

Communication. Distributed systems models. Fault tolerance

Course: Real-Time Backend

Lecturer: Gleb Lobanov

May, 2024



EPIC

Institute of Technology
Powered by <epam>

Contents

01 HTTP

02 RPC

03 Brokers

04 Comparison

05 The two generals problem

06 The Byzantine generals problem

07 System Models

08 Fault tolerance & availability

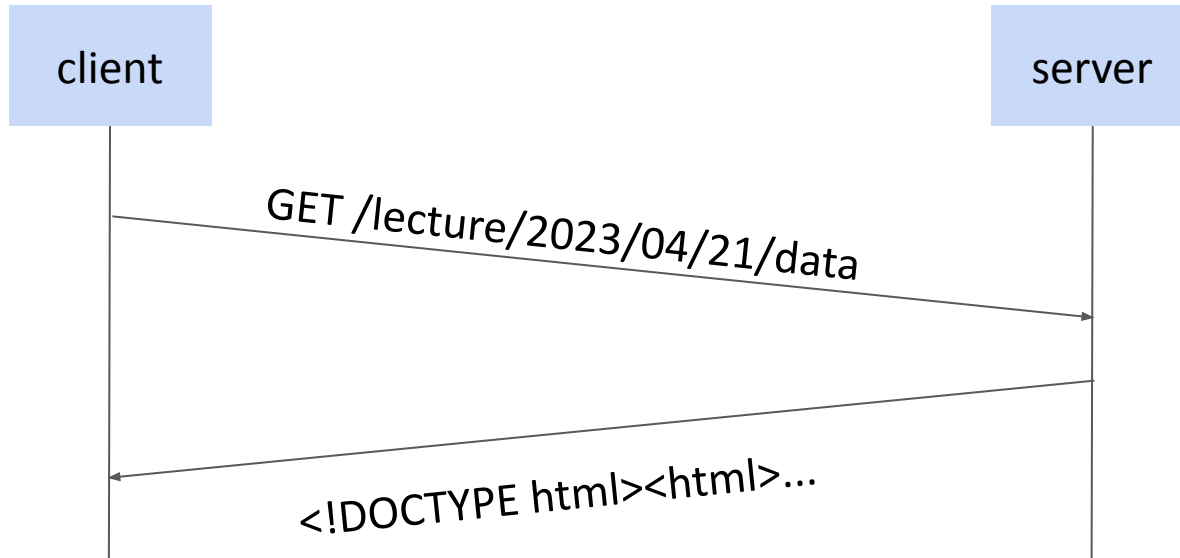
01

HTTP

In this section we will discuss “HTTP” and how is it working

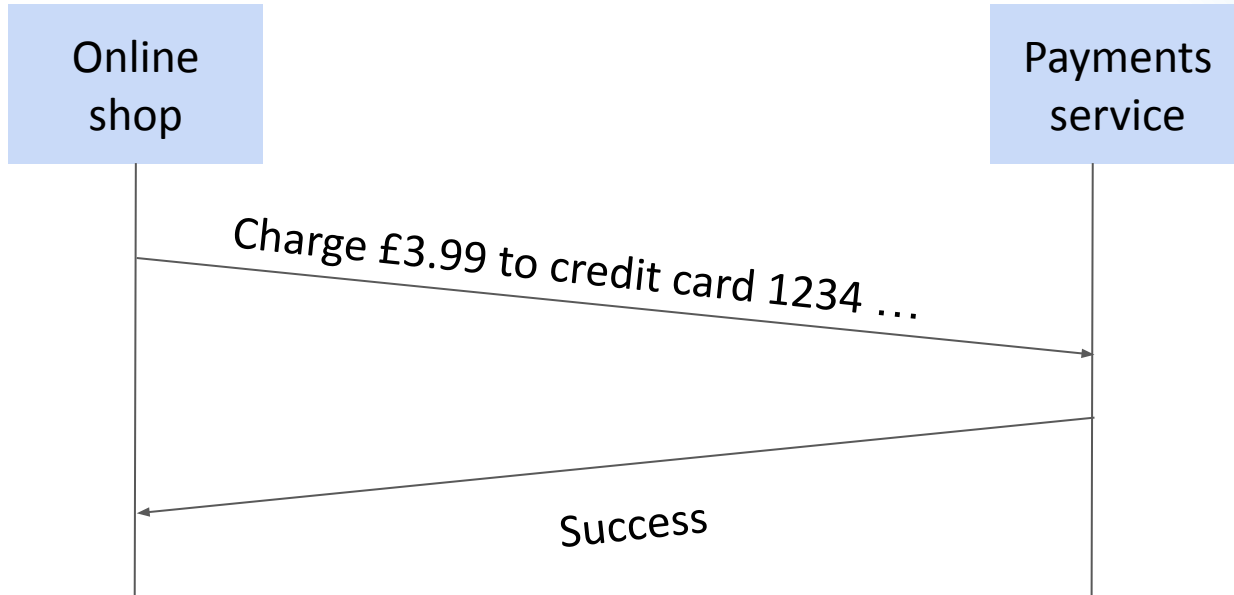
Client-server example: web

Time flows top to bottom



Client-server example: online payments

Time flows top to bottom



REST (representational state transfer)

- Communication is stateless (each request is self-contained and independent from other requests),
- Resources (objects that can be inspected and manipulated) are represented by URLs
- The state of a resource is updated by making a HTTP request with a standard method type, such as POST or PUT, to the appropriate URL.

APIs that adhere to these principles are called RESTful

**EPIC**Institute of Technology
Powered by ePam

Rest example in Javascript

```
Let args = {amount: 3.99, currency: 'GPG', /*...*/};
Let request = {
  method : 'POST',
  body: JSON.stringify(args),
  Headers: {'Content-Type': 'application/json'}
};

Fetch ('https://example.com/payments', request)
  .then((response)⇒ {
    if (response.ok) success (response.json());
    else failure (response.status); // server error
  })
  .catch ((error); //network error
});
```

02

RPC

In this section we will discuss “Remote procedure call” and how is it working

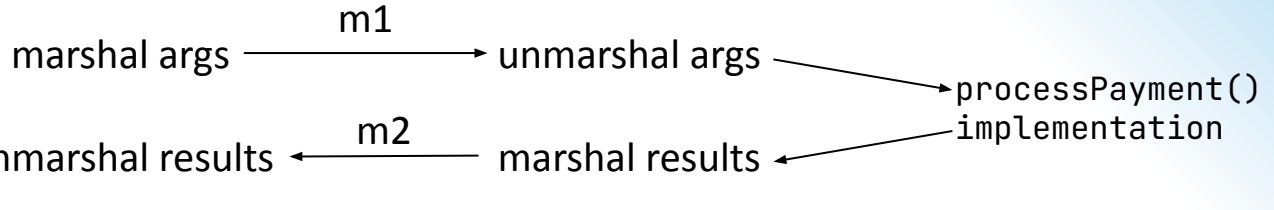
**EPIC**Institute of Technology
Powered by epan>

Online shop

RPC client

RPC server

Payment service

processPayment()
stub

Function returns

```
m1 = {
  "Request": "processPayment",
  "Card": {
    "Number": "1234567887654321",
    "expiryDate": "10/2024",
    "CVC": "123"
  },
  "amount": 3.99,
  "currency": "GBP"
}
```

```
m2 = {
  "Result": "success",
  "Id": "XP61hHw2Rvo"
}
```

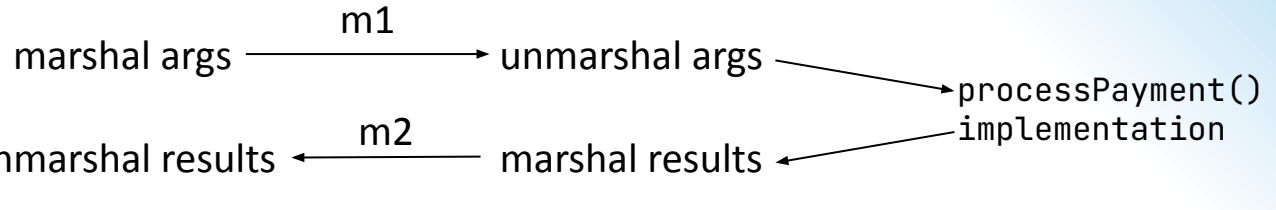
Online shop

RPC client

RPC server

Payment service

processPayment()
stub



Function returns


m1 =

```
{
  "Request": "processPayment",
  "Card": {
    "Number": "1234567887654321",
    "expiryDate": "10/2024",
    "CVC": "123"
  },
  "amount": 3.99,
  "currency": "GBP"
}
```



m2 =

```
{
  "Result": "success",
  "Id": "XP61hHw2Rvo"
}
```



RPC example

```
// online shop handling customer's card details
```

```
Card card = new Card ();
```

```
card.setCardNumber("1234 5678 8765 4321");
```

```
card.setExpiryDate("10/2024");
```

```
card.setCVC("123");
```

Implementation of this function is on
another node

```
Result result = paymentsService.processPayment(card, 3.99, Currency.GPB);
```

```
If (result.isSuccess()) {  
    fulfilOrder();  
}
```

Remote procedure call (RPC)

Ideally, RPC makes a call to a remote function look the same as a local function call.

In practice ...

- What if the service crashes during the function call?
- What if a message is lost?
- What if a message is delayed?
- If something goes wrong, is it safe to retry?

RPC in enterprise

“Service-oriented architecture” (SOA) / “microservices” :

Splitting a large software application into multiple services (on multiple nodes) that communicate via RPC.

Different services implement in different languages:

- **Interoperability:** datatype conversions
- **Interface Definition Language (IDL):** language-independent API specification

gRPC IDL definition

```
message PaymentRequest {  
    message Card {  
        required string cardNumber = 1;  
        optional int32 expiryMonth = 2;  
        optional int32 expiryYear = 3;  
        optional int32 CVC = 4;  
    }  
    enum Currency {GBP = 1; USD = 2;}  
  
    required Card card = 1;  
    required int64 amount = 2;  
    required Currency currency = 3;  
}
```

Message definition

```
message PaymentStatus {  
    required bool success = 1;  
    optional string errorMessage = 2;  
}
```

```
service PaymentService {  
    Rpc ProcessPayment (PaymentRequest) returns (PaymentStatus) {}  
}
```

