Distributed consensus and fault tolerance

Lecture 1 / 2

Georgios Bitzes, CERN
iCSC 2017

Let's talk about distributed systems

Lecture 1

- Introduction to distributed systems
- Replication and split-brain
- Strong vs eventual consistency
- The raft consensus algorithm

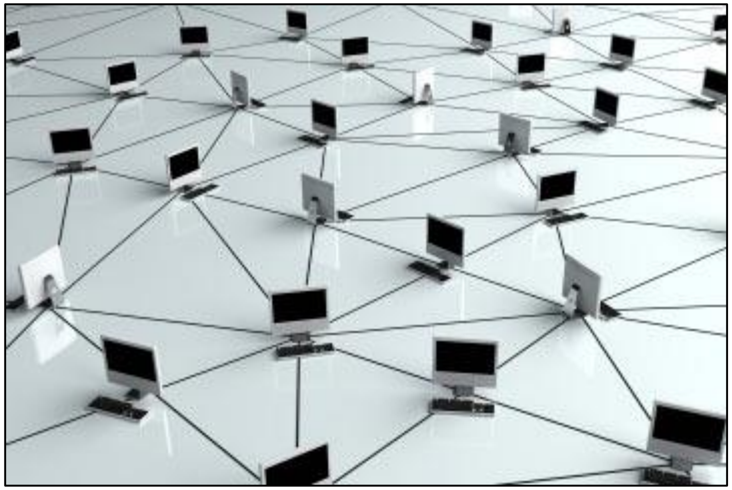
Lecture 2

- Two Generals, Byzantine Generals
- Byzantine fault tolerance
- Bitcoin and blockchain consensus

Introduction to distributed systems

Distributed systems are all around us

- Infrastructure:
  - Networking, routing algorithms
  - Flight control systems
  - Banking systems, ATMs



- Internet services: running these on a single machine would be unthinkable
  - Facebook, Twitter
  - Google search, gmail
  - Github

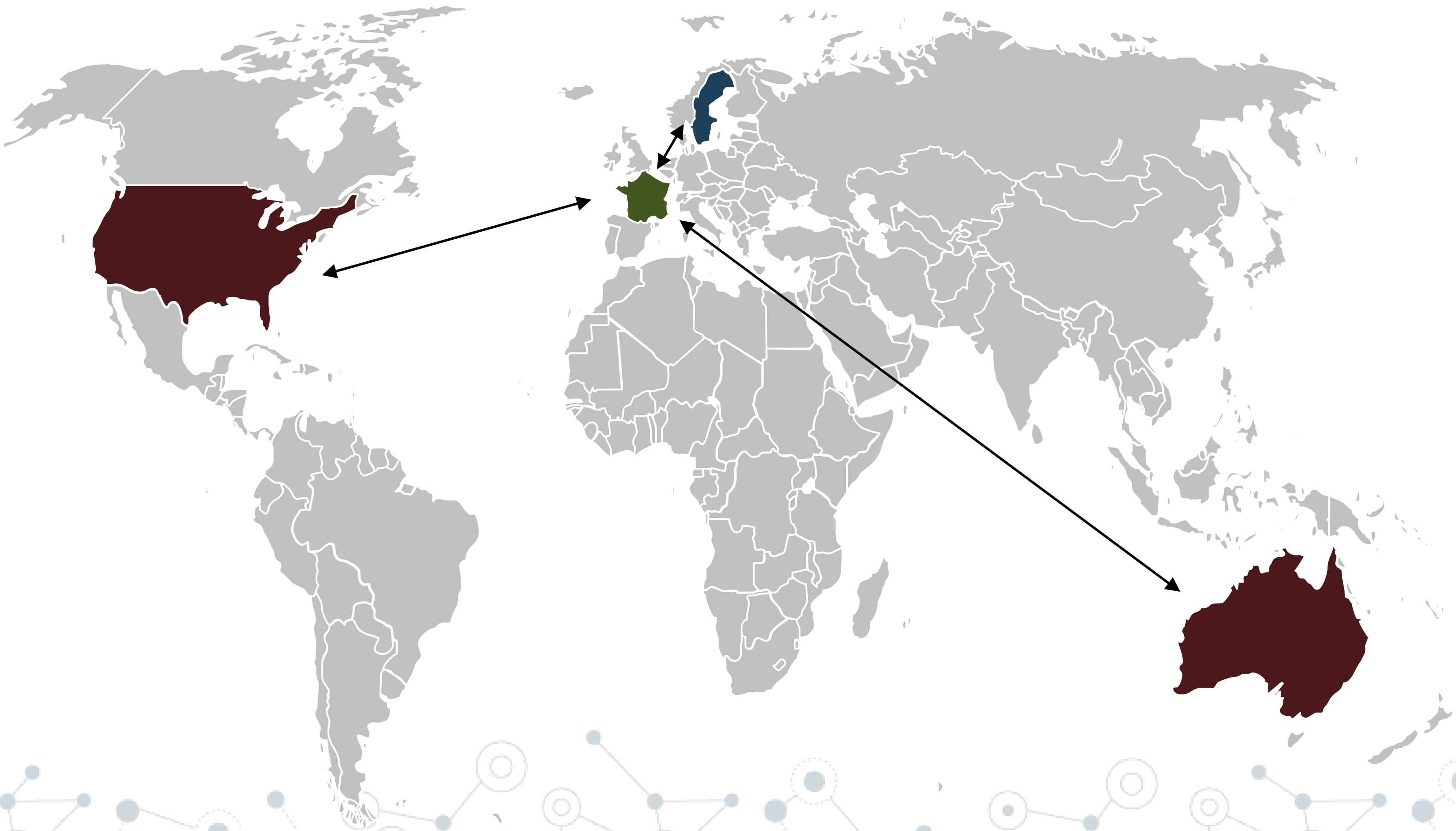
- But the need for distributed systems appears long before we have to scale to millions of users

Scalability

- As the load increases, one server no longer enough to serve all clients
- We’re bound to hit some bottleneck
  - CPU power
  - Memory size
  - Memory bandwidth
  - IO operations per second
  - Network bandwidth
  - Software, more often than not cannot fully exploit our powerful hardware

Latency

With only a single server, some clients will suffer from high latency



Fault tolerance

- Our system has a <u>single point of failure</u> – one faulty hard drive and the service goes down

- Downtime of certain critical services can cause great disruption: credit card processing, air traffic control, or… stackoverflow.com

Distributed systems to the rescue

- A distributed system solves the above problems nicely
  - **Scalable performance:** add more machines as needed
  - **Lower latencies:** clients connect to the server nearest to them
  - **Fault tolerance:** if a machine goes down, the others can detect it and take over its responsibilities

- **Caveat:** distributed systems add *a lot* of complexity

What is a distributed system?

