

# Distributed Systems

(The foundations/theory part of the course)

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>



This work is published under a  
Creative Commons BY-SA license.



TUM School of Computation, Information and Technology  
Technical University of Munich

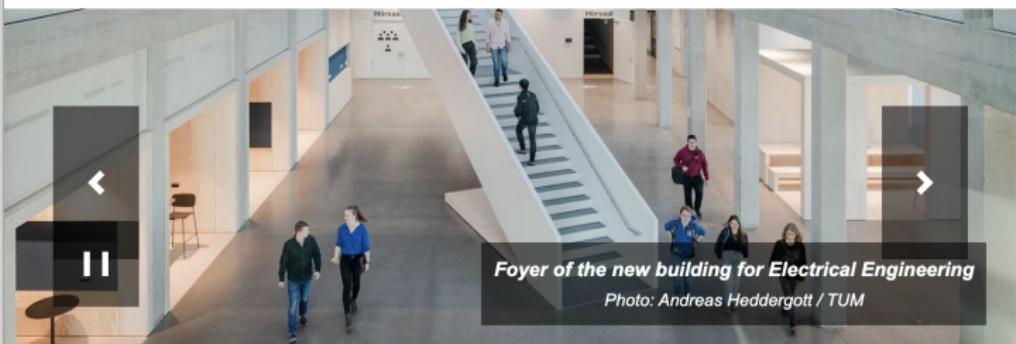


Home

Studies

Research

School



# School of Computation, Information and Technology



# Client-server example: the web

Time flows from top to bottom.



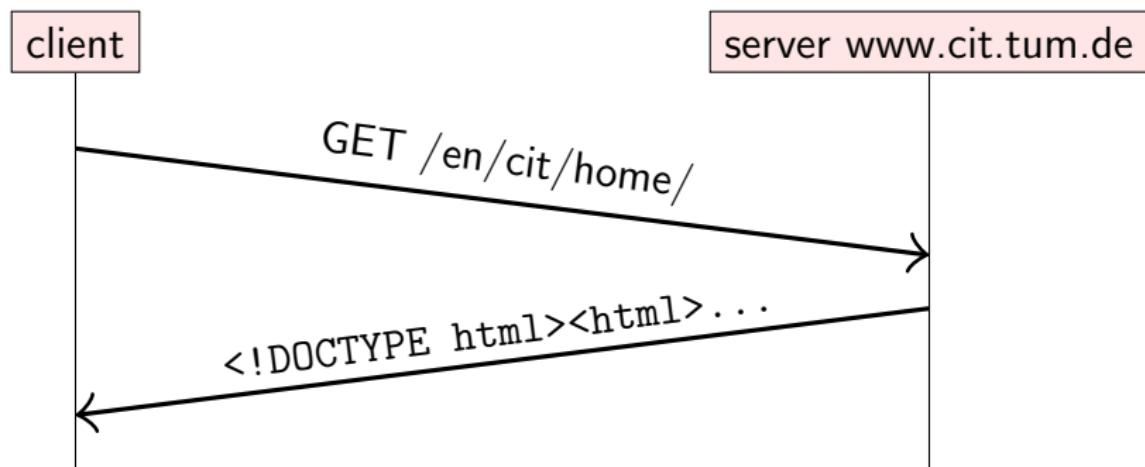
# Client-server example: the web

Time flows from top to bottom.



# Client-server example: the web

Time flows from top to bottom.



Charles 4.6.2 - Session 1

Code	Method	Host	Path	Duration	Size	Status	Info
200	GET	www.cit.tum.de	/en/cit/home/	366 ms	10.53 KB	Complete	

Filter:  Focused  Settings

Overview **Contents** Summary Chart Notes

GET /en/cit/home/ HTTP/1.1  
Host www.cit.tum.de  
Connection keep-alive  
Cache-Control max-age=0

Headers Cookies Raw

```
1 <!DOCTYPE html>
2 <html dir="ltr" lang="en" class="no-js">
3 <head>
4
5 <meta charset="utf-8">
6 <!--
7 This website is powered by TYPO3 - inspiring people to share!
8 TYPO3 is a free open source Content Management Framework initially created by Kasper Skaarhoj and licensed under GN
9 TYPO3 is copyright 1998-2022 of Kasper Skaarhoj. Extensions are copyright of their respective owners.
10 Information and contribution at https://typo3.org/
11 -->
12
13
14 <link rel="shortcut icon" href="/favicon.ico" type="image/vnd.microsoft.icon">
15 <title>TUM School of Computation, Information and Technology - TUM School of Computation, Information and Technology
```

Headers Text Hex Compressed **HTML** Raw

CONNECT https://gateway.icloud.com

Charles 4.6.2 - Session 1

Structure Sequence

Code	Method	Host	Path	Duration	Size	Status	Info
200	GET	www.cit.tum.de	/en/cit/home/	366 ms	10.53 KB	Complete	

Filter:  Focused

Overview Contents Summary Chart Notes

GET /en/cit/home/ HTTP/1.1  
Host www.cit.tum.de  
Connection keep-alive  
Cache-Control max-age=0

Headers Cookies Raw

```
1 <!DOCTYPE html>
2 <html dir="ltr" lang="en" class="no-js">
3 <head>
4 
5 <meta charset="utf-8">
6 --
7 This website is powered by TYPO3 - inspiring people to share!
8 TYPO3 is a free open source Content Management Framework initially created by Kasper Skaarhoj and licensed under GN
9 TYPO3 is copyright 1998-2022 of Kasper Skaarhoj. Extensions are copyright of their respective owners.
10 Information and contribution at https://typo3.org/
11 >
12 
13 
14 <link rel="shortcut icon" href="/favicon.ico" type="image/vnd.microsoft.icon">
15 <title>TUM School of Computation, Information and Technology - TUM School of Computation, Information and Technology
```

Headers Text Hex Compressed HTML Raw

CONNECT https://gateway.icloud.com

request message

response message

## Wi-Fi: en0

tcp.stream eq 2

Expression...

No.	Time	Source	Destination	Protocol	Length	Info
7	2.317550	192.168.2.213	129.187.255.73	TCP	78	53322 → 443 [SYN] Seq...
15	2.361519	129.187.255.73	192.168.2.213	TCP	70	443 → 53322 [SYN, ACK...]
16	2.361565	192.168.2.213	129.187.255.73	TCP	66	53322 → 443 [ACK] Seq...
18	2.363919	192.168.2.213	129.187.255.73	TLSv1.2	357	Client Hello
21	2.401583	129.187.255.73	192.168.2.213	TLSv1.2	213	Server Hello, Change ...
23	2.401808	192.168.2.213	129.187.255.73	TCP	66	53322 → 443 [ACK] Seq...
25	2.404349	192.168.2.213	129.187.255.73	TLSv1.2	72	Change Cipher Spec
27	2.404688	192.168.2.213	129.187.255.73	TLSv1.2	111	Encrypted Handshake M...
29	2.406527	102.168.2.212	129.187.255.73	TLSv1.2	1000	Application Data

- > Frame 18: 357 bytes on wire (2856 bits), 357 bytes captured (2856 bits) on interface 0
- > Ethernet II, Src: Apple\_b3:d7:02 (f0:18:98:b3:d7:02), Dst: Arcadyan\_08:99:7c (4c:1b:86:08:99:7c)
- > Internet Protocol Version 4, Src: 192.168.2.213, Dst: 129.187.255.73
- > Transmission Control Protocol, Src Port: 53322, Dst Port: 443, Seq: 1, Ack: 1, Len: 291
- > Transport Layer Security

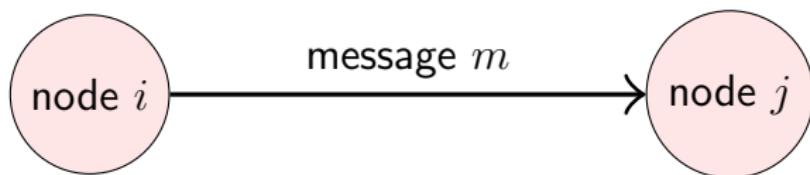
```

0000  4c 1b 86 08 99 7c f0 18  98 b3 d7 02 08 00 45 00  L.....|....E.
0010  01 57 00 00 40 00 40 06  f5 1e c0 a8 02 d5 81 bb  .W..@. .....
0020  ff 49 d0 4a 01 bb 21 e2  80 f6 a1 69 ed c1 80 18  .I.J.! ..i....
0030  ff ff ac 01 00 00 01 01  08 0a c4 45 fe f5 15 00  ..... .E....
0040  75 bc 16 03 03 01 1e 01  00 01 1a 03 03 02 61 41  u..... .aA
0050  81 4d f1 9c f1 bc 75 e5  b0 b9 e6 a4 6a 2e e9 04  .M...u....j...
0060  3e 11 c3 b3 91 31 4c 62  7a f7 69 04 21 20 fe 44  >...1Lb z.i!.D
0070  e1 bf 12 49 15 63 66 8c  c1 1d f8 11 3c 69 42 80  ...I.cf...<1B
0080  b8 fe ef 23 e5 42 63 78  81 34 ca 18 c7 b0 00 14  ...#.Bcx..4...{..
0090  c0 2b c0 2f c0 30 c0 13  c0 14 00 9c 00 9d  ..+/,..0 ...
00a0  00 2f 00 35 01 00 00 bd  00 00 00 13 00 11 00 00  ./5.....
00b0  0e 77 77 77 2e 63 69 74  2e 74 75 6d 2e 64 65 00  .www.cit.tum.de.

```

# Abstracting over networking details

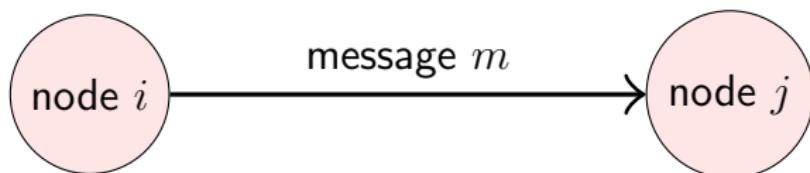
Network packets are too much low-level detail.  
We use a simple abstraction of communication:



# Abstracting over networking details

Network packets are too much low-level detail.

We use a simple abstraction of communication:



Reality is much more complex:

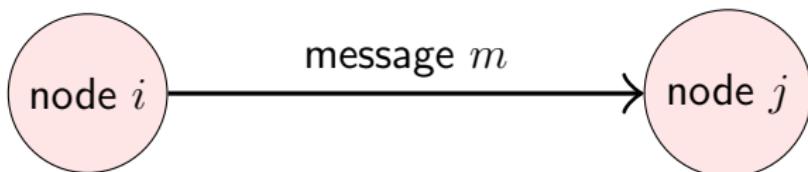
- ▶ **Node:**

- server, desktop computer, phone, car, robot, sensor, ...

# Abstracting over networking details

Network packets are too much low-level detail.

We use a simple abstraction of communication:



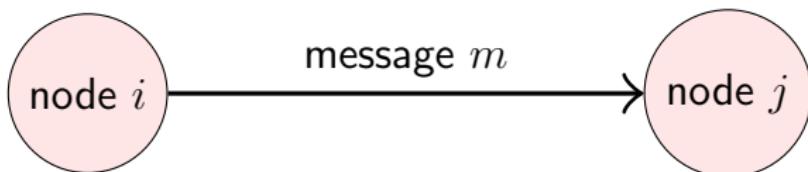
Reality is much more complex:

- ▶ **Node:**  
server, desktop computer, phone, car, robot, sensor, ...
- ▶ **Various network operators:**  
eduroam, home DSL, cellular data, coffee shop wifi, submarine cable, satellite...

# Abstracting over networking details

Network packets are too much low-level detail.

We use a simple abstraction of communication:



Reality is much more complex:

- ▶ **Node:**  
server, desktop computer, phone, car, robot, sensor, ...
- ▶ **Various network operators:**  
eduroam, home DSL, cellular data, coffee shop wifi, submarine cable, satellite...
- ▶ **Physical communication:**  
electric current, radio waves, laser, hard drives in a van...

# Hard drives in a van?!



<https://docs.aws.amazon.com/snowball/latest/ug/using-device.html>

High latency, high bandwidth!

# Latency and bandwidth

**Latency:** time until message arrives

- ▶ In the same building/datacenter:  $\approx 1$  ms
- ▶ One continent to another:  $\approx 100$  ms
- ▶ Hard drives in a van:  $\approx 1$  day

**Latency:** time until message arrives

- ▶ In the same building/datacenter:  $\approx 1$  ms
- ▶ One continent to another:  $\approx 100$  ms
- ▶ Hard drives in a van:  $\approx 1$  day

**Bandwidth:** data volume per unit time

- ▶ 3G cellular data:  $\approx 1$  Mbit/s
- ▶ Home broadband:  $\approx 10$  Mbit/s
- ▶ Hard drives in a van: 50 TB/box  $\approx 1$  Gbit/s

(Very rough numbers, vary hugely in practice!)

<b>networking</b>	<b>distributed systems</b>
how to get data from A to B (packets, routing, ...)	how to achieve some goal by sending/receiving messages

<b>networking</b>	<b>distributed systems</b>
how to get data from A to B (packets, routing, ...)	how to achieve some goal by sending/receiving messages
“TCP is reliable” (dropped packets are automatically retransmitted)	any message can be lost (unplug the network cable ⇒ TCP timeout)



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

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 requests in a day get a response in 200 ms”

**Service-Level Agreement (SLA):**

contract specifying some SLO, penalties for violation

# Achieving high availability: fault tolerance



**Failure:** system as a whole isn't working

**Fault:** some part of the system isn't working

- ▶ Node fault: crash, deadlock, ...
- ▶ Network fault: dropping or significantly delaying messages

**Failure:** system as a whole isn't working

**Fault:** some part of the system isn't working

- ▶ Node fault: crash, deadlock, ...
- ▶ Network fault: dropping or significantly delaying messages

To increase availability: have fewer faults, or **tolerate** faults

**Fault tolerance:**

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

**Failure:** system as a whole isn't working

**Fault:** some part of the system isn't working

- ▶ Node fault: crash, deadlock, ...
- ▶ Network fault: dropping or significantly delaying messages

To increase availability: have fewer faults, or **tolerate** faults

**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

Fault tolerance is also useful for **software updates**:

reboot one node at a time while continuing to serve users

## **Practice:**

How can we achieve good performance in the common case?

## **Theory:**

How can we guarantee correctness in all possible scenarios?

## Practice:

How can we achieve good performance in the common case?

## Theory:

How can we guarantee correctness in all possible scenarios?

Build a system without understanding the theory?

- ▶ works fine for a while...
- ▶ but one day it fails catastrophically due to some weird edge case, and corrupts all your data 

## Practice:

How can we achieve good performance in the common case?

## Theory:

How can we guarantee correctness in all possible scenarios?

Build a system without understanding the theory?

- ▶ works fine for a while...
- ▶ but one day it fails catastrophically due to some weird edge case, and corrupts all your data 

⚠ Distributed systems are notoriously hard to get right.  
The theory helps us build robust systems.

# Why make a system distributed?



# Why make a system distributed?



- ▶ **It's inherently distributed:**

- e.g. sending a message from your mobile phone to your friend's phone

# Why make a system distributed?



- ▶ **It's inherently distributed:**

- e.g. sending a message from your mobile phone to your friend's phone

- ▶ **For better reliability:**

- even if one node fails, the system as a whole keeps functioning (fault tolerance)

# Why make a system distributed?



- ▶ **It's inherently distributed:**  
e.g. sending a message from your mobile phone to your friend's phone
- ▶ **For better reliability:**  
even if one node fails, the system as a whole keeps functioning (fault tolerance)
- ▶ **For better performance:**  
get data from a nearby node rather than one halfway round the world

# Why make a system distributed?



- ▶ **It's inherently distributed:**  
e.g. sending a message from your mobile phone to your friend's phone
- ▶ **For better reliability:**  
even if one node fails, the system as a whole keeps functioning (fault tolerance)
- ▶ **For better performance:**  
get data from a nearby node rather than one halfway round the world
- ▶ **To solve bigger problems:**  
e.g. huge amounts of data, can't fit on one machine (scalability)



www.cit.tum.de



cit.tum.de



## No Internet

No Internet

Try:

- Checking the network cables, modem and router
- Reconnecting to Wi-Fi

DNS\_PROBE\_FINISHED\_NO\_INTERNET

# Why NOT make a system distributed?



The trouble with distributed systems:

- ▶ Communication may fail (and we might not even know it has failed).
- ▶ Processes may crash (and we might not know).
- ▶ All of this may happen nondeterministically and without warning.

# Why NOT make a system distributed?



The trouble with distributed systems:

- ▶ Communication may fail (and we might not even know it has failed).
- ▶ Processes may crash (and we might not know).
- ▶ All of this may happen nondeterministically and without warning.

**Fault tolerance:** we want the system as a whole to continue working, even when some parts are faulty.

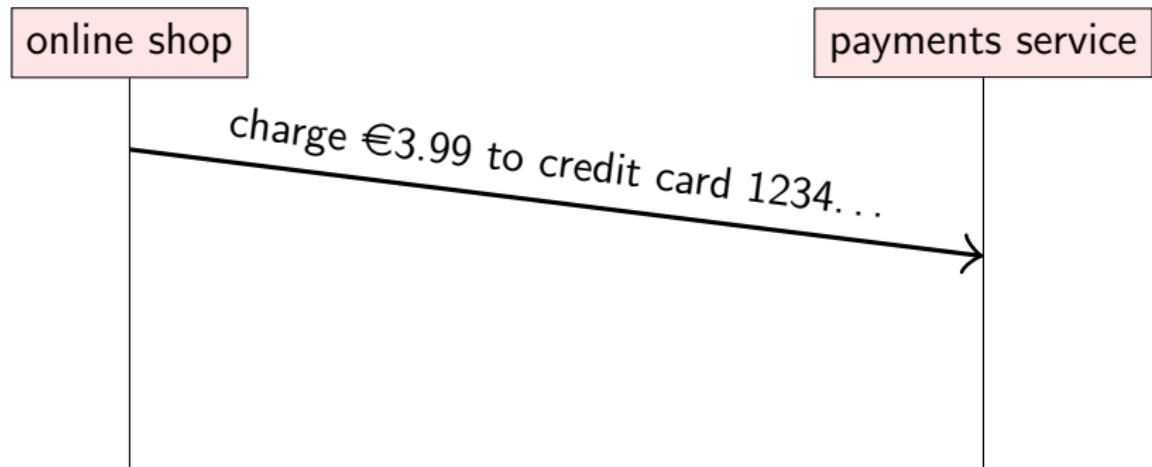
This is hard.

Writing a program to run on a single computer is comparatively easy?!

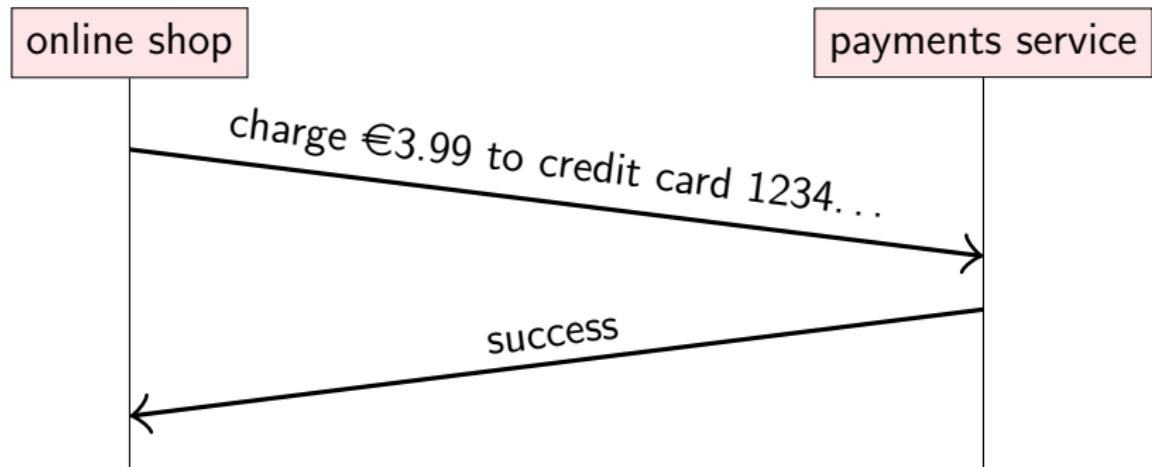
# Client-server example: online payments



# Client-server example: online payments



# Client-server example: online payments



# Remote Procedure Call (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");

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

if (result.isSuccess()) {
    fulfillOrder();
}
```

# Remote Procedure Call (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");

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

if (result.isSuccess()) {
    fulfillOrder();
}
```



Implementation of this function is on another node!

online shop

RPC client

RPC server

payment service

processPayment() stub

waiting

online shop

RPC client

RPC server

payment service

processPayment() stub

waiting

marshal args

$m_1$

unmarshal args

{  
  "request": "processPayment",  
  "card": {  
    "number": "1234567887654321",  
    "expiryDate": "10/2024",  
    "CVC": "123"  
  },  
  "amount": 3.99,  
  "currency": "EUR"  
}

$m_1 =$

online shop

RPC client

RPC server

payment service

processPayment() stub

waiting

marshal args

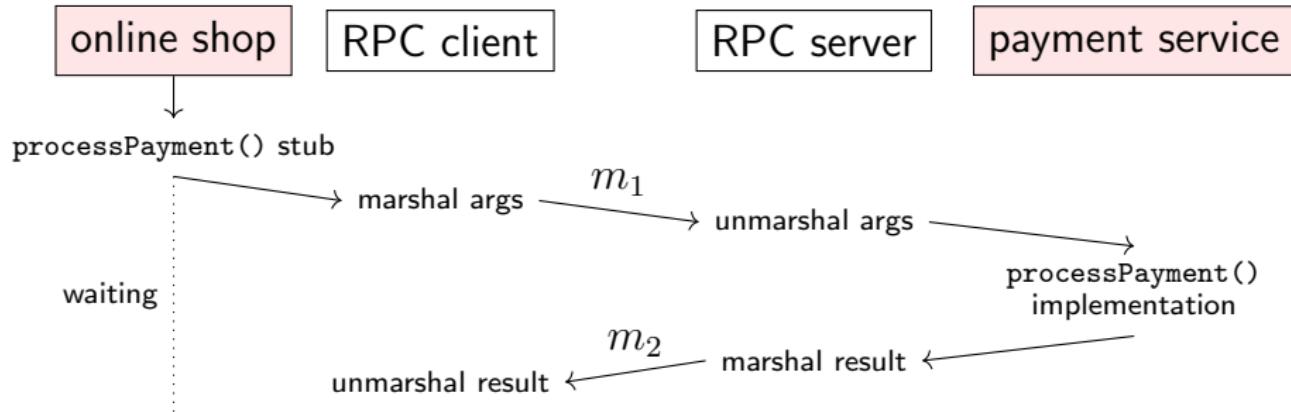
$m_1$

unmarshal args

processPayment()  
implementation

```
{  
    "request": "processPayment",  
    "card": {  
        "number": "1234567887654321",  
        "expiryDate": "10/2024",  
        "CVC": "123"  
    },  
    "amount": 3.99,  
    "currency": "EUR"  
}
```

$m_1 =$

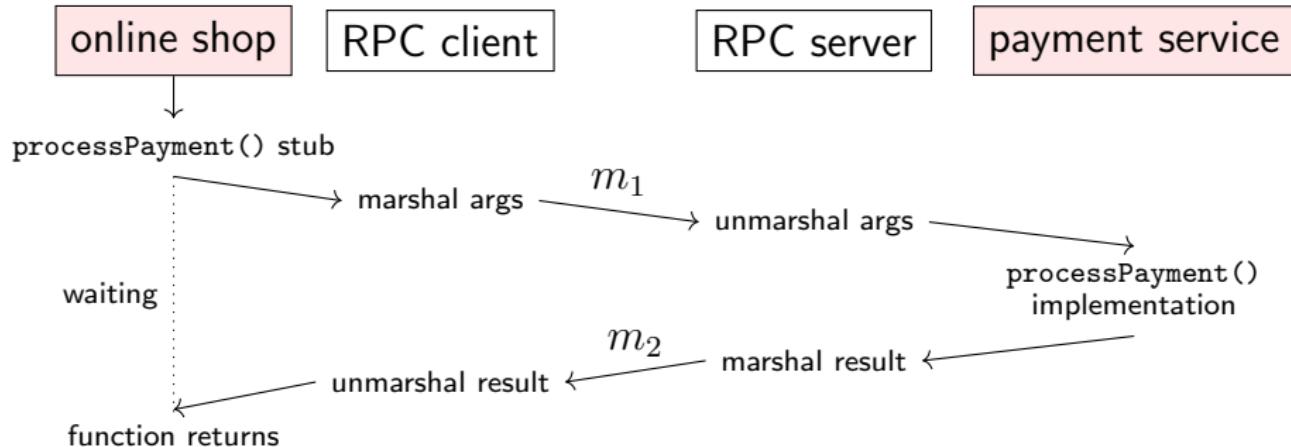


$$m_1 =$$

```
{  
  "request": "processPayment",  
  "card": {  
    "number": "1234567887654321",  
    "expiryDate": "10/2024",  
    "CVC": "123"  
  },  
  "amount": 3.99,  
  "currency": "EUR"  
}
```

$$m_2 =$$

```
{  
    "result": "success",  
    "id": "XP61hHw2Rvo"  
}
```



$$m_1 =$$

```
{  
  "request": "processPayment",  
  "card": {  
    "number": "1234567887654321",  
    "expiryDate": "10/2024",  
    "CVC": "123"  
  },  
  "amount": 3.99,  
  "currency": "EUR"  
}
```

$$m_2 =$$

```
{  
    "result": "success",  
    "id": "XP61hHw2Rvo"  
}
```

# Remote Procedure Call (RPC)



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

**“Location transparency”:**

system hides where a resource is located.

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

**“Location transparency”:**

system hides where a resource is located.

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?

- ▶ SunRPC/ONC RPC (1980s, basis for NFS)
- ▶ CORBA: object-oriented middleware, hot in the 1990s
- ▶ Microsoft's DCOM and Java RMI (similar to CORBA)
- ▶ SOAP/XML-RPC: RPC using XML and HTTP (1998)
- ▶ Thrift (Facebook, 2007)
- ▶ gRPC (Google, 2015)
- ▶ REST (often with JSON)
- ▶ JavaScript in web browsers making server requests  
(XMLHttpRequest, AJAX, fetch API, ...)

# RPC/REST in JavaScript



```
let args = {amount: 3.99, currency: 'EUR', /*...*/};  
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) => {  
        failure(error); // network error  
    });
```

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

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

(Server-to-server RPC within the same company)

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

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

(Server-to-server RPC within the same company)

Different services implemented in different languages:

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

# gRPC IDL example



```
message PaymentRequest {  
    message Card {  
        required string cardNumber = 1;  
        optional int32 expiryMonth = 2;  
        optional int32 expiryYear = 3;  
        optional int32 CVC = 4;  
    }  
    enum Currency { EUR = 1; USD = 2; }  
  
    required Card card = 1;  
    required int64 amount = 2;  
    required Currency currency = 3;  
}  
  
message PaymentStatus {  
    required bool success = 1;  
    optional string errorMessage = 2;  
}  
  
service PaymentService {  
    rpc ProcessPayment(PaymentRequest) returns (PaymentStatus) {}  
}
```

# Models of distributed systems

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

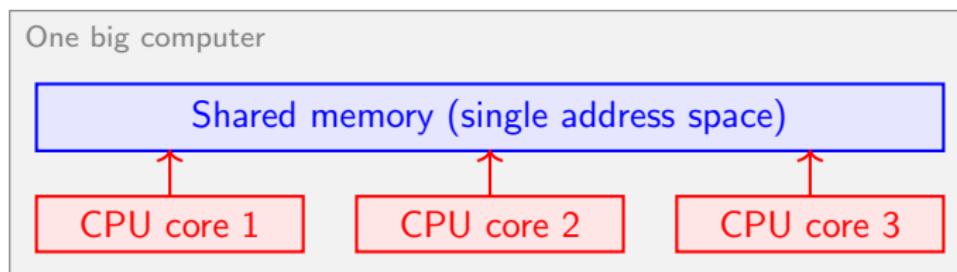
TU Munich

Chair of Decentralized Systems Engineering

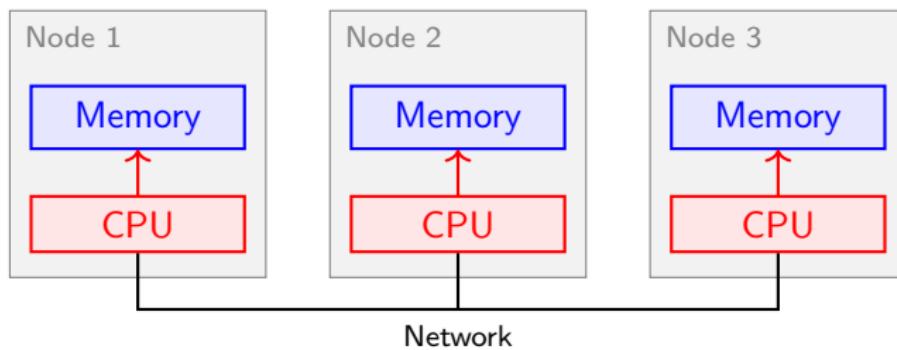
<https://dse.in.tum.de/>

# Two models of concurrency

Shared-memory concurrency:

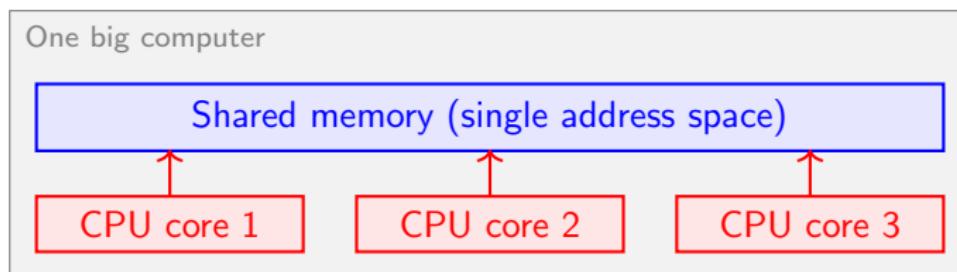


Message-passing distributed systems:

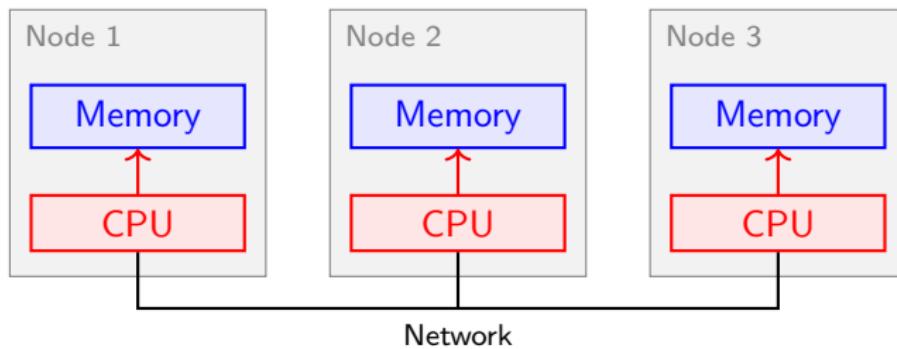


# Two models of concurrency

Shared-memory concurrency:



Message-passing distributed systems: ← **this course**



# A distributed system is...



- ▶ Multiple “nodes” (computers)
- ▶ communicating via an unreliable network
- ▶ trying to achieve some task together

# A distributed system is...

- ▶ Multiple “nodes” (computers)
- ▶ communicating via an unreliable network
- ▶ trying to achieve some task together

<b>shared memory</b>	<b>distributed system</b>
hardware fails ⇒ all threads stop	one machine fails ⇒ others continue running

# A distributed system is...

- ▶ Multiple “nodes” (computers)
- ▶ communicating via an unreliable network
- ▶ trying to achieve some task together

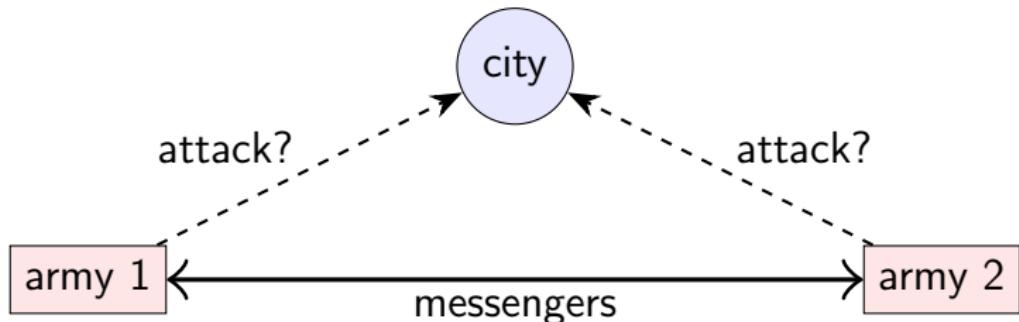
<b>shared memory</b>	<b>distributed system</b>
hardware fails ⇒ all threads stop	one machine fails ⇒ others continue running
reliable communication between CPU cores	unreliable network

# A distributed system is...

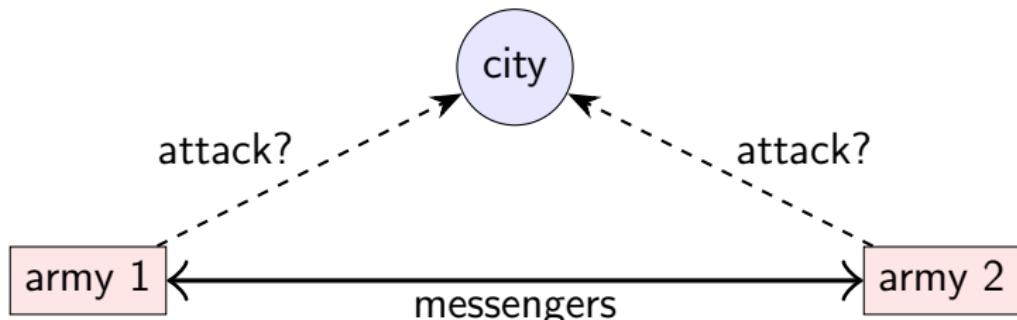
- ▶ Multiple “nodes” (computers)
- ▶ communicating via an unreliable network
- ▶ trying to achieve some task together

<b>shared memory</b>	<b>distributed system</b>
hardware fails ⇒ all threads stop	one machine fails ⇒ others continue running
reliable communication between CPU cores	unreliable network
locks, semaphores, atomic instructions (e.g. CAS)	only message-passing

# The two generals problem



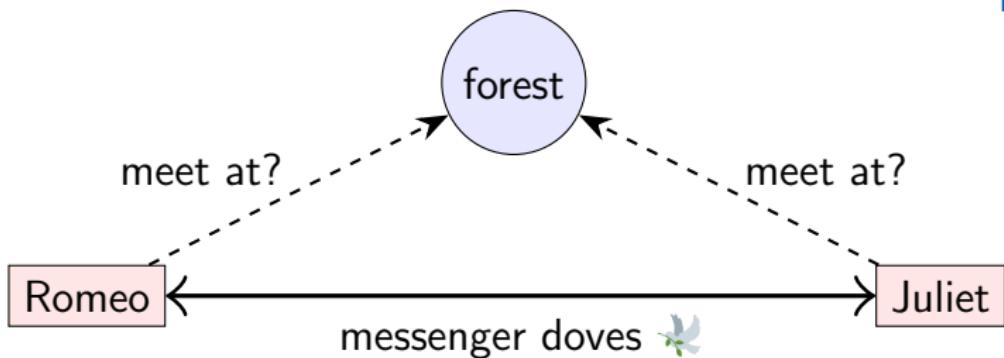
# The two generals problem



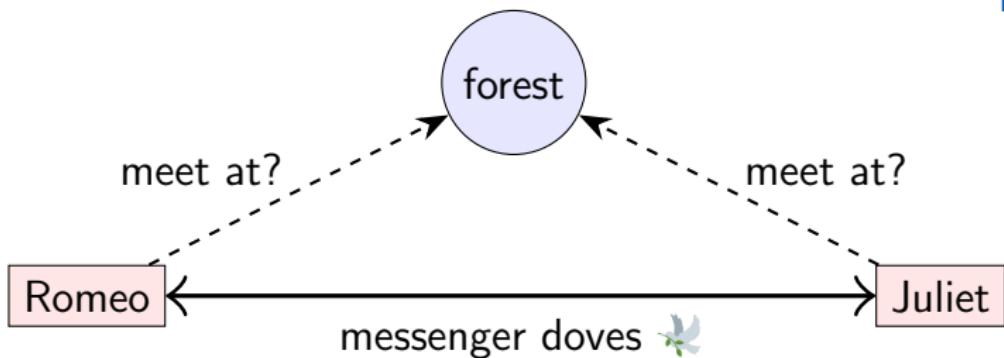
army 1	army 2	outcome
does not attack	does not attack	nothing happens
attacks	does not attack	army 1 defeated
does not attack	attacks	army 2 defeated
attacks	attacks	city captured

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

# The Romeo and Juliet problem



# The Romeo and Juliet problem



Romeo	Juliet	outcome
does not go	does not go	nothing happens
goes	does not go	Romeo gets desperate
does not go	goes	Juliet gets desperate
goes	goes	happy ever after

**Desired:** Romeo goes to the forest *if and only if* Juliet goes

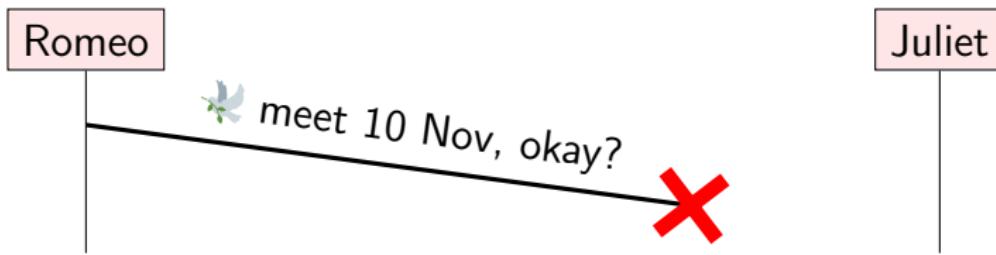
# Reaching agreement with message loss



# Reaching agreement with message loss



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



# How should Romeo and Juliet decide?



1. Romeo always goes into the forest, even if no response is received?
  - ▶ Send lots of messages to increase probability that one will get through
  - ▶ If all are lost, Juliet does not know about the meeting, so Romeo is alone

# How should Romeo and Juliet decide?



1. Romeo always goes into the forest, even if no response is received?
  - ▶ Send lots of messages to increase probability that one will get through
  - ▶ If all are lost, Juliet does not know about the meeting, so Romeo is alone
2. Romeo only goes into the forest if positive response from Juliet is received?
  - ▶ Now Romeo is safe
  - ▶ But Juliet knows that Romeo will only go if Juliet's response gets through
  - ▶ Now Juliet is in the same situation as Romeo in option 1

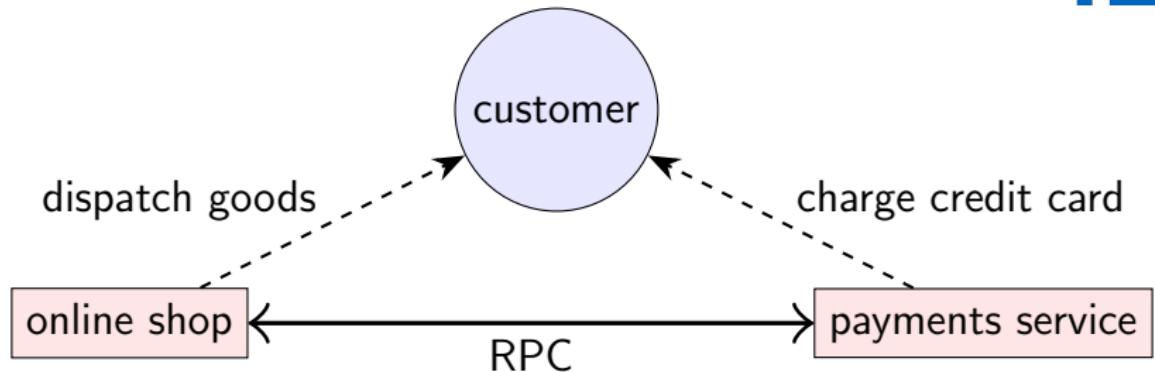
# How should Romeo and Juliet decide?



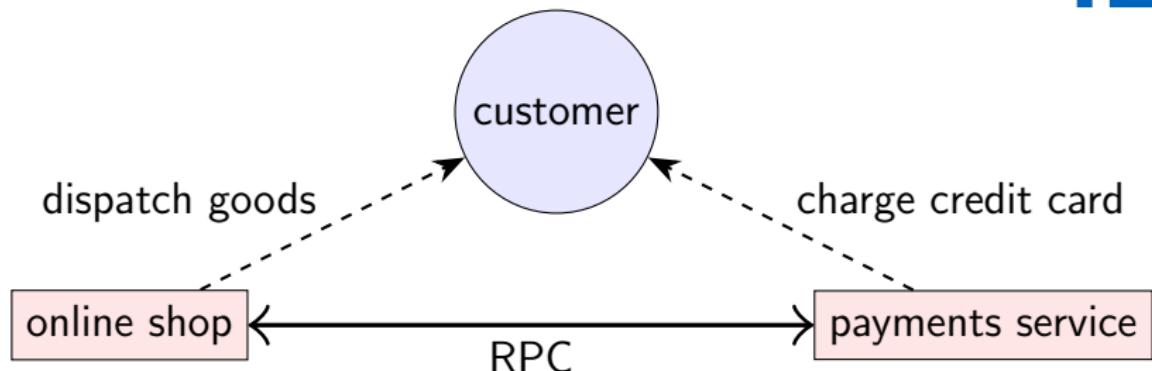
1. Romeo always goes into the forest, even if no response is received?
  - ▶ Send lots of messages to increase probability that one will get through
  - ▶ If all are lost, Juliet does not know about the meeting, so Romeo is alone
2. Romeo only goes into the forest if positive response from Juliet is received?
  - ▶ Now Romeo is safe
  - ▶ But Juliet knows that Romeo will only go if Juliet's response gets through
  - ▶ Now Juliet is in the same situation as Romeo in option 1

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

# The two generals problem applied



# 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

**Desired:** online shop dispatches *if and only if* payment made

# Two generals $\neq$ online shopping



Analysing more carefully, we find that online shopping is not like the two generals after all.

Online shopping can use the following protocol:

# Two generals $\neq$ online shopping



Analysing more carefully, we find that online shopping is not like the two generals after all.

Online shopping can use the following protocol:

1. Try to charge customer's credit card

# Two generals $\neq$ online shopping



Analysing more carefully, we find that online shopping is not like the two generals after all.

Online shopping can use the following protocol:

1. Try to charge customer's credit card
2. If charge was successful, try dispatching goods

# Two generals $\neq$ online shopping



Analysing more carefully, we find that online shopping is not like the two generals after all.

Online shopping can use the following protocol:

1. Try to charge customer's credit card
2. If charge was successful, try dispatching goods
3. If dispatch was unsuccessful (e.g. out of stock):  
**refund the credit card payment**

# Two generals $\neq$ online shopping



Analysing more carefully, we find that online shopping is not like the two generals after all.

Online shopping can use the following protocol:

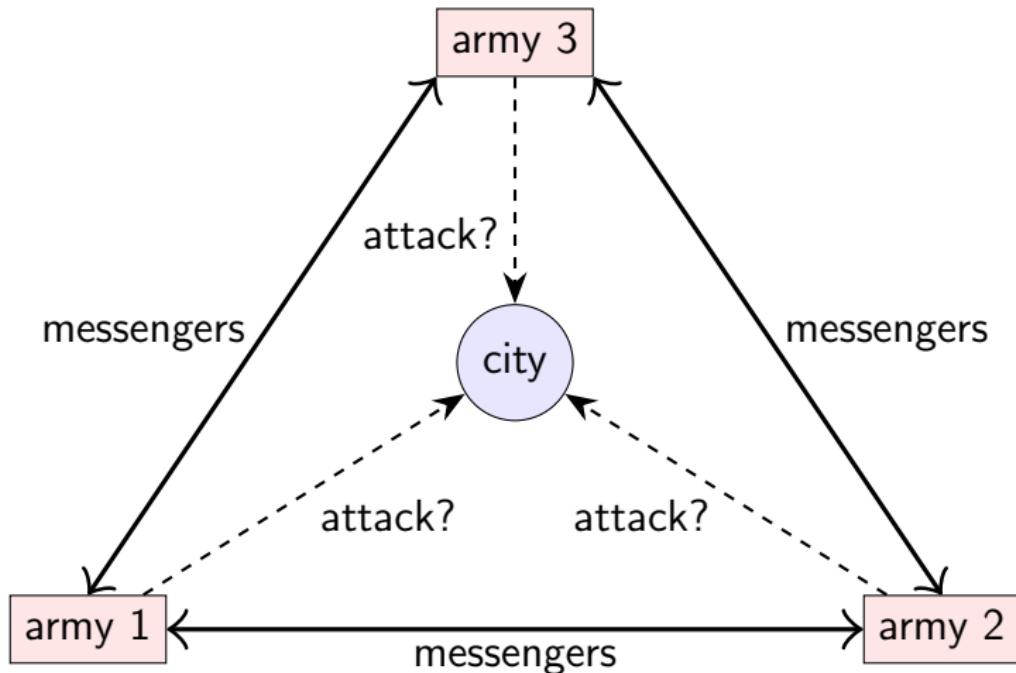
1. Try to charge customer's credit card
2. If charge was successful, try dispatching goods
3. If dispatch was unsuccessful (e.g. out of stock):  
**refund the credit card payment**

The fact that one of the actions (payment) can be undone makes the problem solveable.

Defeat of an army cannot be undone.

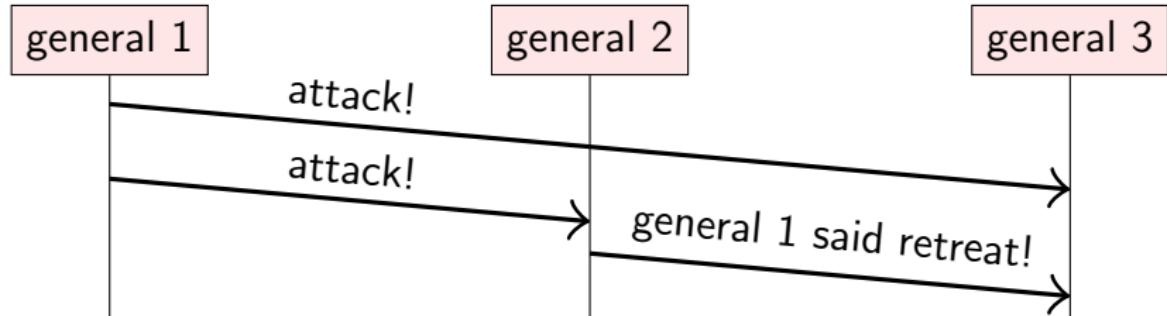
Dispatching goods cannot be undone.

# The Byzantine generals problem

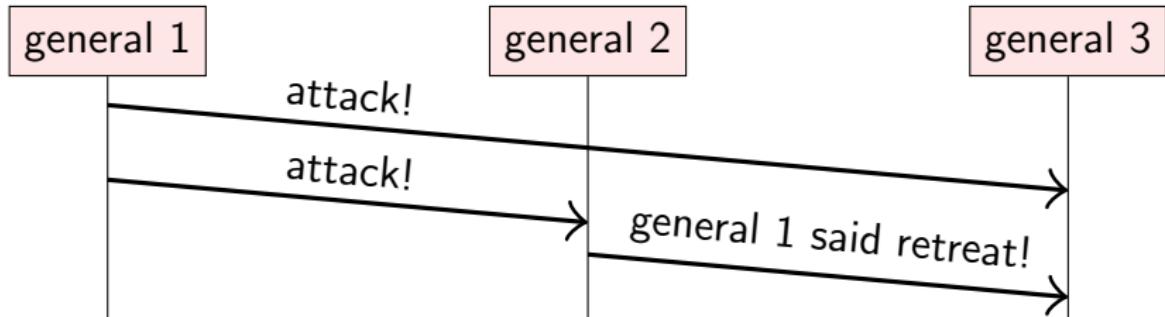


**Problem:** some of the generals might be traitors

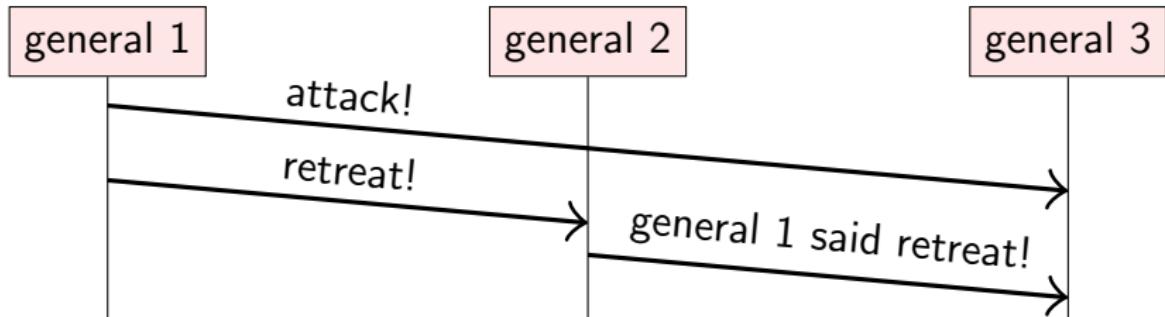
# Generals who might lie



# Generals who might lie



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



# The Byzantine generals problem

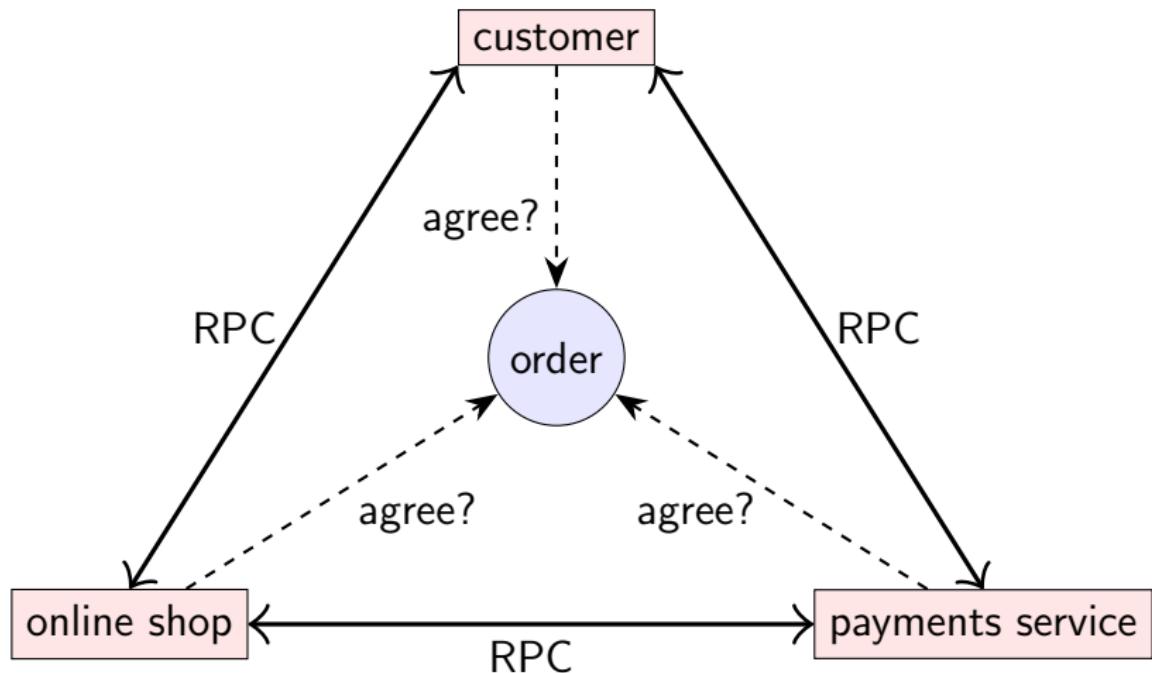


- ▶ Each general is either *malicious* or *honest*
- ▶ Up to  $f$  generals might be malicious
- ▶ Honest generals don't know who the malicious ones are
- ▶ The malicious generals may collude
- ▶ Nevertheless, honest generals must agree on plan

# 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

# Trust relationships and malicious behaviour



Who can trust whom?

# The Byzantine empire (650 CE)

Byzantium/Constantinople/Istanbul



Source: <https://commons.wikimedia.org/wiki/File:Byzantiumby650AD.svg>

**“Byzantine”** has long been used for “excessively complicated, bureaucratic, devious” (e.g. *“the Byzantine tax law”*)

We have seen two thought experiments:

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

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

We have seen two thought experiments:

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

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

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

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

Choice of models for each of these parts.

# Networks are unreliable



In the sea, sharks bite fibre optic cables

[https://slate.com/technology/2014/08/](https://slate.com/technology/2014/08/shark-attacks-threaten-google-s-undersea-internet-cables-video.html)

[shark-attacks-threaten-google-s-undersea-internet-cables-video.html](https://slate.com/technology/2014/08/shark-attacks-threaten-google-s-undersea-internet-cables-video.html)

On land, cows step on the cables

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

# System model: network behaviour



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

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.

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, a message eventually gets through.

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, a message eventually gets through.
- ▶ **Arbitrary** links (active adversary):  
A malicious adversary may interfere with messages  
(eavesdrop, modify, drop, spoof, replay).

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, a 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

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, a message eventually gets through.
  - ▶ **Arbitrary** links (active adversary):  
A malicious adversary may interfere with messages  
(eavesdrop, modify, drop, spoof, replay).
- 
- retry + dedup

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

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.

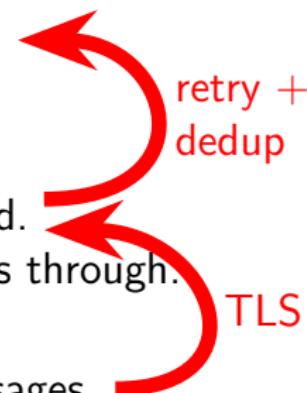
- ▶ **Fair-loss** links:

Messages may be lost, duplicated, or reordered.

If you keep retrying, a 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.

# 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.

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: 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.

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.

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 practice



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

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

# Violations of synchrony in practice



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

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!

## **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

## **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

## **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

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

# Time, clocks, and ordering of events

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

# A detective story

In the night from 30 June to 1 July 2012 (CET), many online services and systems around the world crashed simultaneously.

Servers locked up and stopped responding.

Some airlines could not process any reservations or check-ins for several hours.

What happened?

# Clocks and time in distributed systems



Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers

# Clocks and time in distributed systems



Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling

# Clocks and time in distributed systems



Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred

# Clocks and time in distributed systems



Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred
- ▶ Data with time-limited validity (e.g. cache entries)

Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred
- ▶ Data with time-limited validity (e.g. cache entries)
- ▶ Determining order of events across several nodes

Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred
- ▶ Data with time-limited validity (e.g. cache entries)
- ▶ Determining order of events across several nodes

We distinguish two types of clock:

- ▶ **physical clocks**: count number of seconds elapsed
- ▶ **logical clocks**: count events, e.g. messages sent

Distributed systems often need to measure time, e.g.:

- ▶ Schedulers, timeouts, failure detectors, retry timers
- ▶ Performance measurements, statistics, profiling
- ▶ Log files & databases: record when an event occurred
- ▶ Data with time-limited validity (e.g. cache entries)
- ▶ Determining order of events across several nodes

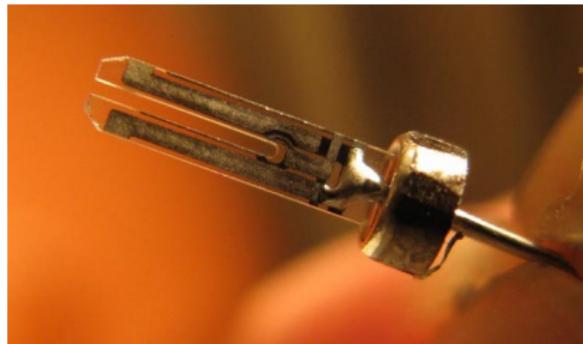
We distinguish two types of clock:

- ▶ **physical clocks**: count number of seconds elapsed
- ▶ **logical clocks**: count events, e.g. messages sent

**NB.** Clock in digital electronics (oscillator)  
≠ clock in distributed systems (source of **timestamps**)

# Quartz clocks

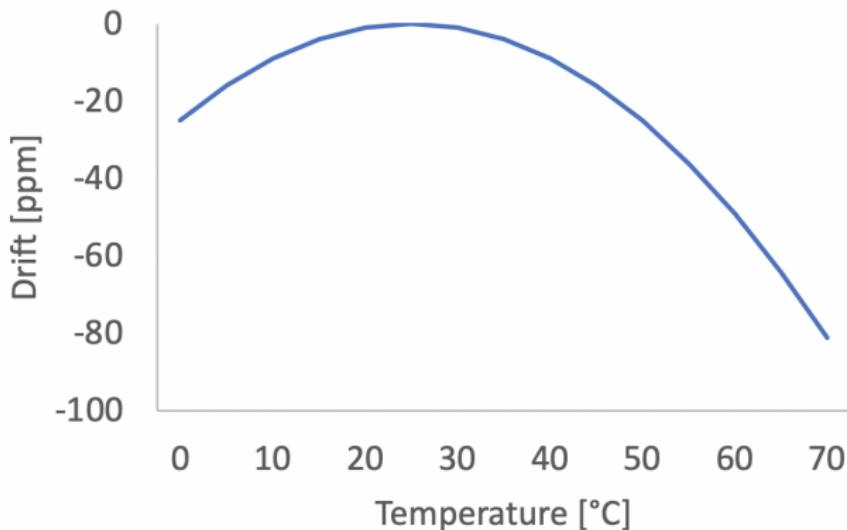
- ▶ Quartz crystal laser-trimmed to mechanically resonate at a specific frequency
- ▶ Piezoelectric effect: mechanical force  $\Leftrightarrow$  electric field
- ▶ Oscillator circuit produces signal at resonant frequency
- ▶ Count number of cycles to measure elapsed time



# Quartz clock error: drift

- ▶ One clock runs slightly fast, another slightly slow
- ▶ Drift measured in **parts per million** (ppm)
- ▶  $1 \text{ ppm} = 1 \text{ microsecond/second} = 86 \text{ ms/day} = 32 \text{ s/year}$
- ▶ Most computer clocks correct within  $\approx 50 \text{ ppm}$

Temperature significantly affects drift



# Atomic clocks

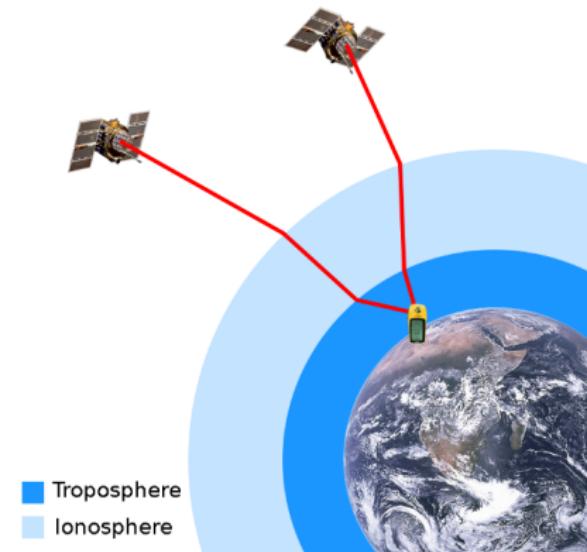
- ▶ Caesium-133 has a resonance (“hyperfine transition”) at  $\approx 9$  GHz
- ▶ Tune an electronic oscillator to that resonant frequency
- ▶ 1 second = 9,192,631,770 periods of that signal
- ▶ Accuracy  $\approx 1$  in  $10^{-14}$  (1 second in 3 million years)
- ▶ Price  $\approx \text{€}20,000$  (?)  
(can get cheaper rubidium clocks for  $\approx \text{€}1,000$ )



https:  
[//www.microsemi.com/product-directory/  
cesium-frequency-references/  
4115-5071a-cesium-primary-frequency-standard](https://www.microsemi.com/product-directory/cesium-frequency-references/4115-5071a-cesium-primary-frequency-standard)

# GPS as time source

- ▶ 31 satellites, each carrying an atomic clock
- ▶ satellite broadcasts current time and location
- ▶ calculate position from speed-of-light delay between satellite and receiver
- ▶ corrections for atmospheric effects, relativity, etc.
- ▶ in datacenters, need antenna on the roof



<https://commons.wikimedia.org/wiki/File:Gps-atmospheric-efects.png>

# Coordinated Universal Time (UTC)



**Greenwich Mean Time** (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian



# Coordinated Universal Time (UTC)



**Greenwich Mean Time** (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

**International Atomic Time** (TAI): 1 day is  $24 \times 60 \times 60 \times 9,192,631,770$  periods of caesium-133's resonant frequency



# Coordinated Universal Time (UTC)



**Greenwich Mean Time** (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

**International Atomic Time** (TAI): 1 day is  $24 \times 60 \times 60 \times 9,192,631,770$  periods of caesium-133's resonant frequency

**Problem:** speed of Earth's rotation is not constant



# Coordinated Universal Time (UTC)



**Greenwich Mean Time** (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

**International Atomic Time** (TAI): 1 day is  $24 \times 60 \times 60 \times 9,192,631,770$  periods of caesium-133's resonant frequency

**Problem:** speed of Earth's rotation is not constant

**Compromise:** UTC is TAI with corrections to account for Earth rotation



# Coordinated Universal Time (UTC)



**Greenwich Mean Time** (GMT, solar time): it's noon when the sun is in the south, as seen from the Greenwich meridian

**International Atomic Time** (TAI): 1 day is  $24 \times 60 \times 60 \times 9,192,631,770$  periods of caesium-133's resonant frequency

**Problem:** speed of Earth's rotation is not constant

**Compromise:** UTC is TAI with corrections to account for Earth rotation

**Time zones** and **daylight savings time** are offsets to UTC



# Leap seconds

Every year, on 30 June and 31 December at 23:59:59 UTC, one of three things happens:

- ▶ The clock immediately jumps forward to 00:00:00, skipping one second (**negative leap second**)
- ▶ The clock moves to 00:00:00 after one second, as usual
- ▶ The clock moves to 23:59:60 after one second, and then moves to 00:00:00 after one further second  
**(positive leap second)**

This is announced several months beforehand.



<http://leapsecond.com/notes/leap-watch.htm>

# How computers represent timestamps



Two most common representations:

- ▶ **Unix time**: number of seconds since 1 January 1970 00:00:00 UTC (the “epoch”), *not counting leap seconds*
- ▶ **ISO 8601**: year, month, day, hour, minute, second, and timezone offset relative to UTC  
example: 2021-11-09T09:50:17+00:00

Two most common representations:

- ▶ **Unix time**: number of seconds since 1 January 1970 00:00:00 UTC (the “epoch”), *not counting leap seconds*
- ▶ **ISO 8601**: year, month, day, hour, minute, second, and timezone offset relative to UTC  
example: 2021-11-09T09:50:17+00:00

Conversion between the two requires:

- ▶ Gregorian calendar: 365 days in a year, except leap years  
`(year % 4 == 0 && (year % 100 != 0 ||  
year % 400 == 0))`
- ▶ Knowledge of past and future leap seconds... ?!

# How most software deals with leap seconds



By ignoring them!



[https://www.flickr.com/  
photos/ru\\_boff/  
37915499055/](https://www.flickr.com/photos/ru_boff/37915499055/)

# How most software deals with leap seconds



**By ignoring them!**

However, OS and DistSys often need timings with sub-second accuracy.



[https://www.flickr.com/  
photos/ru\\_boff/  
37915499055/](https://www.flickr.com/photos/ru_boff/37915499055/)

## By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.

30 June 2012: bug in Linux kernel caused livelock on leap second, causing many Internet services to go down



[https://www.flickr.com/  
photos/ru\\_boff/  
37915499055/](https://www.flickr.com/photos/ru_boff/37915499055/)

## By ignoring them!

However, OS and DistSys often need timings with sub-second accuracy.

