Tampere University

Communication patterns

Kari Systä, 26.10.2021

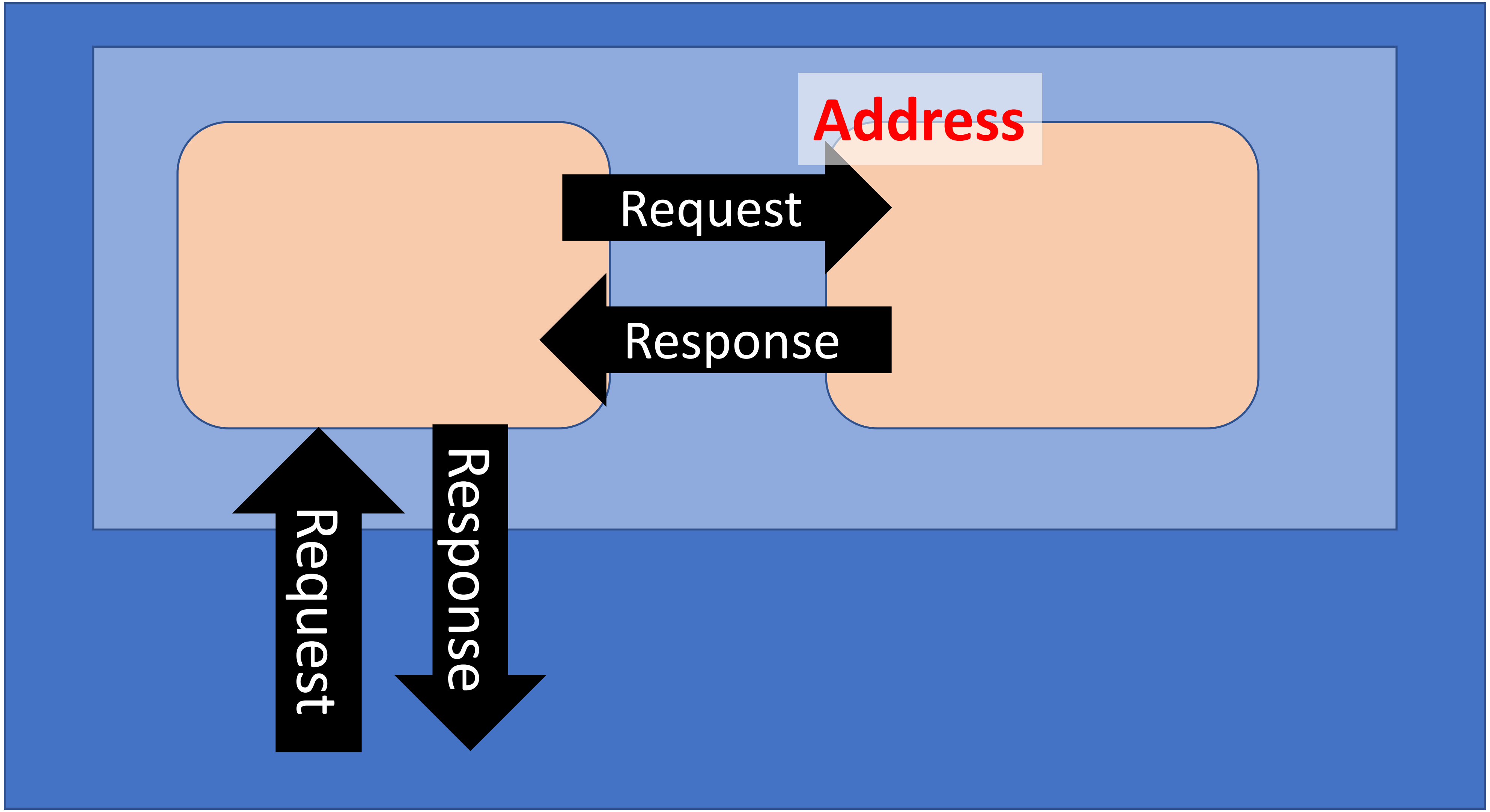
Architectural principles of REST

- **Client-server architecture**
- **Statelessness**
  - **Everybody gets same answer**
  - **Repeated operation (GET, PUT) does not have an effect**
- **Cacheability**
  - **For performance and scalability**
- **Layered system**
  - **Allows proxies etc**
- **Uniform interface**

Uniform interface

- Everything is a resource that is fetched, modified, created, deleted
  - CRUD = CREATE, READ, UPDATE, DELETE
  - HTTP verbs: GET, PUT, POST, DELETE
  - Resource manipulation through representations
- Resource identification in requests
  - URIs
  - Separated from representation (XML, JSON,…)
  - MIME-types
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)

Back to old picture



Corner-stones of REST

- Client-server architecture
  - Separation of concerns
- Statelessness
  - no client context being stored on the server between requests
- Cacheability
- Layered system
  - Client does not know if connected to other end directly
- Uniform interface

Do not call your design for previous exercise REST!