- Distributed system: A collection of independent computers that appears to its users as a **single**, **coherent** system

- Nodes coordinate by **exchanging messages** through the network

Distributed system: added complexity

Making a system distributed adds some complexity:

- **Unreliable network:** messages get lost, delayed, re-ordered, corrupted
- **Node failures:** the more machines we have, the higher the probability of some failing
- **Latency:** communication far slower than using local shared memory or IPC
- **Limited bandwidth**
- **Security:** “is this message really coming from who I think it is” ?

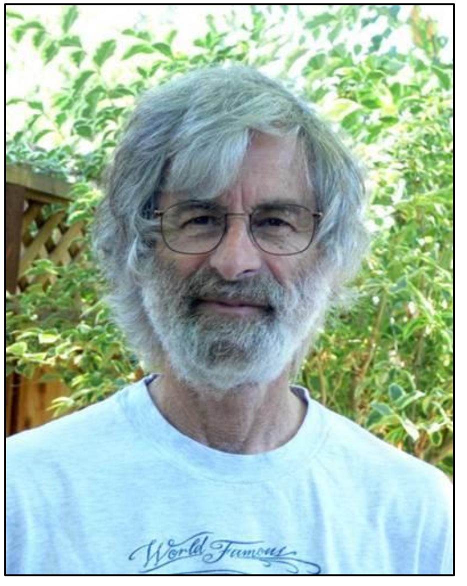
Multiple single points of failure?

- What if every node depended on every other for its correct operation?

- A single failure will bring them all down
- Just because a system is distributed, doesn't mean it's fault tolerant

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Leslie Lamport



Replication and split-brain

Replication as a means to fault tolerance

- A database machine holding 25% of all our data crashes and burns – hard drive is **unrecoverable**
- What happens next? Some possibilities…

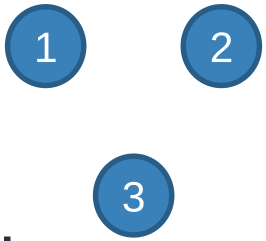
1. No backups, *data lost forever* – oops. Not acceptable, will create bad publicity, erode user trust
2. A backup taken the previous day is manually restored. Much better, but new user data in the last 24h is lost, and *long downtime* during manual restore
3. A replicated database machine takes over immediately after the failure – no data loss, no downtime, *users don't even notice*

Key-value stores (used in future examples)

- Simple database, clients can perform 2 operations
  - **write** a value into a key
  - **read** it back.

```
127.0.0.1:6379> SET favorite_food pickles
OK
127.0.0.1:6379> GET favorite_food
"pickles"
127.0.0.1:6379> SET favorite_language c++
OK
127.0.0.1:6379> GET favorite_language
"c++"
```

A naive replication protocol

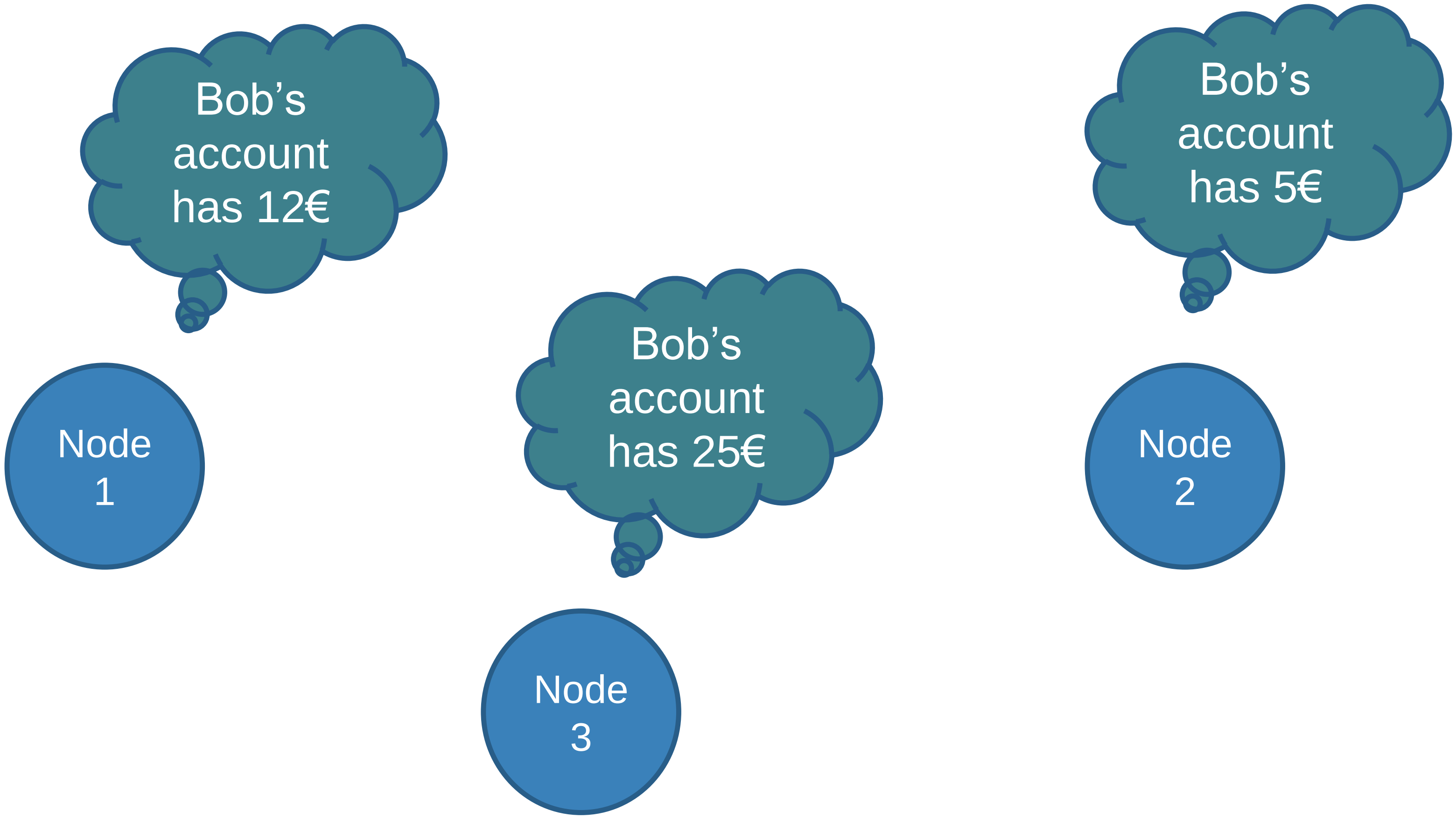


- Assume we want our key-value store replicated on 3 nodes.
- Let’s invent our own simple replication protocol:
  1. A client sends a write request: propagate the change to all other sibling nodes. Ignore errors.
  2. A client does a read: give back the local value stored on the contacted node
  3. On receipt of a propagated change from a sibling node: simply apply it by updating the local value.

