CS 686: Special Topics in Big Data

Distributed Consensus

Lecture 12

There are only two hard problems in distributed systems:

2. Exactly-once delivery
1. Guaranteed order of messages
2. Exactly-once delivery

-- Mathias Verraes

The Great Unknown

- It's hard to be sure about anything
  - True in general, but even more true with distributed systems
- Is a node down, or is the network slow?
- Did we shut the service down, or did it crash?
- Is the system in a steady state?
- If a network breaks into partitions and nobody is around to hear it, does it make a sound?

Today’s Agenda

- Replication and Failures
- Conflict Resolution
- Consensus Algorithms
- Transactions

Today's Agenda

- **Replication and Failures**
- Conflict Resolution
- Consensus Algorithms
- Transactions

Replication

- Maintaining replicas is a great way to make our systems resilient to failures

- We can also leverage replicas as a cache to improve performance
  - If a node is closer, has less load, etc. then we can use it instead of the original copy

Managing Replicas

- Any time we start replicating data across multiple machines, things start to get complicated
- What happens when the replicas get modified at the same time?
  - Vector clocks: one solution we saw from Dynamo
- Another approach is providing distributed **transaction** support
  - Downside: latency

Reaching Consensus

- Solving this problem with replicas is just one example of coming to a **consensus** in distributed systems
- Some other examples:
  - Clock synchronization, broadcasting, leader election
- Reaching a consensus can be difficult due to:
  - Heterogeneity
  - Geography (…latency)
  - Hardware **and** software failures

CAP Theorem (1/3)

- Deals with the guarantees that can be provided by distributed systems, especially during failures
- Observed by Eric Brewer
  - Co-founder of Inktomi
    - Search engine tech, ISP software
  - Professor at UC Berkeley
- Later formalized in 2002 with a proof by Gilbert and Lynch
  - *Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-tolerant Web Services.* SIGACT. 2002.

CAP Theorem (2/3)

- **C**onsistency:

  All nodes see the same data.

- **A**vailability:

  A partial failure does not stop the system.

- **P**artition Tolerance:

  The system can handle network partitions.

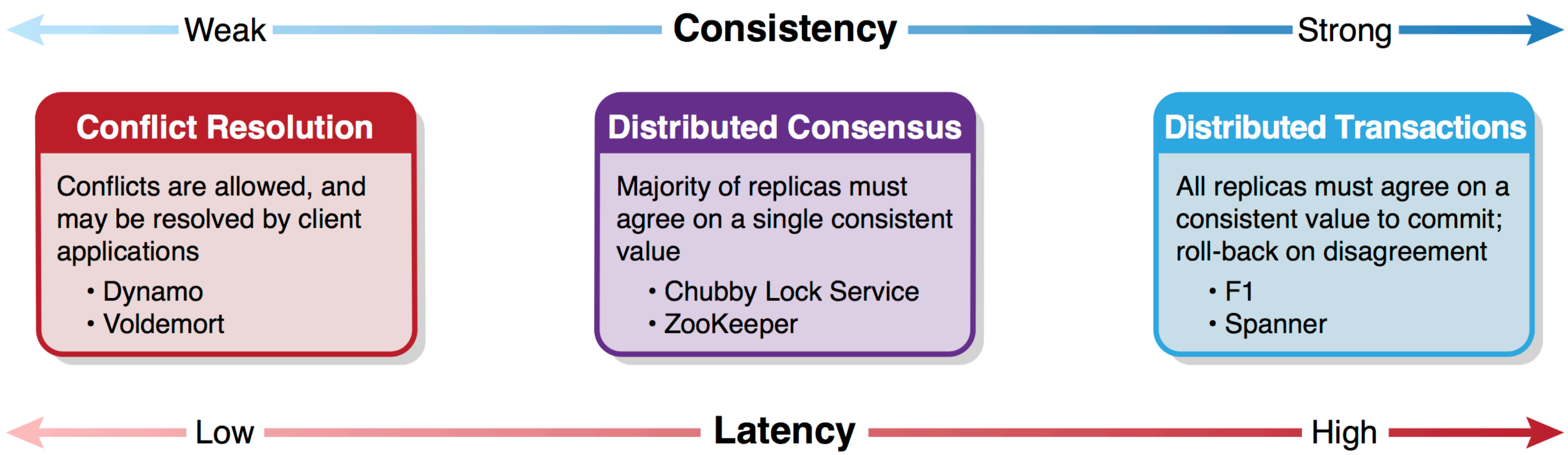
CAP Theorem (3/3)