30 June 2012: bug in Linux kernel caused livelock on leap second, causing many Internet services to go down

Pragmatic solution: “**smear**” (spread out) the leap second over the course of a day



[https://www.flickr.com/  
photos/ru\\_boff/  
37915499055/](https://www.flickr.com/photos/ru_boff/37915499055/)

# Clock synchronisation



Computers track physical time/UTC with a quartz clock  
(with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

Computers track physical time/UTC with a quartz clock  
(with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

**Clock skew**: difference between two clocks at a point in time

Computers track physical time/UTC with a quartz clock  
(with battery, continues running when power is off)

Due to **clock drift**, clock error gradually increases

**Clock skew**: difference between two clocks at a point in time

**Solution**: Periodically get the current time from a server that has a more accurate time source (atomic clock or GPS receiver)

Protocols: Network Time Protocol (**NTP**),  
Precision Time Protocol (**PTP**)



## Date &amp; Time

Search

## Date &amp; Time

## Time Zone

## Clock

Set date and time automatically: Apple Europe (time.euro.apple.com.) 

23/09/2020 11:03:48 

Sep 2020						
Mo	Tu	We	Th	Fr	Sa	Su
31	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20
21	22	23	24	25	26	27
28	29	30	1	2	3	4
5	6	7	8	9	10	11



To set date and time formats, use Language & Region preferences. [Open Language & Region...](#)



Click the lock to prevent further changes.



# Network Time Protocol (NTP)



Many operating system vendors run NTP servers,  
configure OS to use them by default

# Network Time Protocol (NTP)



Many operating system vendors run NTP servers,  
configure OS to use them by default

Hierarchy of clock servers arranged into **strata**:

- ▶ Stratum 0: atomic clock or GPS receiver
- ▶ Stratum 1: synced directly with stratum 0 device
- ▶ Stratum 2: servers that sync with stratum 1, etc.

# Network Time Protocol (NTP)



Many operating system vendors run NTP servers, configure OS to use them by default

Hierarchy of clock servers arranged into **strata**:

- ▶ Stratum 0: atomic clock or GPS receiver
- ▶ Stratum 1: synced directly with stratum 0 device
- ▶ Stratum 2: servers that sync with stratum 1, etc.

May contact multiple servers, discard outliers, average rest

Makes multiple requests to the same server, use statistics to reduce random error due to variations in network latency

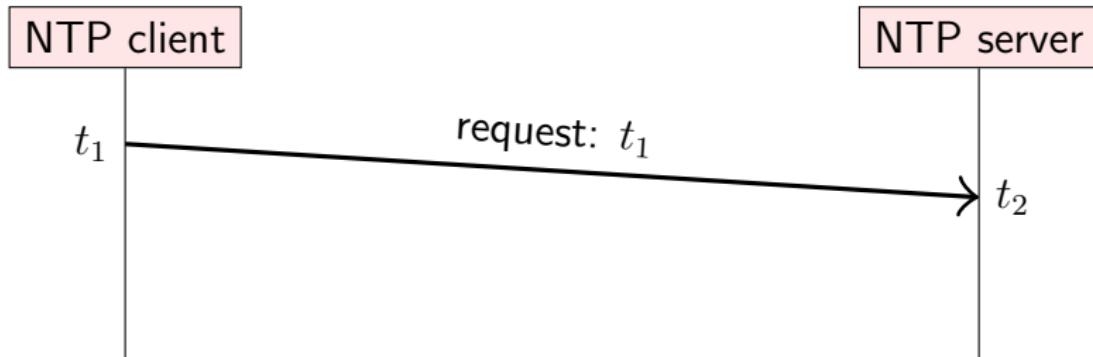
Reduces clock skew to a few milliseconds in good network conditions, but can be much worse!

# Estimating time over a network

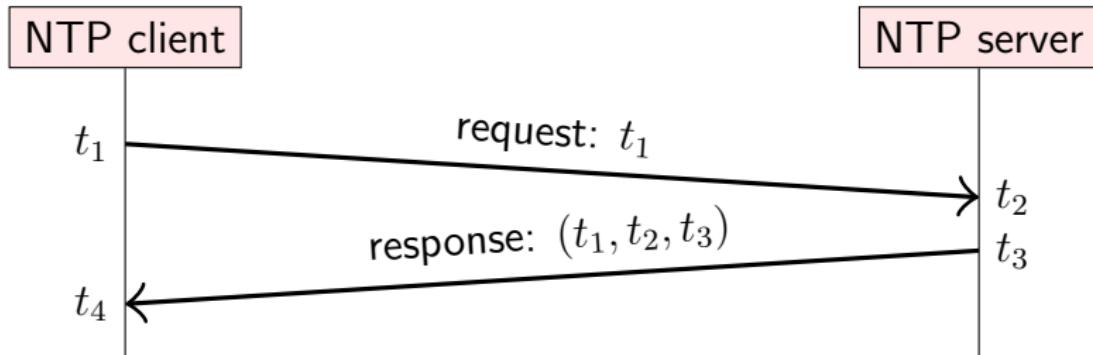
NTP client

NTP server

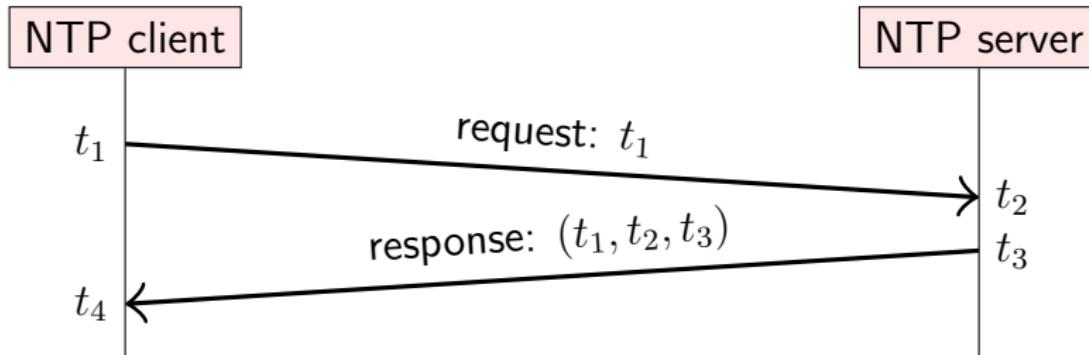
# Estimating time over a network



# Estimating time over a network



# Estimating time over a network

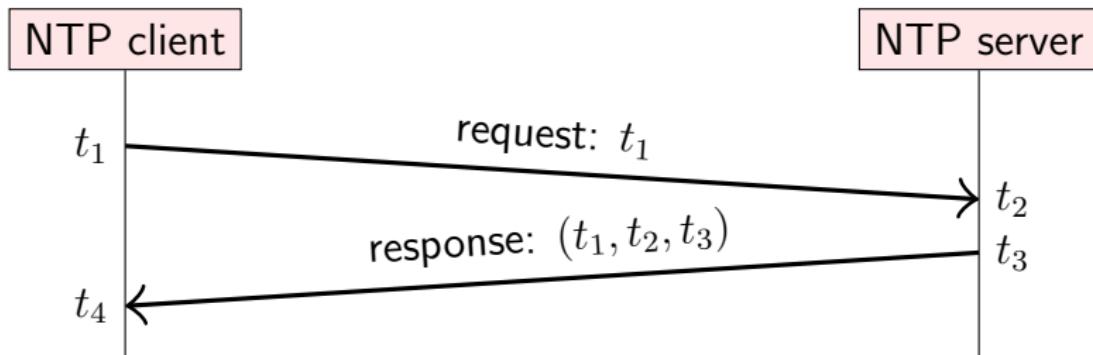


$$\text{Round-trip network delay: } \delta = (t_4 - t_1) - (t_3 - t_2)$$

total time of

wave  
client      wave  
server

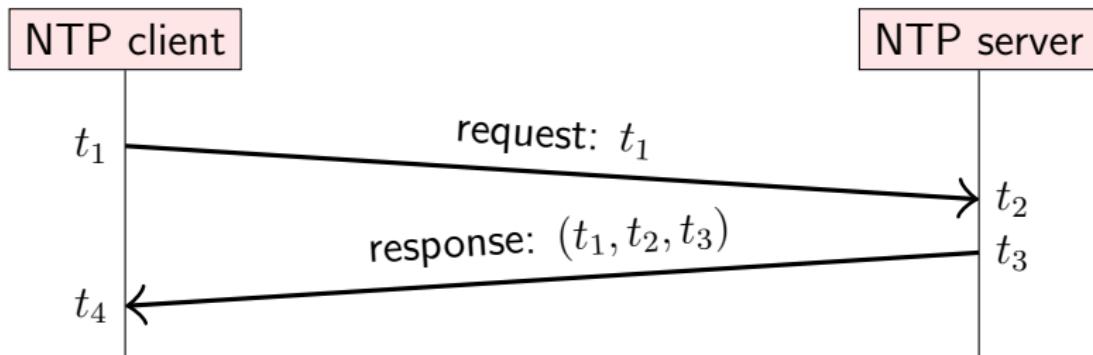
# Estimating time over a network



Round-trip network delay:  $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response:  $t_3 + \frac{\delta}{2}$

# Estimating time over a network



Round-trip network delay:  $\delta = (t_4 - t_1) - (t_3 - t_2)$

Estimated server time when client receives response:  $t_3 + \frac{\delta}{2}$

Estimated clock skew:  $\theta = t_3 + \frac{\delta}{2} - t_4 = \frac{t_2 - t_1 + t_3 - t_4}{2}$

# Correcting clock skew



Once the client has estimated the clock skew  $\theta$ , it needs to apply that correction to its clock.

- ▶ If  $|\theta| < 125$  ms, **slew** the clock:  
slightly speed it up or slow it down by up to 500 ppm  
(brings clocks in sync within  $\approx 5$  minutes)

# Correcting clock skew

Once the client has estimated the clock skew  $\theta$ , it needs to apply that correction to its clock.

- ▶ If  $|\theta| < 125$  ms, **slew** the clock:  
slightly speed it up or slow it down by up to 500 ppm  
(brings clocks in sync within  $\approx 5$  minutes)
- ▶ If  $125 \text{ ms} \leq |\theta| < 1,000 \text{ s}$ , **step** the clock:  
suddenly reset client clock to estimated server timestamp

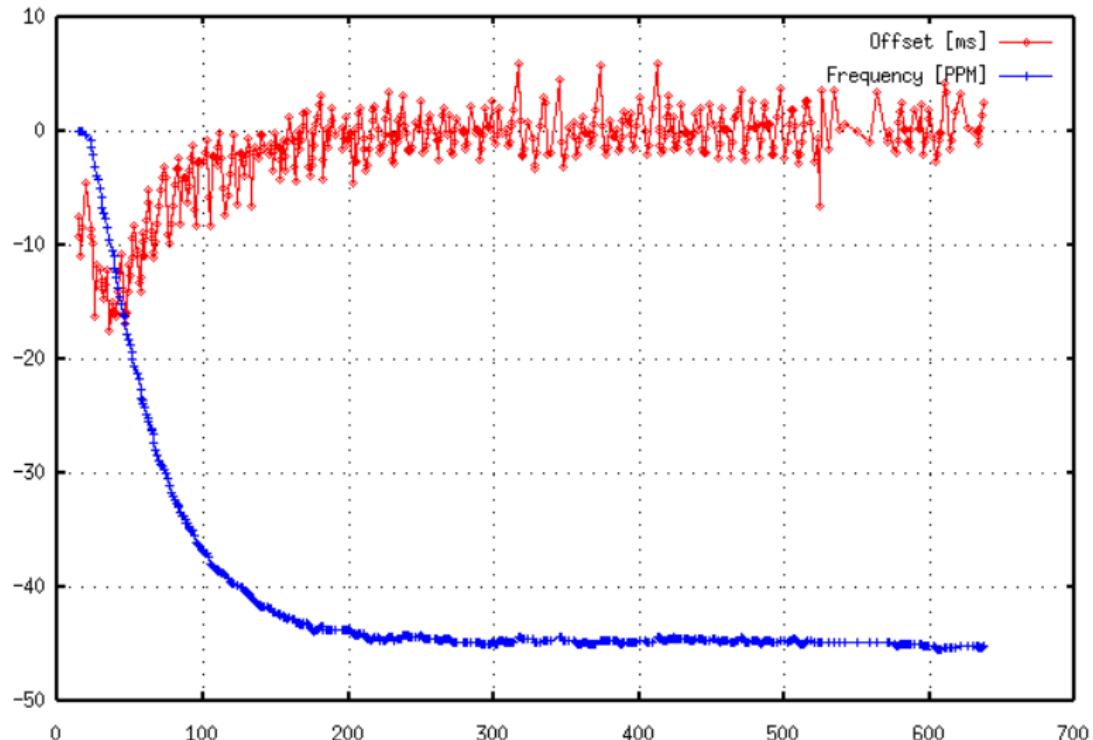
# Correcting clock skew

Once the client has estimated the clock skew  $\theta$ , it needs to apply that correction to its clock.

- ▶ If  $|\theta| < 125$  ms, **slew** the clock:  
slightly speed it up or slow it down by up to 500 ppm  
(brings clocks in sync within  $\approx 5$  minutes)
- ▶ If  $125$  ms  $\leq |\theta| < 1,000$  s, **step** the clock:  
suddenly reset client clock to estimated server timestamp
- ▶ If  $|\theta| \geq 1,000$  s, **panic** and do nothing  
(leave the problem for a human operator to resolve)

Systems that rely on clock sync need to monitor clock skew!

Initial run of NTP 3.5f on HP L2000-44/2



<http://www.ntp.org/ntpfaq/NTP-s-algo.htm>

# Monotonic and time-of-day clocks



// BAD:

```
long startTime = System.currentTimeMillis();
doSomething();

long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

# Monotonic and time-of-day clocks



// BAD:

```
long startTime = System.currentTimeMillis();
doSomething(); ←
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

NTP client steps the clock during this

# Monotonic and time-of-day clocks



// BAD:

```
long startTime = System.currentTimeMillis();
doSomething(); ←
long endTime = System.currentTimeMillis();
long elapsedMillis = endTime - startTime;
// elapsedMillis may be negative!
```

NTP client steps the clock during this

// GOOD:

```
long startTime = System.nanoTime();
doSomething();
long endTime = System.nanoTime();
long elapsedNanos = endTime - startTime;
// elapsedNanos is always >= 0
```

## Time-of-day clock:

- ▶ Time since a fixed date (e.g. 1 January 1970 epoch)

## Monotonic clock:

- ▶ Time since arbitrary point (e.g. when machine booted up)

## Time-of-day clock:

- ▶ Time since a fixed date (e.g. 1 January 1970 epoch)
- ▶ May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments

## Monotonic clock:

- ▶ Time since arbitrary point (e.g. when machine booted up)
- ▶ Always moves forwards at near-constant rate

## Time-of-day clock:

- ▶ Time since a fixed date (e.g. 1 January 1970 epoch)
- ▶ May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments
- ▶ Timestamps can be compared across nodes (if synced)

## Monotonic clock:

- ▶ Time since arbitrary point (e.g. when machine booted up)
- ▶ Always moves forwards at near-constant rate
- ▶ Good for measuring elapsed time on a single node

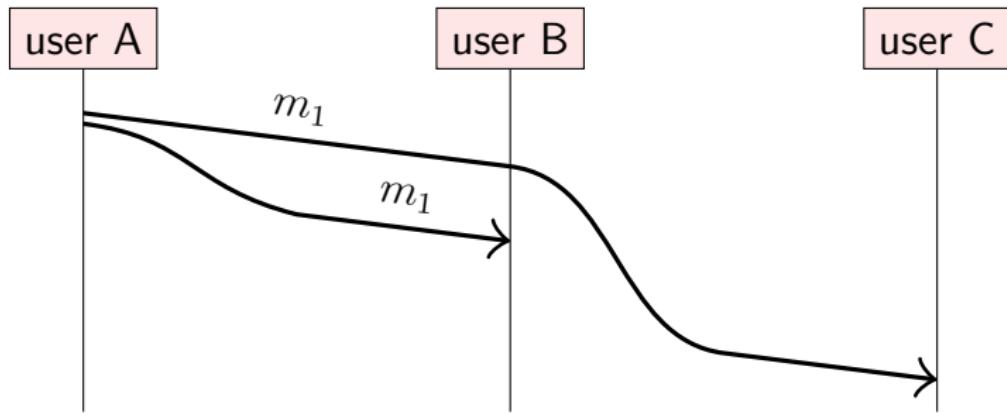
## Time-of-day clock:

- ▶ Time since a fixed date (e.g. 1 January 1970 epoch)
- ▶ May suddenly move forwards or backwards (NTP stepping), subject to leap second adjustments
- ▶ Timestamps can be compared across nodes (if synced)
- ▶ Java: `System.currentTimeMillis()`
- ▶ Linux: `clock_gettime(CLOCK_REALTIME)`

## Monotonic clock:

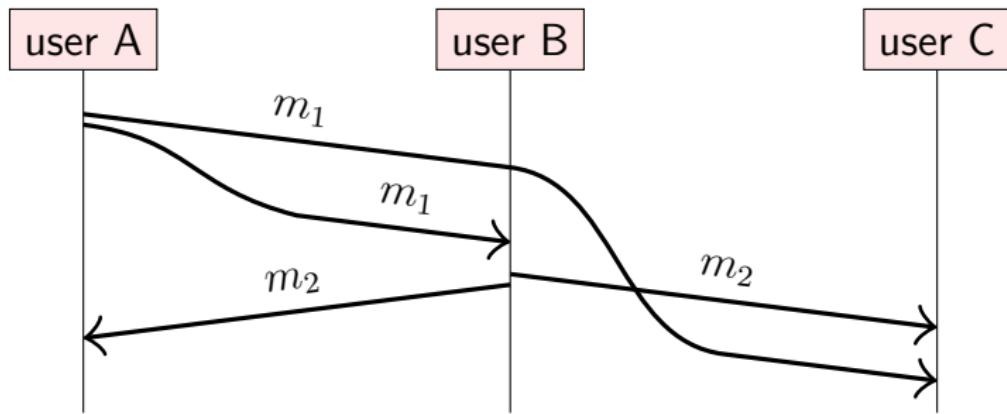
- ▶ Time since arbitrary point (e.g. when machine booted up)
- ▶ Always moves forwards at near-constant rate
- ▶ Good for measuring elapsed time on a single node
- ▶ Java: `System.nanoTime()`
- ▶ Linux: `clock_gettime(CLOCK_MONOTONIC)`

# Ordering of messages



$m_1 = \text{"A says: The moon is made of cheese!"}$

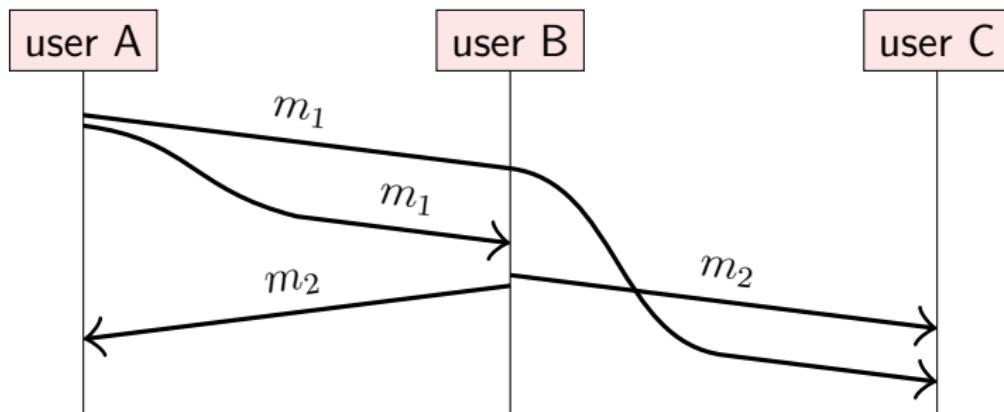
# Ordering of messages



$m_1$  = "A says: The moon is made of cheese!"

$m_2$  = "B says: Oh no it isn't!"

# Ordering of messages

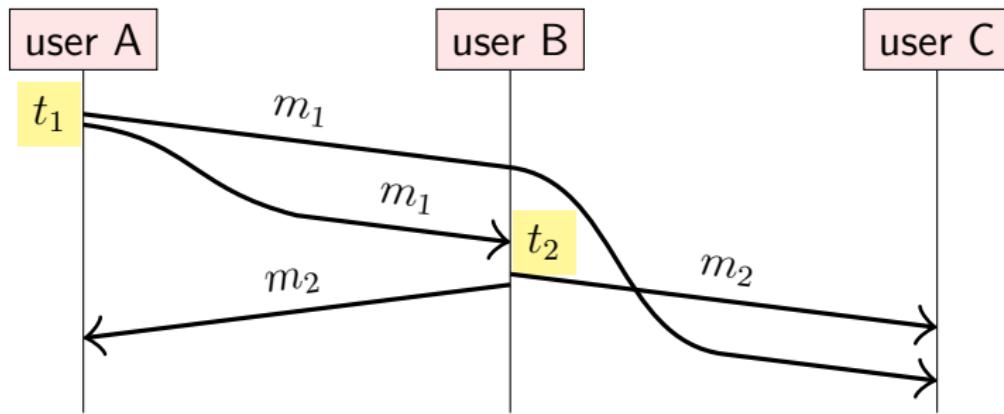


$m_1$  = "A says: The moon is made of cheese!"

$m_2$  = "B says: Oh no it isn't!"

C sees  $m_2$  first,  $m_1$  second,  
even though logically  $m_1$  **happened before**  $m_2$ .

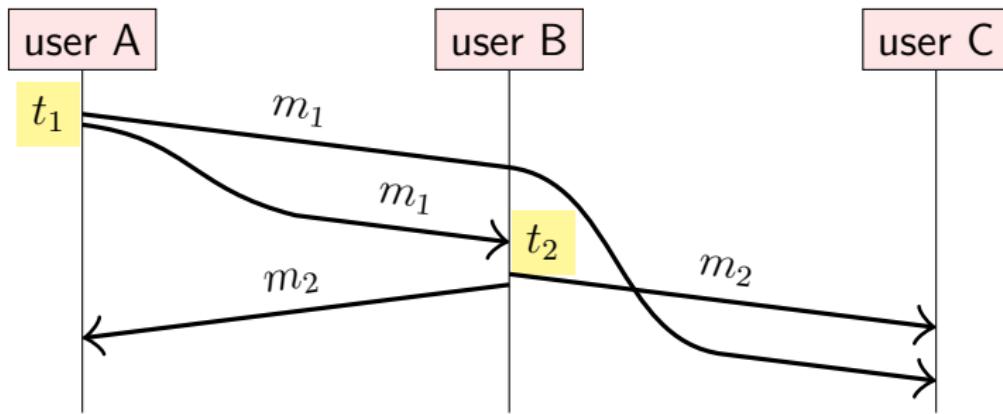
# Ordering of messages using timestamps?



$m_1 = (t_1, \text{"A says: The moon is made of cheese!"})$

$m_2 = (t_2, \text{"B says: Oh no it isn't!"})$

# Ordering of messages using timestamps?



$m_1 = (t_1, \text{"A says: The moon is made of cheese!"})$

$m_2 = (t_2, \text{"B says: Oh no it isn't!"})$

**Problem:** even with synced clocks,  $t_2 < t_1$  is possible.  
Timestamp order is inconsistent with expected order!

# The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event  $a$  **happens before** event  $b$  (written  $a \rightarrow b$ ) iff:

# The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event  $a$  **happens before** event  $b$  (written  $a \rightarrow b$ ) iff:

- ▶  $a$  and  $b$  occurred at the same node, and  $a$  occurred before  $b$  in that node's local execution order; or

# The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event  $a$  **happens before** event  $b$  (written  $a \rightarrow b$ ) iff:

- ▶  $a$  and  $b$  occurred at the same node, and  $a$  occurred before  $b$  in that node's local execution order; or
- ▶ event  $a$  is the sending of some message  $m$ , and event  $b$  is the receipt of that same message  $m$  (assuming sent messages are unique); or

# The happens-before relation

An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event  $a$  **happens before** event  $b$  (written  $a \rightarrow b$ ) iff:

- ▶  $a$  and  $b$  occurred at the same node, and  $a$  occurred before  $b$  in that node's local execution order; or
- ▶ event  $a$  is the sending of some message  $m$ , and event  $b$  is the receipt of that same message  $m$  (assuming sent messages are unique); or
- ▶ there exists an event  $c$  such that  $a \rightarrow c$  and  $c \rightarrow b$ .

# The happens-before relation

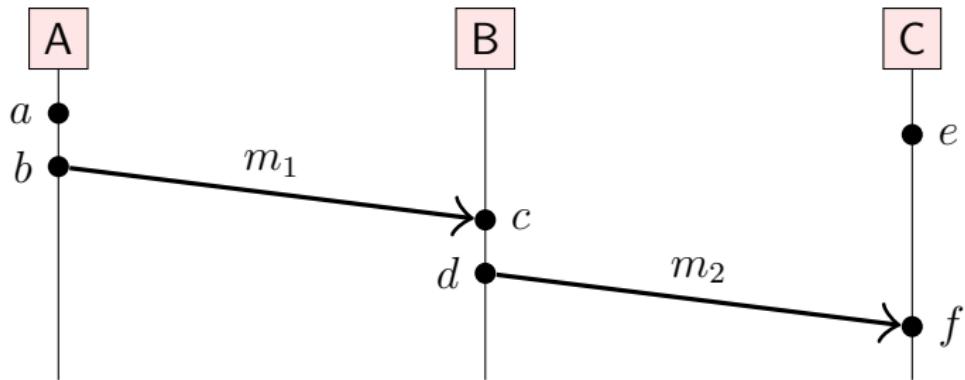
An **event** is something happening at one node (sending or receiving a message, or a local execution step).

We say event  $a$  **happens before** event  $b$  (written  $a \rightarrow b$ ) iff:

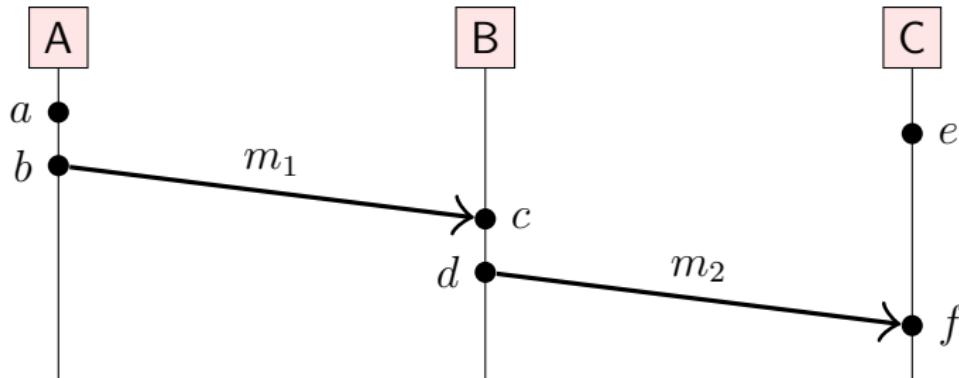
- ▶  $a$  and  $b$  occurred at the same node, and  $a$  occurred before  $b$  in that node's local execution order; or
- ▶ event  $a$  is the sending of some message  $m$ , and event  $b$  is the receipt of that same message  $m$  (assuming sent messages are unique); or
- ▶ there exists an event  $c$  such that  $a \rightarrow c$  and  $c \rightarrow b$ .

The happens-before relation is a partial order: it is possible that neither  $a \rightarrow b$  nor  $b \rightarrow a$ . In that case,  $a$  and  $b$  are **concurrent** (written  $a \parallel b$ ).

# Happens-before relation example

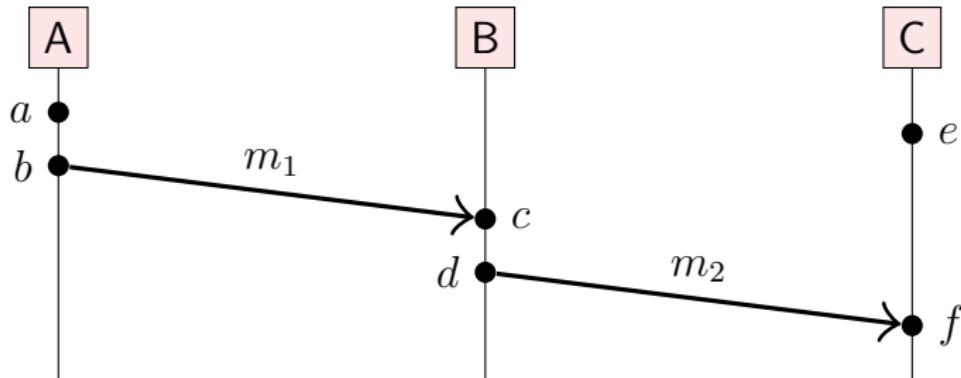


# Happens-before relation example



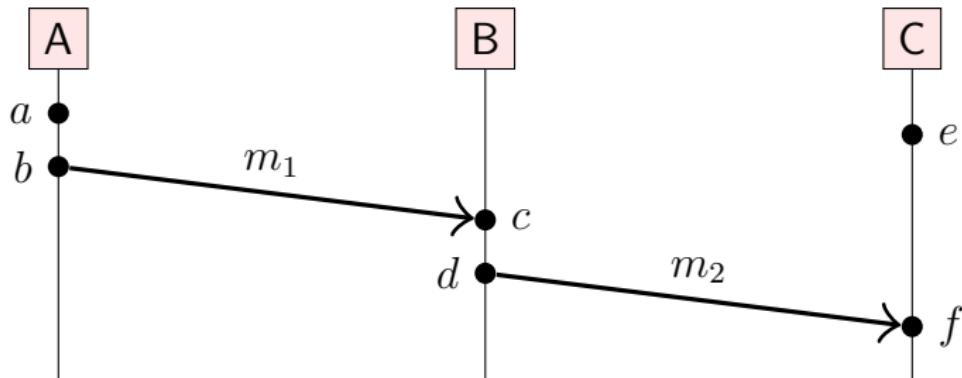
- ▶  $a \rightarrow b$ ,  $c \rightarrow d$ , and  $e \rightarrow f$  due to node execution order

# Happens-before relation example



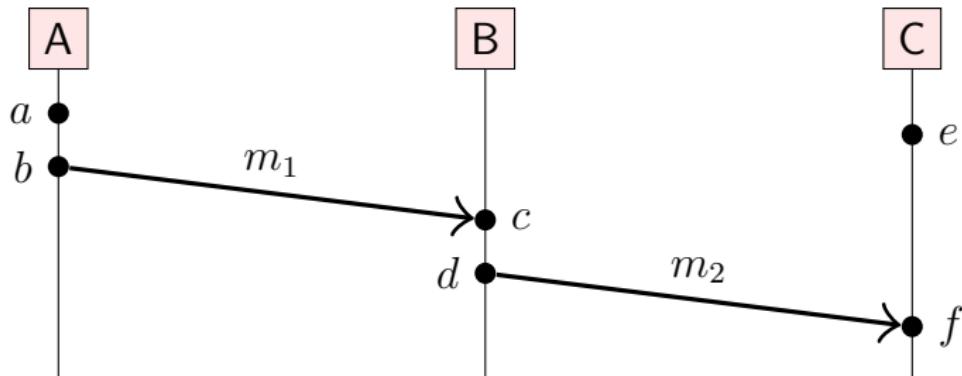
- ▶  $a \rightarrow b$ ,  $c \rightarrow d$ , and  $e \rightarrow f$  due to node execution order
- ▶  $b \rightarrow c$  and  $d \rightarrow f$  due to messages  $m_1$  and  $m_2$

# Happens-before relation example



- ▶  $a \rightarrow b$ ,  $c \rightarrow d$ , and  $e \rightarrow f$  due to node execution order
- ▶  $b \rightarrow c$  and  $d \rightarrow f$  due to messages  $m_1$  and  $m_2$
- ▶  $a \rightarrow c$ ,  $a \rightarrow d$ ,  $a \rightarrow f$ ,  $b \rightarrow d$ ,  $b \rightarrow f$ , and  $c \rightarrow f$  due to transitivity

# Happens-before relation example



- ▶  $a \rightarrow b$ ,  $c \rightarrow d$ , and  $e \rightarrow f$  due to node execution order
- ▶  $b \rightarrow c$  and  $d \rightarrow f$  due to messages  $m_1$  and  $m_2$
- ▶  $a \rightarrow c$ ,  $a \rightarrow d$ ,  $a \rightarrow f$ ,  $b \rightarrow d$ ,  $b \rightarrow f$ , and  $c \rightarrow f$  due to transitivity
- ▶  $a \parallel e$ ,  $b \parallel e$ ,  $c \parallel e$ , and  $d \parallel e$

# Causality

Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

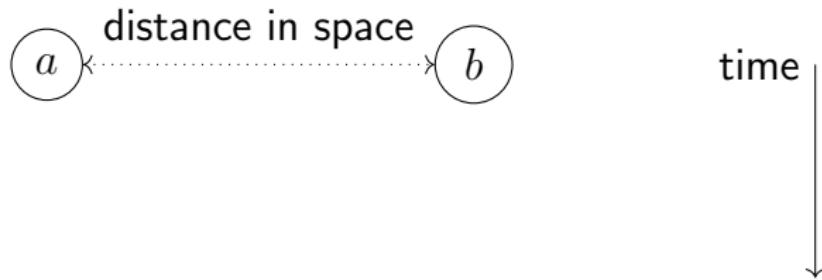
Happens-before relation encodes **potential causality**.

# Causality

Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

Happens-before relation encodes **potential causality**.

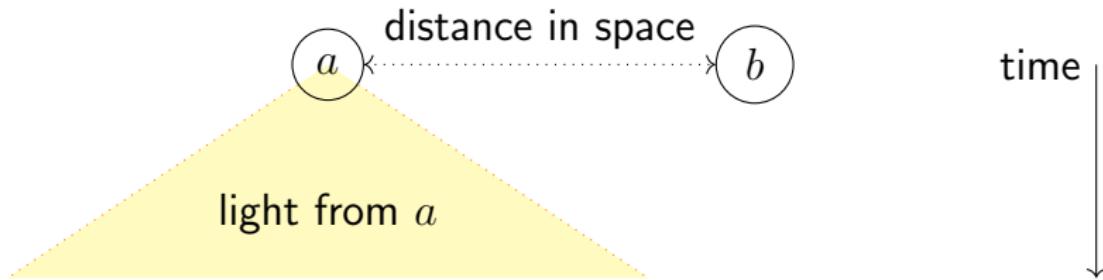


# Causality

Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

Happens-before relation encodes **potential causality**.

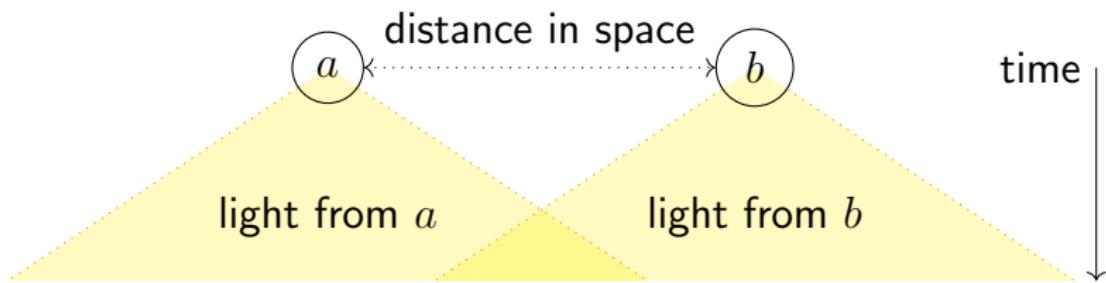


# Causality

Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

Happens-before relation encodes **potential causality**.

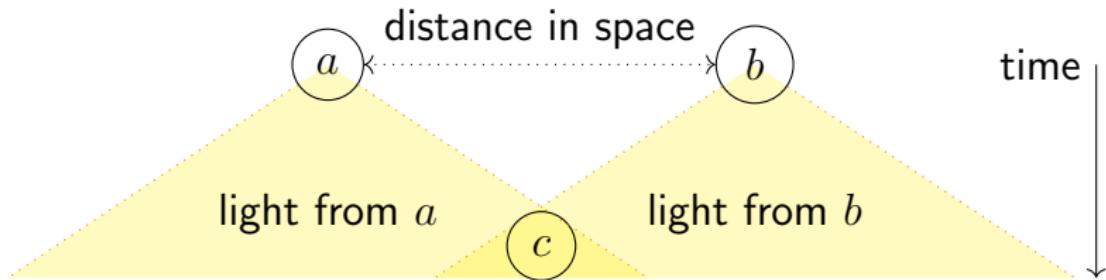


# Causality

Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

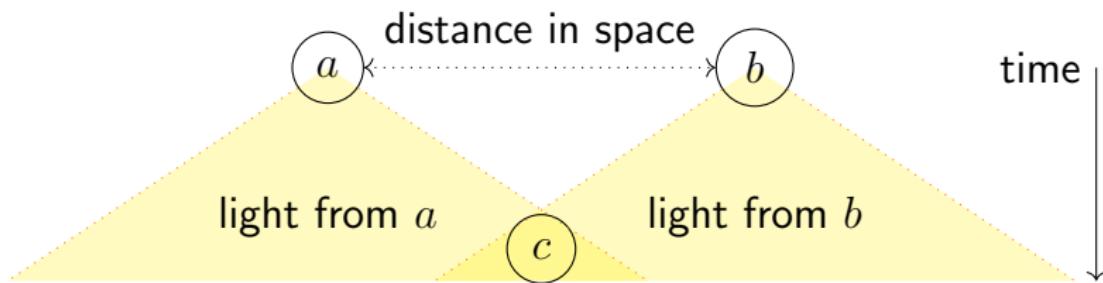
Happens-before relation encodes **potential causality**.



Taken from physics (relativity).

- ▶ When  $a \rightarrow b$ , then  $a$  **might have caused**  $b$ .
- ▶ When  $a \parallel b$ , we know that  $a$  **cannot have caused**  $b$ .

Happens-before relation encodes **potential causality**.



Let  $\prec$  be a strict total order on events.

If  $(a \rightarrow b) \implies (a \prec b)$  then  $\prec$  is a **causal order** (or:  $\prec$  is “consistent with causality”).

**NB.** “causal”  $\neq$  “casual”!

# Broadcast protocols and logical time

Dr. Martin Kleppmann

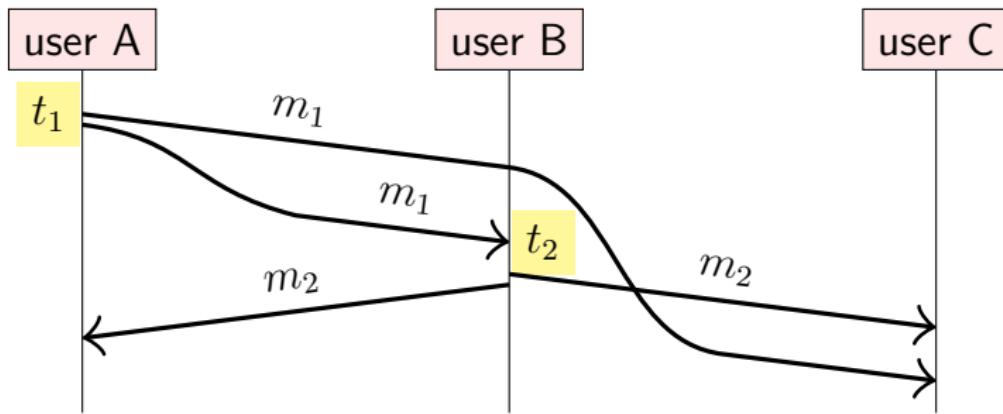
[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

# Timestamps inconsistent with causality



$m_1 = (t_1, \text{"A says: The moon is made of cheese!"})$

$m_2 = (t_2, \text{"B says: Oh no it isn't!"})$

**Problem:** even with synced clocks,  $t_2 < t_1$  is possible.  
Timestamp order is inconsistent with expected order!

# Logical vs. physical clocks

- ▶ Physical clock: count number of **seconds elapsed**
- ▶ Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be  
**inconsistent with causality.**

# Logical vs. physical clocks

- ▶ Physical clock: count number of **seconds elapsed**
- ▶ Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical clocks: designed to **capture causal dependencies**.

$$(e_1 \rightarrow e_2) \implies (T(e_1) < T(e_2))$$

- ▶ Physical clock: count number of **seconds elapsed**
- ▶ Logical clock: count number of **events occurred**

Physical timestamps: useful for many things, but may be **inconsistent with causality**.

Logical clocks: designed to **capture causal dependencies**.

$$(e_1 \rightarrow e_2) \implies (T(e_1) < T(e_2))$$

We will look at two types of logical clocks:

- ▶ Lamport clocks
- ▶ Vector clocks

# Lamport clocks algorithm

**on** initialisation **do**

$t := 0$        $\triangleright$  each node has its own local variable  $t$

**end on**

**on** any event occurring at the local node **do**

$t := t + 1$

**end on**

**on** request to send message  $m$  **do**

$t := t + 1$ ; send  $(t, m)$  via the underlying network link

**end on**

**on** receiving  $(t', m)$  via the underlying network link **do**

$t := \max(t, t') + 1$

deliver  $m$  to the application

**end on**

# Lamport clocks in words

- ▶ Each node maintains a counter  $t$ , incremented on every local event  $e$
- ▶ Let  $L(e)$  be the value of  $t$  after that increment
- ▶ Attach current  $t$  to messages sent over network
- ▶ Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

- ▶ Each node maintains a counter  $t$ , incremented on every local event  $e$
- ▶ Let  $L(e)$  be the value of  $t$  after that increment
- ▶ Attach current  $t$  to messages sent over network
- ▶ Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Properties of this scheme:

- ▶ If  $a \rightarrow b$  then  $L(a) < L(b)$

- ▶ Each node maintains a counter  $t$ , incremented on every local event  $e$
- ▶ Let  $L(e)$  be the value of  $t$  after that increment
- ▶ Attach current  $t$  to messages sent over network
- ▶ Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

Properties of this scheme:

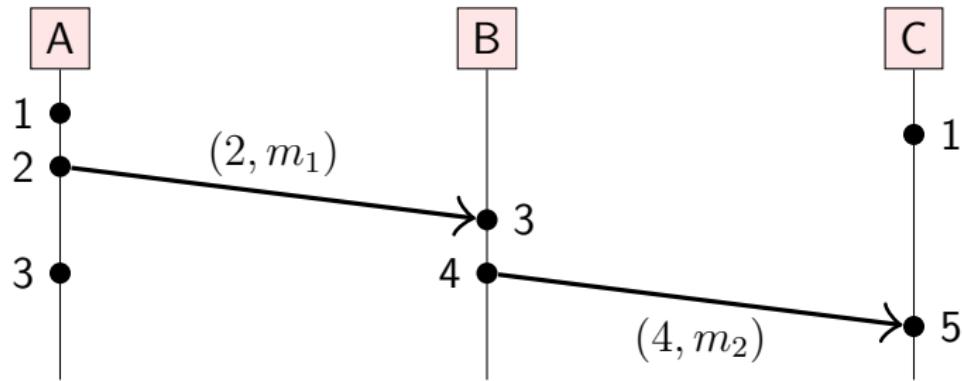
- ▶ If  $a \rightarrow b$  then  $L(a) < L(b)$
- ▶ However,  $L(a) < L(b)$  does not imply  $a \rightarrow b$

- ▶ Each node maintains a counter  $t$ , incremented on every local event  $e$
- ▶ Let  $L(e)$  be the value of  $t$  after that increment
- ▶ Attach current  $t$  to messages sent over network
- ▶ Recipient moves its clock forward to timestamp in the message (if greater than local counter), then increments

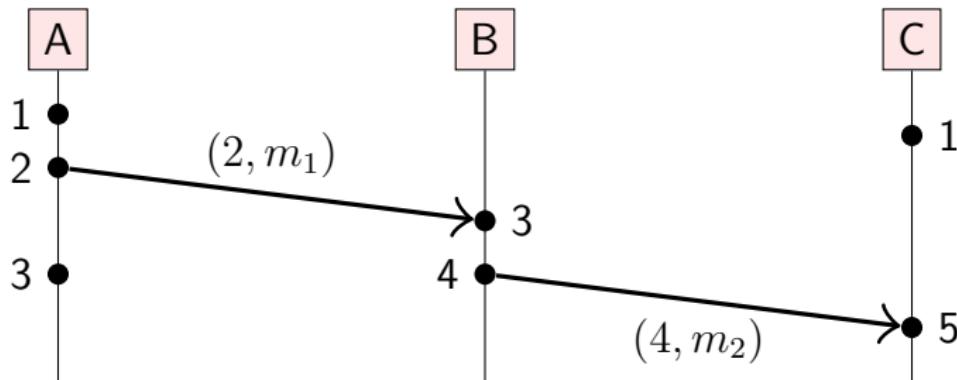
Properties of this scheme:

- ▶ If  $a \rightarrow b$  then  $L(a) < L(b)$
- ▶ However,  $L(a) < L(b)$  does not imply  $a \rightarrow b$
- ▶ Possible that  $L(a) = L(b)$  for  $a \neq b$

# Lamport clocks example

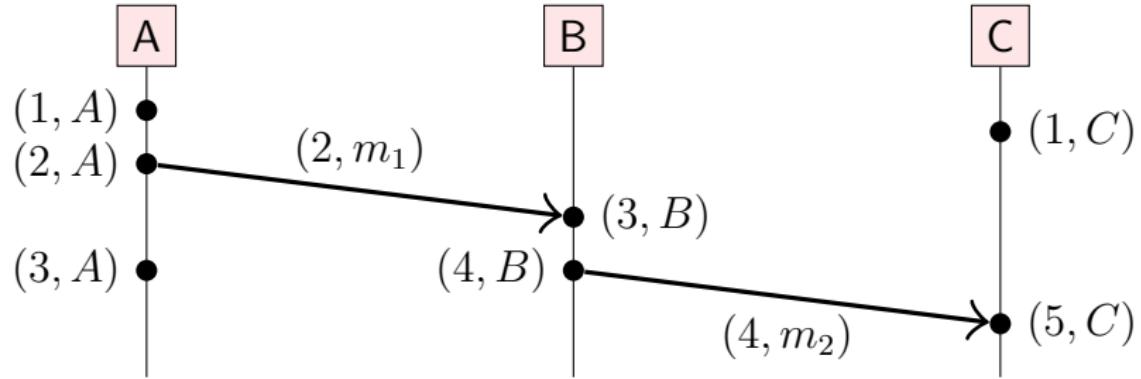


# Lamport clocks example



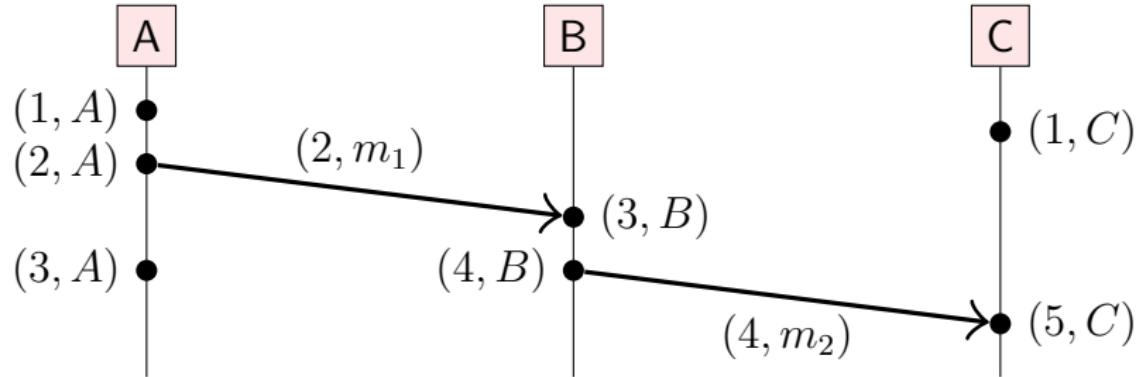
Let  $N(e)$  be the node at which event  $e$  occurred.  
Then the pair  $(L(e), N(e))$  **uniquely identifies** event  $e$ .

# Lamport clocks example



Let  $N(e)$  be the node at which event  $e$  occurred.  
Then the pair  $(L(e), N(e))$  **uniquely identifies** event  $e$ .

# Lamport clocks example



Let  $N(e)$  be the node at which event  $e$  occurred.  
Then the pair  $(L(e), N(e))$  **uniquely identifies** event  $e$ .

Define a **total order**  $\prec$  using Lamport timestamps:

$$(a \prec b) \iff (L(a) < L(b) \text{ or } (L(a) = L(b) \text{ and } N(a) < N(b)))$$

This order is **causal**:  $(a \rightarrow b) \implies (a \prec b)$

Given Lamport timestamps  $L(a)$  and  $L(b)$  with  $L(a) < L(b)$  we can't tell whether  $a \rightarrow b$  or  $a \parallel b$ .

If we want to detect which events are concurrent, we need **vector clocks**:

Given Lamport timestamps  $L(a)$  and  $L(b)$  with  $L(a) < L(b)$  we can't tell whether  $a \rightarrow b$  or  $a \parallel b$ .

If we want to detect which events are concurrent, we need **vector clocks**:

- ▶ Assume  $n$  nodes in the system,  $N = \langle N_0, N_1, \dots, N_{n-1} \rangle$

Given Lamport timestamps  $L(a)$  and  $L(b)$  with  $L(a) < L(b)$  we can't tell whether  $a \rightarrow b$  or  $a \parallel b$ .

If we want to detect which events are concurrent, we need **vector clocks**:

- ▶ Assume  $n$  nodes in the system,  $N = \langle N_0, N_1, \dots, N_{n-1} \rangle$
- ▶ Vector timestamp of event  $a$  is  $V(a) = \langle t_0, t_1, \dots, t_{n-1} \rangle$
- ▶  $t_i$  is number of events observed by node  $N_i$

Given Lamport timestamps  $L(a)$  and  $L(b)$  with  $L(a) < L(b)$  we can't tell whether  $a \rightarrow b$  or  $a \parallel b$ .

If we want to detect which events are concurrent, we need **vector clocks**:

- ▶ Assume  $n$  nodes in the system,  $N = \langle N_0, N_1, \dots, N_{n-1} \rangle$
- ▶ Vector timestamp of event  $a$  is  $V(a) = \langle t_0, t_1, \dots, t_{n-1} \rangle$
- ▶  $t_i$  is number of events observed by node  $N_i$
- ▶ Each node has a current vector timestamp  $T$
- ▶ On event at node  $N_i$ , increment vector element  $T[i]$

Given Lamport timestamps  $L(a)$  and  $L(b)$  with  $L(a) < L(b)$  we can't tell whether  $a \rightarrow b$  or  $a \parallel b$ .

If we want to detect which events are concurrent, we need **vector clocks**:

- ▶ Assume  $n$  nodes in the system,  $N = \langle N_0, N_1, \dots, N_{n-1} \rangle$
- ▶ Vector timestamp of event  $a$  is  $V(a) = \langle t_0, t_1, \dots, t_{n-1} \rangle$
- ▶  $t_i$  is number of events observed by node  $N_i$
- ▶ Each node has a current vector timestamp  $T$
- ▶ On event at node  $N_i$ , increment vector element  $T[i]$
- ▶ Attach current vector timestamp to each message
- ▶ Recipient merges message vector into its local vector

# Vector clocks algorithm

**on** initialisation at node  $N_i$  **do**

$T := \langle 0, 0, \dots, 0 \rangle$

▷ local variable at node  $N_i$

**end on**

**on** any event occurring at node  $N_i$  **do**

$T[i] := T[i] + 1$

**end on**

**on** request to send message  $m$  at node  $N_i$  **do**

$T[i] := T[i] + 1$ ; send  $(T, m)$  via network

**end on**

**on** receiving  $(T', m)$  at node  $N_i$  via the network **do**

$T[j] := \max(T[j], T'[j])$  for every  $j \in \{1, \dots, n\}$

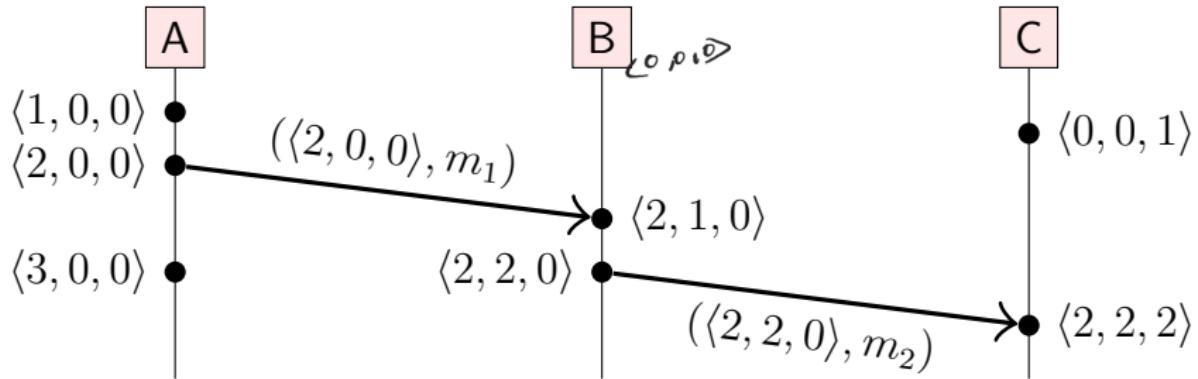
$T[i] := T[i] + 1$ ; deliver  $m$  to the application

**end on**

# Vector clocks example

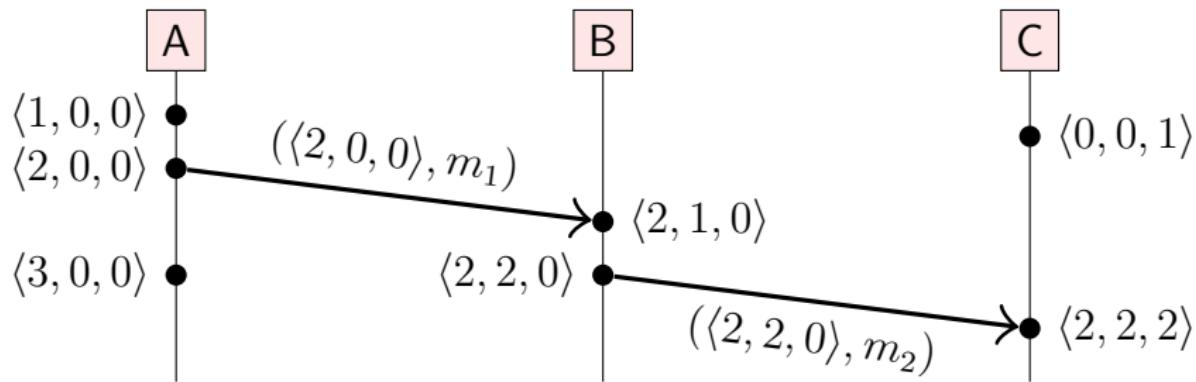
Assuming the vector of nodes is  $N = \langle A, B, C \rangle$ :

$$\langle 0, 0, 0 \rangle$$



# Vector clocks example

Assuming the vector of nodes is  $N = \langle A, B, C \rangle$ :



The vector timestamp of an event  $e$  represents a set of events,  $e$  and its causal dependencies:  $\{e\} \cup \{a \mid a \rightarrow e\}$

*2 events from A and B*

For example,  $\langle 2, 2, 0 \rangle$  represents the first two events from A, the first two events from B, and no events from C.

