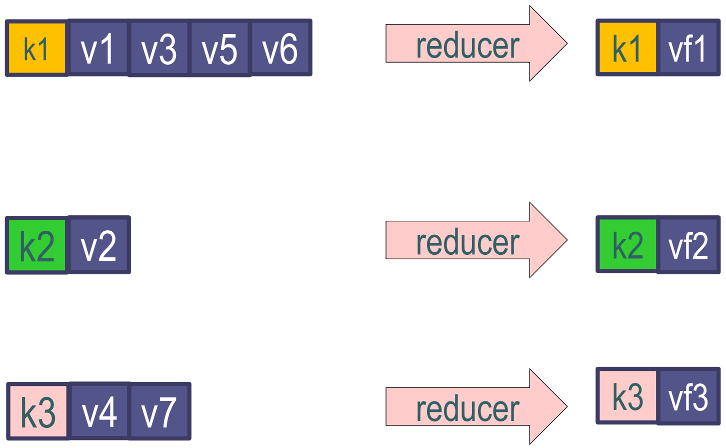
Step 2: Merge and sort

System merges and sorts intermediate results according to the keys

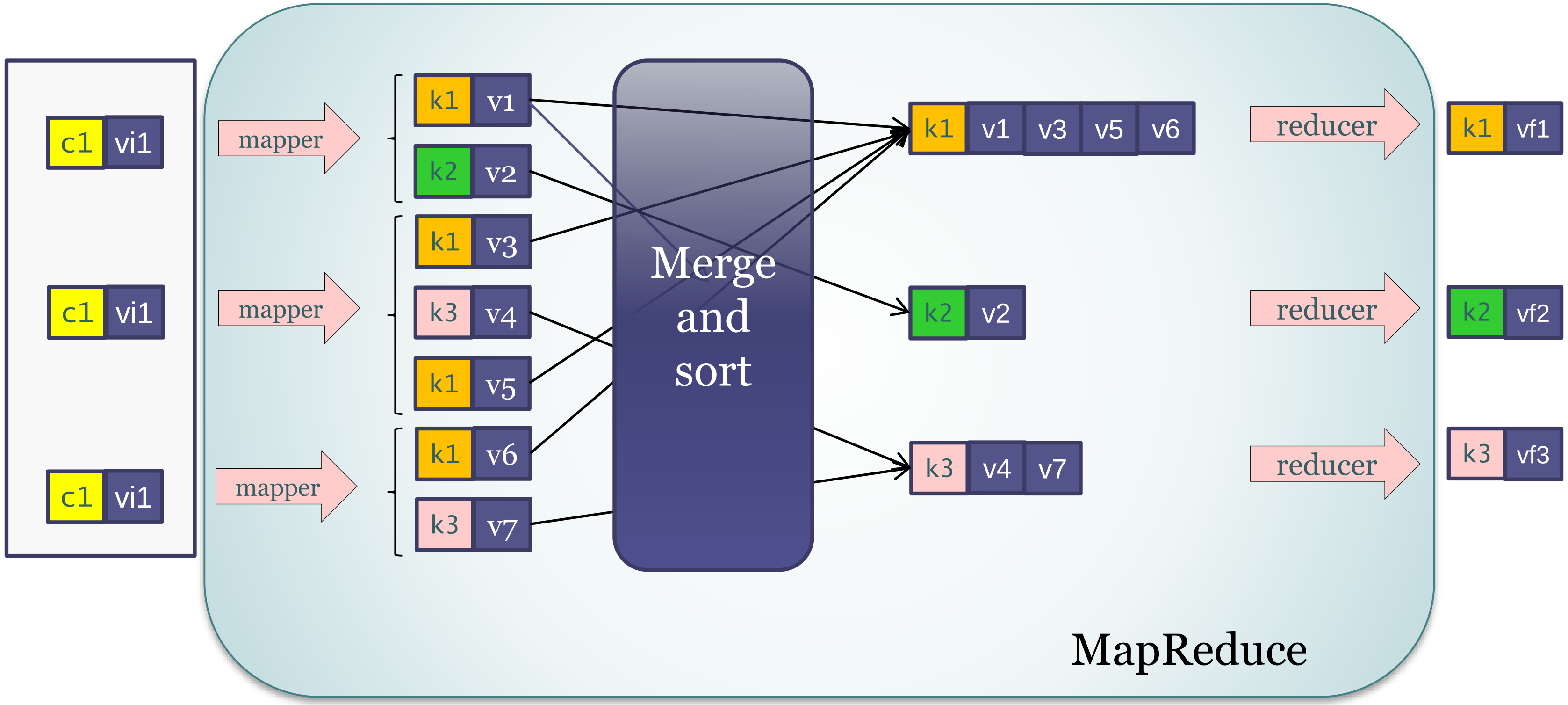


Step 3: Reducers

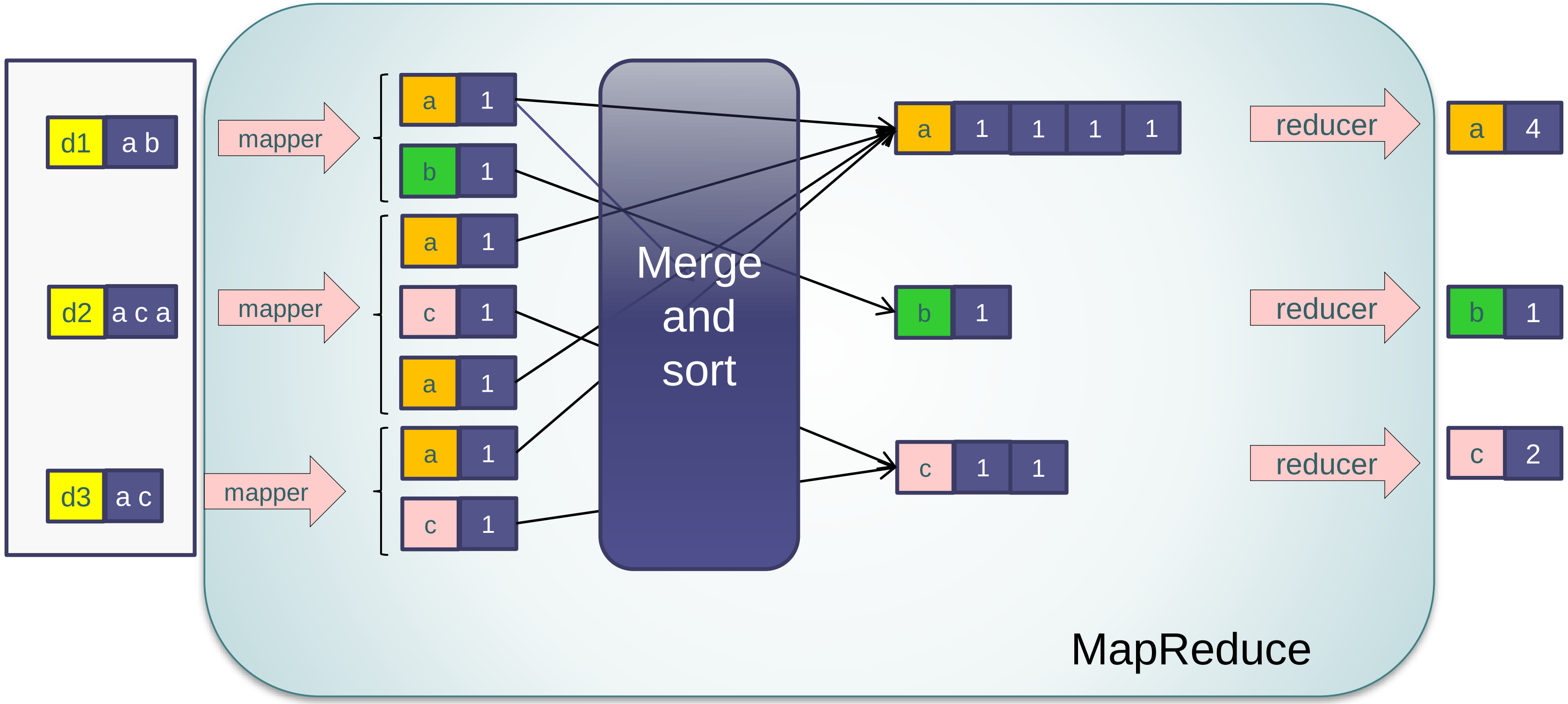
reducer: `(Key2, [Value2]) → (Key2,Value2)`



MapReduce - general scheme



MapReduce - count words




```
// return each work with 1
mapper(d,ps) {
 for each p in ps:
   emit (p, 1)
}
```

```
// sum the list of numbers of each word
reducer(p,ns) {
   sum = 0
   for each n in ns { sum += n; }
   emit (p, sum)
}
```