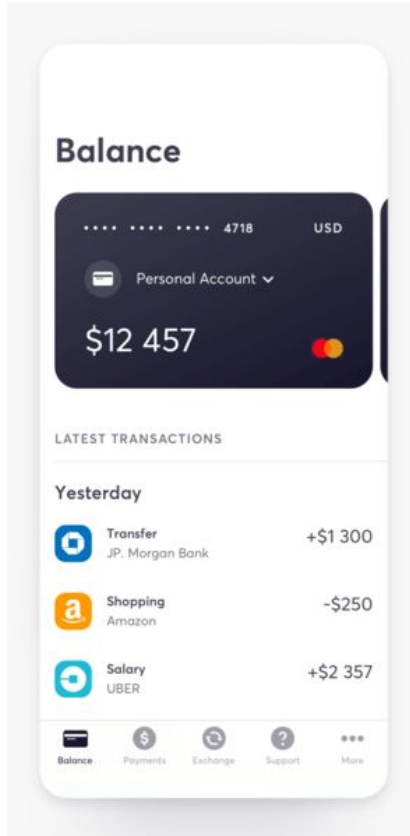
Service definition

03

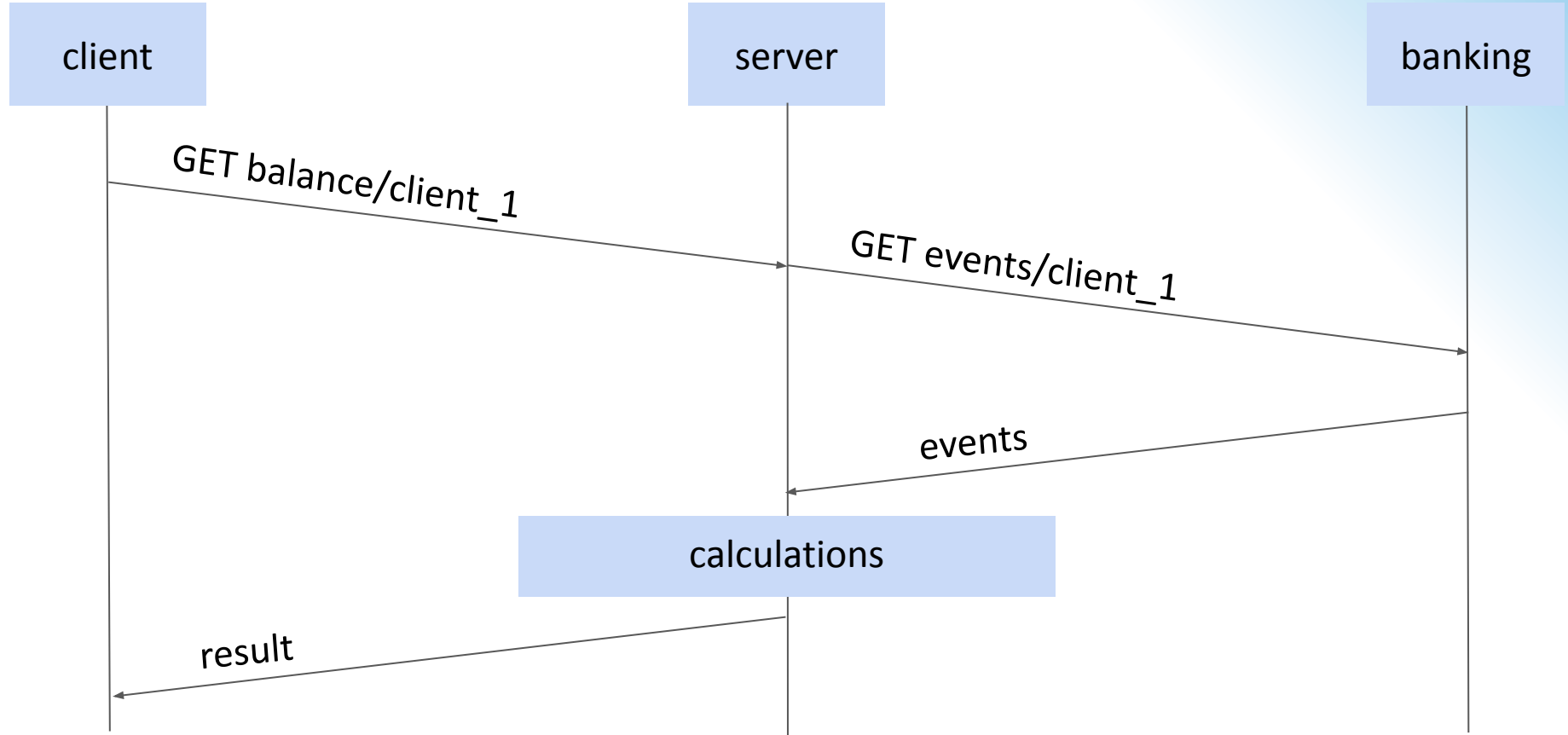
Brokers

In this section we will discuss what is brokers and how to use them)