# Vector clocks ordering

Define the following order on vector timestamps

(in a system with  $n$  nodes):

$$\langle 1, 1, 2 \rangle \leq \langle 1, 1, 2 \rangle$$

►  $T = T'$  iff  $T[i] = T'[i]$  for all  $i \in \{1^0, \dots, n\}^{n-1}$

►  $T \leq T'$  iff  $T[i] \leq T'[i]$  for all  $i \in \{1^0, \dots, n\}^{n-1}$

►  $T < T'$  iff  $T \leq T'$  and  $T \neq T'$

►  $T \parallel T'$  iff  $T \not\leq T'$  and  $T' \not\leq T$      $\langle 1, 0, 0 \rangle \parallel \langle 0, 1, 0 \rangle$

can't compare it

incomparable

# Vector clocks ordering

Define the following order on vector timestamps  
(in a system with  $n$  nodes):

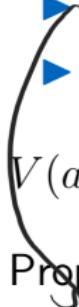
- ▶  $T = T'$  iff  $T[i] = T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T \leq T'$  iff  $T[i] \leq T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T < T'$  iff  $T \leq T'$  and  $T \neq T'$
- ▶  $T \parallel T'$  iff  $T \not\leq T'$  and  $T' \not\leq T$

$$V(a) \leq V(b) \text{ iff } (\{a\} \cup \{e \mid e \rightarrow a\}) \subseteq (\{b\} \cup \{e \mid e \rightarrow b\})$$

# Vector clocks ordering

Define the following order on vector timestamps  
(in a system with  $n$  nodes):

- ▶  $T = T'$  iff  $T[i] = T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T \leq T'$  iff  $T[i] \leq T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T < T'$  iff  $T \leq T'$  and  $T \neq T'$
- ▶  $T \parallel T'$  iff  $T \not\leq T'$  and  $T' \not\leq T$


$$V(a) \leq V(b) \text{ iff } (\{a\} \cup \{e \mid e \rightarrow a\}) \subseteq (\{b\} \cup \{e \mid e \rightarrow b\})$$

Properties of this order:

- ▶  $(V(a) < V(b)) \iff (a \rightarrow b)$
- ▶  $(V(a) = V(b)) \iff (a = b)$
- ▶  $(V(a) \parallel V(b)) \iff (a \parallel b)$

# Broadcast protocols

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic
- ▶ If one node is faulty, remaining group members carry on

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic
- ▶ If one node is faulty, remaining group members carry on
- ▶ Note: concept is more general than IP multicast  
(we build upon point-to-point messaging)

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic
- ▶ If one node is faulty, remaining group members carry on
- ▶ Note: concept is more general than IP multicast  
(we build upon point-to-point messaging)

Build upon system models from earlier lecture:

- ▶ Can be **best-effort** (may drop messages) or  
**reliable** (non-faulty nodes deliver every message,  
by retransmitting dropped messages)

Broadcast (multicast) is **group communication**:

- ▶ One node sends message, all nodes in group deliver it
- ▶ Set of group members may be fixed (static) or dynamic
- ▶ If one node is faulty, remaining group members carry on
- ▶ Note: concept is more general than IP multicast  
(we build upon point-to-point messaging)

Build upon system models from earlier lecture:

- ▶ Can be **best-effort** (may drop messages) or **reliable** (non-faulty nodes deliver every message, by retransmitting dropped messages)
- ▶ Asynchronous/partially synchronous timing model  
⇒ **no upper bound** on message latency

# Receiving versus delivering

Node A:

Application

Broadcast algorithm  
(middleware)

Node B:

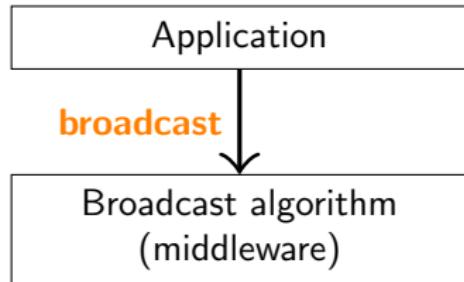
Application

Broadcast algorithm  
(middleware)

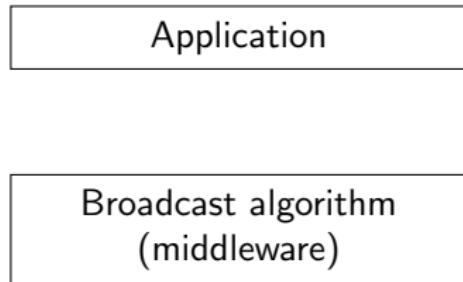
Network

# Receiving versus delivering

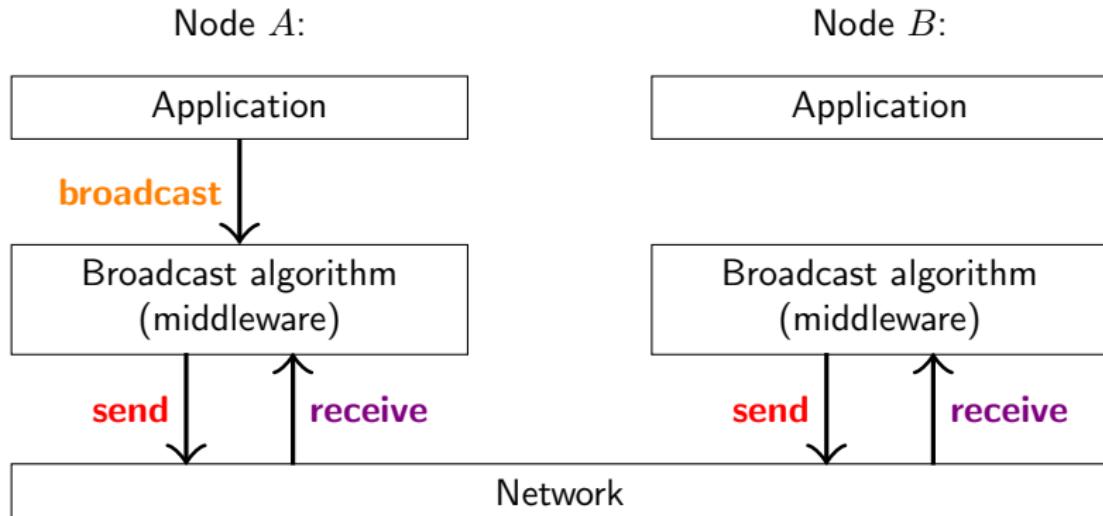
Node A:



Node B:

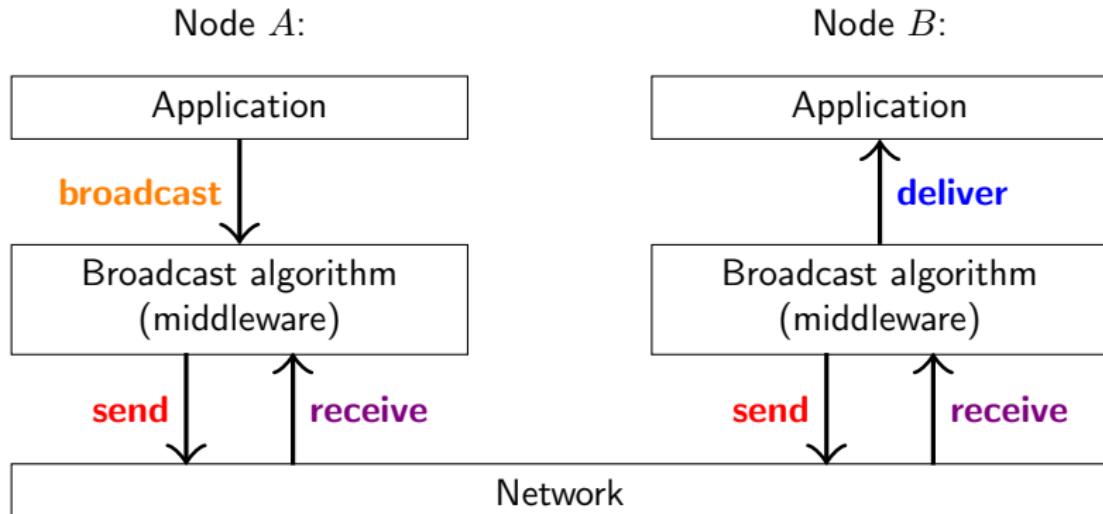


# Receiving versus delivering



Assume network provides point-to-point **send/receive**

# Receiving versus delivering



Assume network provides point-to-point **send/receive**

After broadcast algorithm **receives** message from network, it may buffer/queue it before **delivering** to the application

## FIFO broadcast:

If  $m_1$  and  $m_2$  are broadcast by the same node, and  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$ , then  $m_1$  must be delivered before  $m_2$

## FIFO broadcast:

If  $m_1$  and  $m_2$  are broadcast by the same node, and  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$ , then  $m_1$  must be delivered before  $m_2$

## Causal broadcast:

If  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$  then  $m_1$  must be delivered before  $m_2$

## FIFO broadcast:

If  $m_1$  and  $m_2$  are broadcast by the same node, and  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$ , then  $m_1$  must be delivered before  $m_2$

## Causal broadcast:

If  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$  then  $m_1$  must be delivered before  $m_2$

## Total order broadcast:

If  $m_1$  is delivered before  $m_2$  on one node, then  $m_1$  must be delivered before  $m_2$  on all nodes

## FIFO broadcast:

If  $m_1$  and  $m_2$  are broadcast by the same node, and  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$ , then  $m_1$  must be delivered before  $m_2$

## Causal broadcast:

If  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$  then  $m_1$  must be delivered before  $m_2$

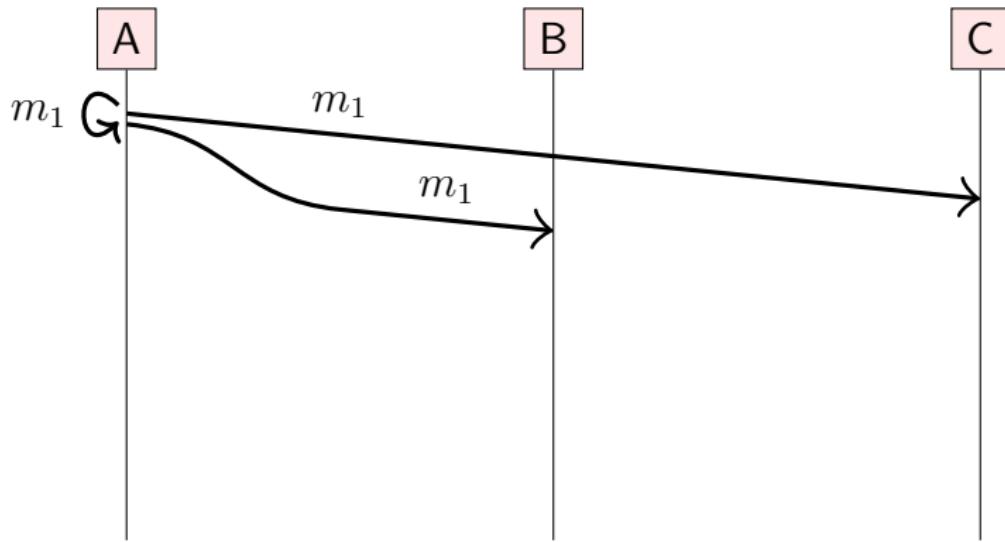
## Total order broadcast:

If  $m_1$  is delivered before  $m_2$  on one node, then  $m_1$  must be delivered before  $m_2$  on all nodes

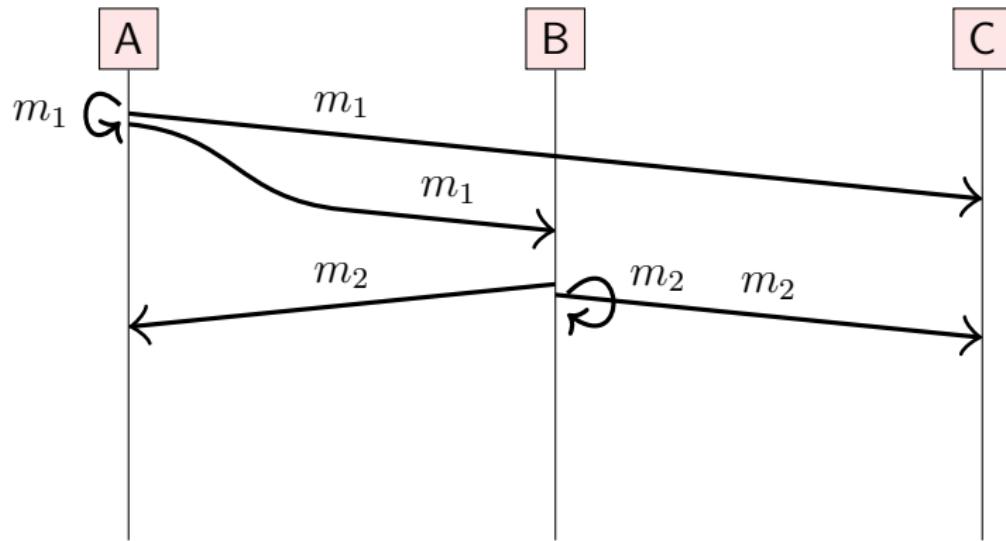
## FIFO-total order broadcast:

Combination of FIFO broadcast and total order broadcast

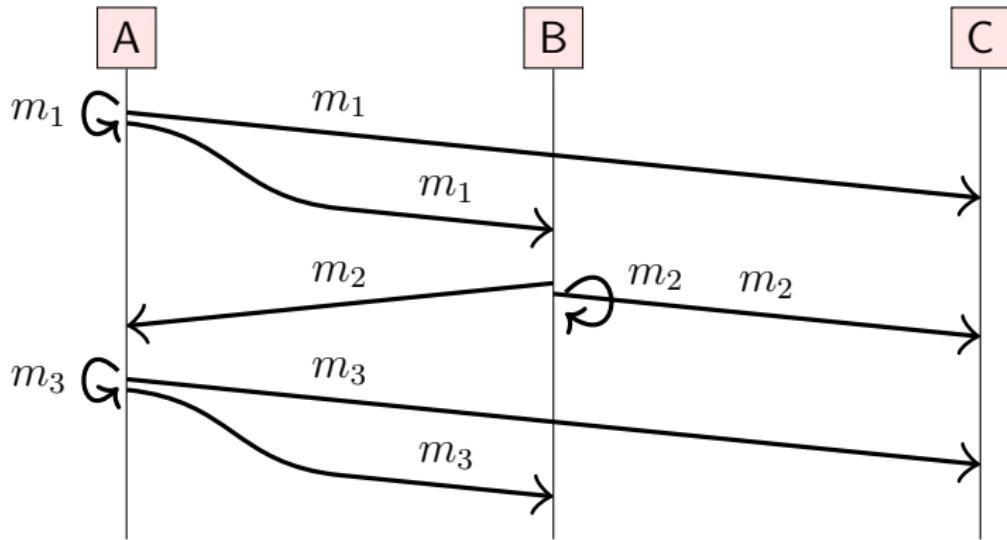
# FIFO broadcast



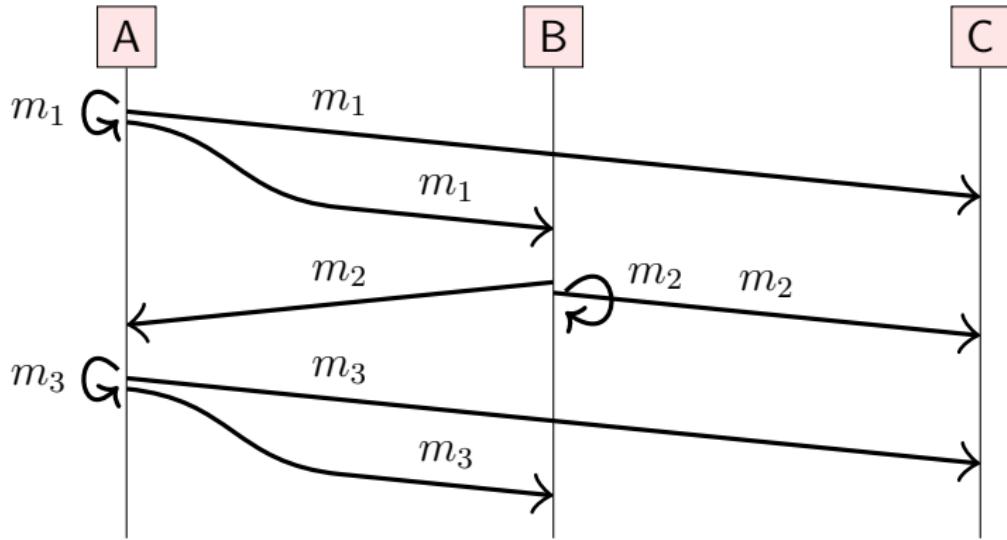
# FIFO broadcast



# FIFO broadcast

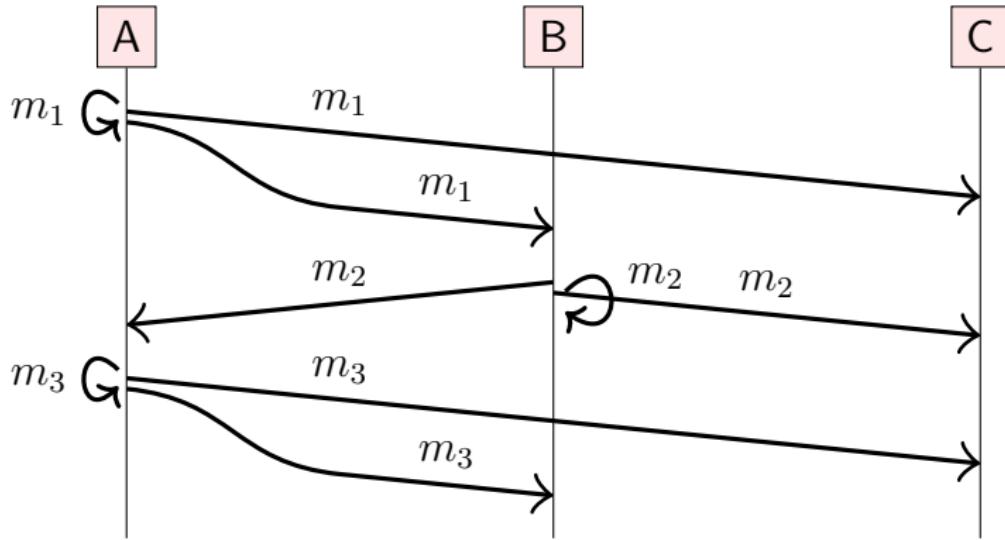


# FIFO broadcast



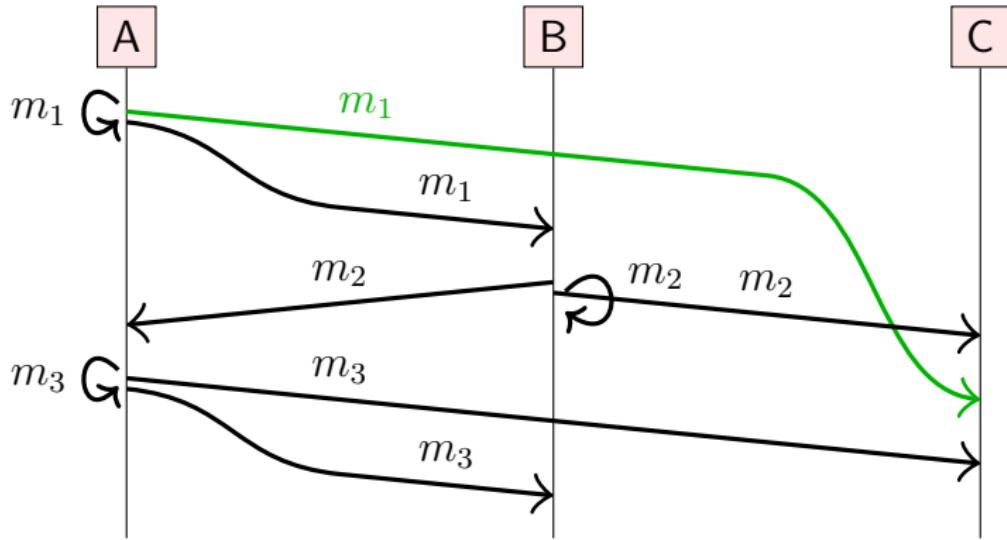
Messages sent by the same node must be delivered in the order they were sent.

Messages sent by different nodes can be delivered in any order.



Messages sent by the same node must be delivered in the order they were sent.

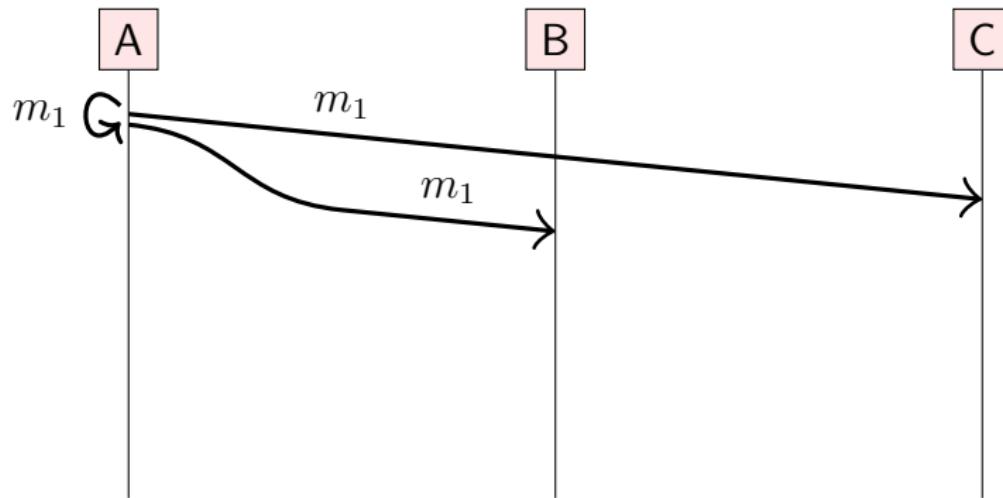
Messages sent by different nodes can be delivered in any order.  
Valid orders:  $(m_2, m_1, m_3)$  or  $(m_1, m_2, m_3)$  or  $(m_1, m_3, m_2)$



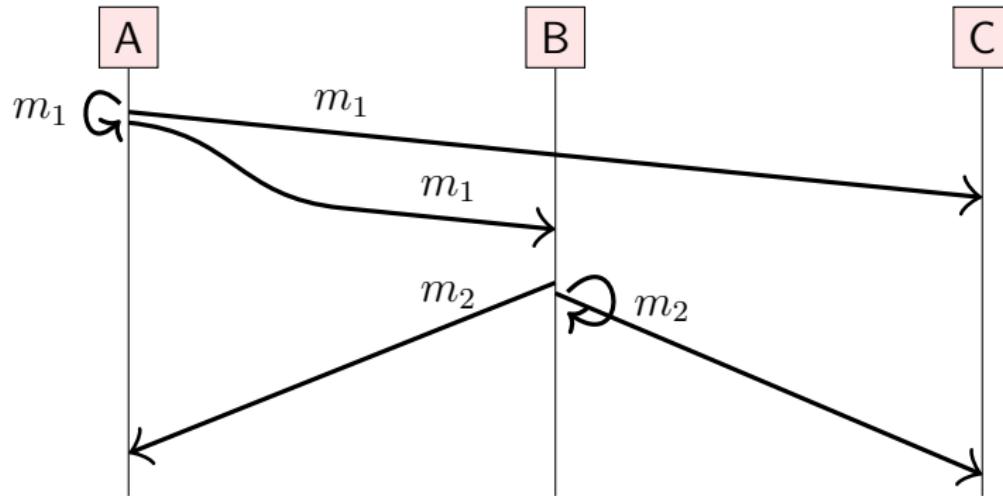
Messages sent by the same node must be delivered in the order they were sent.

Messages sent by different nodes can be delivered in any order.  
Valid orders:  $(m_2, m_1, m_3)$  or  $(m_1, m_2, m_3)$  or  $(m_1, m_3, m_2)$

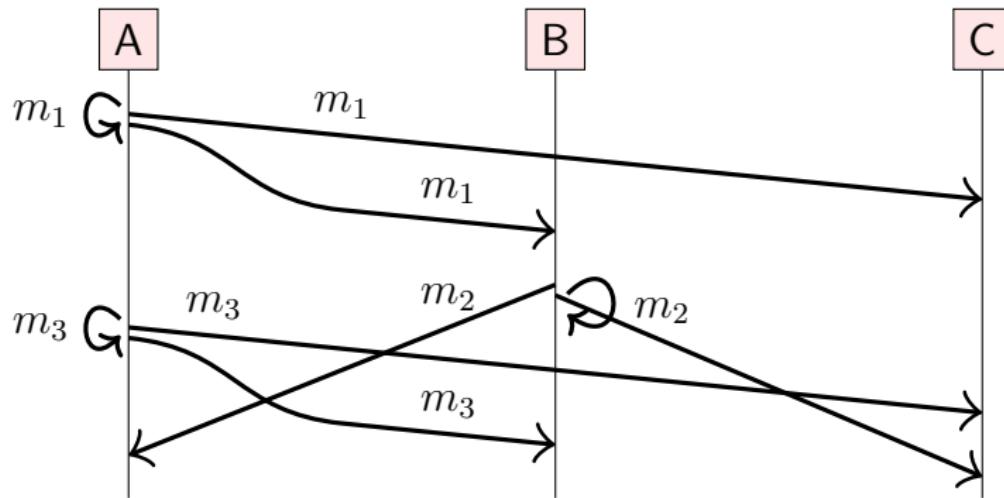
# Causal broadcast



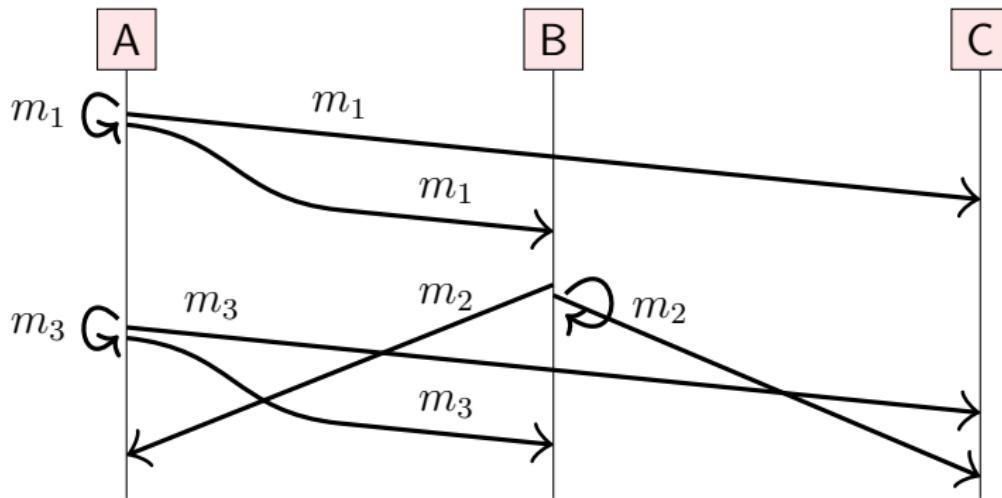
# Causal broadcast



# Causal broadcast

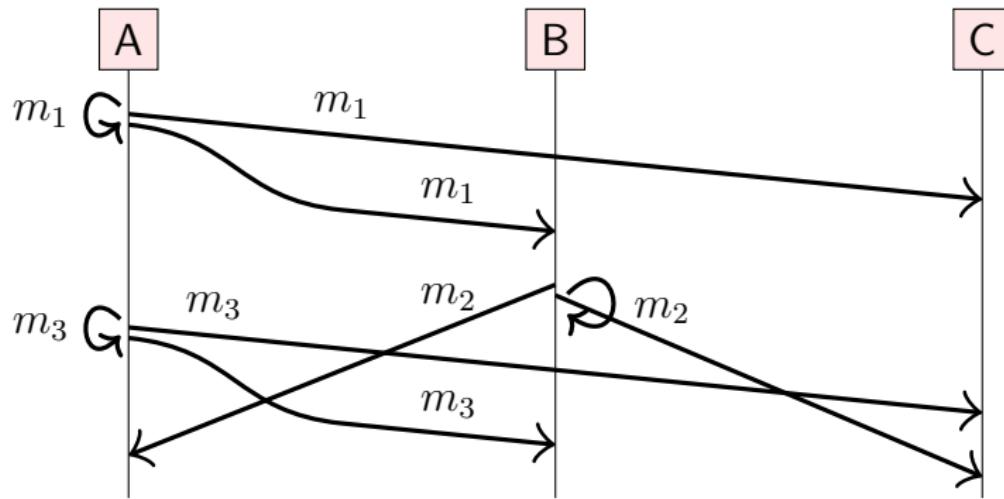


# Causal broadcast



Causally related messages must be delivered in causal order.  
Concurrent messages can be delivered in any order.

# Causal broadcast



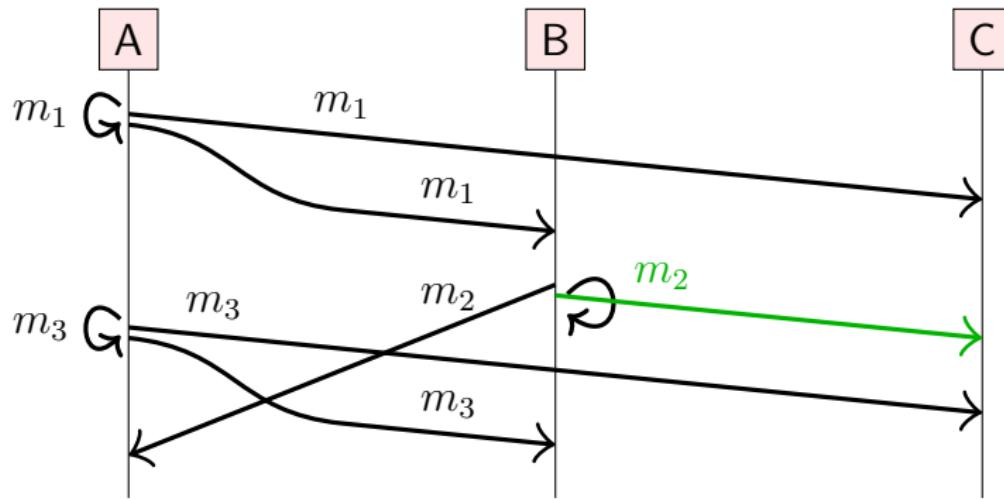
Causally related messages must be delivered in causal order.  
Concurrent messages can be delivered in any order.

Here:  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$  and

$\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_3)$

$\Rightarrow$  valid orders are:  $(m_1, m_2, m_3)$  or  $(m_1, m_3, m_2)$

# Causal broadcast



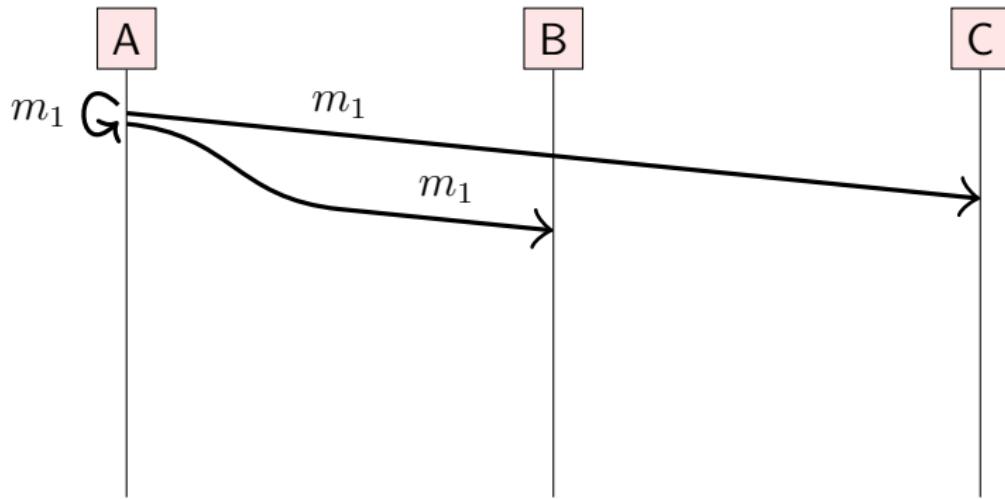
Causally related messages must be delivered in causal order.  
Concurrent messages can be delivered in any order.

Here:  $\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_2)$  and

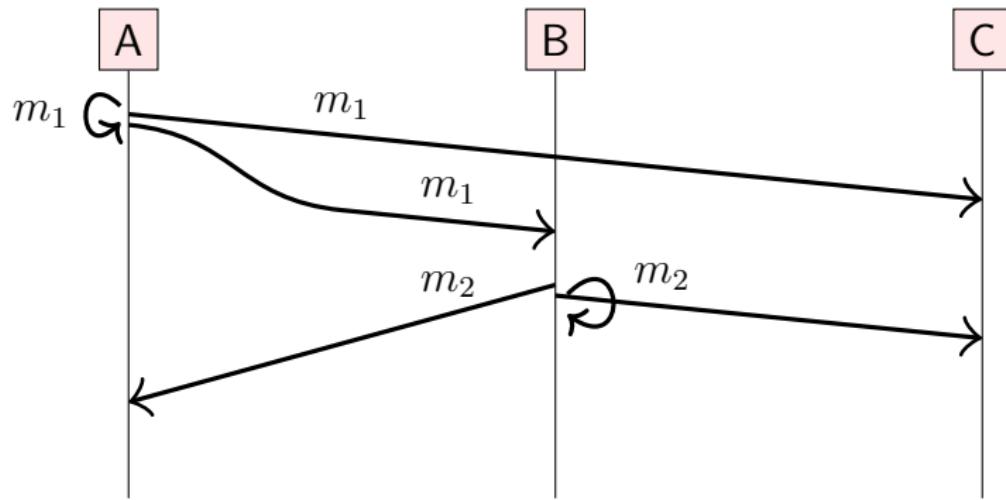
$\text{broadcast}(m_1) \rightarrow \text{broadcast}(m_3)$

$\Rightarrow$  valid orders are:  $(m_1, m_2, m_3)$  or  $(m_1, m_3, m_2)$

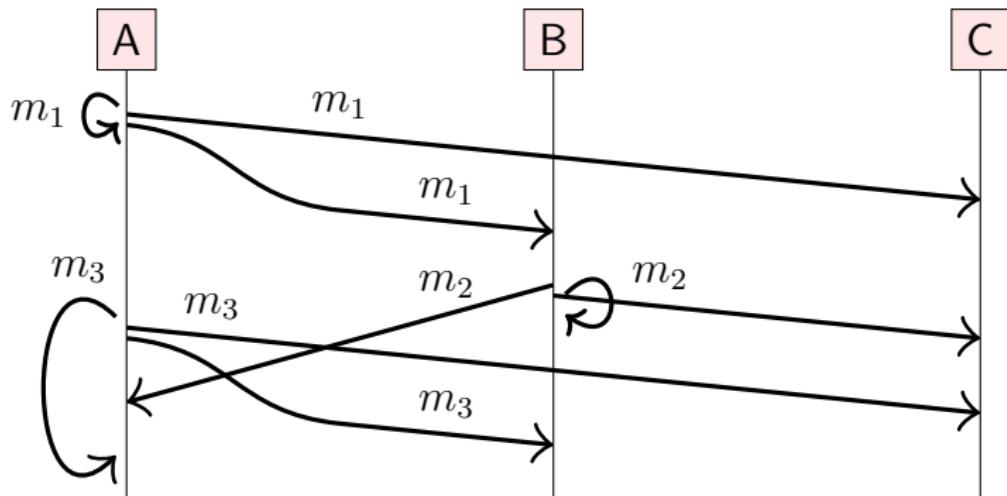
# Total order broadcast (1)



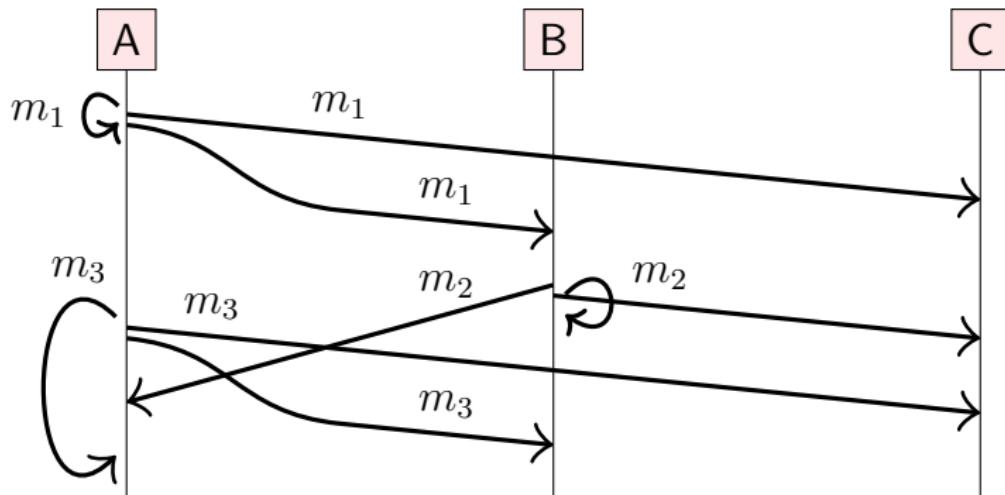
# Total order broadcast (1)



# Total order broadcast (1)

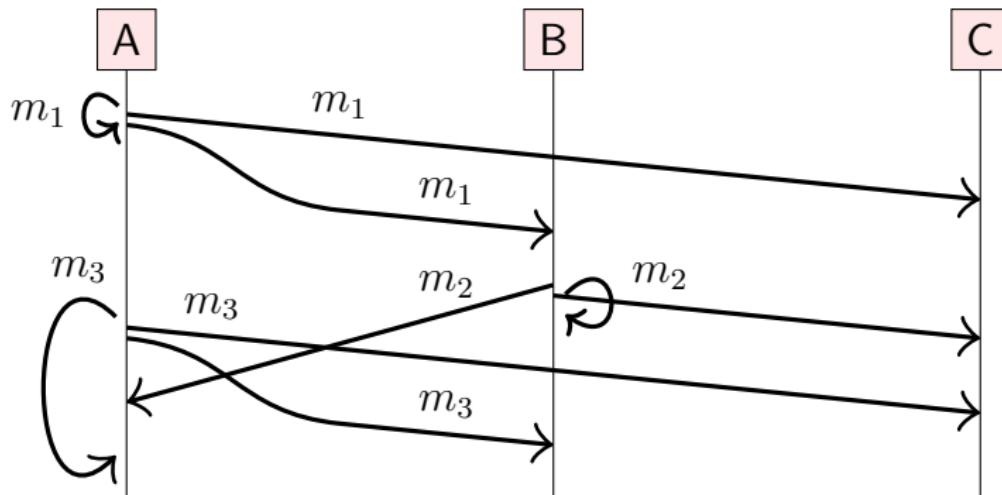


# Total order broadcast (1)



All nodes must deliver messages in **the same** order  
(here:  $m_1, m_2, m_3$ )

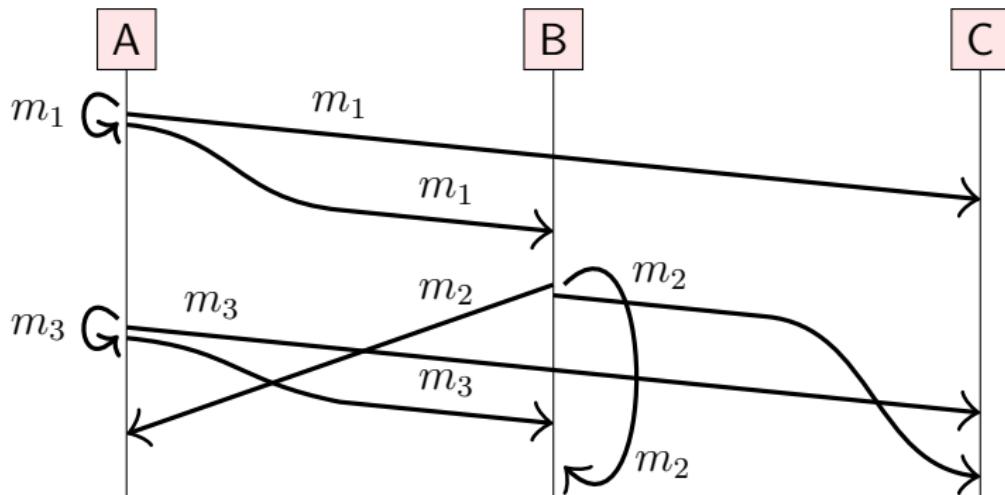
# Total order broadcast (1)



All nodes must deliver messages in **the same** order  
(here:  $m_1, m_2, m_3$ )

This includes a node's deliveries to itself!

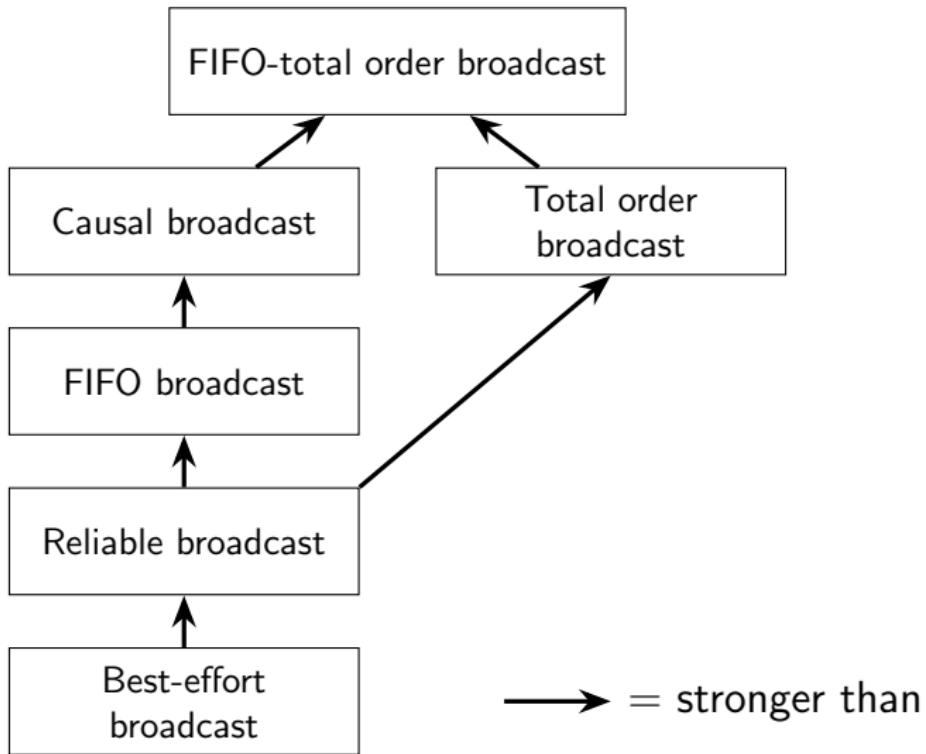
# Total order broadcast (2)



All nodes must deliver messages in **the same** order  
(here:  $m_1, m_3, m_2$ )

This includes a node's deliveries to itself!

# Relationships between broadcast models



# Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

# Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

First attempt: **broadcasting node sends message directly to every other node**

- ▶ Use reliable links (retry + deduplicate)

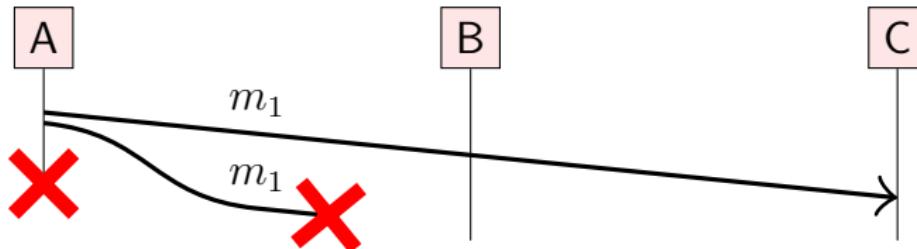
# Broadcast algorithms

Break down into two layers:

1. Make best-effort broadcast reliable by retransmitting dropped messages
2. Enforce delivery order on top of reliable broadcast

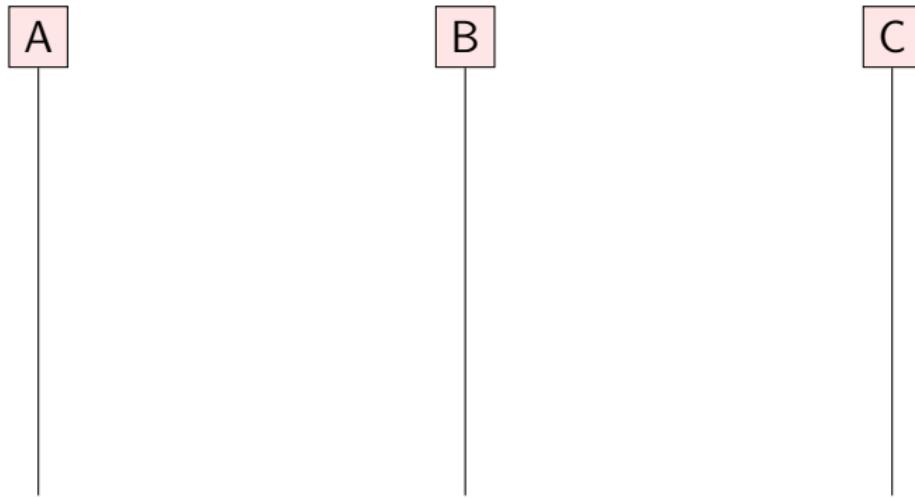
First attempt: **broadcasting node sends message directly to every other node**

- ▶ Use reliable links (retry + deduplicate)
- ▶ Problem: node may crash before all messages delivered



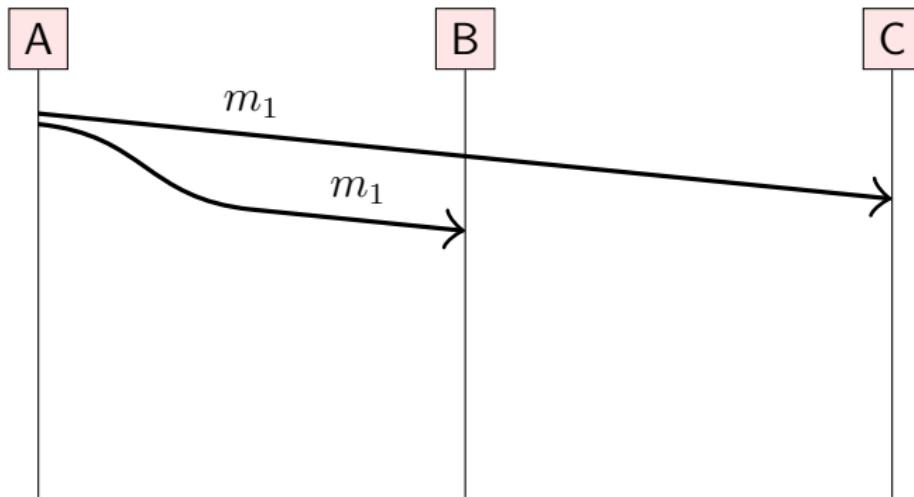
# Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



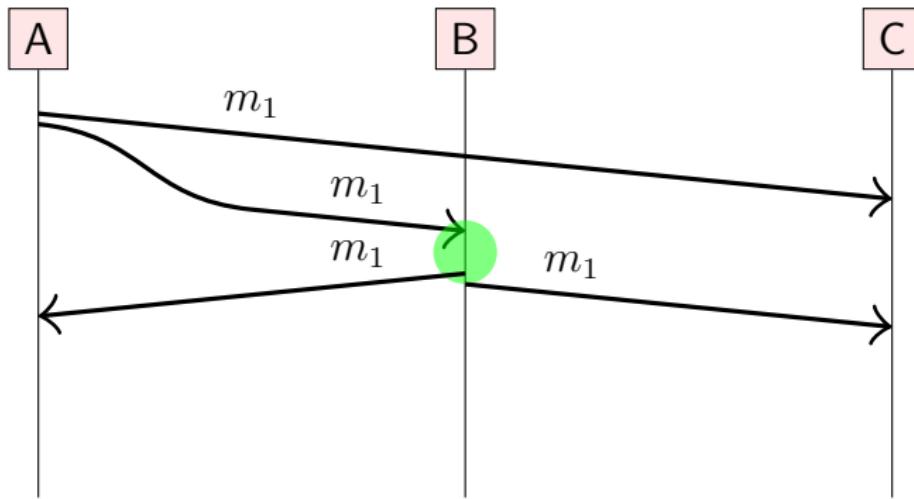
# Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



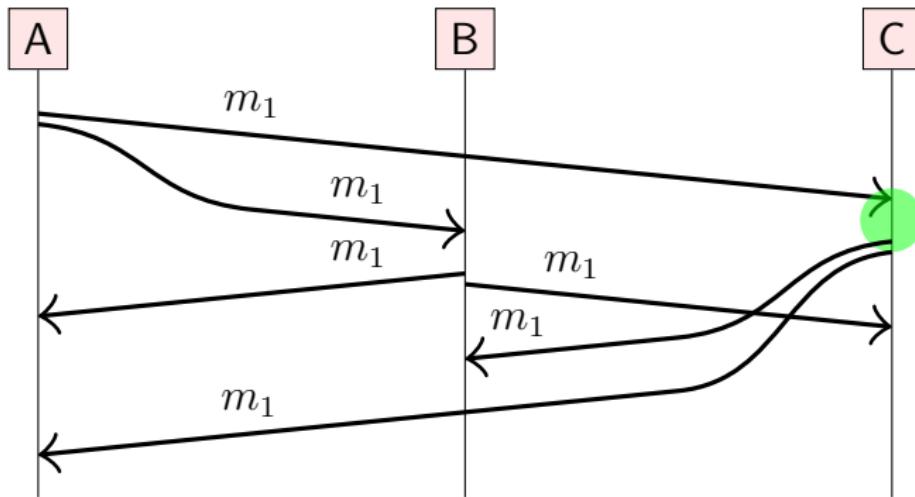
# Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



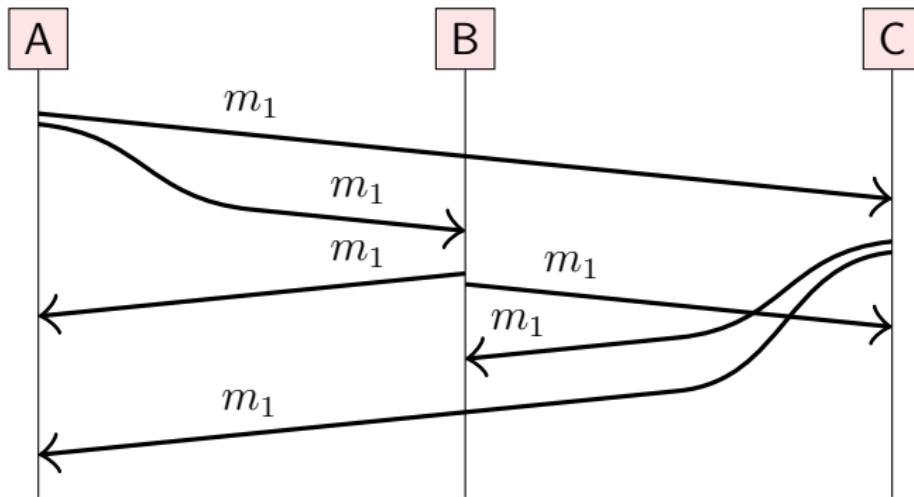
# Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).



# Eager reliable broadcast

Idea: the **first time** a node receives a particular message, it **re-broadcasts** to each other node (via reliable links).

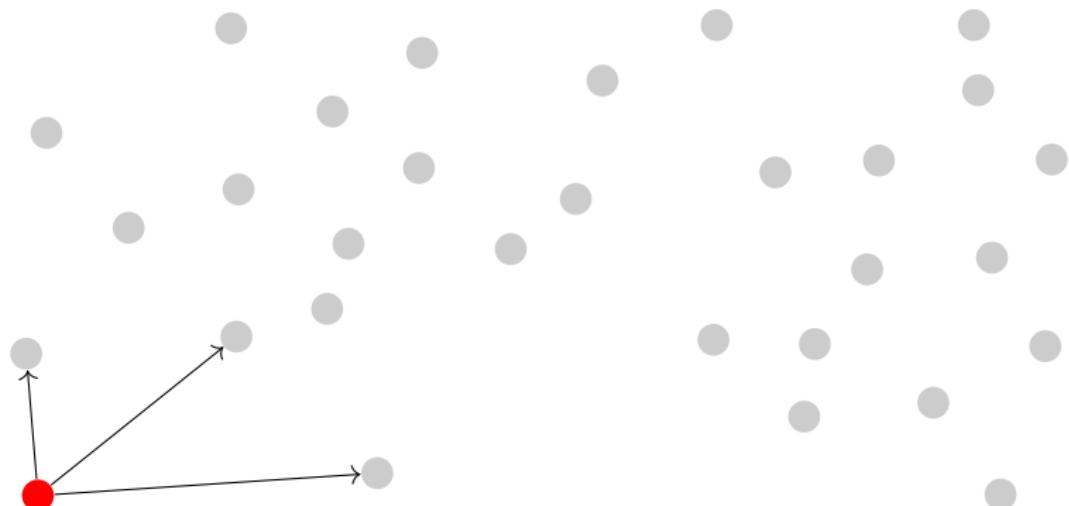


Reliable, but... up to  $O(n^2)$  messages for  $n$  nodes!

# Gossip protocols

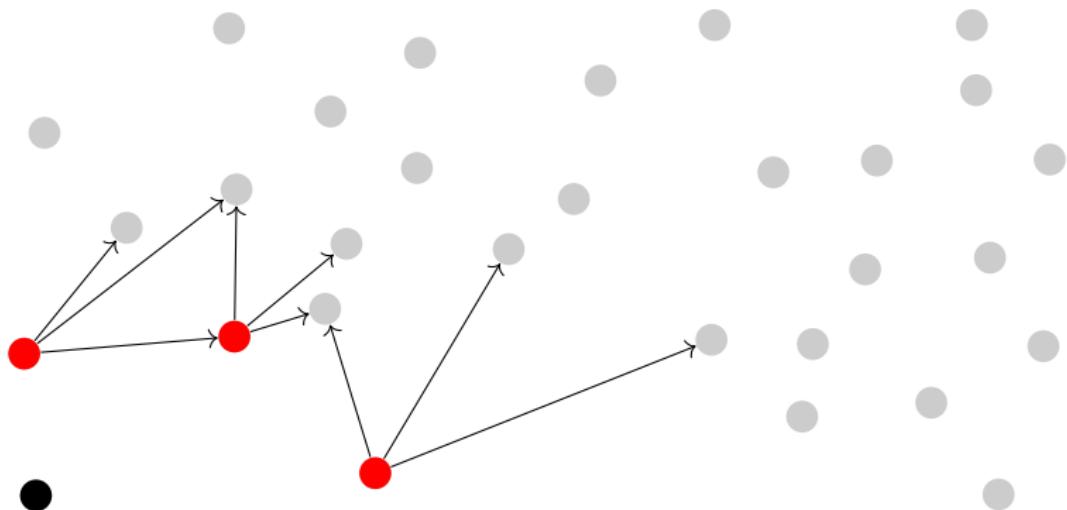
Useful when broadcasting to a large number of nodes.

Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

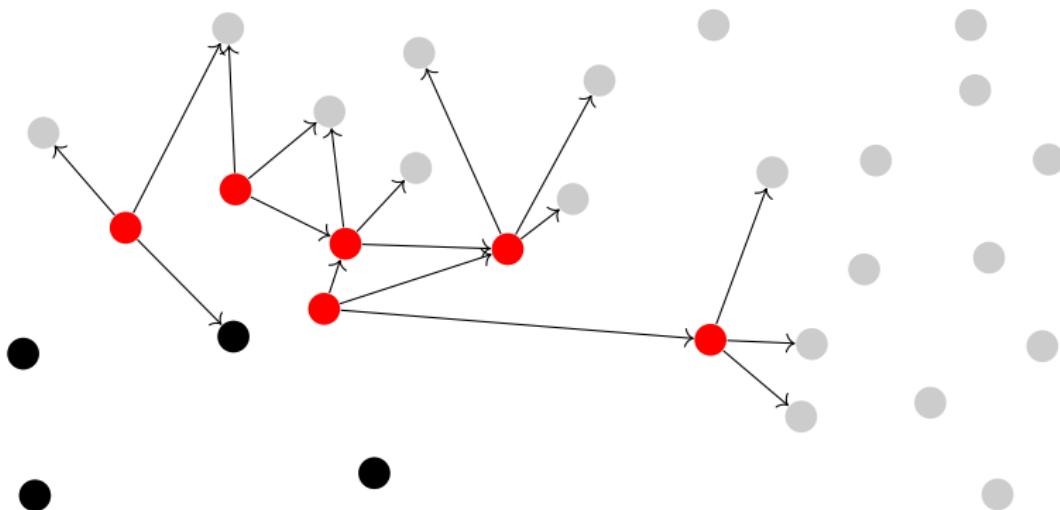
Useful when broadcasting to a large number of nodes.  
Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

Useful when broadcasting to a large number of nodes.

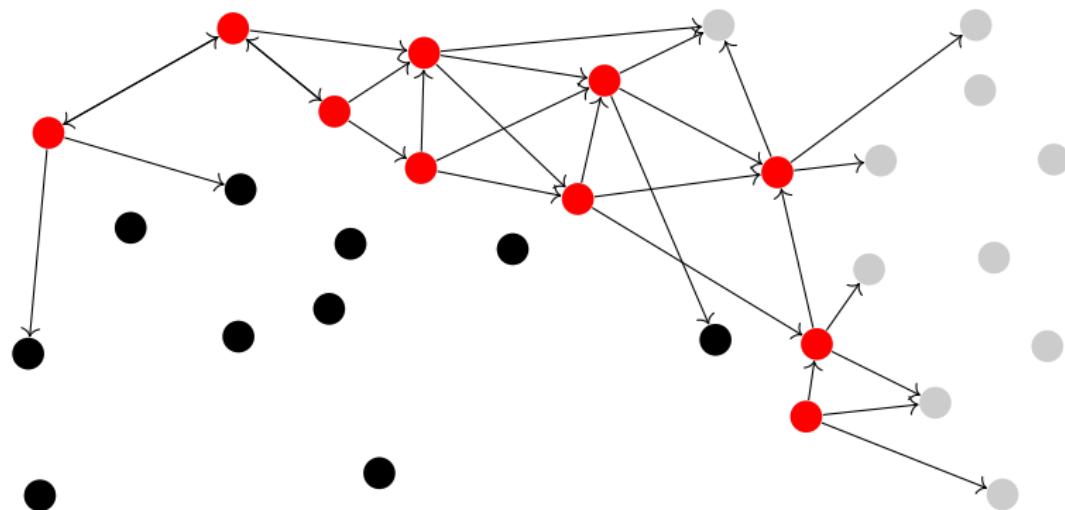
Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

Useful when broadcasting to a large number of nodes.

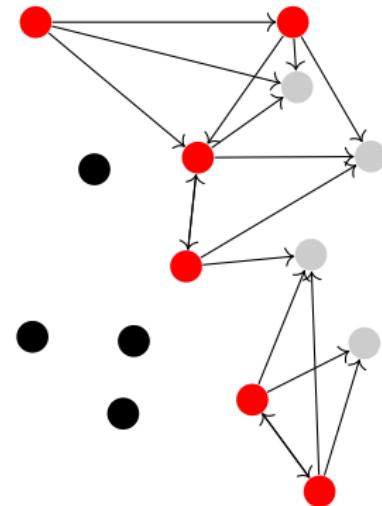
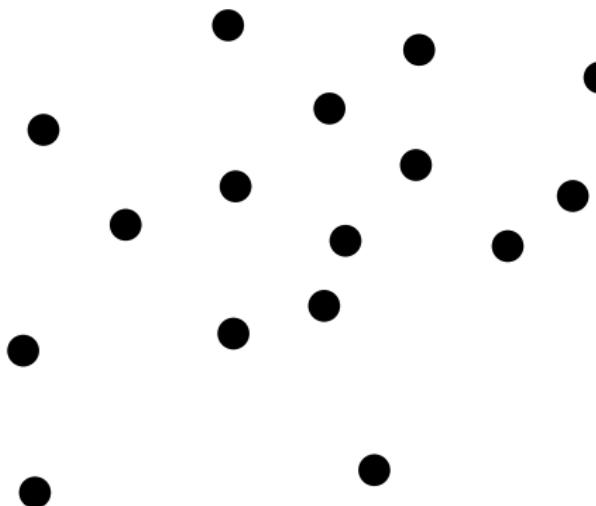
Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

Useful when broadcasting to a large number of nodes.

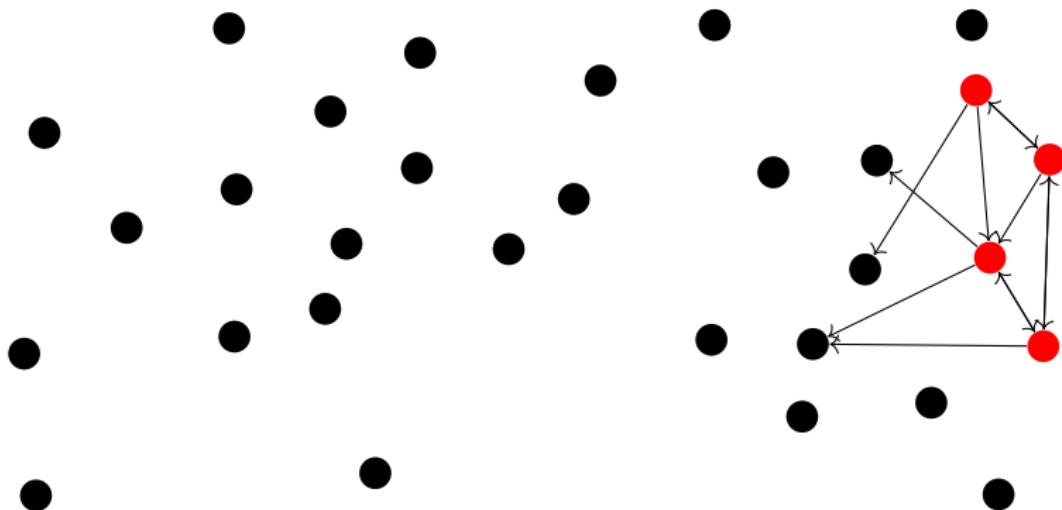
Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

Useful when broadcasting to a large number of nodes.

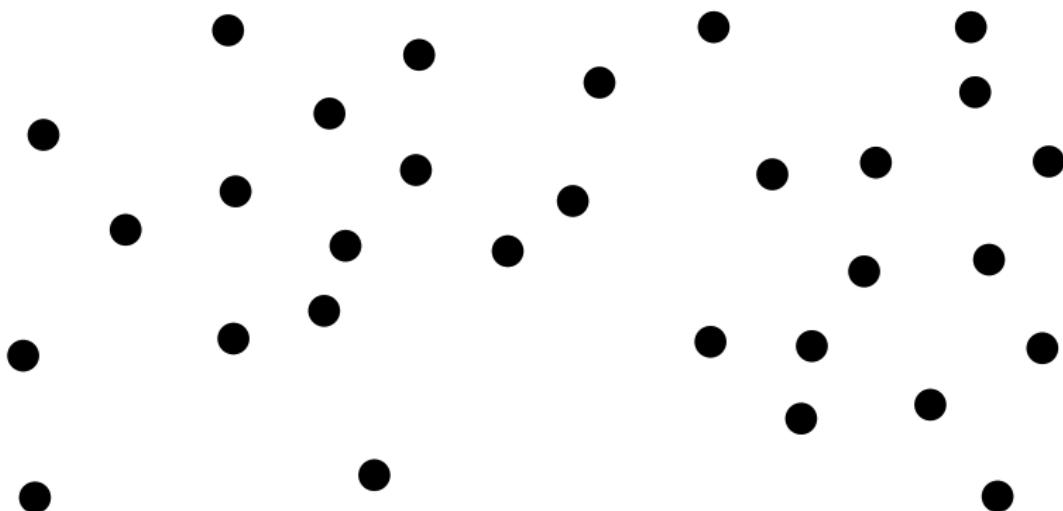
Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



# Gossip protocols

Useful when broadcasting to a large number of nodes.

Idea: when a node receives a message for the first time,  
**forward it to 3 other nodes**, chosen randomly.



Eventually reaches all nodes (with high probability).

# FIFO broadcast algorithm

**on** initialisation **do**

$sendSeq := 0$ ;  $delivered := \langle 0, 0, \dots, 0 \rangle$ ;  $buffer := \{\}$

**end on**

**on** request to broadcast  $m$  at node  $N_i$  **do**

send  $(i, sendSeq, m)$  via reliable broadcast

$sendSeq := sendSeq + 1$

**end on**

**on** receiving  $msg$  from reliable broadcast at node  $N_i$  **do**

$buffer := buffer \cup \{msg\}$

**while**  $\exists sender, m.$   $(sender, delivered[sender], m) \in buffer$  **do**

deliver  $m$  to the application

$delivered[sender] := delivered[sender] + 1$

**end while**

**end on**

# Causal broadcast algorithm

**on** initialisation **do**

$sendSeq := 0$ ;  $delivered := \langle 0, 0, \dots, 0 \rangle$ ;  $buffer := \{\}$

**end on**

**on** request to broadcast  $m$  at node  $N_i$  **do**

$deps := delivered$ ;  $deps[i] := sendSeq$

    send  $(i, deps, m)$  via reliable broadcast

$sendSeq := sendSeq + 1$

**end on**

**on** receiving  $msg$  from reliable broadcast at node  $N_i$  **do**

$buffer := buffer \cup \{msg\}$

**while**  $\exists (sender, deps, m) \in buffer. deps \leq delivered$  **do**

        deliver  $m$  to the application

$buffer := buffer \setminus \{(sender, deps, m)\}$

$delivered[sender] := delivered[sender] + 1$

**end while**

**end on**

# Vector clocks ordering

Define the following order on vector timestamps  
(in a system with  $n$  nodes):

- ▶  $T = T'$  iff  $T[i] = T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T \leq T'$  iff  $T[i] \leq T'[i]$  for all  $i \in \{1, \dots, n\}$
- ▶  $T < T'$  iff  $T \leq T'$  and  $T \neq T'$
- ▶  $T \parallel T'$  iff  $T \not\leq T'$  and  $T' \not\leq T$

## **Single leader** approach:

- ▶ One node is designated as leader (sequencer)
- ▶ To broadcast message, send it to the leader; leader broadcasts it via FIFO broadcast.

## **Single leader** approach:

- ▶ One node is designated as leader (sequencer)
- ▶ To broadcast message, send it to the leader; leader broadcasts it via FIFO broadcast.
- ▶ Problem: leader crashes  $\implies$  no more messages delivered
- ▶ Changing the leader safely is difficult

## **Single leader** approach:

- ▶ One node is designated as leader (sequencer)
- ▶ To broadcast message, send it to the leader; leader broadcasts it via FIFO broadcast.
- ▶ Problem: leader crashes  $\implies$  no more messages delivered
- ▶ Changing the leader safely is difficult

## **Lamport clocks** approach:

- ▶ Attach Lamport timestamp to every message
- ▶ Deliver messages in total order of timestamps

## **Single leader** approach:

- ▶ One node is designated as leader (sequencer)
- ▶ To broadcast message, send it to the leader; leader broadcasts it via FIFO broadcast.
- ▶ Problem: leader crashes  $\implies$  no more messages delivered
- ▶ Changing the leader safely is difficult

## **Lamport clocks** approach:

- ▶ Attach Lamport timestamp to every message
- ▶ Deliver messages in total order of timestamps
- ▶ Problem: how do you know if you have seen all messages with timestamp  $< T$ ? Need to use FIFO links and wait for message with timestamp  $\geq T$  from every node

# Replication

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

- ▶ Keeping a copy of the same data on multiple nodes
- ▶ Databases, filesystems, caches, . . .
- ▶ A node that has a copy of the data is called a **replica**

- ▶ Keeping a copy of the same data on multiple nodes
- ▶ Databases, filesystems, caches, . . .
- ▶ A node that has a copy of the data is called a **replica**
- ▶ If some replicas are faulty, others are still accessible
- ▶ Spread load across many replicas

- ▶ Keeping a copy of the same data on multiple nodes
- ▶ Databases, filesystems, caches, . . .
- ▶ A node that has a copy of the data is called a **replica**
- ▶ If some replicas are faulty, others are still accessible
- ▶ Spread load across many replicas
- ▶ Easy if the data doesn't change: just copy it
- ▶ We will focus on data changes

- ▶ Keeping a copy of the same data on multiple nodes
- ▶ Databases, filesystems, caches, . . .
- ▶ A node that has a copy of the data is called a **replica**
- ▶ If some replicas are faulty, others are still accessible
- ▶ Spread load across many replicas
- ▶ Easy if the data doesn't change: just copy it
- ▶ We will focus on data changes

Compare to **RAID** (Redundant Array of Independent Disks):  
replication within a single computer

- ▶ RAID has single controller; in distributed system, each node acts independently
- ▶ Replicas can be distributed around the world, near users

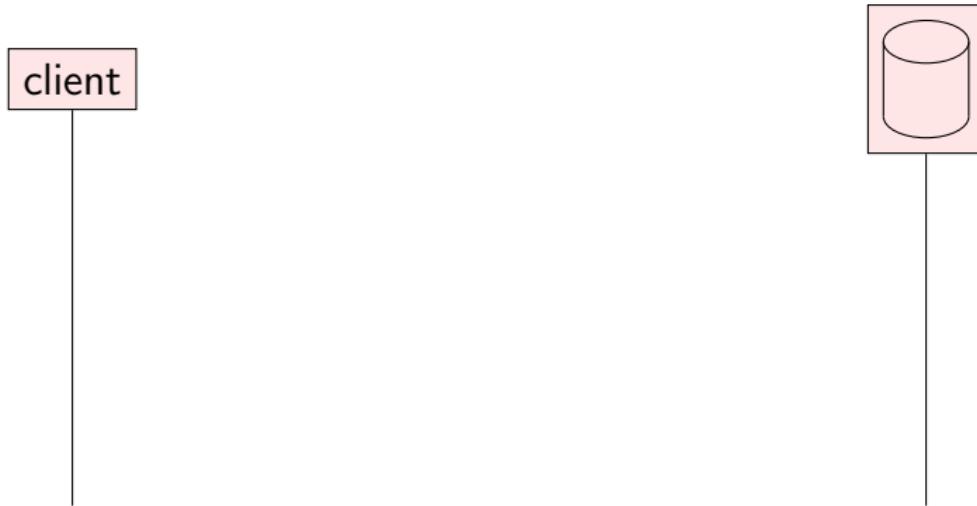
# Retrying state updates



**User A:** The moon is not actually made of cheese!

Like

12,300 people like this.



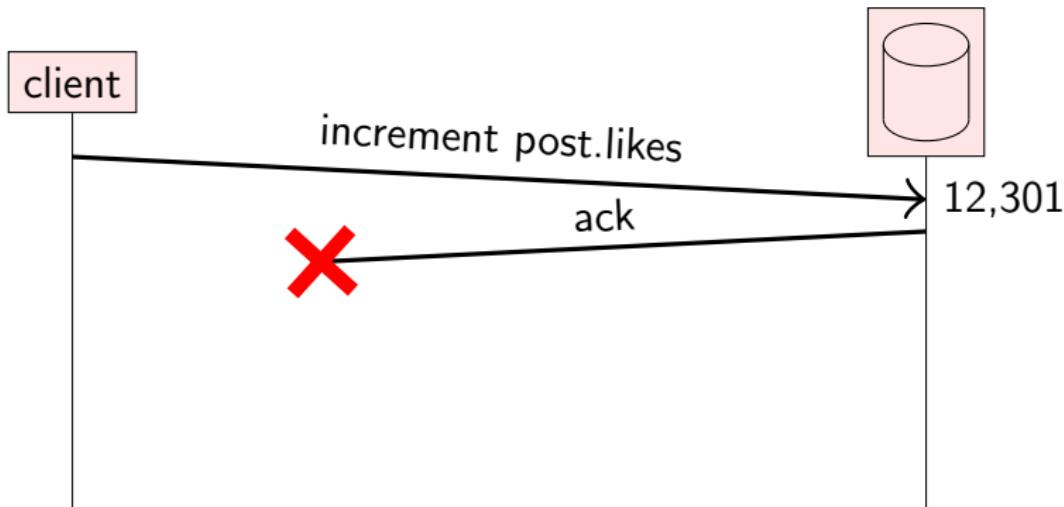
# Retrying state updates



**User A:** The moon is not actually made of cheese!

Like

12,300 people like this.



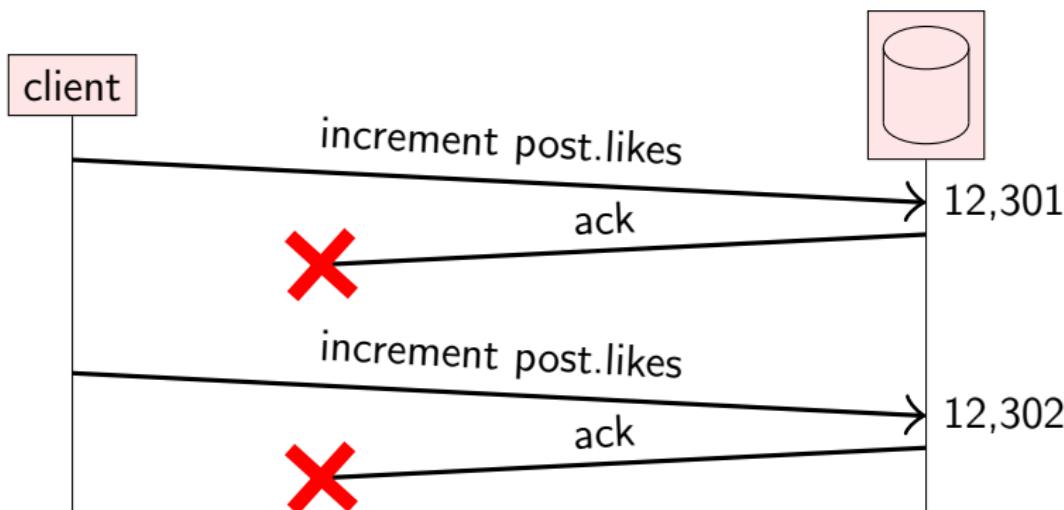
# Retrying state updates



**User A:** The moon is not actually made of cheese!

Like

12,300 people like this.