MapReduce - execution environment

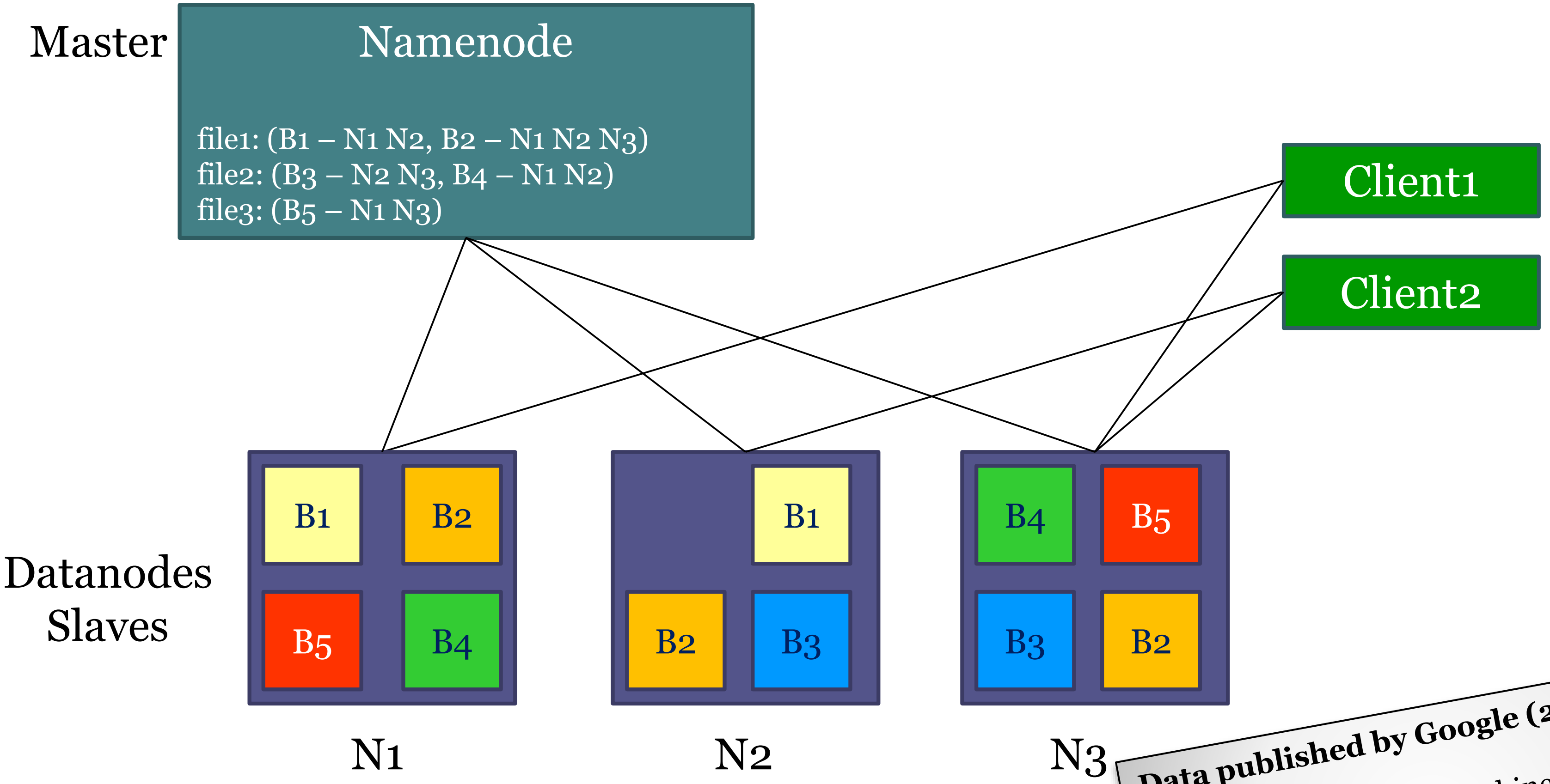
Execution environment is in charge of:

- Planning: Each job is divided in tasks
- Placement of data/code
  - Each node contains its data locally
- Synchronization:
  - *reduce* tasks must wait *map* phase
- Error and failure handling
  - High tolerance to computational nodes failures

MapReduce - File system

- Google developed a distributed file system - GFS
- Hadoop created HDFS
  - Files are divided in chunks
  - 2 node types:
    - Namenode (master), datanodes (data servers)
  - Datanodes store different chunks
    - Block replication
  - *Namenode* contains metadata
    - Where is each chunk
    - Direct communication between clients and datanodes

MapReduce - File system



Data published by Google (2007)
200+ clusters
Lots of clusters 1000+ machines
Pools with thousands of clients
4+ PB
HW/SW fault tolerance

MapReduce

Advantages

- Distributed computations
  - Split input data
    - Replicated repository
  - Fault tolerant
    - Hardware/software heterogeneous
- Large amount of data
  - Write-once. Read-many

Challenges

- Dependency on master node
- Non interactivity
- Data conversion to MapReduce
  - Adapt input data
  - Convert output data

MapReduce: Applications

Lots of applications:

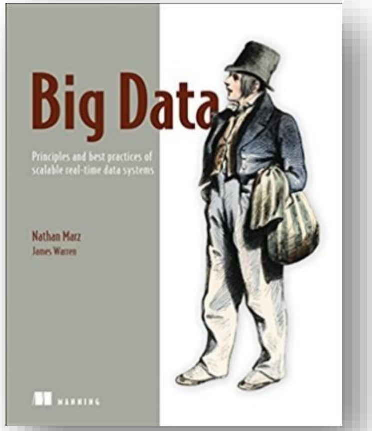
- Google, 2007, 20petabytes/day, around 100,000 mapreduce jobs/day
- PageRank algorithm can be implemented as MapReduce
- Success stories:
  - Automatic translation, similarity, sorting, ...
- Other companies: last.fm, facebook, Yahoo!, twitter, etc.