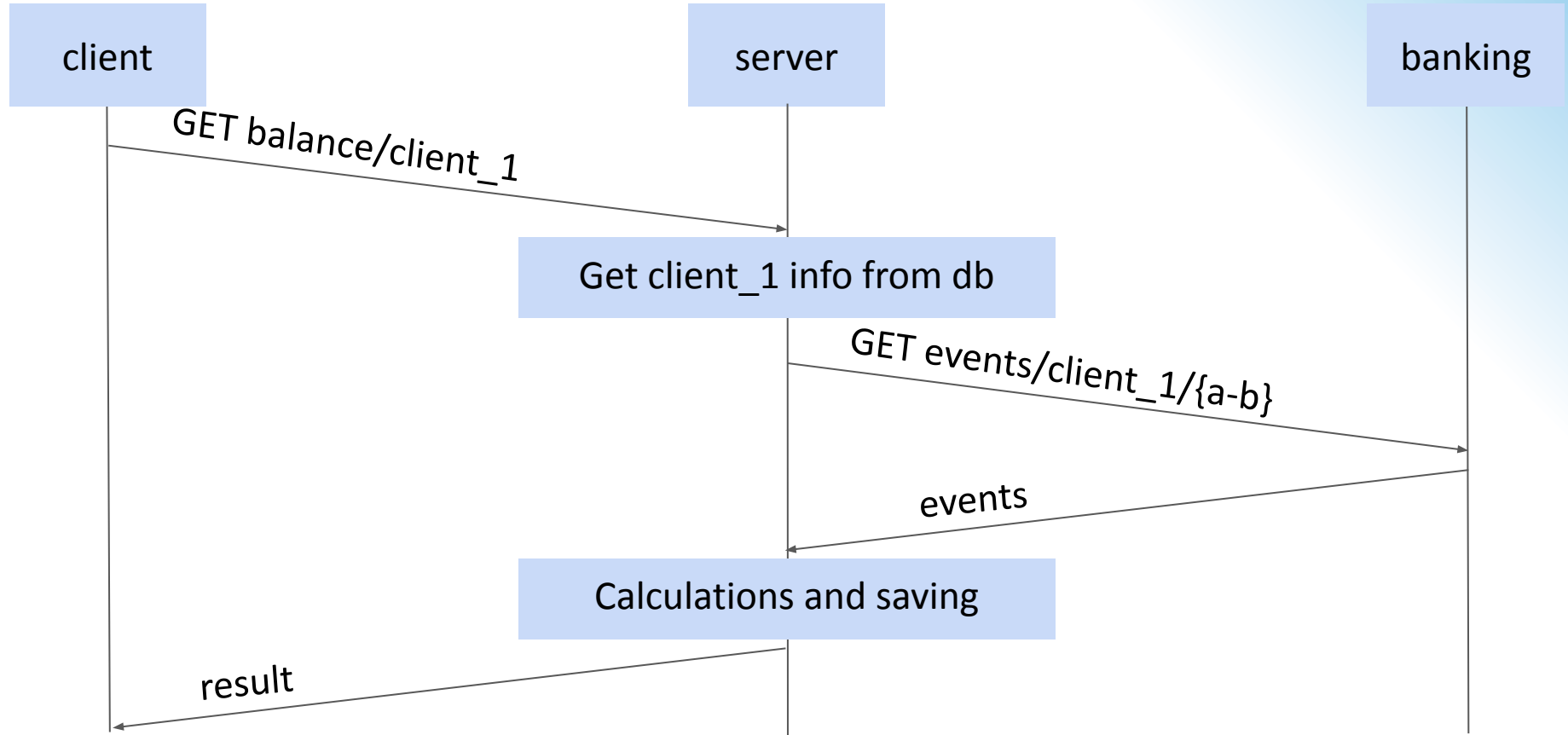
Banking



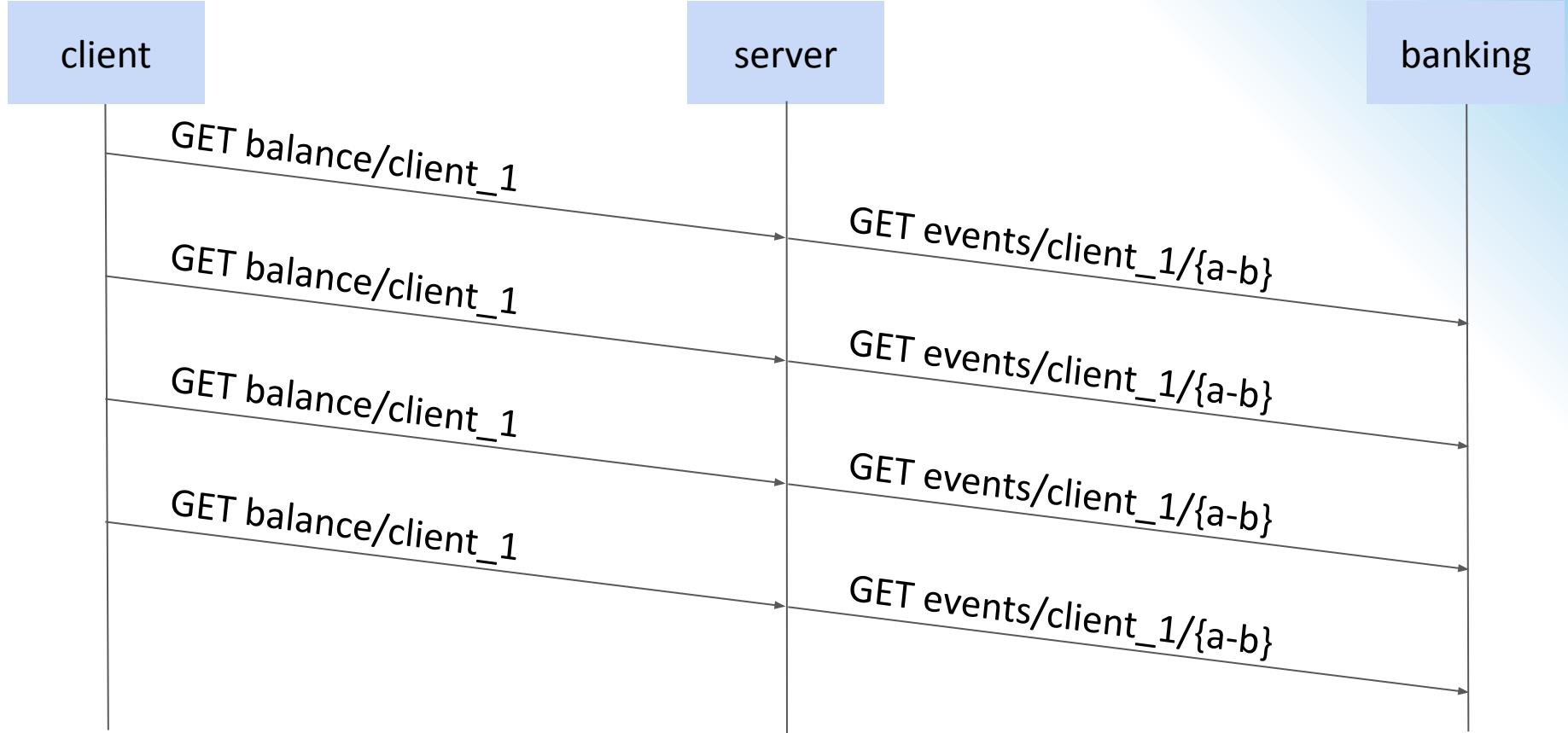
Business logic



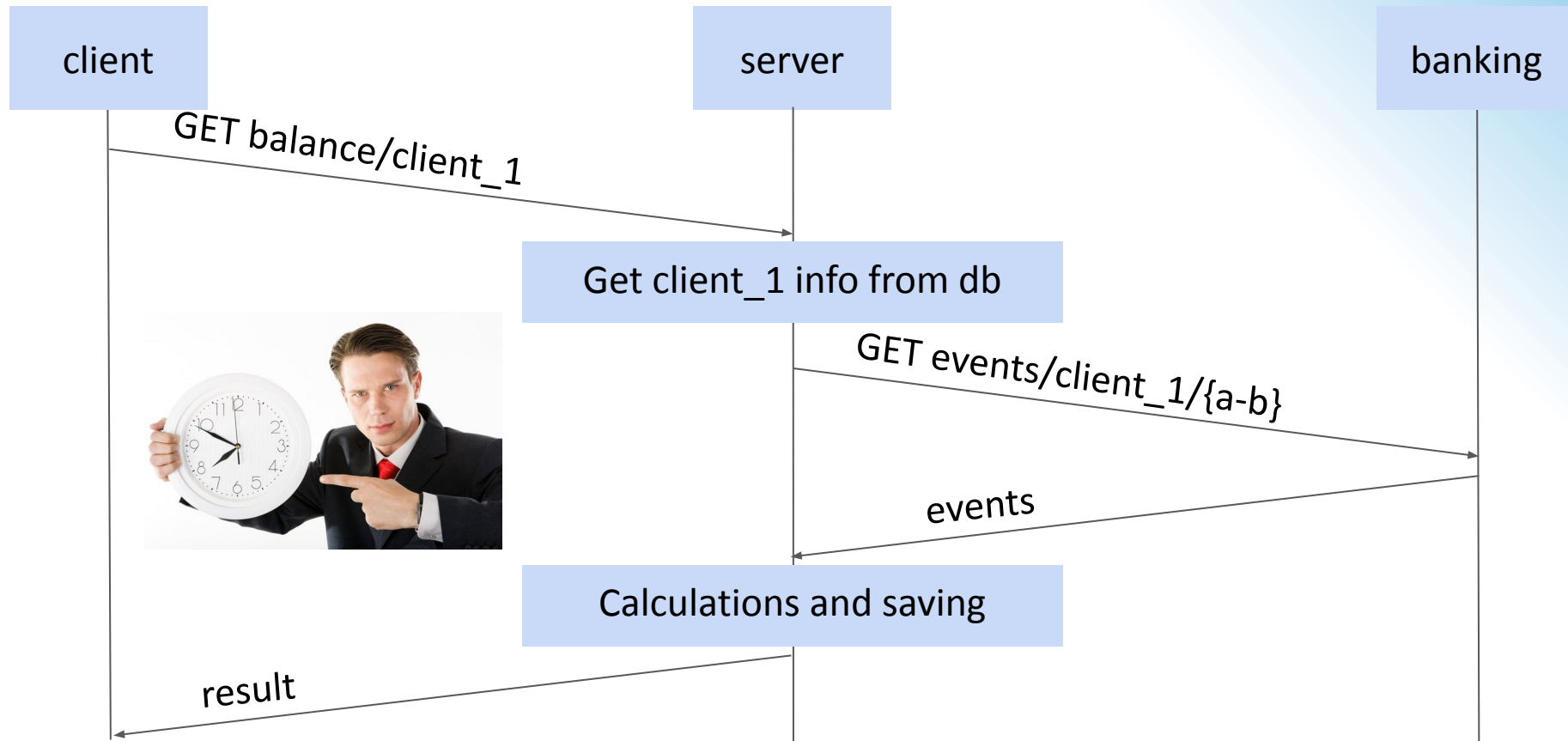
Business logic



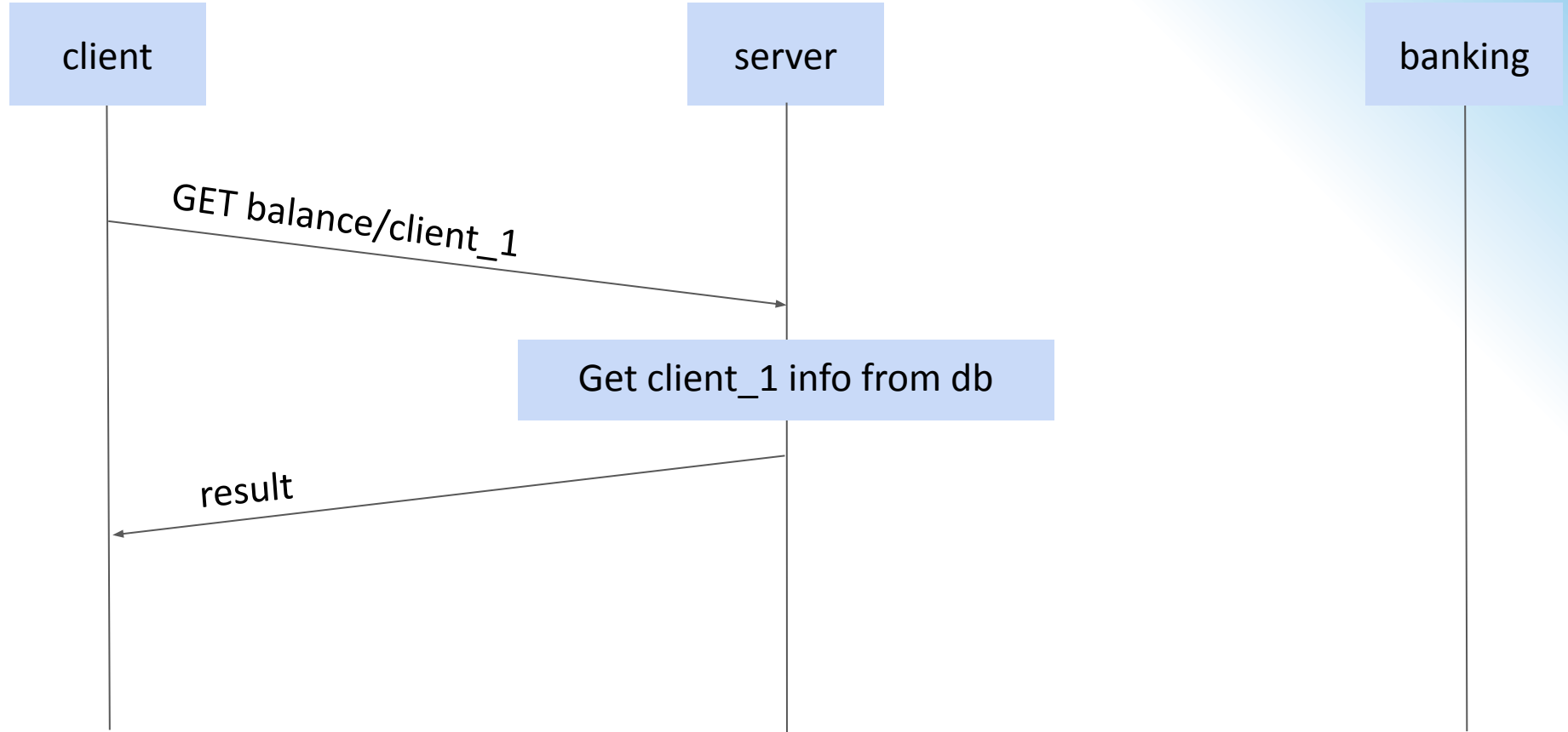
Problem



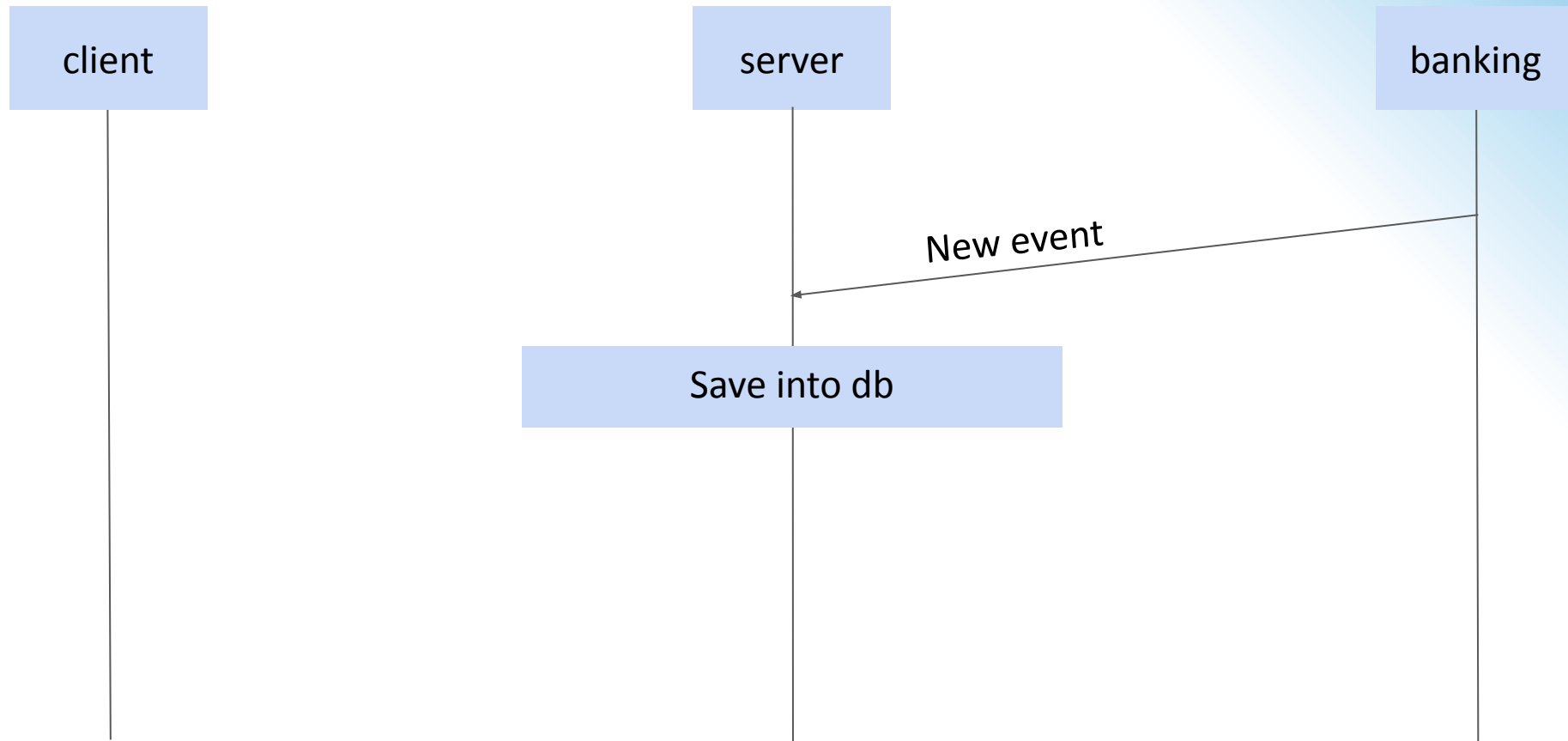
Problem



Response part



Filling part

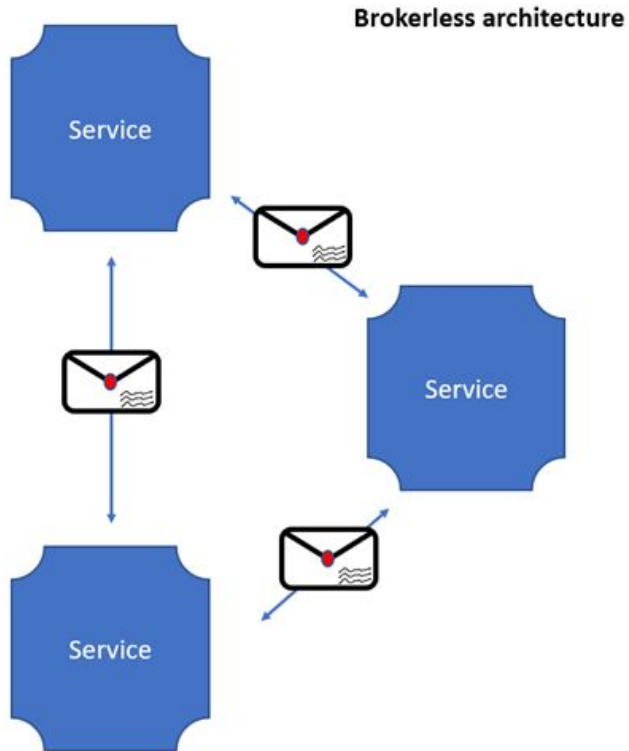