Tampereen yliopisto
Tampere University

Uniform representation

- Resource identification in requests
  - URIs
  - Separated from representation (XML, JSON,…)
- Resource manipulation through representations
- Self-descriptive messages
- Hypermedia as the engine of application state (HATEOAS)
- Application to HTTP
  - URL's
  - GET, PUT, POST, DELETE
  - MIME-types

Tampereen yliopisto
Tampere University

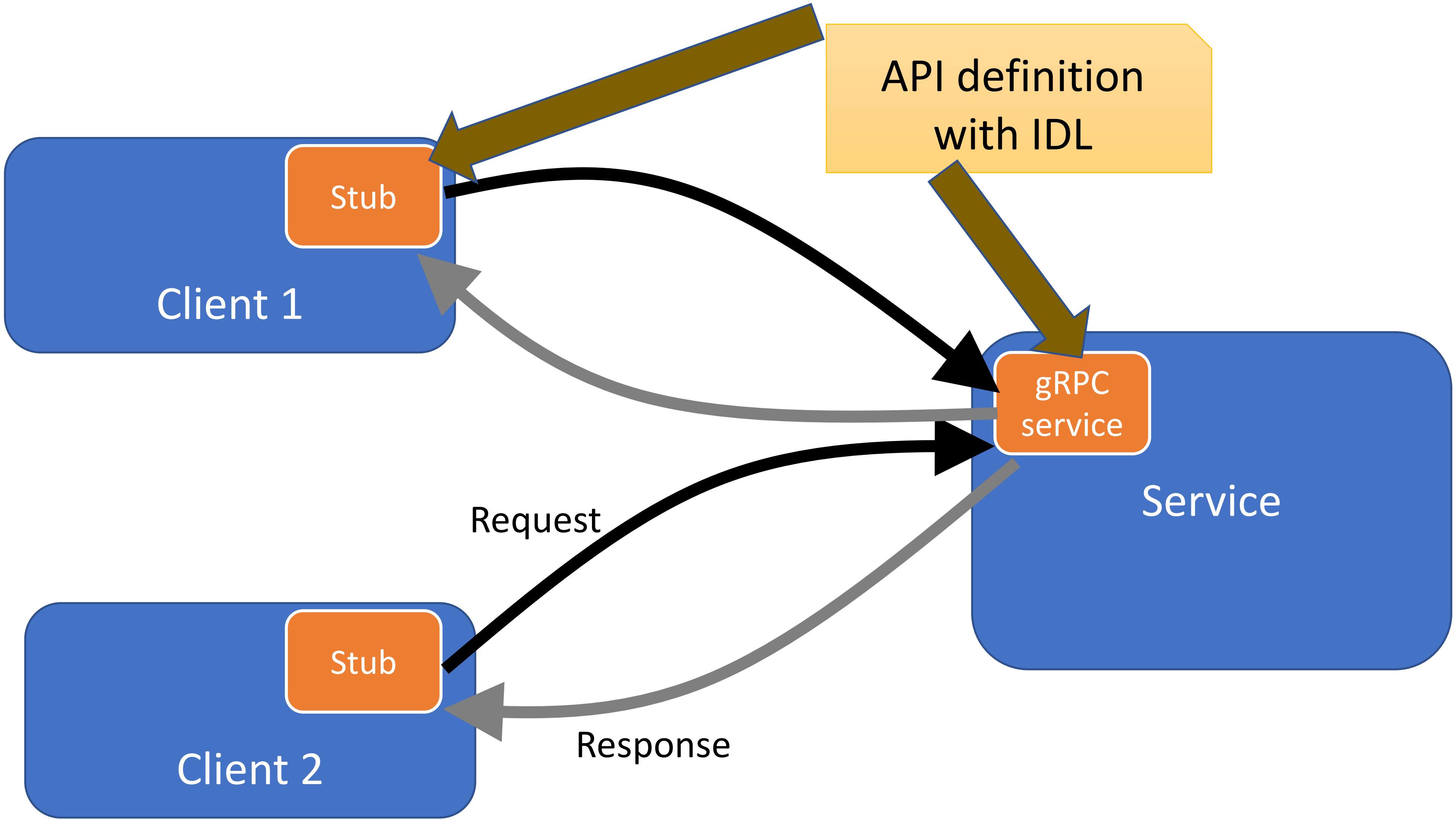
But the ”calls” can be laborous

```
let message = "Hello from " + req.client.remoteAddress + ":" +
req.client.remotePort + " to " + req.client.localAddress + ":" +
req.client.localPort;

request('http://server2:4000/getServer’,  { json: true },
        (err, response, body) => {
                                    if (err) {
                                      return console.log(err);
                                    }
                                    res.send(message + " " + body); });
```

REST vs RPC

gRPC – RPC over HTTP



Example API description

```
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
  // Sends another greeting
  rpc SayHelloAgain (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest { string name = 1; }

// The response message containing the greetings message
HelloReply { string message = 1; }
```