A naive replication protocol (2)

- In simple cases and good network conditions, this protocol might appear to work.
  - https://gbitzes.github.io/icsc/animation/#naiverep
- What if a node **goes down** for maintenance for 5 minutes?
  - All writes within that window are not replicated onto it
- What if certain nodes receive the updates in a **different order**?

The problem: Split brain



The problem: Split brain (2)

- Our naive protocol will **inevitably** lead to split brain
- What we want: replicated nodes to agree on the state of the key-value store, that they all come to a **consensus** about each update
- We'll talk about a correct algorithm later

Failure model

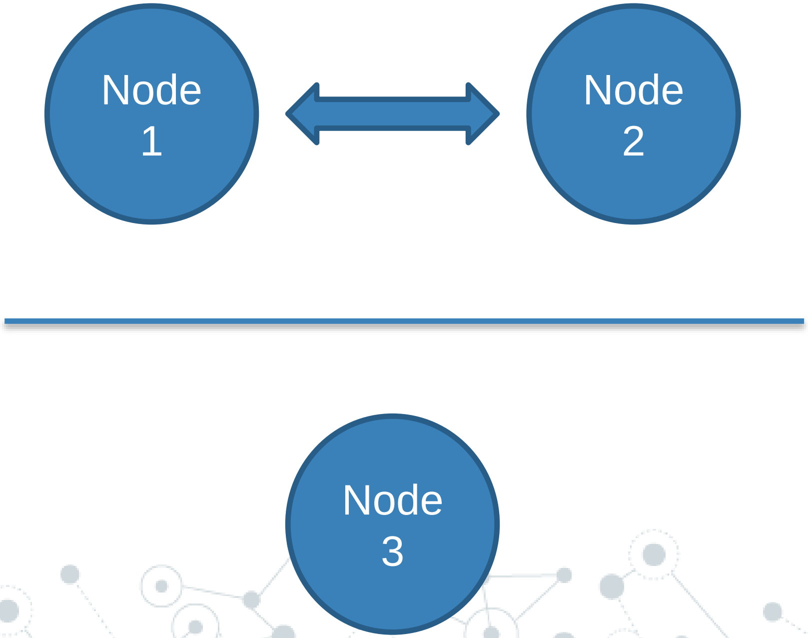
The failure model we'll concern ourselves for now:

1. **Fail-recover faults**
   - A node goes down, stops responding to messages.
   - The other nodes can detect this through timeouts – "if 127.0.0.10 doesn't respond in 100ms, it's down"
   - … but failed nodes can recover
2. **Delayed / lost messages:** Messages between nodes can be arbitrarily delayed or lost, but **not** corrupted

- Later on: **byzantine faults**

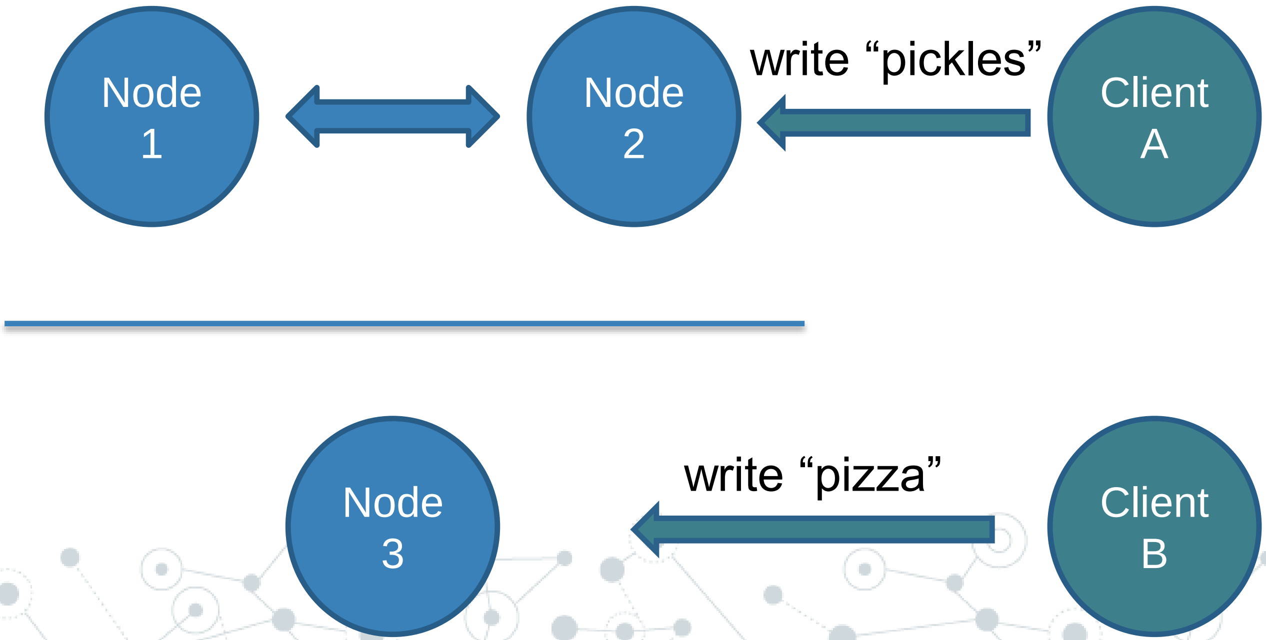
Network partitions

- Tolerance to delayed and lost messages implies tolerance to network partitions
- **Network partition:** parts of the system become disconnected from each other



Network partitions (2)

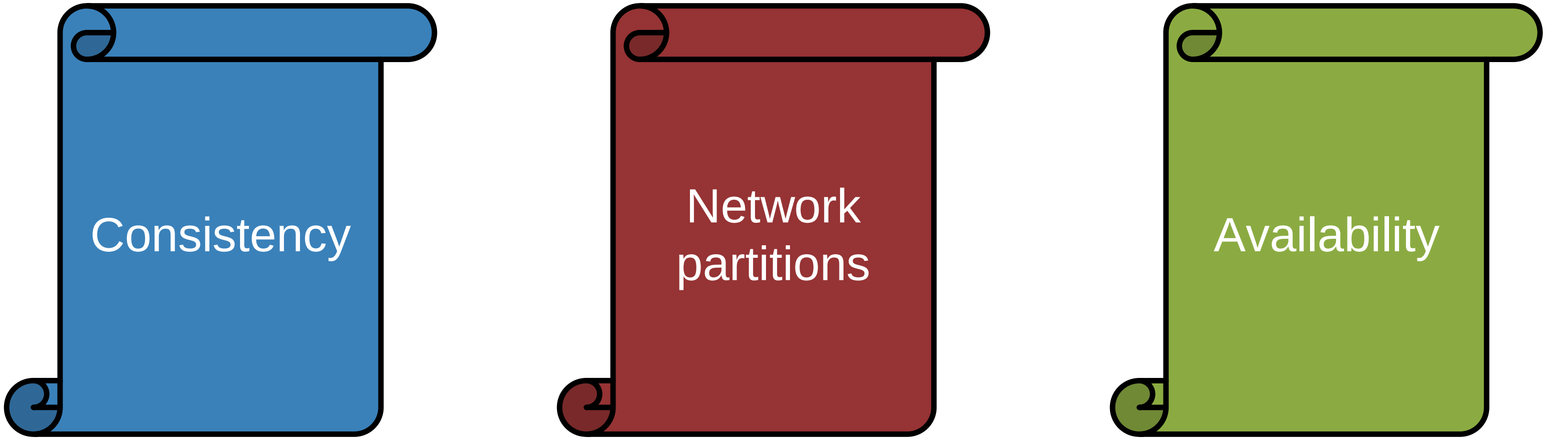
- Problematic because each part is making decisions *independently*
- What if clients try to write different values to each partitioned part? Which value is "correct"?