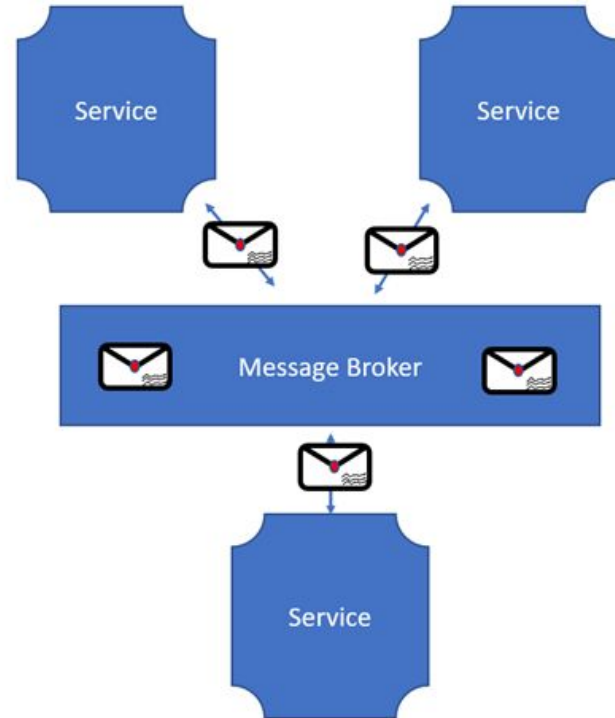
EPIC

Institute of Technology
Powered by ePam

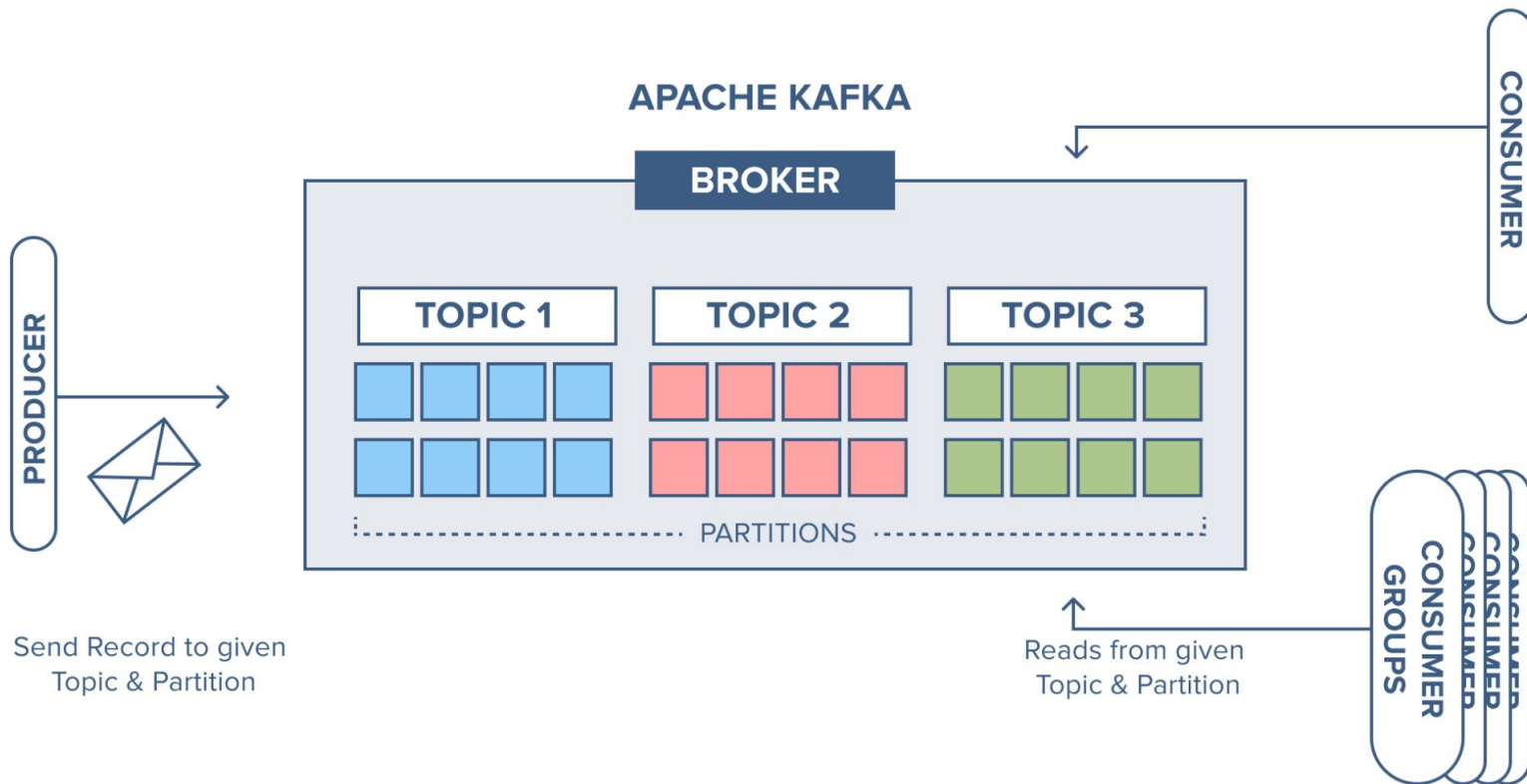
Also broker



Broker-based-architecture



Kafka

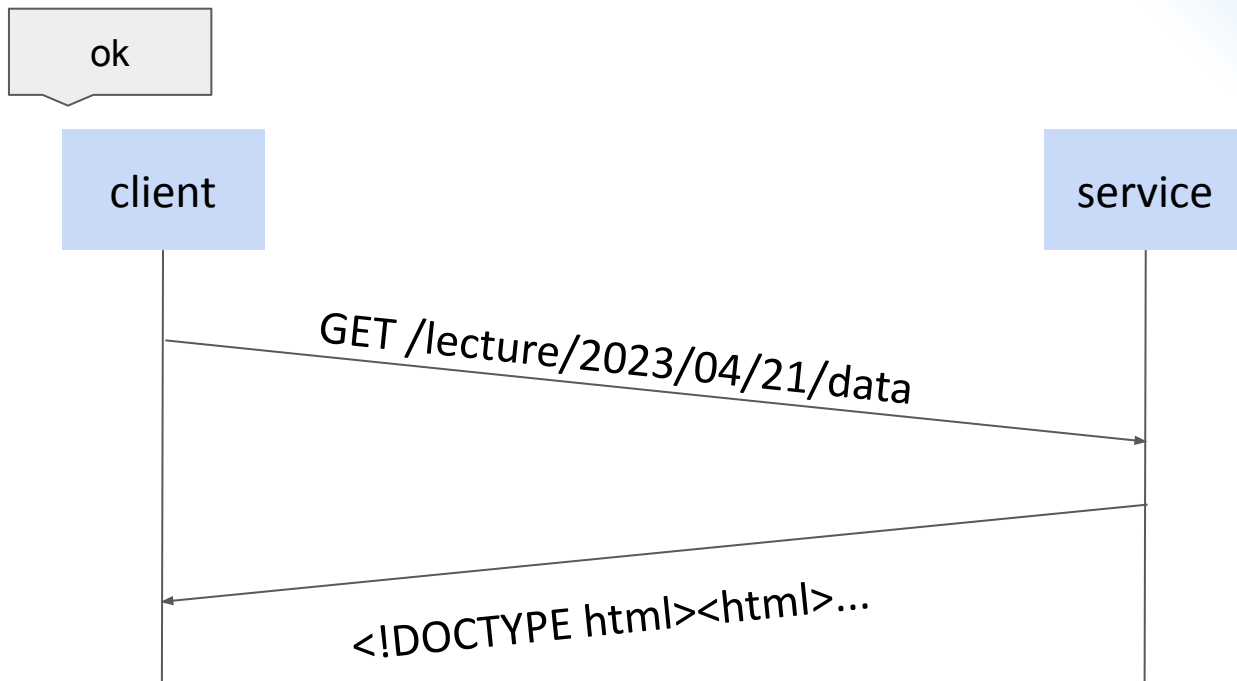


04

Comparison

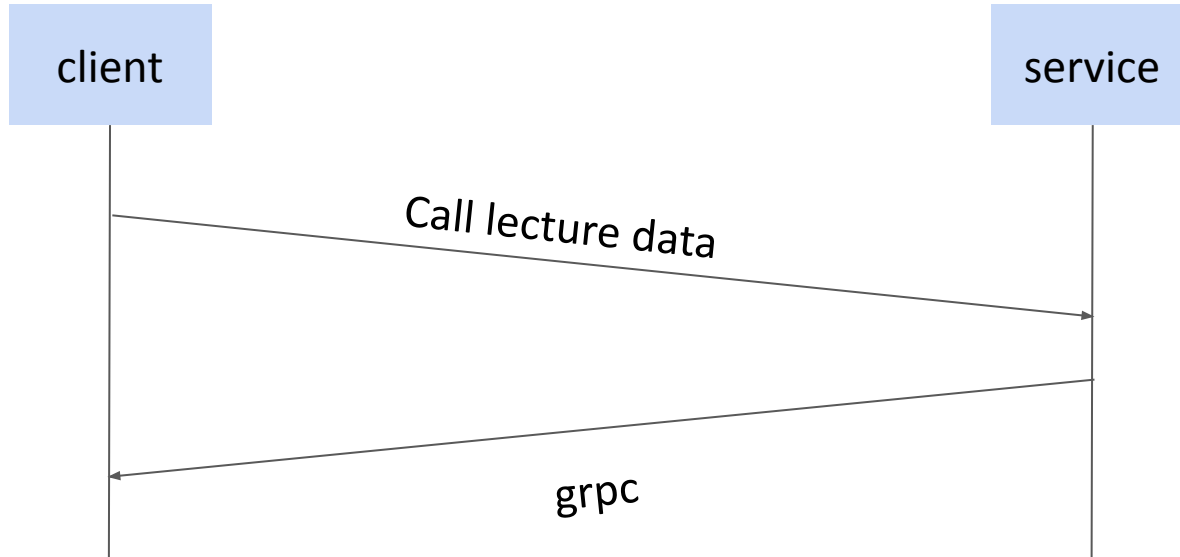
In this section we will compare all these methods

