Distributed Coordination

Luís Osório

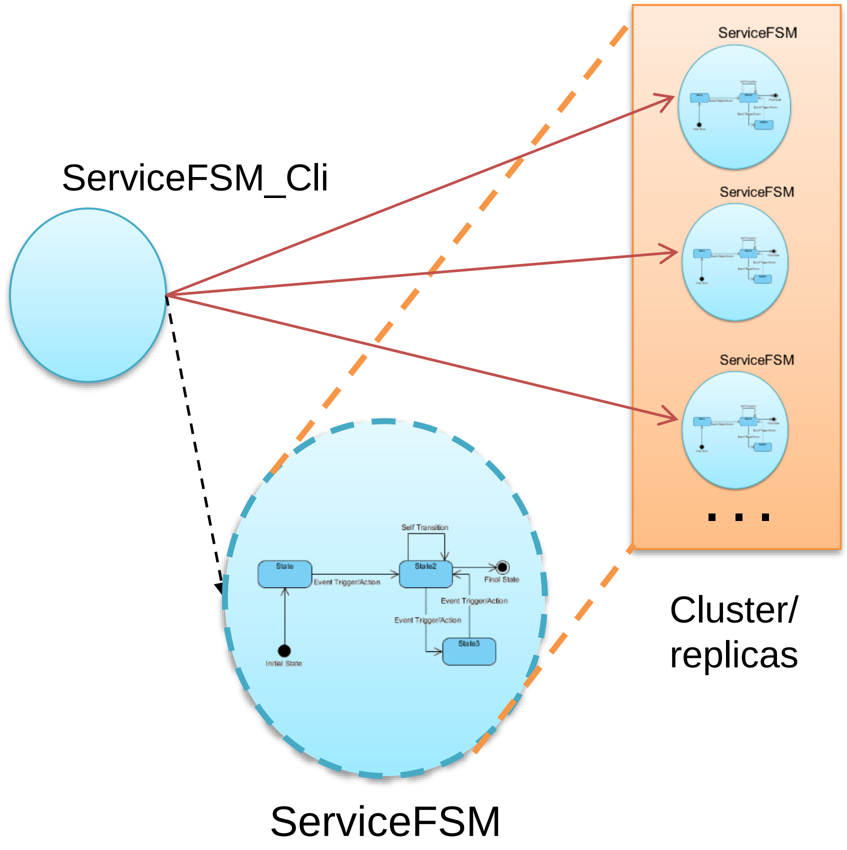
The Problem Motivating Distributed Coordination

The Failure

- The computer system failure (CPU, memory, power supply, etc
  - Commonly total-failure
- Software running error, e.g., memory violation, communication error, etc. leading to process/thread failure
  - Commonly partial-failure
- In (link, or the pdf) Schneider summarizes definitions, from both Lamport (link), the Bizantine failure, and, from himself (link) Fail-stop failure:
  - "***Byzantine Failures***. *The component can exhibit arbitrary and malicious behavior, perhaps involving collusion with other faulty components [Lamport et a182].*"
  - "***Fail-stop Failures****. In response to a failure, the component changes to a state that permits other components to detect that a failure has occurred and then stops [Schneider 84].*"
- Bizantine failures as discussed by Lamport (link)
  - "*A **reliable computer system** must be able to cope with the failure of one or more of its components. A failed component may exhibit a type of behavior that is often overlooked--namely, sending conflicting information to different parts of the system. The problem of coping with this type of failure is expressed abstractly as the **Byzantine Generals Problem**.*"

The Problem Motivating Distributed Coordination

- Consider a Service element responsible for executing a Finite State Machine
  - The next state depends on inputs and current state
  - State transition occurs when a special input, the event clock is triggered (synchronous)
- As discussed by Schneider in (link)
  - "… every **protocol** we know of that **employs replication** - be it for masking failures or simply to facilitate cooperation without centralized control - can be **derived** using the **state machine** approach."
  - The question is:
    - How to guarantee an FSM evolves consistently independent of a failure thar may occur?
  - Consider, a reliable ServiceFSM.