- **Important**: this isn't a "pick two of the three" kind of situation
  - A mistake that is made frequently
- Rather, the CAP theorem describes what a system does when it encounters a network failure (partition)
- If everything is operating normally, the system can provide both high availability **and** consistency

CAP Classifications

- AP systems: highly available
  - Can result in inconsistent views of the dataset
  - Shopping cart
- CP systems: highly consistent
  - **Can** experience downtime if a partition occurs
    - That's okay, because we're assuming it's better to be offline than cause inconsistencies!
  - Billing system

Consistency-Latency Tradeoff



Today's Agenda

- Replication and Failures
- **Conflict Resolution**
- Consensus Algorithms
- Transactions

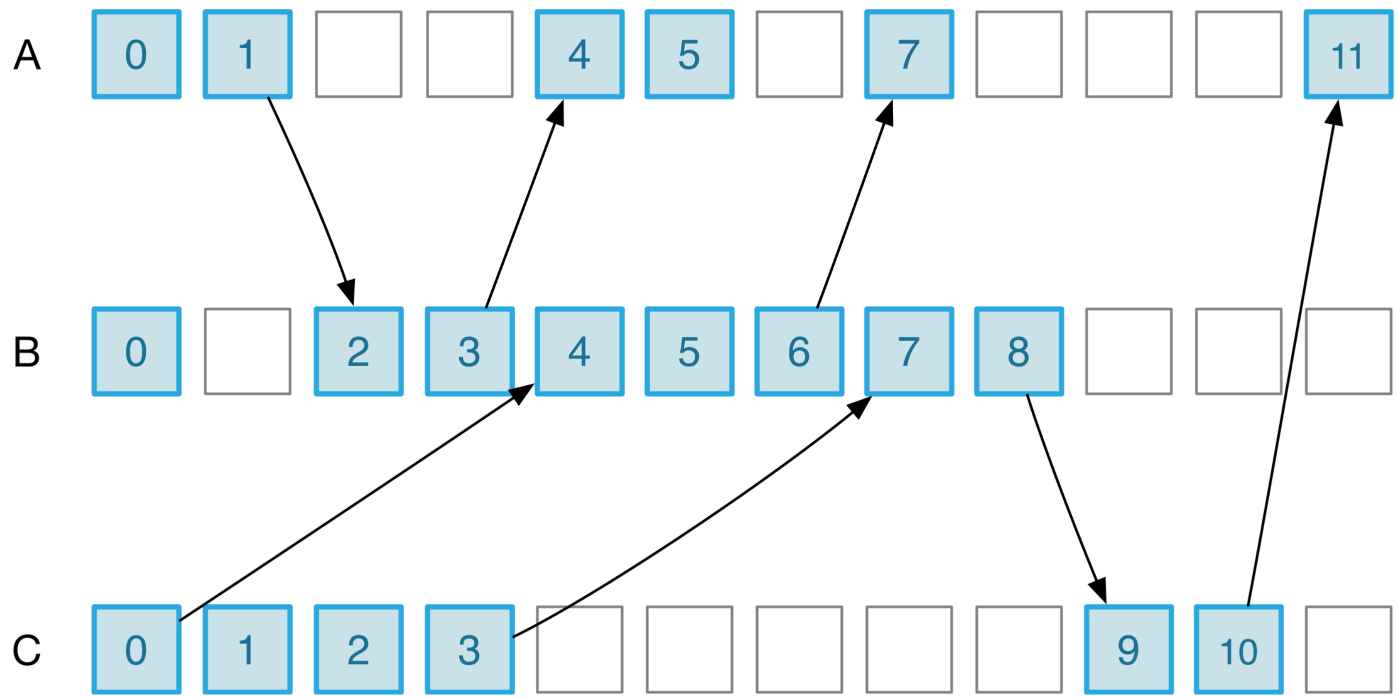
Lamport Clocks

- **Logical** clocks used to determine the order of events in a distributed system
- Establishes a *happens before* relationship between events:
  - **A** happened before **B**
  - Often this is just as useful as synchronizing clocks (common example: Makefiles)
- The transitive property applies:
  - **A** happened before **B**
  - **B** happened before **C**
  - Then **A** happened before **C**