RPC vs HTTP



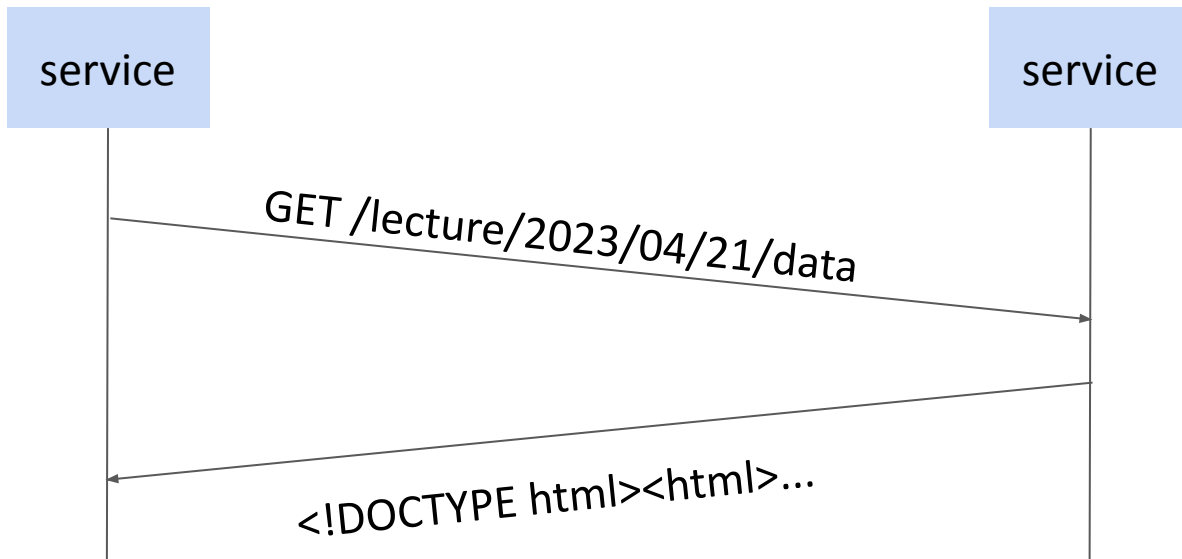
RPC vs HTTP

What is it

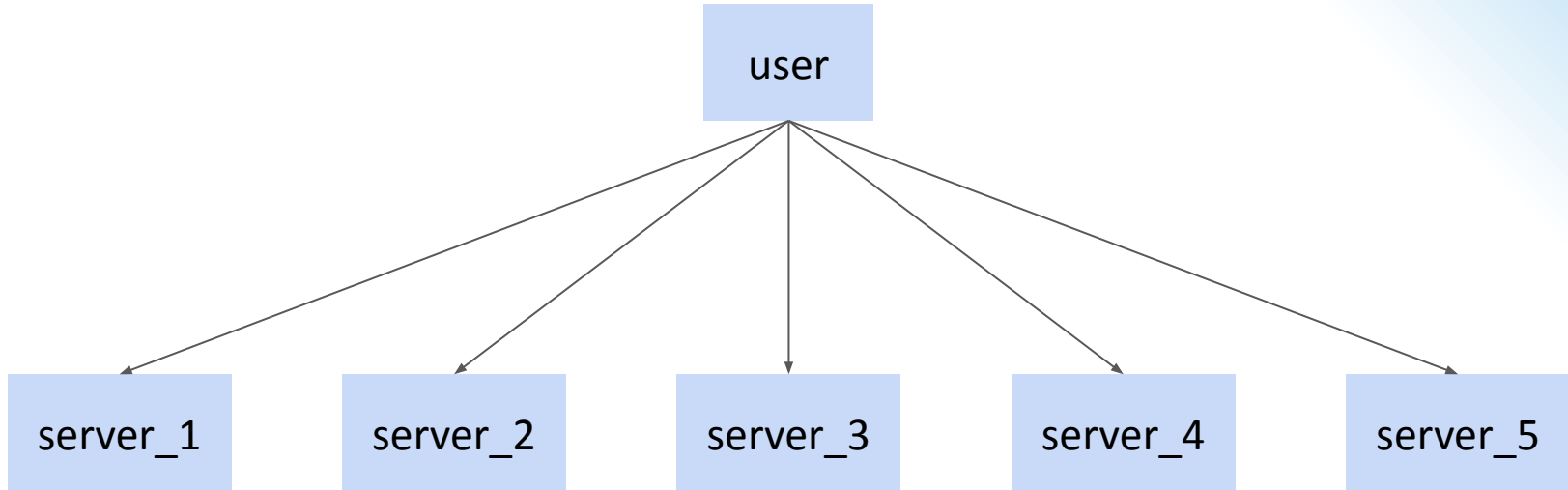


RPC vs HTTP

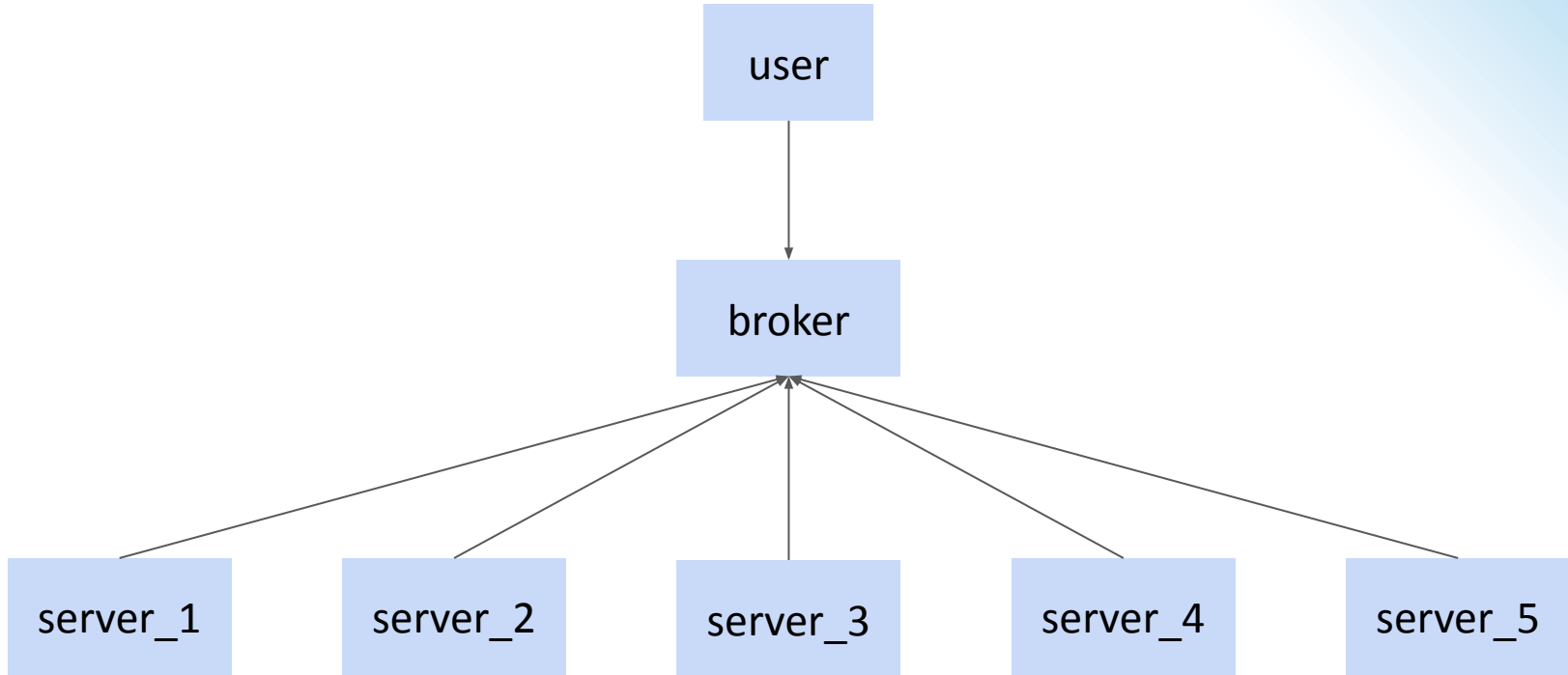
ok, but less comfort



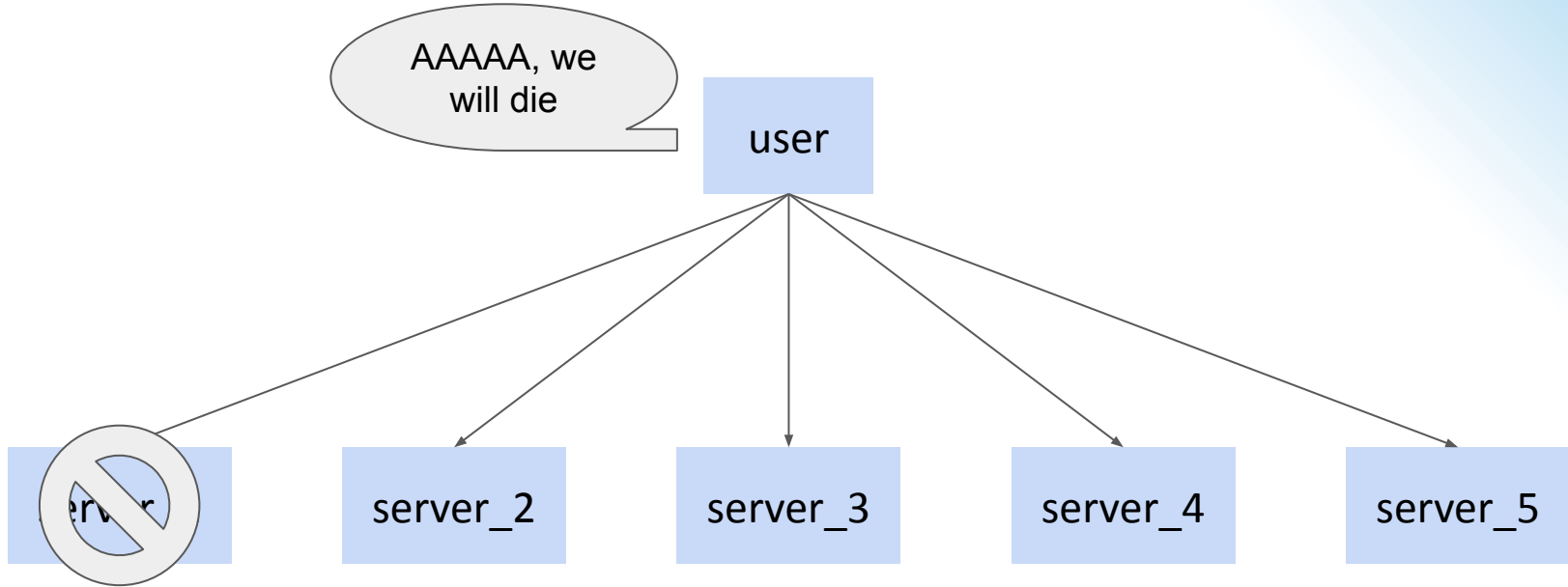
WO broker



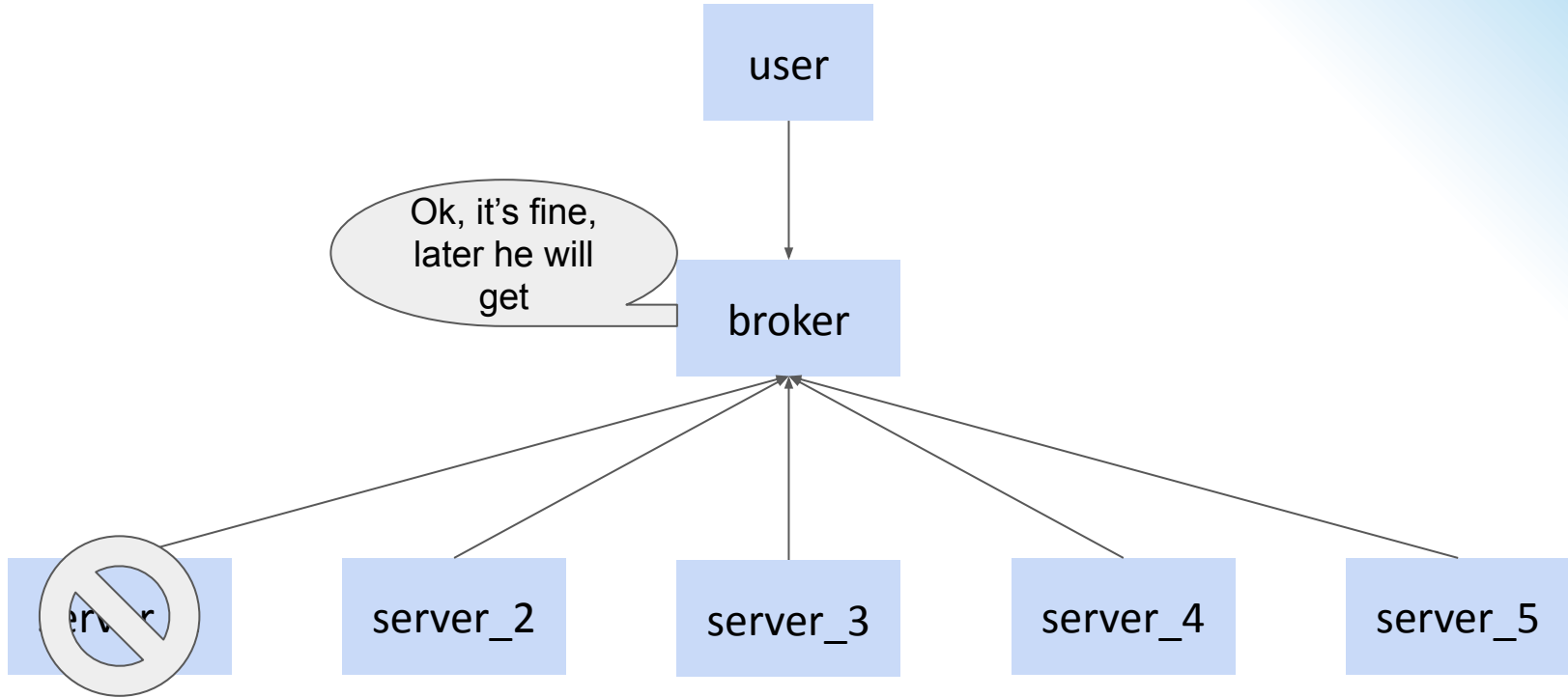
With broker



Server_1 is broken?



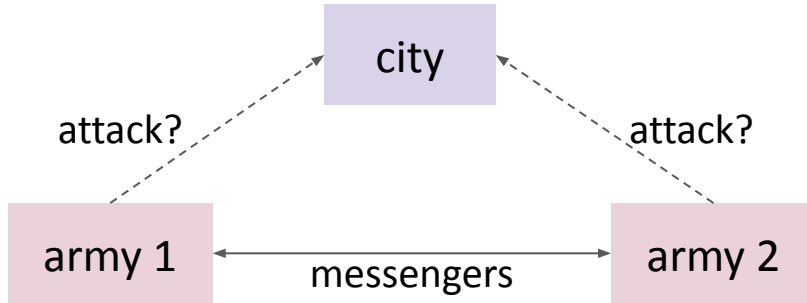
Server_1 is broken?



05

The two generals problem

The two generals problem

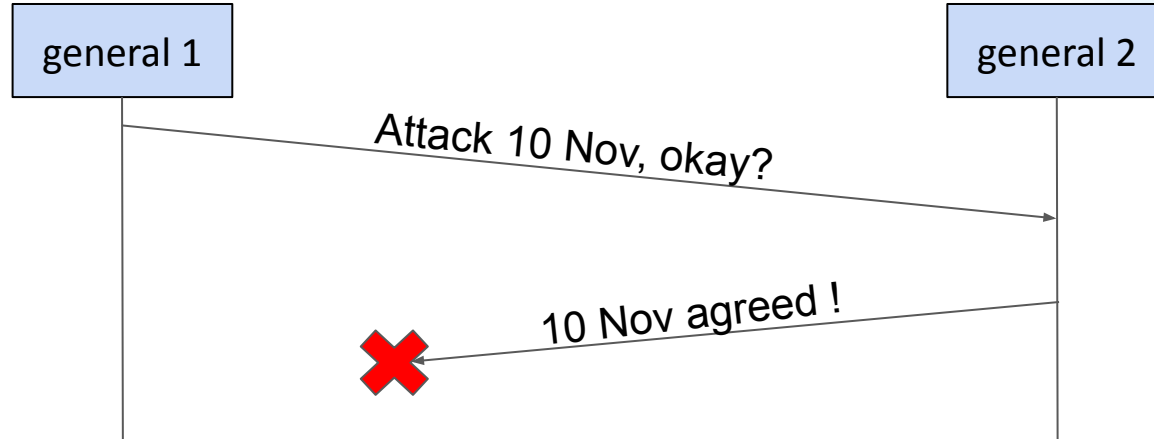