From: Zookeeper, Distributed Process Coordination;
ISBN: 9781449361303, O’Reilly, 2013

Fault Tolerance with the Zookeeper System

Distributed Coordination

Libraries and Documentation

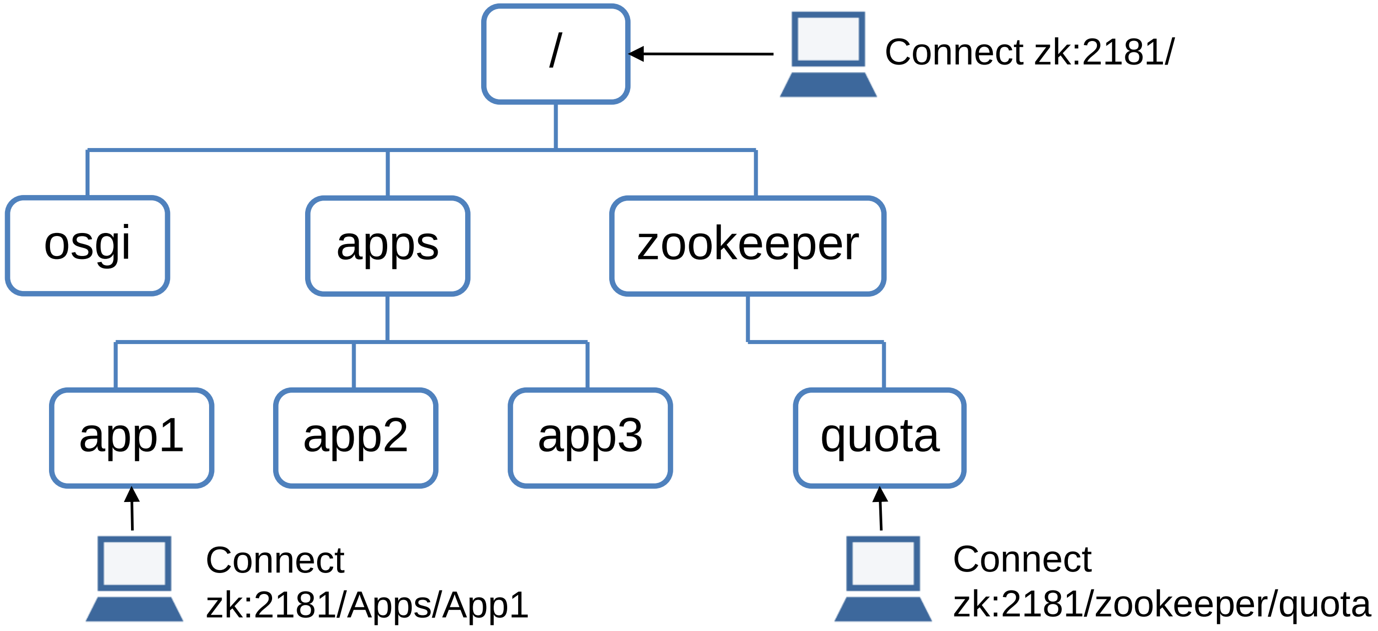
Apache Curator

- Enhanced Client library for Zookeeper
  - https://curator.apache.org/index.html
- References
  - https://www.baeldung.com/apache-curator
- Source and Examples
  - https://github.com/apache/curator