Strong vs eventual consistency

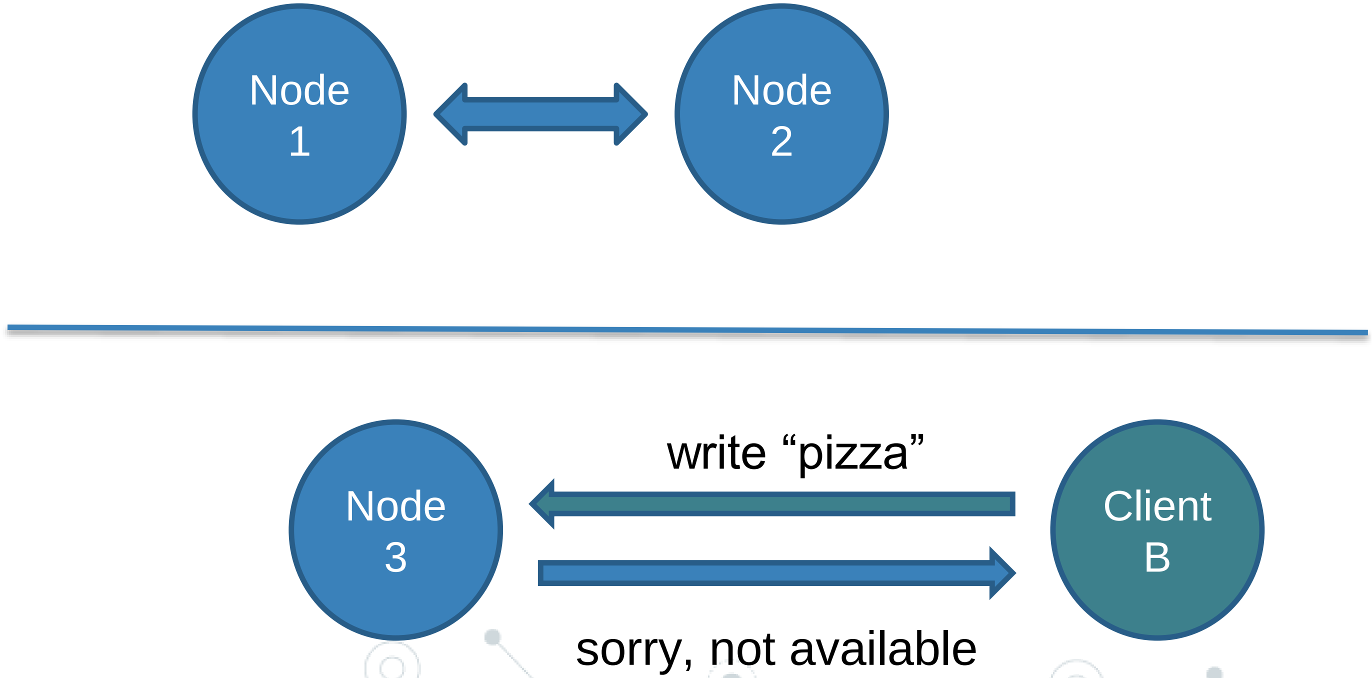
CAP theorem

- When faced with *Network Partitions* (P), a distributed system can be either *Consistent* (C) or *Available* (A)



Strong consistency

- System appears **externally consistent** to clients
- Requests are **refused** if consistency cannot be guaranteed

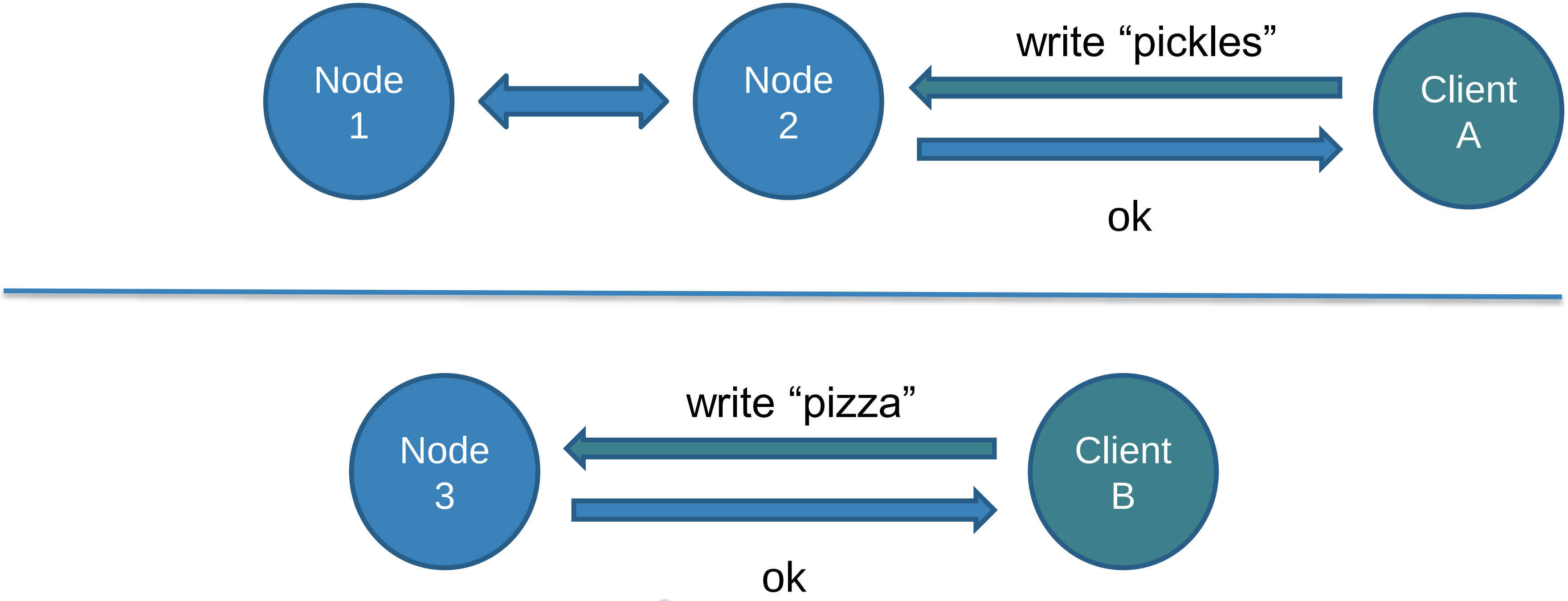


Strong consistency (2)