Tampereen yliopisto
Tampere University

Call in JavaScript and Python

```
function main() {
  var client = new hello_proto.Greeter('localhost:50051',
                                       grpc.credentials.createInsecure());
  client.sayHello({name: 'you'}, function(err, response) {
    console.log('Greeting:', response.message);
  });
  client.sayHelloAgain({name: 'you'}, function(err, response) {
    console.log('Greeting:', response.message);
  });
}
```

```
def run():
    channel = grpc.insecure_channel('localhost:50051')
    stub = helloworld_pb2_grpc.GreeterStub(channel)
    response = stub.SayHello(helloworld_pb2.HelloRequest(name='you'))
    print("Greeter client received: " + response.message)
    response = stub.SayHelloAgain(helloworld_pb2.HelloRequest(name='you'))
    print("Greeter client received: " + response.message)
```

Tampereen yliopisto
Tampere University

And C++

```cpp
std::string SayHelloAgain(const std::string& user) {
  // Follows the same pattern as SayHello.
  HelloRequest request;
  request.set_name(user);
  HelloReply reply;
  ClientContext context;

  // Here we can use the stub's newly available method we just added.
  Status status = stub_->SayHelloAgain(&context, request, &reply);
  if (status.ok()) {
    return reply.message();
  } else {
    std::cout << status.error_code() << ": " << status.error_message()
              << std::endl;
    return "RPC failed";
  }
}
```

GraphQL(examples from https://medium.com/tech-tajawal/backend-for-frontend-using-graphql-under-microservices-5b63bbfcd7d9)

- REST request

```
GET http://127.0.0.1/api/accounts
```

- Response

```
[
  {
    "id": 88,
    "name": "Mena Meseha",
    "photo": "http://..m/photo.jpg"
  },
...
]
```

- GraphQL request

```
POST http://127.0.0.1/graphql
```

- Payload

```
query {accounts {id, name, photo}}
```

- Response

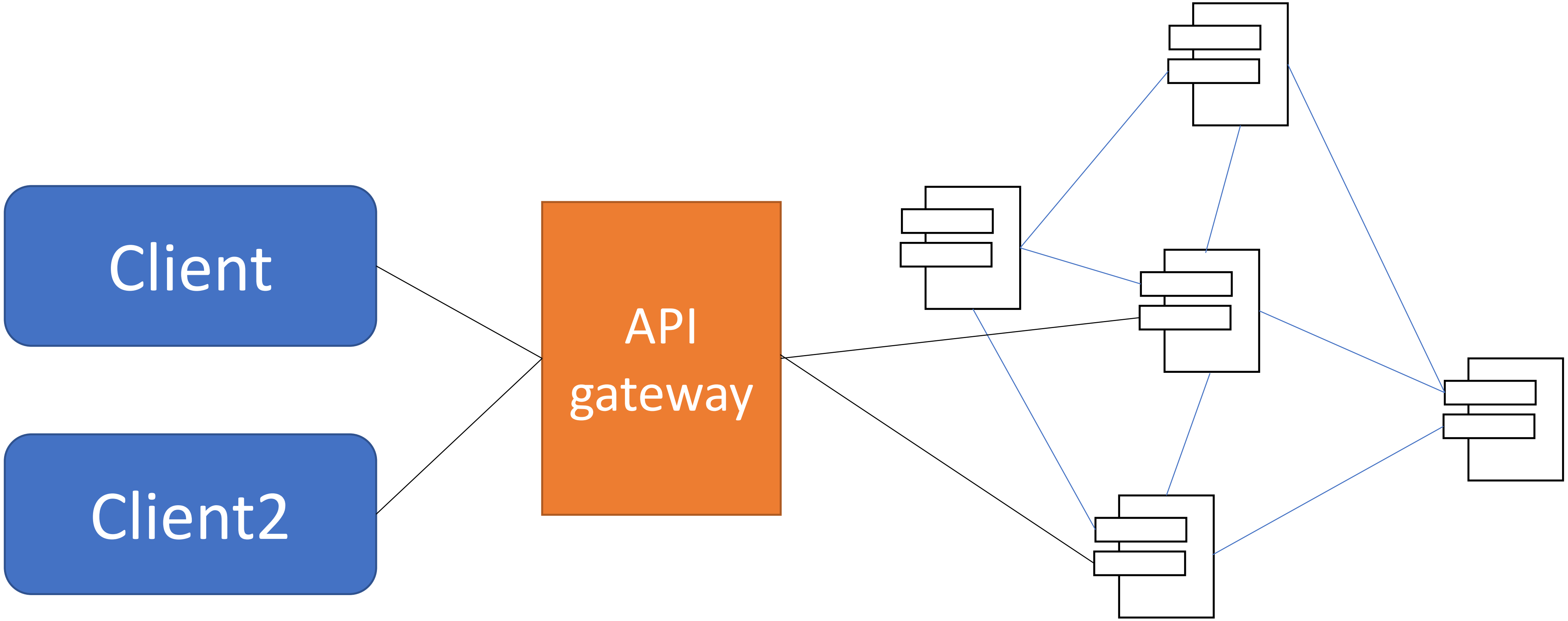
```
{
 "data": {
  "accounts": [ {
    "id": 88,
    "name": "Mena Meseha",
    "photo":
      "http://...com/photo.jpg"
    },
   ...
  ]
 }
}
```