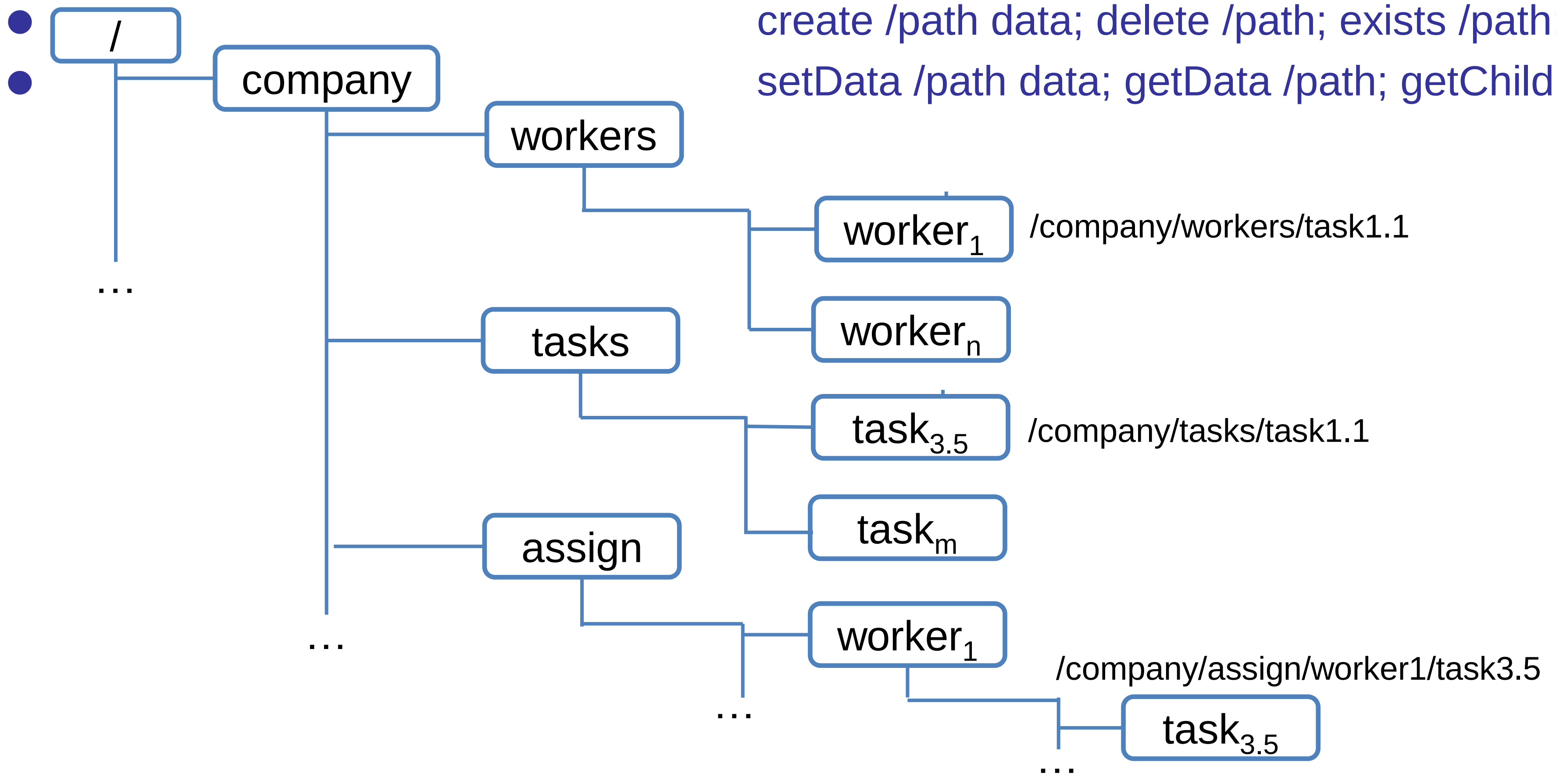
Spring Cloud Zookeeper

- https://spring.io/projects/spring-cloud-zookeeper
  - Service Discovery
  - Client side load-balancer (Ribbon, Spring Cloud Netflix)
    - Ribbon is a Inter Process Communication (remote procedure calls) library with built in software load balancers (https://github.com/Netflix/ribbon)
  - Supports Zuul, a gateway service that provides dynamic routing, monitoring, resiliency, security, … (https://github.com/Netflix/zuul)
  - Service Discovery
    - Zookeeper as a data store (e.g., OSGi Remote Services)

- The Zookeeper system is a reliable node (znode) persistence server.
  - A znode is part of a tree structure starting in a root '/', the root znode, and linked to zero or more children (it is a mix of file and a directory)
    - By initialization, Zoopkeeper starts with a root with one znode, the zookeeper znode that has a child quota



- It can be used to reliably store state through a simple API.



create /path data; delete /path; exists /path;

setData /path data; getData /path; getChildren /path

Important issues

- The zookeeper is an application domain independent system (infrastructure?) responsible for reliably make persistent znodes structured according the needs of the informatics system that uses it as a composite element;

- The clients have the above mentioned six simple operations
  - A znode can't be partially changed (only overridden / substituido)
    - App operations over znodes are idempotent (discussed later)

- A znode can be
  - Persistent (CreateMode.PERSISTENT)
    - Is maintained persistent even when the client that create it disconnects
  - Ephemeral (CreateMode.EPHEMERAL)
    - The znode is discarded when the client that create it disconnects
  - Sequential (Appends a sequence number to a node id)
    - Persistent (CreateMode.PERSISTENT_SEQUENTIAL)
    - Ephemeral CreateMode.EPHEMERAL_SEQUENTIAL)

