SCC.311: Fault Tolerance II

Prof. Barry Porter

A note on interfaces and types

- Our coursework is a distributed service which advertises and implements a well-defined interface to clients
  - As long as a client conforms to our advertised interface definition, *any* client can use our service

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

A note on interfaces and types

- Type compatibility in Java is strict…
  - Are these two types compatible?

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public interface Potato {
AuctionItem getSpec(int itemID);
}
```

```
public class MyThing {
…
Auction a = makeAuctionThing();
Potato p = makePotatoThing();
a = p; //what happens?
…
}
```

A note on interfaces and types

- Type compatibility in Java is strict...
  - In a distributed system, we can easily have two types with the same name, but different definitions...

client

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

server

```
public interface Auction {
AuctionItem getSpec(int itemID);
int getAuctionCount();
}
```

Are these two types compatible?

A note on interfaces and types

- Type compatibility in Java is strict…
  - In a distributed system, we can easily have two types with the same name, but different definitions…

client

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

server

```
public class AuctionItem {
int itemID;
String name;
String description;
int clara;
}
```

Are these two types compatible?

A note on interfaces and types

- Type compatibility in Java is strict…
  - In a distributed system, we can easily have two types with the same name, but different definitions…

client

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

server

```
package org.me.services;
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

Another example: are these two types compatible?

A note on interfaces and types

- To make any client compatible, our server must strictly implement the exact types that we advertised

client

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

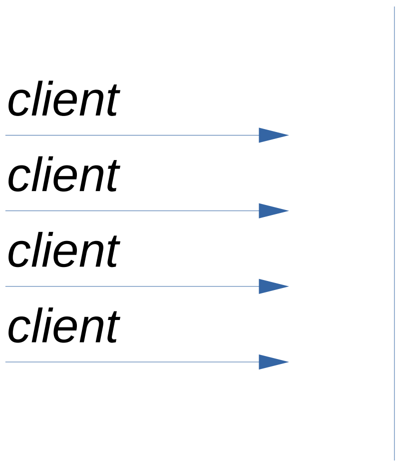
server

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

A note on interfaces and types

- ...but, behind these types, our server-side can be *implemented* however we like



server

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

```
public class MyServer implements Auction {
 ... everything in one class ...
}
```

A note on interfaces and types

- ...but, behind these types, our server-side can be *implemented* however we like

server

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

client

client

client

client

```
public class MyServer implements Auction {
 ... or in many classes, used by this class ...
}
```

```
public class Users {
 bool addUser(..){..}
}
```

```
public class Items {
 bool addItem(..){..}
}
```

A note on interfaces and types

- ...but, behind these types, our server-side can be *implemented* however we like

client

client

client

client

server

```
public interface Auction {
AuctionItem getSpec(int itemID);
}
```

```
public class AuctionItem {
int itemID;
String name;
String description;
int highestBid;
}
```

```
public class MyServer implements Auction {
 ... or this class could
     even be a load-balancer ...
}
```

serverA

serverB

serverC

serverD

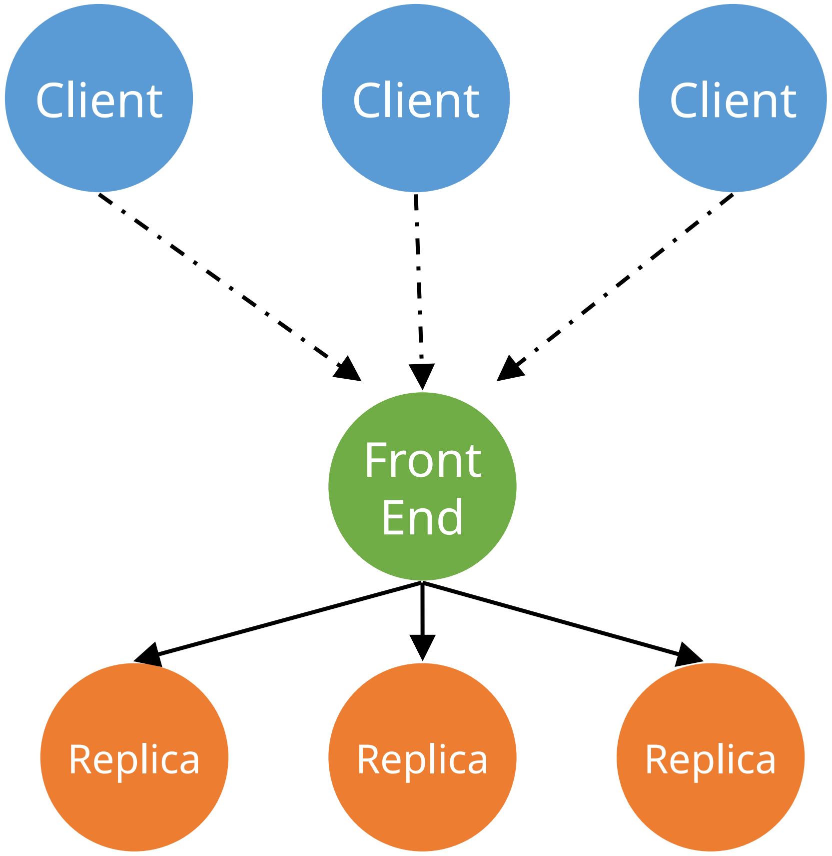
serverE

Overview for today

- On Monday we saw replication schemes for fault tolerance

- Now we look at **message ordering** for group communication (used in replication), and why we need **consensus algorithms** to agree on a value

Active Replication review...



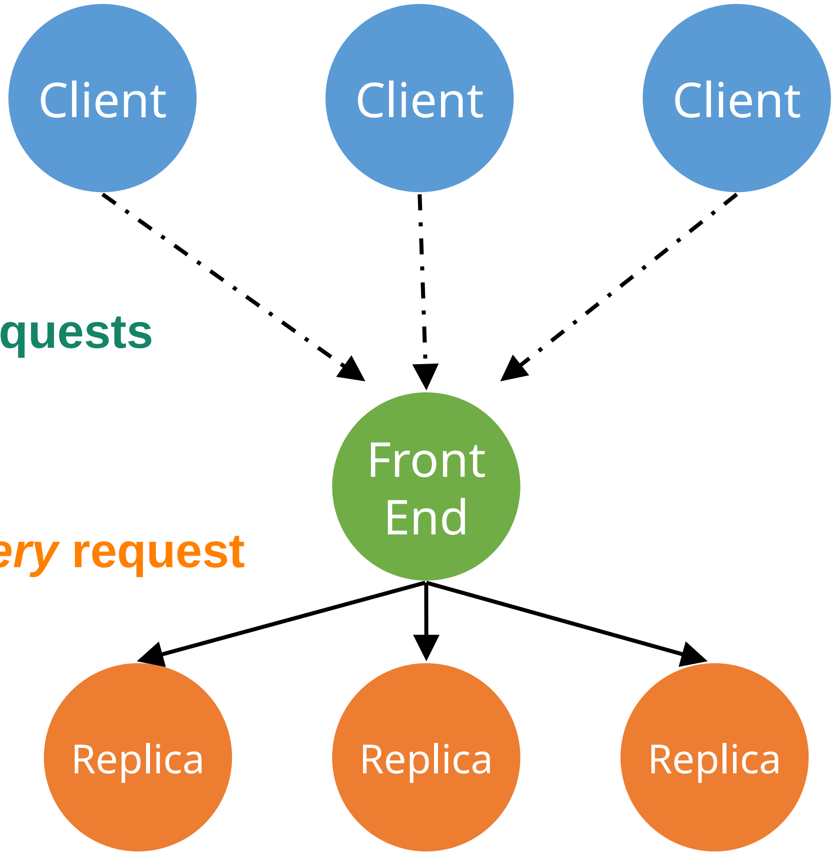
Supports crash failures

Supports Byzantine failures

Active Replication review...

1: Front End receives client requests

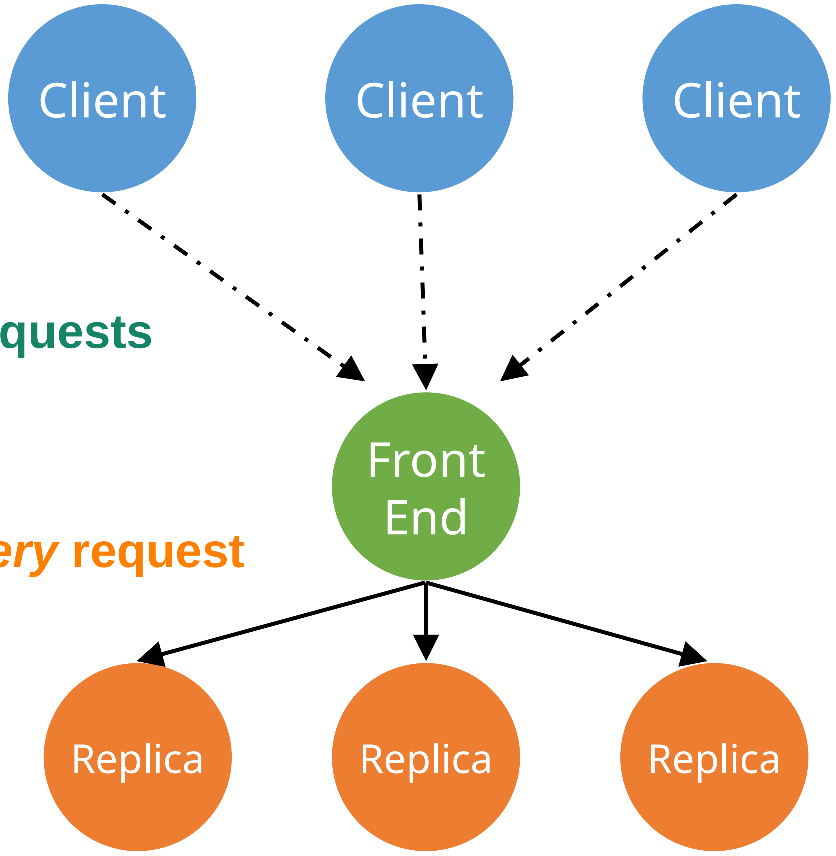
2: Every replica processes *every* request