Tampereen yliopisto
Tampere University

Let's analyze some claims of the previous source

- **1. Data Acquisition:** REST lacks scalability and GraphQL can be accessed on demand. The payload can be extended when the GraphQL API is called.
- **2. API calls:** REST's operation for each resource is an endpoint, and GraphQL only needs a single endpoint, but the post body is not the same.
- **3. Complex data requests:** REST requires multiple calls for nested complex data, GraphQL calls once, reducing network overhead.
- **4. Error code processing:** REST can accurately return HTTP error code, GraphQL returns 200 uniformly, and wraps error information.
- **5. Version number:** REST is implemented via v1/v2, and GraphQL is implemented through the Schema extension.

How about external calls?



API gateway pattern
https://microservices.io/patterns/apigateway.html

Problem

- How do the clients of a Microservices-based application access the individual services?

Forces

- The granularity of APIs provided by microservices is often different than what a client needs and too fine grained.
- Different clients need different data.
- Network performance is different for different types of clients.
- Partitioning into services can change over time and should be hidden from clients
- Services might use a diverse set of protocols, some of which might not be web friendly

Solution

- Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services.

Tampereen yliopisto
Tampere University

RECALL Interface segregation principle

"many client-specific interfaces are better than one general-purpose interface."

"Make fine grained interfaces that are client specific"

"Clients should not be forced to depend upon methods they do not use"

- Big system with many dependencies = small change causes changed everywhere
- Large interfaces are split to smaller and role-base interfaces.

  ⇒changes do not affect everybody

  ⇒New features are easier to add

  ⇒Interfaces are easier to learn

Tampereen yliopisto
Tampere University

Other Concerns

Application architecture patterns

- Which architecture should you choose for an application?

Decomposition

- How to decompose an application into services?

Data management

- How to maintain data consistency and implement queries?

Transactional messaging

- How to publish messages as part of a database transaction?

Testing

- How to make testing easier?

Deployment patterns

- How to deploy an application's services?

Cross cutting concerns

- How to handle cross cutting concerns?

Communication patterns

Message queue approach

Message-bus instead of HTTP

- Challenges of REST and RPC: increased network operations, tight service coupling
- Message bus helps to define how services communicate, service discovery reduces operational complexity
- Asynchronous messaging leads to
  - loosed coupling
  - More complex logic (async a cousin of parallelism)
- Actually, there are multiple options
  - RPC, REST, Asynchronous message, application-specific protocols