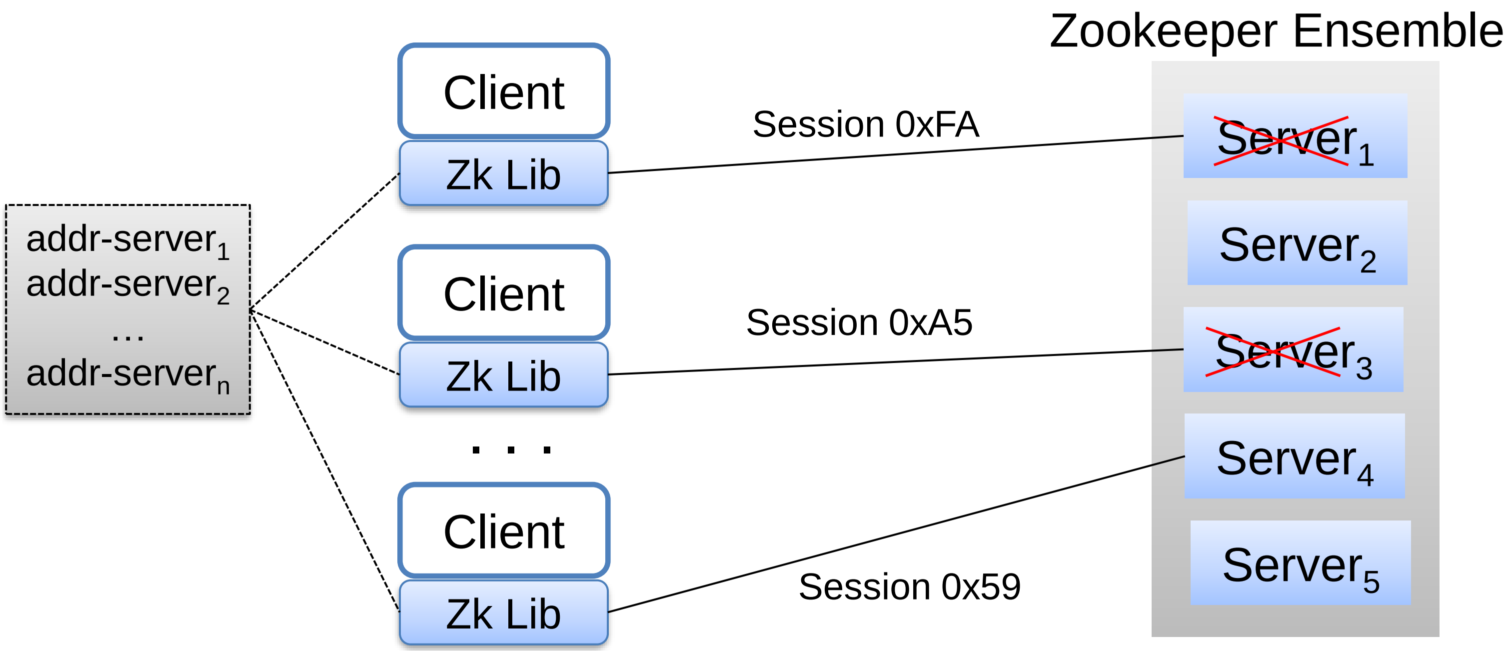
Important issues

- A client accesses zookeeper server based on a Session
  - Preserve FIFO order for any sequence of calls (through TCP client configured port)
    - For a same session;
      - if an initial session fails and another is established, the order is not any more guaranteed
  - A session is created through a ZooKeeper object
    - new ZooKeeper (ZOOKEEPER_SERVER, TimeOut [, Watcher]); where ZOOKEEPER_SERVER is the hostname or IpAddress of the host where Zookeeper server is running (or the Ensemble list of servers; a list host:clientPort)
  - A client session can also be created by starting the zkCli command
    - The prompt is [zk: siserver0.local(CONNECTED) 0] help
    - Through the help command is possible to access available commands.
  - ACL (Access Control List) Permissions
    - ZooKeeper supports the following permissions:
      - CREATE: you can create a child node
      - READ: you can get data from a node and list its children.
      - WRITE: you can set data for a node
      - DELETE: you can delete a child node
      - ADMIN: you can set permissions