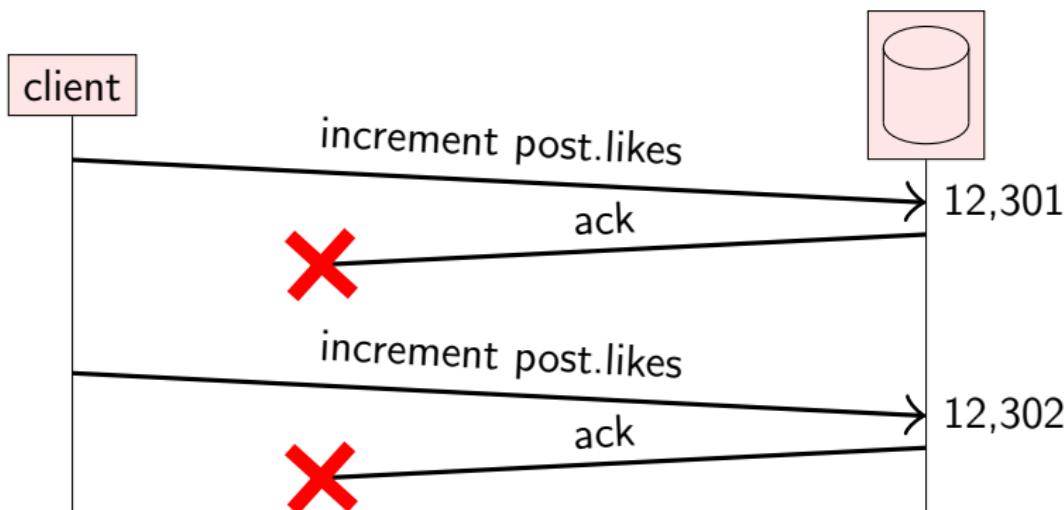
# Retrying state updates



**User A:** The moon is not actually made of cheese!

Like

12,300 people like this.



Deduplicating requests requires that the database tracks which requests it has already seen (in stable storage)



TWEETS

6,219

FOLLOWING

-20

FOLLOWERS

24.1K



Follow

# Лепра

@leprasonium

 Добро пожаловать отсюда

Default City

**Лепра** @leprasonium · 2h

Викторианские советы

Часть 2 [pic.twitter.com/21PraRYBaO](https://pic.twitter.com/21PraRYBaO)[Details](#)**Лепра** @leprasonium · 2h

Викторианские советы

Часть 1 [pic.twitter.com/BVE6ao8711](https://pic.twitter.com/BVE6ao8711)[Details](#)[Go to full profile](#)

A function  $f$  is idempotent if  $f(x) = f(f(x))$ .

- ▶ **Not idempotent:**  $f(\text{likeCount}) = \text{likeCount} + 1$
- ▶ **Idempotent:**  $f(\text{likeSet}) = \text{likeSet} \cup \{\text{userID}\}$

Idempotent requests can be retried without deduplication.

A function  $f$  is idempotent if  $f(x) = f(f(x))$ .

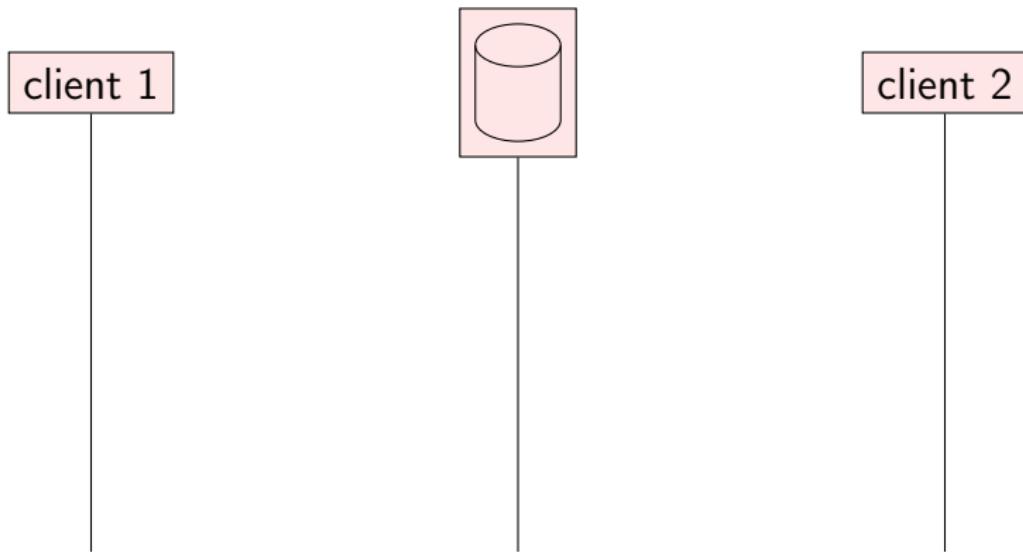
- ▶ **Not idempotent:**  $f(\text{likeCount}) = \text{likeCount} + 1$
- ▶ **Idempotent:**  $f(\text{likeSet}) = \text{likeSet} \cup \{\text{userID}\}$

Idempotent requests can be retried without deduplication.

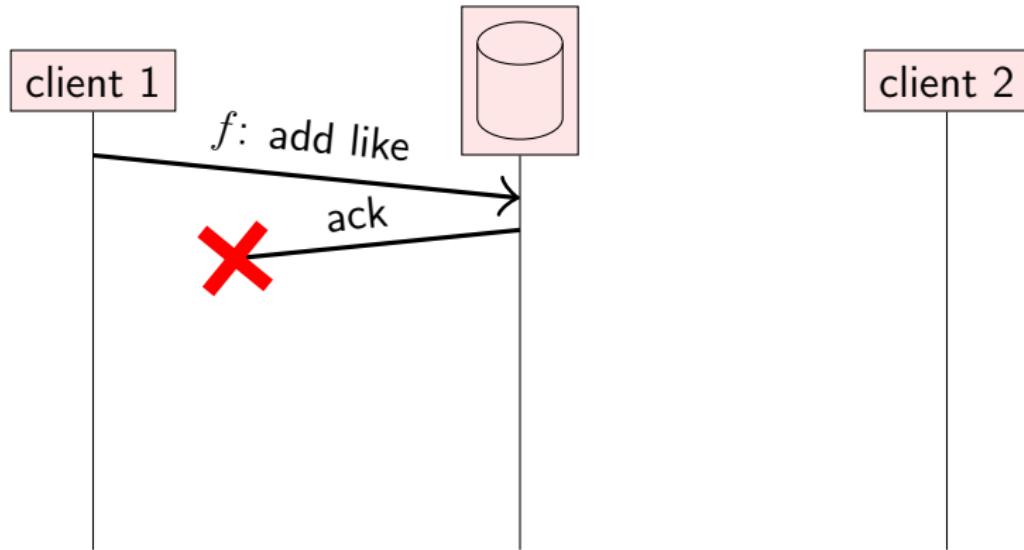
Choice of retry behaviour:

- ▶ **At-most-once** semantics:  
send request, don't retry, update may not happen
- ▶ **At-least-once** semantics:  
retry request until acknowledged, may repeat update
- ▶ **Exactly-once** semantics:  
retry + idempotence or deduplication

# Adding and then removing again

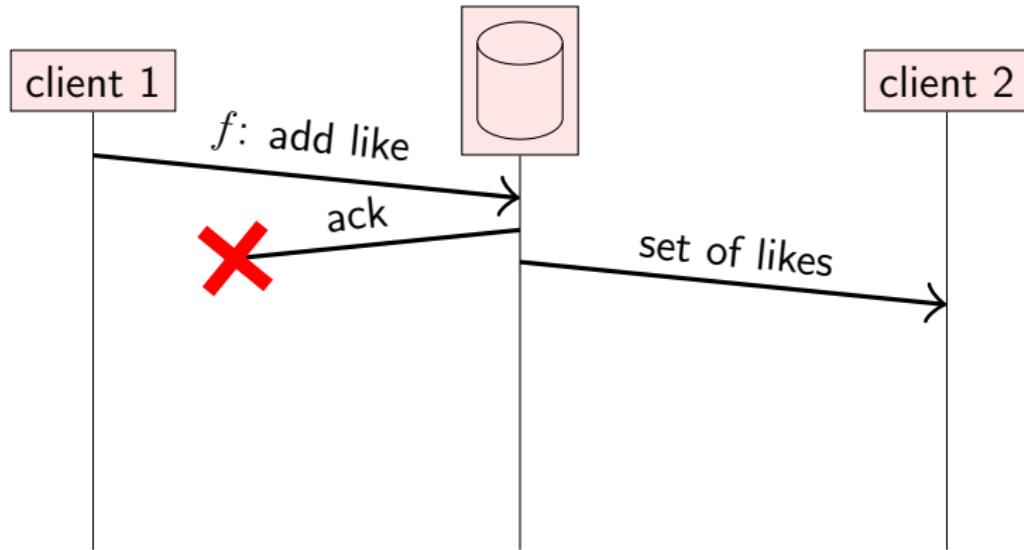


# Adding and then removing again



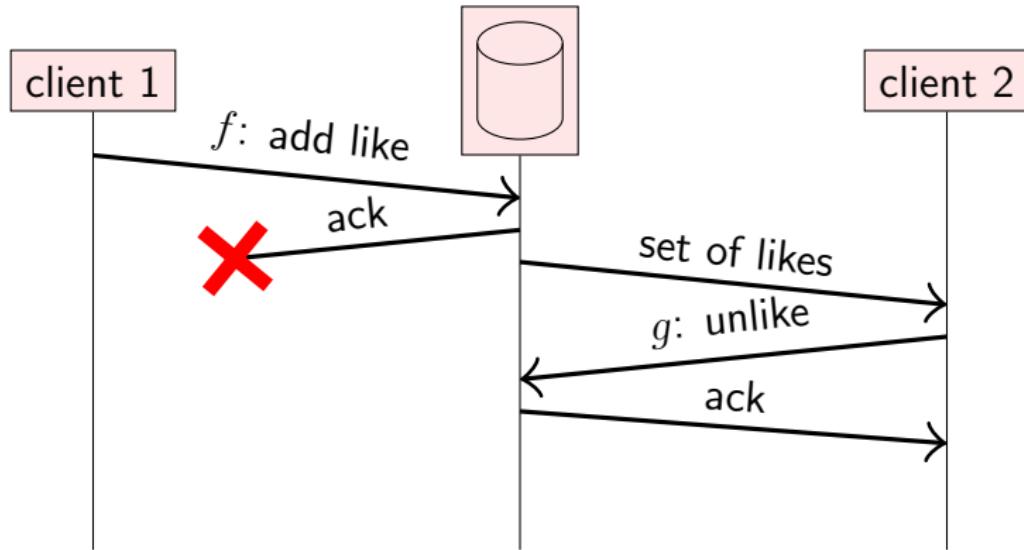
$$f(\text{likes}) = \text{likes} \cup \{\text{userID}\}$$

# Adding and then removing again



$$f(\text{likes}) = \text{likes} \cup \{\text{userID}\}$$

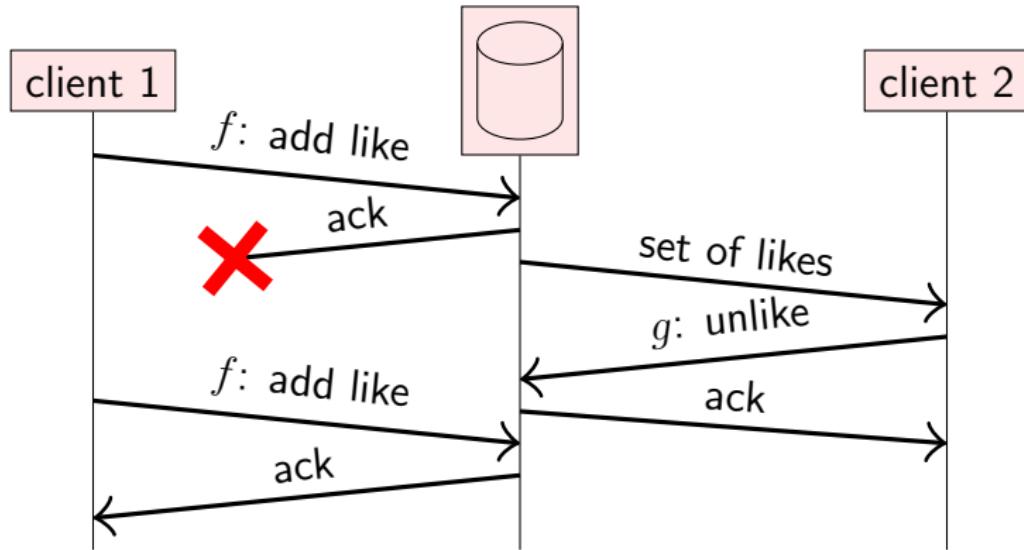
# Adding and then removing again



$$f(\text{likes}) = \text{likes} \cup \{\text{userID}\}$$

$$g(\text{likes}) = \text{likes} \setminus \{\text{userID}\}$$

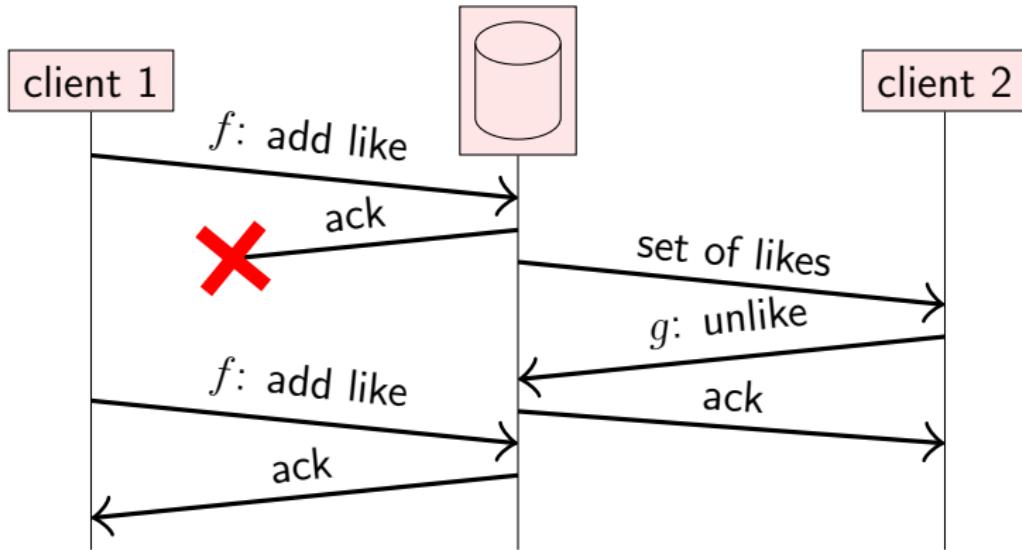
# Adding and then removing again



$$f(\text{likes}) = \text{likes} \cup \{\text{userID}\}$$

$$g(\text{likes}) = \text{likes} \setminus \{\text{userID}\}$$

# Adding and then removing again

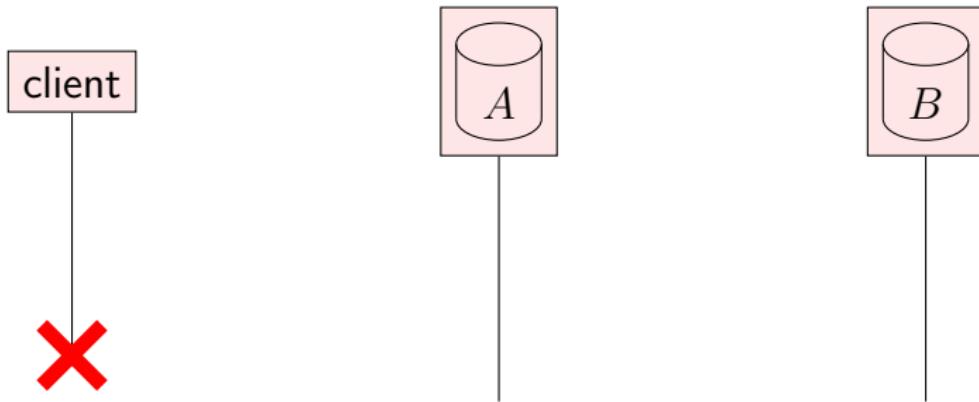


$$f(\text{likes}) = \text{likes} \cup \{\text{userID}\}$$

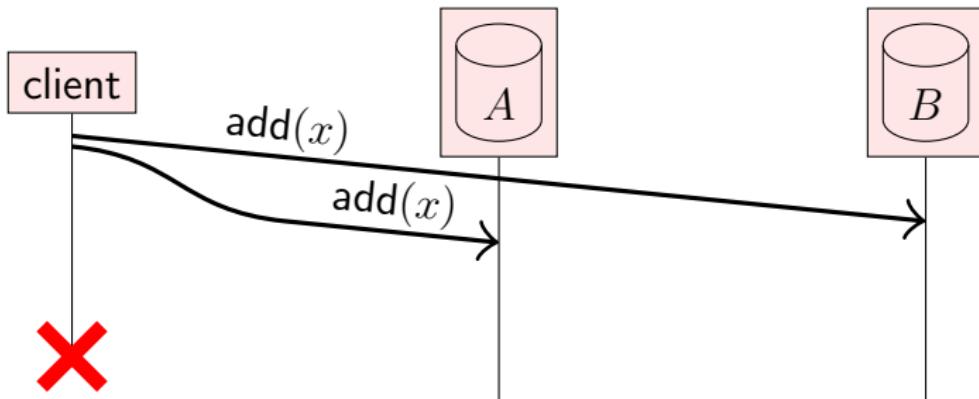
$$g(\text{likes}) = \text{likes} \setminus \{\text{userID}\}$$

**Idempotent?**  $f(f(x)) = f(x)$  but  $f(g(f(x))) \neq g(f(x))$

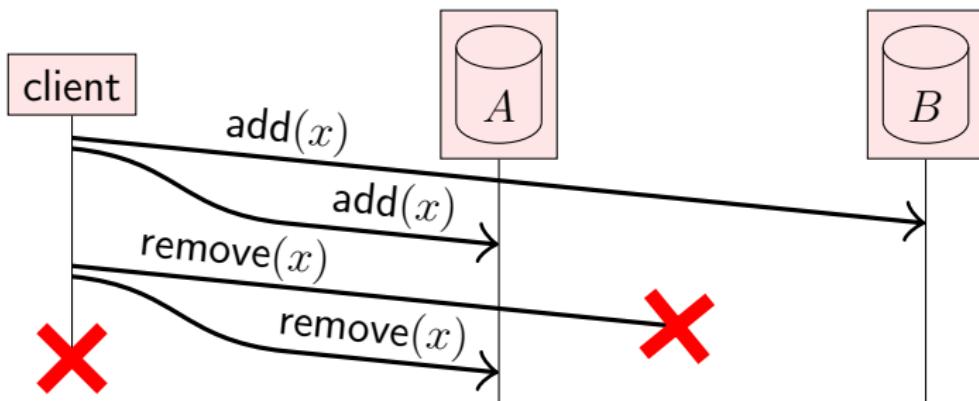
# Another problem with adding and removing



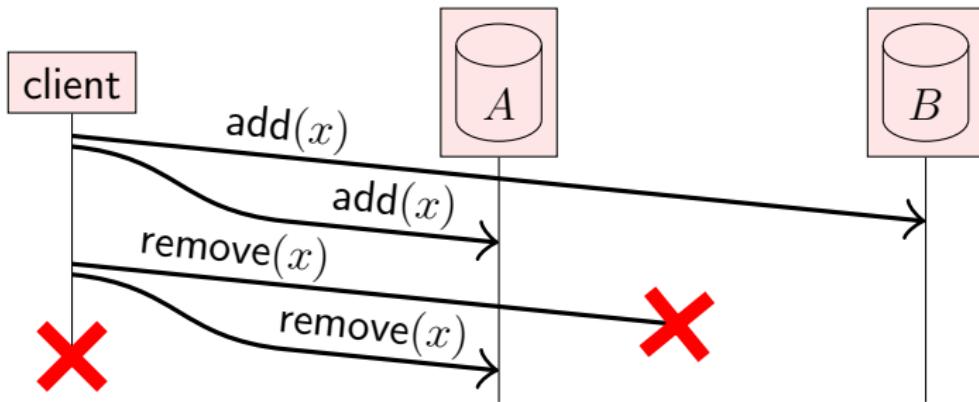
# Another problem with adding and removing



# Another problem with adding and removing

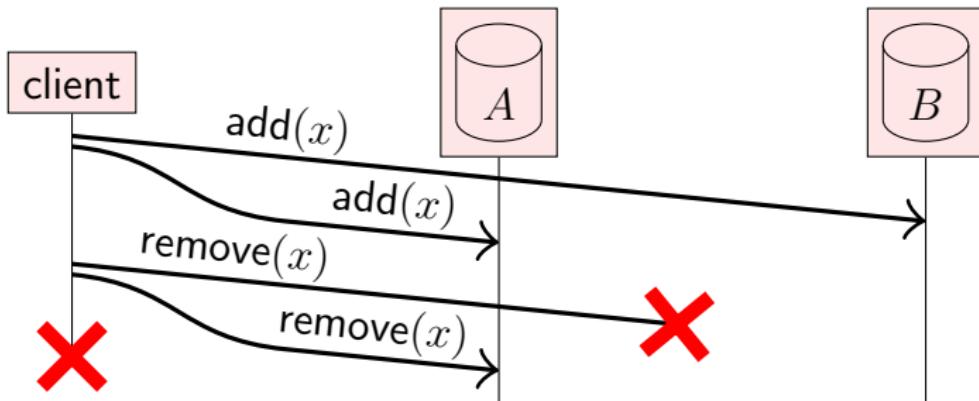


# Another problem with adding and removing

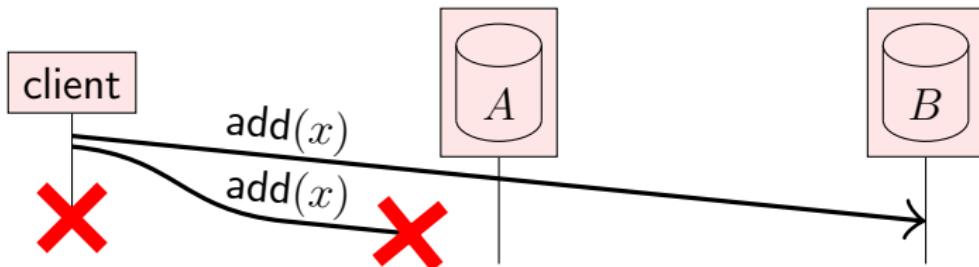


Final state ( $x \notin A, x \in B$ ) is the same as in this case:

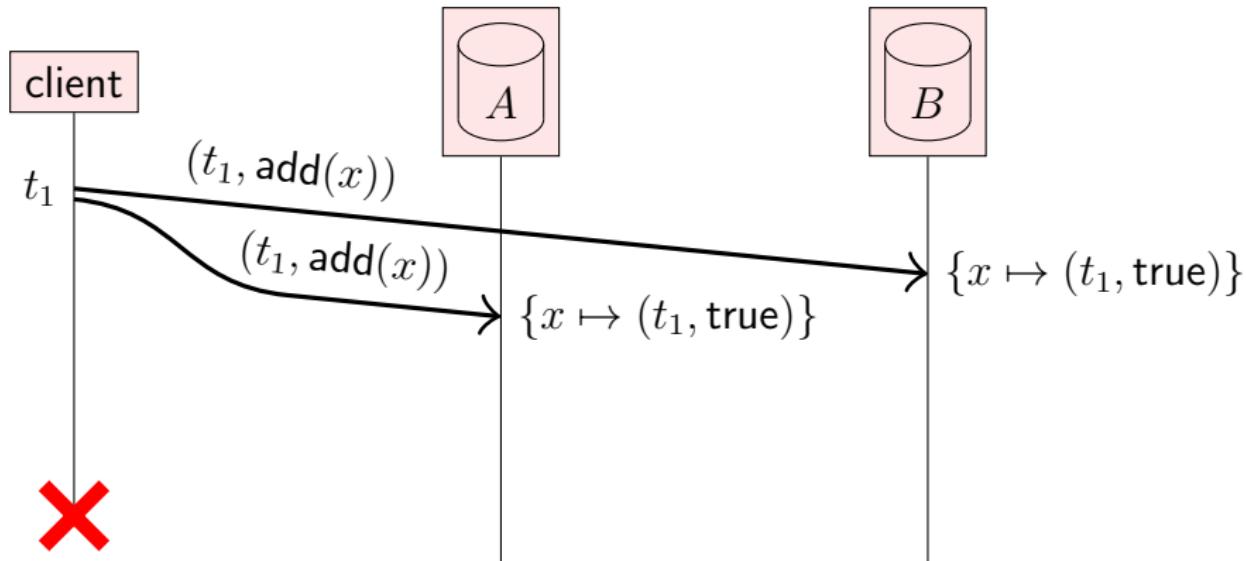
# Another problem with adding and removing



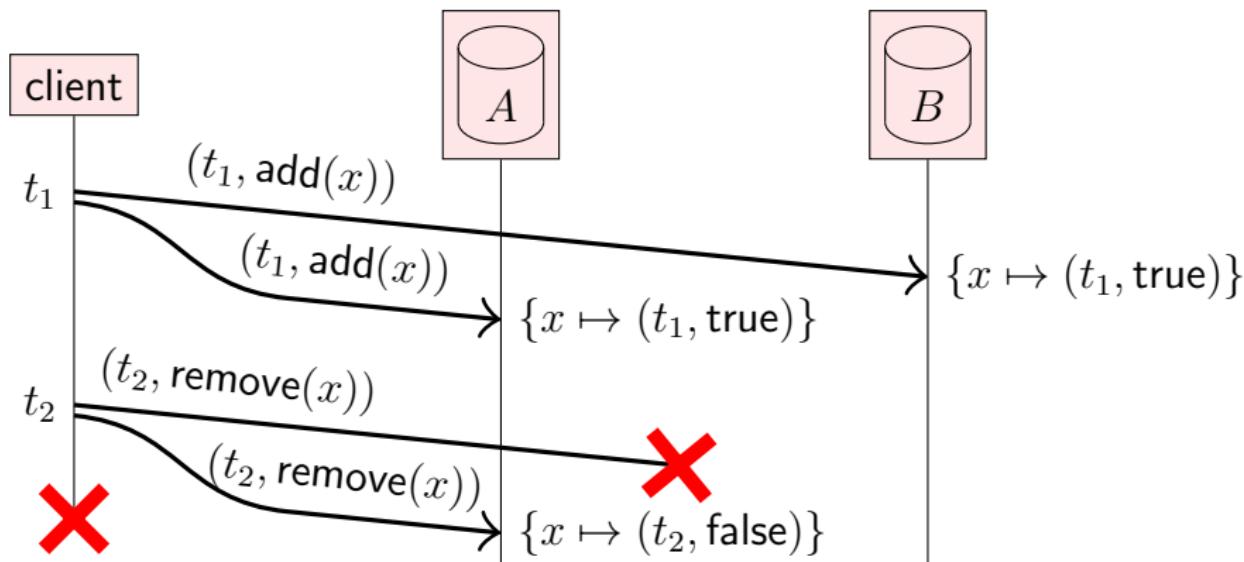
Final state ( $x \notin A, x \in B$ ) is the same as in this case:



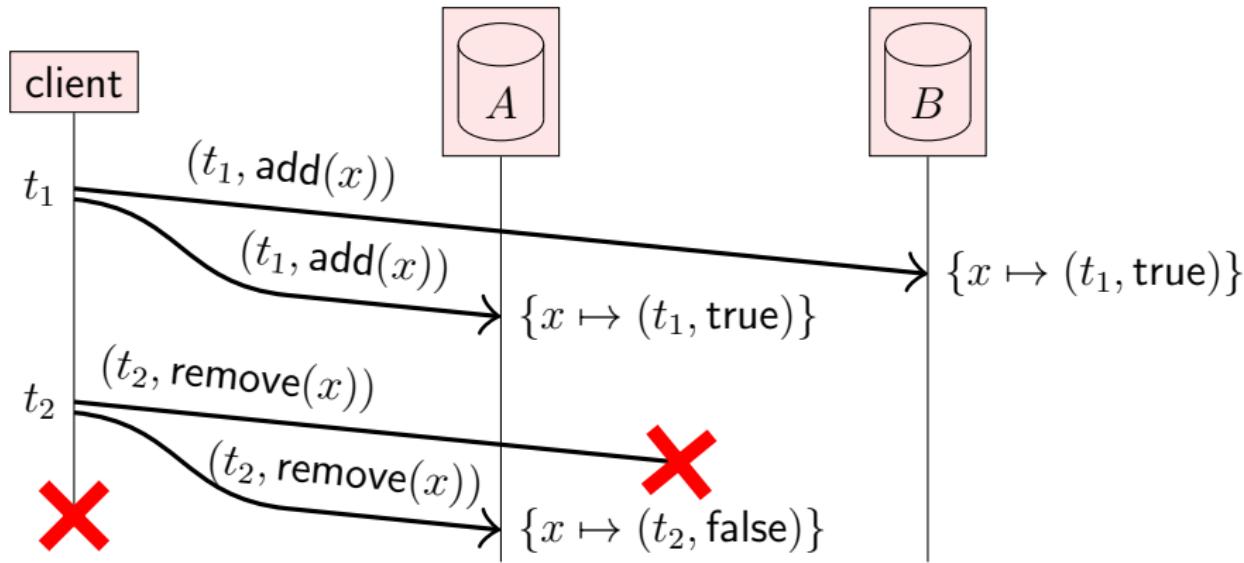
# Timestamps and tombstones



# Timestamps and tombstones

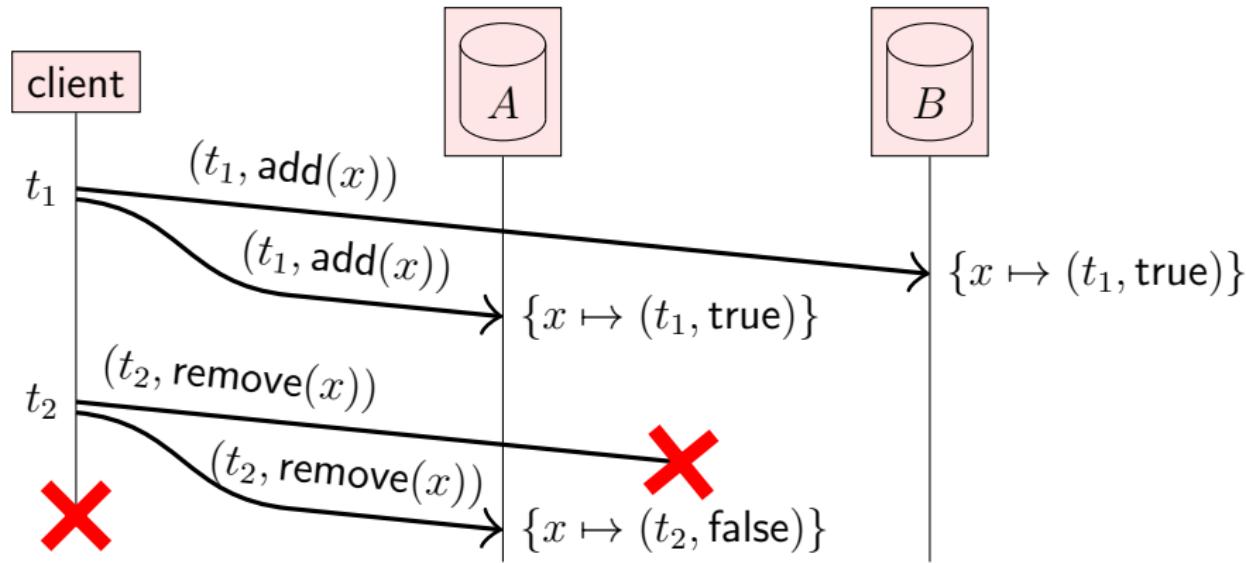


# Timestamps and tombstones



“remove( $x$ )” doesn't actually remove  $x$ : it labels  $x$  with “false” to indicate it is invisible (a **tombstone**)

# Timestamps and tombstones

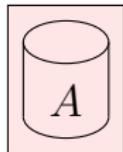


“remove( $x$ )” doesn't actually remove  $x$ : it labels  $x$  with “false” to indicate it is invisible (a **tombstone**)

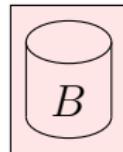
Every record has **logical timestamp** of last write

# Reconciling replicas

Replicas periodically communicate among themselves  
to check for any inconsistencies.



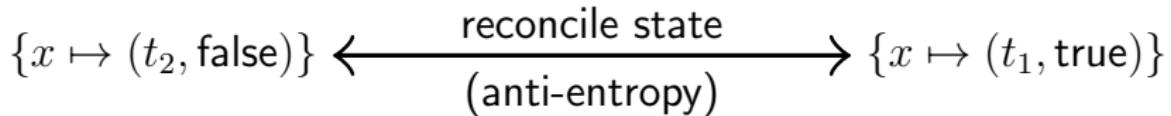
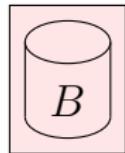
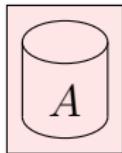
$\{x \mapsto (t_2, \text{false})\}$



$\{x \mapsto (t_1, \text{true})\}$

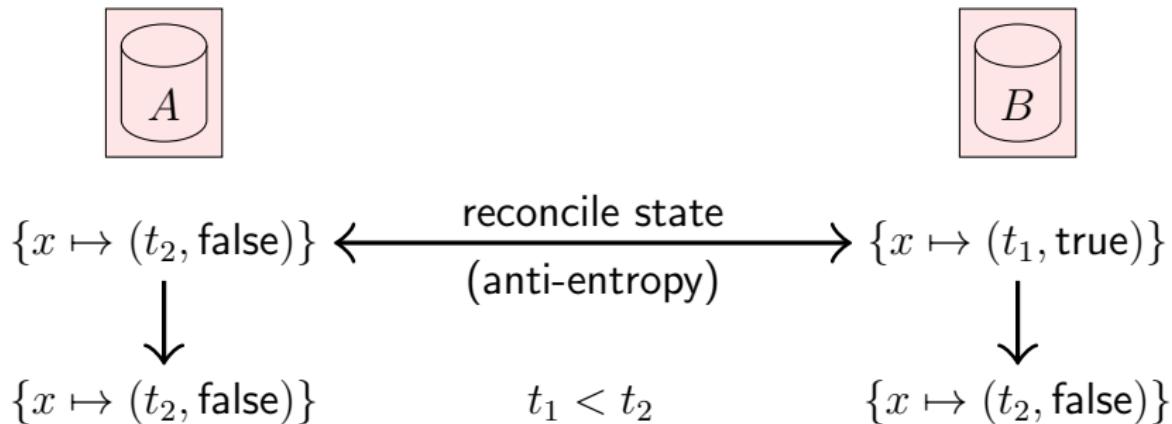
# Reconciling replicas

Replicas periodically communicate among themselves  
to check for any inconsistencies.



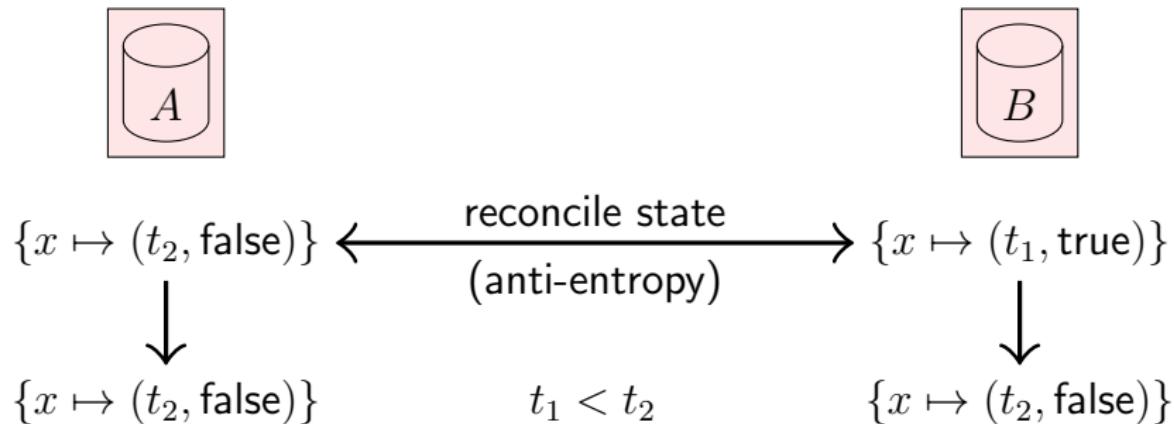
# Reconciling replicas

Replicas periodically communicate among themselves  
to check for any inconsistencies.



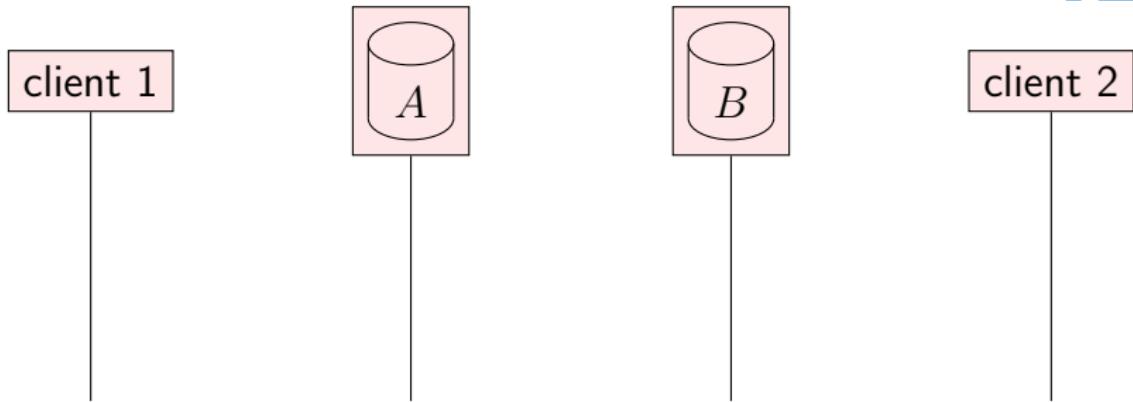
# Reconciling replicas

Replicas periodically communicate among themselves  
to check for any inconsistencies.

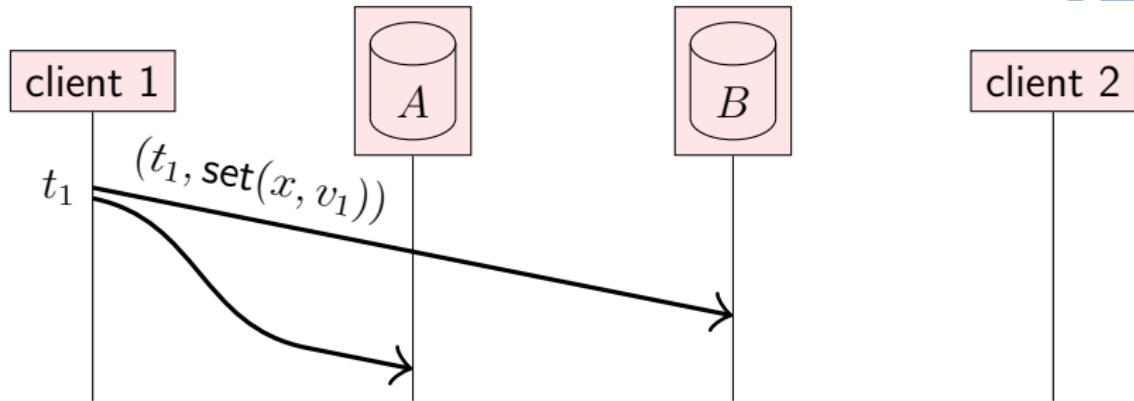


Propagate the record with the latest timestamp,  
discard the records with earlier timestamps  
(for a given key).

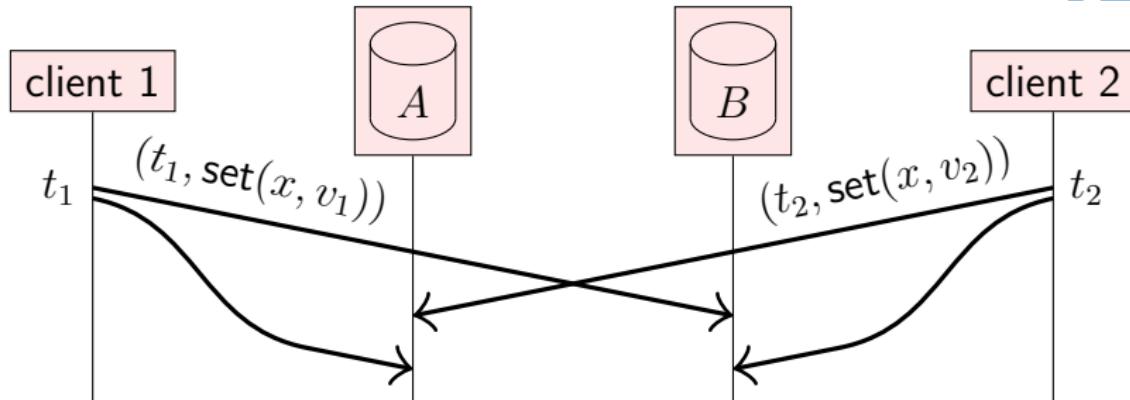
# Concurrent writes by different clients



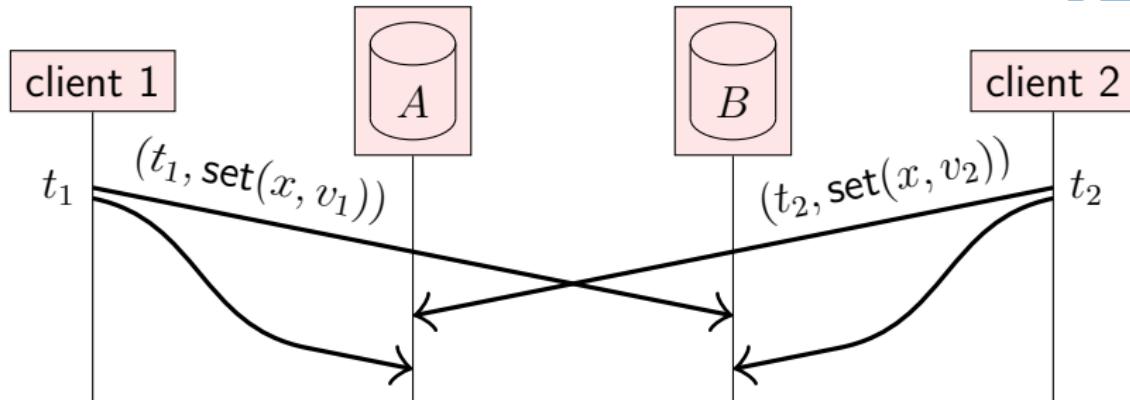
# Concurrent writes by different clients



# Concurrent writes by different clients



# Concurrent writes by different clients



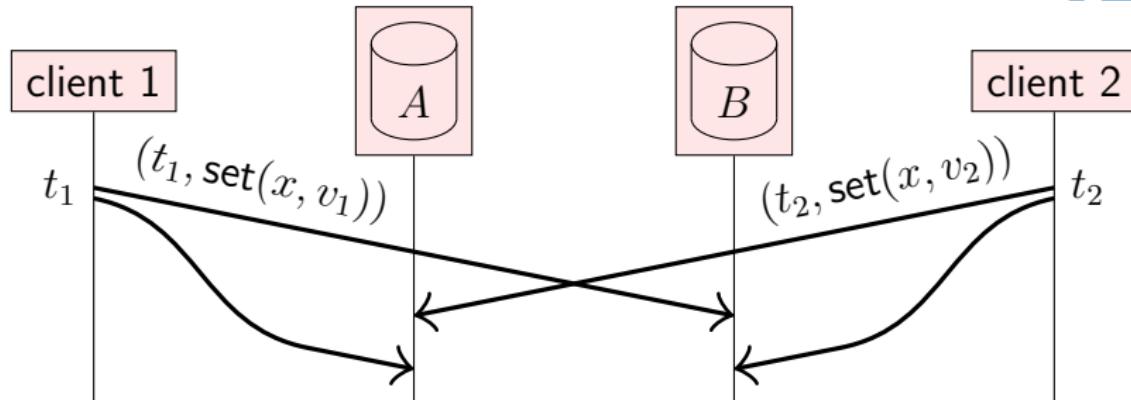
Two common approaches:

- ▶ **Last writer wins (LWW) register:**

Use timestamps with total order (e.g. Lamport clock)

Keep  $v_2$  and discard  $v_1$  if  $t_2 > t_1$ . Note: **data loss!**

# Concurrent writes by different clients



Two common approaches:

- ▶ **Last writer wins (LWW) register:**

Use timestamps with total order (e.g. Lamport clock)

Keep  $v_2$  and discard  $v_1$  if  $t_2 > t_1$ . Note: **data loss!**

- ▶ **Multi-value register:**

Use timestamps with partial order (e.g. vector clock)

$v_2$  replaces  $v_1$  if  $t_2 > t_1$ ; preserve both  $\{v_1, v_2\}$  if  $t_1 \parallel t_2$

# Probability of faults



A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

# Probability of faults



A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability  $p$  of being faulty or unavailable at any one time, and that faults are independent.  
(Not actually true! But okay approximation for now.)

# Probability of faults



A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability  $p$  of being faulty or unavailable at any one time, and that faults are independent.  
(Not actually true! But okay approximation for now.)

Probability of **all**  $n$  replicas being faulty:  $p^n$

Probability of  $\geq 1$  out of  $n$  replicas being faulty:  $1 - (1 - p)^n$

# Probability of faults

A replica may be **unavailable** due to network partition or node fault (e.g. crash, hardware problem).

Assume each replica has probability  $p$  of being faulty or unavailable at any one time, and that faults are independent.  
(Not actually true! But okay approximation for now.)

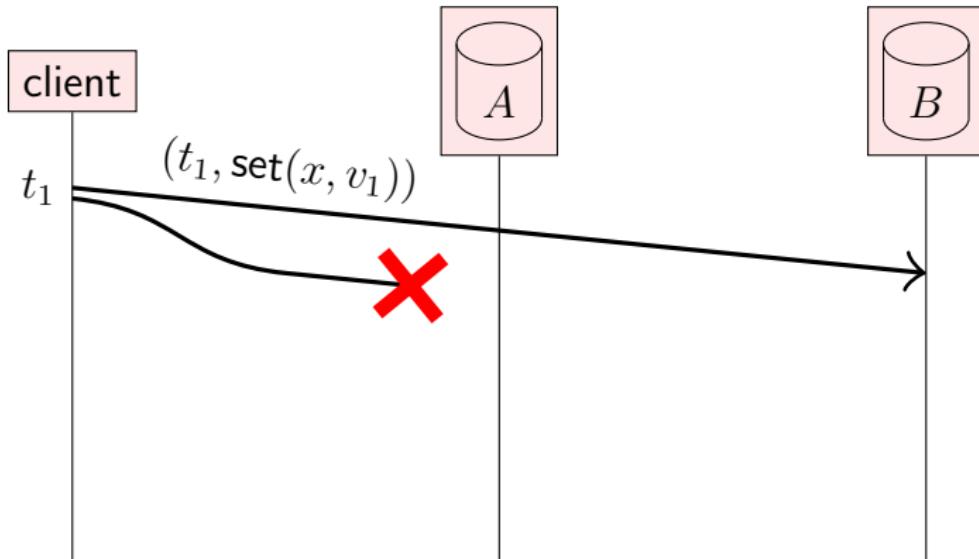
Probability of **all**  $n$  replicas being faulty:  $p^n$

Probability of  $\geq 1$  out of  $n$  replicas being faulty:  $1 - (1 - p)^n$

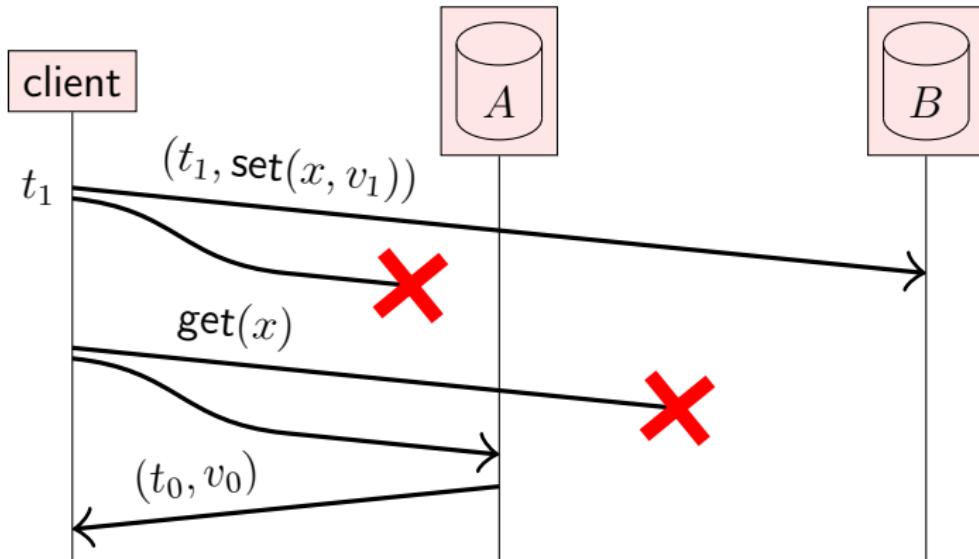
Example with  $p = 0.01$ :

replicas $n$	$P(\geq 1 \text{ faulty})$	$P(\geq \frac{n+1}{2} \text{ faulty})$	$P(\text{all } n \text{ faulty})$
1	0.01	0.01	0.01
3	0.03	$3 \cdot 10^{-4}$	$10^{-6}$
5	0.049	$1 \cdot 10^{-5}$	$10^{-10}$
100	0.63	$6 \cdot 10^{-74}$	$10^{-200}$

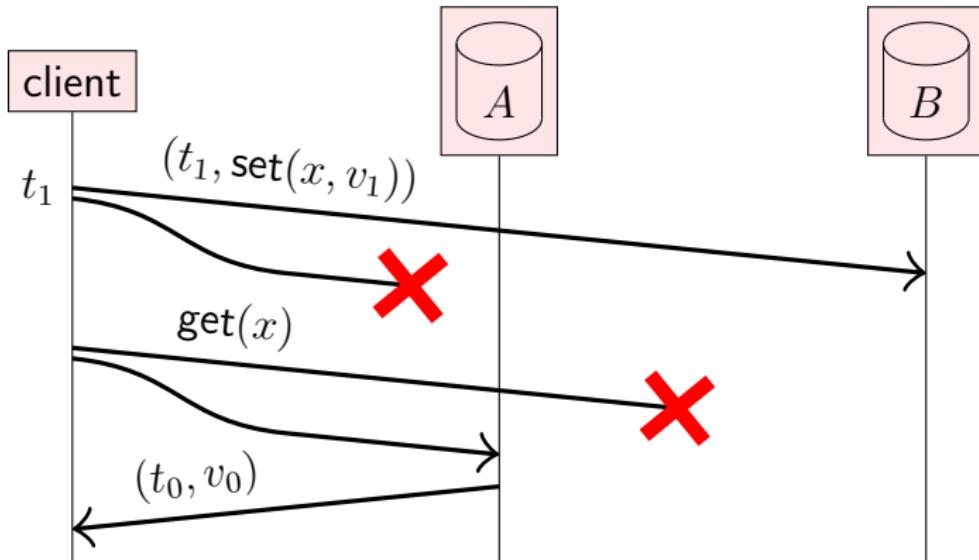
# Read-after-write consistency



# Read-after-write consistency

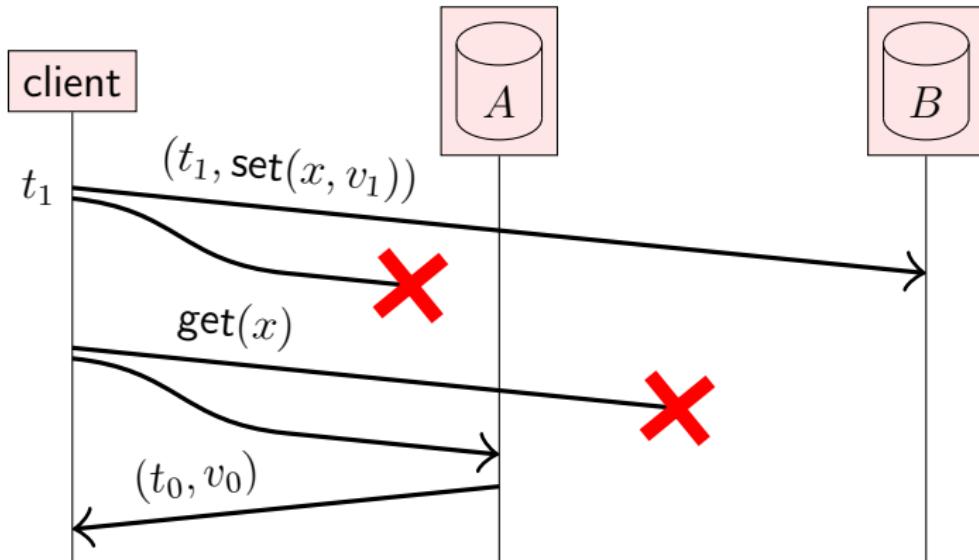


# Read-after-write consistency



Writing to one replica, reading from another: client does not read back the value it has written

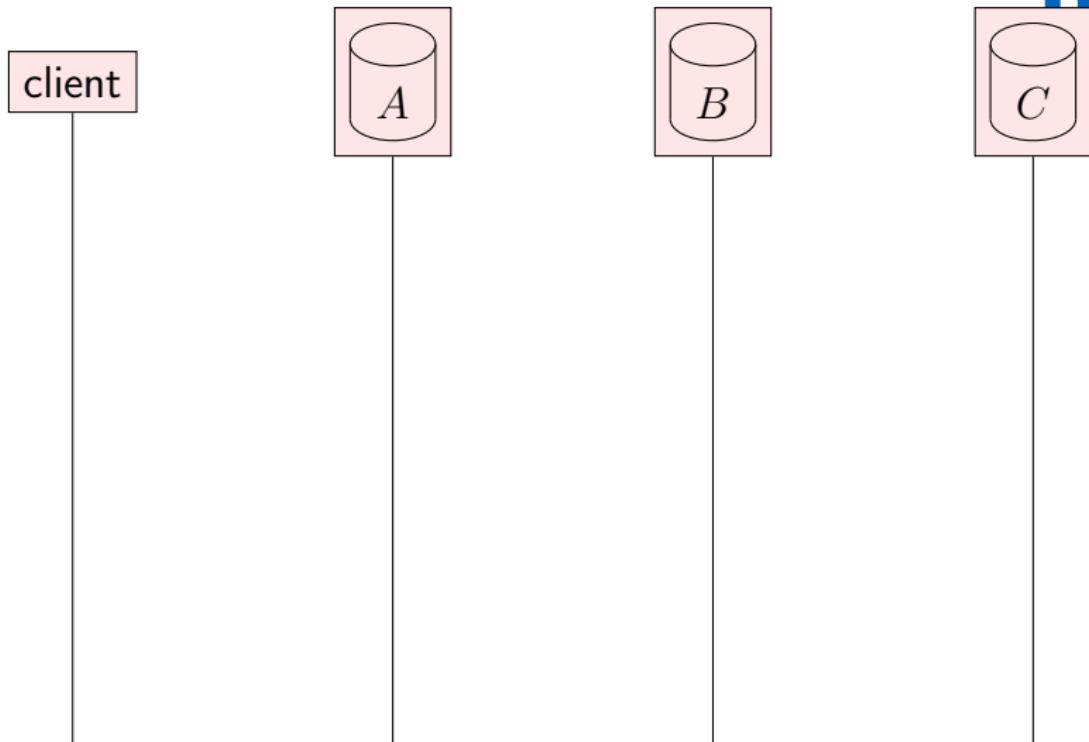
# Read-after-write consistency



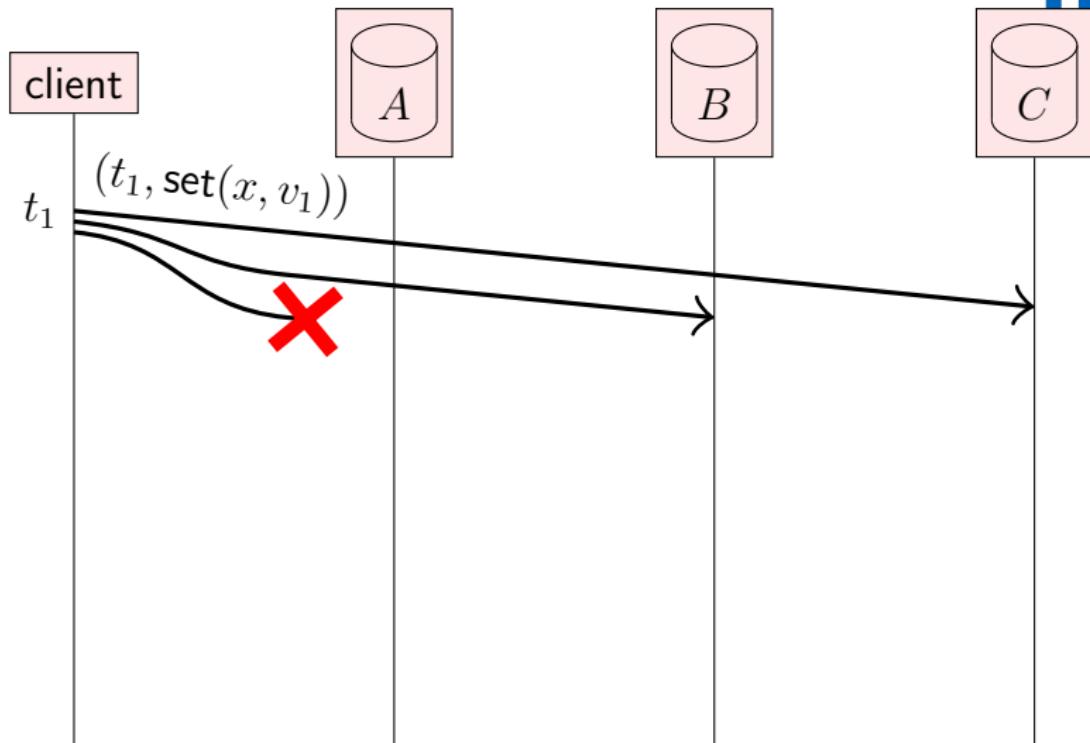
Writing to one replica, reading from another: client does not read back the value it has written

Require writing to/reading from both replicas  $\implies$  cannot write/read if one replica is unavailable

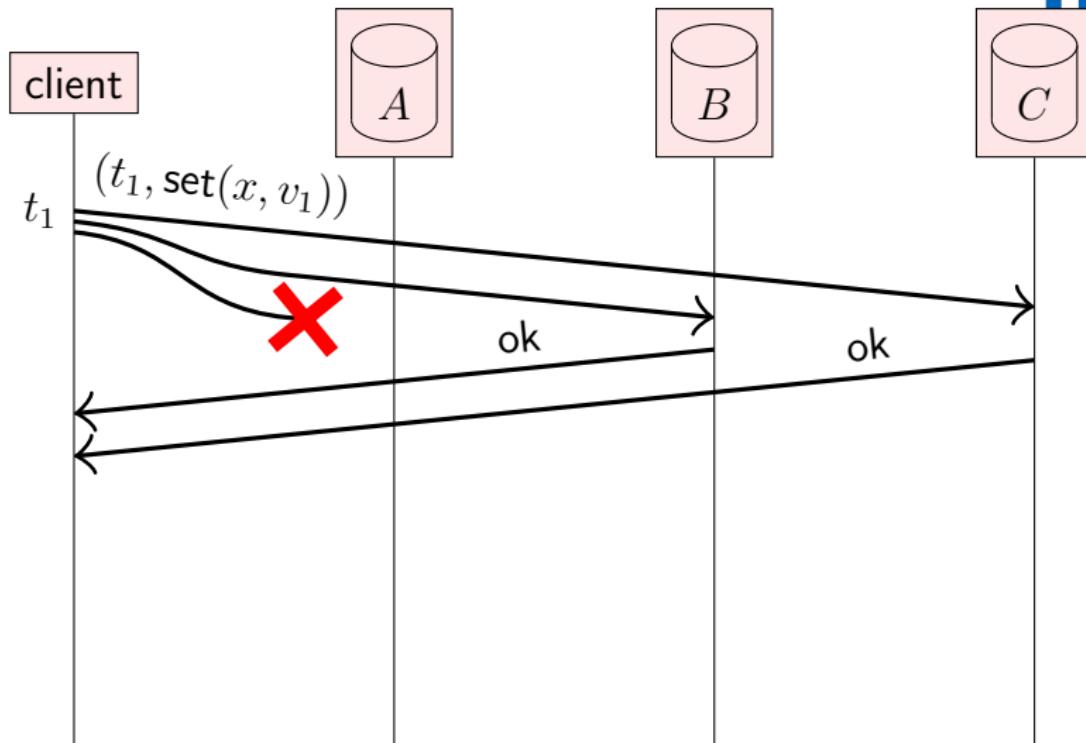
# Quorum (2 out of 3)