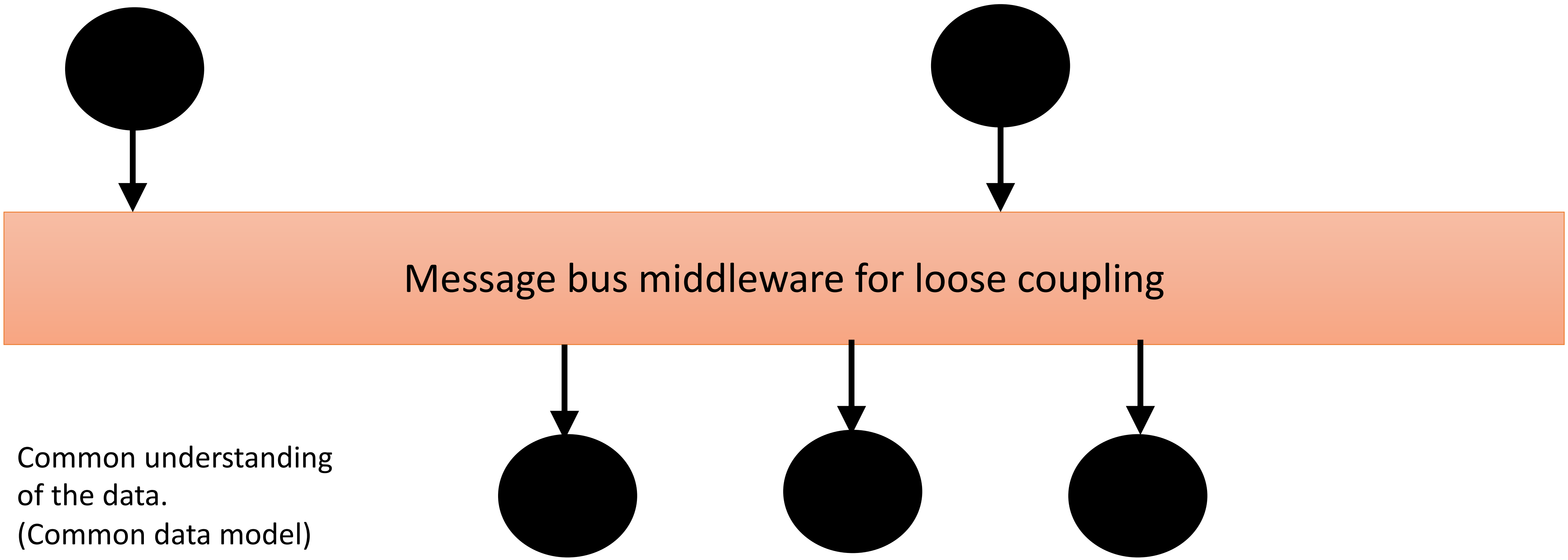
Tampereen yliopisto
Tampere University

Message-bus instead of HTTP

- Challenges: increased network operations, tight service coupling
- Message bus helps to define how services communicate, service discovery reduces operational complexity
- Asynchronous messaging leads to
  - loosed coupling
  - More complex logic (async is a cousin of parallelism)
- Actually, there are multiple options
  - RPC, REST, Asynchronous message, application-specific protocols

Tampereen yliopisto
Tampere University

The message bus approach



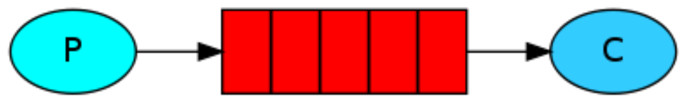
Common understanding
of the data.
(Common data model)

RabbitMQ

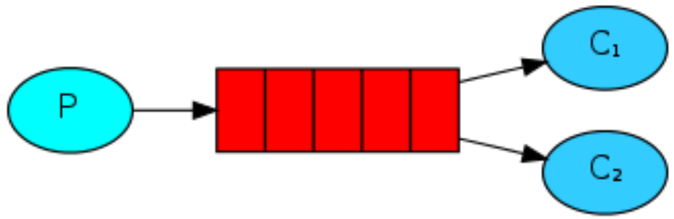
- An example of message queue technology
- Can be used to implement various architectures