Challenging Distributed Reliability

- According the Consistency, Availability, and Partition-tolerance (CAP) theorem
  - No one distributed system can handle the three dimensions simultaneously; by Eric Brewer, "*Towards robust distributed systems*", 2000 (link)
  - Later revisited "CAP twelve years later- Discusses further the theorem", 2012 (link)
- The Zookeeper system challenges such a possibility through a server, the zookeeper server, that can be configured considering the following modes:
  - Standalone (not reliable) (used in OSGi remote services)
  - Quorum, based on an **Ensemble** of servers
    - That, depending on the number of servers, failure of some is made transparent to client systems
    - The minimal number of ensemble servers is established by the minimum quorum that requires a majority of servers, like in real life general assemblies

- The reliability is related to its implementation strategy
  - Reliability, or at least an higher reliability level, requires the activation of the quorum mode
  - The quorum mode requires a group of servers to guarantee some level of replication of zookeep server state
    - Such a set of servers is named ***Ensemble***



The Quorum concept

- The quorum is in fact the minimal number of servers necessary for Zookeeper to work (to guarantee znode state replication)
  - Similar to the minimal number of presences in a general assembly to be able to take decisions, as mentioned already
- The question now is, what should be the minimal number of servers to make a quorum?
  - Suppose in an Ensemble of 5 we assume a quorum of 2, is it enough?
    - Suppose that servers 1 and 3 acknowledge the creation of the node /osgi, but before the change Is propagated to the other servers (2, 4, and 5) the servers 1 and 3 split the partition (known as ***Split-brain*** situation), by some failure.
    - Based on the established minimal quorum of 2, client continues convinced that /osgi was persisted. However as the partition happened before the replication process was complete, in fact, servers 2, 4, and 5 do not have the created znode
- Important to consider
  - A majority of nodes is necessary to guarantee consistency of the replicated state (FSM, our vector or other critical data
  - How can we scale a stateful service element?
  - How to deal with scale?

The Quorum concept

- A quorum requires a majority of servers of an ensemble
  - This means that for an ensemble of 5 the quorum is at least 3
  - In this case the tolerance to failure is of two servers
- Formally if E represents an ensemble with |E| servers and N_F the number of faulty servers,
  - From the assumption of Lamport, $N > 3N_F$ consensus can be obtained, and
  - Therefore the minimal quorum for one faulty server of an Ensemble, is
    - **$N = 3*N_F + 1$**; for the minimal quorum of 3, |E| = 3*1 + 1 = 4, the required number of nodes (independent servers)

Noteworthy

- The Service elements instances shall be running on failure-independent execution environments
  - E.g., each node running on different racks with independent power supplying
- If disaster fault tolerance is to consider
  - Nodes need to be in different geographical locations, and communication infrastructure needs to show redundancy

Coordination Strategy

- Based on the Zookeeper Atomic Broadcast (Zab) protocol
  - It was proposed by Flavio P. Junqueira, Benjamin C. Reed, and Marco Serafini, from Yahoo! Research, in the paper "Zab: High-performance broadcast for primary-backup systems", 2011 (link)
  - It is an evolution/simplification from the Paxos consensus algorithm initially proposed by Leslie Lamport in the paper "The Part-Time Parliament", 2000 (link)
    - "*Recent archaeological discoveries on the island of **Paxos** reveal that the **parliament functioned** despite the peripatetic propensity of its **part-time legislators**. The legislators maintained **consistent copies of the parliamentary record**, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the **state-machine approach to the design of distributed systems**.*"
- It is based also on the Byzantine Generals problem
  - How to guarantee that only **loyal generals** will **reach agreement** on a **military strategy**?

Zookeeper

- Considers three classes of servers:
  - Leader, Followers, and Observers

The main role of each class of servers

- Leader
  - The one that **coordinates commits** (of zxid transactions) of the server's state
  - The **state change** is processes by the **leader**
- Followers
  - **Forward state changes** to the Leader
  - Process **reads**
  - Initiate **leader election** if they can't contact the leader
- Observers
  - Process reads (scalability reasons)
    - As they don't count for quorum, their number can vary according the need to solve bottlenecks for reads making more Observers to help and free Followers for updates
  - Do not participate in the election process

Operations over znodes

- Read
  - exists /path; getData /path; getChildren /path
- Update
  - create /path data; delete /path; setData /path data

Transactions

- The transactions are **idempotent** operations, i.e., operations can be replayed without changing the expected result
- A zookeeper transaction, named zxid is a 64 bit long with two parts
  - Epoch, 32 bits
  - Counter, 32 bits