Lamport Clock Implementation

- Algorithm based on a simple counter
- Each event increments the counter
  - Sending/receiving messages, storing a file, etc.
- When sending messages, a **timestamp** is attached with the current value of the counter
- When receiving messages, if the timestamp is greater than the local clock, it skips ahead

Lamport Clocks: 3 Processes

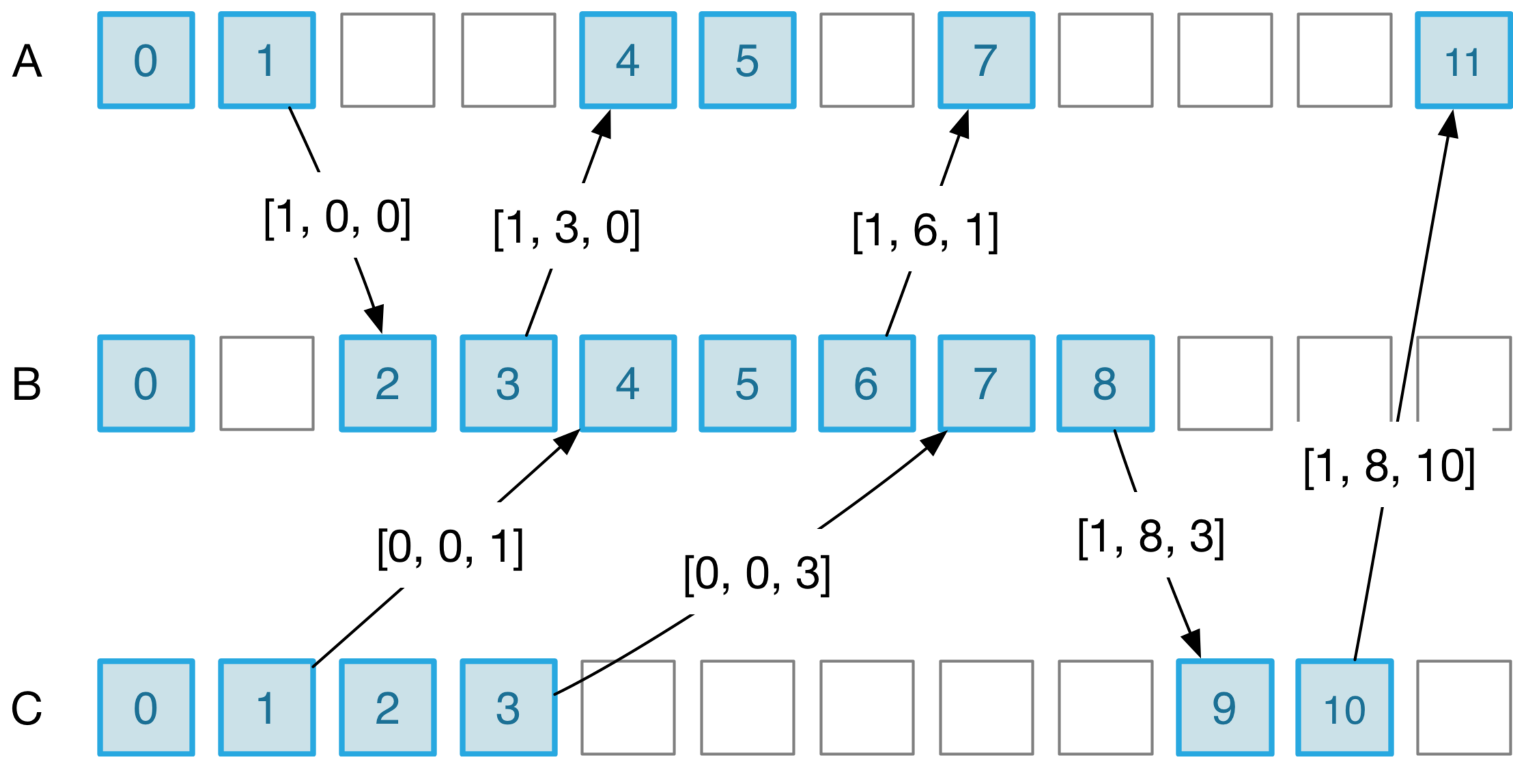


- Example concurrent events: C1 and B5
- We cannot conclude that C0 causally precedes A1

Vector Clocks

- Lamport clocks are simple, but we can only determine the **total ordering** of events
- With **vector clocks**, we assume we know about each participating process
- Instead of sending a single timestamp, send a **vector** of timestamps for each process
  - Update pairwise, same as Lamport clocks
- Enables causality to be captured