MapReduce: Applications

- Implementations
  - Google (internal)
  - Hadoop (*open source*)
  - …
- Libraries
  - Hive (Hadoop): query language inspired by SQL
  - Pig (Hadoop): specific language that can define data flows
  - Cascading: API that can specify distributed data flows
  - Flume Java (Google)
  - Dryad (Microsoft)

Lambda architecture

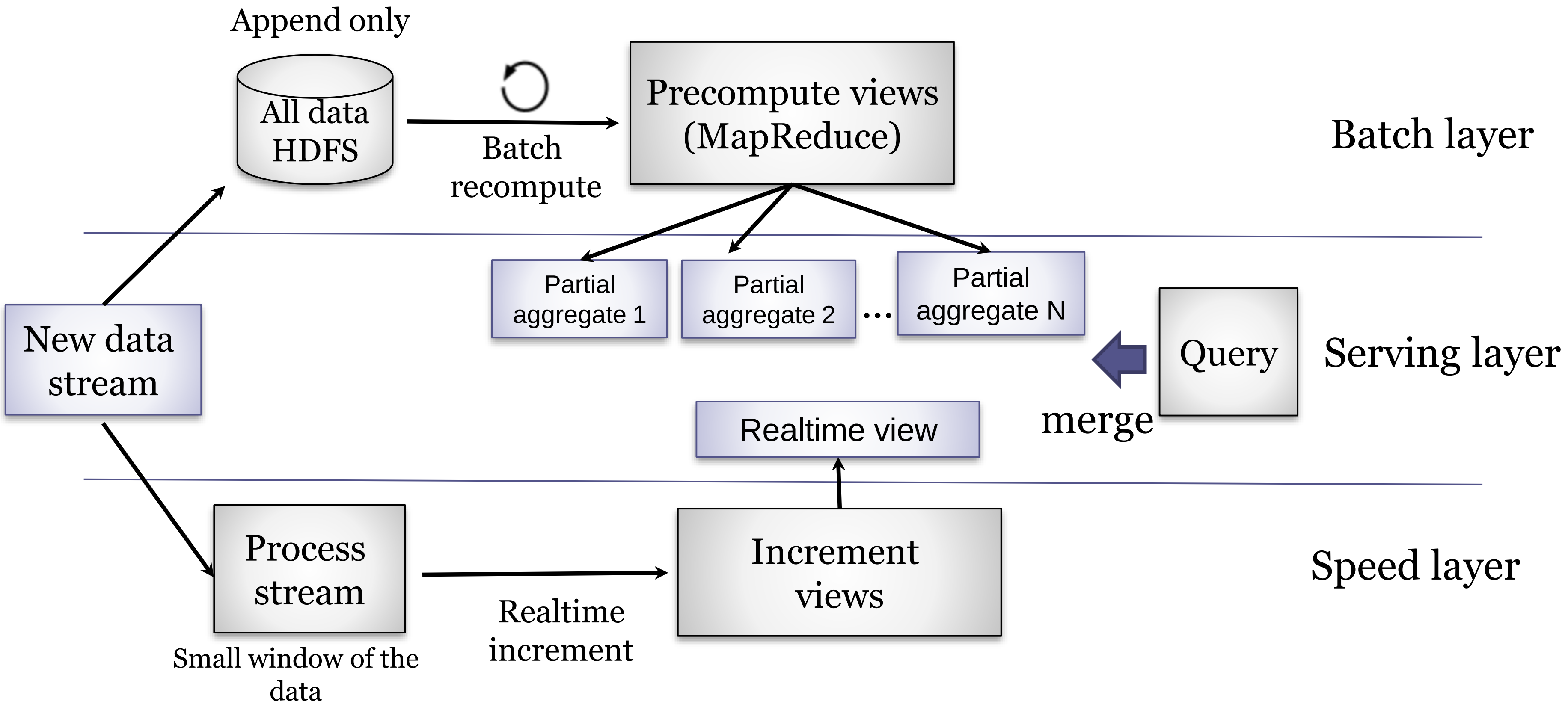
- Handle Big Data & real time analytics
- Proposed by Nathan Marz, 2011
- 3 layers
  - Batch layer: precomputes all data with MapReduce
    - Generates partial aggregate views
    - Recomputes from all data
  - Speed layer: real time, small window of data
    - Generates fast real time views
  - Serving layer: handles queries
    - Merges the different views