Active Replication review…

1: Front End receives client requests

2: Every replica processes *every* request



3: Front End receives responses from every replica for a given request

4: Front End *compares* responses to check they're all the same

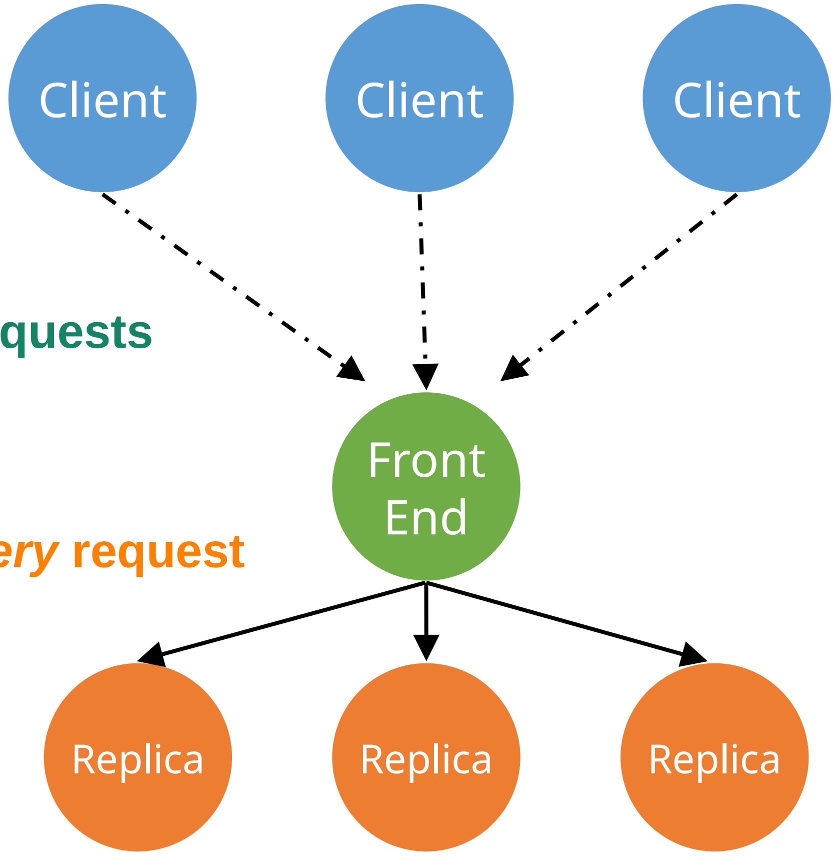
5: Front End sends *majority* response value to client

Supports Byzantine failures

Active Replication review…

1: Front End receives client requests

2: Every replica processes *every* request



3: Front End receives responses from every replica for a given request

4: Front End *compares* responses to check they're all the same

5: Front End sends *majority* response value to client

Supports Byzantine failures

Q: In a highly parallel environment (many clients), how should I order the messages to replicas?

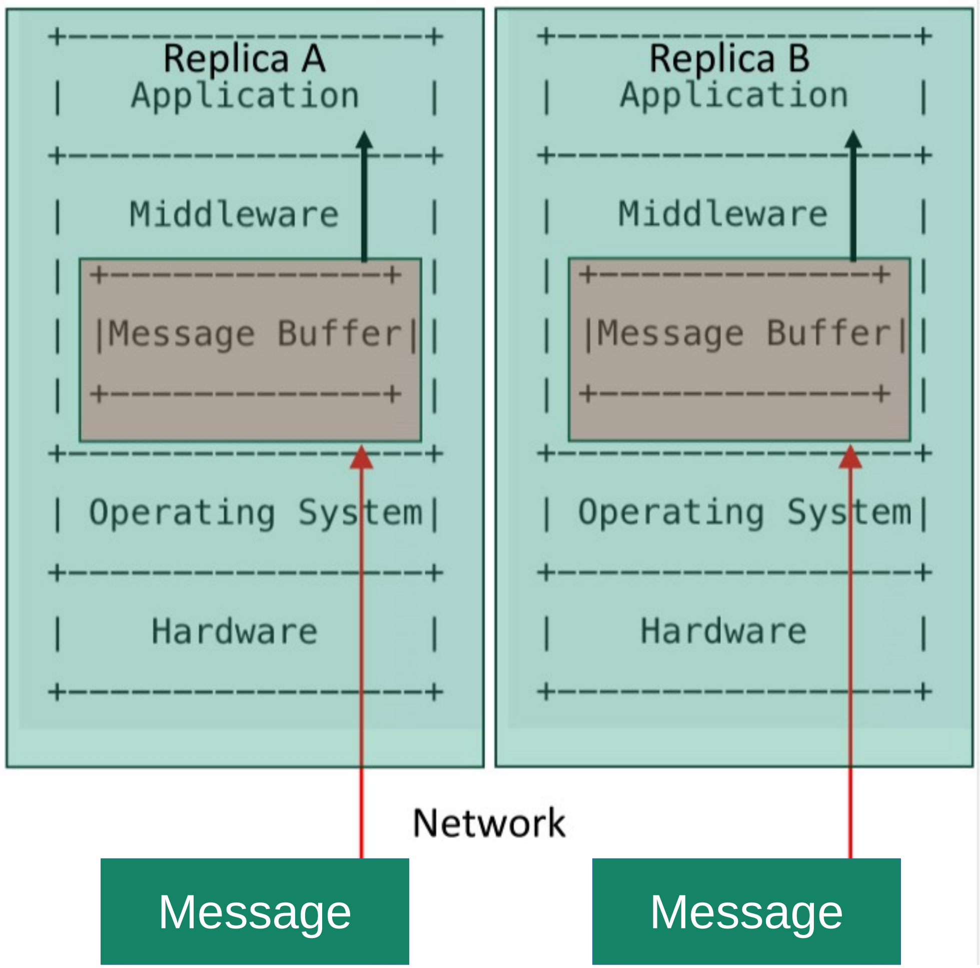
Message ordering

- We've already looked at group communication in general, in which we can have different levels of reliability: best-effort, reliable, atomic

- In a distributed system with many clients, the *ordering* of messages is also important to consider in conjunction with the reliability level
    - We can have schemes such as no ordering, FIFO, causal ordering, and total ordering
    - The more strict our ordering semantics are, the more we need to pay in synchronisation costs and we lose in message handling throughput

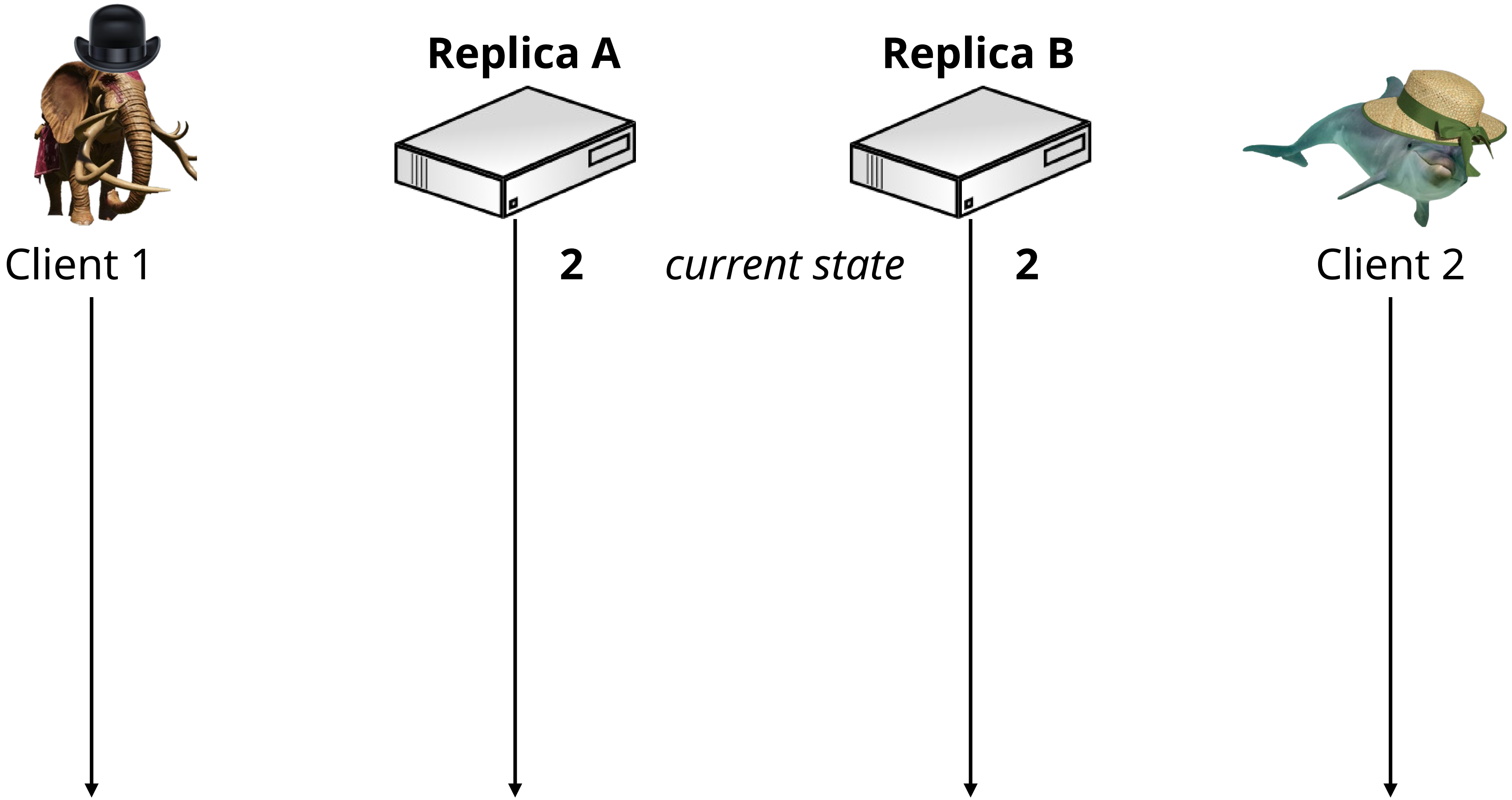
System Model

- We assume regular Internet communication, i.e. best-effort: network can arbitrarily reorder messages during the transit
- Incoming messages pass through a "group communication middleware" before arriving at the application (replica)
- The middleware can optionally buffer incoming messages and re-order them based on chosen ordering semantics, before delivering them to the application
- By "message order", we mean the order of messages as they are delivered to the application