Vector Clocks: 3 Processes



Comparing Vectors

- Consider two vectors, **X** and **Y**:
- If each element of X is <= Y:

  X causally precedes Y
- If each timestamp in X is >= Y:

  Y causally precedes X
- Else: X and Y are concurrent

Today’s Agenda

- Replication and Failures
- Conflict Resolution
- **Consensus Algorithms**
- Transactions

Paxos

- Described in *The Part Time Parliament* by Leslie Lamport
- Describes a fictional parliamentary consensus protocol used by legislators in Paxos, Greece
  - Took around 10 years to get published… it was a bit unconventional
- Used frequently to achieve distributed consensus

Paxos Protocol

- Paxos is **quorum-based**
  - A majority of nodes must agree
  - Nodes play a variety of roles: leader, proposer, client, acceptor, learner
- Workflow:
  1. A leader is elected to coordinate the process
  2. A proposed value is sent to participating nodes
  3. Once a majority of nodes agrees on the value, consensus is reached

Fault Tolerance

- Everything moves along nicely when there are no network failures
  - When a failure occurs, multiple leaders can be elected
- As long as a leader receives a majority of votes (from its overall Paxos group), writes will succeed
- If a majority can't be obtained, writes will fail
  - Guarantees safety but **not** liveness
  - Often used by CP systems

Paxos Variants

- **Single decree Paxos:** reaching an agreement on a single object
  - Replica, file, log entry, etc.
- **Multi-Paxos:** re-uses leader nodes for multiple agreements

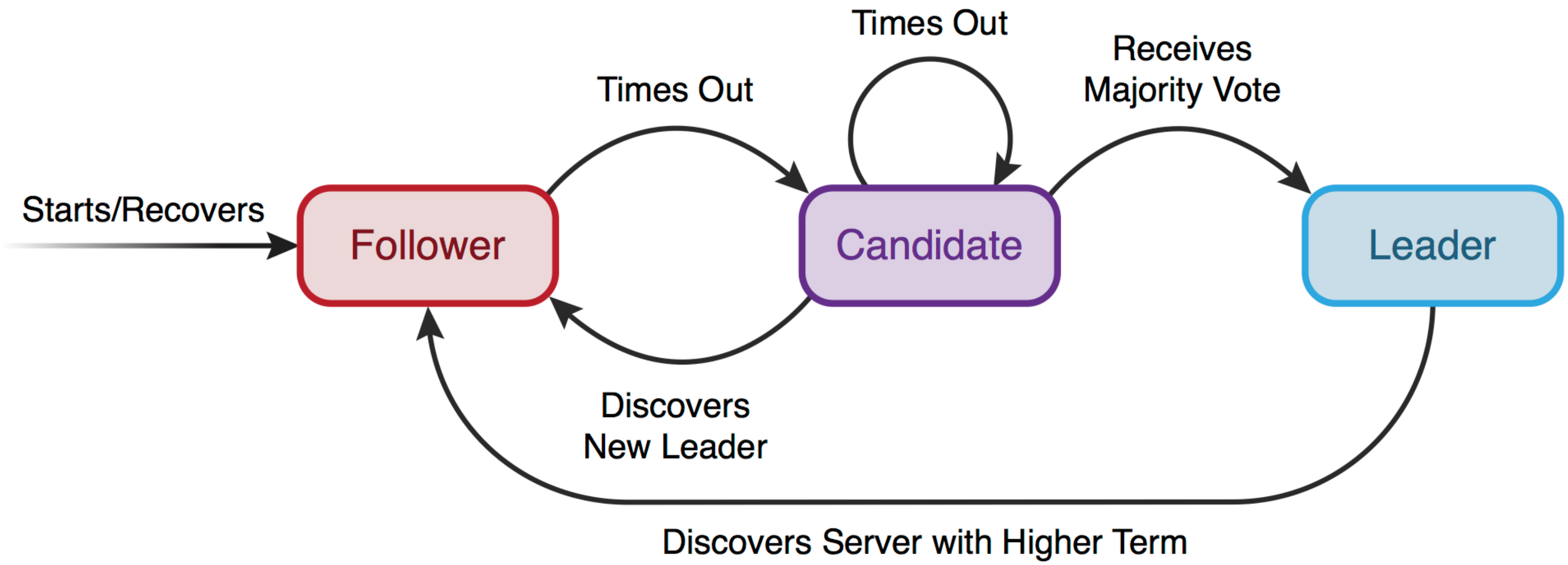
Implementation Difficulties

- Paxos is notoriously difficult to get right
- A simple protocol with lots of edge cases
- Google published a paper on Paxos-related engineering challenges:
  Paxos Made Live – An Engineering Perspective
  - Paxos is used by their **Chubby Lock Service**
- There's also *Paxos Made Simple* by Lamport
  - "Simple" is a bit generous