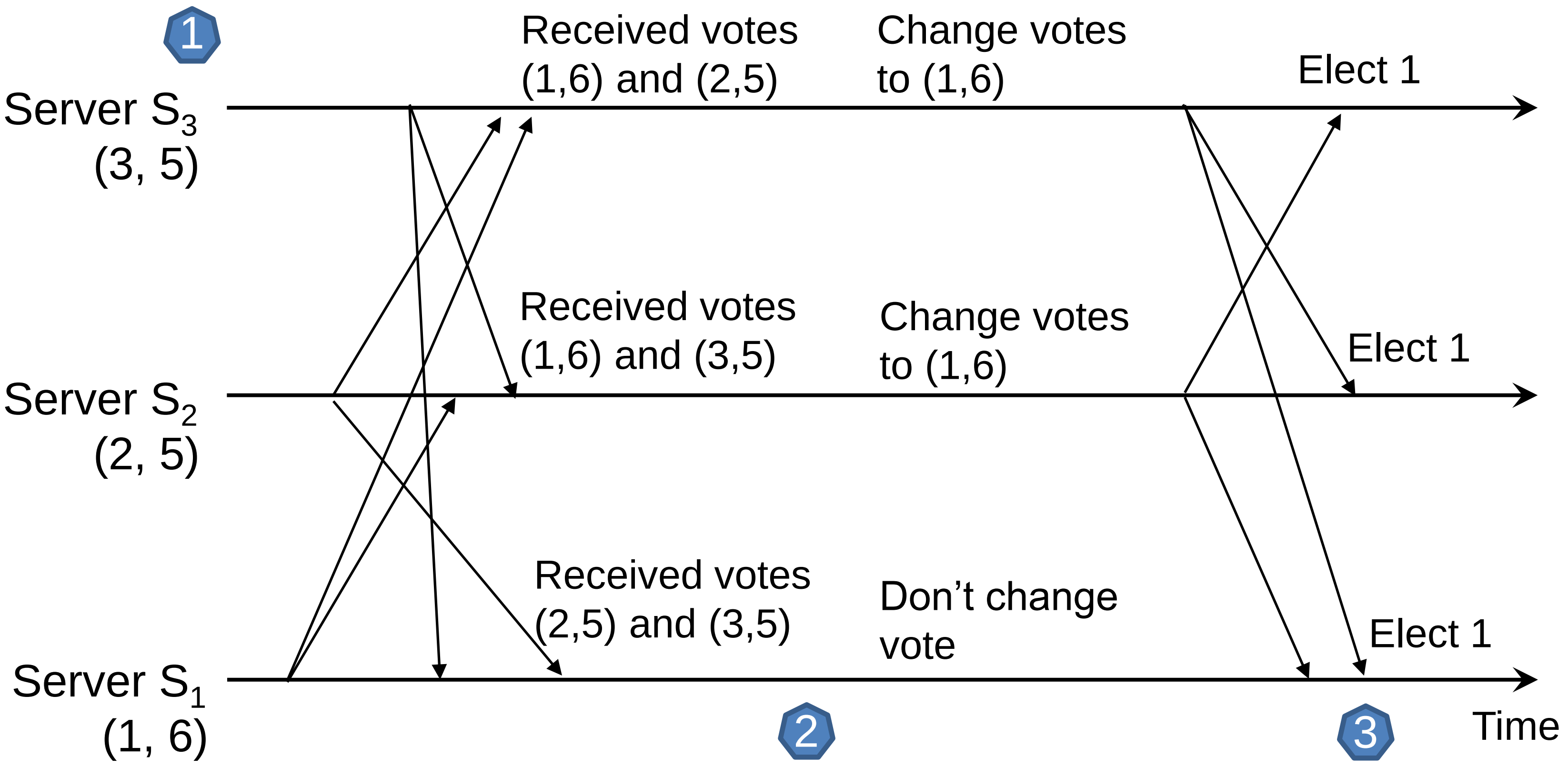
Leader Election

- Is the one chosen among ensemble members to take the Leader role
- A main purpose is to order client requests to change the zookeeper state (create /path data; delete /path; setData /path data)
- Each server starts in the LOOKING state
  - Elect a new server, or
  - Find the existing one
- If election is needed, depending on the result
  - One server enters the LEADING state, and
  - The others enter the FOLLOWING state