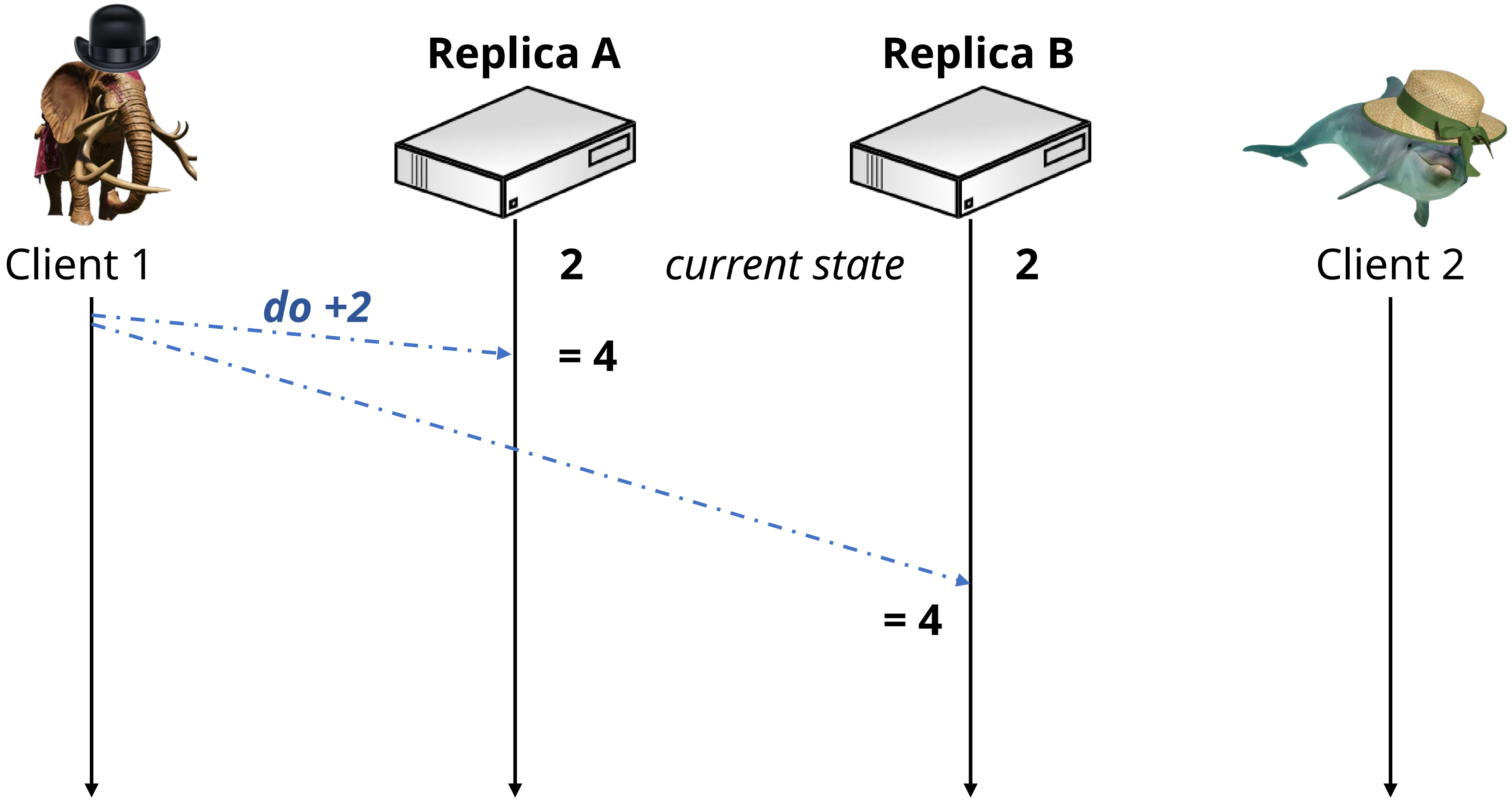
Message ordering example

Imagine a simple example service which can just add or multiply a stored number...



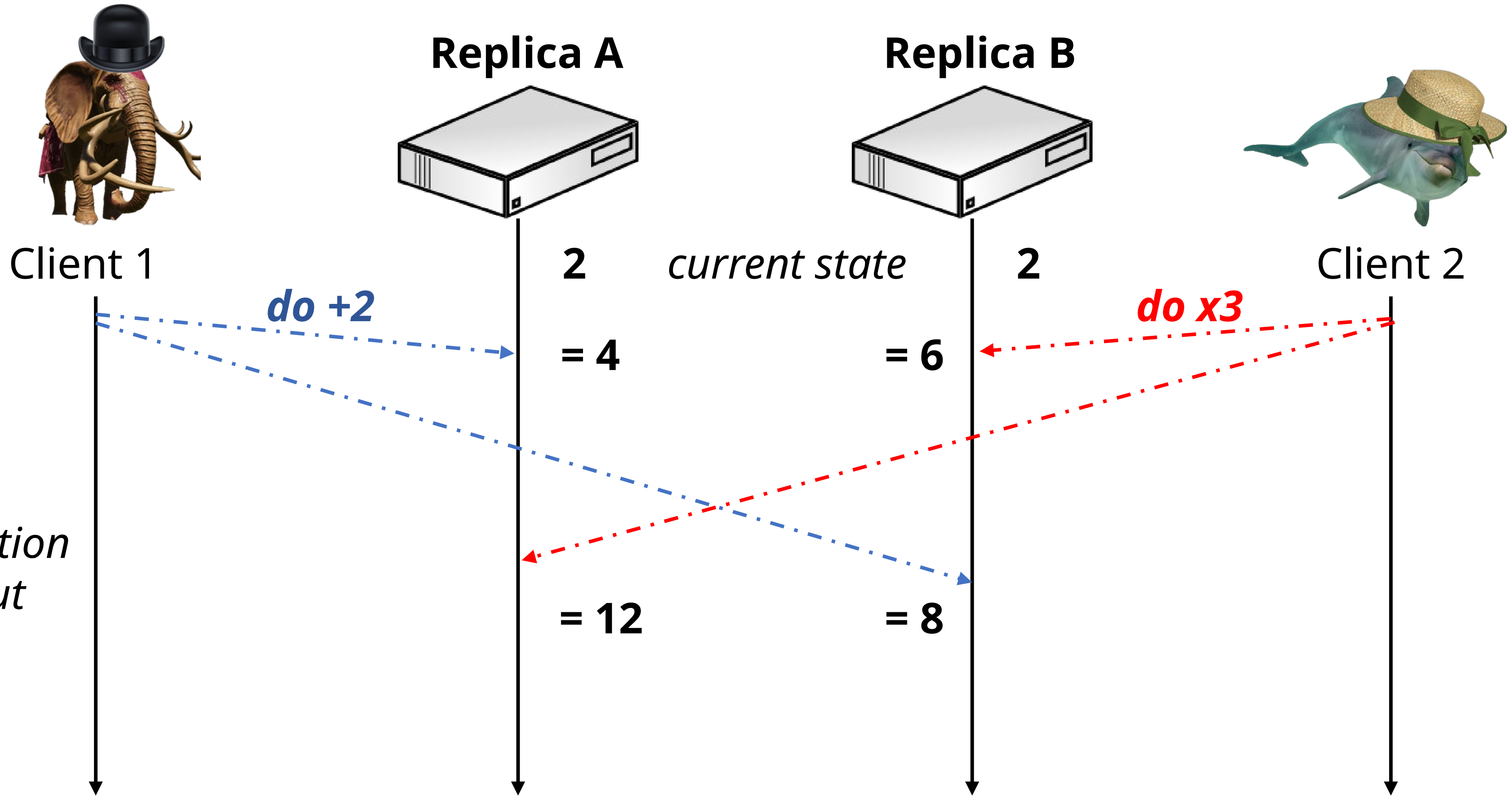
Message ordering example

Imagine a simple example service which can just add or multiply a stored number…



Message ordering example

Imagine a simple example service which can just add or multiply a stored number...



If our group communication service doesn't care about ordering, we can get varying states among group members which eventually disagree...

Message ordering

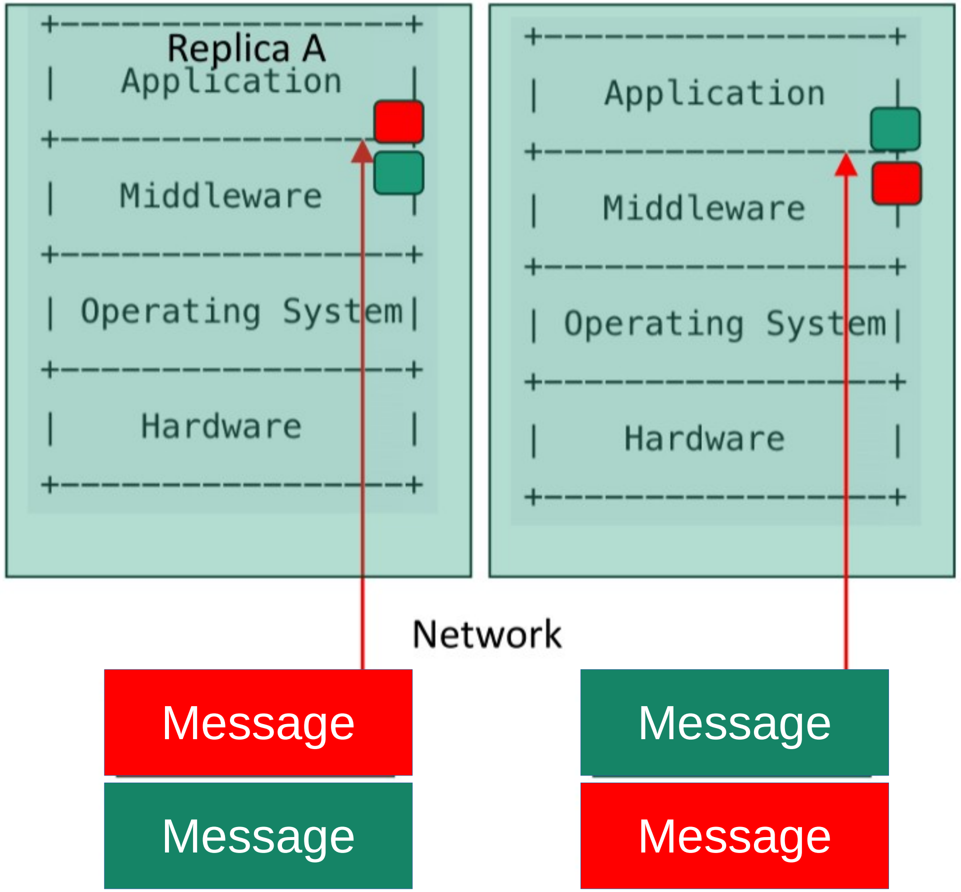
- As with other communication protocol semantics, we need to decide carefully on the minimum ordering constraints that are required

- This allows us to maximise throughput of a system, and to avoid wasting resources on unnecessary messaging

Message ordering // unordered

- This is the same as just using best-effort/reliable/atomic multicast with no constraints over message ordering

- We assume that different members of the group may receive messages in different orders, and our application is OK with that