Examples of RabbitMQ use
https://www.rabbitmq.com/getstarted.html

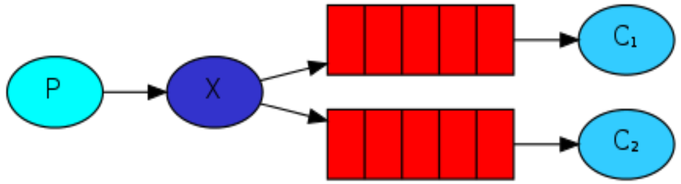
Simple queue



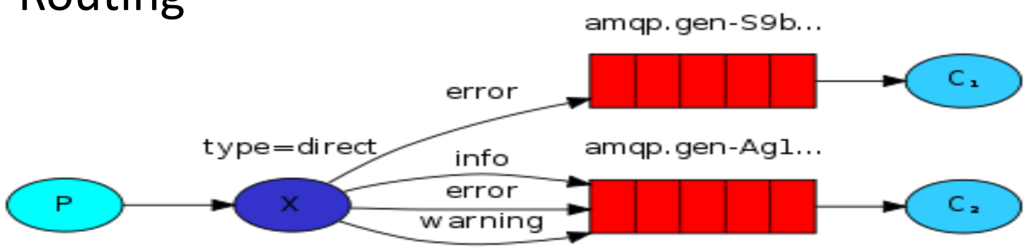
Task distribution



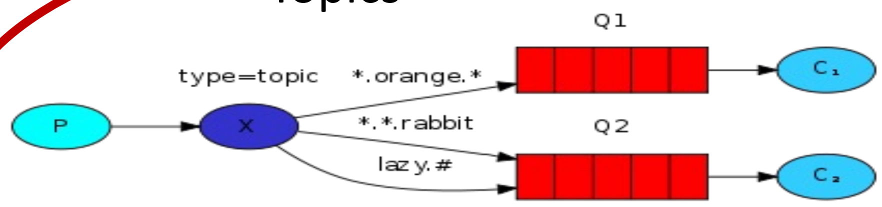
Publish/subscribe



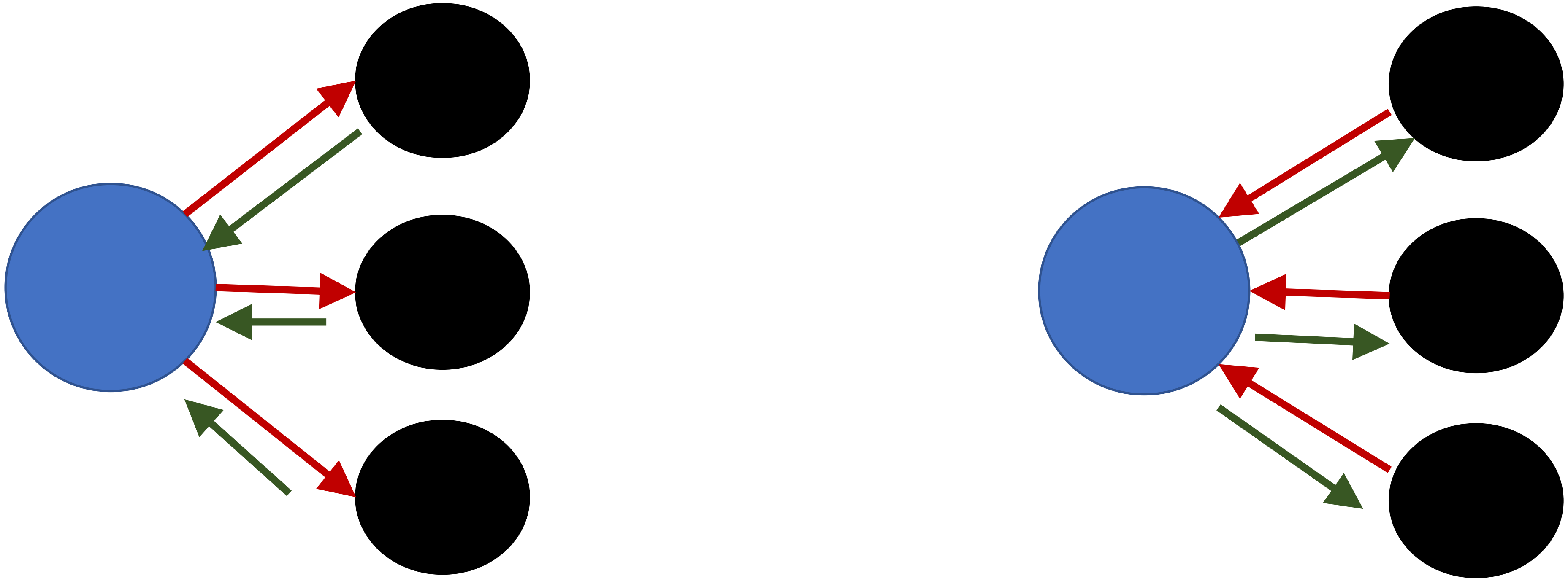
Routing



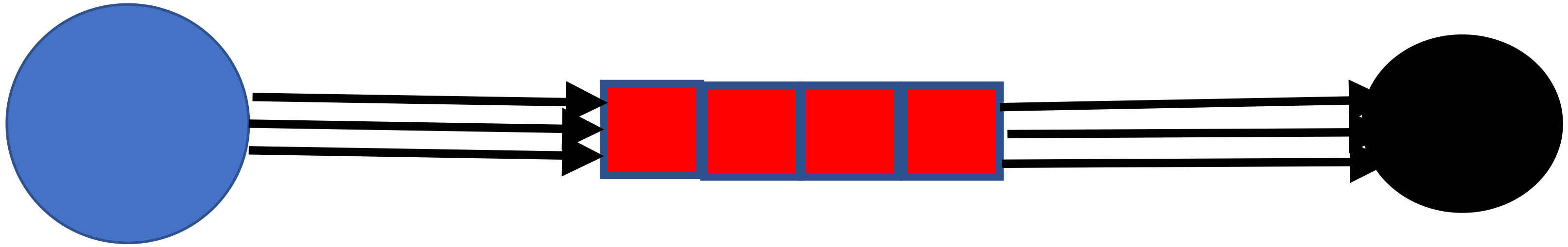
Topics



Publish-subscribe

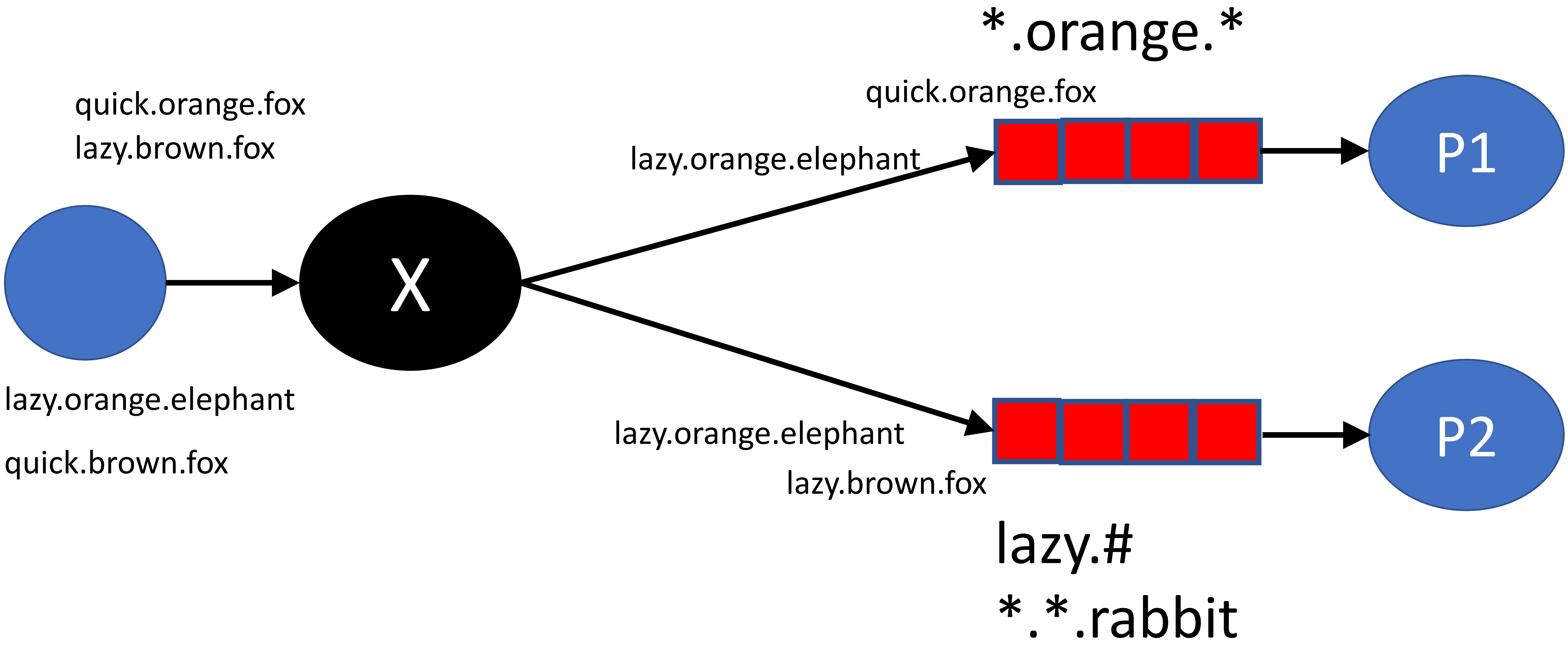


Message queue



An example of topic-based communication

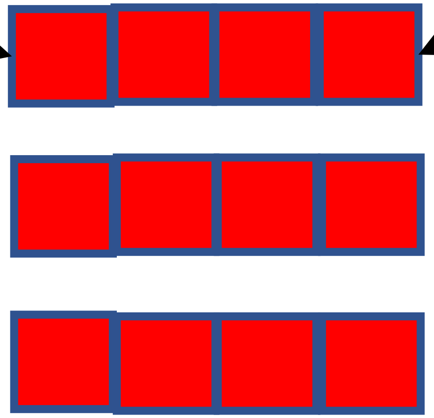
(adopted from https://www.rabbitmq.com/tutorials/tutorial-five-python.html)



Tampereen yliopisto
Tampere University

RabbitMQ – steps in practice

Connect
Create Channel
Send
Wait…
Close



Connect
Create Channel
Consume

https://www.rabbitmq.com/tutorials/tutorial-one-javascript.html

This tutorial assumes RabbitMQ is installed and running on localhost on standard port (5672). In case you use a different host, port or credentials, connections settings would require adjusting.

Tampereen yliopisto
Tampere University

Comparison

Tampereen yliopisto
Tampere University

Consequences

	Independent development	Independent deployment	Minimum centralized management
REST			
gRPC			
Message queue			

Tampereen yliopisto
Tampere University

s ces

Can be used in many ways

Designed for intependent

Standard ways to document
Designed for intependent

Practically none on top of Network infra

Practically none on top of Network infra

	Independent development	Independen deployment	Minimum ce manageme
REST			
gRPC			
Message queue			