- Internal, temporary inconsistencies: **inevitable** in distributed systems
- Strong consistency: internal inconsistencies are resolved and **not exposed** to clients
- The cost: **sacrifice** availability in favor of consistency

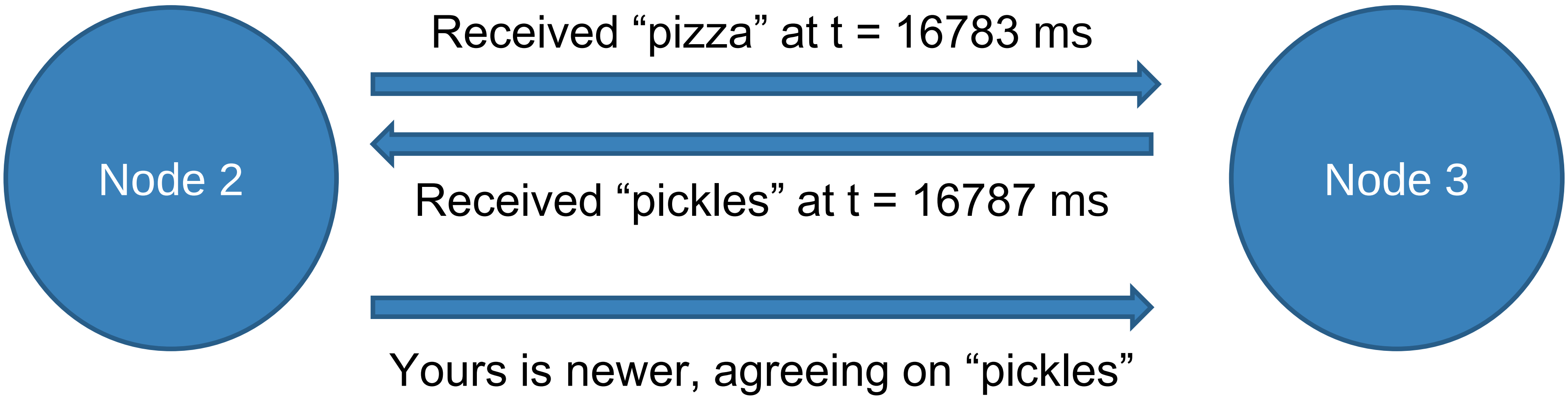
Eventual consistency

- Best-effort: try to be consistent, but without guarantees
- Internal inconsistencies **may be exposed** to clients



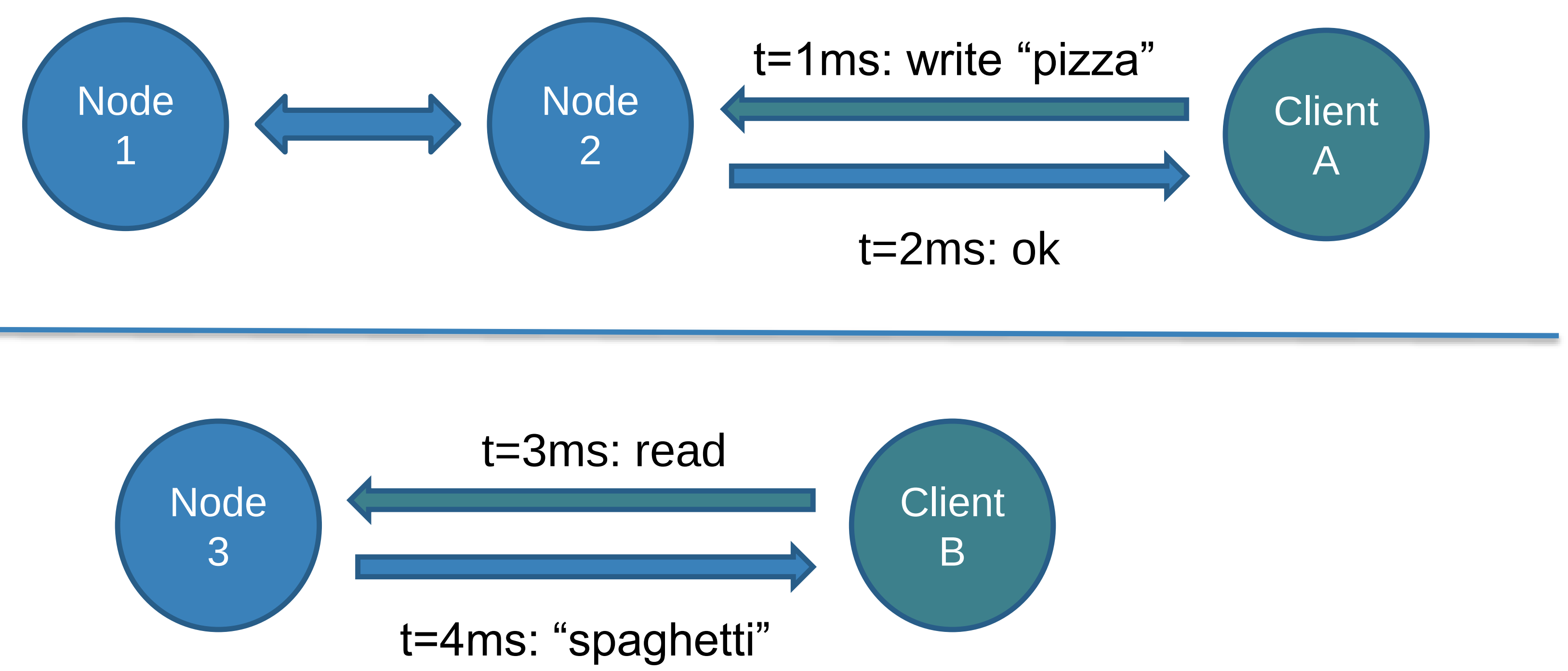
Eventual consistency (2)

- Conflicts are resolved after partition heals
- Common approach: <u>*“last writer wins”*</u>



Eventual consistency (3)

- Clients may receive stale values



Which one is better?

- Depends on the **application**, both are useful
- Eventual consistency: generally more **performant** and **scalable**
  - example: DNS, Amazon S3
- Strong consistency: **safety guarantee**, every read receives the most recent write or error
  - example: certain SQL databases

Question

Which form of consistency does our naive replication protocol provide?