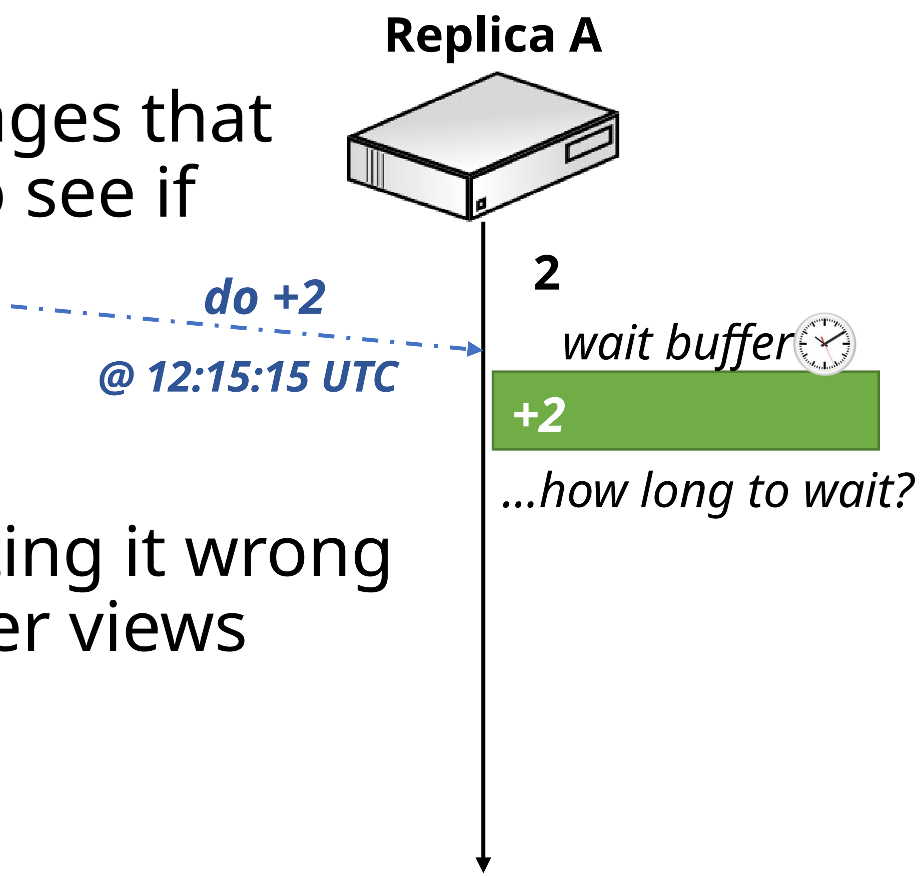
Message ordering // global time ordering

- We could use a simple approach in which each sender of a message stamps that message with the current system time
- Whenever a node receives a message, we use the timestamp to decide on the order in which to process these messages
- To achieve this, receivers need to buffer messages before processing them to allow for network delays between different senders
  - Knowing how long to buffer messages is problematic
  - Having a globally synchronised clock to a high enough accuracy is unlikely
  - We have to assume no two messages are ever sent at exactly the same time

Message ordering // global time ordering

- A wait buffer approach would place messages that arrive into a temporary queue, and wait to see if any other messages arrive with an earlier timestamp

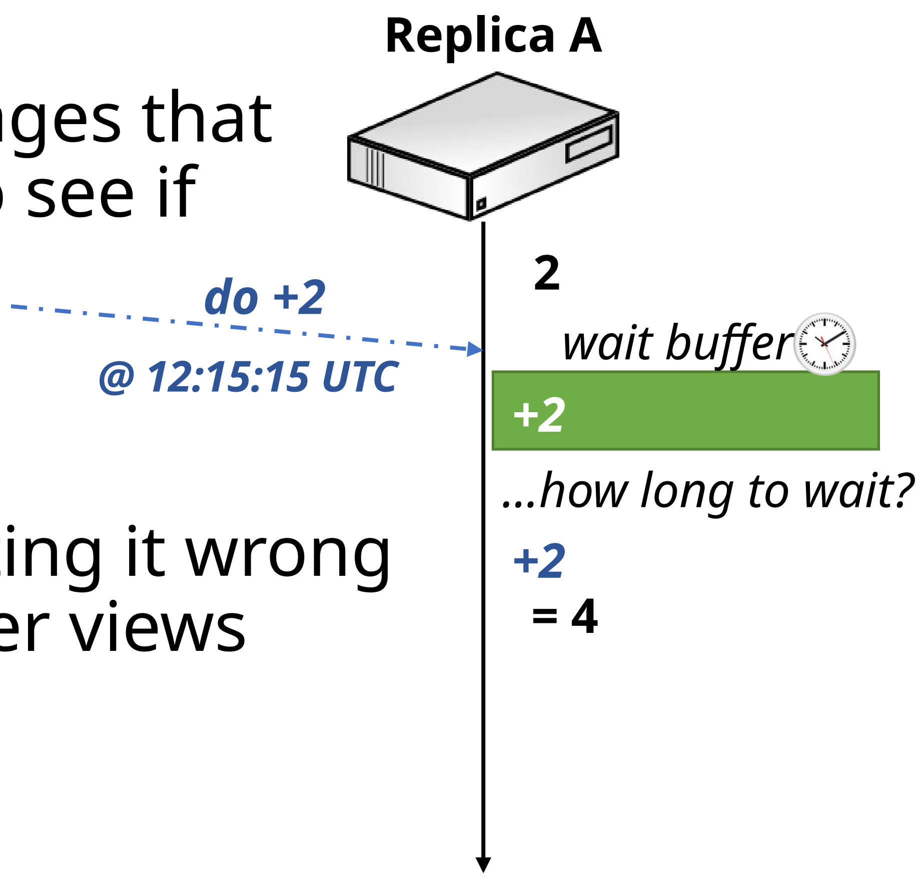
- How long we wait is unknowable, and getting it wrong can easily break consistency of client/server views



Message ordering // global time ordering

- A wait buffer approach would place messages that arrive into a temporary queue, and wait to see if any other messages arrive with an earlier timestamp

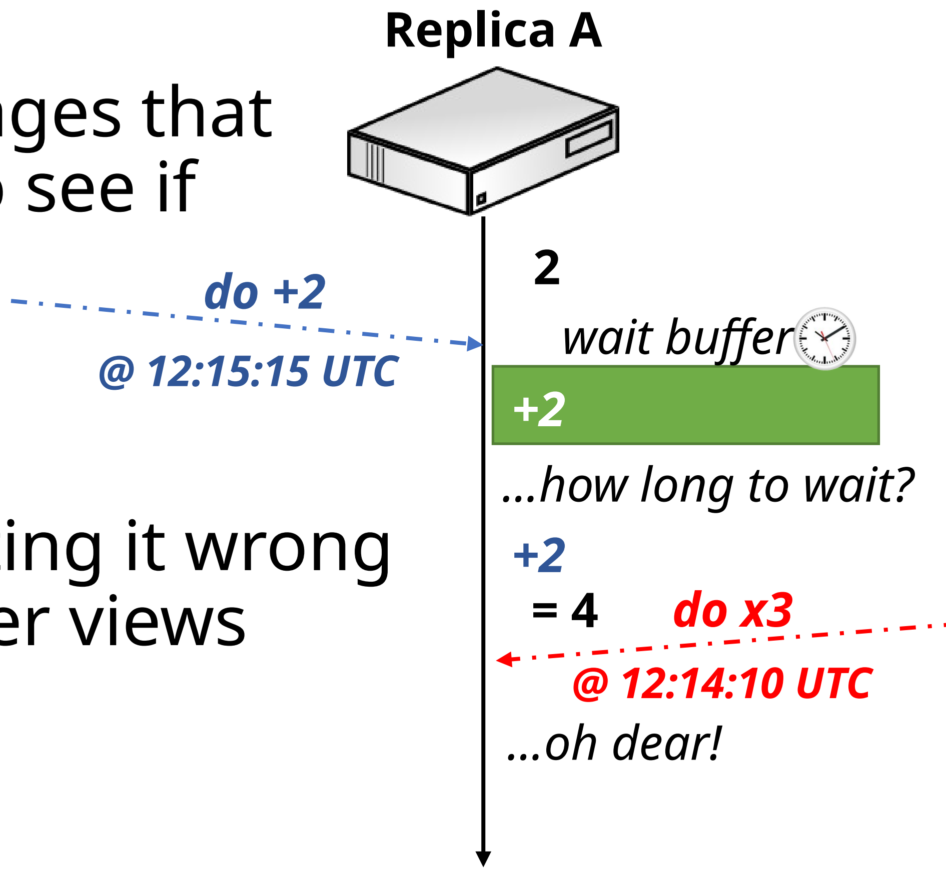
- How long we wait is unknowable, and getting it wrong can easily break consistency of client/server views



Message ordering // global time ordering

- A wait buffer approach would place messages that arrive into a temporary queue, and wait to see if any other messages arrive with an earlier timestamp

- How long we wait is unknowable, and getting it wrong can easily break consistency of client/server views



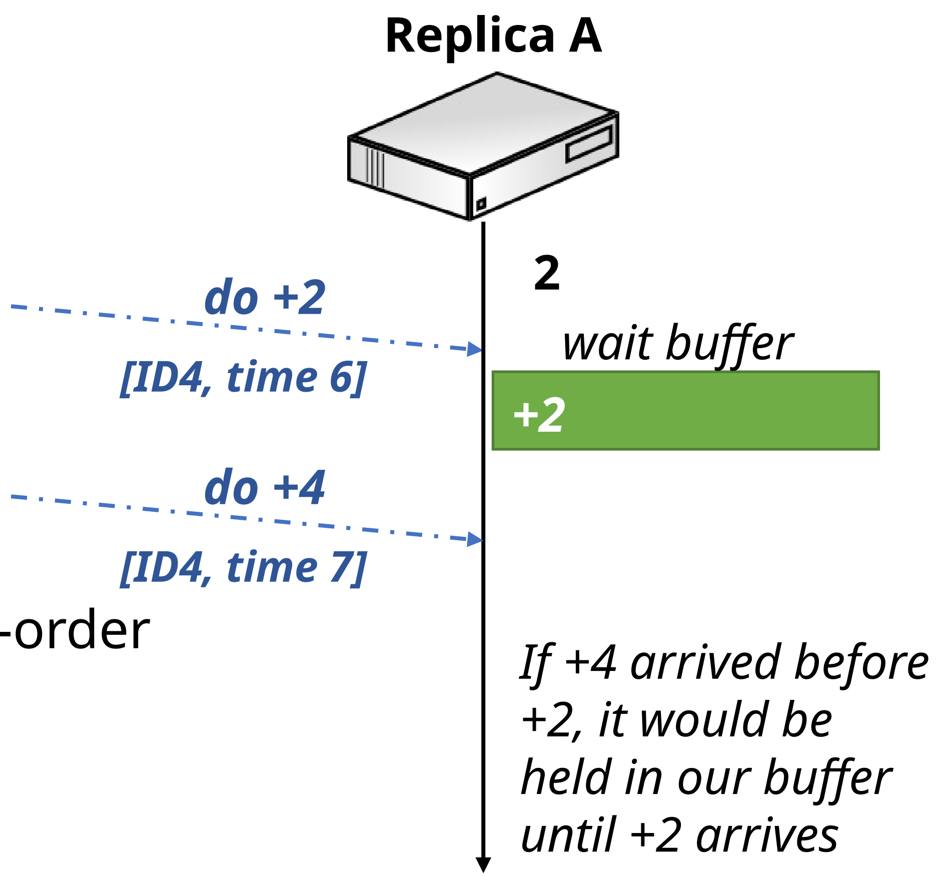
Message ordering // FIFO

- Messages from the same process are handled at receivers in the same order with which they were sent
- We don't care about ordering among multiple different senders, only that messages from each individual sender and handled in-order
- This is easy to implement by using a logical message time (an int) on each message that we send, where each new message has a time +1
  - TCP guarantees this property at a network level