- Messengers can be captured (not delivered)
- Messages cannot be modified

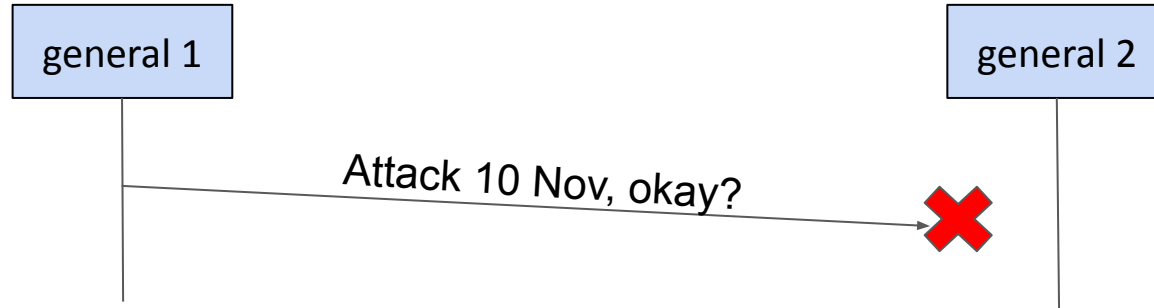
Army 1	Army 2	Outcome
Does not attack	Does not attack	Nothing happens
Attacks	Does not attack	Army 1 defeated
Does not attack	Attaks	Army 2 defeated
Attaks	Attacks	City captured

Desired: army 1 attacks *if and only if* army 2 attacks

The two generals problem



From general 1's point of view, this is indistinguishable from:

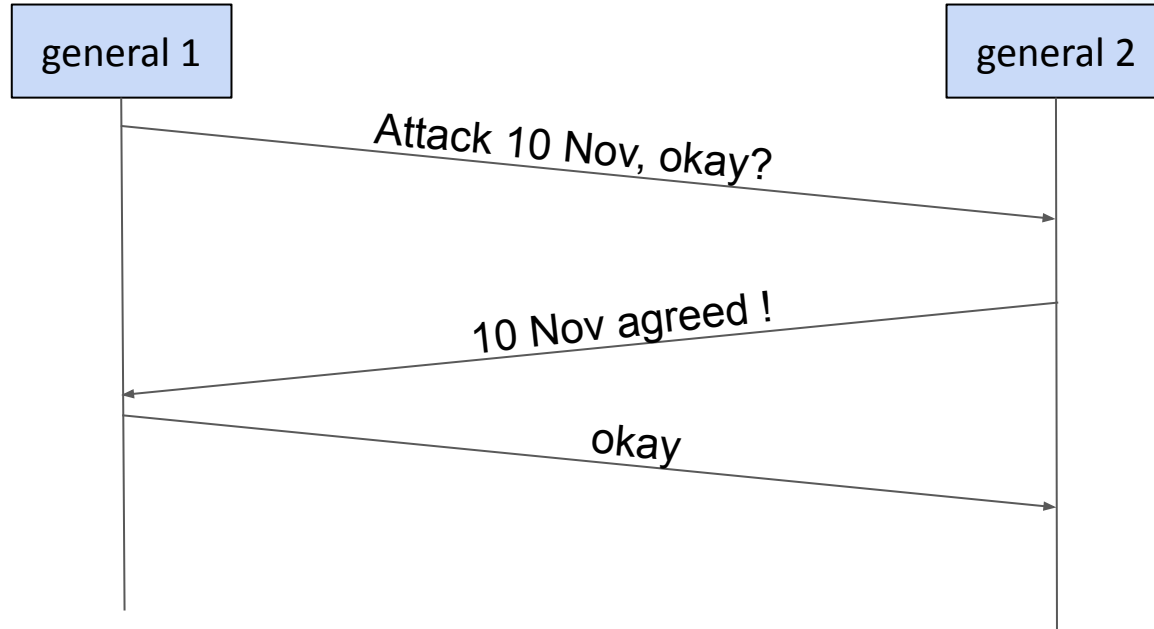


What should generals decide?