Lambda architecture

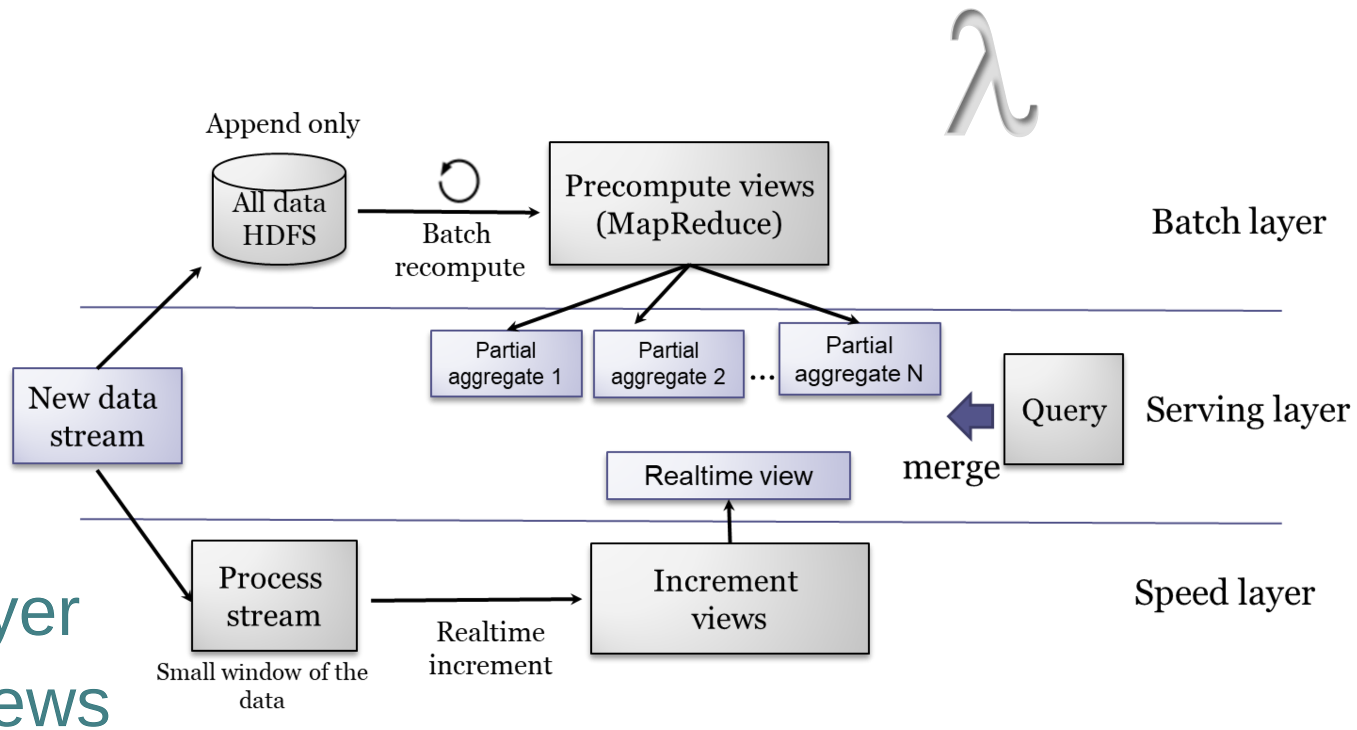
λ

Combines Real time with batch processing



University of Oviedo

Lambda architecture



Constraints

- All data is stored in the batch layer
- The batch layer precomputes views
- The results of the speed layer may not be accurate
- Serving layer combines precomputed views
- The views can be simple DBs for querying