# Quorum (2 out of 3)

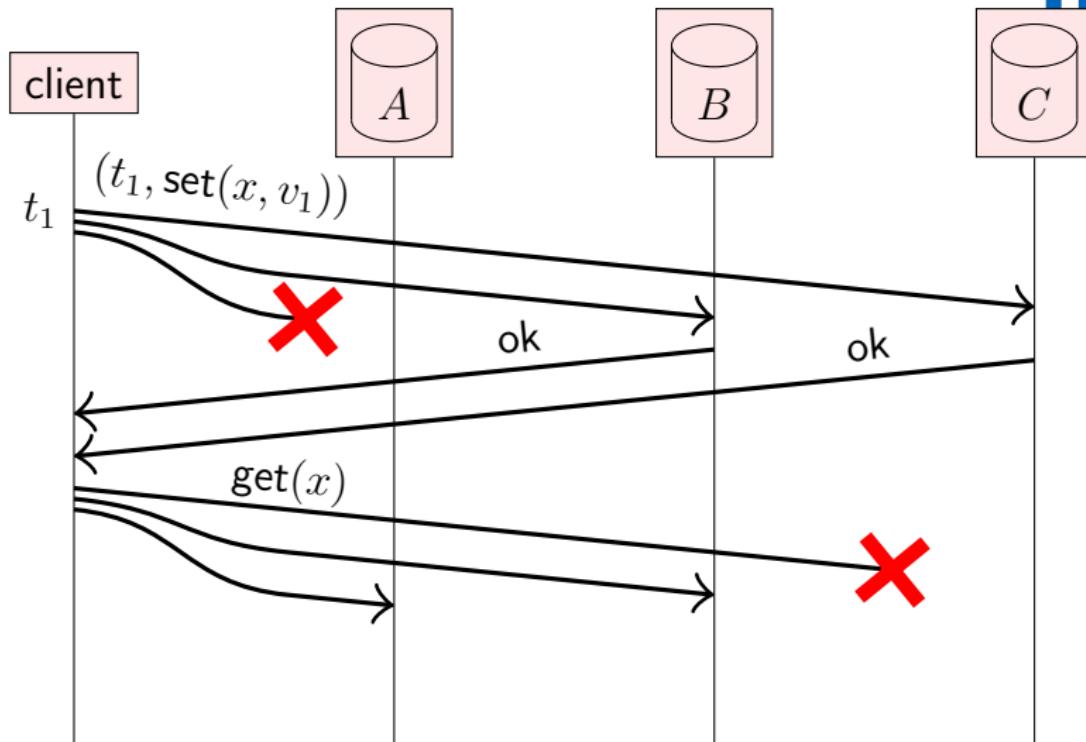


# Quorum (2 out of 3)



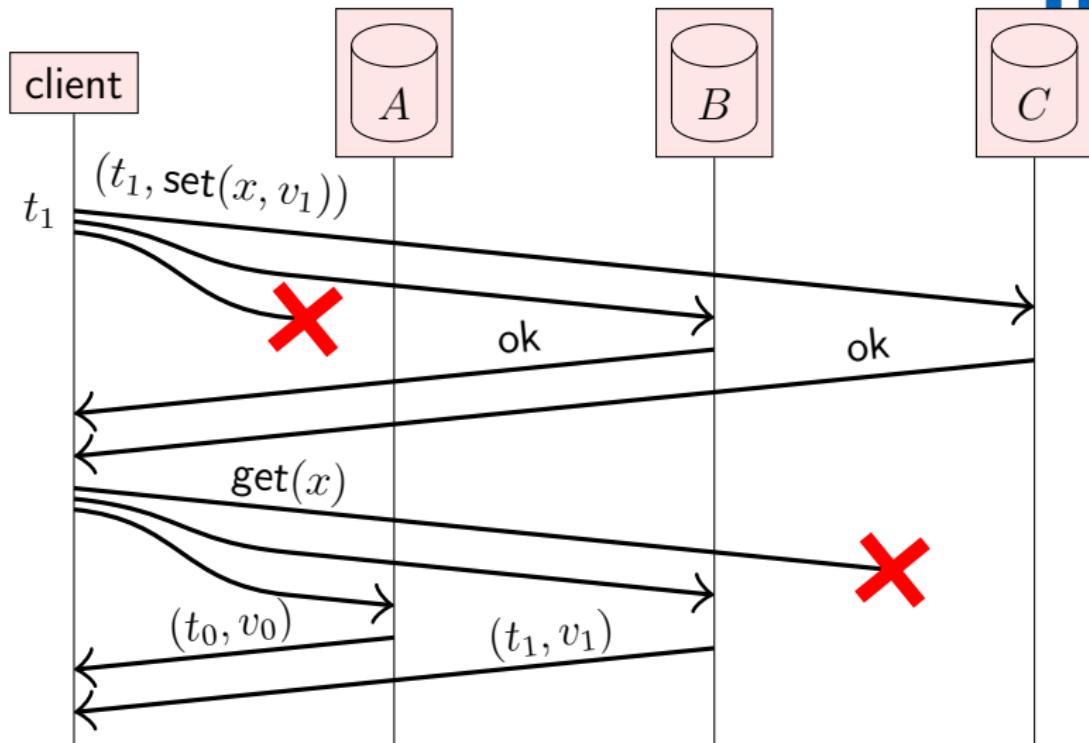
Write succeeds on  $B$  and  $C$

# Quorum (2 out of 3)



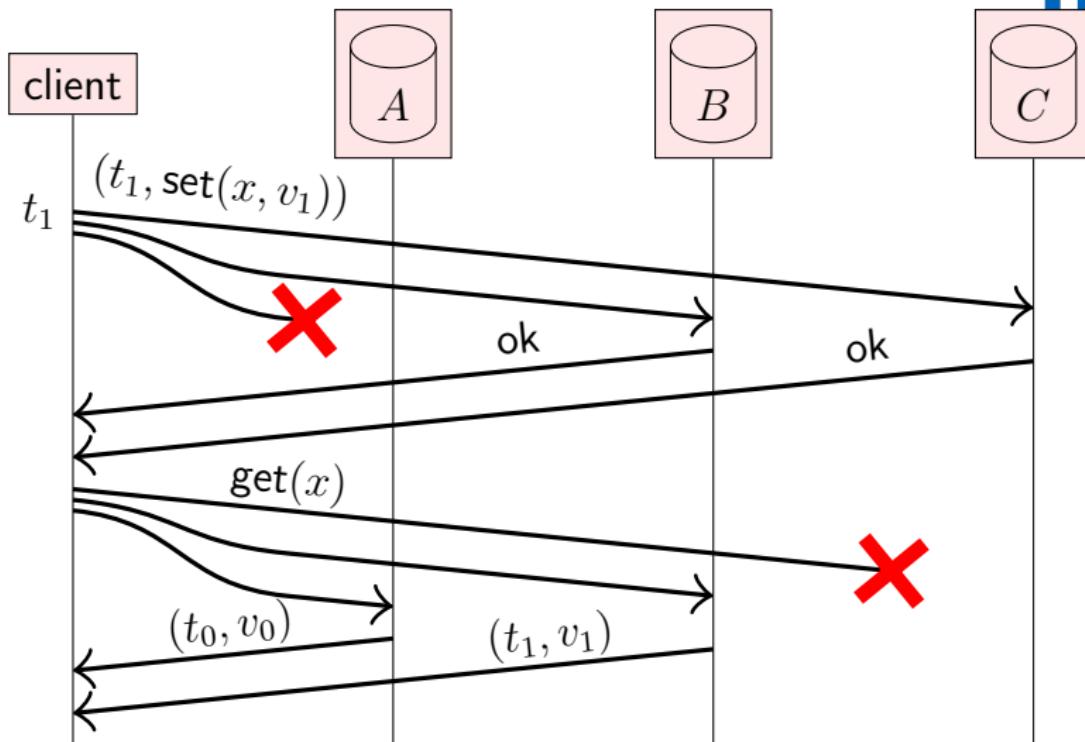
Write succeeds on  $B$  and  $C$

# Quorum (2 out of 3)



Write succeeds on  $B$  and  $C$ ; read succeeds on  $A$  and  $B$

# Quorum (2 out of 3)



Write succeeds on  $B$  and  $C$ ; read succeeds on  $A$  and  $B$   
Choose between  $(t_0, v_0)$  and  $(t_1, v_1)$  based on timestamp

# Read and write quorums

In a system with  $n$  replicas:

- ▶ If a write is acknowledged by  $w$  replicas (**write quorum**),

# Read and write quorums

In a system with  $n$  replicas:

- ▶ If a write is acknowledged by  $w$  replicas (**write quorum**),
- ▶ and we subsequently read from  $r$  replicas (**read quorum**),
- ▶ and  $r + w > n$ ,

# Read and write quorums

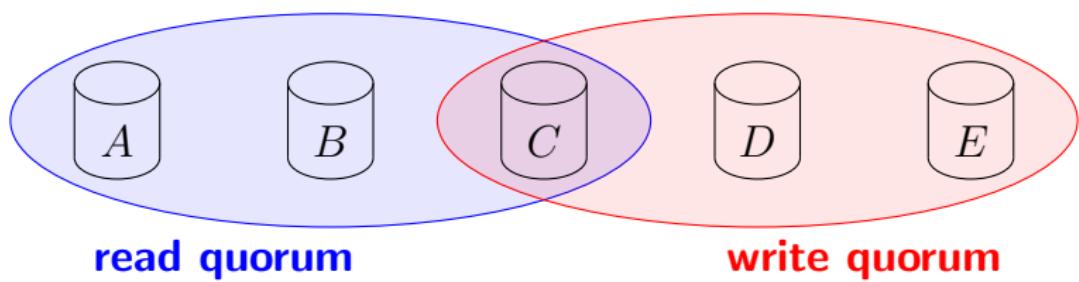
In a system with  $n$  replicas:

- ▶ If a write is acknowledged by  $w$  replicas (**write quorum**),
- ▶ and we subsequently read from  $r$  replicas (**read quorum**),
- ▶ and  $r + w > n$ ,
- ▶ ... then the read will see the previously written value  
(or a value that subsequently overwrote it)

# Read and write quorums

In a system with  $n$  replicas:

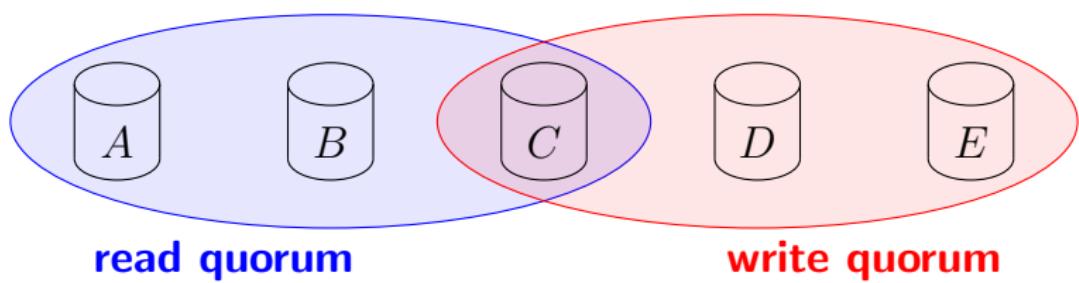
- ▶ If a write is acknowledged by  $w$  replicas (**write quorum**),
- ▶ and we subsequently read from  $r$  replicas (**read quorum**),
- ▶ and  $r + w > n$ ,
- ▶ ... then the read will see the previously written value  
(or a value that subsequently overwrote it)
- ▶ Read quorum and write quorum share  $\geq 1$  replica



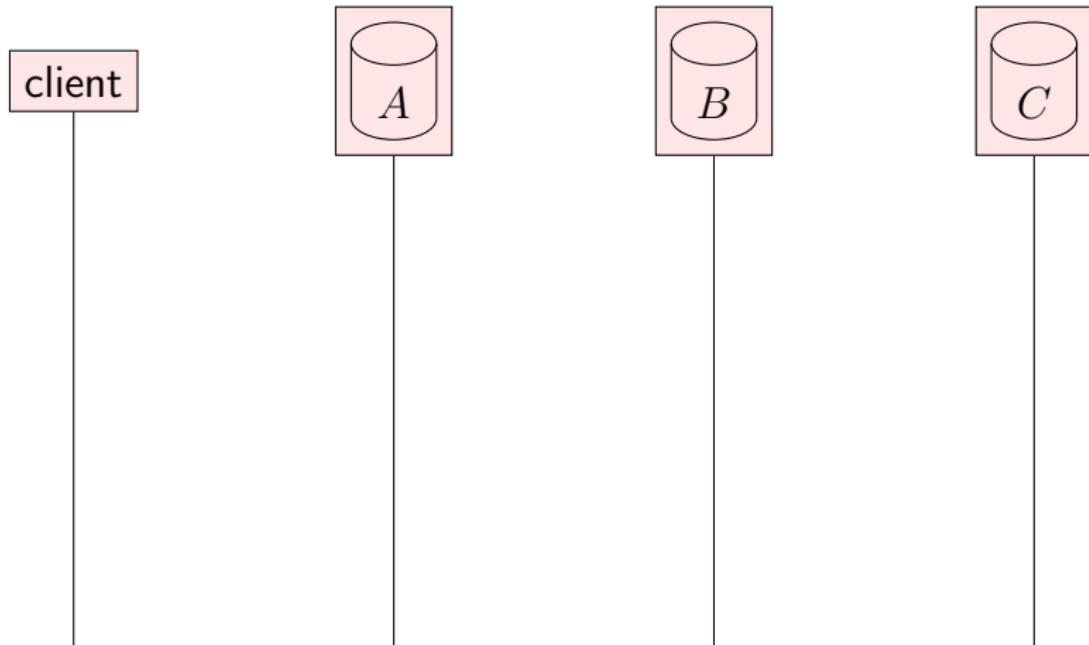
# Read and write quorums

In a system with  $n$  replicas:

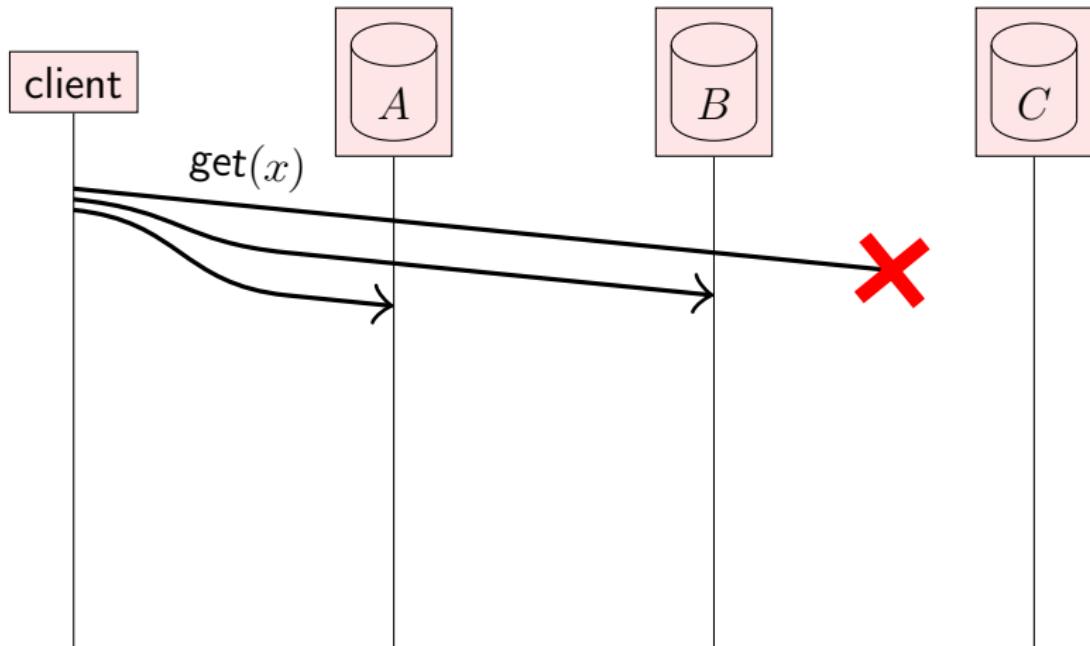
- ▶ If a write is acknowledged by  $w$  replicas (**write quorum**),
- ▶ and we subsequently read from  $r$  replicas (**read quorum**),
- ▶ and  $r + w > n$ ,
- ▶ ... then the read will see the previously written value (or a value that subsequently overwrote it)
- ▶ Read quorum and write quorum share  $\geq 1$  replica
- ▶ Typical:  $r = w = \frac{n+1}{2}$  for  $n = 3, 5, 7, \dots$  (majority)
- ▶ Reads can tolerate  $n - r$  unavailable replicas, writes  $n - w$



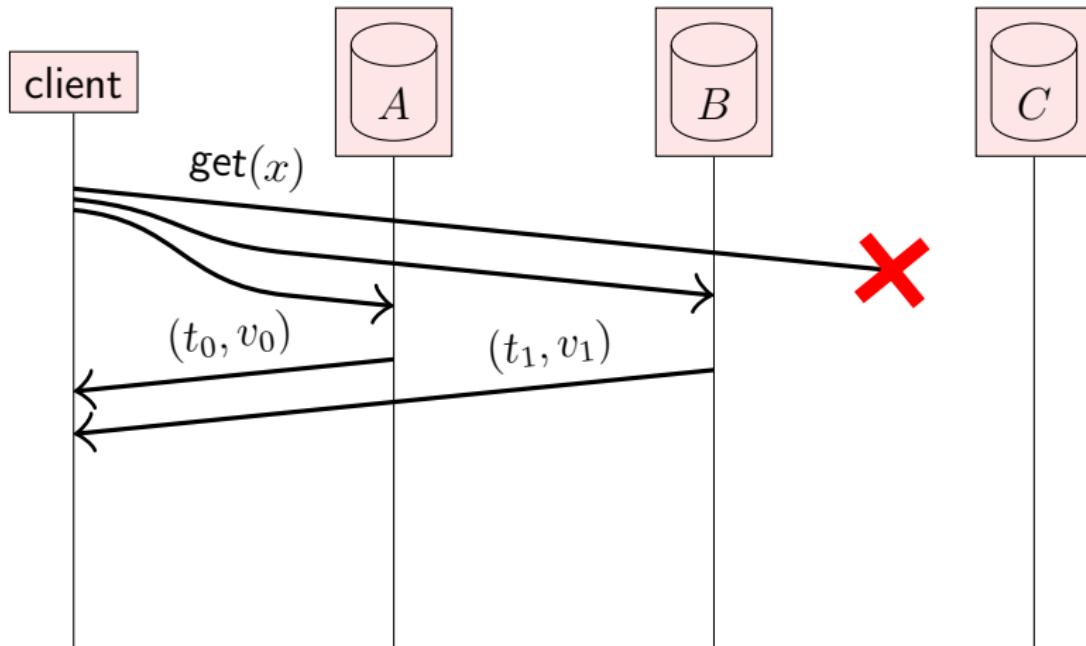
# Read repair



# Read repair

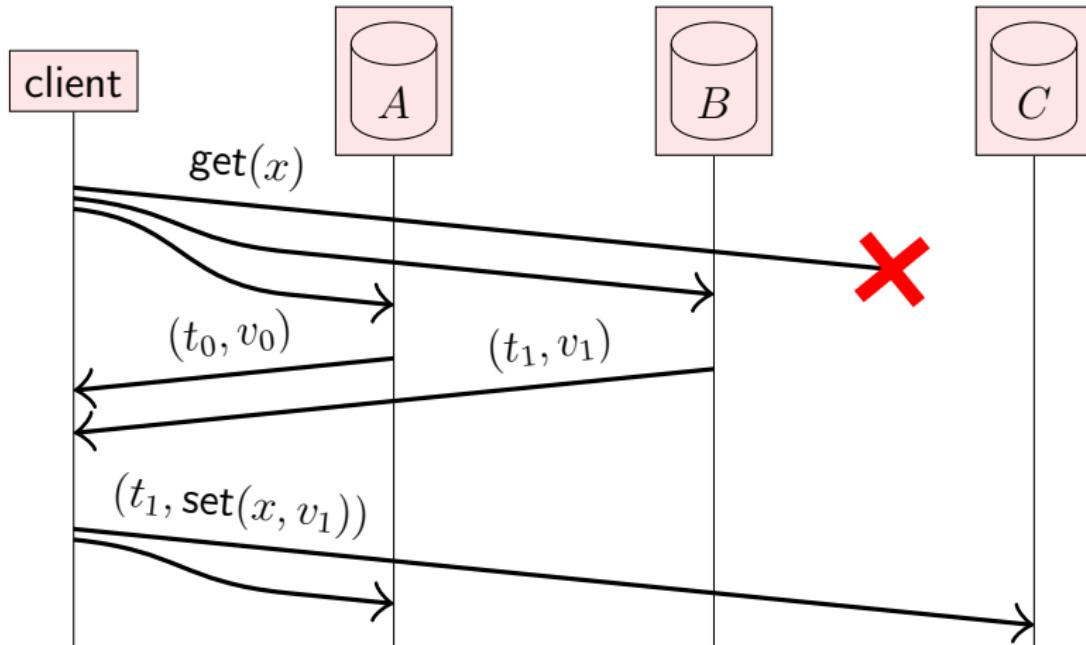


# Read repair



Update  $(t_1, v_1)$  is more recent than  $(t_0, v_0)$  since  $t_0 < t_1$ .

# Read repair



Update  $(t_1, v_1)$  is more recent than  $(t_0, v_0)$  since  $t_0 < t_1$ .  
Client helps **propagate**  $(t_1, v_1)$  to other replicas.

# State machine replication



So far we have used best-effort broadcast for replication.  
What about stronger broadcast models?

# State machine replication

So far we have used best-effort broadcast for replication.  
What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

So far we have used best-effort broadcast for replication.  
What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

**State machine replication** (SMR):

- ▶ FIFO-total order broadcast every update to all replicas
- ▶ Replica delivers update message: apply it to own state

So far we have used best-effort broadcast for replication.  
What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

**State machine replication** (SMR):

- ▶ FIFO-total order broadcast every update to all replicas
- ▶ Replica delivers update message: apply it to own state
- ▶ Applying an update is deterministic

So far we have used best-effort broadcast for replication.  
What about stronger broadcast models?

Total order broadcast: every node delivers the **same messages** in the **same order**

**State machine replication** (SMR):

- ▶ FIFO-total order broadcast every update to all replicas
- ▶ Replica delivers update message: apply it to own state
- ▶ Applying an update is deterministic
- ▶ Replica is a **state machine**: starts in fixed initial state, goes through same sequence of state transitions in the same order  $\implies$  all replicas end up in the same state

# State machine replication

**on** request to perform update  $u$  **do**

send  $u$  via FIFO-total order broadcast

**end on**

**on** delivering  $u$  through FIFO-total order broadcast **do**

update state using arbitrary deterministic logic!

**end on**

# State machine replication

**on** request to perform update  $u$  **do**

send  $u$  via FIFO-total order broadcast

**end on**

**on** delivering  $u$  through FIFO-total order broadcast **do**

update state using arbitrary deterministic logic!

**end on**

Closely related ideas:

- ▶ Serializable transactions (execute in delivery order)

# State machine replication

```
on request to perform update  $u$  do  
    send  $u$  via FIFO-total order broadcast  
end on
```

```
on delivering  $u$  through FIFO-total order broadcast do  
    update state using arbitrary deterministic logic!  
end on
```

Closely related ideas:

- ▶ Serializable transactions (execute in delivery order)
- ▶ Blockchains, distributed ledgers, smart contracts

**on** request to perform update  $u$  **do**

send  $u$  via FIFO-total order broadcast

**end on**

**on** delivering  $u$  through FIFO-total order broadcast **do**

update state using arbitrary deterministic logic!

**end on**

Closely related ideas:

- ▶ Serializable transactions (execute in delivery order)
- ▶ Blockchains, distributed ledgers, smart contracts

Limitations:

- ▶ Cannot update state immediately, have to wait for delivery through broadcast

```
on request to perform update  $u$  do  
    send  $u$  via FIFO-total order broadcast  
end on
```

```
on delivering  $u$  through FIFO-total order broadcast do  
    update state using arbitrary deterministic logic!  
end on
```

Closely related ideas:

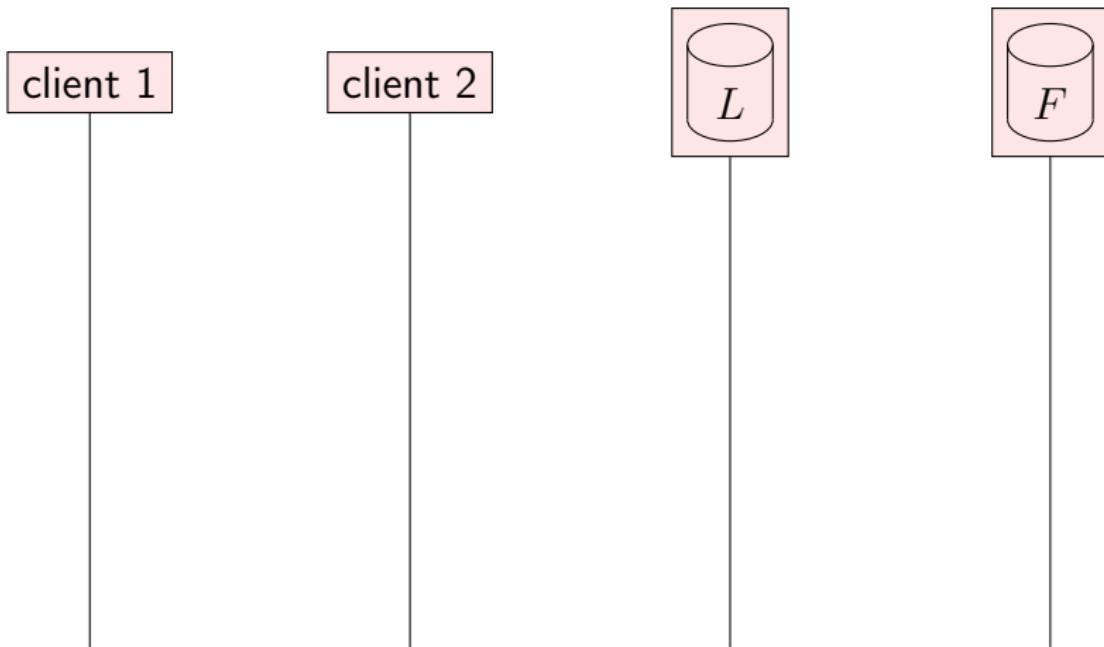
- ▶ Serializable transactions (execute in delivery order)
- ▶ Blockchains, distributed ledgers, smart contracts

Limitations:

- ▶ Cannot update state immediately, have to wait for delivery through broadcast
- ▶ Need fault-tolerant total order broadcast: see next lecture

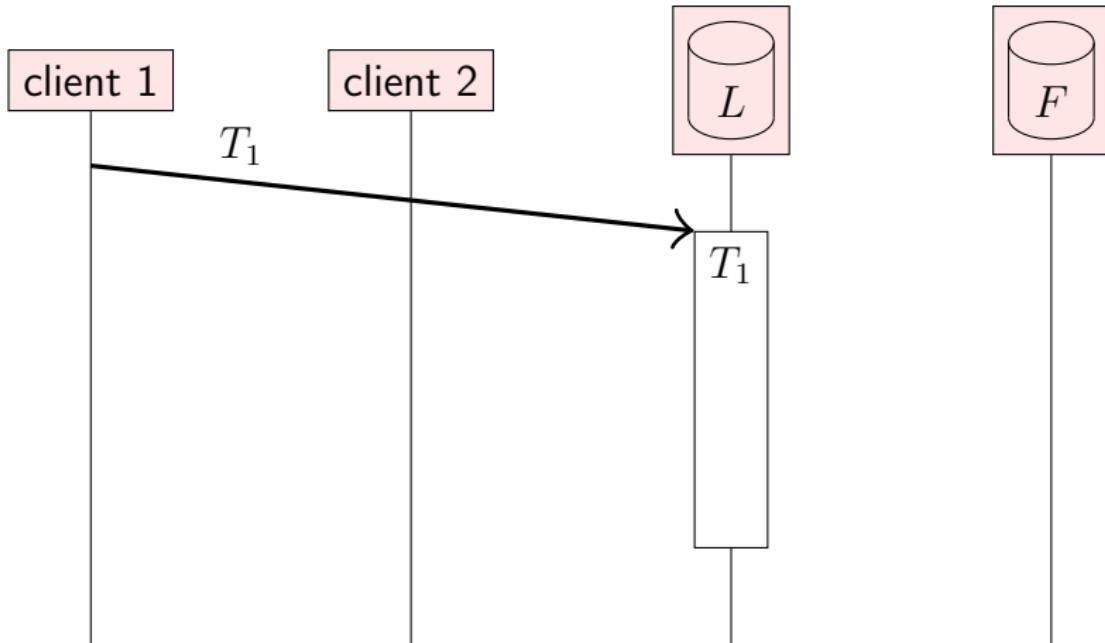
# Database leader replica

Leader database replica  $L$  ensures total order broadcast



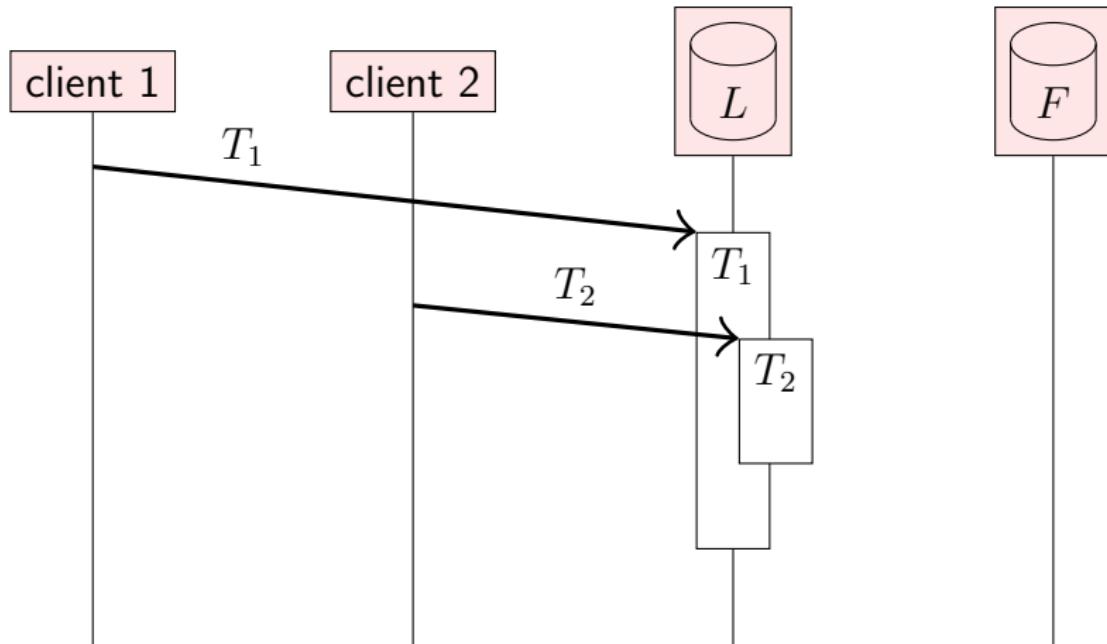
# Database leader replica

Leader database replica  $L$  ensures total order broadcast



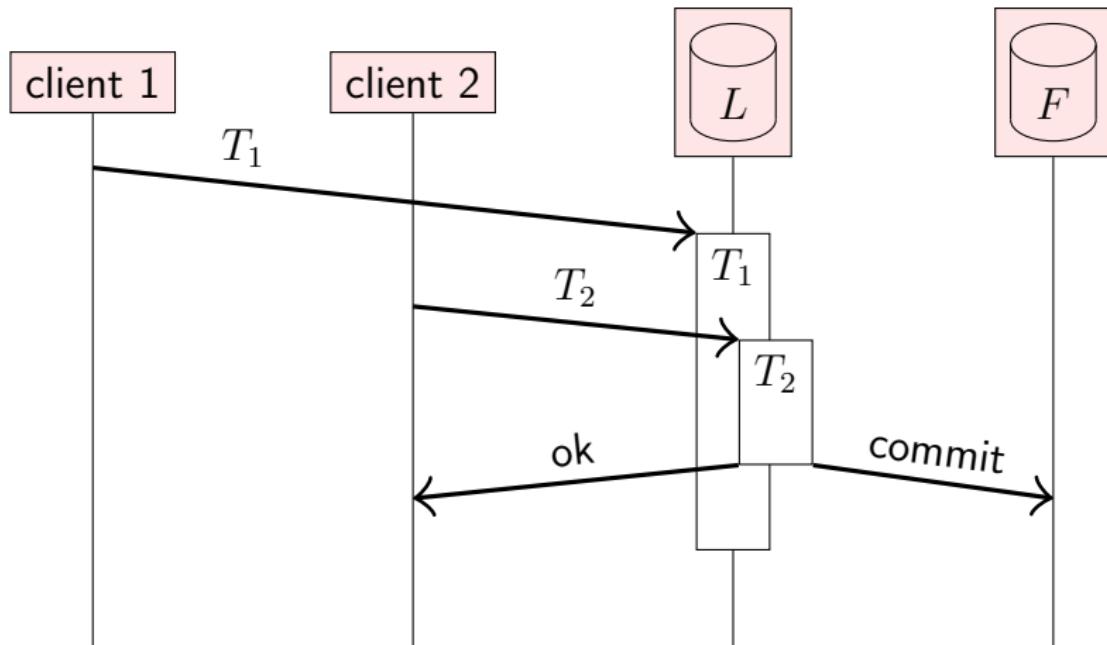
# Database leader replica

Leader database replica  $L$  ensures total order broadcast



# Database leader replica

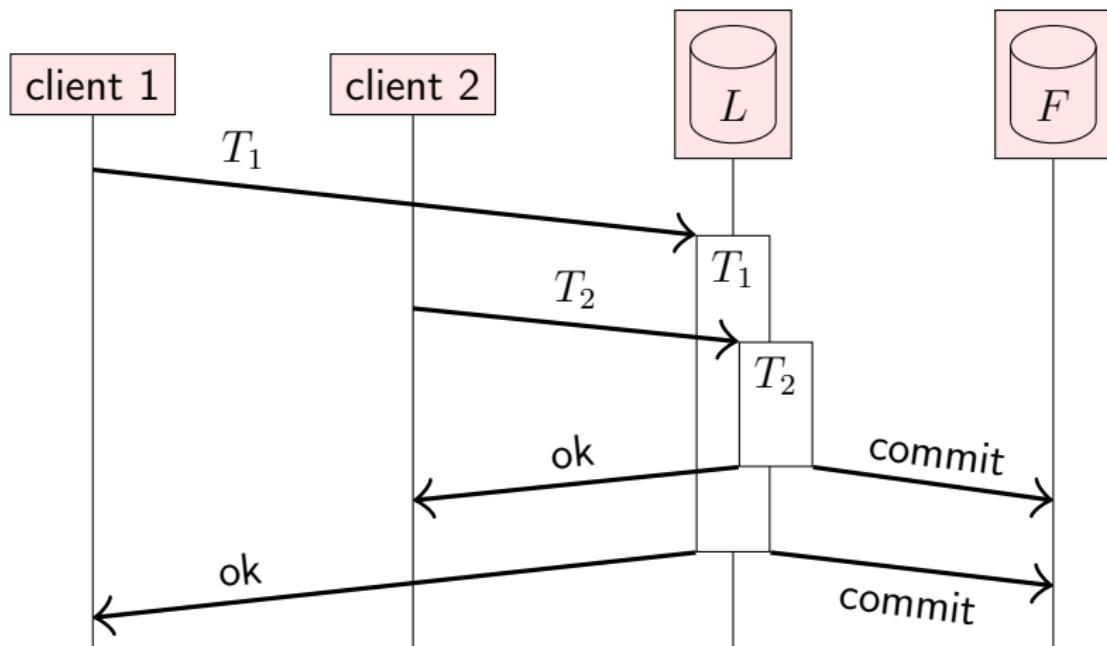
Leader database replica  $L$  ensures total order broadcast



Follower  $F$  applies transaction log in commit order

# Database leader replica

Leader database replica  $L$  ensures total order broadcast



Follower  $F$  applies transaction log in commit order

# Replication using weaker broadcast



State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

# Replication using weaker broadcast



State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates  $f$  and  $g$  are commutative if  $f(g(x)) = g(f(x))$

# Replication using weaker broadcast



State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates  $f$  and  $g$  are commutative if  $f(g(x)) = g(f(x))$

broadcast	assumptions about state update function
total order	deterministic (SMR)

# Replication using weaker broadcast

State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates  $f$  and  $g$  are commutative if  $f(g(x)) = g(f(x))$

broadcast	assumptions about state update function
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute

# Replication using weaker broadcast

State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates  $f$  and  $g$  are commutative if  $f(g(x)) = g(f(x))$

broadcast	assumptions about state update function
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute
reliable	deterministic, all updates commute

# Replication using weaker broadcast

State machine replication uses (FIFO-)total order broadcast.  
Can we use weaker forms of broadcast too?

If replica state updates are **commutative**, replicas can process updates in different orders and still end up in the same state.

Updates  $f$  and  $g$  are commutative if  $f(g(x)) = g(f(x))$

broadcast	assumptions about state update function
total order	deterministic (SMR)
causal	deterministic, concurrent updates commute
reliable	deterministic, all updates commute
best-effort	deterministic, commutative, idempotent, tolerates message loss

# Replica consistency

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

# “Consistency”



A word that means many different things in different contexts!

# “Consistency”



A word that means many different things in different contexts!

- ▶ **ACID:** a transaction transforms the database from one “consistent” state to another

# “Consistency”

A word that means many different things in different contexts!

- ▶ **ACID:** a transaction transforms the database from one “consistent” state to another

Here, “consistent” = satisfying application-specific invariants

e.g. “every course with students enrolled must have at least one lecturer”

# “Consistency”

A word that means many different things in different contexts!

- ▶ **ACID:** a transaction transforms the database from one “consistent” state to another

Here, “consistent” = satisfying application-specific invariants

e.g. “every course with students enrolled must have at least one lecturer”

- ▶ **Read-after-write consistency** (lecture 5)

# “Consistency”



A word that means many different things in different contexts!

- ▶ **ACID:** a transaction transforms the database from one “consistent” state to another

Here, “consistent” = satisfying application-specific invariants

e.g. “every course with students enrolled must have at least one lecturer”

- ▶ **Read-after-write consistency** (lecture 5)
- ▶ **Replication:** replica should be “consistent” with other replicas

# “Consistency”

A word that means many different things in different contexts!

- ▶ **ACID:** a transaction transforms the database from one “consistent” state to another

Here, “consistent” = satisfying application-specific invariants

e.g. “every course with students enrolled must have at least one lecturer”

- ▶ **Read-after-write consistency** (lecture 5)
- ▶ **Replication:** replica should be “consistent” with other replicas

“consistent” = in the same state? (when exactly?)

“consistent” = read operations return same result?

- ▶ **Consistency model:** many to choose from

Recall **atomicity** in the context of ACID transactions:

- ▶ A transaction either **commits** or **aborts**

Recall **atomicity** in the context of ACID transactions:

- ▶ A transaction either **commits** or **aborts**
- ▶ If it commits, its updates are durable
- ▶ If it aborts, it has no visible side-effects

Recall **atomicity** in the context of ACID transactions:

- ▶ A transaction either **commits** or **aborts**
- ▶ If it commits, its updates are durable
- ▶ If it aborts, it has no visible side-effects
- ▶ ACID consistency (preserving invariants) relies on atomicity

Recall **atomicity** in the context of ACID transactions:

- ▶ A transaction either **commits** or **aborts**
- ▶ If it commits, its updates are durable
- ▶ If it aborts, it has no visible side-effects
- ▶ ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

- ▶ Either all nodes must commit, or all must abort

Recall **atomicity** in the context of ACID transactions:

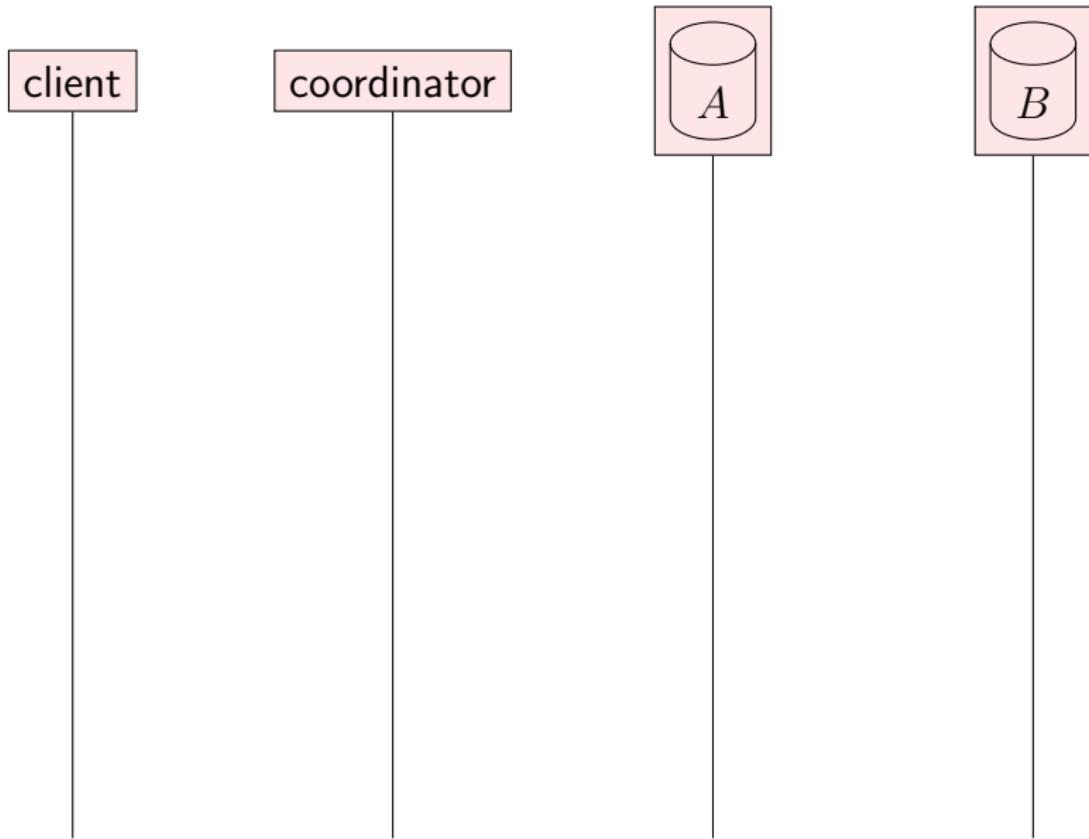
- ▶ A transaction either **commits** or **aborts**
- ▶ If it commits, its updates are durable
- ▶ If it aborts, it has no visible side-effects
- ▶ ACID consistency (preserving invariants) relies on atomicity

If the transaction updates data on multiple nodes, this implies:

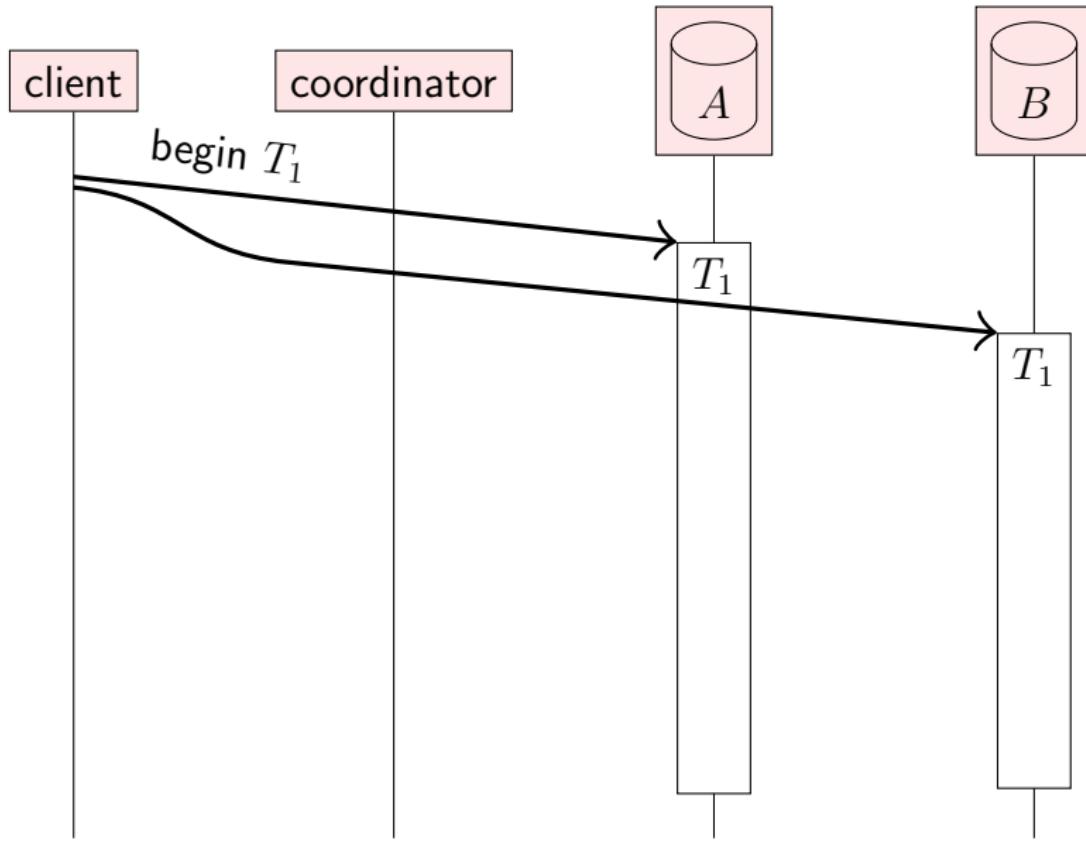
- ▶ Either all nodes must commit, or all must abort
- ▶ If any node crashes, all must abort

Ensuring this is the **atomic commitment** problem.

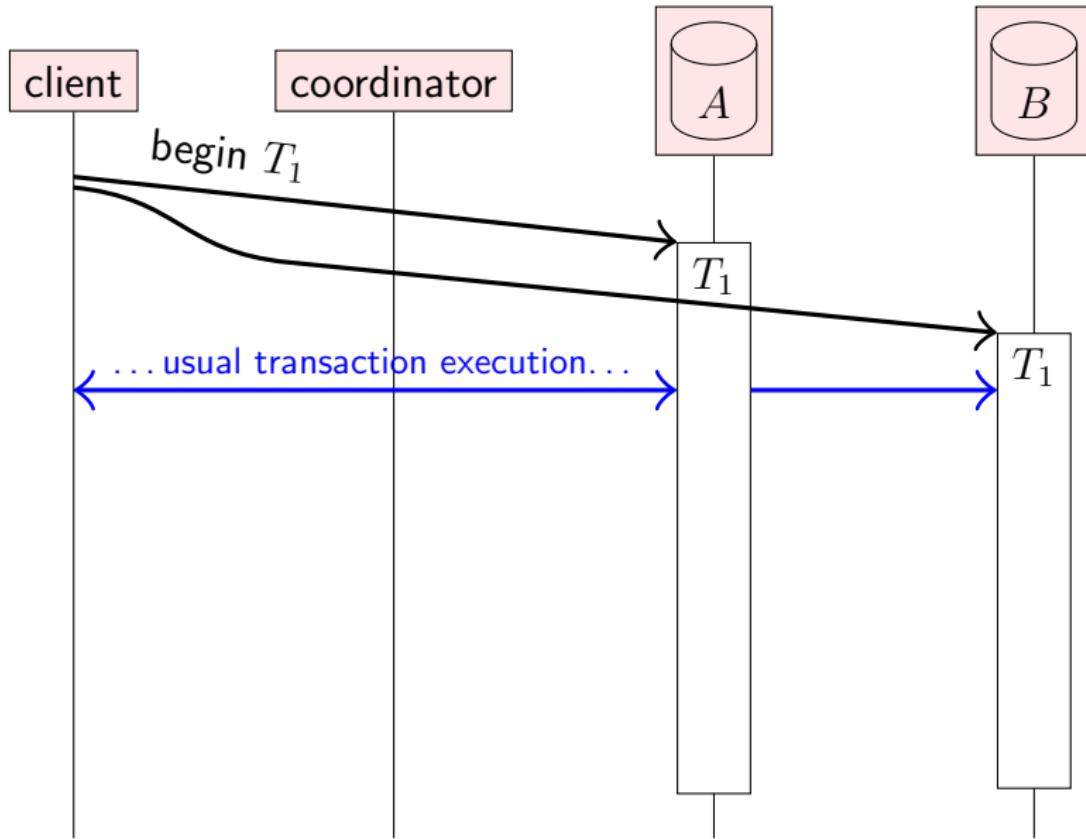
# Two-phase commit (2PC)



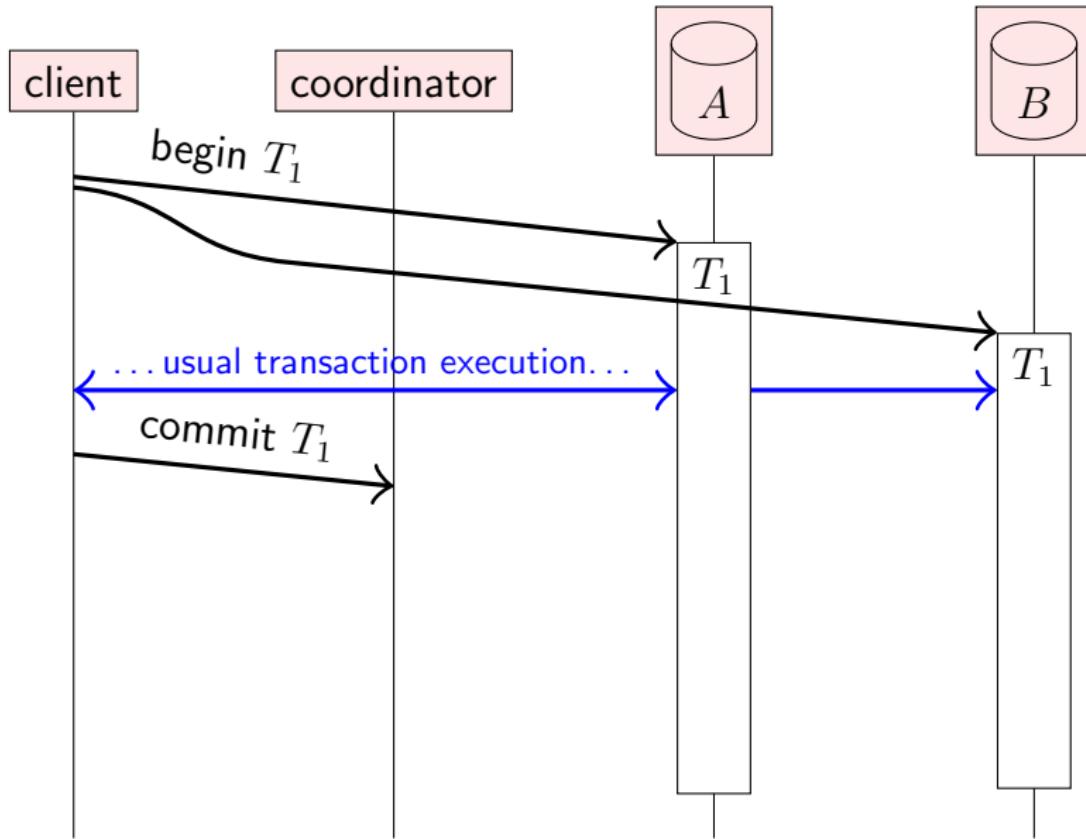
# Two-phase commit (2PC)



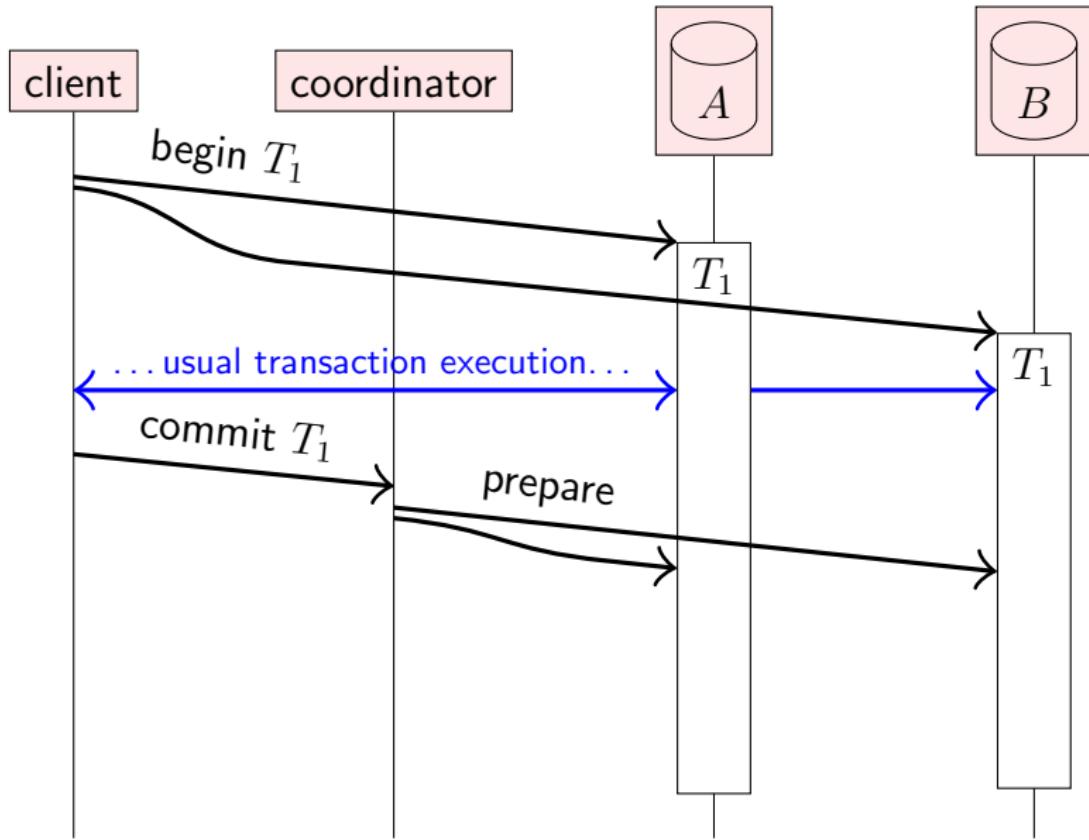
# Two-phase commit (2PC)



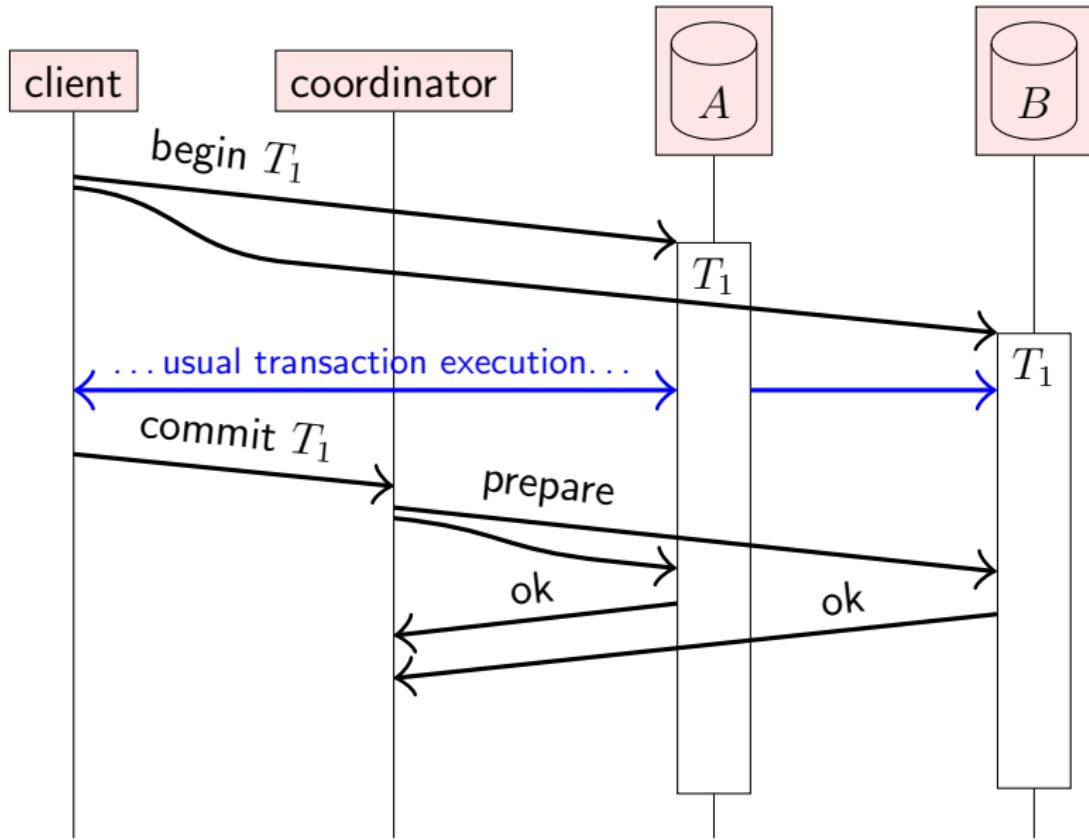
# Two-phase commit (2PC)



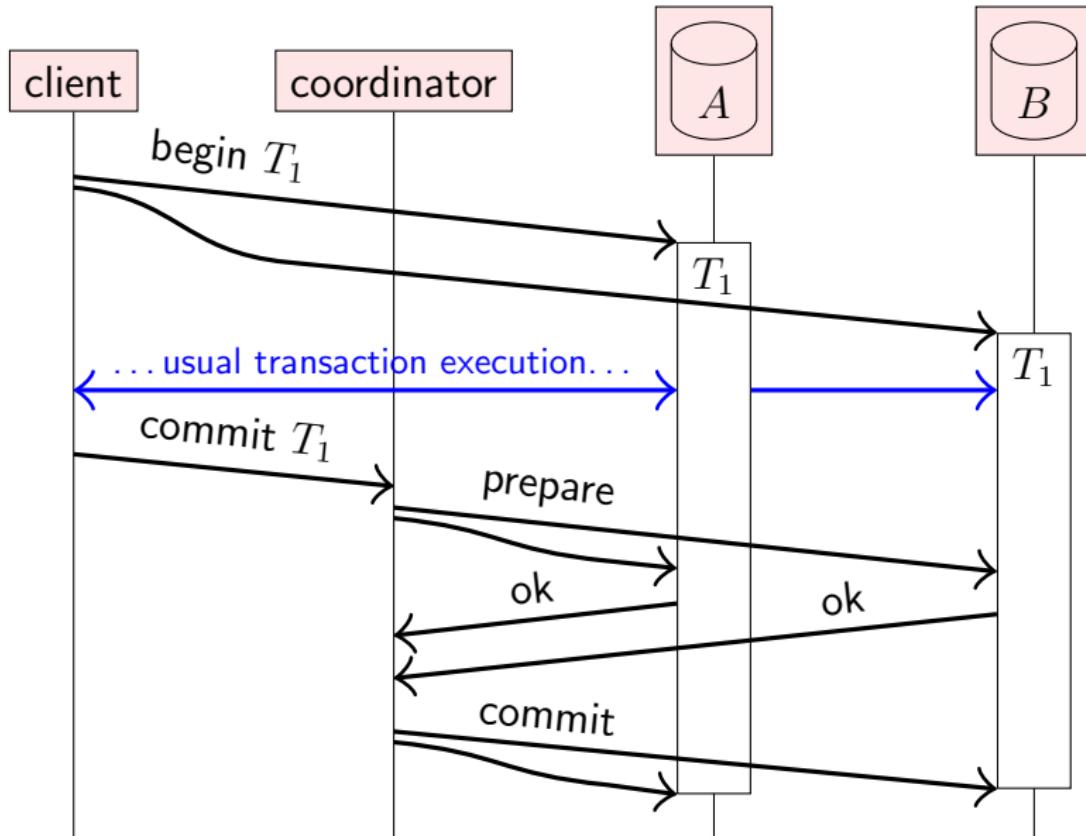
# Two-phase commit (2PC)



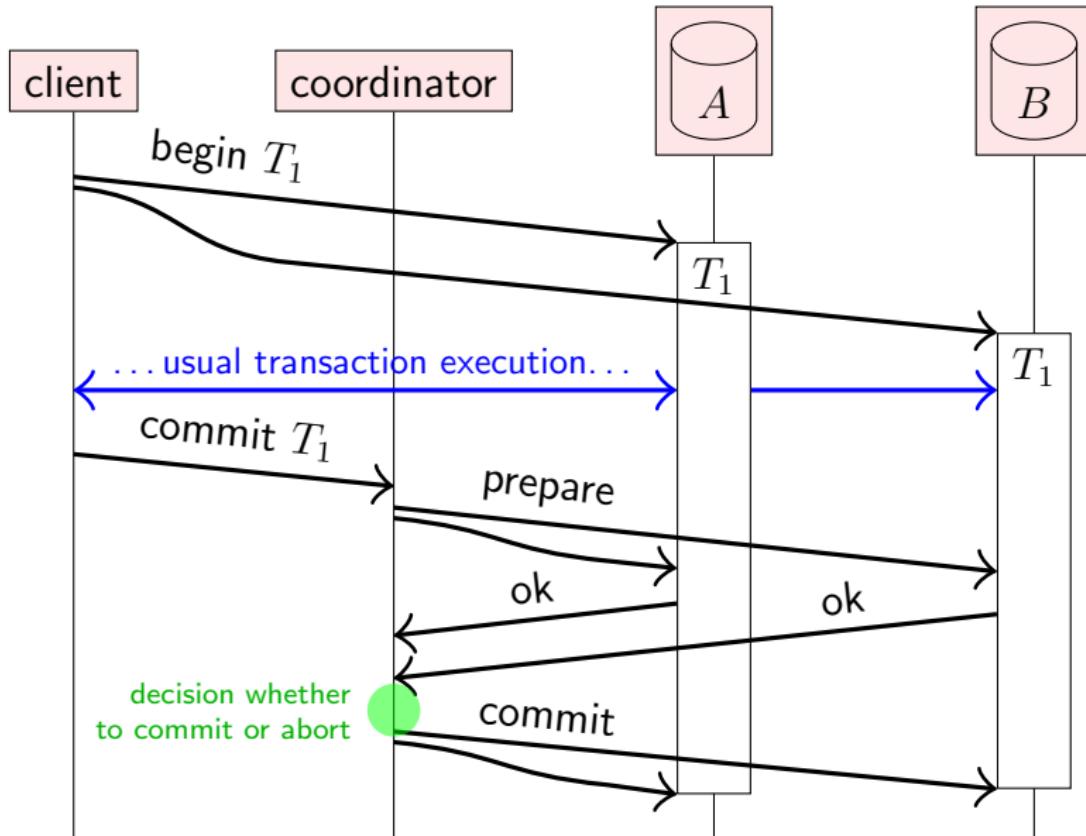
# Two-phase commit (2PC)



# Two-phase commit (2PC)



# Two-phase commit (2PC)



# The coordinator in two-phase commit



What if the coordinator crashes?

# The coordinator in two-phase commit



What if the coordinator crashes?

- ▶ Coordinator writes its decision to disk
- ▶ When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)

What if the coordinator crashes?

- ▶ Coordinator writes its decision to disk
- ▶ When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- ▶ **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided

What if the coordinator crashes?

- ▶ Coordinator writes its decision to disk
- ▶ When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- ▶ **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- ▶ Replicas participating in transaction cannot commit or abort after responding “ok” to the *prepare* request (otherwise we risk violating atomicity)

What if the coordinator crashes?

- ▶ Coordinator writes its decision to disk
- ▶ When it recovers, read decision from disk and send it to replicas (or abort if no decision was made before crash)
- ▶ **Problem:** if coordinator crashes after prepare, but before broadcasting decision, other nodes do not know how it has decided
- ▶ Replicas participating in transaction cannot commit or abort after responding “ok” to the *prepare* request (otherwise we risk violating atomicity)
- ▶ Algorithm is blocked until coordinator recovers

# Fault-tolerant two-phase commit (1/2)

**on** initialisation for transaction  $T$  **do**

$commitVotes[T] := \{\}$ ;  $replicas[T] := \{\}$ ;  $decided[T] := \text{false}$

**end on**

**on** request to commit transaction  $T$  with participating nodes  $R$  **do**

**for each**  $r \in R$  **do** send (Prepare,  $T, R$ ) to  $r$

**end on**

**on** receiving (Prepare,  $T, R$ ) at node  $replicaId$  **do**

$replicas[T] := R$

$ok = \text{"is transaction } T \text{ able to commit on this replica?"}$

total order broadcast (Vote,  $T, replicaId, ok$ ) to  $replicas[T]$

**end on**

**on** a node suspects node  $replicaId$  to have crashed **do**

**for each** transaction  $T$  in which  $replicaId$  participated **do**

total order broadcast (Vote,  $T, replicaId, \text{false}$ ) to  $replicas[T]$

**end for**

**end on**

# Fault-tolerant two-phase commit (2/2)

```
on delivering ( $\text{Vote}, T, \text{replicaId}, ok$ ) by total order broadcast do
    if  $\text{replicaId} \notin \text{commitVotes}[T] \wedge \text{replicaId} \in \text{replicas}[T] \wedge$ 
         $\neg \text{decided}[T]$  then
        if  $ok = \text{true}$  then
             $\text{commitVotes}[T] := \text{commitVotes}[T] \cup \{\text{replicaId}\}$ 
            if  $\text{commitVotes}[T] = \text{replicas}[T]$  then
                 $\text{decided}[T] := \text{true}$ 
                commit transaction  $T$  at this node
            end if
        else
             $\text{decided}[T] := \text{true}$ 
            abort transaction  $T$  at this node
        end if
    end if
end on
```

# Linearizability



Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

# Linearizability



Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished

Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished
- ▶ All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)

Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished
- ▶ All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- ▶ Consequence: every operation returns an “up-to-date” value, a.k.a. “strong consistency”

Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished
- ▶ All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- ▶ Consequence: every operation returns an “up-to-date” value, a.k.a. “strong consistency”
- ▶ Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

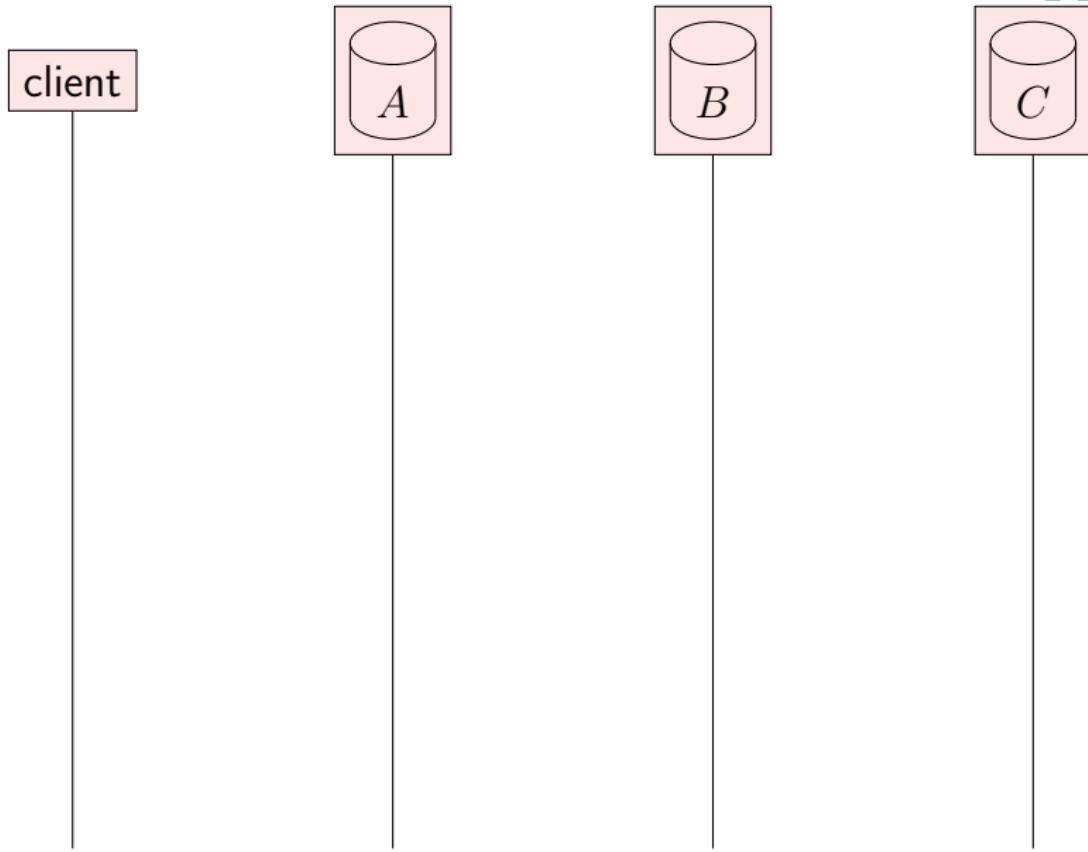
Multiple nodes concurrently accessing replicated data.  
How do we define “consistency” here?