Message ordering // FIFO

- Implementing FIFO still requires a buffer at the receivers in case messages arrive out of order, **but** in this case we don't have uncertainty about how long to buffer messages

- Each message is stamped with a sender ID, and a logical time (just a monotonically increasing int)
  - The receiver then know if a message arrived out-of-order and can wait for the earlier message



Message ordering // causal

- Causal ordering considers all processes/senders in the system (not just per-process as in FIFO), but employs a *partial* ordering semantic to enable higher message processing rates at receivers
  - And so higher degrees of scalability / parallelism across servers in a group

- If we have two messages m1 and m2, causal ordering will ensure that m1 is processed before m2 only if there is a "happened-before" relationship between the two messages
  - If there isn't such a relationship, then we don't care about the ordering

Message ordering // causal

- Example: post an update an Facebook, and a friend posts a comment
  - Here the order is important because you can't comment on a non-existent update, so the creation of the update must be processed first

- By comparison, if we have two friends posting a comment on the same update, the ordering of those comments doesn't matter
  - Note that FIFO is a subset of causal, because for a *single* process the "happened-before" relationship would always exist between its sent messages; but causal ordering also applies selectively *between* processes

Message ordering // total

- This approach ensures that all receivers in a group always process **all** messages, from all senders, in exactly the same order

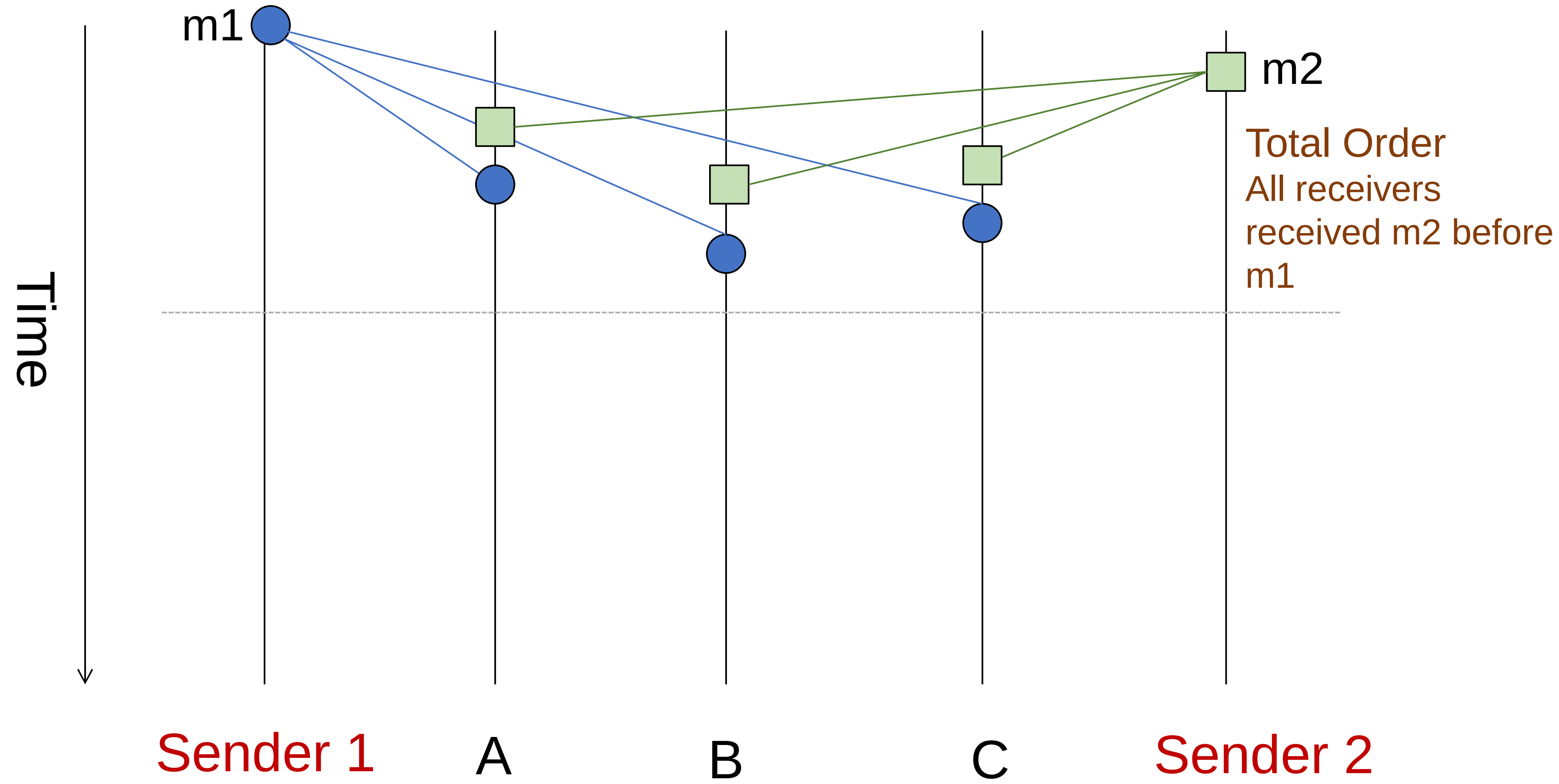
- If we have two messages **`m1`** and **`m2`** from two different senders, every receiver will either process (**`m1`** then **`m2`**), or will all process in the order (**`m2`** then **`m1`**)

Message ordering // total

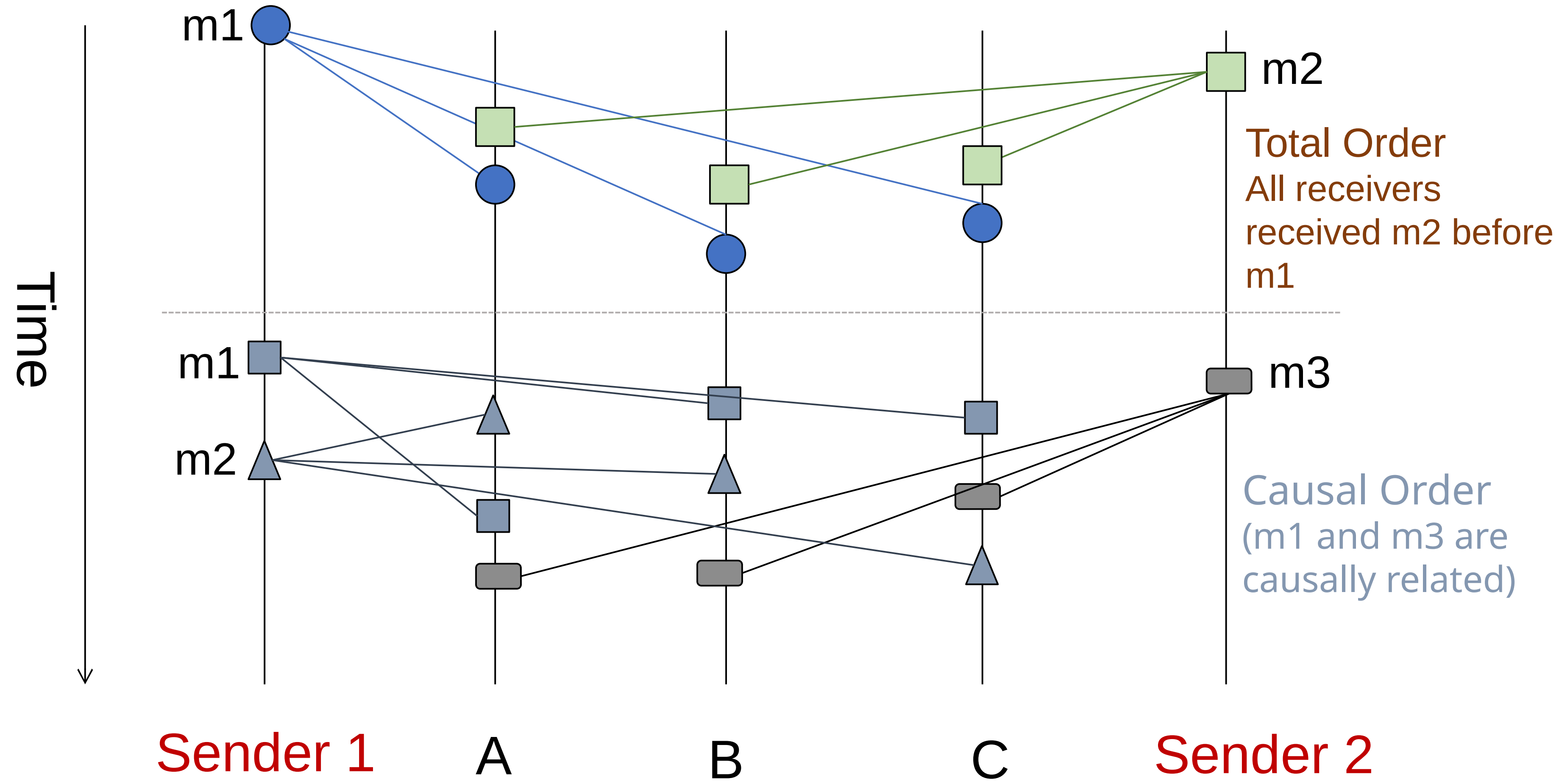
- One approach to achieving this is to use an elected leader which decides for everyone on the ordering of messages
  - The leader is one member of the receiver group, and nobody else in the group processes any receiver messages until the leader tells them which order to use

- This algorithm is often used in replicated databases, but effectively enforces a centralised global lock on the group; it’s therefore relatively slow (we avoid using total ordering if we don't really need it)
  - It requires us to be able to decide on a leader, and change leader if it fails

Message ordering // total vs causal example



Message ordering // total vs causal example



When to use which ordering?