1. General 1 always attacks, even if no response is received?
 - Send lots of messengers to increase probability that one will get through
 - If all are captured, general 2 does not know about the attack, so general 1 loses

2. General 1 only attacks if positive response from general 2 is received?
 - Now general 1 is safe
 - But general 2 knows that general 1 will only attack if general 2's response gets through
 - Now general 2 is in the same situation as general 1 in option 1

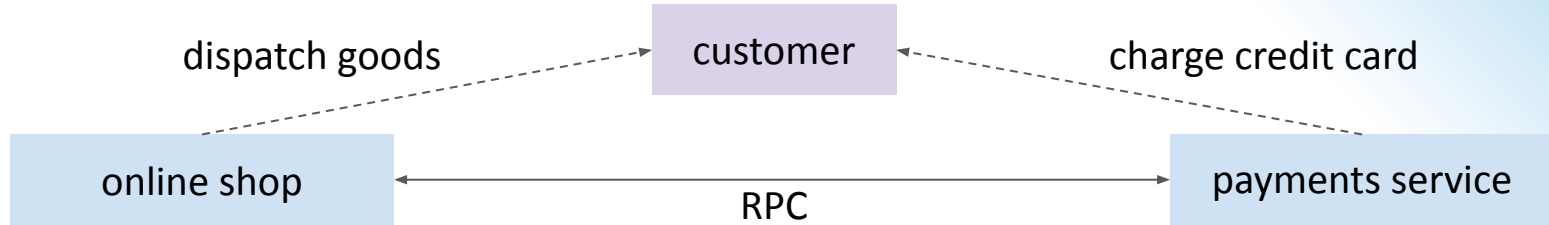
Perfect



What should generals decide?

No common knowledge: the only way of knowing something is to communicate it

The two generals problem applied

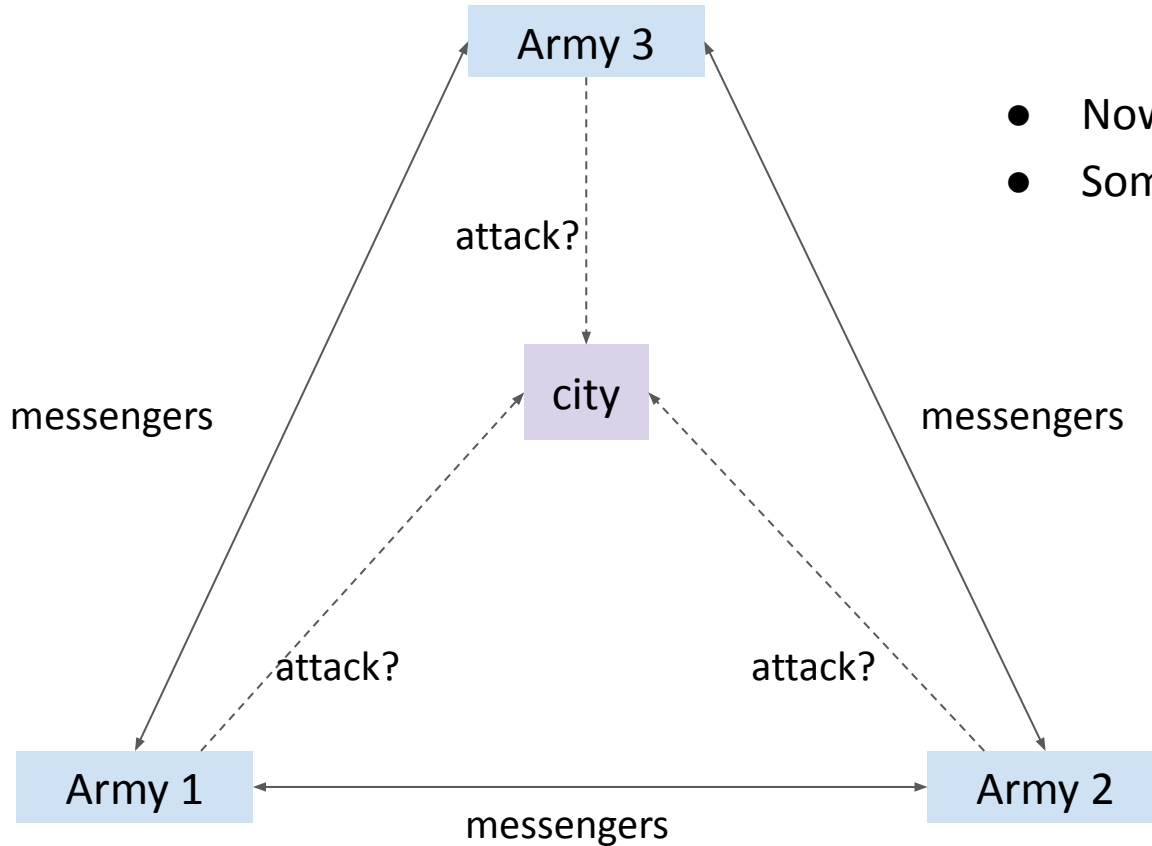


Online shop	Payments service	Outcome
Does not dispatch	Does not charge	Nothing happens
Dispatches	Does not charge	Shop loses money
Does not dispatch	charges	Customer complaint
Dispatches	charges	Everyone happy

06

The Byzantine generals problem

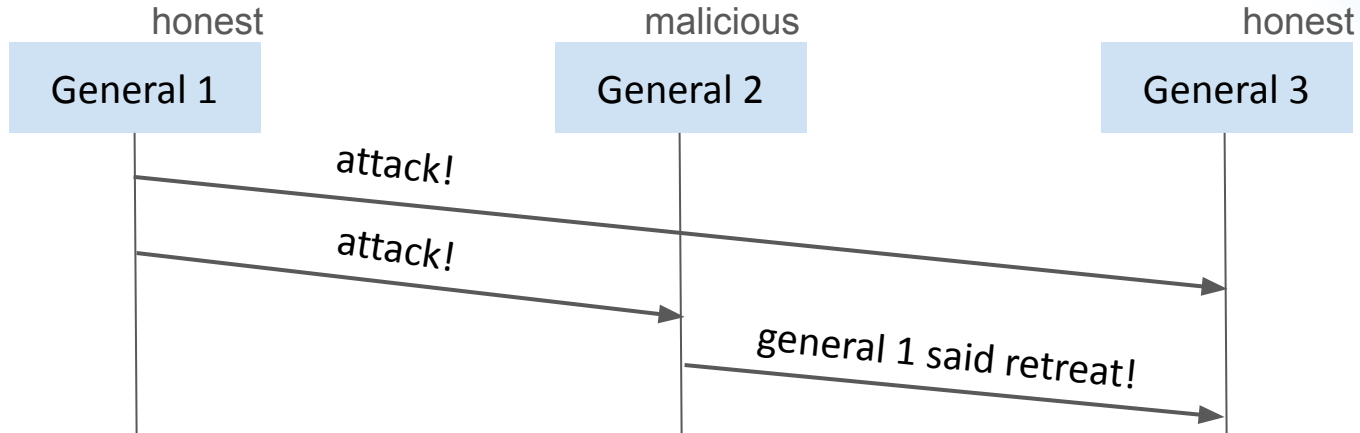
The Byzantine generals problem



- Now all messages are delivered
- Some generals might be traitors

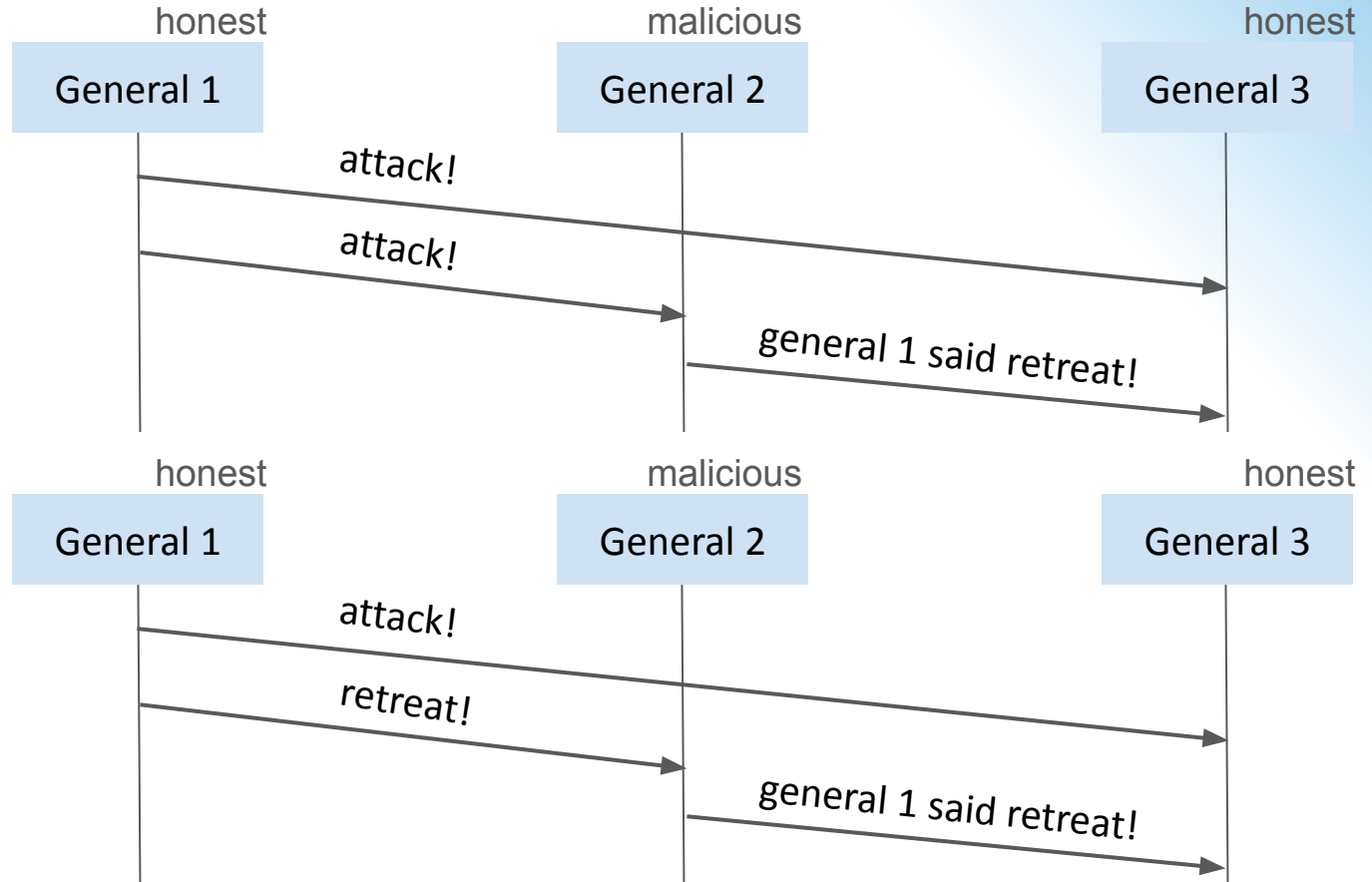
The Byzantine generals problem

Generals might be *honest* (sending valid messages) or *malicious* (sending any messages)



The Byzantine generals problem

From general 3's
point of view, this is
indistinguishable
from:



The Byzantine generals problem

- Each general is either malicious or honest
 - Up to f generals might be malicious
 - Honest generals don't know who the malicious ones are
 - The malicious generals may collude
 - Nevertheless, honest generals must agree on plan
-
- Theorem: need $3f + 1$ generals in total to tolerate f malicious generals (i.e. $< \frac{1}{3}$ may be malicious)
 - Cryptography (digital signatures) helps - but problem remains hard

The Byzantine generals problem: how to solve?