Leader Election Algorithm

- Leader election messages
  - ***leader election notifications***, or ***notifications***
- When a server enters the LOOKING state it sends a batch of notification messages
  - The message contains its current vote, the server id (sid) and the most recent executed transaction (zxid)
- Upon receiving a vote, a server changes its vote according the following rules:
  - Let vote(Id, Zxid) be the identifier and the zixd in the current received vote, whereas myZxid and mySid are the values of the receiver itself
  - If (voteZxid < myZxid) or (voteZxid = myZxid and voteId > mySid), keep the current vote
  - Otherwise, change my vote by assigning myZxid to vote voteZxid and mySid to vote Zxid
- The server that is most up to date is the winer
  - I.e., has an higher value for the zxId
  - For a same zxId, prevails the higher Sid server

Leader Election



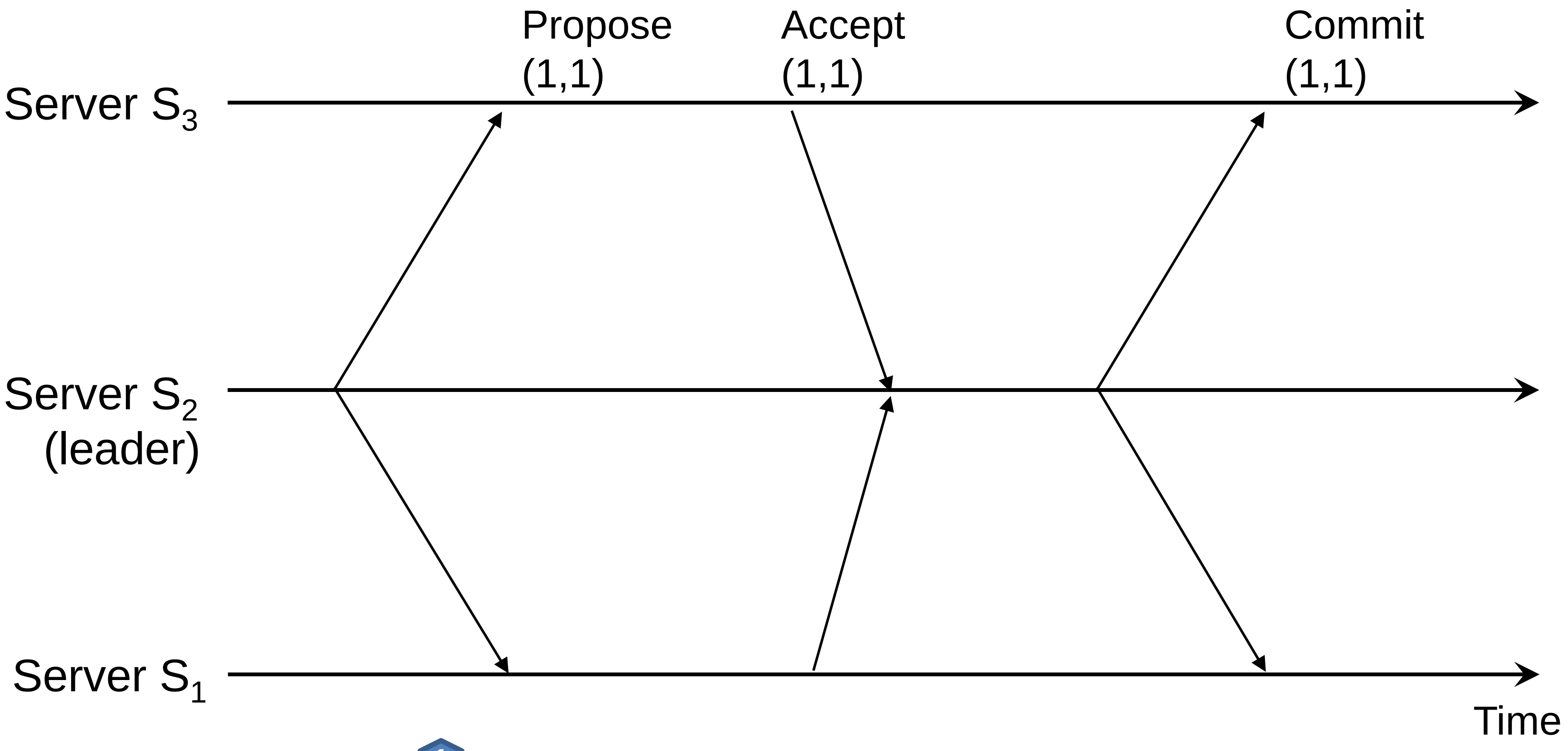
1. Server S_1 starts with vote (1, 6), server S_2 with (2, 5) and S_3 with (3, 5)
2. Server S_1 and S_3 change their vote to (1, 6) and send a new batch of notifications
3. All the three servers receive the same vote from a quorum and elect server S_1