- One consideration is whether the application really cares (e.g. group chat messages occasionally arriving out of order for different group members)
- A more formal criteria is **commutativity** of operations in a state machine:

  new_state = state_update_function(current_state, operation)

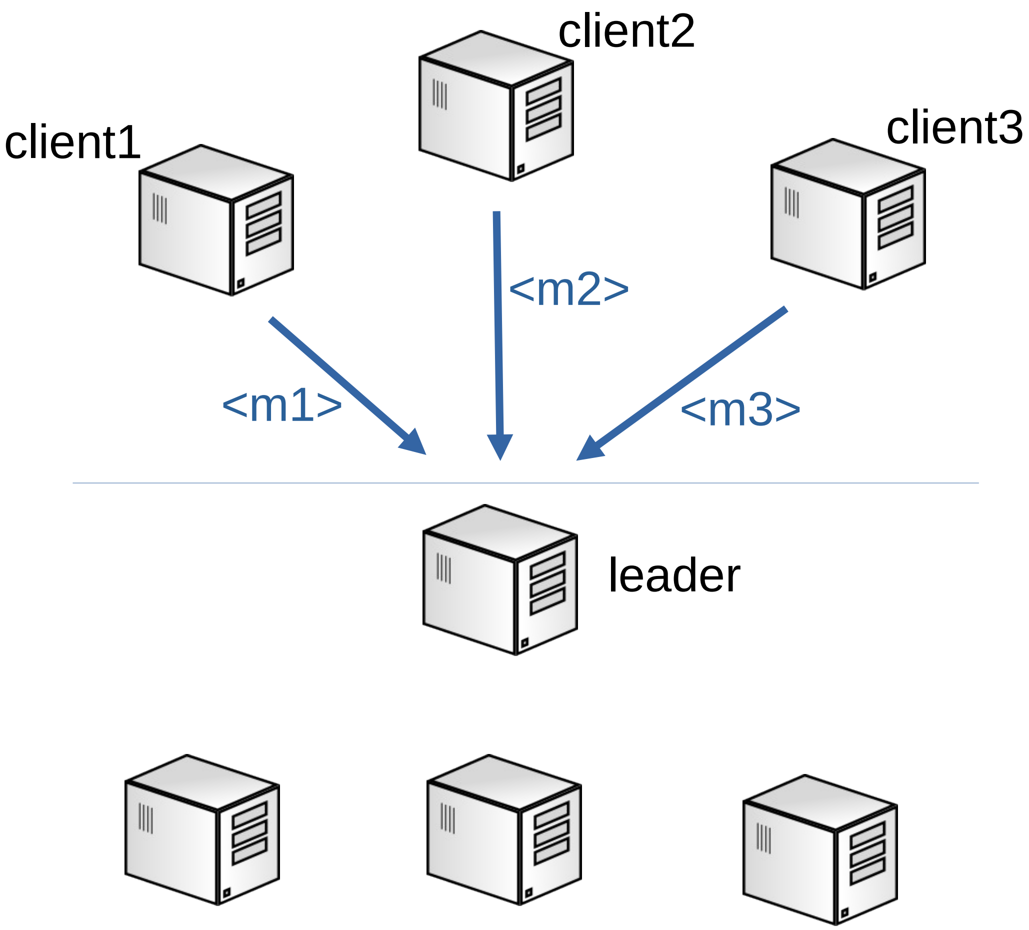
  commutative *is the property that the result is the same regardless of the ordering: c * d == d * c*

Message Ordering	State update function
Unordered	All operations are commutative
FIFO	Operations by different processes (senders) are commutative
Causal	Non-causal operations are commutative
Total Order	Operations are not commutative

Total ordering with no failures is easy

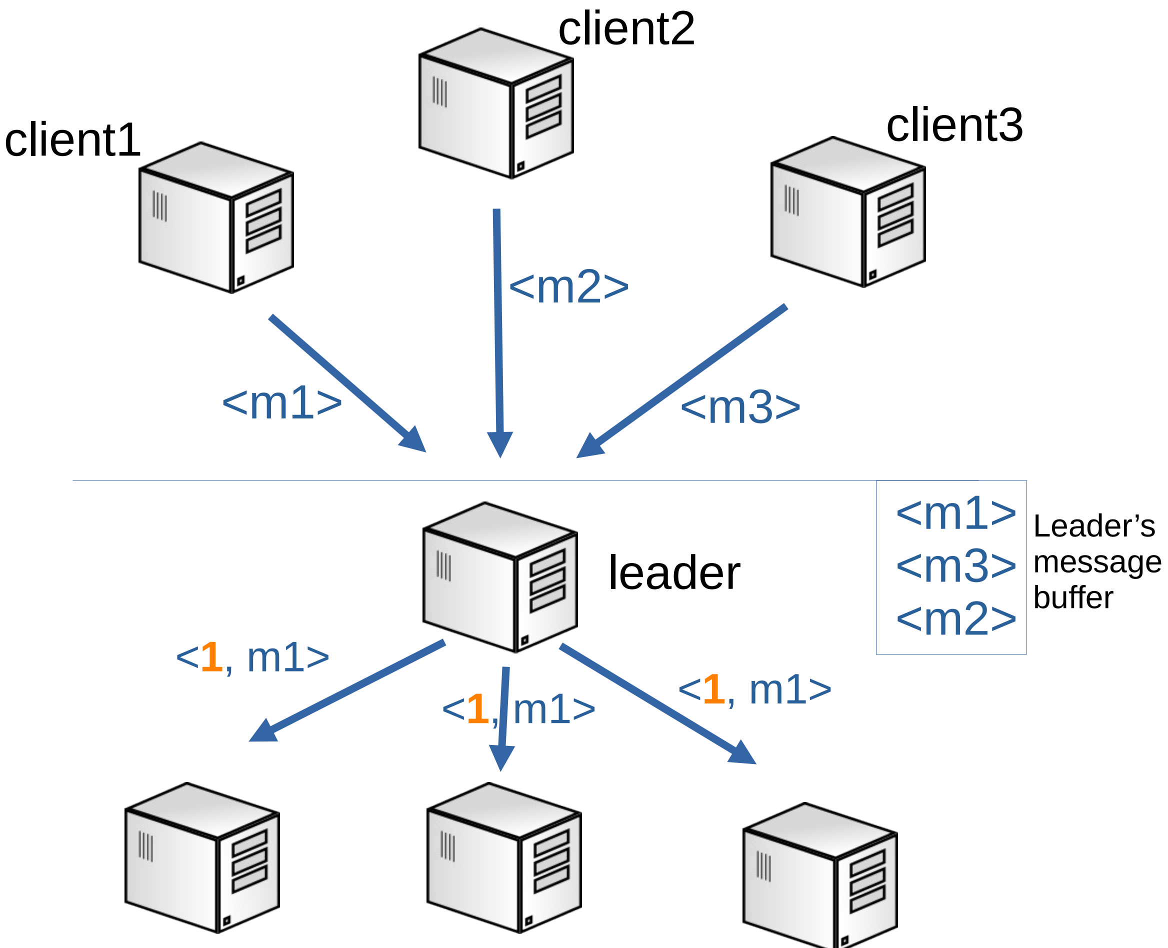
- We can use a leader to decide on the correct ordering for everyone in the group

- To send a message to a group:
  - Send message only to leader
  - Leader uses FIFO multicast to send to the group