- Consider the case $f = 1$ & $n = 4$
- Each general sends 2 messages to everyone else
- Instead they need to understand if the total army size of honest generals is enough

The Byzantine generals problem: how to solve?

1. First step
 - a. Honest general: send ARMY_SIZE to everyone else
 - b. Malicious general: send any number to everyone else
2. Gather vectors from the data sent
3. Second step
 - a. Honest general: send current vector to everyone else
 - b. Malicious general: send any vector to everyone else
4. Use $N-1$ (ignore i -th general's vector) vectors to compute ARMY_SIZE(i) check if i -th number repeats $N - F - 1$ times

**EPIC**Institute of Technology
Powered by epan

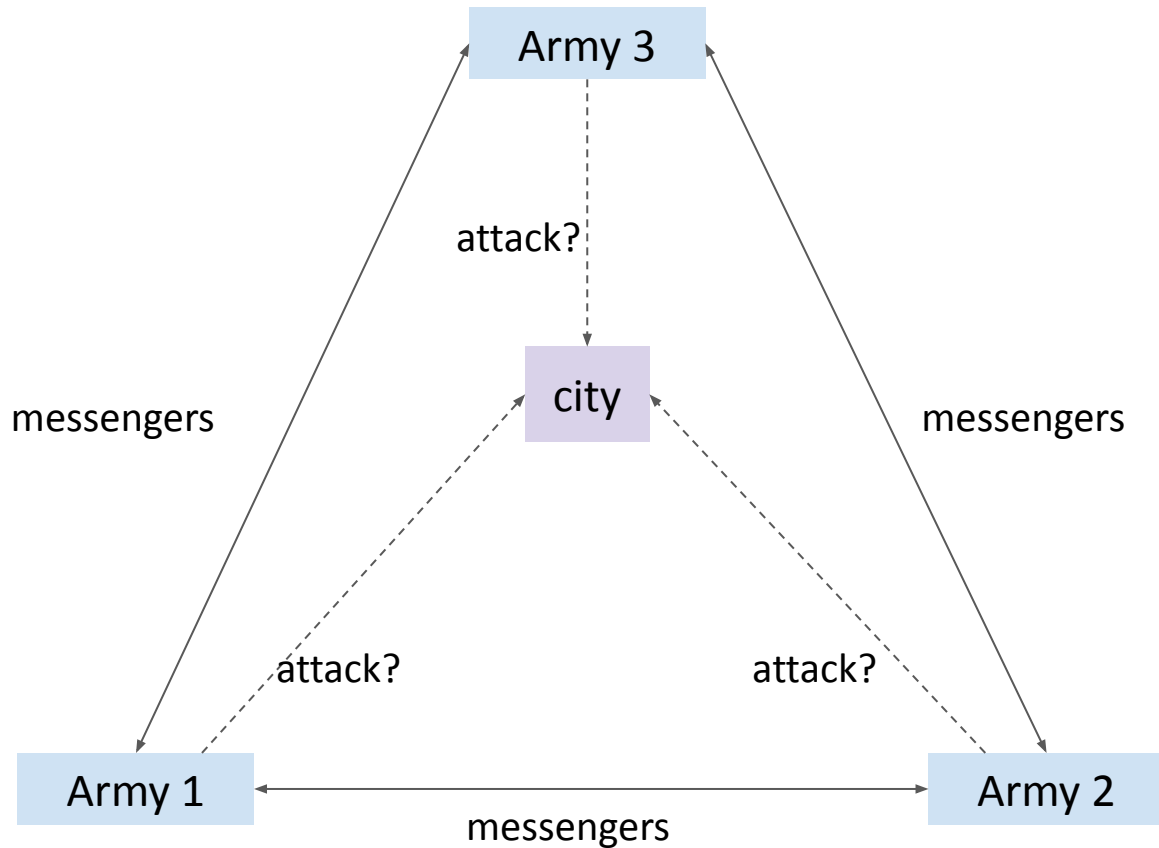
The Byzantine generals problem: example

- $F = 1$, $N = 4$, 3rd is a traitor
- $ARMY_SIZE(i) = i$
- After 1st step:
 - 1st: (1,2,a,4)
 - 2nd: (1,2,b,4)
 - 3rd: (1,2,3,4)
 - 4th: (1,2,c,4)
- After 2nd step:
 - 1st: (1,2,a,4), (1,2,b,4), (x,y,z,l), (1,2,c 4)
 - 2nd: (1,2,a,4), (1,2,b,4), (x1,y2,z2,l2), (1,2,c 4)
 - 3rd: (1,2,a,4), (1,2,b,4), (x2,y2,z2,l2), (1,2,c 4)
 - 4th: (1,2,a,4), (1,2,b,4), (x3,y3,z3,l3), (1,2,c 4)

The Byzantine generals problem: example

- Final decision (1st general):
 - $(1,2,a,4), (1,2,b,4), (x,y,z,l), (1,2,c,4)$
 - Knows own army size
 - 2nd army: $f(2,y,2) = 2$
 - 3rd army: $f(a,b,c) = ?$
 - 4th army: $f(4,4,l) = 4$

The Byzantine generals problem: application



Who can trust whom?

07

System Models

System Models

We have seen two thought experiments:

- Two generals problem: a model of networks
- Byzantine generals problems: a model of node behaviour

In real systems, both nodes and networks may be faulty!

Capture assumptions in a **system model** consisting of:

- Network behavior (e.g. message loss)
- Node behaviour (e.g. crashes)
- Timing behaviour (e.g. latency)

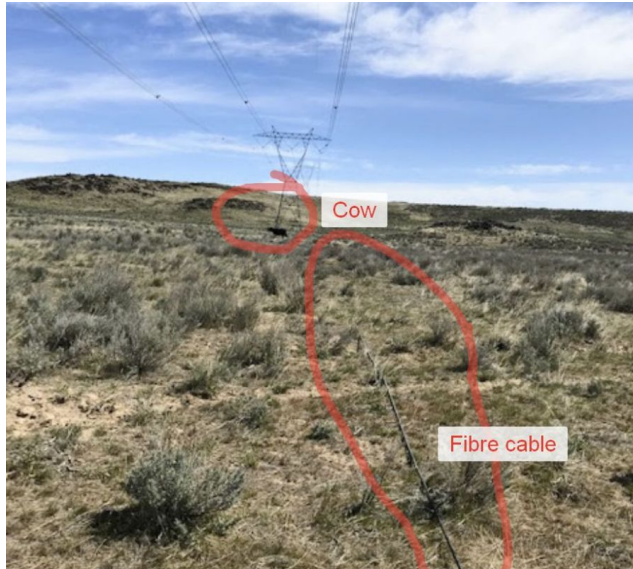
Choice of models for each of these parts.



EPIC

Institute of Technology
Powered by <epam>

Networks are unreliable



<https://twitter.com/uhoelzle/status/1263333283107991558>

On land, cows step on the
cables



<https://slate.com/technology/2014/08>

In the sea, sharks bite fibre optic
cables

System model: network behaviour

Assume bidirectional **point-to-point** communication between two nodes, with one of:

- **Reliable** (perfect) links:

A message is received if and only if it is sent.

Messages may be reordered.

- **Fair-loss** links:

Messages may be lost, duplicated, or reordered.

If you keep retrying, message eventually gets through.

- **Arbitrary** links (active adversary):

A malicious adversary may interfere with messages (eavesdrop, modify, drop, spoof, replay).

Network partition: some links dropping/delaying all messages for extended period of time

System model: node behaviour

Each node executes a specified algorithm, assuming one of the following:

- **Crash-stop** (fail-stop):

A node is faulty if it crashes (at any moment).

After crashing, it stops executing forever.

- **Crash-recovery** (fail-recovery):

A node may crash at any moment, losing its in-memory state. It may resume executing sometime later. Data stored on disk survives the crash.

- **Byzantine** (fail-arbitrary):

A node is faulty if it deviates from the algorithm. Faulty nodes may do anything, including crashing or malicious behaviour.

A node that is not faulty is called “**correct**”

System model: synchrony (timing) assumptions

Assume one of the following for network and nodes:

- **Synchronous:**

Message latency no greater than a known upper bound.

Nodes execute algorithm at a known speed.

- **Partially synchronous:**

The system is asynchronous for some finite (but unknown) periods of time, synchronous otherwise.

- **Asynchronous:**

Messages can be delayed arbitrarily.

Nodes can pause execution arbitrarily.

No timing guarantees at all.

Note: other parts of computer science use the terms “synchronous” and “asynchronous” differently.

Violations of synchrony in practise

Networks usually have quite predictable latency, which can occasionally increase:

- Message loss requiring retry
- Congestion / contention causing queueing
- Network / route reconfiguration

Nodes usually execute code at a predictable speed, with occasional pauses:

- Operating system scheduling issues, e.g. priority inversion
- Stop-the-world garbage collection pauses
- Page faults, swap, thrashing

Real-time operating systems (RTOS) provide scheduling guarantees, but most distributed systems do not use RTOS

Systems models summary

For each of the three parts, pick one:

- Network:

Reliable, fair-loss, or arbitrary

- Nodes:

Crash-stop, crash-recovery, or Byzantine

- Timing:

Synchronous, partially synchronous, or asynchronous

This is the basis for any distributed algorithm.

If your assumptions are wrong, all bets are off!

08

Fault tolerance & availability

Availability

Online shop wants to sell stuff 24/7!

Service unavailability = downtime = losing money

Availability = uptime = fraction of time that a service is functioning correctly

- “Two nines” = 99% up = down 3.7 days/year
- “Three nines” = 99.9% up = down 8.8 hours/year
- “Four nines” = 99.99% up = down 53 minutes/year
- “Five nines” = 99.999% up = down 5.3 minutes/year

Service-Level Objective (SLO):

E.g. “99.9% of request in a day get a response in 200 ms”

Service-Level Agreement (SLA):

Contact specifying some SLO, penalties for violation

High availability: fault tolerance required

Failure: system as a whole isn't working

Fault: some part of the system isn't working

- Node fault: crash (crash-stop/crash-recovery), deviating from algorithm (Byzantine)
- Network fault: dropping or significantly delaying

Fault tolerance: system as a whole continues working, despite faults (up to some maximum number of faults)

Single point of failure (SPOF): node/network link whose fault leads to failure

Detecting failures

Failure detector:

Algorithm that detects whether another node is faulty

Perfect failure detector:

Labels a node as faulty if and only if it has crashed

Typical implementation for crash-stop/crash-recovery: send message, await response, label node as crashed if no reply within some timeout

Problem:

Cannot tell the difference between crashed node, temporarily unresponsive node, lost message, and delayed message

Detecting failures in partially synchronous systems

Perfect timeout-based failure detector exists only in a synchronous crash-stop system with reliable links.

Eventually perfect failure detector:

- May temporarily label a node as crashed, even though it is correct
- May temporarily label a node as correct, even though it has crashed
- But eventually, labels a node as crashed if and only if it has crashed

Reflects fact that detection is not instantaneous, and we may have spurious timeouts

That's All Folks!