School of Computer Science

Lambda architecture

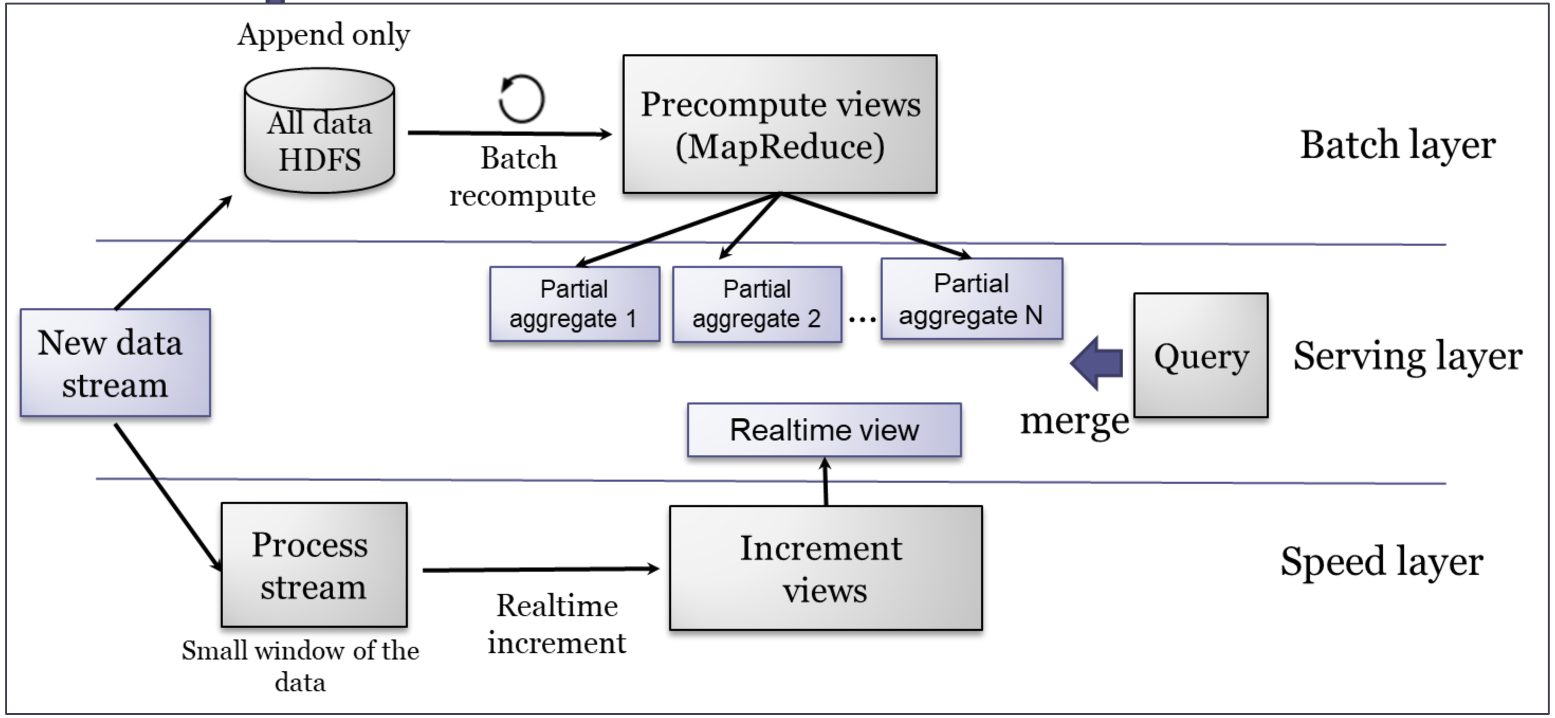
λ

Advantages

- Scalability (Big data)
- Real time
- Decoupling
- Fault tolerant
- Keep all input data
- Reprocessing

Challenges

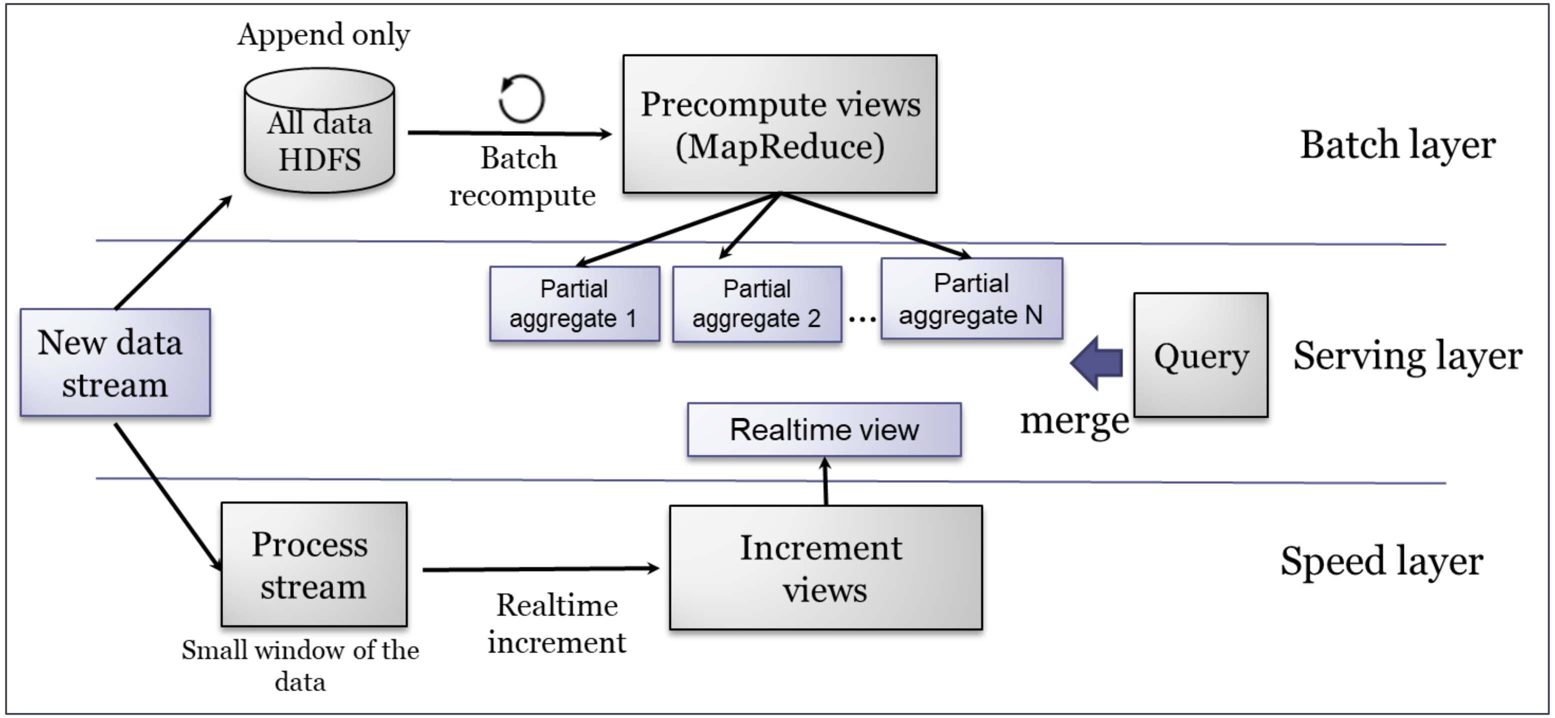
- Inherent complexity
- Merging views can be innacurate
  - Losing some events