The strongest option: **linearizability**

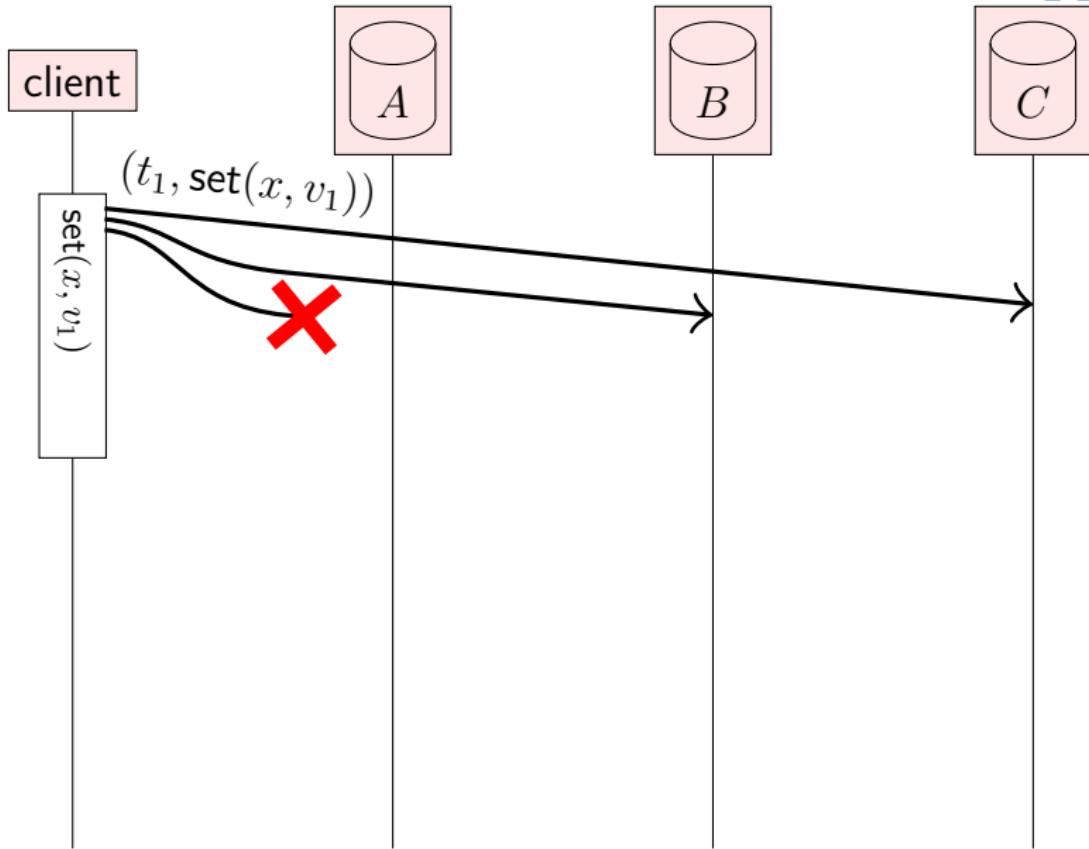
- ▶ Informally: every operation takes effect **atomically** sometime after it started and before it finished
- ▶ All operations behave as if executed on a **single copy** of the data (even if there are in fact multiple replicas)
- ▶ Consequence: every operation returns an “up-to-date” value, a.k.a. “strong consistency”
- ▶ Not just in distributed systems, also in shared-memory concurrency (memory on multi-core CPUs is not linearizable by default!)

Note: linearizability  $\neq$  serializability!

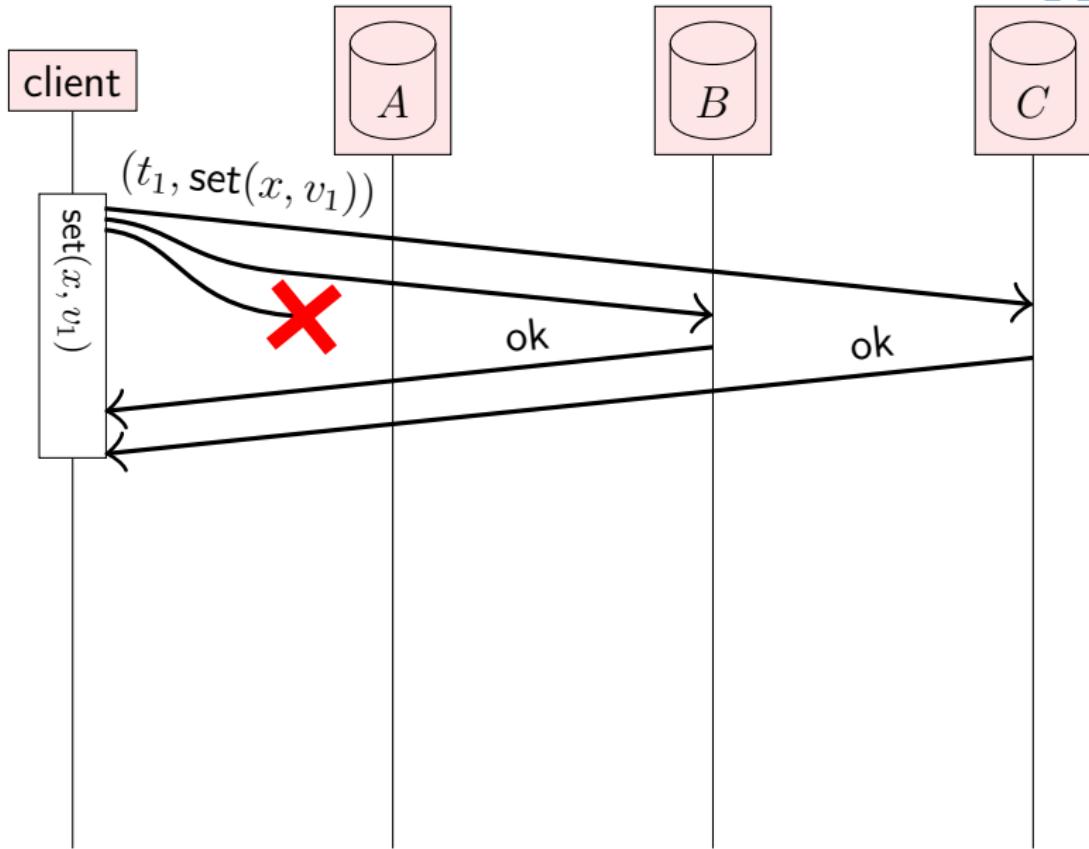
# Read-after-write consistency revisited



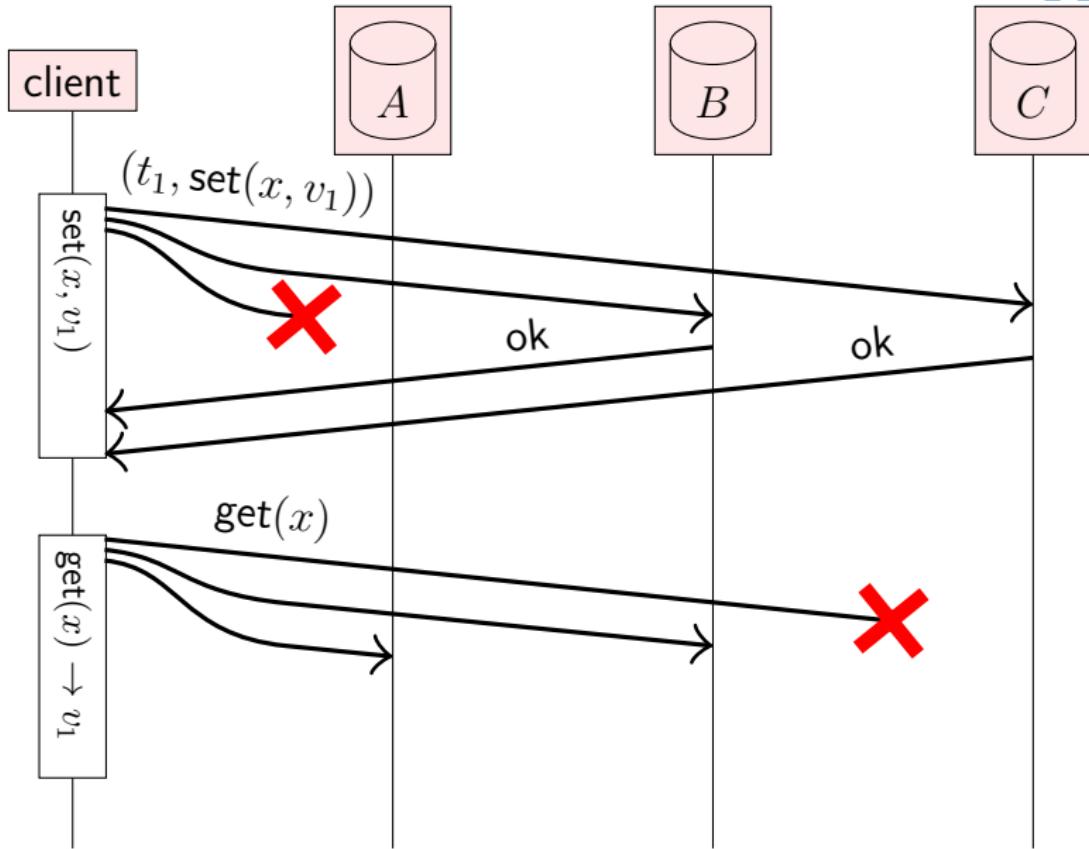
# Read-after-write consistency revisited



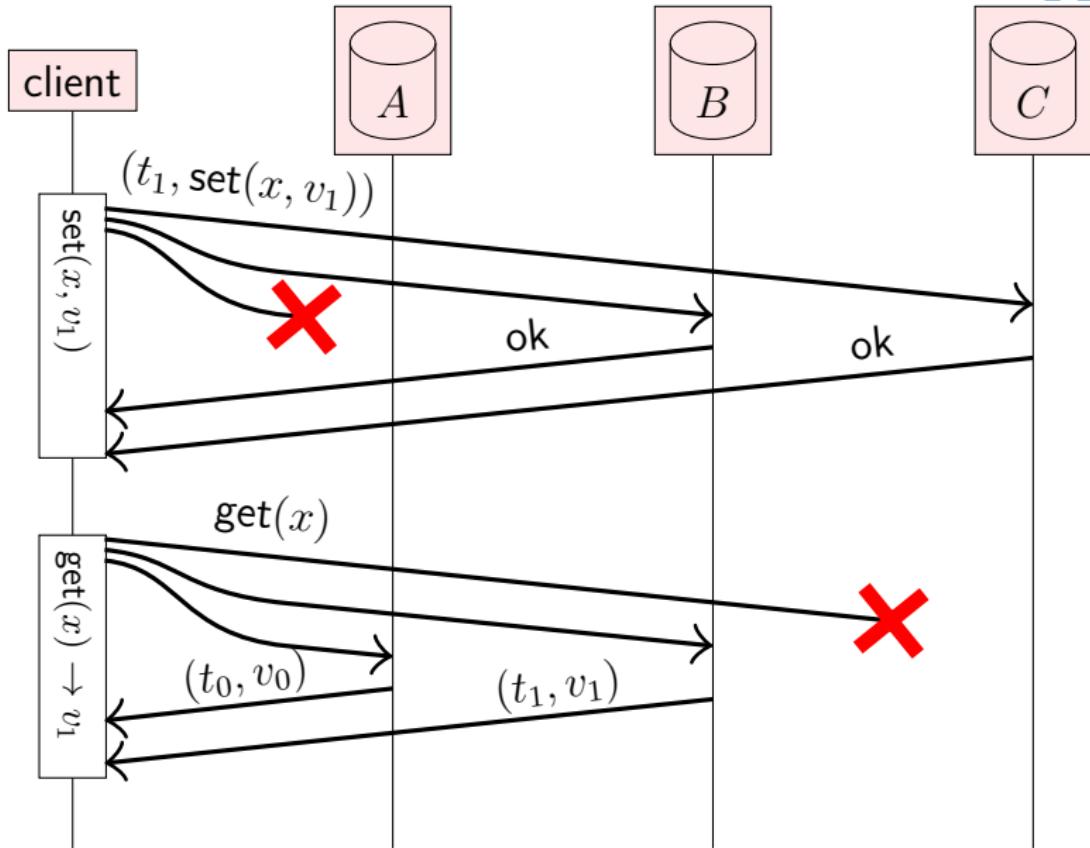
# Read-after-write consistency revisited



# Read-after-write consistency revisited



# Read-after-write consistency revisited



# From the client's point of view

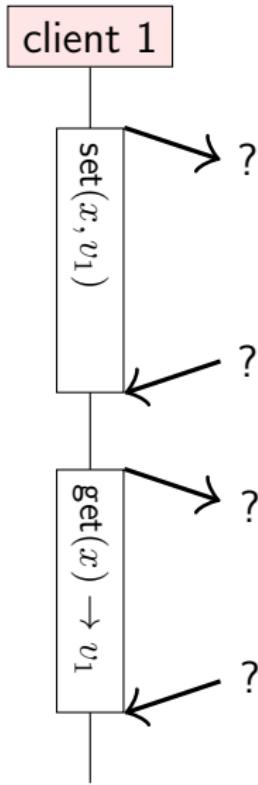
client 1

$\text{set}(x, v_1)$

$\text{get}(x) \rightarrow v_1$

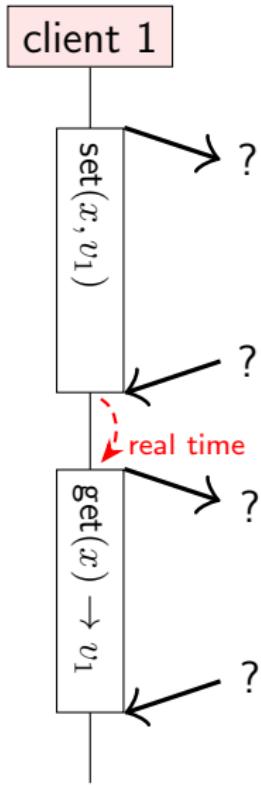
- ▶ Focus on client-observable behaviour: when and what an operation returns

# From the client's point of view



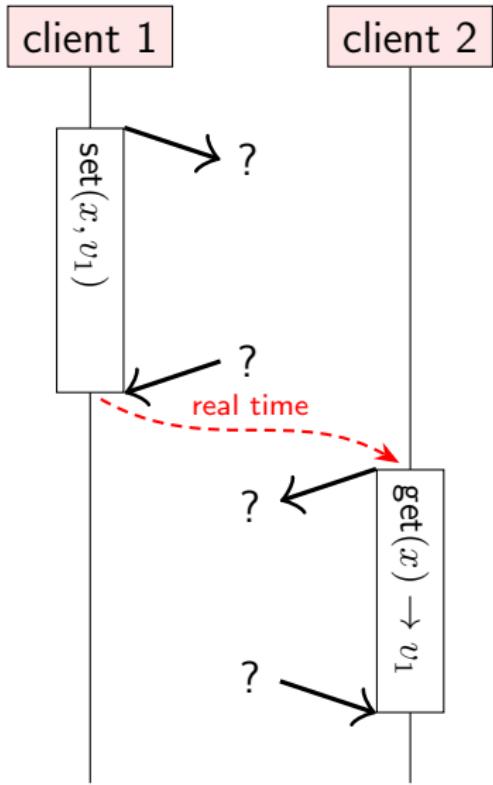
- ▶ Focus on client-observable behaviour: when and what an operation returns
- ▶ Ignore how the replication system is implemented internally

# From the client's point of view



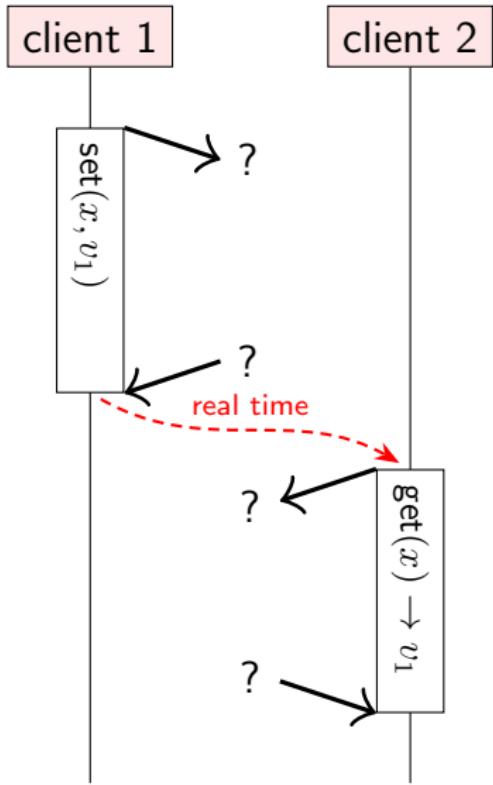
- ▶ Focus on client-observable behaviour: when and what an operation returns
- ▶ Ignore how the replication system is implemented internally
- ▶ Did operation  $A$  finish before operation  $B$  started?

# From the client's point of view



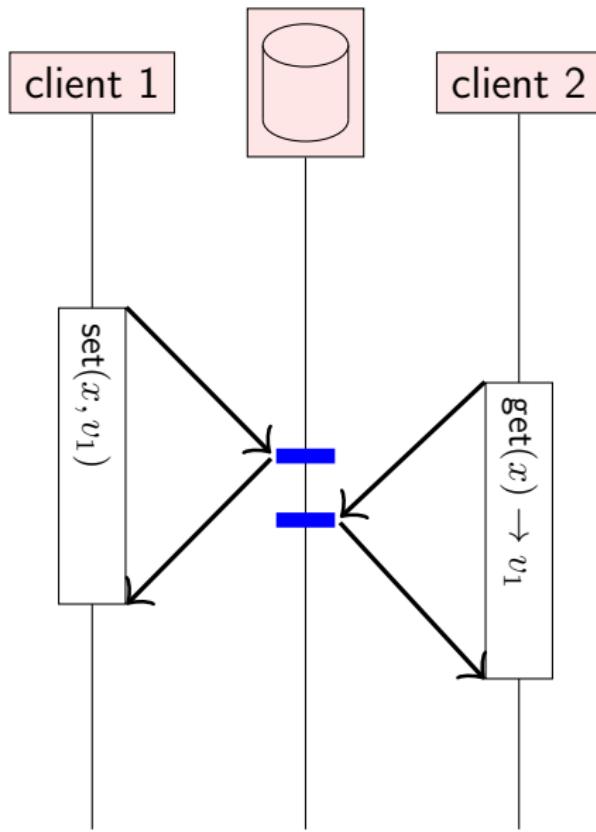
- ▶ Focus on client-observable behaviour: when and what an operation returns
- ▶ Ignore how the replication system is implemented internally
- ▶ Did operation *A* finish before operation *B* started?
- ▶ Even if the operations are on different nodes?

# From the client's point of view



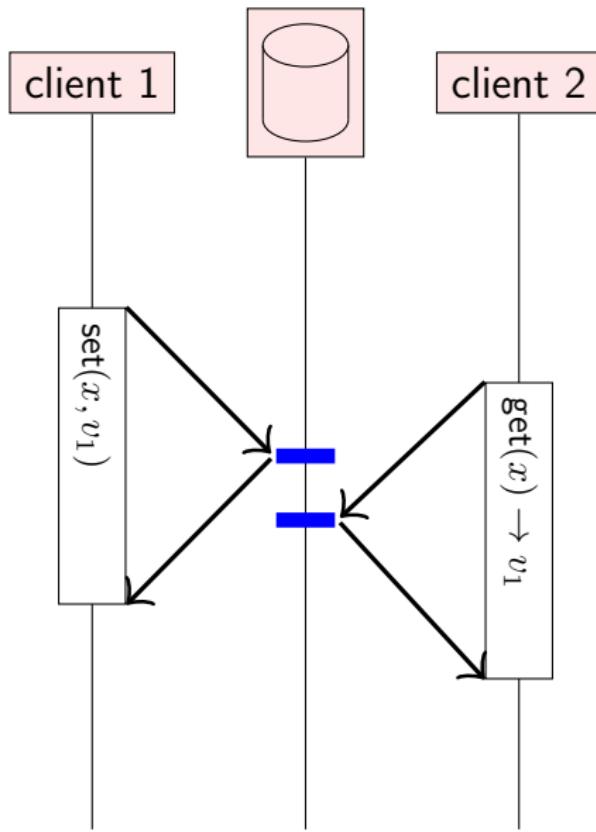
- ▶ Focus on client-observable behaviour: when and what an operation returns
- ▶ Ignore how the replication system is implemented internally
- ▶ Did operation  $A$  finish before operation  $B$  started?
- ▶ Even if the operations are on different nodes?
- ▶ **This is not happens-before:** we want client 2 to read value written by client 1, even if the clients have not communicated!

# Operations overlapping in time



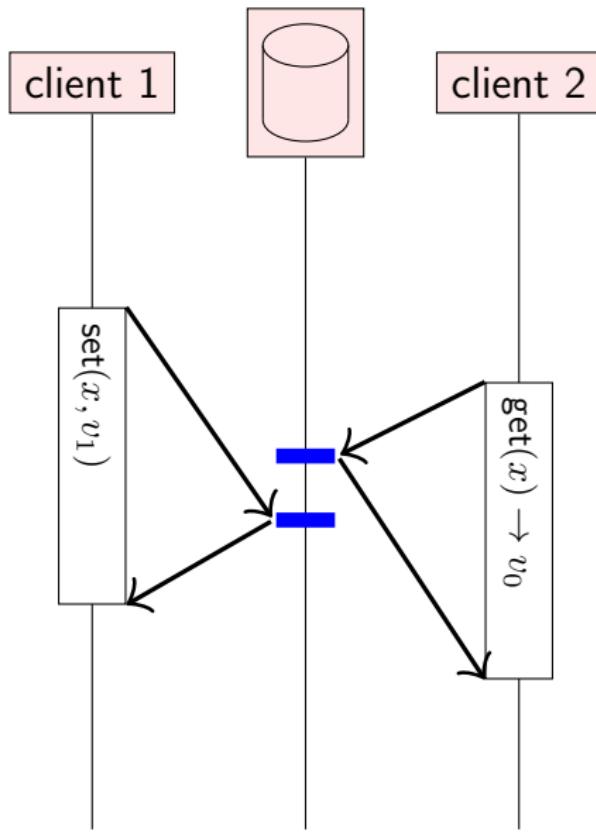
- ▶ Client 2's get operation overlaps in time with client 1's set operation

# Operations overlapping in time



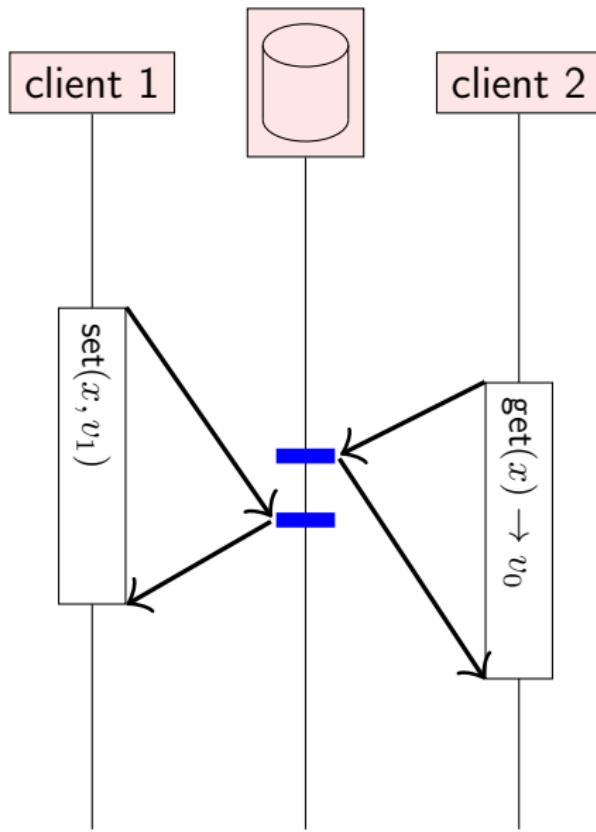
- ▶ Client 2's get operation overlaps in time with client 1's set operation
- ▶ Maybe the set operation takes effect first?

# Operations overlapping in time



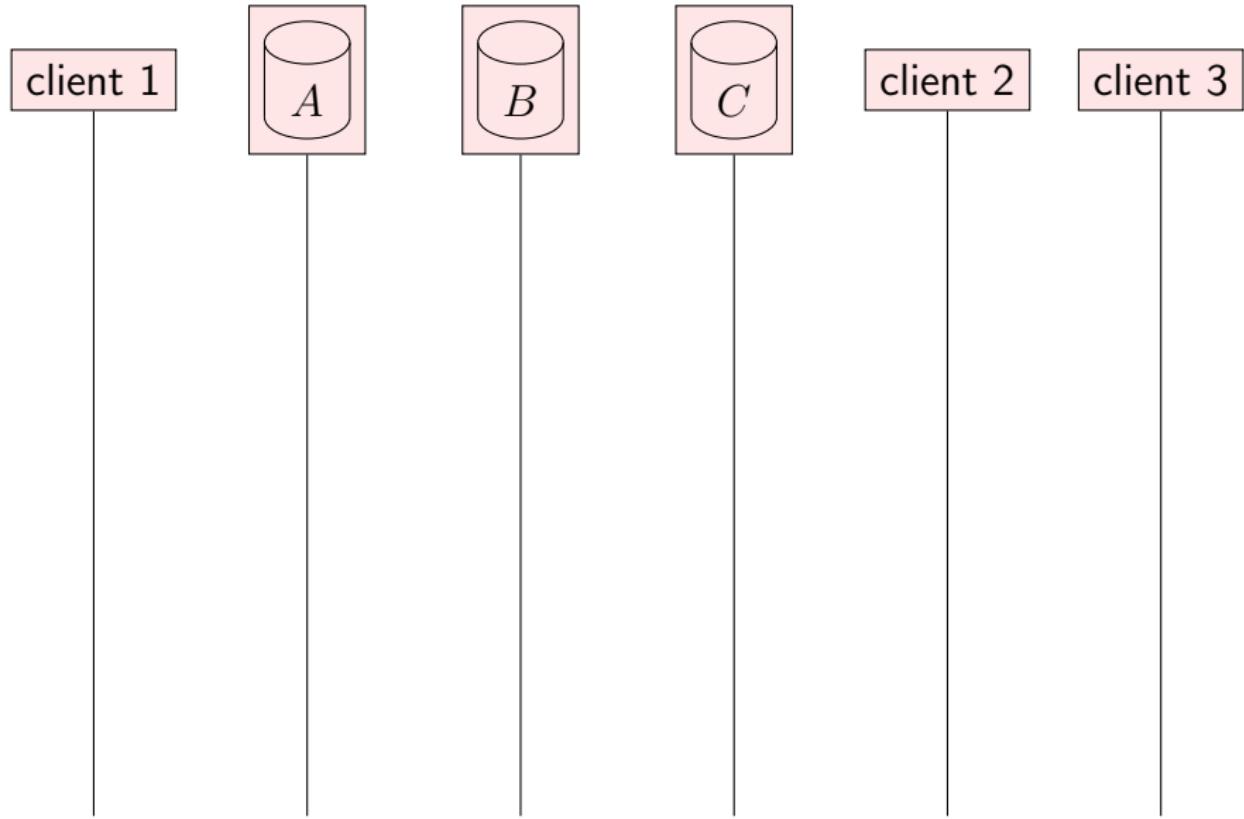
- ▶ Client 2's get operation overlaps in time with client 1's set operation
- ▶ Maybe the set operation takes effect first?
- ▶ Just as likely, the get operation may be executed first

# Operations overlapping in time

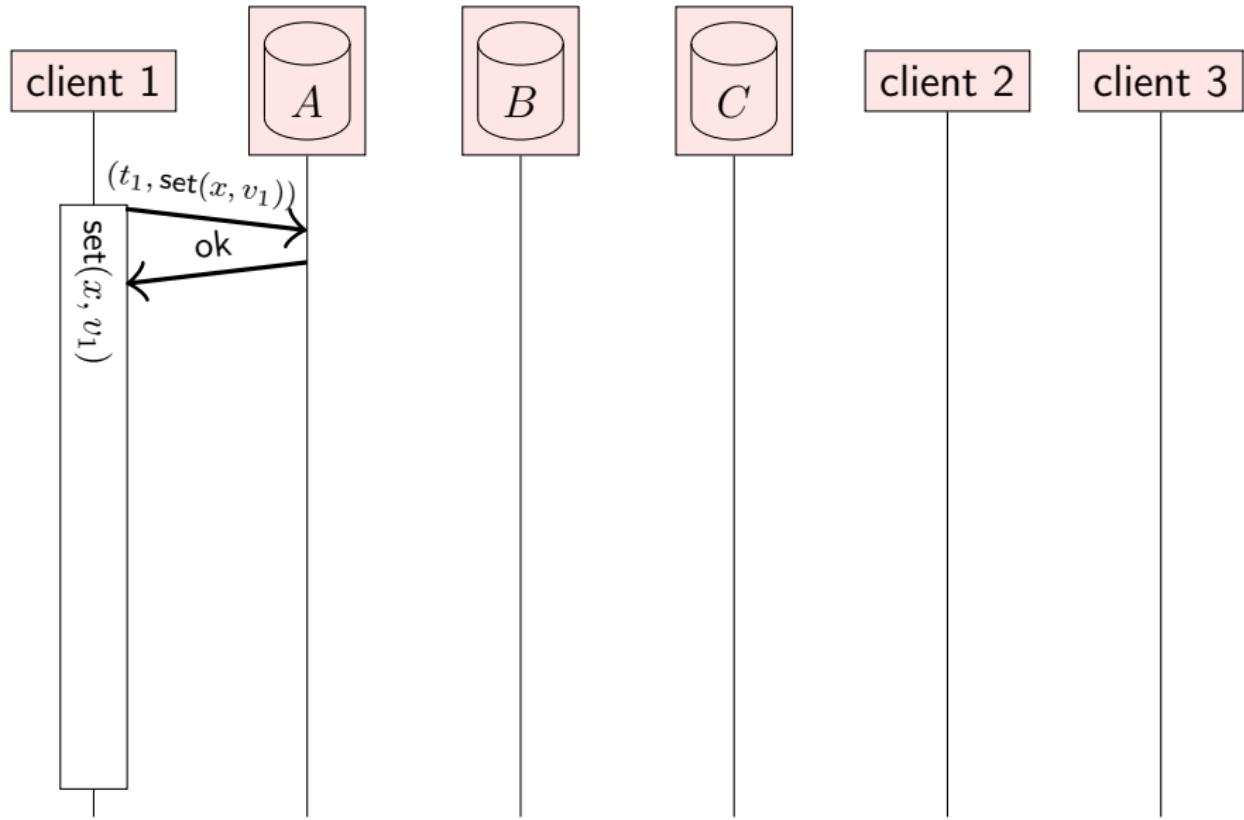


- ▶ Client 2's get operation overlaps in time with client 1's set operation
- ▶ Maybe the set operation takes effect first?
- ▶ Just as likely, the get operation may be executed first
- ▶ Either outcome is fine in this case

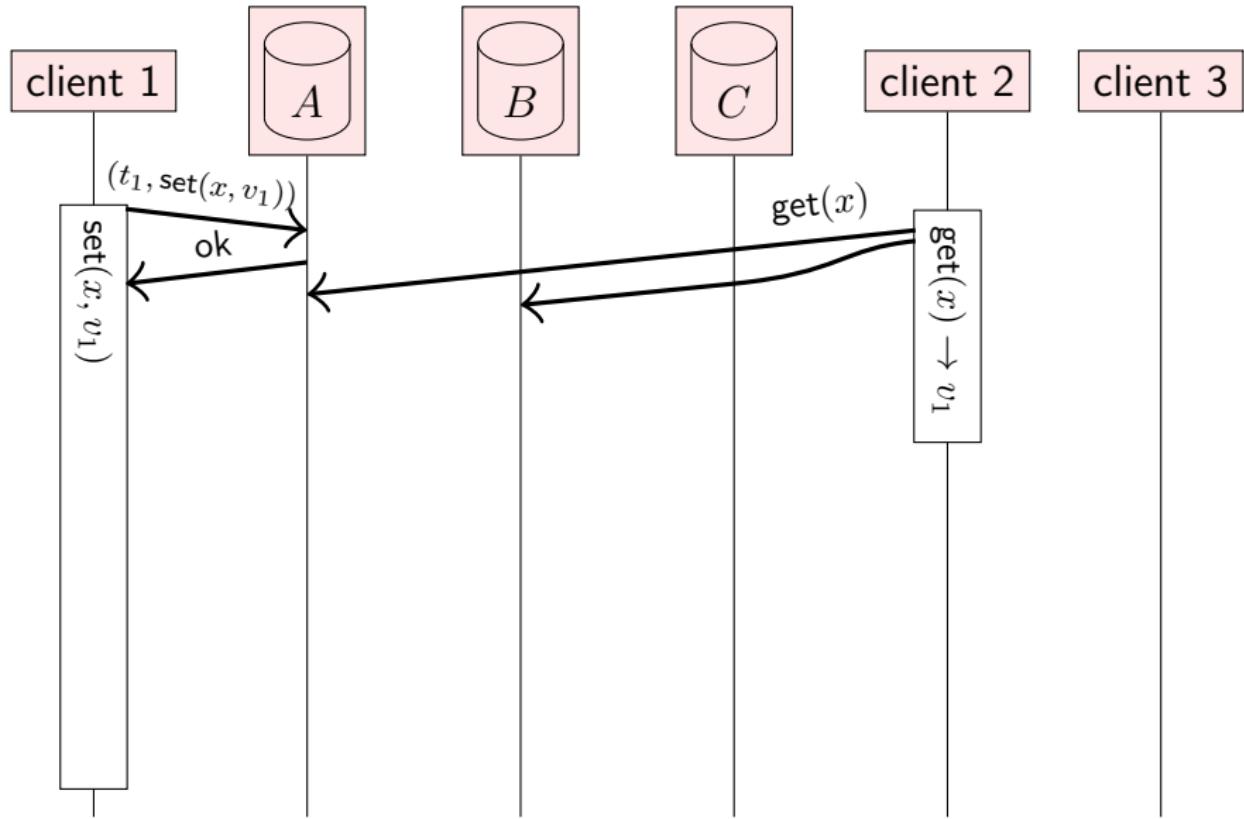
# Not linearizable, despite quorums



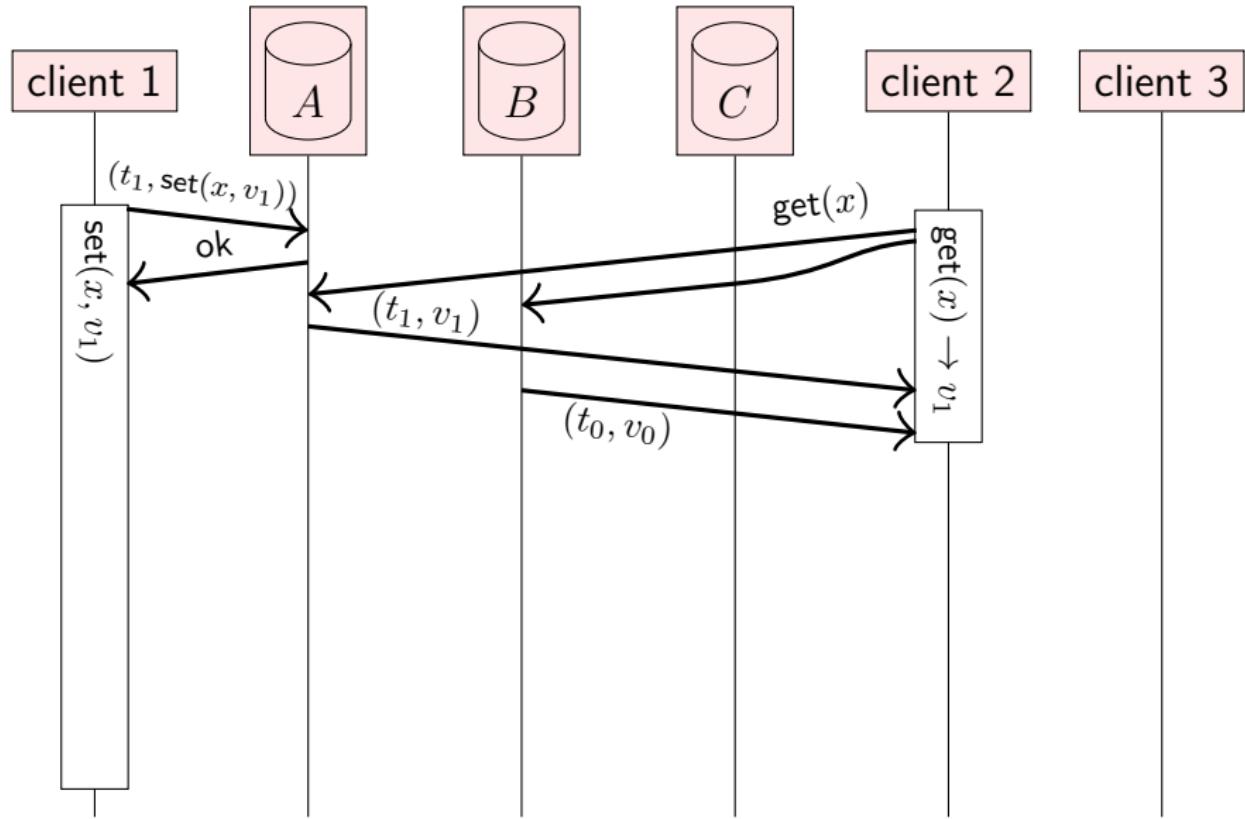
# Not linearizable, despite quorums



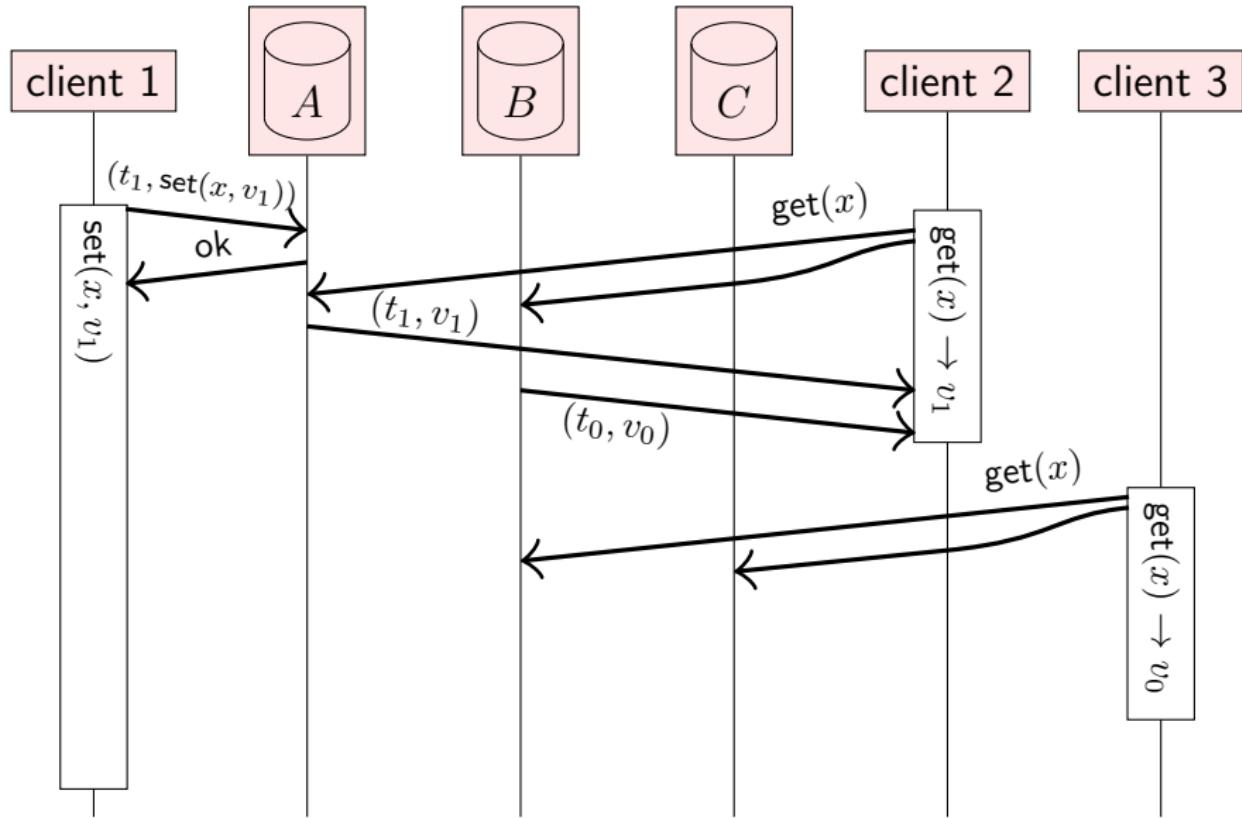
# Not linearizable, despite quorums



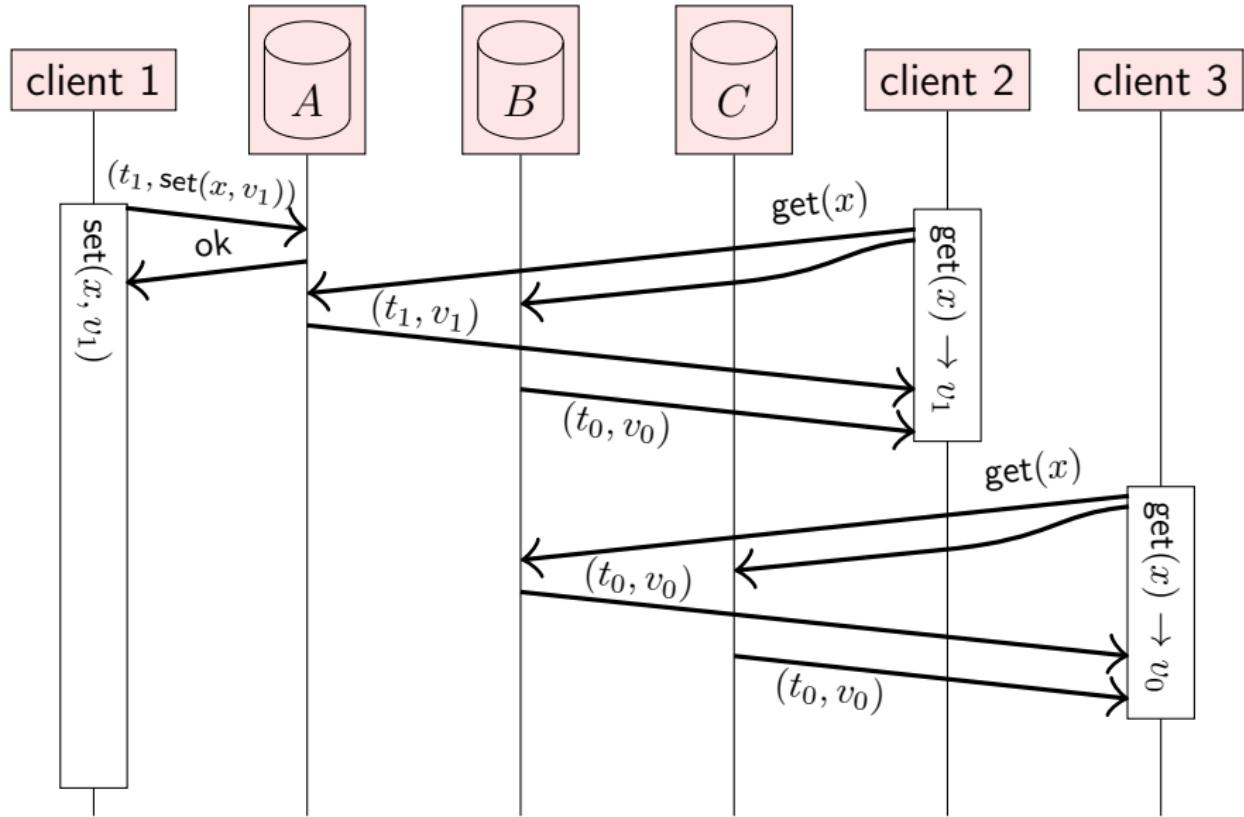
# Not linearizable, despite quorums



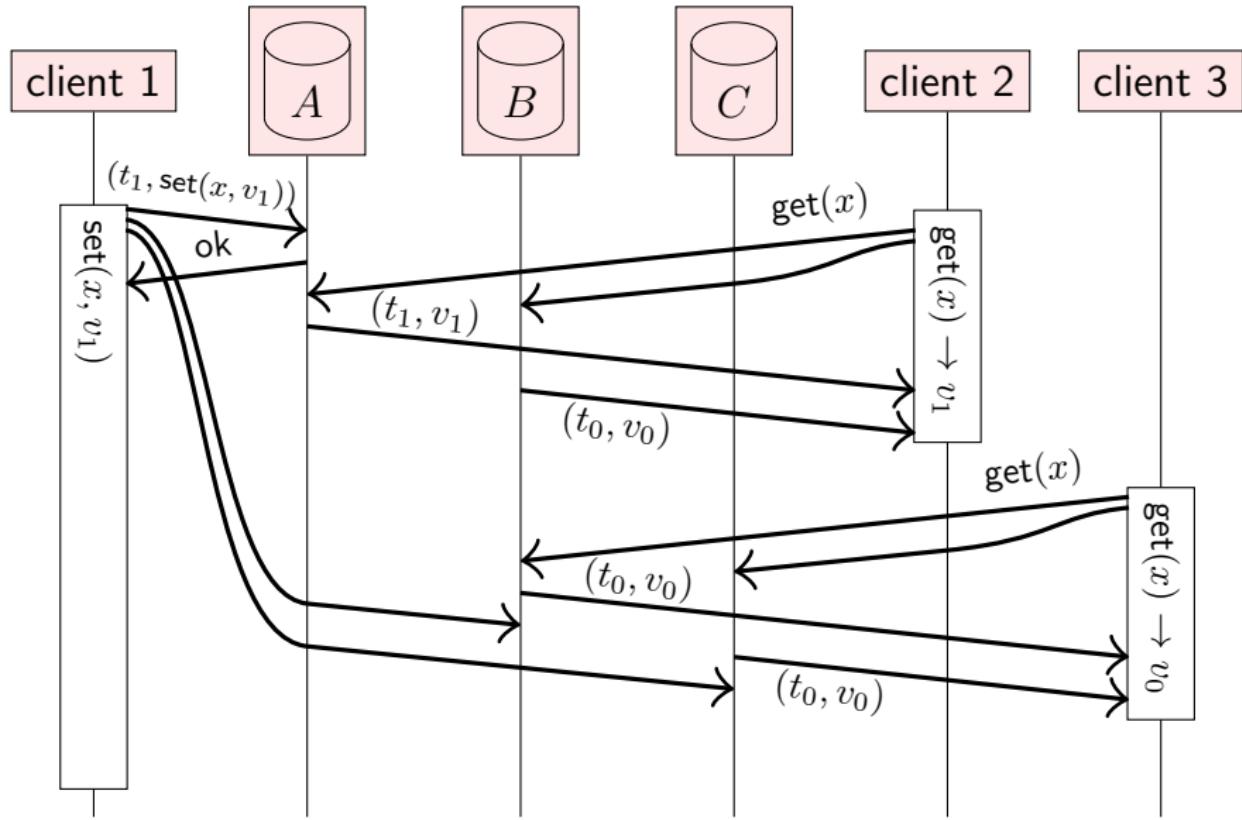
# Not linearizable, despite quorums



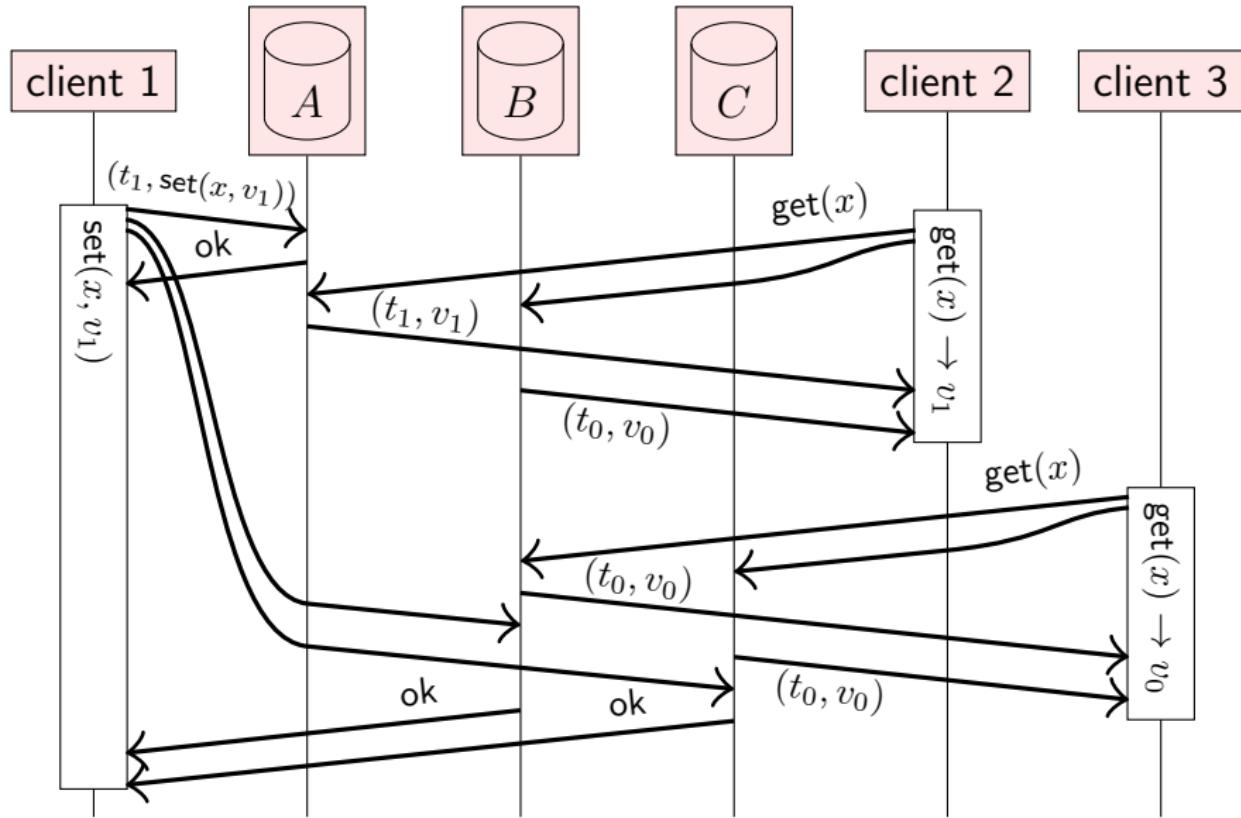
# Not linearizable, despite quorums



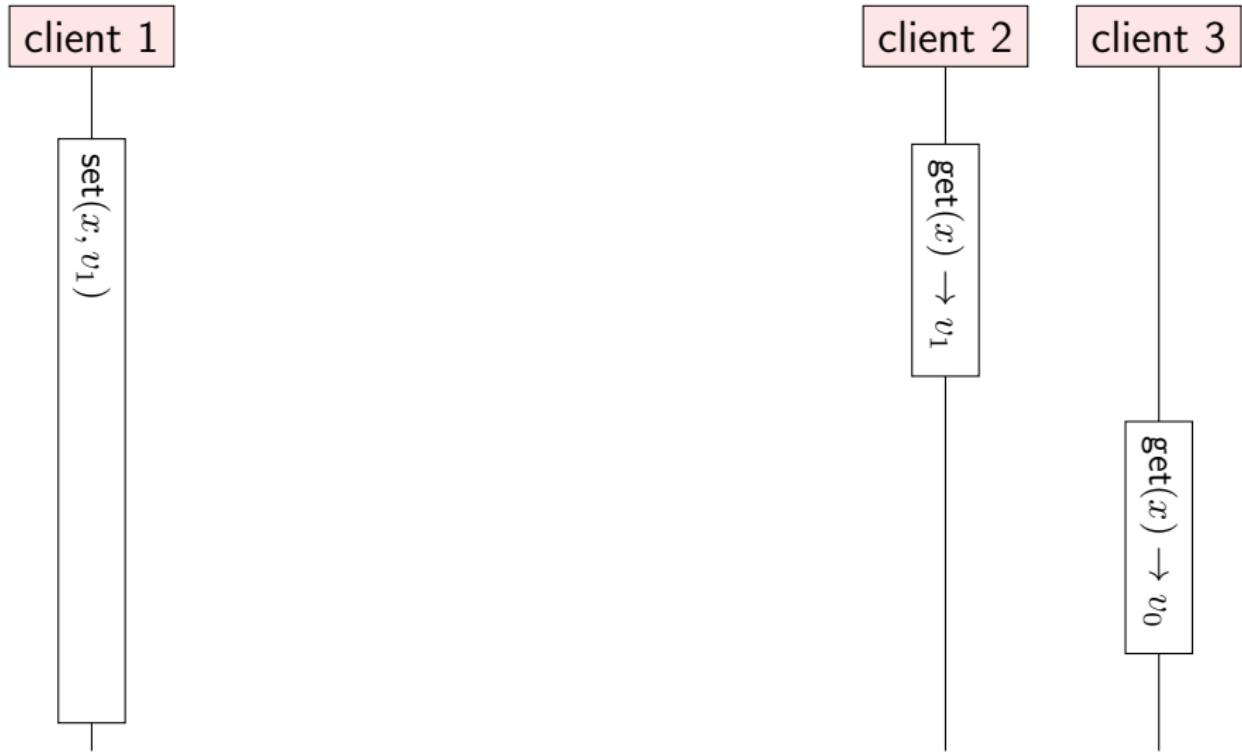
# Not linearizable, despite quorums



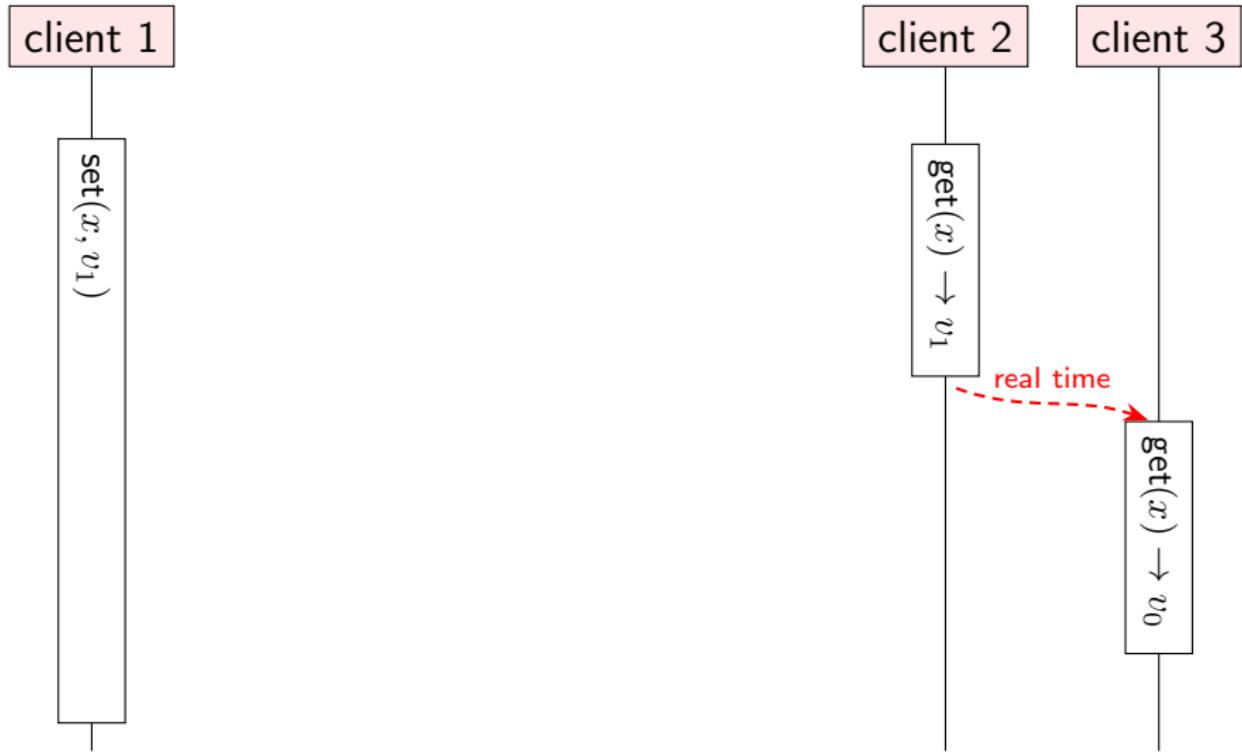
# Not linearizable, despite quorums



# Not linearizable, despite quorums



# Not linearizable, despite quorums



# Not linearizable, despite quorums



$\text{set}(x, v_1)$

- ▶ Client 2's operation finishes before client 3's operation starts
- ▶ Linearizability therefore requires client 3's operation to observe a state no older than client 2's operation
- ▶ This example violates linearizability because  $v_0$  is older than  $v_1$