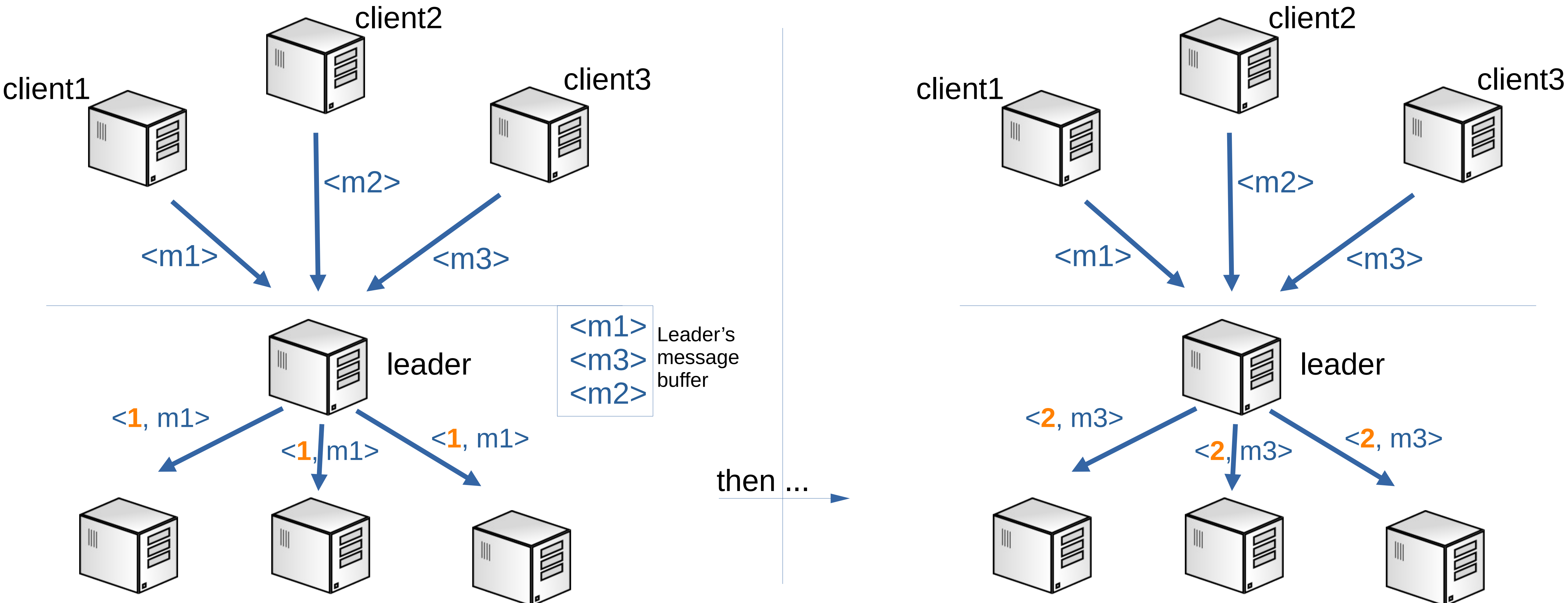
Total ordering with no failures is easy

- Our leader uses a logical timestamp to order messages

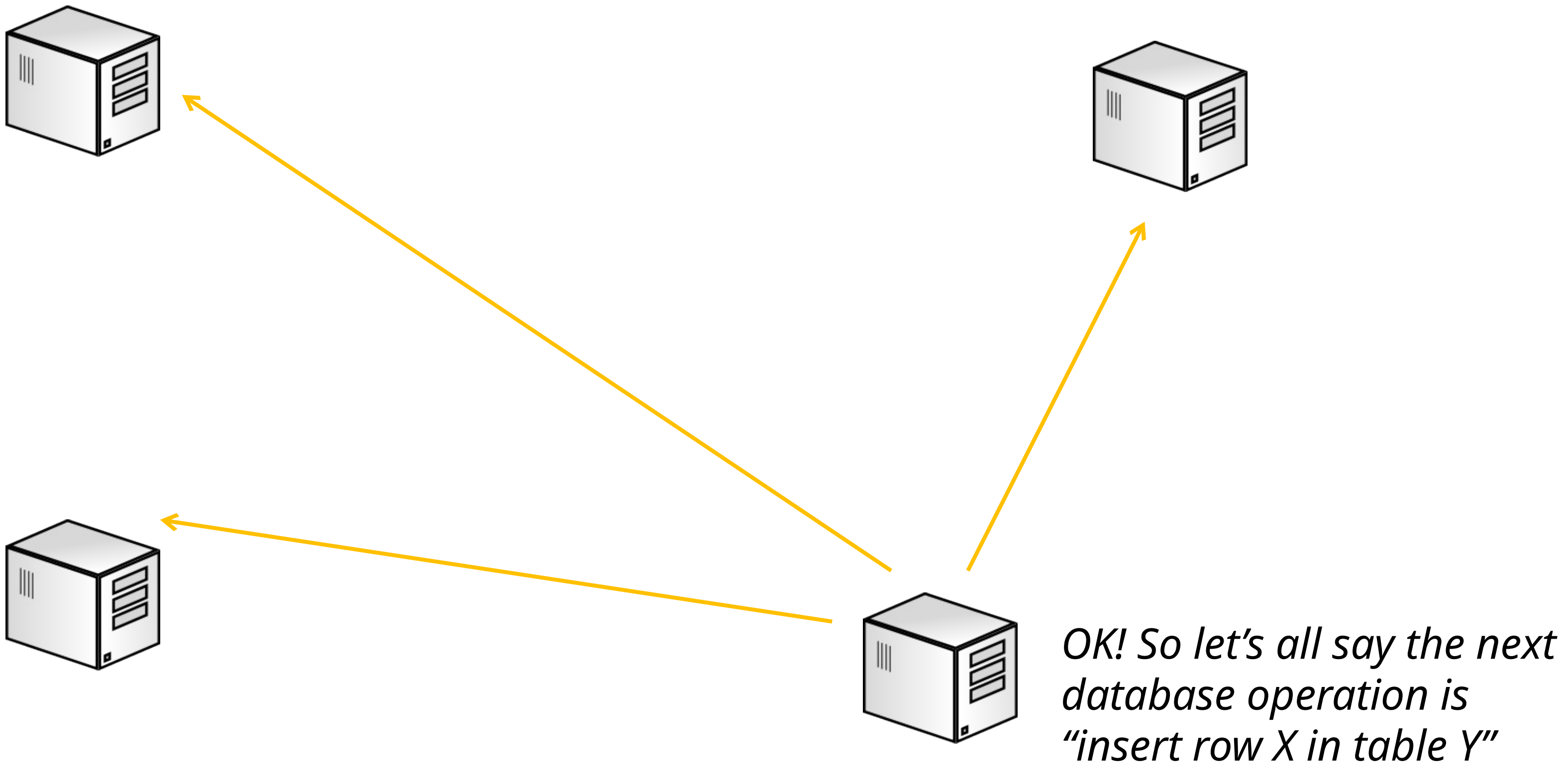


Total ordering with no failures is easy

- Our leader uses a logical timestamp to order messages

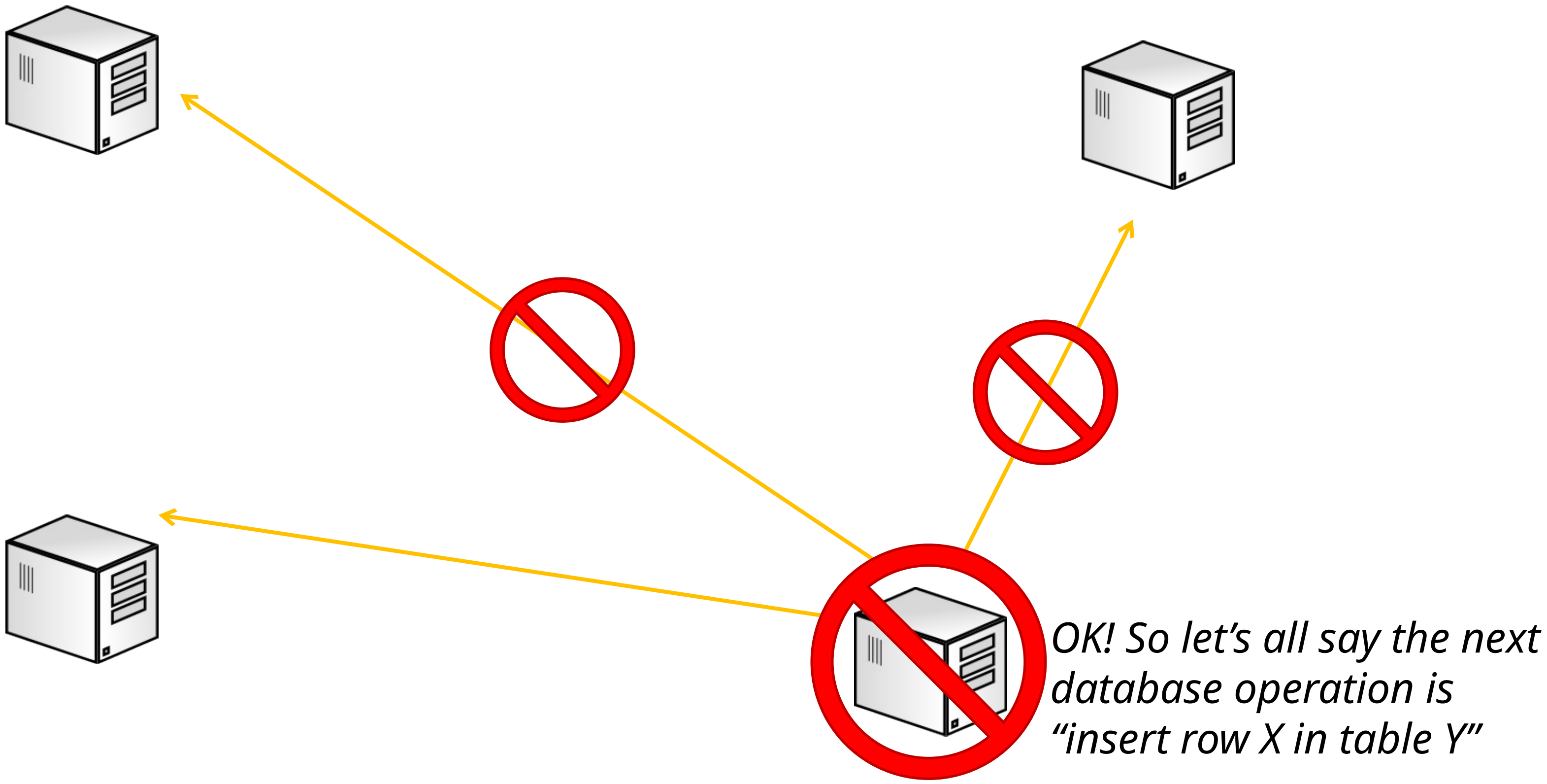


The problem…

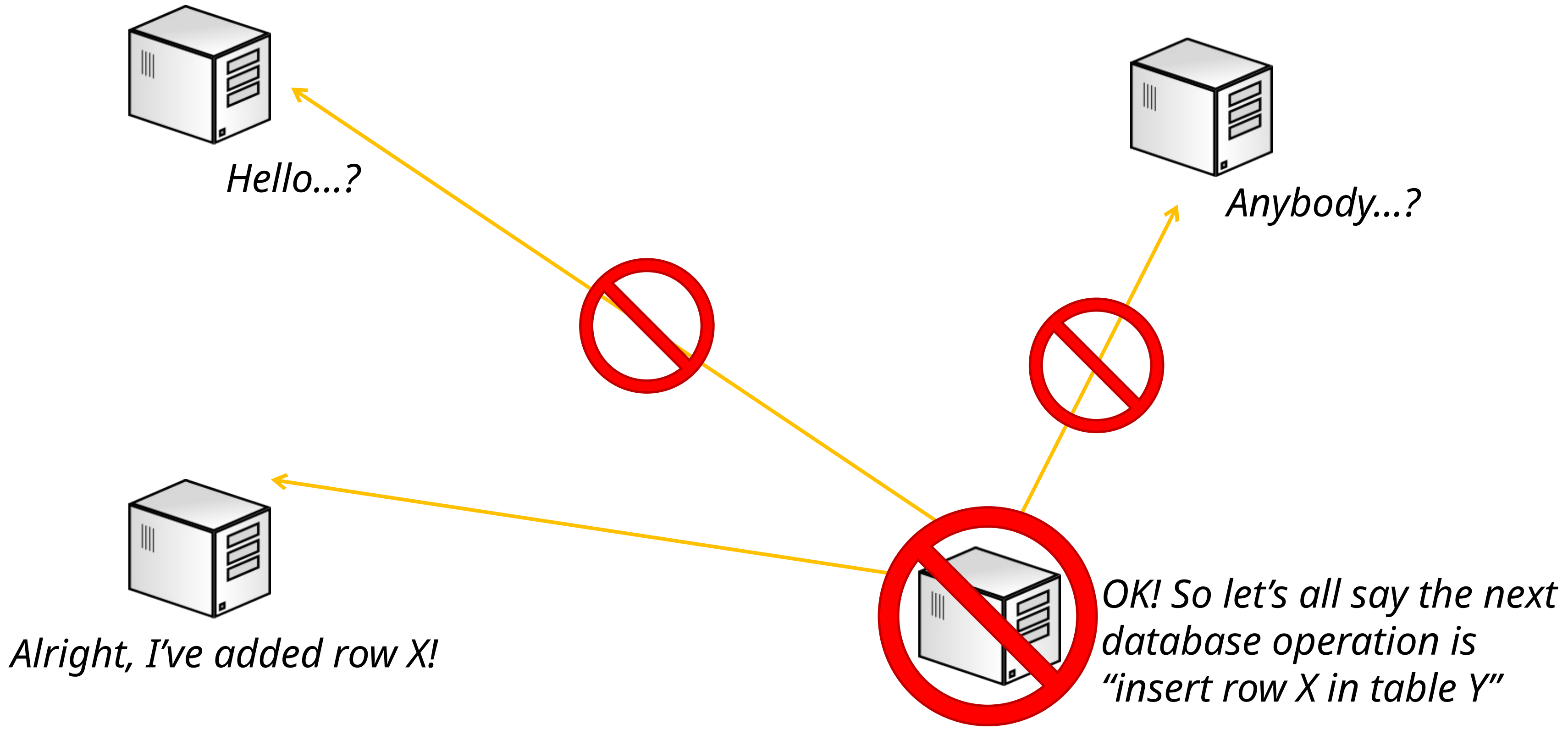


OK! So let's all say the next database operation is "insert row X in table Y"