Lambda architecture

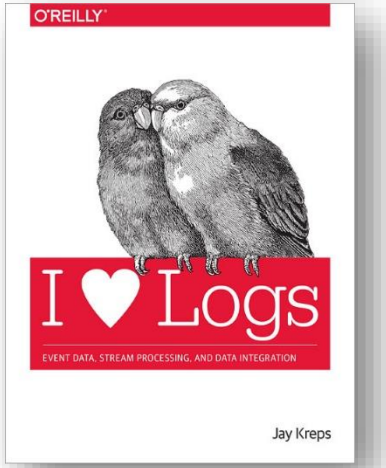
λ

- Applications
  - Spotify, Alibaba, …
- Libraries
  - Apache Storm
  - Netflix Suro project



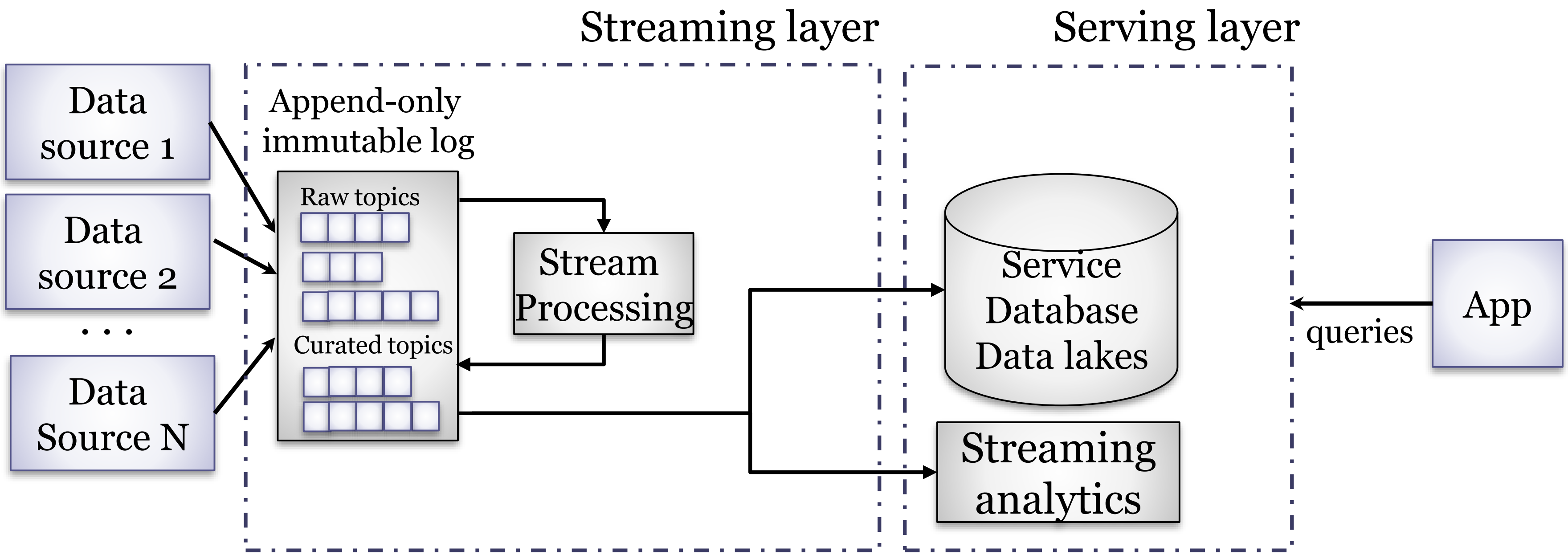
Kappa architecture

- Proposed by Jay Krepps (Apache Kafka), 2013
- Handle Big data & Real time with logs
  - Simplifies Lambda architecture
    - Removes the batch layer
  - Based on a distributed ordered log
    - Replicated cluster
    - The log can be very large