No standards: need to be agreed on

The queue even suppors interrupts

The message queue needs to be maintained

Next exercise

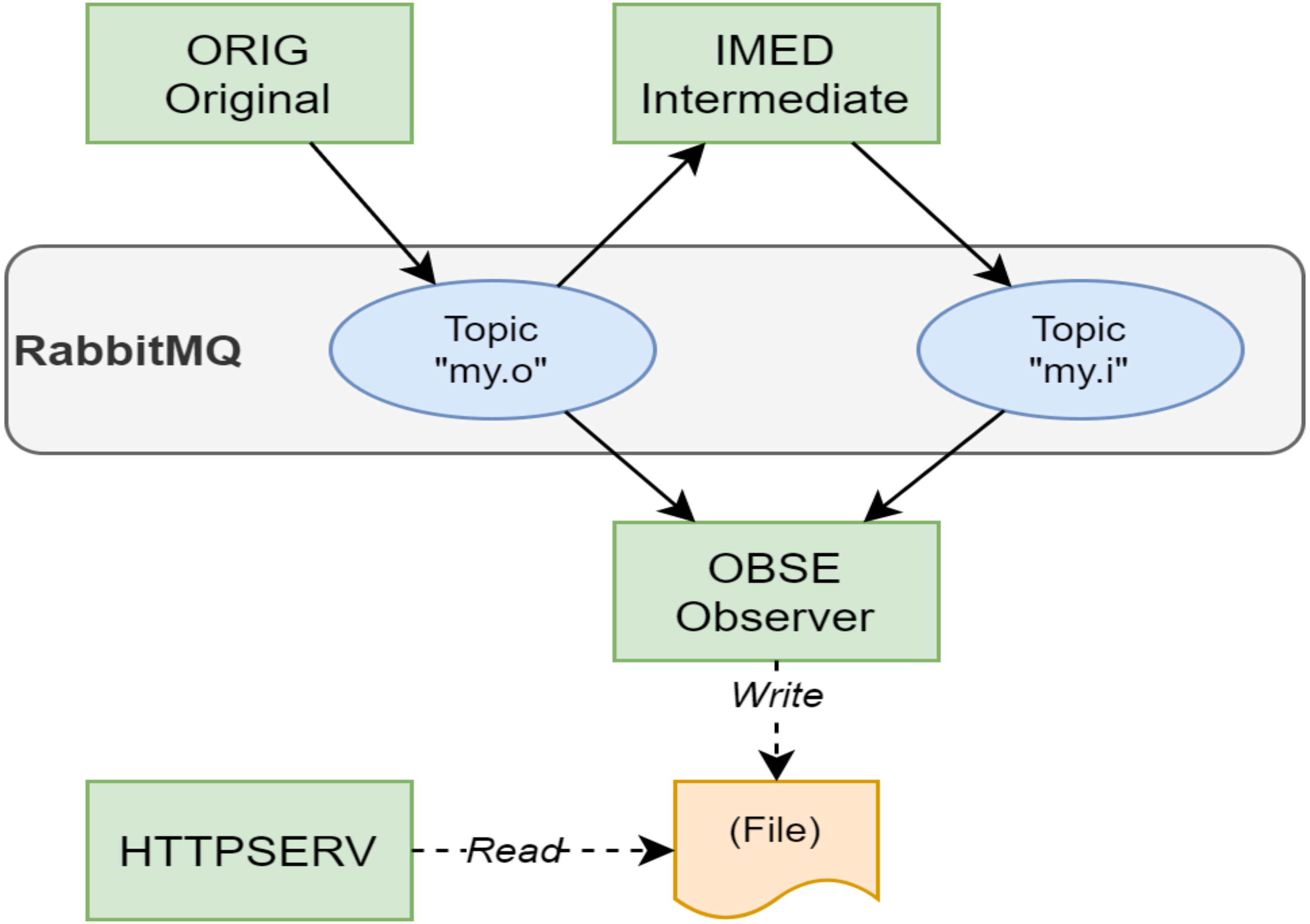
You create a bigger system of several processes and message queue infrastructure

Grading policy:

- maximum 6 points are given (total of the course will be about 50)
- missing the deadline: points reduced by 0.5 points / day
- how well the requirements are met: 2p
- following the good programming and docker practices: 2p
- quality of the document: 2p

Deadlines:

- for full points: 09.11
- for any points: 21.11


ORIG
Original
IMED
Intermediate
RabbitMQ
Topic
"my.o"
Topic
"my.i"
OBSE
Observer
Write
HTTPSERV
Read
(File)

Tampere University

Behavior

- ORIG publishes 3 messages to topic *my.o :*

  MSG_1

  (Wait for 3 seconds)

  MSG_2

  (Wait for 3 seconds)

  MSG_3

- IMED

  Every time IMED receives a message from topic my.o:

  IMED waits for 1 second

  After waiting, IMED publishes “Got {received message}” without quotes to topic my.i

  For example:

  ```
  Got MSG_1
  ```

- OBSE

  On any message from any of the topics:

  builds a string “{timestamp} Topic {topic}: {message}” without quotes

  {timestamp} must be in the format YYYY-MM-DDThh:mm:ss.sssZ (ISO 8601)

  Time zone is UTC

  {topic} is the topic that delivered the message

  {message} is the message body

  example:

  ```
  2020-10-01T06:35:01.373Z Topic my.o: MSG_1
  ```

  writes the string into a file in a Docker volume

  If OBSE is run multiple times, the file must be deleted/cleared on startup

- HTTPSERV

  When requested, returns content of the file created by OBSE (Nothing else)

  Port: 8080

  Example:

  ```
  2020-10-01T06:35:01.373Z Topic my.o: MSG_1
  2020-10-01T06:35:01.973Z Topic my.i: Got MSG_1
  ```

Returning

Source code of your application

Docker Compose file (YAML)

All Docker files

Any other files required to build and run the system

A document in which you cover at least

- Perceived (in your mind) benefits of the topic-based communication compared to request-response (HTTP)
- Your main learnings

Tampere University

Testing

```
$ git clone <the git url you gave>
$ docker-compose build --no-cache
$ docker-compose up -d
(Wait for at most 30 seconds...)
$ curl localhost:8080
<output should follow the requirements>
$ docker-compose down
```