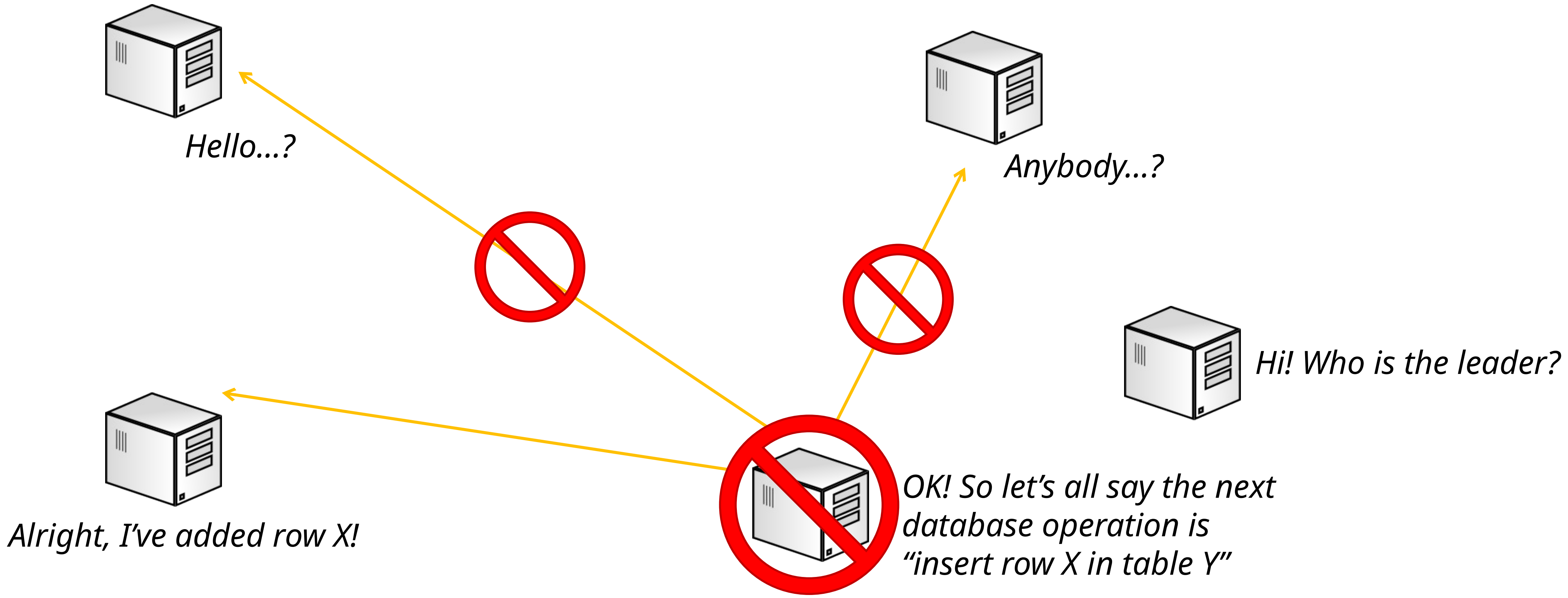
The problem…


OK! So let's all say the next
database operation is
"insert row X in table Y"

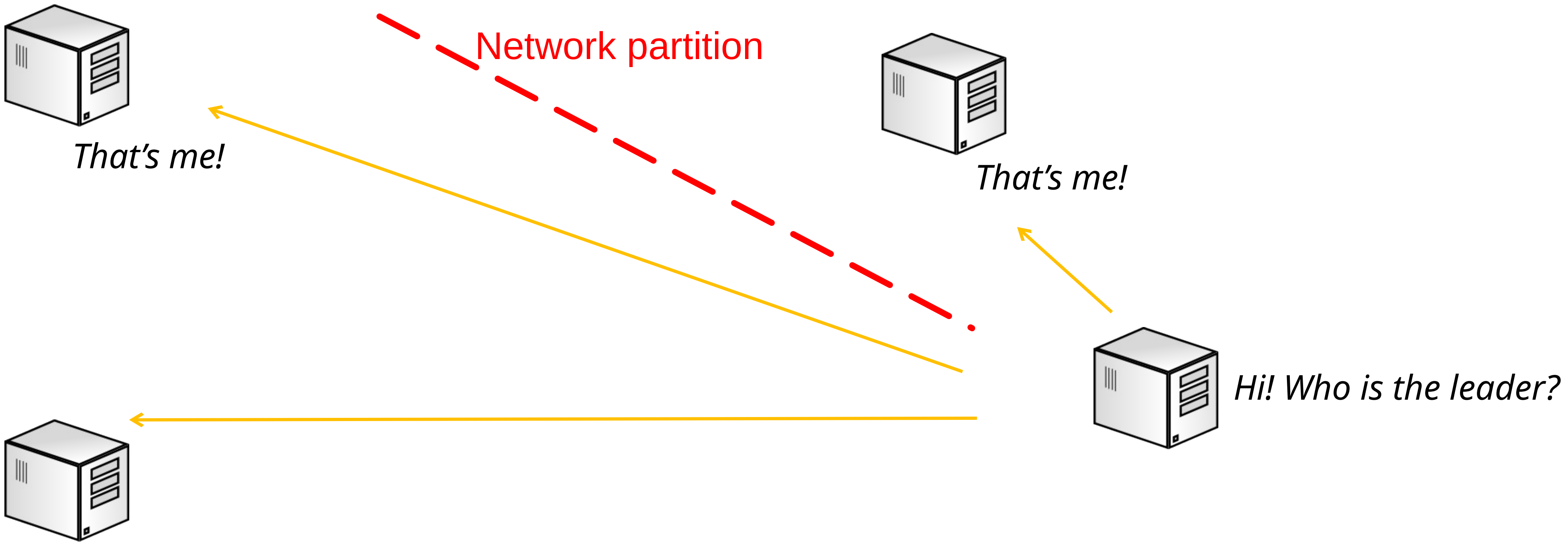
The problem…


Hello…?
Anybody…?
Alright, I've added row X!
OK! So let's all say the next database operation is "insert row X in table Y"

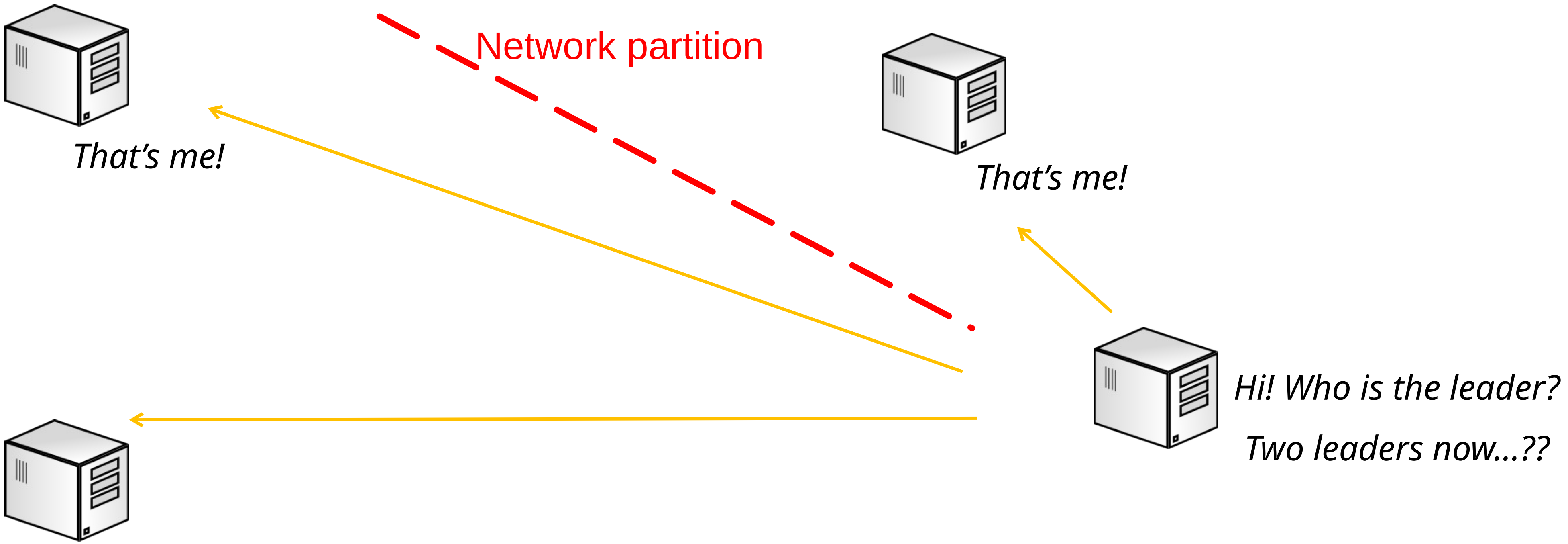
The problem…


Hello…?
Anybody…?
Hi! Who is the leader?
Alright, I've added row X!
OK! So let's all say the next database operation is "insert row X in table Y"

The problem…



The problem…



Network partition
That's me!
That's me!
Hi! Who is the leader?
Two leaders now…??

Consensus

- Distributed consensus is the problem of multiple computers needing to **agree on a value** – e.g. which is the primary replica, what order things happened in, which response is correct, who is the leader

- There are two major consensus algorithms in use today:
  - Paxos
  - RAFT

Summary

- Examined message ordering for group communication, and its implications on message processing throughput

- Examined distributed consensus as one way of agreeing on a value in the presence of crash failures, such as deciding on who is the leader of a group

Further reading

- Section 7.5 (Distributed Commit) and 7.6 (Recovery) of Tanenbaum & van Steen; Sections 16 & 17 of Coulouris & al
- Chapter 7. Fault Tolerance of Tanenbaum & van Steen; Chapter 18 Coulouris & et. Al
- The part-time parliament, L. LAMPORT, ACM Transactions on Computer Systems, 1998
- Paxos Made Live - An Engineering Perspective, Chandra et al., PODC 2007