Raft

- **Raft** is an attempt to build a more understandable consensus algorithm
- Each component can be explained in isolation
  - Leader, candidate, follower
- Uses **strong leaders**
  - One leader per term
  - When a failed node comes back up, it assumes that it is a follower and waits for a timeout rather than trying to become a leader immediately
- Each leader election increments the term number

Raft: Components and Flow



Understanding Raft

- Raft is simpler, and tends to be better understood

- This has led to plenty of resources for learning Raft:
  - http://thesecretlivesofdata.com/raft/

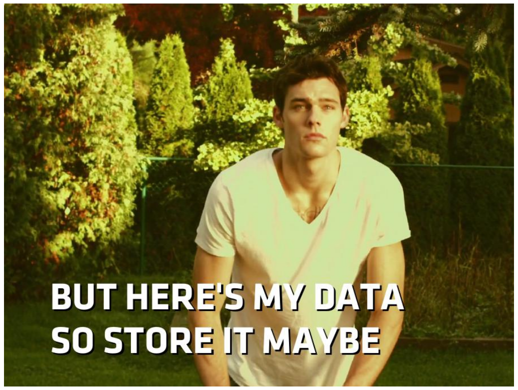
- There are also **lots** of library implementations available for nearly all programming languages

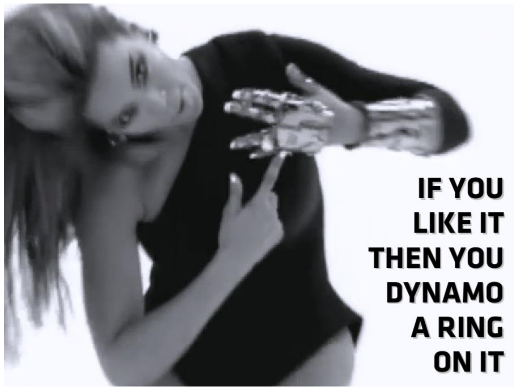
Zookeeper Atomic Broadcast

- Zookeeper is often used to coordinate between components and detect failures
- Supports **atomic broadcast**, where not only consensus must be reached but event ordering matters
  - ZAB
- Three phases: discovery, synchronization, broadcast

Call Me Maybe: Jepsen

- For some great reading material, check out the **Jepsen** articles by Kyle Kingsbury:
  - https://aphyr.com/tags/jepsen
- Breaks down systems' consistency claims
  - Even includes illustrations!





Today's Agenda

- Replication and Failures
- Conflict Resolution
- Consensus Algorithms
- **Transactions**

Distributed Transactions

- Thus far, we've discussed distributed **agreement**
  - Majority rules, and we can all agree on the outcome
- This isn't always good enough:
  1. Request 1: decrement account by $500
  2. Request 2: add 10% interest to account
- What we need is support for **transactions**:
  - Ensuring serializability
  - All nodes **commit** to a particular value/event

Two-Phase Commit

- Rather than a simple majority, two-phase commit (2PC) requires consensus from all nodes
- During a transaction, locks are acquired across all replicas
  - Increases latency
- Replicas attempt to apply the transaction to their log
  - Allows roll-back in the case of disagreement
- If all replicas agree, the transaction is **finalized**

Three-Phase Commit

- 2PC is a **blocking** operation
  - Guarantees safety
  - If a failure occurs, the system will hang
- In three-phase commit, a timeout is added
- If the transaction doesn't complete, it is aborted
- Weakness: only handles node failures, not network partitions
  - What happens when everyone agrees, but only some of the participants get the finalize message?

2PC on Paxos

- Google Spanner and F1 execute 2PC on top of Paxos groups
- Each group becomes one participant in 2PC
- Hierarchical consistency model: guarantees cross-group consistency
- Increases latency, but the Spanner/F1 designers saw an increase in developer productivity because they no longer had to deal with consistency issues