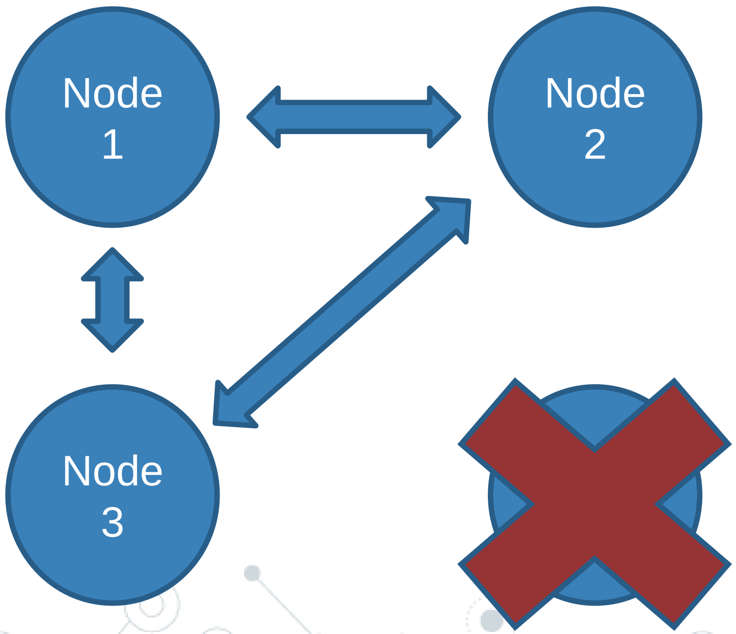
1. Strong consistency
2. Eventual consistency
3. Neither

- Correct answer: neither.
- In our protocol, inconsistencies are never resolved – not even “eventually”

The raft consensus algorithm

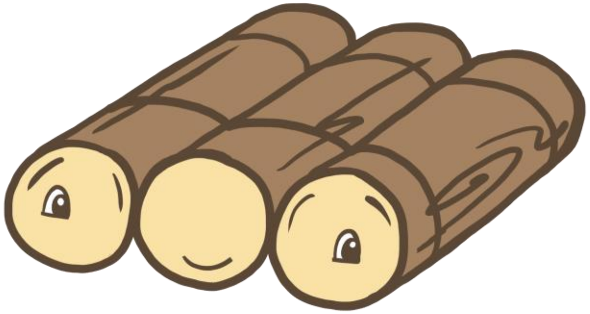
Distributed consensus

- A number of nodes come to an **agreement** about a **value**
- Any node can **crash** or **recover** at any time



The raft consensus algorithm

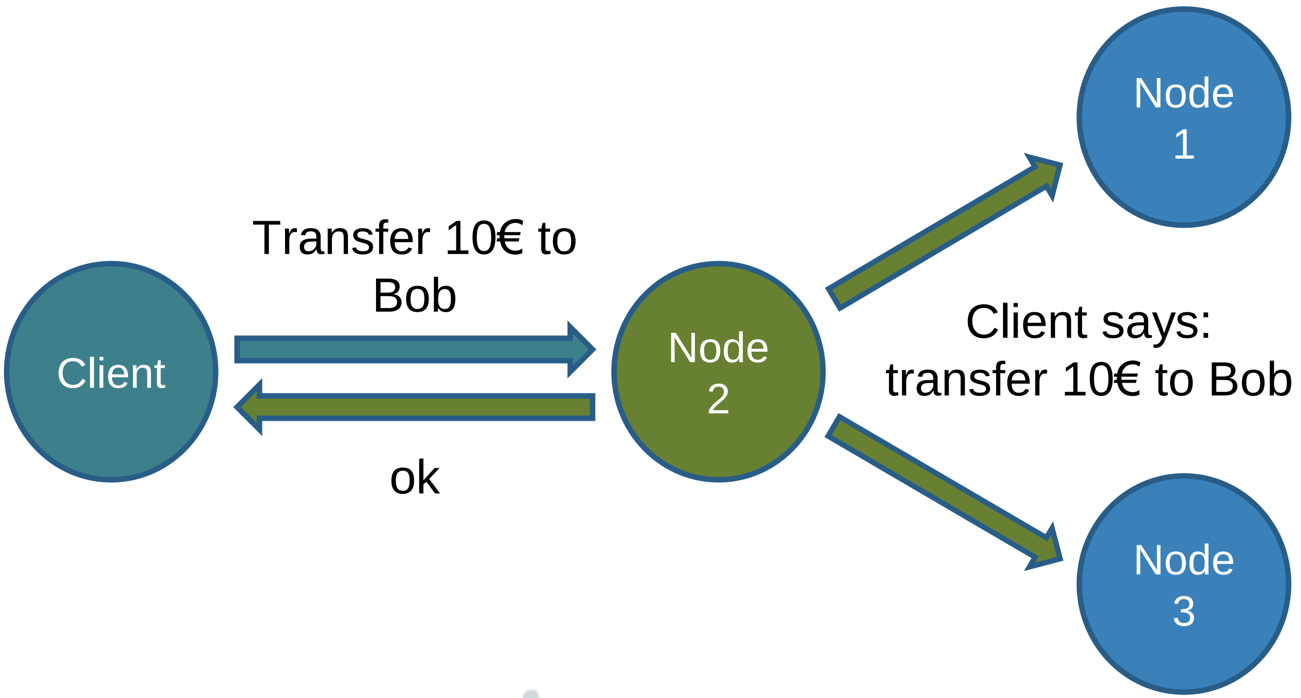
- Raft is a proven, correct consensus algorithm, most suitable for strongly consistent system



- Other consensus algorithms also exist – most notably paxos

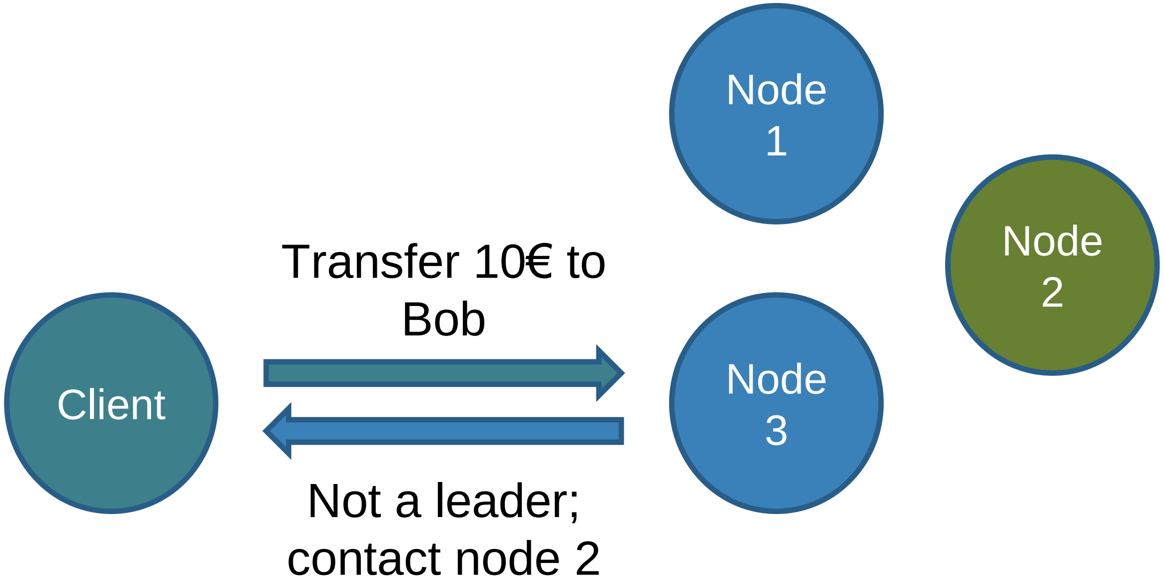
Master – slave replication

- One of the nodes is elected to become the *master* (or *leader*)