Kappa architecture

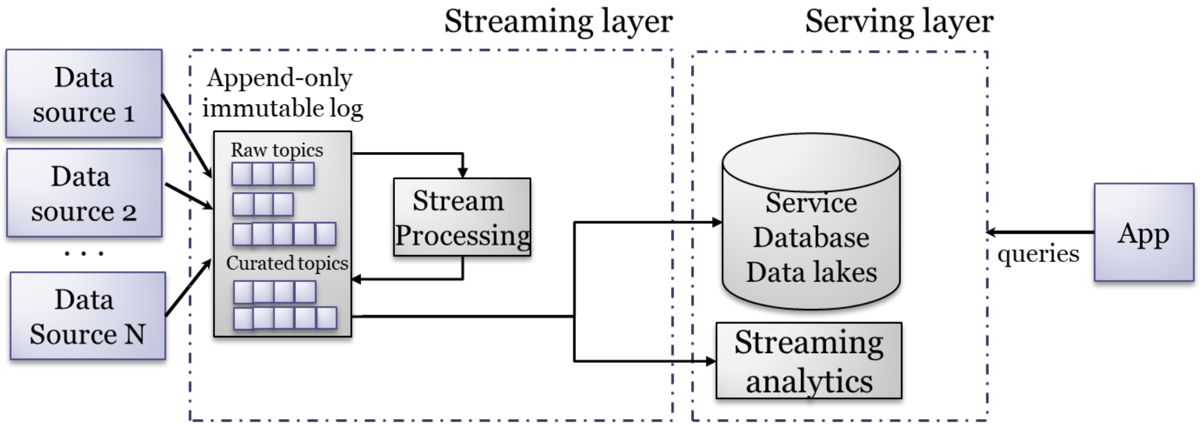
Diagram



Kappa architecture

Constraints

- The event log is append-only
- The events in the log are immutable
- Stream processing can request events at any position
  - To handle failures or doing recomputations



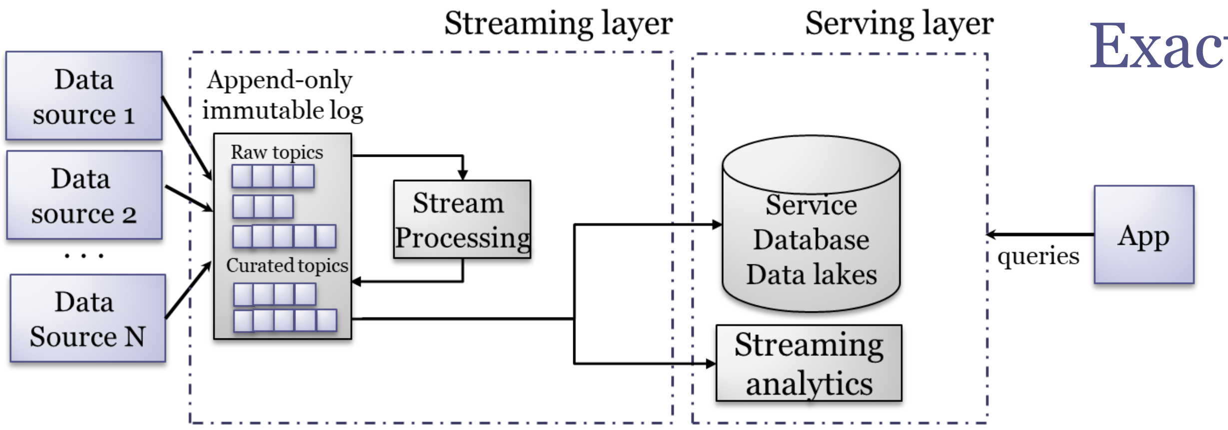
Kappa architecture

Advantages

- Scalable (big data)
- Real time processing
- Simpler than lambda
    - No batch layer

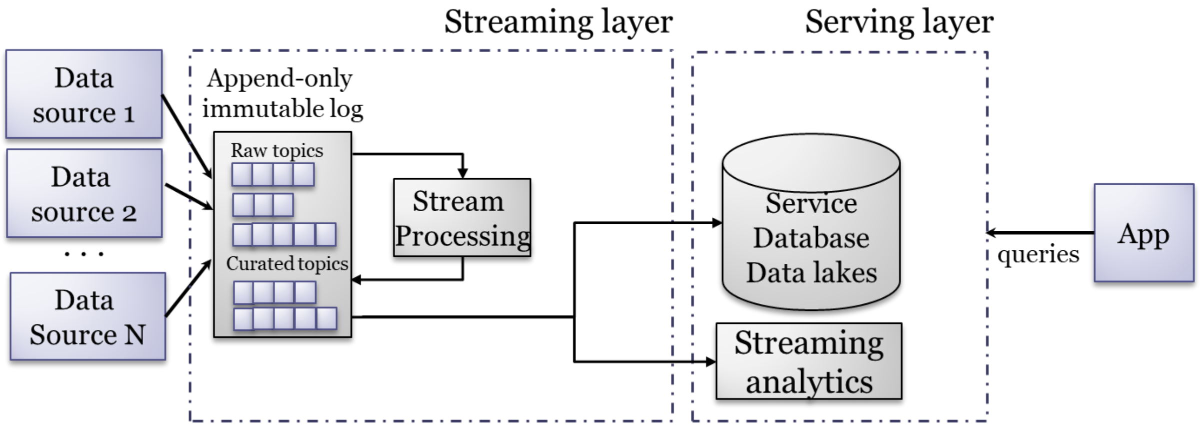
Challenges

- Space requirements
    - Duplication of log and DB
    - Log compaction
- Ordering of events
- Delivery paradigms
    - At least once, at most once (it may be lost), Exactly once



Kappa architecture

- Applications & libraries
  - Apache Kafka
  - Apache Samza
  - Spark Streaming
- LinkedIn



University of Oviedo

End of presentation

School of Computer Science