From: Zookeeper, Distributed Process Coordination;
ISBN: 9781449361303, O’Reilly, 2013

The Zab (Zookeeper Atomic Broadcast) protocol

- The question now is how to determine the commit of a transaction
- It refers any change to the Zookeeper DataTree, the tree of znodes
  - Any change, corresponds to a DataTree state change that needs to be maintained consistent in the quorum of servers
- The protocol to commit a transaction resembles the two-phase commit algorithms as established by the X/Open standard
  - 1. The Leader sends a PROPOSAL message p to all the followers
  - 2. Upon receiving p, followers respond to the leader with an ACK, informing the leader that it has accepted the proposal
  - 3. Upon receiving acknowledgments from a quorum (including the leader), the leader sends a message informing the followers to COMMIT the transaction
- Zab guarantees that:
  - If the leader broadcast T and T’ in that order, each server must commit T before T’
  - If any server commits transactions T and T’ in that order, all servers must commit T before T’

Commit Transactions



1. Leader sends propose messages
2. Followers ack the proposal
3. Leader commit the proposal

From: Zookeeper, Distributed Process Coordination;
ISBN: 9781449361303, O'Reilly, 2013

Initialize zookeeper server in standalone mode

- Install from Apache zookeeper project (stable version)
  - Installation: C:\Java\apache-zookeeper-3.9.2-bin\
    - On Linux (Ubuntu) /opt/zookeeper
  - C:\Java\apache-zookeeper-3.9.2-bin\conf\zoo.cfg
    - tickTime=2000
    - dataDir=\tmp\zookeeper
      - On Linux: /tmp/zookeeper
    - clientPort=2181
      - ***tickTime*** - time unit in milliseconds (heartbeats, minimum session timeout 2 * tickTime)
      - ***dataDir*** - in-memory database and transaction log
      - ***clientPort*** - the port to listen for client connections
  - Start the server
    - C:\...>Java\apache-zookeeper-3.9.2-bin\bin\zkServer.cmd start
  - Start a client
    - C:\...>\ Java\apache-zookeeper-3.9.2-bin \bin\zkCli.cmd –server siserver0.local:2181

Initialize zookeeper quorum mode

- Install Zookeeper in all the participating servers
  - C:\Java\apache-zookeeper-3.9.2-bin\conf\zoo.cfg

    tickTime=2000

    dataDir=/tmp/zookeeper

    clientPort=2181

    server.1=siserver0.local:2888:3888

    server.2=siserver1.local:2888:3888

    server.3=siserver2.local:2888:3888
  - *Replicate the configuration file to all the servers*
  - Start the servers

    si@siserver0:~$ sudo /usr/local/zookeeper/bin/zkServer.sh start

    si@siserver1:~$ sudo /usr/local/zookeeper/bin/zkServer.sh start

    si@siserver2:~$ sudo /usr/local/zookeeper/bin/zkServer.sh start
  - Start a client
    - C:\...>\Java\apache-zookeeper-3.9.2-bin\bin\zkCli.cmd –server siserver0.local:2181
    - C:\...>\ Java\apache-zookeeper-3.9.2-bin\bin\zkCli.cmd –server "siserver0.local:2181,siserver1.local:2181,siserver2.local:2181"

The technology Landscape of an Organization (IT)

- Is composed of (ISoS objects)
  - One $ISystem_0$,
  - Zero or more ISystem
- An ISystem (Informatics System), is composed of
  - One or more Cooperation Enable Service (CES)
- A CES is composed of
  - One or more Service, as independent (autonomous) computational entity
  - The Service entity is in fact the artifact that embeds computational logic and resources necessary to operationalize a computational responsibility.

The $ISystem_0$ serves as ISystem/CES/Service Registry

- A REST interface (I_0) accessible at
  - isos.<organization_domain>:2058
- Has an Administration user interface (Admin)
  - Navigation through the ISoS tree (ISystem/CES/Service)

Reliable ISoS Reference Implementation