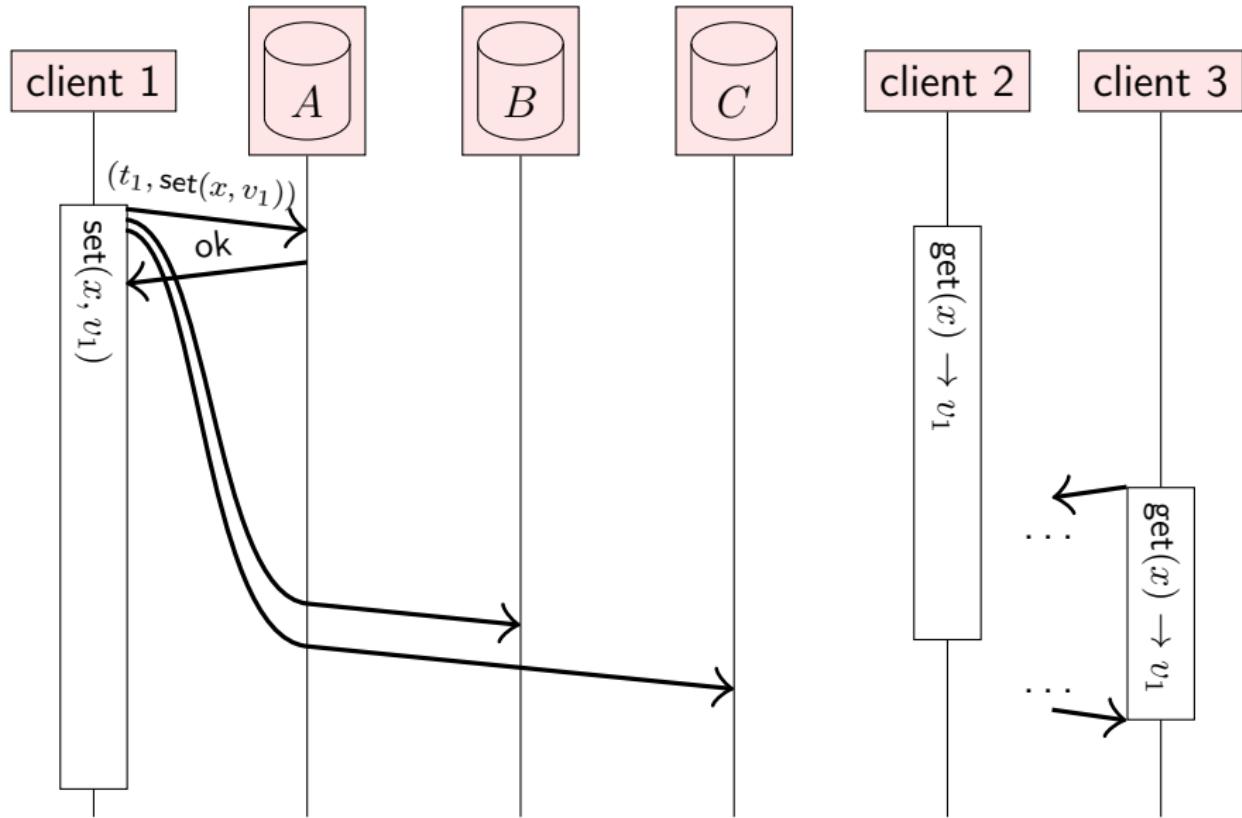
$\text{get}(x) \rightarrow v_1$

real time

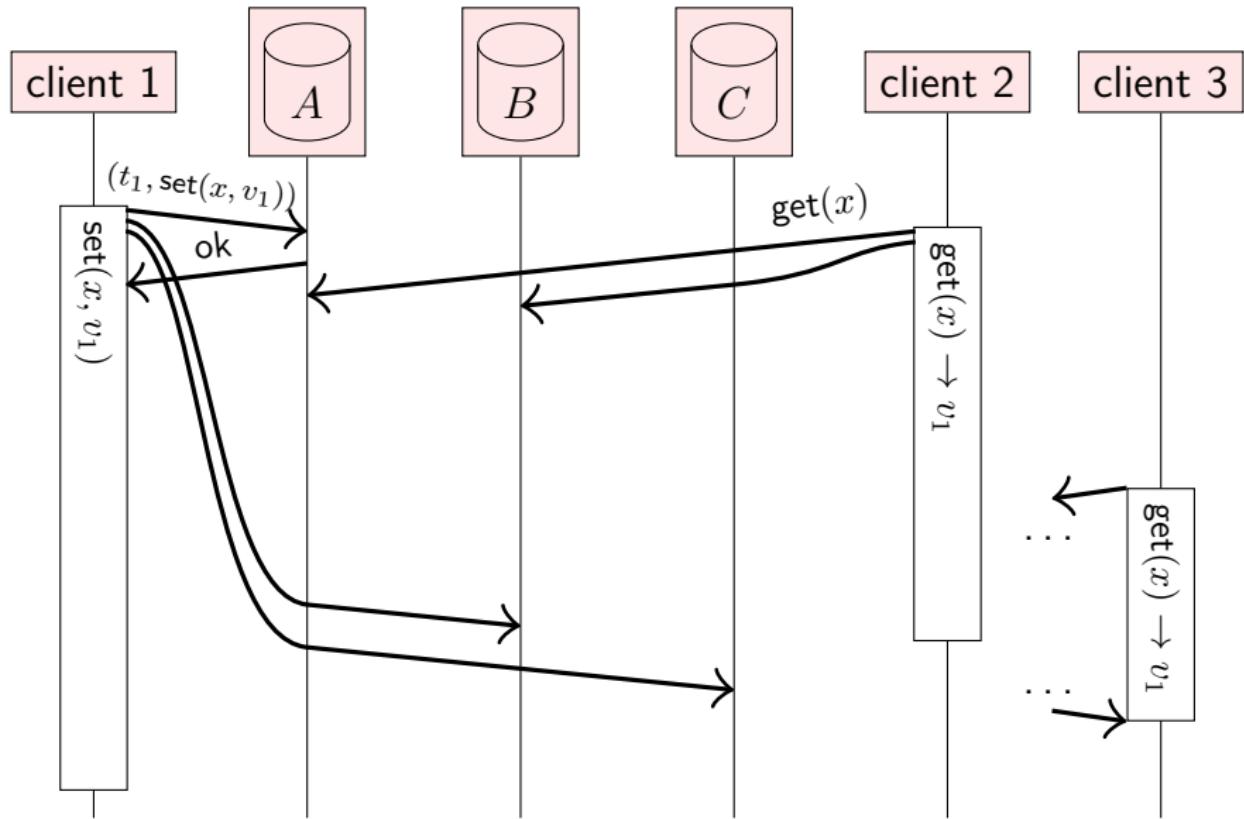


$\text{get}(x) \rightarrow v_0$

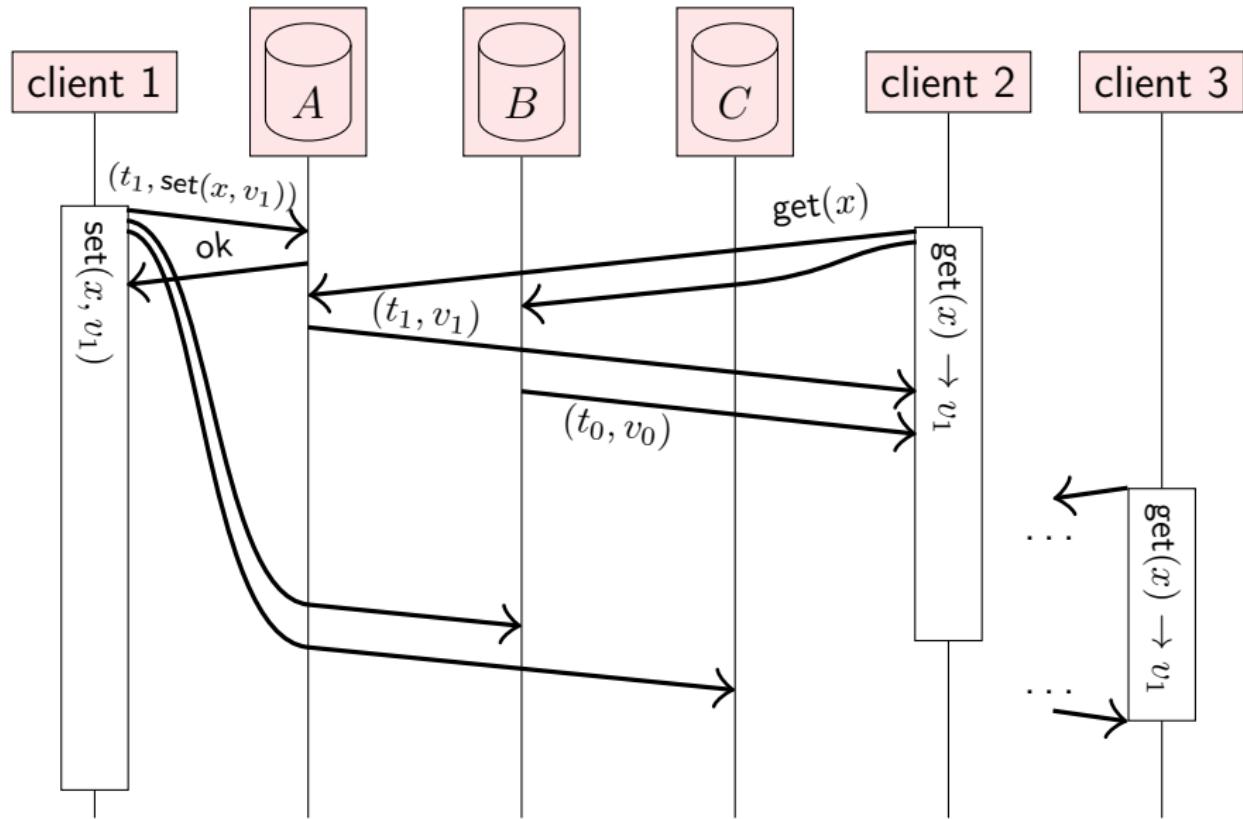
# ABD: Linearizable quorum reads/writes



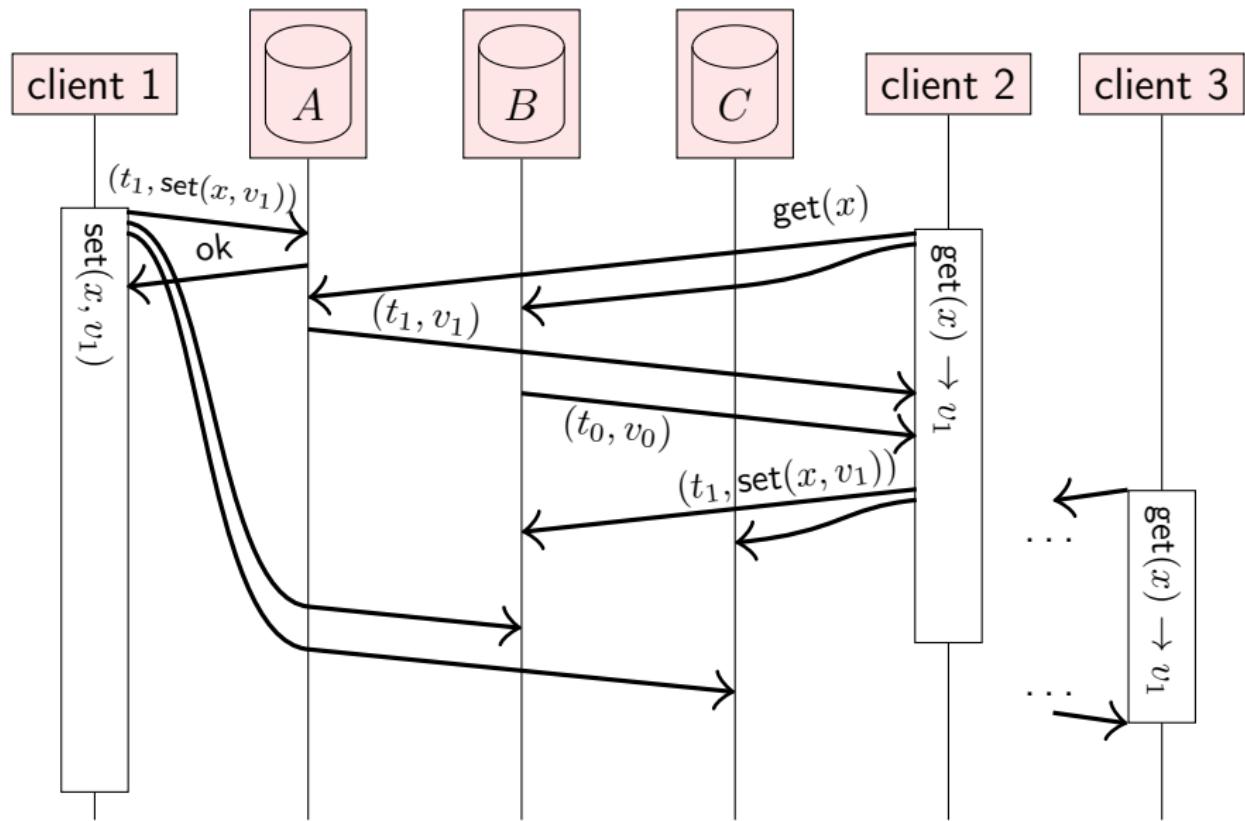
# ABD: Linearizable quorum reads/writes



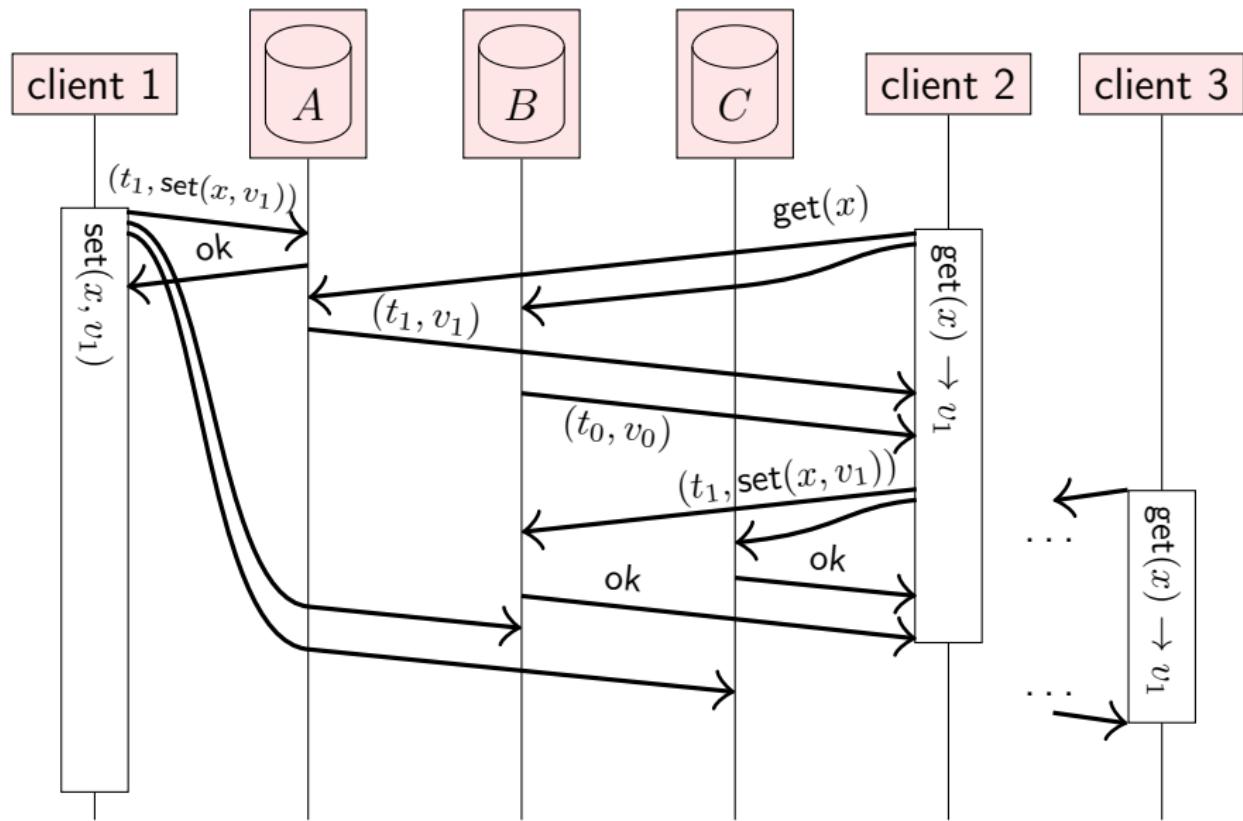
# ABD: Linearizable quorum reads/writes



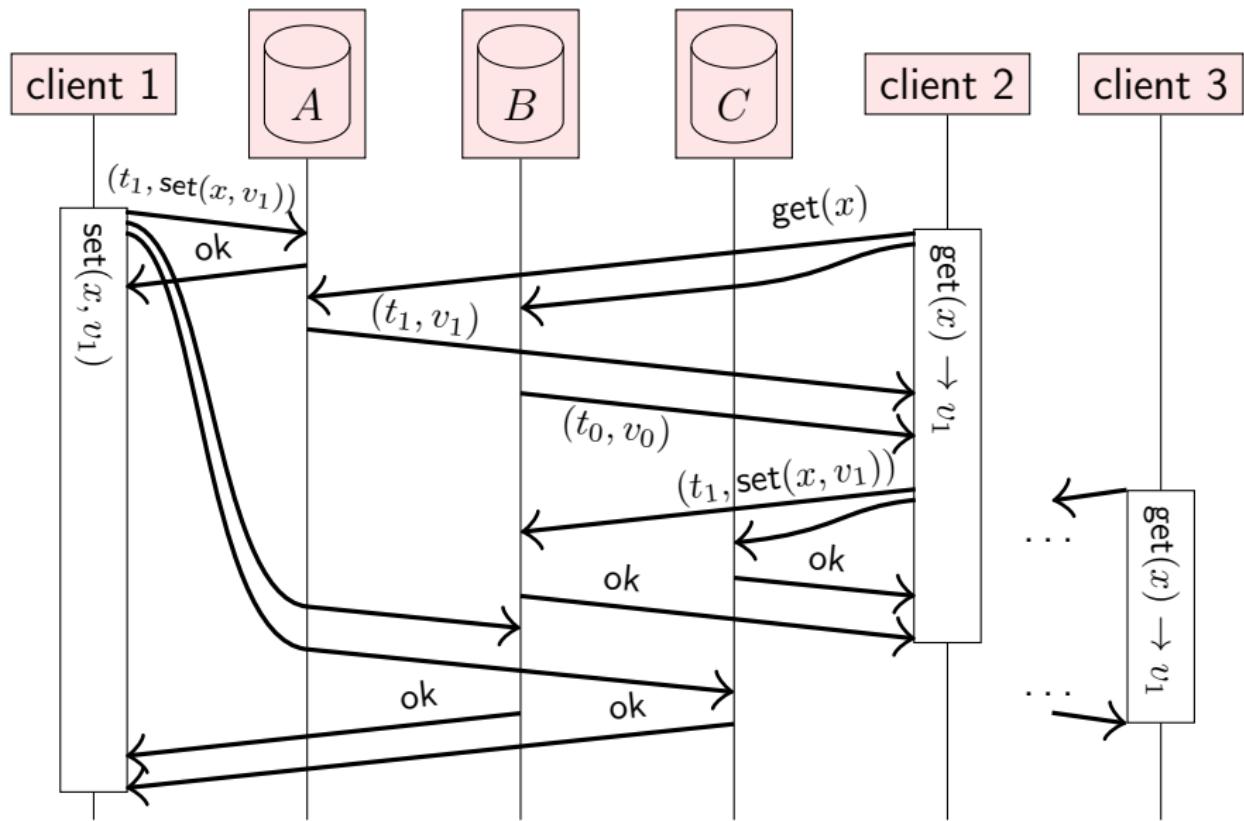
# ABD: Linearizable quorum reads/writes



# ABD: Linearizable quorum reads/writes



# ABD: Linearizable quorum reads/writes



# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value

# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value
- ▶ Multiple concurrent writes may overwrite each other

# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value
- ▶ Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- ▶ CAS( $x, oldValue, newValue$ ) sets  $x$  to  $newValue$  iff current value of  $x$  is  $oldValue$
- ▶ Previously discussed in shared-memory concurrency

# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value
- ▶ Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- ▶ CAS( $x, oldValue, newValue$ ) sets  $x$  to  $newValue$  iff current value of  $x$  is  $oldValue$
- ▶ Previously discussed in shared-memory concurrency
- ▶ Can we implement **linearizable** compare-and-swap in a distributed system?

# Linearizability for different operations



This ensures linearizability of get (quorum read) and set (**blind write** to quorum)

- ▶ When an operation finishes, the value read/written is stored on a quorum of replicas
- ▶ Every subsequent quorum operation will see that value
- ▶ Multiple concurrent writes may overwrite each other

What about an atomic **compare-and-swap** operation?

- ▶ CAS( $x, oldValue, newValue$ ) sets  $x$  to  $newValue$  iff current value of  $x$  is  $oldValue$
- ▶ Previously discussed in shared-memory concurrency
- ▶ Can we implement **linearizable** compare-and-swap in a distributed system?
- ▶ **Yes:** total order broadcast to the rescue again!

# Linearizable compare-and-swap (CAS)



**on** request to perform  $\text{get}(x)$  **do**

total order broadcast ( $\text{get}, x$ ) and wait for delivery

**end on**

**on** request to perform  $\text{CAS}(x, old, new)$  **do**

total order broadcast ( $\text{CAS}, x, old, new$ ) and wait for delivery

**end on**

**on** delivering ( $\text{get}, x$ ) by total order broadcast **do**

**return**  $\text{localState}[x]$  as result of operation  $\text{get}(x)$

**end on**

**on** delivering ( $\text{CAS}, x, old, new$ ) by total order broadcast **do**

*success* := false

**if**  $\text{localState}[x] = old$  **then**

$\text{localState}[x] := new$ ; *success* := true

**end if**

**return** *success* as result of operation  $\text{CAS}(x, old, new)$

**end on**

# Consensus and total order broadcast

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

# Fault-tolerant total order broadcast



Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- ▶ **Manual failover:** a human operator chooses a new leader, and reconfigures each node to use new leader  
Used in many databases! Fine for planned maintenance.

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- ▶ **Manual failover:** a human operator chooses a new leader, and reconfigures each node to use new leader
  - Used in many databases! Fine for planned maintenance.
  - Unplanned outage? Humans are slow, may take a long time until system recovers...

Total order broadcast is very useful for state machine replication.

Can implement total order broadcast by sending all messages via a single **leader**.

Problem: what if leader crashes/becomes unavailable?

- ▶ **Manual failover:** a human operator chooses a new leader, and reconfigures each node to use new leader
  - Used in many databases! Fine for planned maintenance.
  - Unplanned outage? Humans are slow, may take a long time until system recovers...
- ▶ Can we **automatically choose a new leader?**

# Consensus and total order broadcast



- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**

# Consensus and total order broadcast

- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**

# Consensus and total order broadcast



- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**
- ▶ Once one node **decides** on a certain message order, all nodes will decide the same order

- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**
- ▶ Once one node **decides** on a certain message order, all nodes will decide the same order
- ▶ Consensus and total order broadcast are formally equivalent

- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**
- ▶ Once one node **decides** on a certain message order, all nodes will decide the same order
- ▶ Consensus and total order broadcast are formally equivalent

Common consensus algorithms:

- ▶ **Paxos**: single-value consensus  
**Multi-Paxos**: generalisation to total order broadcast

- ▶ Traditional formulation of consensus: several nodes want to come to **agreement** about a single **value**
- ▶ In context of total order broadcast: this value is the **next message to deliver**
- ▶ Once one node **decides** on a certain message order, all nodes will decide the same order
- ▶ Consensus and total order broadcast are formally equivalent

Common consensus algorithms:

- ▶ **Paxos**: single-value consensus  
**Multi-Paxos**: generalisation to total order broadcast
- ▶ **Raft, Viewstamped Replication, Zab**:  
FIFO-total order broadcast by default

# Atomic commit versus consensus



Atomic commit	Consensus
Every node votes whether to commit or abort	One or more nodes propose a value

# Atomic commit versus consensus



Atomic commit	Consensus
Every node votes whether to commit or abort	One or more nodes propose a value
Must commit if all nodes vote to commit; must abort if $\geq 1$ nodes vote to abort	Any one of the proposed values is decided

Atomic commit	Consensus
Every node votes whether to commit or abort	One or more nodes propose a value
Must commit if all nodes vote to commit; must abort if $\geq 1$ nodes vote to abort	Any one of the proposed values is decided
Must abort if a participating node crashes	Crashed nodes can be tolerated, as long as a quorum is working

# How total order broadcast should behave



Properties of total order broadcast fall into two categories:

- ▶ **Safety:** “*nothing bad happens*”

- ▶ **Liveness:** “*something good eventually happens*”

# How total order broadcast should behave



Properties of total order broadcast fall into two categories:

► **Safety:** “*nothing bad happens*”

1. Let  $N_1$  and  $N_2$  be two nodes that each deliver two messages  $m_1$  and  $m_2$ , and assume that  $N_1$  delivers  $m_1$  before  $m_2$ . Then  $N_2$  also delivers  $m_1$  before  $m_2$ .

► **Liveness:** “*something good eventually happens*”

# How total order broadcast should behave



Properties of total order broadcast fall into two categories:

► **Safety:** “*nothing bad happens*”

1. Let  $N_1$  and  $N_2$  be two nodes that each deliver two messages  $m_1$  and  $m_2$ , and assume that  $N_1$  delivers  $m_1$  before  $m_2$ . Then  $N_2$  also delivers  $m_1$  before  $m_2$ .
2. If some node delivers a message  $m$ , then  $m$  was previously broadcast by some node.

► **Liveness:** “*something good eventually happens*”

# How total order broadcast should behave



Properties of total order broadcast fall into two categories:

► **Safety:** “*nothing bad happens*”

1. Let  $N_1$  and  $N_2$  be two nodes that each deliver two messages  $m_1$  and  $m_2$ , and assume that  $N_1$  delivers  $m_1$  before  $m_2$ . Then  $N_2$  also delivers  $m_1$  before  $m_2$ .
2. If some node delivers a message  $m$ , then  $m$  was previously broadcast by some node.
3. A node does not deliver the same message more than once.

► **Liveness:** “*something good eventually happens*”

Properties of total order broadcast fall into two categories:

► **Safety:** “*nothing bad happens*”

1. Let  $N_1$  and  $N_2$  be two nodes that each deliver two messages  $m_1$  and  $m_2$ , and assume that  $N_1$  delivers  $m_1$  before  $m_2$ . Then  $N_2$  also delivers  $m_1$  before  $m_2$ .
2. If some node delivers a message  $m$ , then  $m$  was previously broadcast by some node.
3. A node does not deliver the same message more than once.

► **Liveness:** “*something good eventually happens*”

4. If a node broadcasts a message  $m$  and does not crash, then eventually that node delivers  $m$ .

Properties of total order broadcast fall into two categories:

► **Safety:** “*nothing bad happens*”

1. Let  $N_1$  and  $N_2$  be two nodes that each deliver two messages  $m_1$  and  $m_2$ , and assume that  $N_1$  delivers  $m_1$  before  $m_2$ . Then  $N_2$  also delivers  $m_1$  before  $m_2$ .
2. If some node delivers a message  $m$ , then  $m$  was previously broadcast by some node.
3. A node does not deliver the same message more than once.

► **Liveness:** “*something good eventually happens*”

4. If a node broadcasts a message  $m$  and does not crash, then eventually that node delivers  $m$ .
5. If one node delivers a message  $m$ , then every other node that does not crash eventually delivers  $m$ .

# Consensus system models



Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- ▶ **FLP result** (Fischer, Lynch, Paterson):

There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- ▶ **FLP result** (Fischer, Lynch, Paterson):  
There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
- ▶ Paxos, Raft, etc. use clocks only used for timeouts/failure detector to ensure liveness. Safety does not depend on timing.

Paxos, Raft, etc. assume a **partially synchronous, crash-recovery** system model.

Why not asynchronous?

- ▶ **FLP result** (Fischer, Lynch, Paterson):  
There is no deterministic consensus algorithm that is guaranteed to terminate in an asynchronous crash-stop system model.
- ▶ Paxos, Raft, etc. use clocks only used for timeouts/failure detector to ensure liveness. Safety does not depend on timing.

There are also consensus algorithms for a partially synchronous **Byzantine** system model (used in blockchains)

# Leader election



Multi-Paxos, Raft, etc. use a leader to sequence messages.

# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.

# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.
- ▶ Prevent **two leaders at the same time** ("split-brain")!

# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.
- ▶ Prevent **two leaders at the same time** ("split-brain")!

Ensure  $\leq 1$  leader per **term**:

- ▶ Term is incremented every time a leader election is started

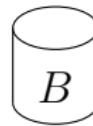
# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.
- ▶ Prevent **two leaders at the same time** ("split-brain")!

Ensure  $\leq 1$  leader per **term**:

- ▶ Term is incremented every time a leader election is started
- ▶ A node can only **vote once** per term
- ▶ Require a **quorum** of nodes to elect a leader in a term



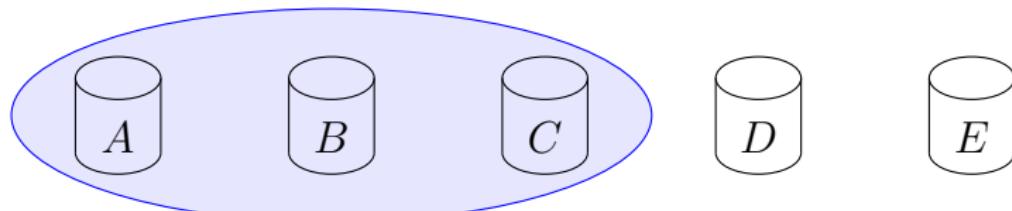
# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.
- ▶ Prevent **two leaders at the same time** ("split-brain")!

Ensure  $\leq 1$  leader per **term**:

- ▶ Term is incremented every time a leader election is started
- ▶ A node can only **vote once** per term
- ▶ Require a **quorum** of nodes to elect a leader in a term



elects a leader

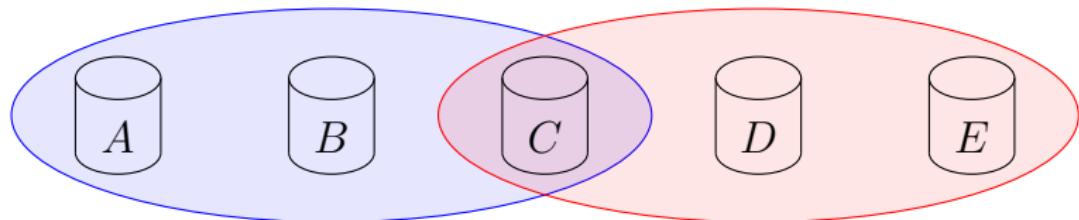
# Leader election

Multi-Paxos, Raft, etc. use a leader to sequence messages.

- ▶ Use a **failure detector** (timeout) to determine suspected crash or unavailability of leader.
- ▶ On suspected leader crash, **elect a new one**.
- ▶ Prevent **two leaders at the same time** ("split-brain")!

Ensure  $\leq 1$  leader per **term**:

- ▶ Term is incremented every time a leader election is started
- ▶ A node can only **vote once** per term
- ▶ Require a **quorum** of nodes to elect a leader in a term



elects a leader

cannot elect a different leader  
because *C* already voted

# Can we guarantee there is only one leader?



Can guarantee unique leader **per term**.

# Can we guarantee there is only one leader?



Can guarantee unique leader **per term**.

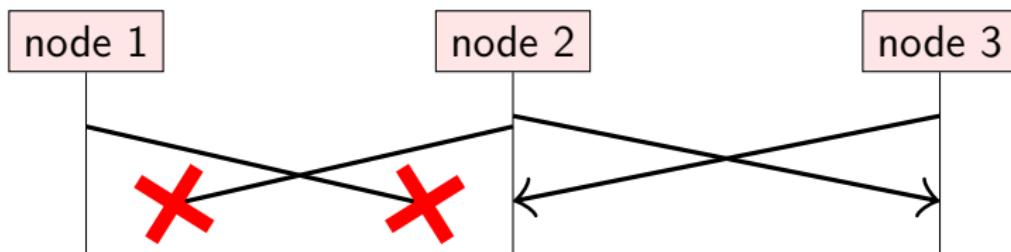
**Cannot** prevent having multiple leaders from different terms.

# Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

**Cannot** prevent having multiple leaders from different terms.

Example: node 1 is leader in term  $t$ , but due to a network partition it can no longer communicate with nodes 2 and 3:



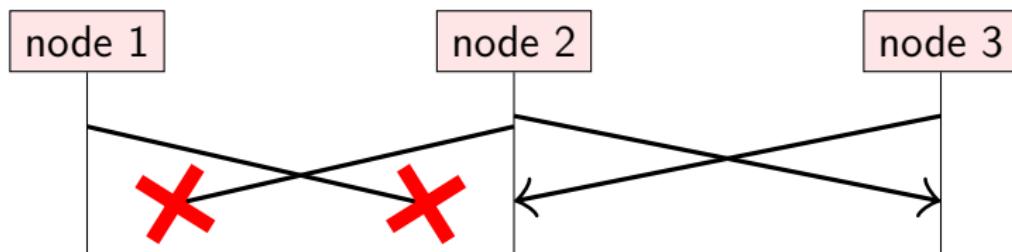
Nodes 2 and 3 may elect a new leader in term  $t + 1$ .

# Can we guarantee there is only one leader?

Can guarantee unique leader **per term**.

**Cannot** prevent having multiple leaders from different terms.

Example: node 1 is leader in term  $t$ , but due to a network partition it can no longer communicate with nodes 2 and 3:

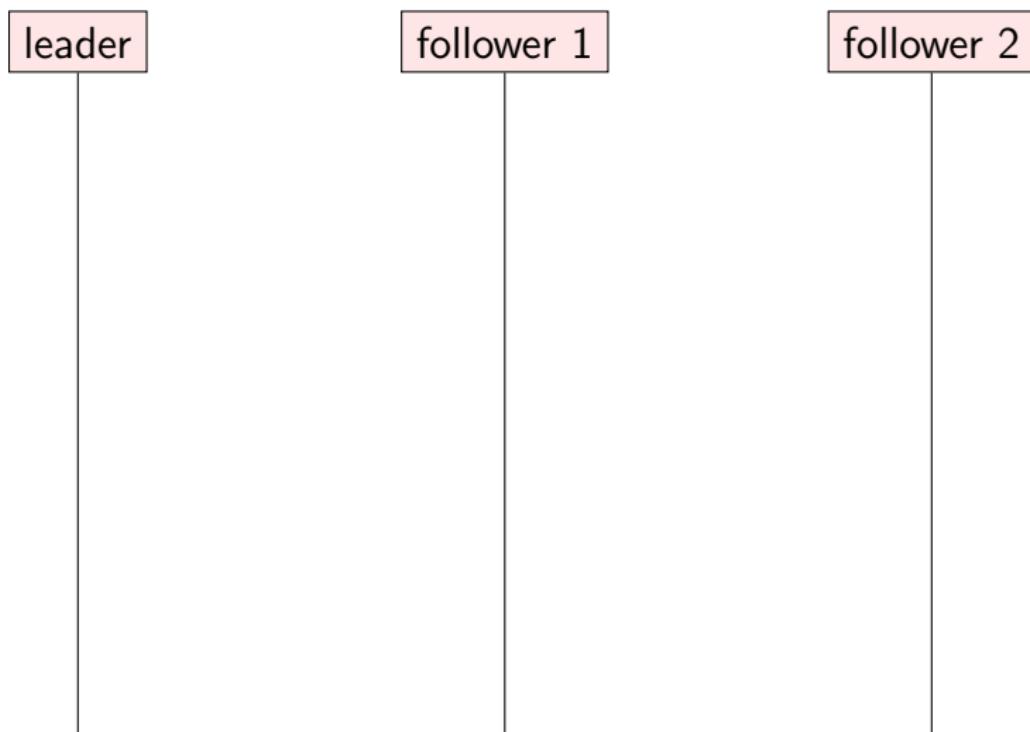


Nodes 2 and 3 may elect a new leader in term  $t + 1$ .

Node 1 may not even know that a new leader has been elected!

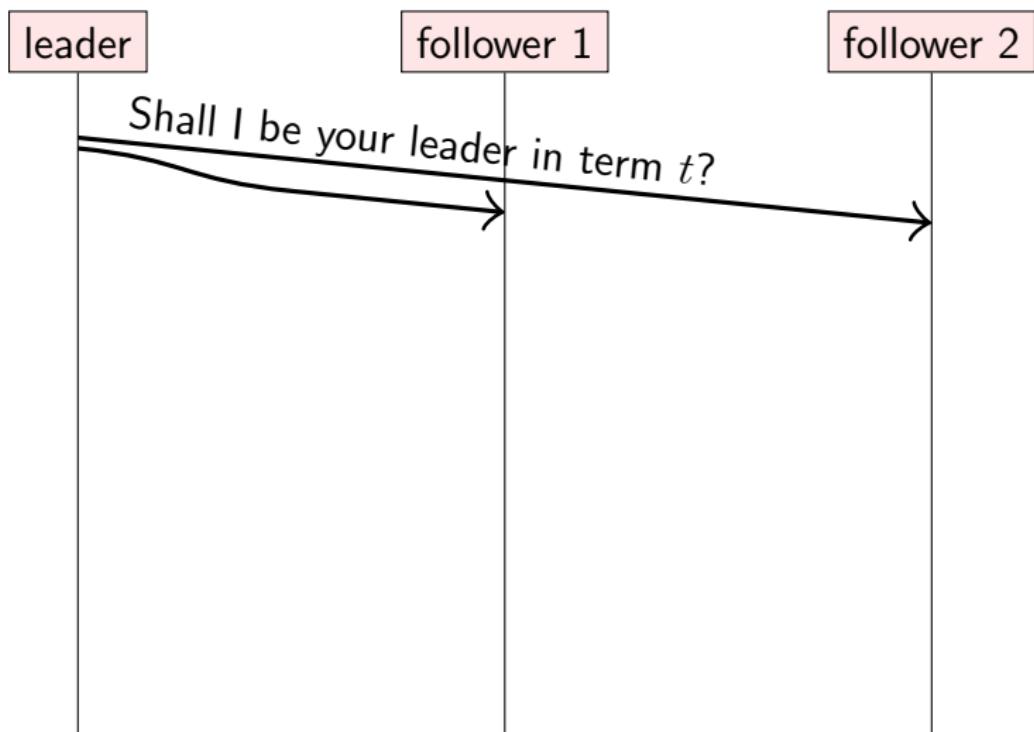
# Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



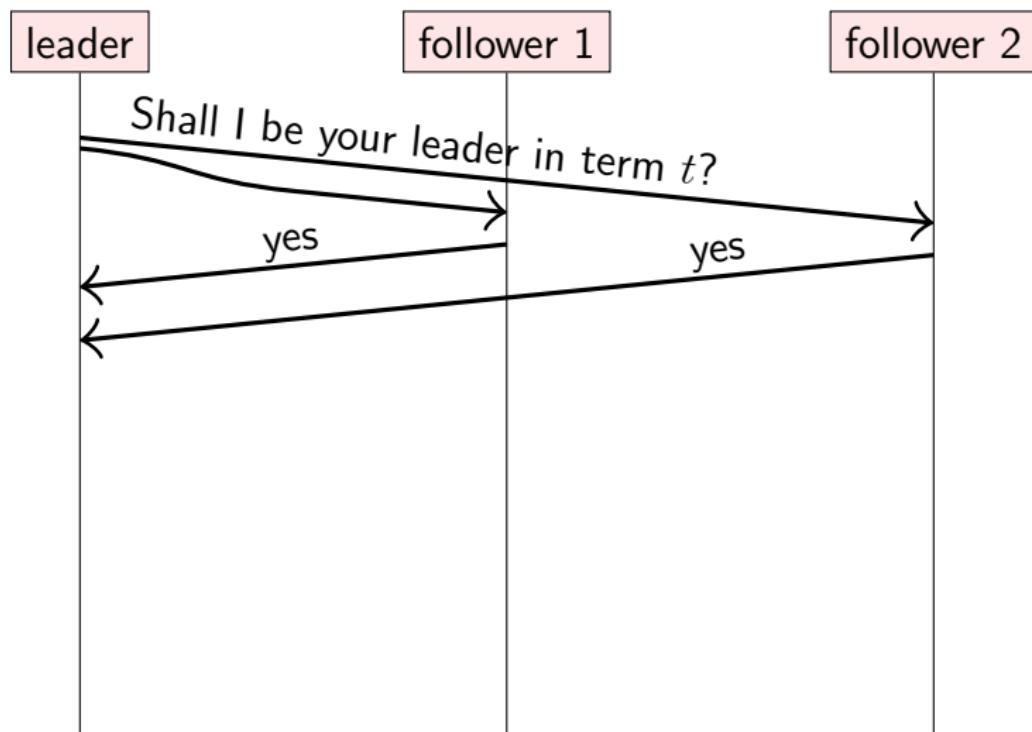
# Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



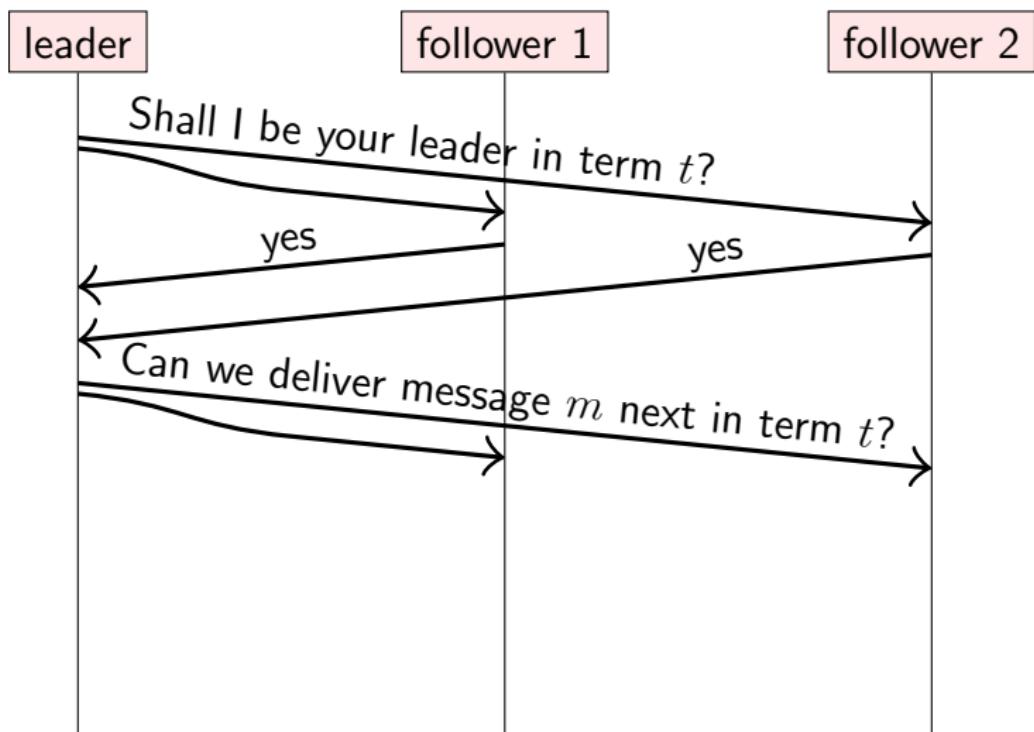
# Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



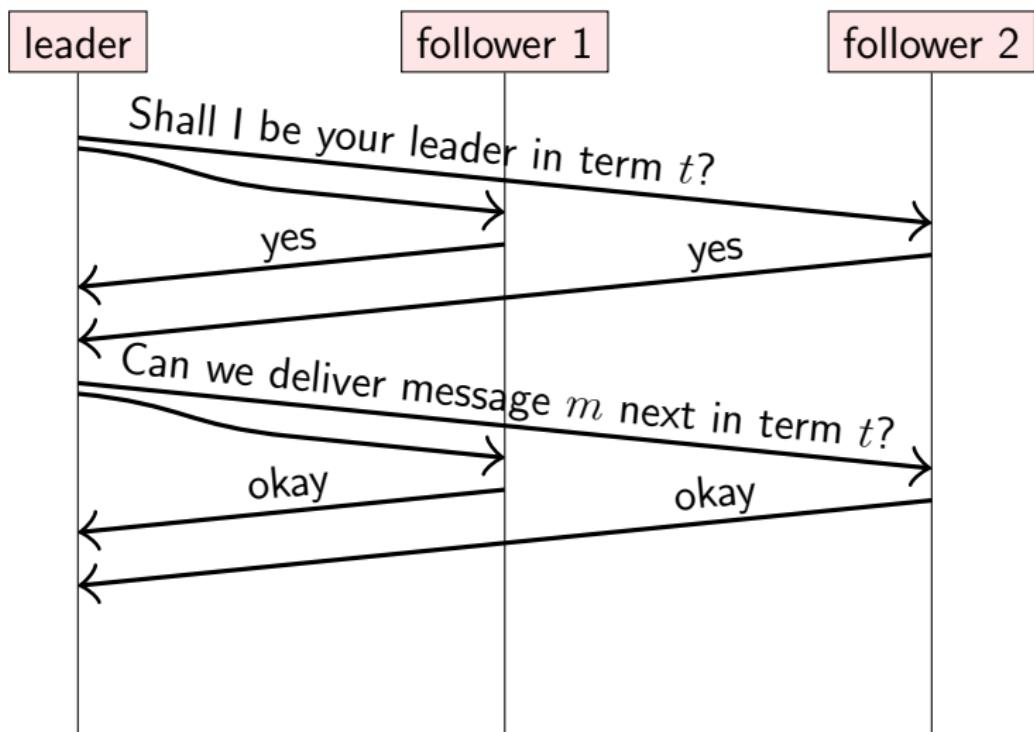
# Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



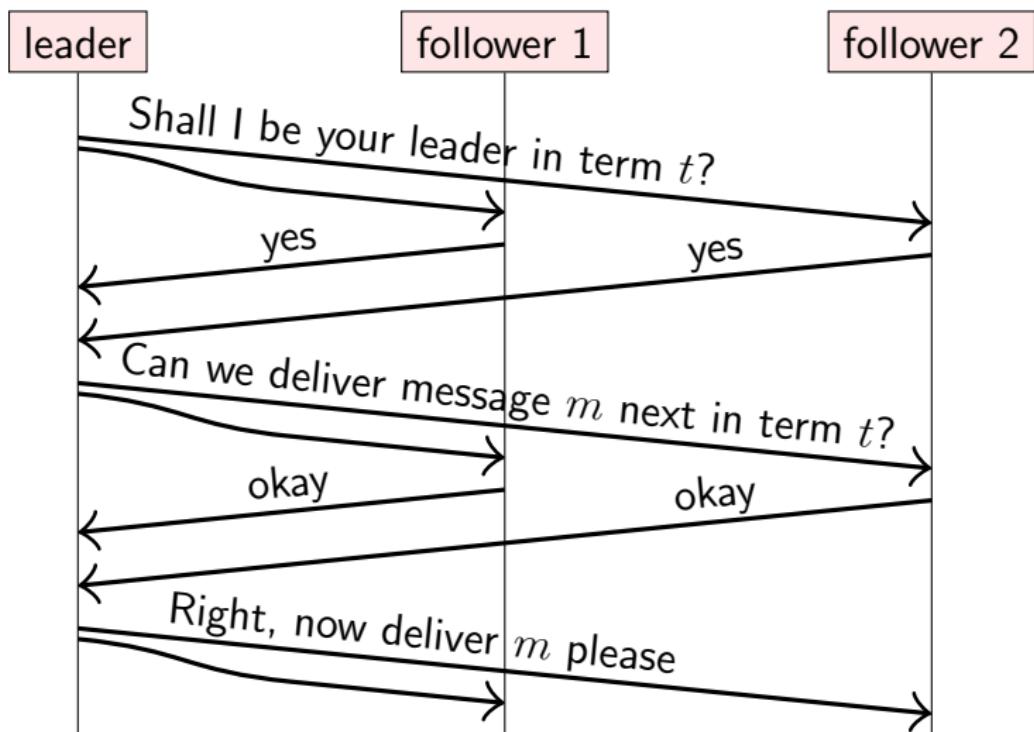
# Checking if a leader has been voted out

For every decision (message to deliver), the leader must first get acknowledgements from a quorum.

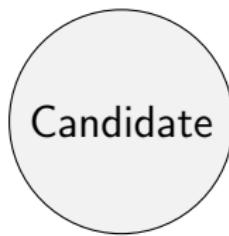
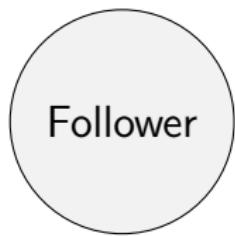


# Checking if a leader has been voted out

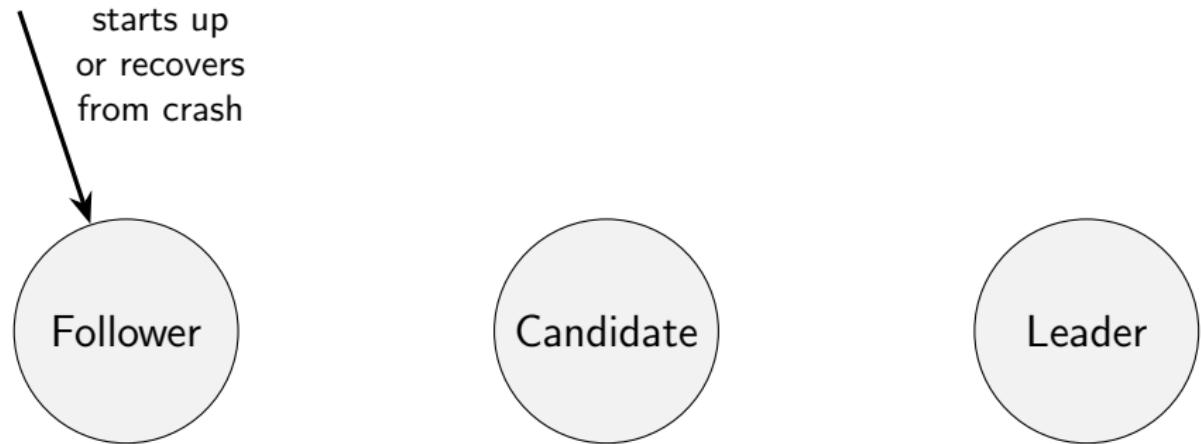
For every decision (message to deliver), the leader must first get acknowledgements from a quorum.



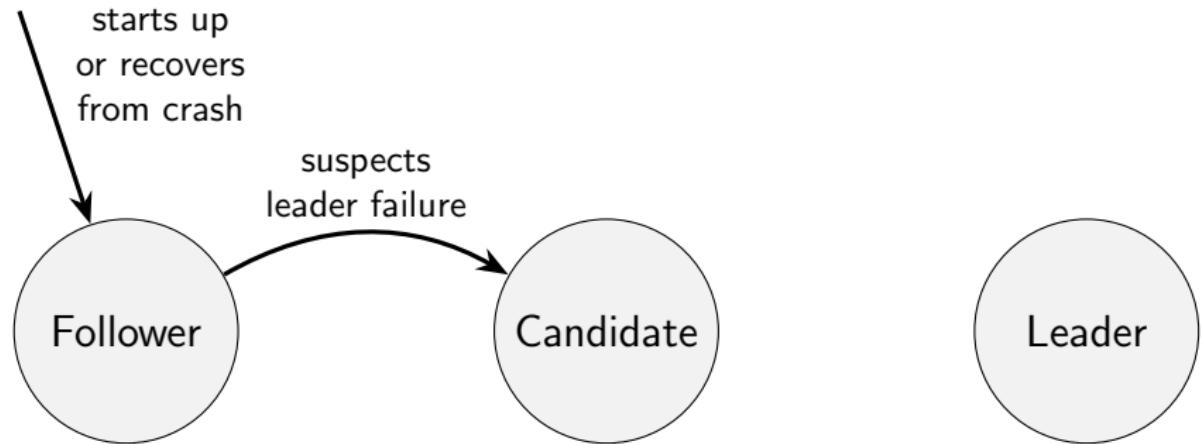
# Node state transitions in Raft



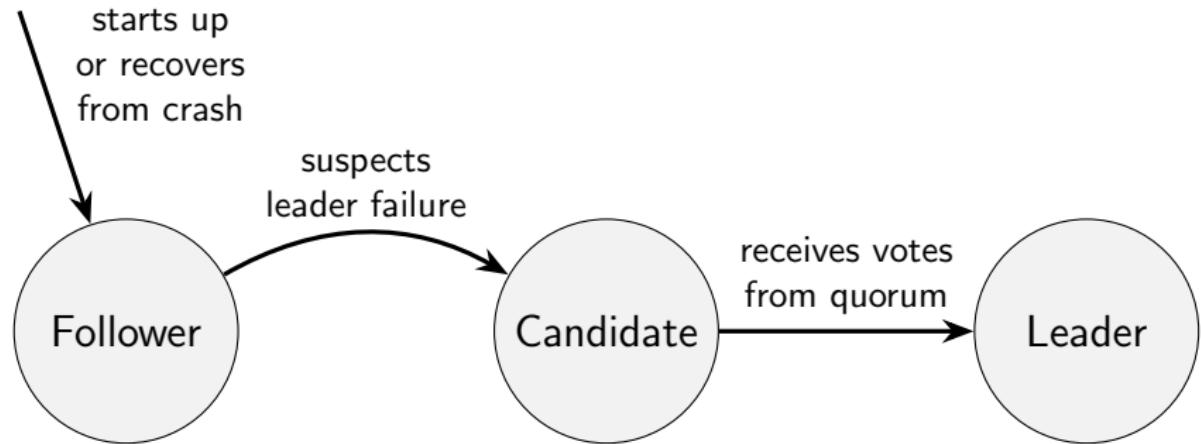
# Node state transitions in Raft



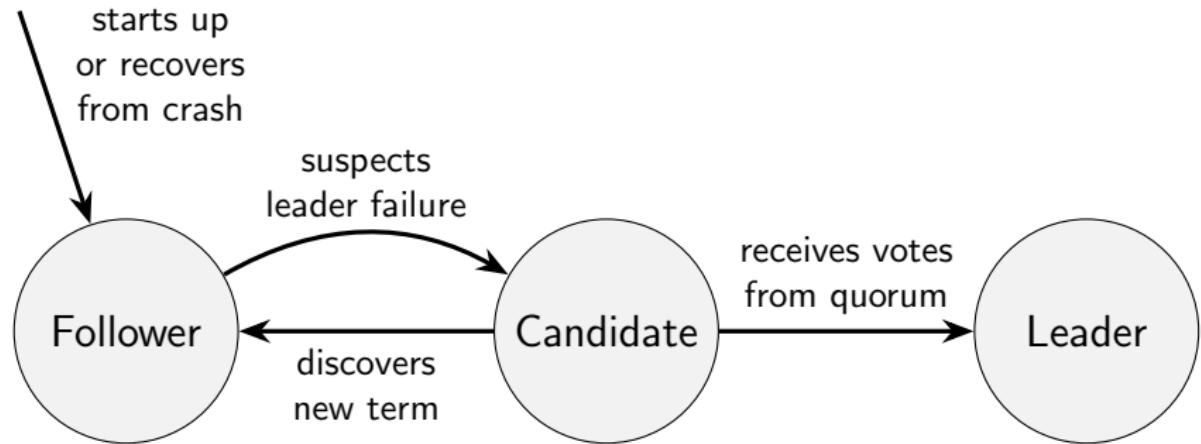
# Node state transitions in Raft



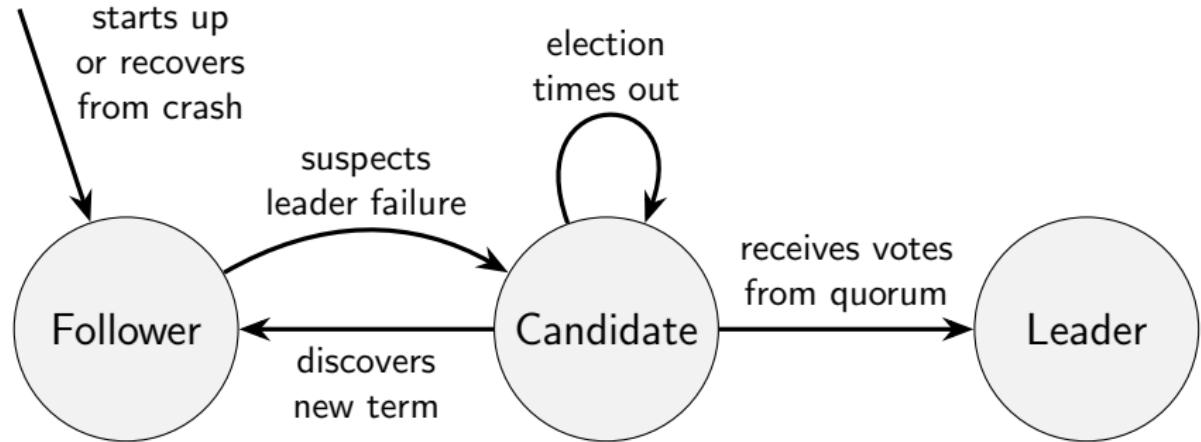
# Node state transitions in Raft



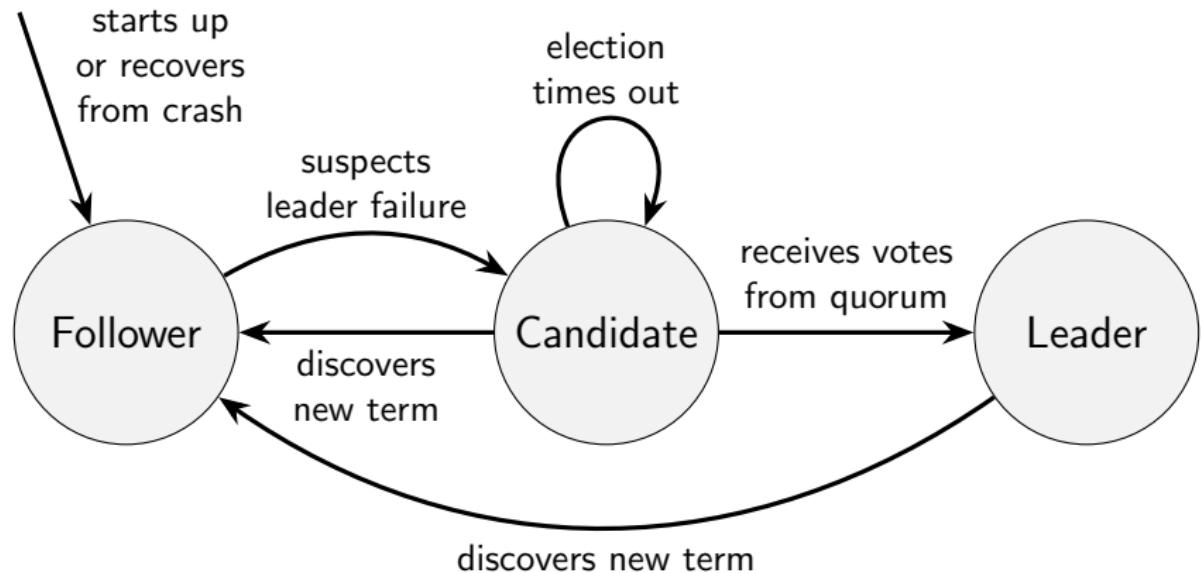
# Node state transitions in Raft



# Node state transitions in Raft



# Node state transitions in Raft



# Raft (1/9): initialisation

**on** initialisation **do**

*currentTerm* := 0; *votedFor* := null

*log* := ⟨⟩; *commitLength* := 0

*currentRole* := follower; *currentLeader* := null

*votesReceived* := {}; *sentLength* := ⟨⟩; *ackedLength* := ⟨⟩

**end on**

**on** recovery from crash **do**

*currentRole* := follower; *currentLeader* := null

*votesReceived* := {}; *sentLength* := ⟨⟩; *ackedLength* := ⟨⟩

**end on**

**on** node *nodeId* suspects leader has failed, or on election timeout **do**

*currentTerm* := *currentTerm* + 1; *currentRole* := candidate

*votedFor* := *nodeId*; *votesReceived* := {*nodeId*}; *lastTerm* := 0

**if** *log.length* > 0 **then** *lastTerm* := *log*[*log.length* - 1].term; **end if**

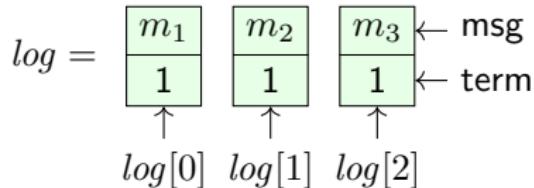
*msg* := (VoteRequest, *nodeId*, *currentTerm*, *log.length*, *lastTerm*)

**for each** *node* ∈ *nodes*: **send** *msg* to *node*

start election timer

**end on**

# Raft (1/9): initialisation



**on** initialisation **do**

*currentTerm* := 0; *votedFor* := null

*log* :=  $\langle \rangle$ ; *commitLength* := 0

*currentRole* := follower; *currentLeader* := null

*votesReceived* :=  $\{\}$ ; *sentLength* :=  $\langle \rangle$ ; *ackedLength* :=  $\langle \rangle$

**end on**

**on** recovery from crash **do**

*currentRole* := follower; *currentLeader* := null

*votesReceived* :=  $\{\}$ ; *sentLength* :=  $\langle \rangle$ ; *ackedLength* :=  $\langle \rangle$

**end on**

**on** node *nodeId* suspects leader has failed, or on election timeout **do**

*currentTerm* := *currentTerm* + 1; *currentRole* := candidate

*votedFor* := *nodeId*; *votesReceived* :=  $\{nodeId\}$ ; *lastTerm* := 0

**if** *log.length* > 0 **then** *lastTerm* := *log[log.length - 1].term*; **end if**

*msg* := (VoteRequest, *nodeId*, *currentTerm*, *log.length*, *lastTerm*)

**for each** *node*  $\in$  *nodes*: **send** *msg* **to** *node*

start election timer

**end on**

# Raft (2/9): voting on a new leader

```
on receiving (VoteRequest, cId, cTerm, cLogLength, cLogTerm)
    at node nodeId do
        if cTerm > currentTerm then
            currentTerm := cTerm; currentRole := follower
            votedFor := null
        end if
        lastTerm := 0
        if log.length > 0 then lastTerm := log[log.length - 1].term; end if
        logOk := (cLogTerm > lastTerm) ∨
                  (cLogTerm = lastTerm ∧ cLogLength ≥ log.length)

        if cTerm = currentTerm ∧ logOk ∧ votedFor ∈ {cId, null} then
            votedFor := cId
            send (VoteResponse, nodeId, currentTerm, true) to node cId
        else
            send (VoteResponse, nodeId, currentTerm, false) to node cId
        end if
    end on
```

## Raft (2/9): voting on a new leader

*c* for candidate



```
on receiving (VoteRequest, cId, cTerm, cLogLength, cLogTerm)
    at node nodeId do
        if cTerm > currentTerm then
            currentTerm := cTerm; currentRole := follower
            votedFor := null
        end if
        lastTerm := 0
        if log.length > 0 then lastTerm := log[log.length - 1].term; end if
        logOk := (cLogTerm > lastTerm) ∨
                  (cLogTerm = lastTerm ∧ cLogLength ≥ log.length)

        if cTerm = currentTerm ∧ logOk ∧ votedFor ∈ {cId, null} then
            votedFor := cId
            send (VoteResponse, nodeId, currentTerm, true) to node cId
        else
            send (VoteResponse, nodeId, currentTerm, false) to node cId
        end if
    end on
```

# Raft (3/9): collecting votes

```
on receiving (VoteResponse, voterId, term, granted) at nodeId do
    if currentRole = candidate ∧ term = currentTerm ∧ granted then
        votesReceived := votesReceived ∪ {voterId}
        if |votesReceived| ≥ ⌈(|nodes| + 1)/2⌉ then
            currentRole := leader; currentLeader := nodeId
            cancel election timer
            for each follower ∈ nodes \ {nodeId} do
                sentLength[follower] := log.length
                ackedLength[follower] := 0
                REPLICATELOG(nodeId, follower)
            end for
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
        cancel election timer
    end if
end on
```

# Raft (4/9): broadcasting messages

```
on request to broadcast msg at node nodeId do
    if currentRole = leader then
        append the record (msg : msg, term : currentTerm) to log
        ackedLength[nodeId] := log.length
        for each follower ∈ nodes \ {nodeId} do
            REPLICATELOG(nodeId, follower)
        end for
    else
        forward the request to currentLeader via a FIFO link
    end if
end on

periodically at node nodeId do
    if currentRole = leader then
        for each follower ∈ nodes \ {nodeId} do
            REPLICATELOG(nodeId, follower)
        end for
    end if
end do
```

# Raft (5/9): replicating to followers



Called on the leader whenever there is a new message in the log, and also periodically. If there are no new messages, *suffix* is the empty list. LogRequest messages with *suffix* =  $\langle \rangle$  serve as heartbeats, letting followers know that the leader is still alive.

```
function REPLICATELOG(leaderId, followerId)
    prefixLen := sentLength[followerId]
    suffix :=  $\langle \log[\text{prefixLen}], \log[\text{prefixLen} + 1], \dots,$ 
               $\log[\log.\text{length} - 1] \rangle$ 
    prefixTerm := 0
    if prefixLen > 0 then
        prefixTerm :=  $\log[\text{prefixLen} - 1].\text{term}$ 
    end if
    send (LogRequest, leaderId, currentTerm, prefixLen,
          prefixTerm, commitLength, suffix) to followerId
end function
```

# Raft (6/9): followers receiving messages

```
on receiving (LogRequest, leaderId, term, prefixLen, prefixTerm,  
           leaderCommit, suffix) at node nodeId do  
    if term > currentTerm then  
        currentTerm := term; votedFor := null  
        cancel election timer  
    end if  
    if term = currentTerm then  
        currentRole := follower; currentLeader := leaderId  
    end if  
    logOk := (log.length ≥ prefixLen) ∧  
              (prefixLen = 0 ∨ log[prefixLen - 1].term = prefixTerm)  
    if term = currentTerm ∧ logOk then  
        APPENDENTRIES(prefixLen, leaderCommit, suffix)  
        ack := prefixLen + suffix.length  
        send (LogResponse, nodeId, currentTerm, ack, true) to leaderId  
    else  
        send (LogResponse, nodeId, currentTerm, 0, false) to leaderId  
    end if  
end on
```

# Raft (7/9): updating followers' logs

```
function APPENDENTRIES(prefixLen, leaderCommit, suffix)
    if suffix.length > 0  $\wedge$  log.length > prefixLen then
        index := min(log.length, prefixLen + suffix.length) - 1
        if log[index].term  $\neq$  suffix[index - prefixLen].term then
            log := ⟨log[0], log[1], ..., log[prefixLen - 1]⟩
        end if
    end if
    if prefixLen + suffix.length > log.length then
        for i := log.length - prefixLen to suffix.length - 1 do
            append suffix[i] to log
        end for
    end if
    if leaderCommit > commitLength then
        for i := commitLength to leaderCommit - 1 do
            deliver log[i].msg to the application
        end for
        commitLength := leaderCommit
    end if
end function
```

# Raft (8/9): leader receiving acks

```
on receiving (LogResponse, follower, term, ack, success) at nodeId do
    if term = currentTerm ∧ currentRole = leader then
        if success = true ∧ ack ≥ ackedLength[follower] then
            sentLength[follower] := ack
            ackedLength[follower] := ack
            COMMITLOGENTRIES()
        else if sentLength[follower] > 0 then
            sentLength[follower] := sentLength[follower] - 1
            REPLICATELOG(nodeId, follower)
        end if
    else if term > currentTerm then
        currentTerm := term
        currentRole := follower
        votedFor := null
        cancel election timer
    end if
end on
```

# Raft (9/9): leader committing log entries



Any log entries that have been acknowledged by a quorum of nodes are ready to be committed by the leader. When a log entry is committed, its message is delivered to the application.

```
define acks(length) = |{n ∈ nodes | ackedLength[n] ≥ length}|  
  
function COMMITLOGENTRIES  
    minAcks := ⌈(|nodes| + 1)/2⌉  
    ready := {len ∈ {1, …, log.length} | acks(len) ≥ minAcks}  
    if ready ≠ {} ∧ max(ready) > commitLength ∧  
        log[max(ready) - 1].term = currentTerm then  
            for i := commitLength to max(ready) - 1 do  
                deliver log[i].msg to the application  
            end for  
            commitLength := max(ready)  
    end if  
end function
```

A real implementation would need to do more:

- ▶ Efficient **log reconciliation** when  $\neg logOk$
- ▶ Allow **reconfiguration**:  
allow administrators to add or remove nodes, adjusting quorums accordingly
- ▶ Better **throughput**:  
avoid having to do everything through the leader?  
(some Paxos variants are less leader-centric)

A real implementation would need to do more:

- ▶ Efficient **log reconciliation** when  $\neg logOk$
- ▶ Allow **reconfiguration**:  
allow administrators to add or remove nodes, adjusting quorums accordingly
- ▶ Better **throughput**:  
avoid having to do everything through the leader?  
(some Paxos variants are less leader-centric)

Going even further:

- ▶ Raft assumes all nodes are honest;  
**Byzantine consensus** required for blockchains

# Eventual Consistency

Dr. Martin Kleppmann

[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

TU Munich

Chair of Decentralized Systems Engineering

<https://dse.in.tum.de/>

# Eventual consistency



Linearizability advantages:

- ▶ Makes a distributed system behave as if it were non-distributed
- ▶ Simple for applications to use

# Eventual consistency



Linearizability advantages:

- ▶ Makes a distributed system behave as if it were non-distributed
- ▶ Simple for applications to use

Downsides:

- ▶ **Performance** cost: lots of messages and waiting for responses

Linearizability advantages:

- ▶ Makes a distributed system behave as if it were non-distributed
- ▶ Simple for applications to use

Downsides:

- ▶ **Performance** cost: lots of messages and waiting for responses
- ▶ **Scalability** limits: leader can be a bottleneck

Linearizability advantages:

- ▶ Makes a distributed system behave as if it were non-distributed
- ▶ Simple for applications to use

Downsides:

- ▶ **Performance** cost: lots of messages and waiting for responses
- ▶ **Scalability** limits: leader can be a bottleneck
- ▶ **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

Linearizability advantages:

- ▶ Makes a distributed system behave as if it were non-distributed
- ▶ Simple for applications to use

Downsides:

- ▶ **Performance** cost: lots of messages and waiting for responses
- ▶ **Scalability** limits: leader can be a bottleneck
- ▶ **Availability** problems: if you can't contact a quorum of nodes, you can't process any operations

**Eventual consistency**: a weaker model than linearizability.  
Different trade-off choices.

Calendars +

Day Week Month Year United Kingdom Time ⌂ Search

# 5 November 2020

Thursday

all-day 07:00

08:00

09:00

10:00

11:00

12:00 12:00 Distributed systems lecture

13:00

14:00 14:00 Test

15:00

16:00

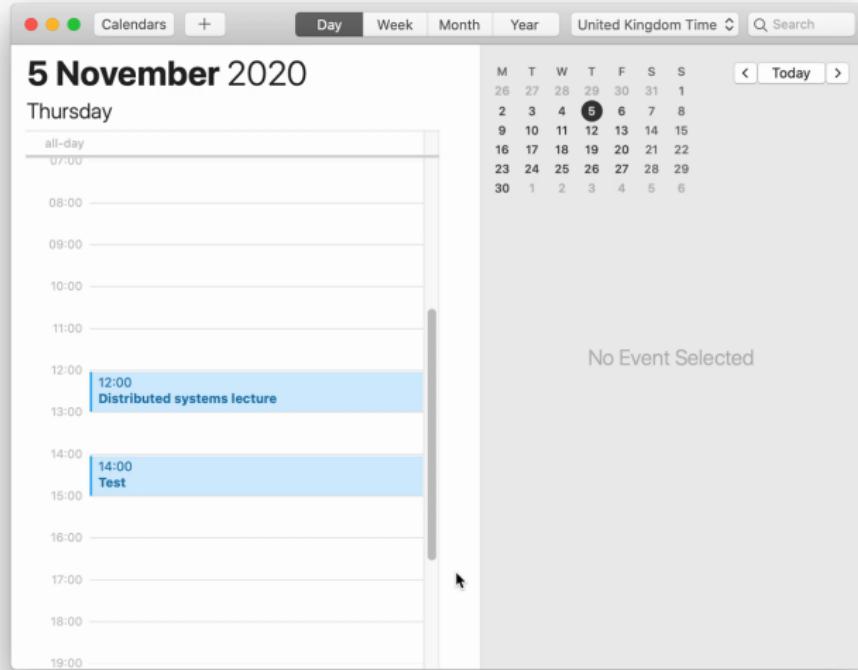
17:00

18:00

19:00

M T W T F S S  
26 27 28 29 30 31 1  
2 3 4 5 6 7 8  
9 10 11 12 13 14 15  
16 17 18 19 20 21 22  
23 24 25 26 27 28 29  
30 1 2 3 4 5 6

No Event Selected



09:41 100% 🔋

November

M T W T F S S  
2 3 4 5 6 7 8

Thursday 5 November 2020

10:00

11:00

12:00 Distributed systems lecture

13:00

14:00 Test

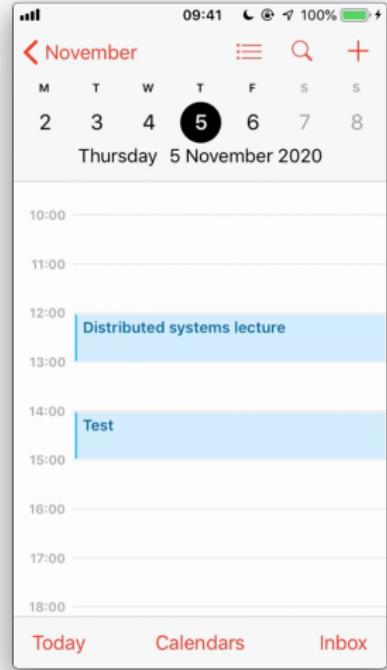
15:00

16:00

17:00

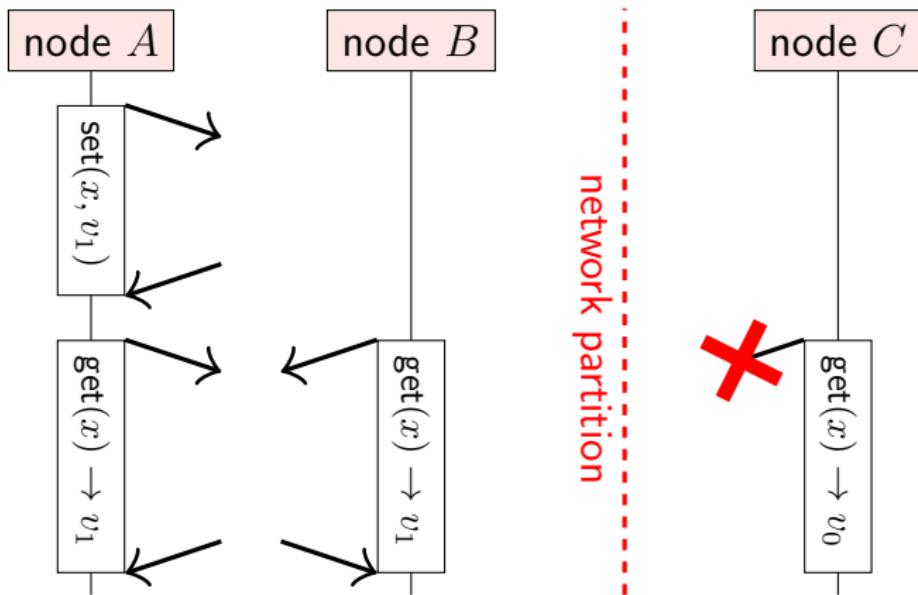
18:00

Today Calendars Inbox



# The CAP theorem

A system can be either strongly **Consistent** (linearizable) or **Available** in the presence of a network **Partition**



$C$  must either wait indefinitely for the network to recover, or return a potentially stale value

# Eventual consistency



Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

# Eventual consistency



Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

**Strong eventual consistency:**

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

## **Strong eventual consistency:**

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state  
(even if updates were processed in a different order).