Master – slave replication (2)

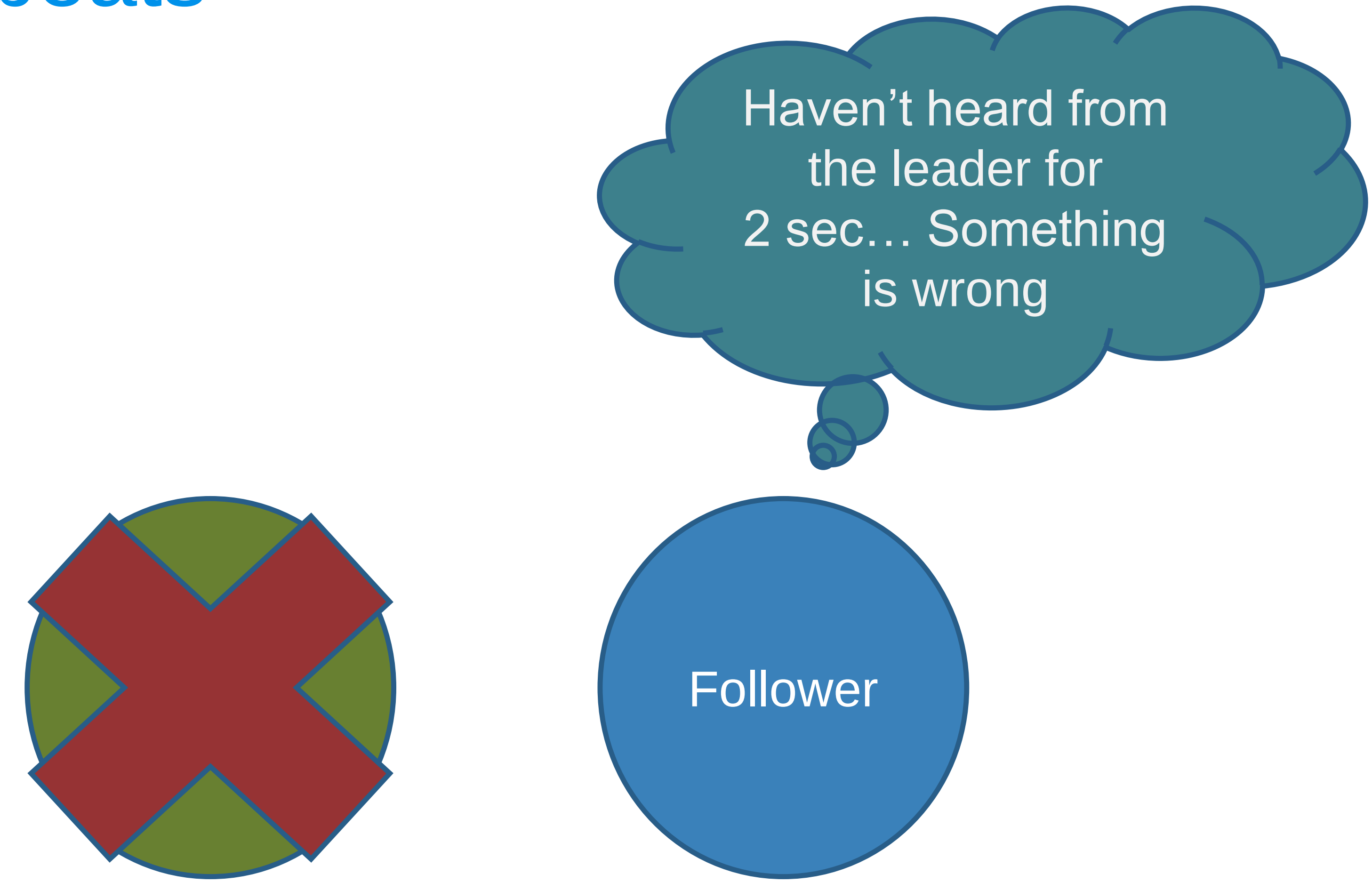
- Easy way to give a **global ordering** to writes from clients – can reconcile simultaneous updates
- Clients contacting followers are **redirected**



Leader failover

- The leader sends regular heartbeats to all followers
- If a follower stops receiving heartbeats, it assumes leader failure and triggers an **election**
- An election is won if a node receives positive votes from at least a **majority** of the cluster