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

## **Strong eventual consistency:**

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state  
(even if updates were processed in a different order).

## Properties:

- ▶ Does not require waiting for network communication
- ▶ Causal broadcast (or weaker) can disseminate updates

Replicas process operations based only on their local state.

If there are no more updates, **eventually** all replicas will be in the same state. (No guarantees how long it might take.)

## **Strong eventual consistency:**

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state  
(even if updates were processed in a different order).

## Properties:

- ▶ Does not require waiting for network communication
- ▶ Causal broadcast (or weaker) can disseminate updates
- ▶ Concurrent updates  $\implies$  **conflicts** need to be resolved

# Summary of system model requirements

Problem	Must wait for communication	Requires synchrony
atomic commit	all participating nodes	partially synchronous

↑  
strength of assumptions

# Summary of system model requirements

Problem	Must wait for communication	Requires synchrony
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous

↑  
strength of assumptions

# Summary of system model requirements

Problem	Must wait for communication	Requires synchrony
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous
linearizable get/set	quorum	asynchronous

↑  
strength of assumptions

# Summary of system model requirements

Problem	Must wait for communication	Requires synchrony
atomic commit	all participating nodes	partially synchronous
consensus, total order broadcast, linearizable CAS	quorum	partially synchronous
linearizable get/set	quorum	asynchronous
eventual consistency, causal broadcast, FIFO broadcast	local replica only	asynchronous

↑ strength of assumptions

# Local-first software



**End-user device is a full replica;** servers are just for backup.

“Local-first”: a term introduced by me and my colleagues

<https://www.inkandswitch.com/local-first/>

Calendar app with cross-device sync is an example:

**End-user device is a full replica**; servers are just for backup.

“Local-first”: a term introduced by me and my colleagues

<https://www.inkandswitch.com/local-first/>

Calendar app with cross-device sync is an example:

- ▶ App works **offline** (can both read and modify data)
- ▶ **Fast**: no need to wait for network round-trip
- ▶ **Sync** with other devices when online
- ▶ **Real-time collaboration** with other users

**End-user device is a full replica**; servers are just for backup.

“Local-first”: a term introduced by me and my colleagues

<https://www.inkandswitch.com/local-first/>

Calendar app with cross-device sync is an example:

- ▶ App works **offline** (can both read and modify data)
- ▶ **Fast**: no need to wait for network round-trip
- ▶ **Sync** with other devices when online
- ▶ **Real-time collaboration** with other users
- ▶ **Longevity**: even if cloud service shuts down, you have a copy of your files on your own computer
- ▶ Supports **end-to-end encryption** for better security
- ▶ Simpler **programming model** than RPC
- ▶ **User control** and agency over their own data

# Collaboration and conflict resolution



Nowadays we use a lot of **collaboration software**:

- ▶ **Examples:** calendar sync, text editors (Google Docs), spreadsheets, presentations, graphics apps...

Nowadays we use a lot of **collaboration software**:

- ▶ **Examples:** calendar sync, text editors (Google Docs), spreadsheets, presentations, graphics apps. . .
- ▶ Several users/devices working on a shared file/document
- ▶ Each user device has local replica of the data

Nowadays we use a lot of **collaboration software**:

- ▶ **Examples:** calendar sync, text editors (Google Docs), spreadsheets, presentations, graphics apps. . .
- ▶ Several users/devices working on a shared file/document
- ▶ Each user device has local replica of the data
- ▶ Update local replica optimistically, sync with others asynchronously (waiting for round trip is too slow)

Nowadays we use a lot of **collaboration software**:

- ▶ **Examples:** calendar sync, text editors (Google Docs), spreadsheets, presentations, graphics apps. . .
- ▶ Several users/devices working on a shared file/document
- ▶ Each user device has local replica of the data
- ▶ Update local replica optimistically, sync with others asynchronously (waiting for round trip is too slow)
- ▶ **Challenge:** how to reconcile concurrent updates?

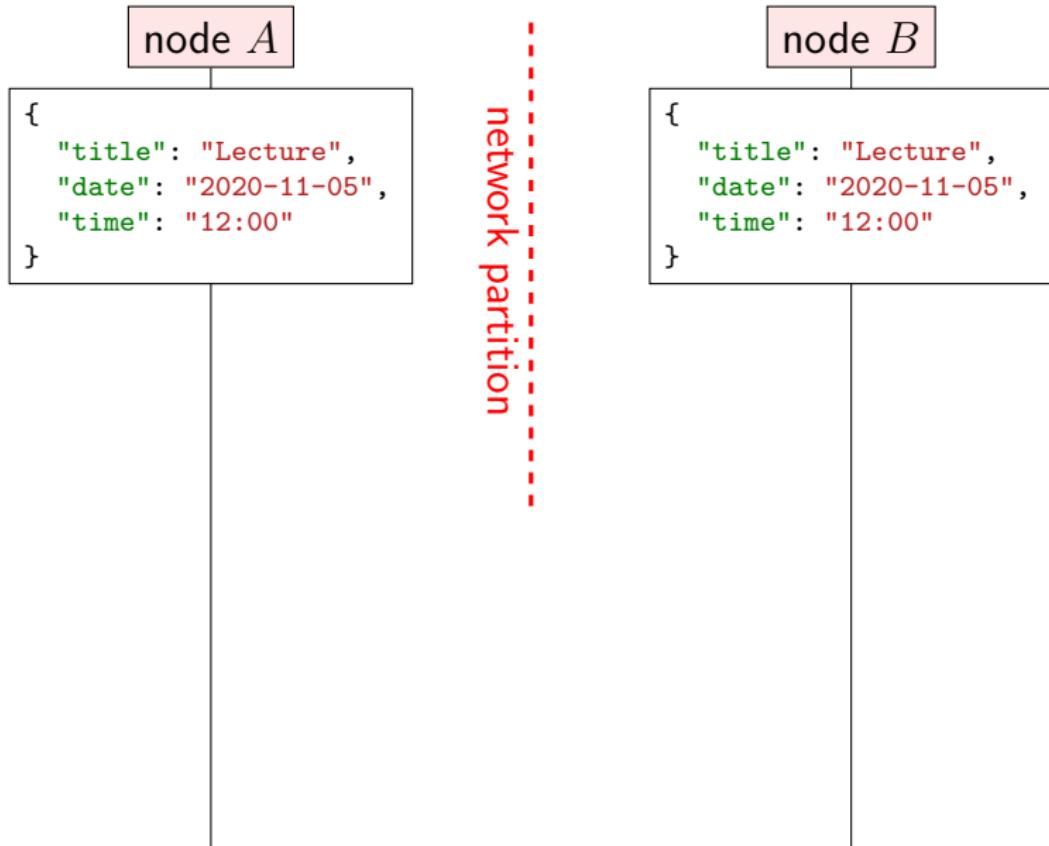
Nowadays we use a lot of **collaboration software**:

- ▶ **Examples:** calendar sync, text editors (Google Docs), spreadsheets, presentations, graphics apps. . .
- ▶ Several users/devices working on a shared file/document
- ▶ Each user device has local replica of the data
- ▶ Update local replica optimistically, sync with others asynchronously (waiting for round trip is too slow)
- ▶ **Challenge:** how to reconcile concurrent updates?

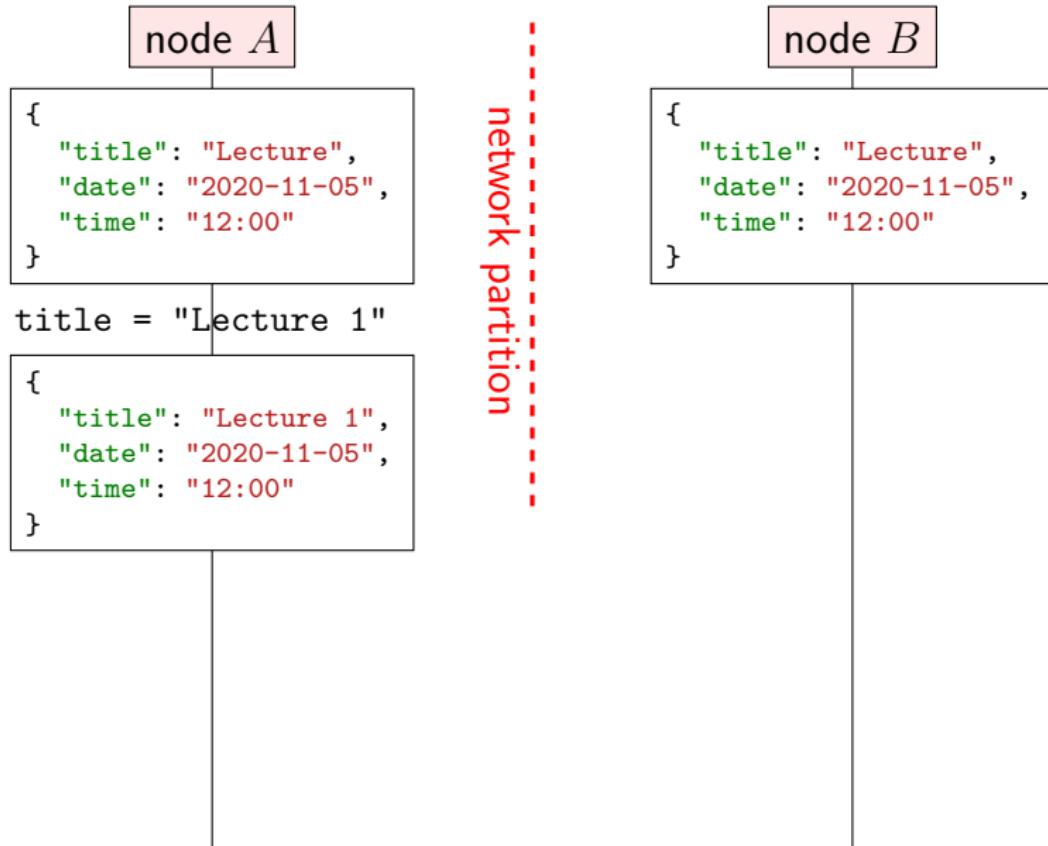
Families of **algorithms**:

- ▶ Conflict-free Replicated Data Types (**CRDTs**)
  - ▶ Operation-based
  - ▶ State-based
- ▶ Operational Transformation (**OT**)

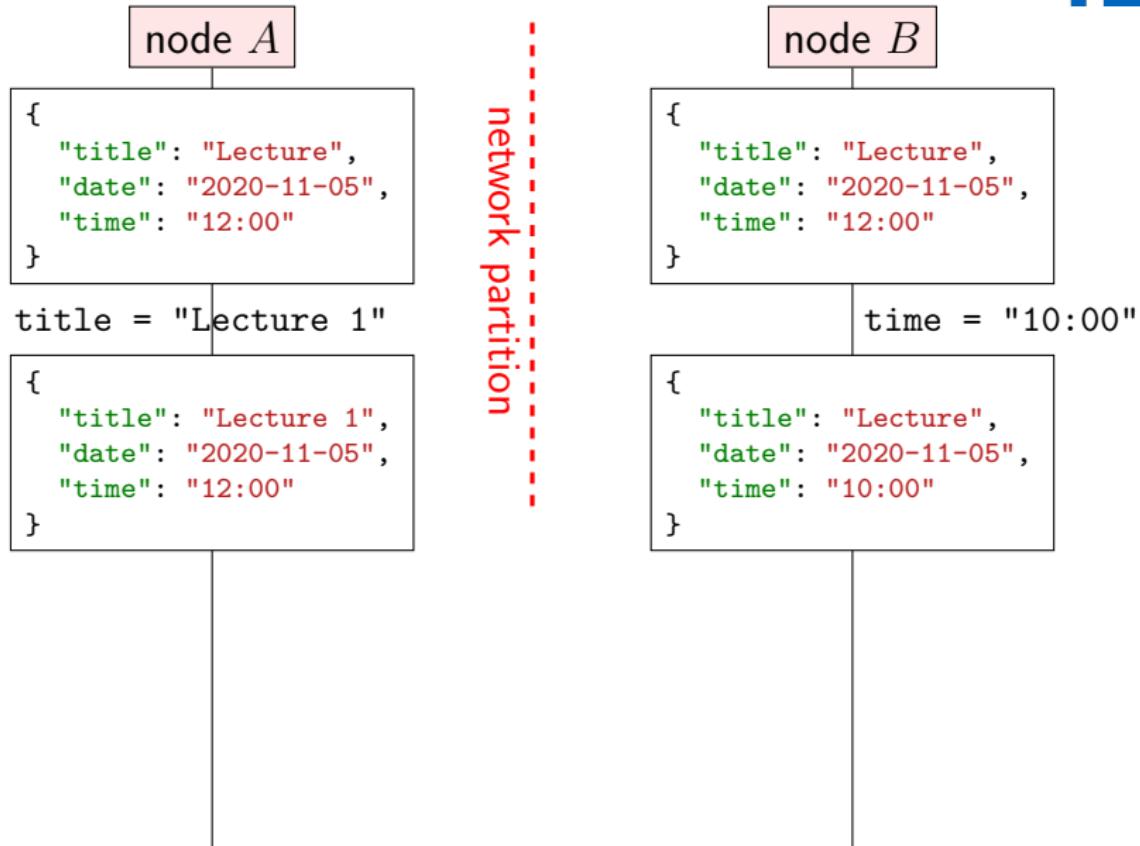
# Conflicts due to concurrent updates



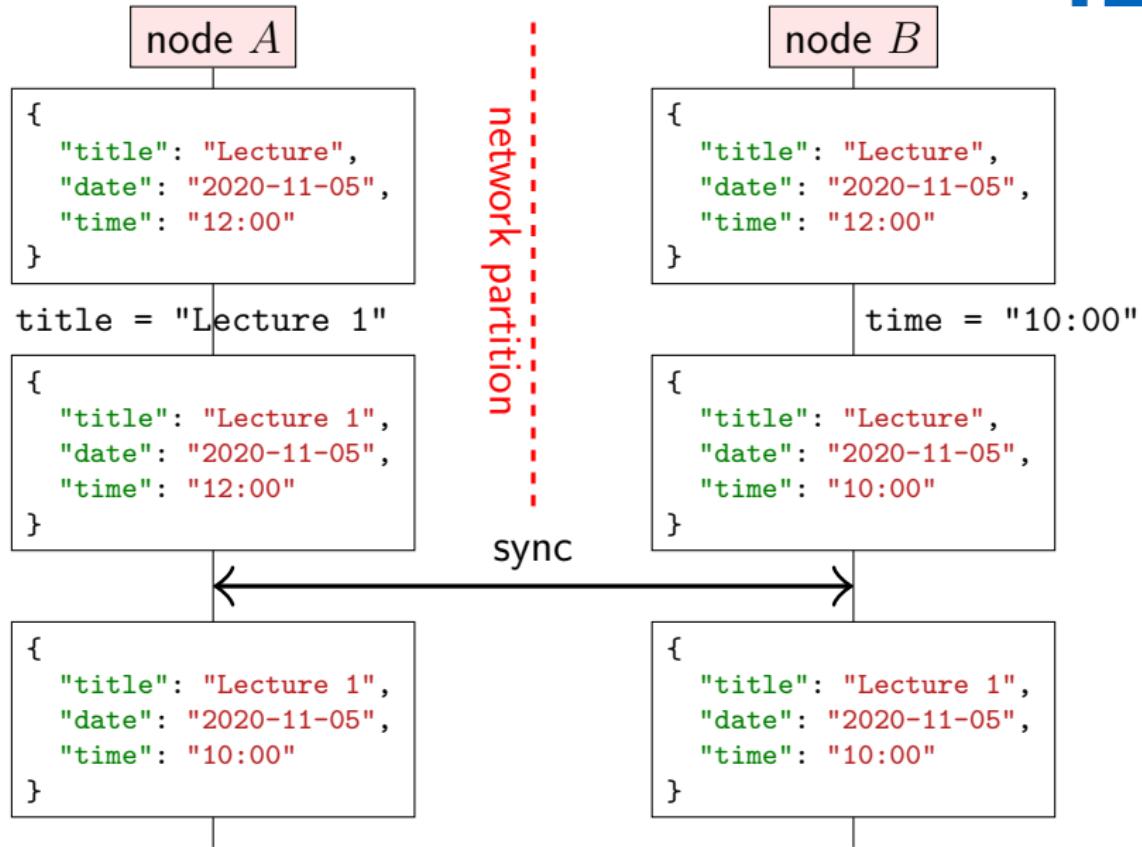
# Conflicts due to concurrent updates



# Conflicts due to concurrent updates



# Conflicts due to concurrent updates



# Operation-based map CRDT

```
on initialisation do
    values := {}
end on
```

```
on request to read value for key k do
    if  $\exists t, v. (t, k, v) \in values$  then return v else return null
end on
```

```
on request to set key k to value v do
    t := newTimestamp()  $\triangleright$  globally unique, e.g. Lamport timestamp
    broadcast (set, t, k, v) by reliable broadcast (including to self)
end on
```

```
on delivering (set, t, k, v) by reliable broadcast do
    previous :=  $\{(t', k', v') \in values \mid k' = k\}$ 
    if previous = {}  $\vee \forall (t', k', v') \in previous. t' < t$  then
        values := (values \ previous)  $\cup \{(t, k, v)\}$ 
    end if
end on
```

# Operation-based CRDTs



Reliable broadcast may deliver updates in any order:

- ▶ broadcast (set,  $t_1$ , "title", "Lecture 1")
- ▶ broadcast (set,  $t_2$ , "time", "10:00")

# Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- ▶ broadcast ( $\text{set}, t_1$ , "title", "Lecture 1")
- ▶ broadcast ( $\text{set}, t_2$ , "time", "10:00")

Recall **strong eventual consistency**:

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state

# Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- ▶ broadcast ( $\text{set}, t_1$ , "title", "Lecture 1")
- ▶ broadcast ( $\text{set}, t_2$ , "time", "10:00")

Recall **strong eventual consistency**:

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state

CRDT algorithm implements this:

- ▶ Reliable broadcast ensures every operation is eventually delivered to every (non-crashed) replica

# Operation-based CRDTs

Reliable broadcast may deliver updates in any order:

- ▶ broadcast ( $\text{set}, t_1$ , "title", "Lecture 1")
- ▶ broadcast ( $\text{set}, t_2$ , "time", "10:00")

Recall **strong eventual consistency**:

- ▶ **Eventual delivery:** every update made to one non-faulty replica is eventually processed by every non-faulty replica.
- ▶ **Convergence:** any two replicas that have processed the same set of updates are in the same state

CRDT algorithm implements this:

- ▶ Reliable broadcast ensures every operation is eventually delivered to every (non-crashed) replica
- ▶ Applying an operation is **commutative**: order of delivery doesn't matter

# State-based map CRDT

The operator  $\sqcup$  merges two states  $s_1$  and  $s_2$  as follows:

$$s_1 \sqcup s_2 = \{(t, k, v) \in (s_1 \cup s_2) \mid \nexists (t', k', v') \in (s_1 \cup s_2). k' = k \wedge t' > t\}$$

**on** initialisation **do**

*values* := {}

**end on**

**on** request to read value for key  $k$  **do**

**if**  $\exists t, v. (t, k, v) \in \text{values}$  **then return**  $v$  **else return** null

**end on**

**on** request to set key  $k$  to value  $v$  **do**

$t := \text{newTimestamp}()$   $\triangleright$  globally unique, e.g. Lamport timestamp

*values* :=  $\{(t', k', v') \in \text{values} \mid k' \neq k\} \cup \{(t, k, v)\}$

**broadcast** *values* by best-effort broadcast

**end on**

**on** delivering  $V$  by best-effort broadcast **do**

*values* := *values*  $\sqcup V$

**end on**

Merge operator  $\sqcup$  must satisfy:  $\forall s_1, s_2, s_3 \dots$

- ▶ **Commutative:**  $s_1 \sqcup s_2 = s_2 \sqcup s_1$ .
- ▶ **Associative:**  $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$ .
- ▶ **Idempotent:**  $s_1 \sqcup s_1 = s_1$ .

Merge operator  $\sqcup$  must satisfy:  $\forall s_1, s_2, s_3 \dots$

- ▶ **Commutative:**  $s_1 \sqcup s_2 = s_2 \sqcup s_1$ .
- ▶ **Associative:**  $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$ .
- ▶ **Idempotent:**  $s_1 \sqcup s_1 = s_1$ .

State-based versus operation-based:

- ▶ Op-based CRDT typically has smaller messages
- ▶ State-based CRDT can tolerate message loss/duplication

Merge operator  $\sqcup$  must satisfy:  $\forall s_1, s_2, s_3 \dots$

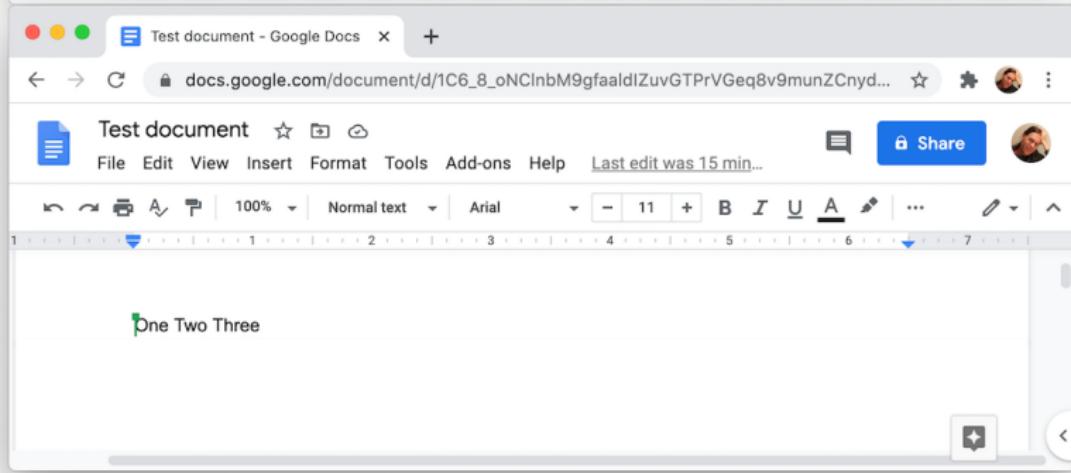
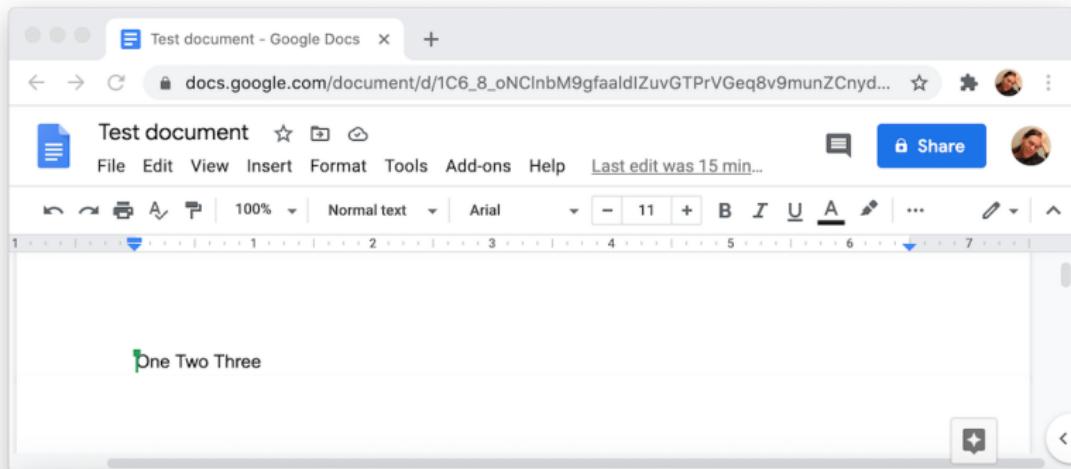
- ▶ **Commutative:**  $s_1 \sqcup s_2 = s_2 \sqcup s_1$ .
- ▶ **Associative:**  $(s_1 \sqcup s_2) \sqcup s_3 = s_1 \sqcup (s_2 \sqcup s_3)$ .
- ▶ **Idempotent:**  $s_1 \sqcup s_1 = s_1$ .

State-based versus operation-based:

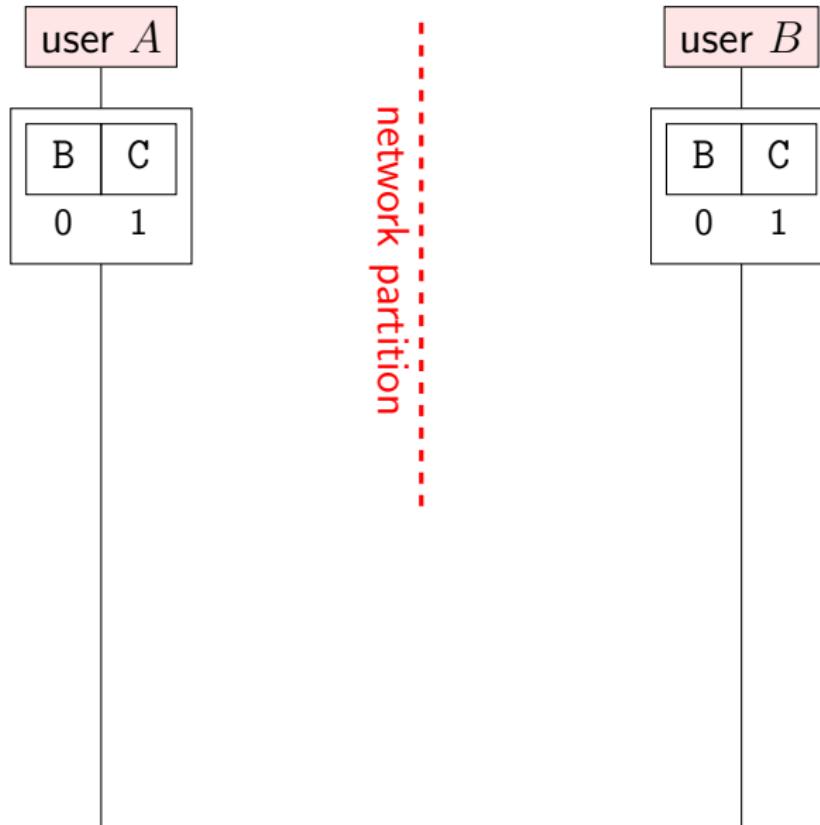
- ▶ Op-based CRDT typically has smaller messages
- ▶ State-based CRDT can tolerate message loss/duplication

Not necessarily uses broadcast:

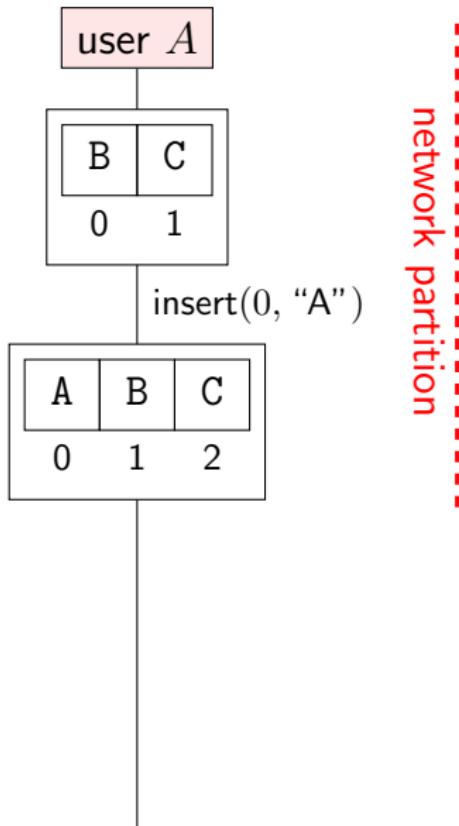
- ▶ Can also merge concurrent updates to replicas e.g. in quorum replication, anti-entropy, ...



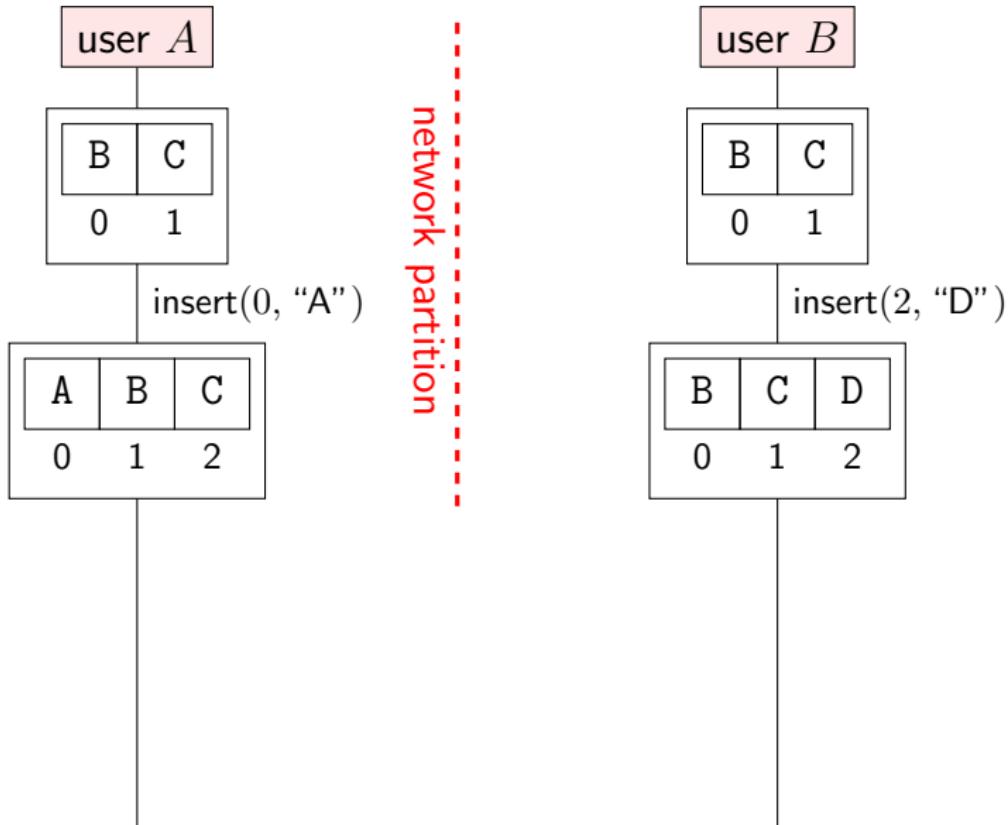
# Collaborative text editing: the problem



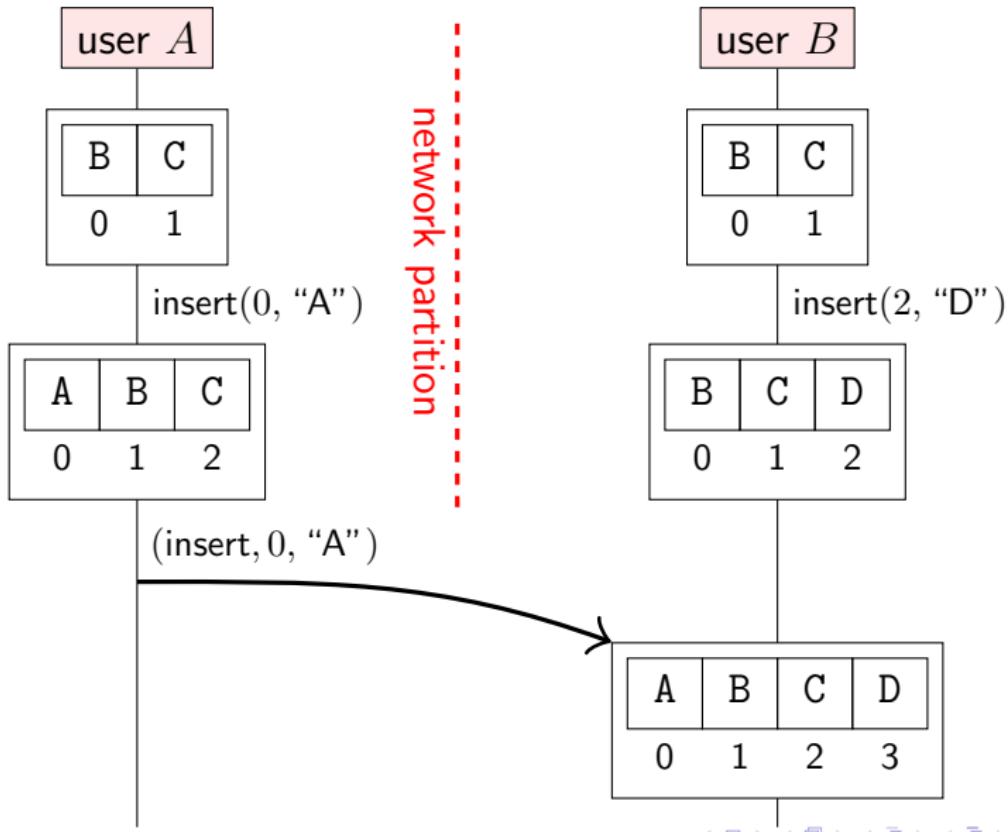
# Collaborative text editing: the problem



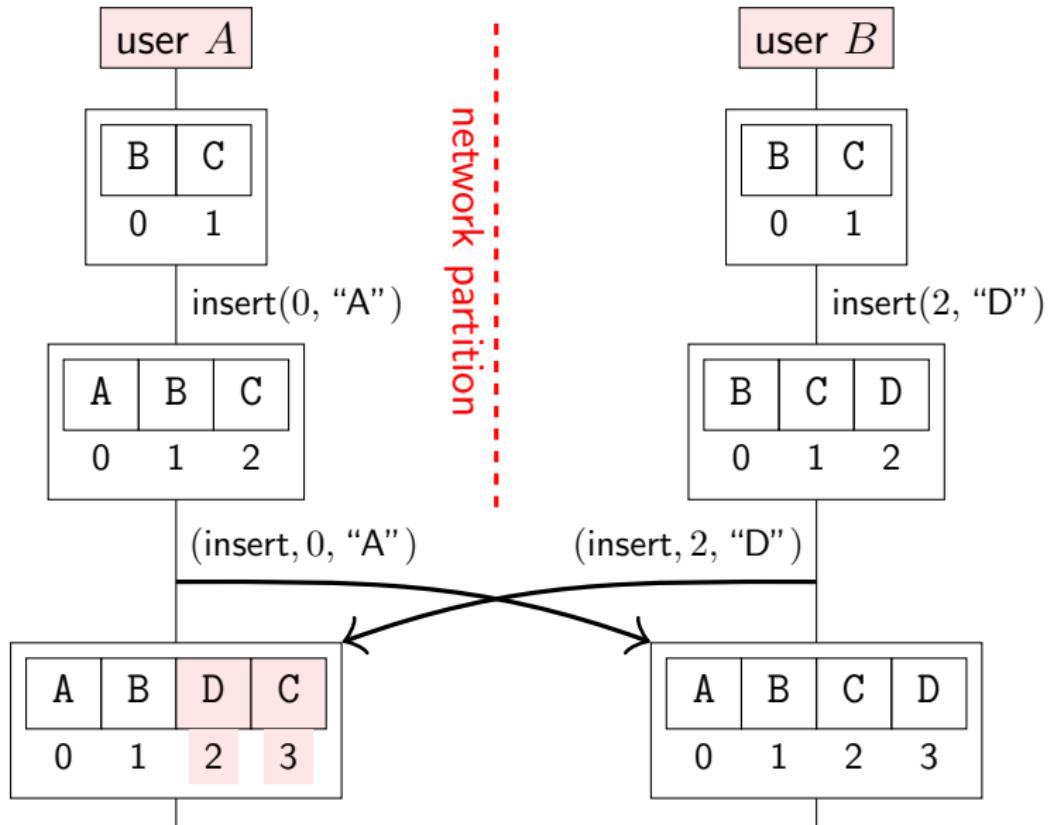
# Collaborative text editing: the problem



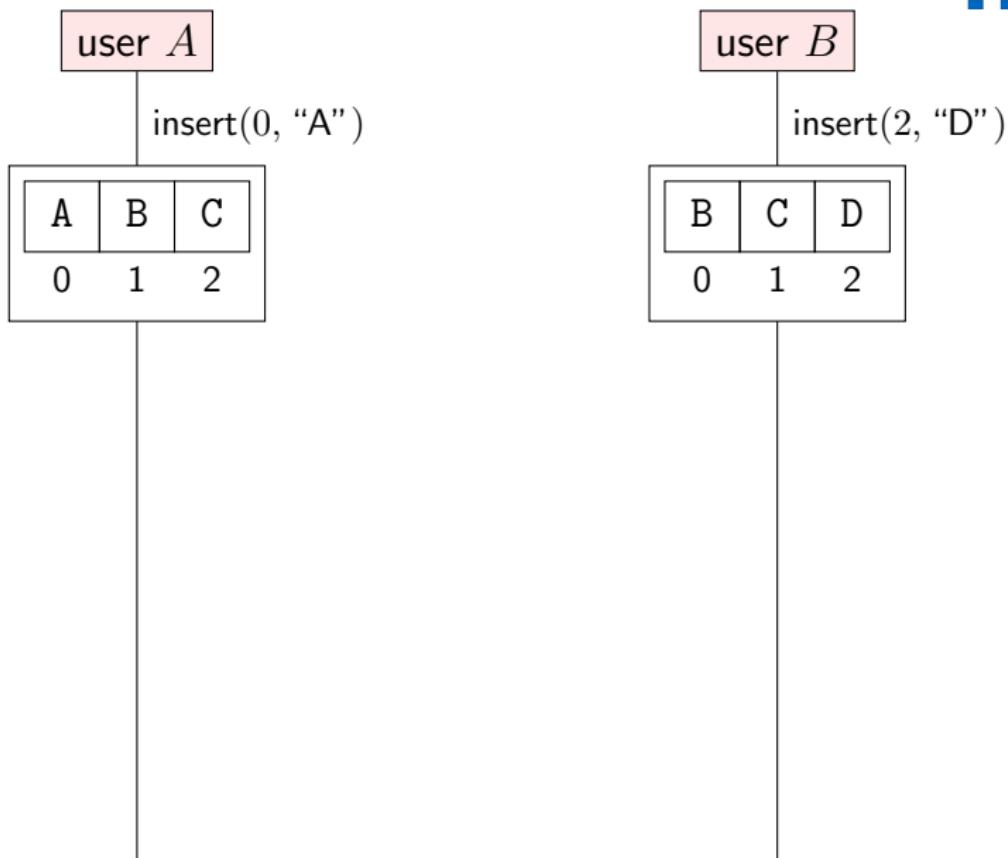
# Collaborative text editing: the problem



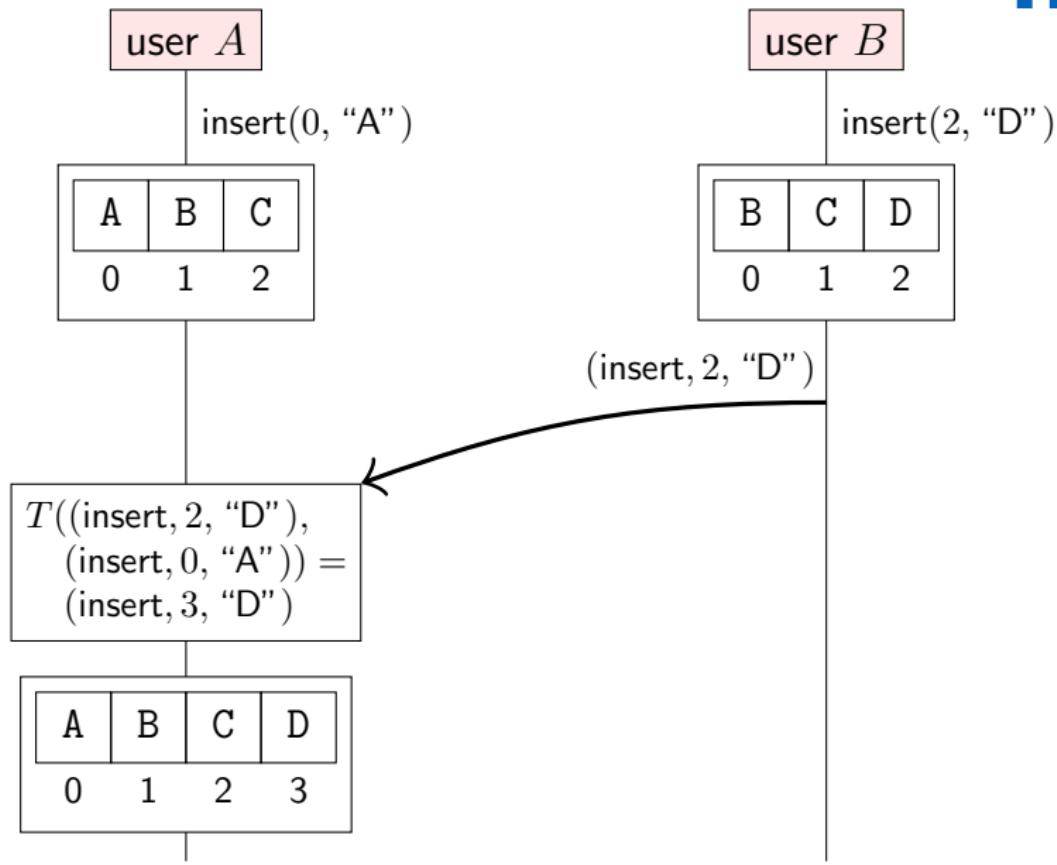
# Collaborative text editing: the problem



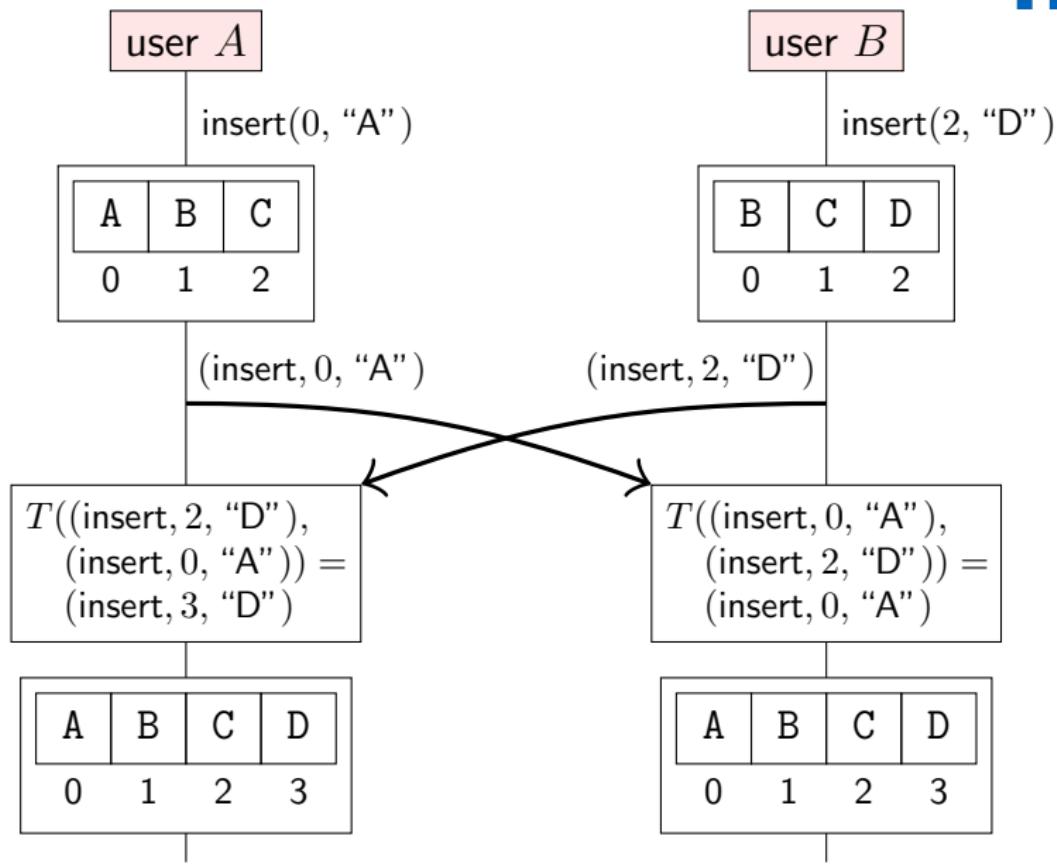
# Operational transformation



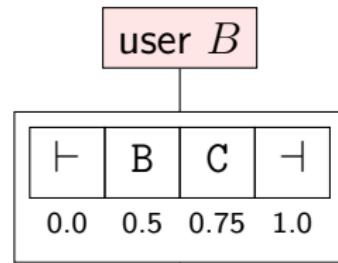
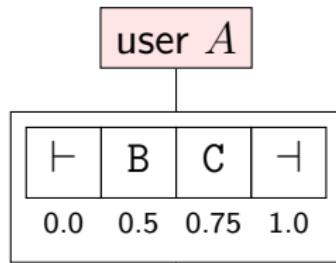
# Operational transformation



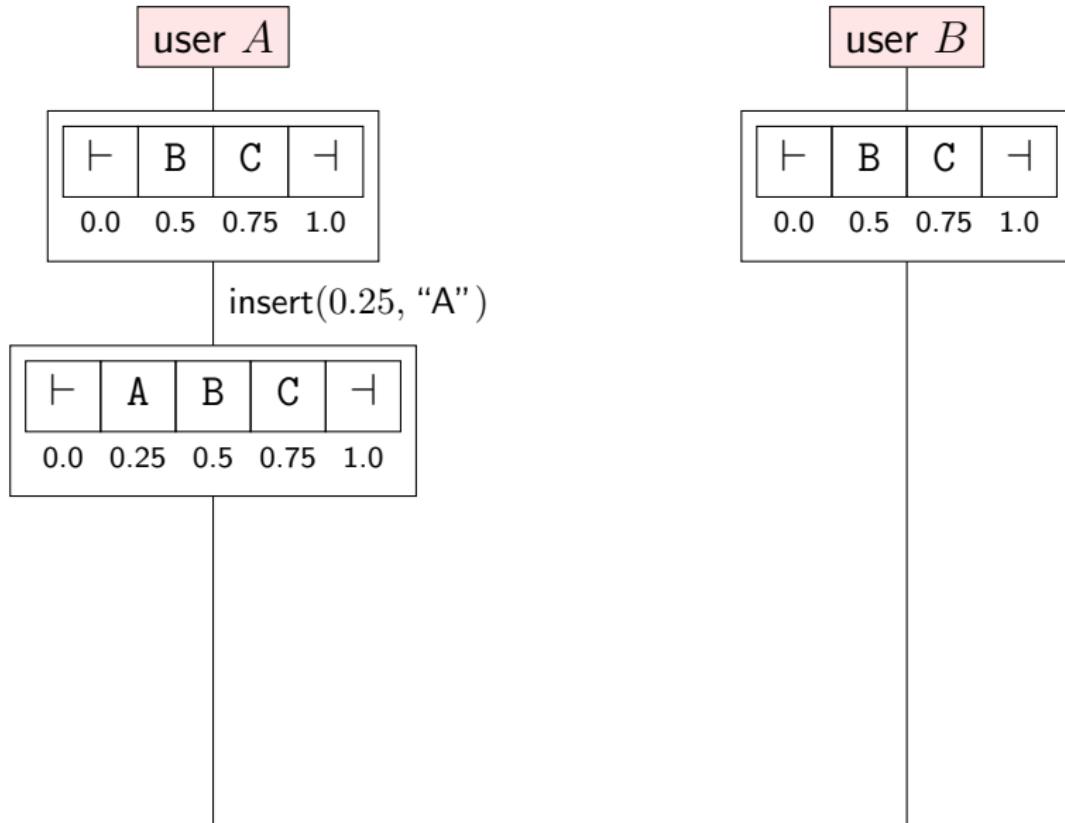
# Operational transformation



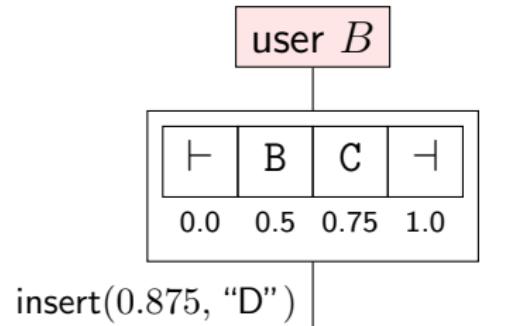
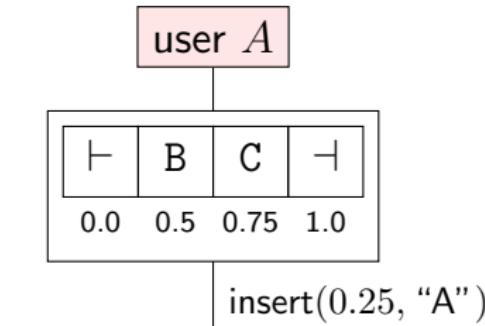
# Text editing CRDT



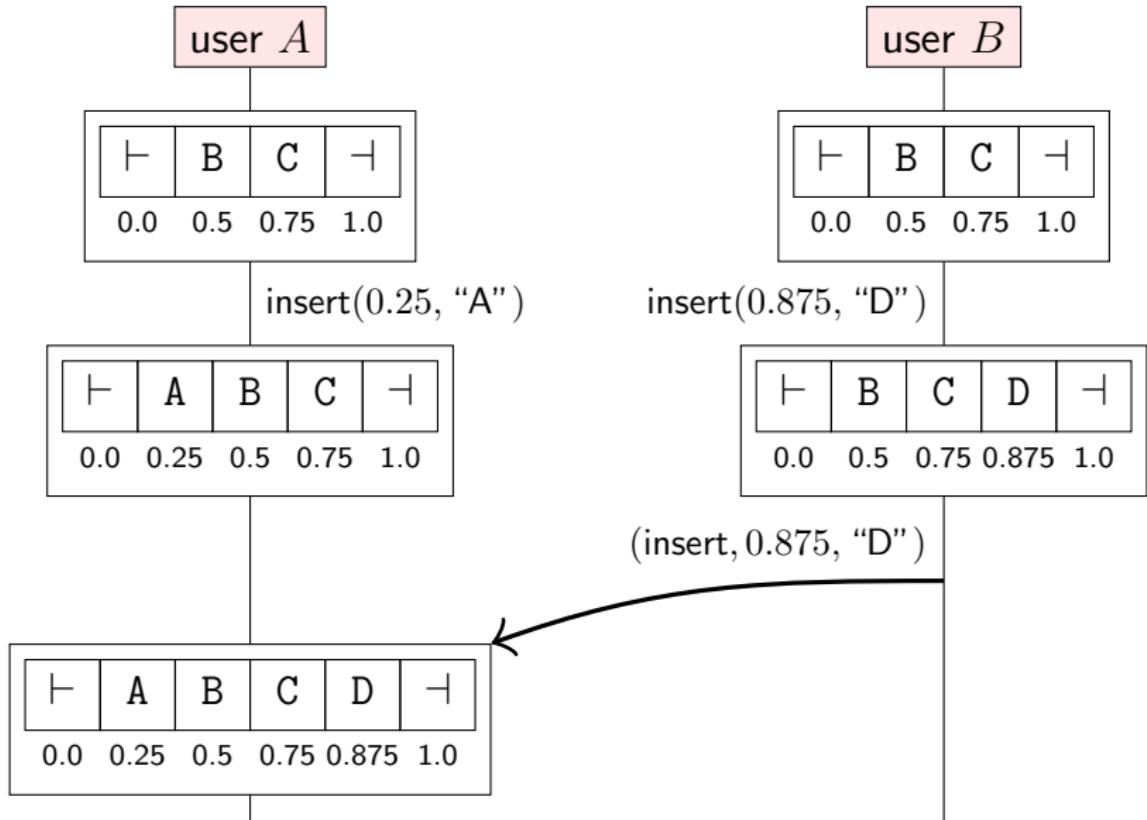
# Text editing CRDT



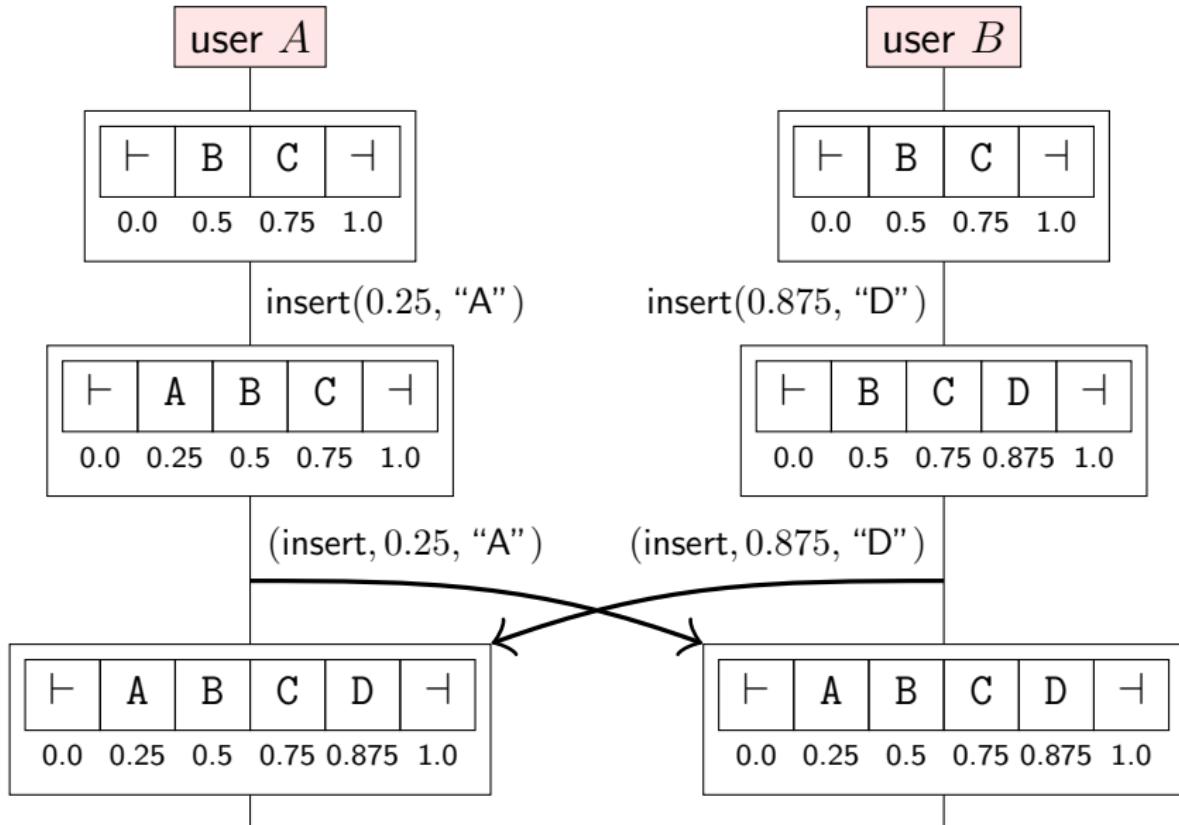
# Text editing CRDT



# Text editing CRDT



# Text editing CRDT



# Operation-based text CRDT (1/2)

**function** ELEMENTAT(*chars*, *index*)

*min* = the unique triple  $(p, n, v) \in \text{chars}$  such that

$\nexists (p', n', v') \in \text{chars}. p' < p \vee (p' = p \wedge n' < n)\}$

**if** *index* = 0 **then return** *min*

**else return** ELEMENTAT(*chars* \ {*min*}, *index* - 1)

**end function**

**on** initialisation **do**

*chars* := {(0, null, ⊢), (1, null, ⊢)}

**end on**

**on** request to read character at index *index* **do**

**let**  $(p, n, v) := \text{ELEMENTAT}(\text{chars}, \text{index} + 1)$ ; **return** *v*

**end on**

**on** request to insert character *v* at index *index* at node *nodeId* **do**

**let**  $(p_1, n_1, v_1) := \text{ELEMENTAT}(\text{chars}, \text{index})$

**let**  $(p_2, n_2, v_2) := \text{ELEMENTAT}(\text{chars}, \text{index} + 1)$

**broadcast** (insert,  $(p_1 + p_2)/2$ , *nodeId*, *v*) by causal broadcast

**end on**

# Operation-based text CRDT (2/2)

**on** delivering (*insert*,  $p$ ,  $n$ ,  $v$ ) by causal broadcast **do**  
     $chars := chars \cup \{(p, n, v)\}$   
**end on**

**on** request to delete character at index  $index$  **do**  
    **let**  $(p, n, v) := ELEMENTAT(chars, index + 1)$   
    **broadcast** (*delete*,  $p$ ,  $n$ ) by causal broadcast  
**end on**

**on** delivering (*delete*,  $p$ ,  $n$ ) by causal broadcast **do**  
     $chars := \{(p', n', v') \in chars \mid \neg(p' = p \wedge n' = n)\}$   
**end on**

- ▶ Use causal broadcast so that insertion of a character is delivered before its deletion
- ▶ Insertion and deletion of different characters commute

I am taking on research students (master + PhD) for projects in the following areas:

- ▶ implementing local-first collaboration software
- ▶ CRDTs (theory, new algorithms, performance, practical implementations)
- ▶ security protocols, encryption, privacy for collaboration
- ▶ Byzantine fault tolerant CRDTs
- ▶ formal verification (e.g. using Isabelle/HOL) of the above

Please get in touch if you're interested:  
[martin.kleppmann@in.tum.de](mailto:martin.kleppmann@in.tum.de)

## Summary:

- ▶ Distributed systems are everywhere
- ▶ You use them every day: e.g. web apps
- ▶ Key goals: availability, scalability, performance
- ▶ Key problems: concurrency, faults, unbounded latency
- ▶ Key abstractions: replication, broadcast, consensus
- ▶ No one right way, just trade-offs

Next week: Prof. Pramod Bhatotia is back!