Heartbeats

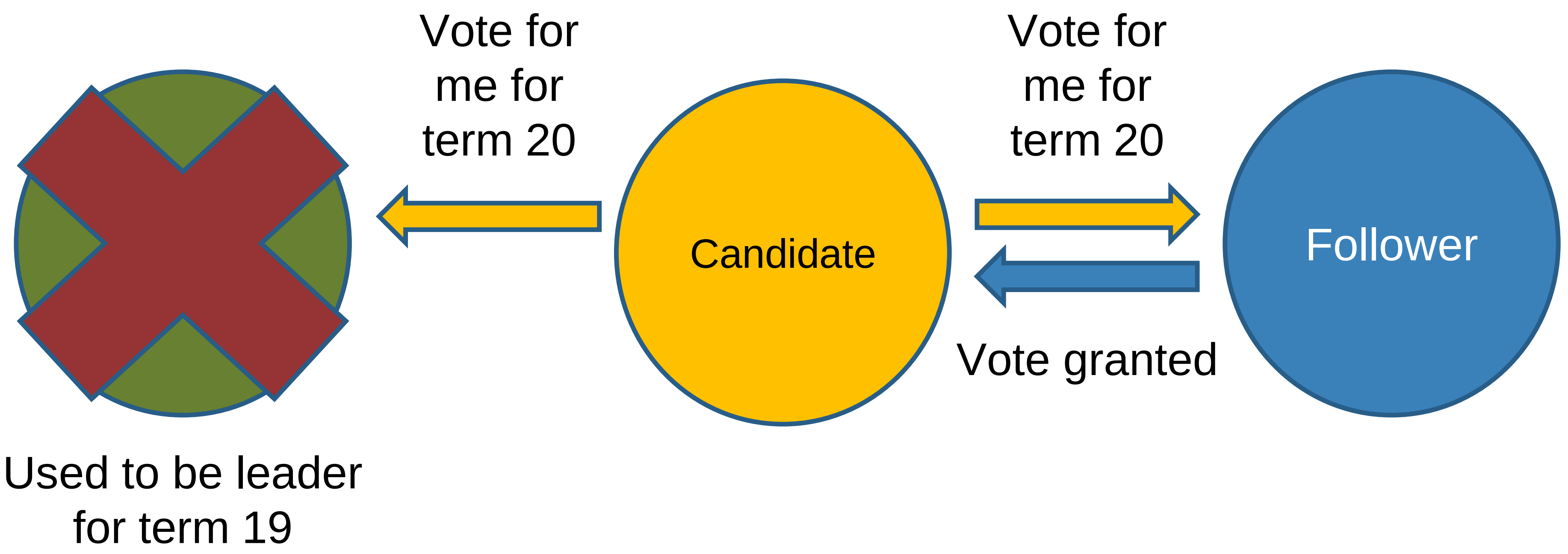

Haven’t heard from the leader for 2 sec… Something is wrong
Follower

Election terms

- A leader is elected for a specific **election term** based on **majority vote**
- Only **one** leader can be elected per term
- Possible to have multiple leaders at a time, due to network partitions – all will have **different** terms
- The leader with the **highest** term “wins” and can override decisions made by the others

Leader election

A successful election: 2 out of 3 nodes agree on the new leader

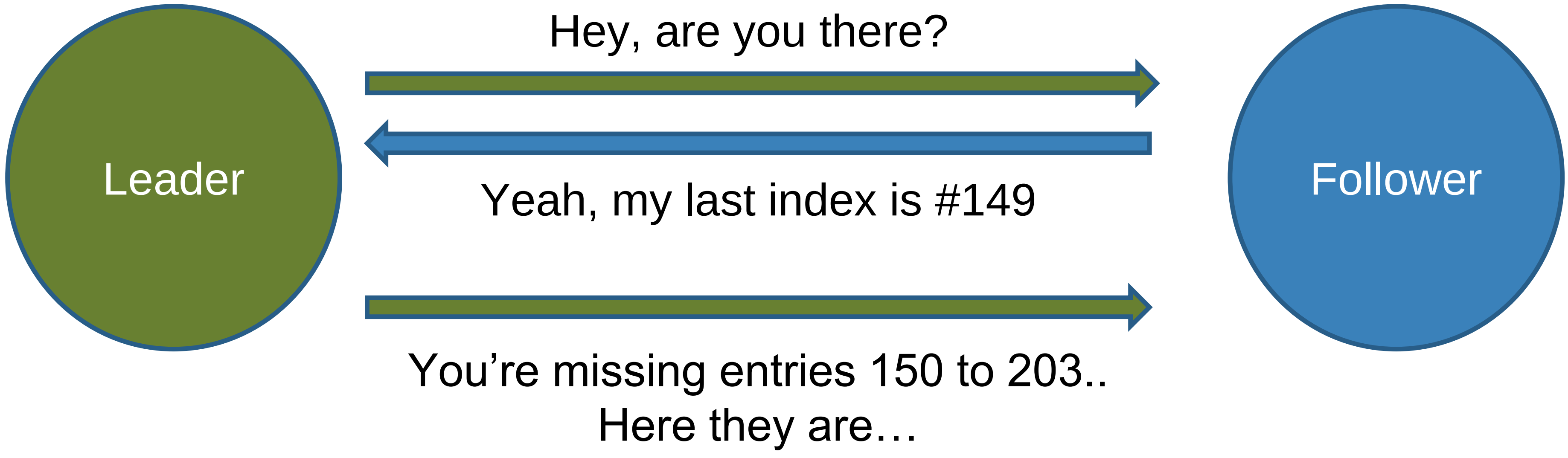


Bringing outdated nodes up-to-date

- One of the followers goes offline for 10 minutes – how to bring it up-to-date?
- Record *<u>all writes</u>* into an indexed log, and replicate it
- Contains also the term during which the entry was recorded

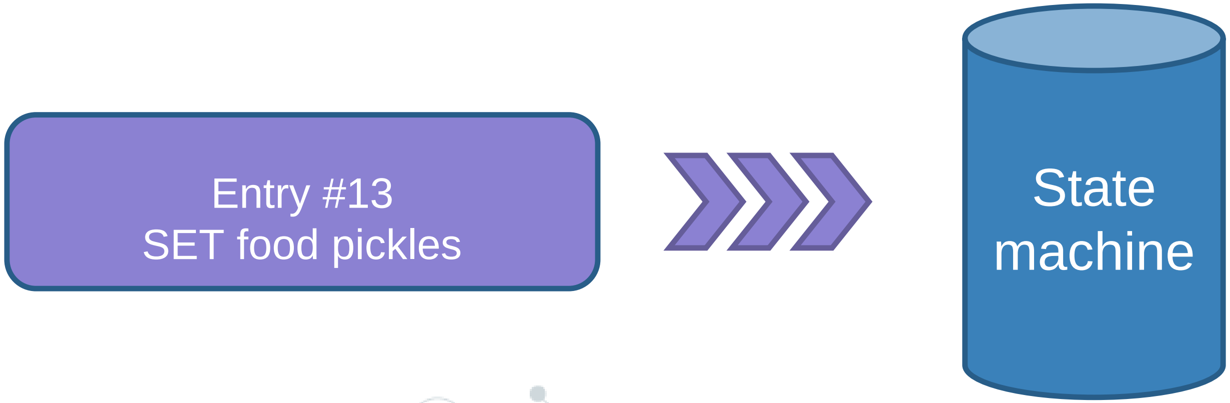
Index	Term	Contents
0	1	SET food pizza
1	1	SET language c++
2	1	SET food pickles
3	5	SET answer_to_life 42
…		….

Log replication



Application of log entries

- Log entries represent *changes* to the database (also called the *state machine*)
- At some point, these changes have to be *applied* to the state machine



State machine replication

- Not safe to apply an entry immediately, since in rare cases it might get **rolled back** by a subsequent leader
- Log entries are considered **committed** once a majority of nodes have them
- Once an entry is committed, it’s guaranteed that it won’t be rolled back, and can be *safely applied* to the state machine
- Only nodes that have *all committed entries* can ever succeed during a leader election

Summary

- Distributed systems are ***necessary*** to provide scalability, low latency, and fault tolerance
- Replicated nodes can offer transparent failover, but be careful of *split brain*
- Several approaches to replication, need to compromise between *consistency*, *availability*, *scalability*, and more
- The *Raft consensus algorithm* offers a good basis for a strongly consistent distributed system

Bedtime reading

- PACELC theorem, extension to CAP theorem
  https://en.wikipedia.org/wiki/PACELC_theorem

- The raft paper
  https://raft.github.io/raft.pdf

- Paxos algorithm
  https://en.wikipedia.org/wiki/Paxos_(computer_science)

- Strong consistency models
  https://aphyr.com/posts/313-strong-consistency-models

Thanks

Any questions?

See you in Lecture 2

Georgios.bitzes@cern.ch