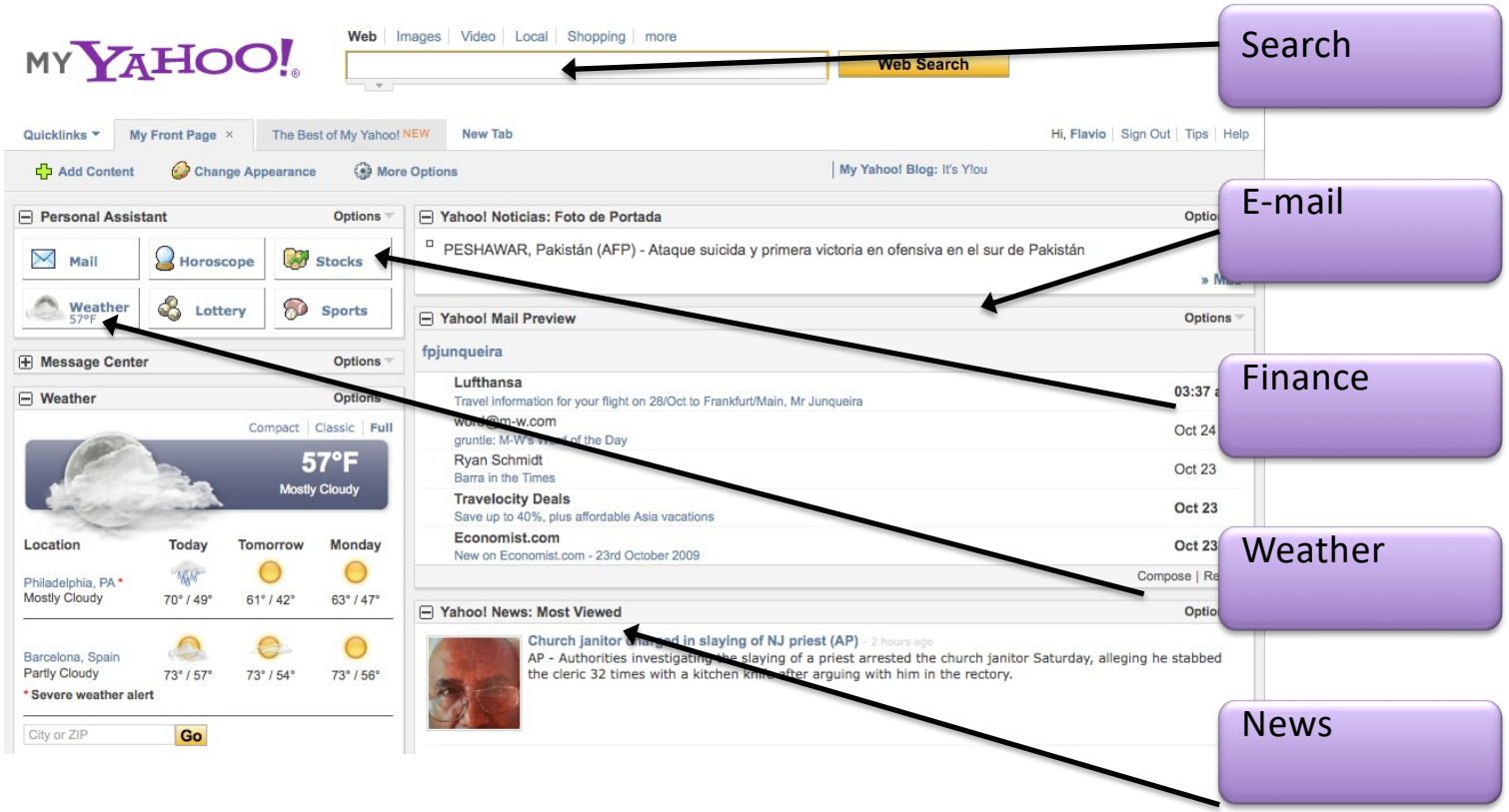
CS 5450

Zookeeper

"Because coordinating distributed systems is a Zoo"

Slides from Ken Birman and EuroSys 2011 Zookeeper tutorial

Yahoo! Portal (2011)



Yahoo! Workload (2011)

- Home page
  - 38 million users a day (USA)
  - 2.5 billion users a month (USA)
- Web search
  - 3 billion queries a month
- E-mail
  - 90 million actual users
  - 10 min/visit

Yahoo! Infrastructure

- Lots of servers
- Lots of processes
- High volumes of data
- Highly complex software systems
- … and developers are mere mortals

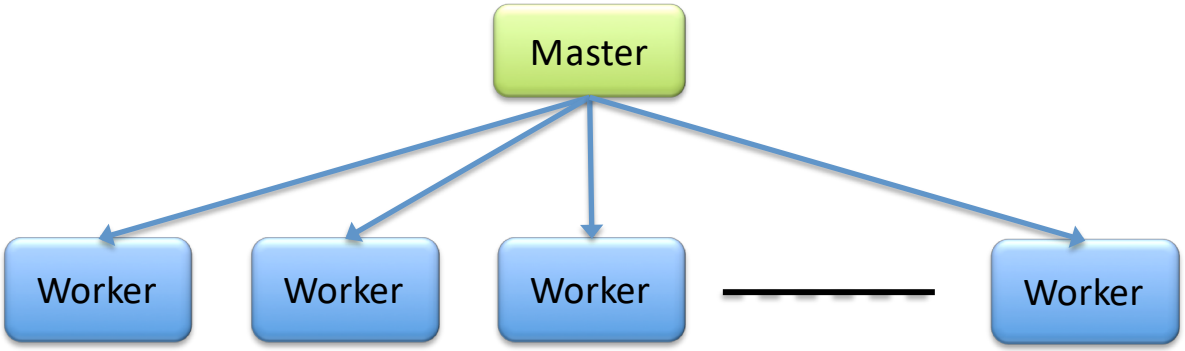
Coordination is Important

Coordination Primitives

- Semaphores
- Queues
- Leader election
- Group membership
- Barriers
- Configuration

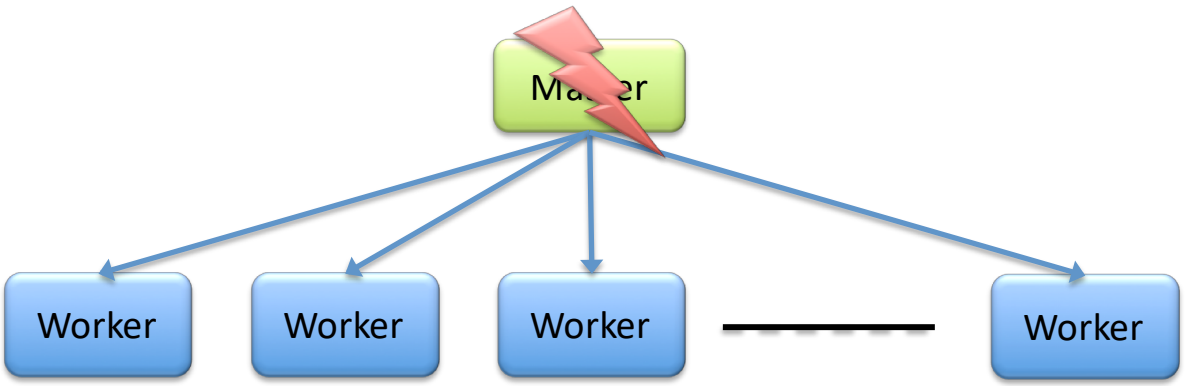
A Simple Master-Worker Model

- Master assigns work
- Workers execute tasks assigned by master



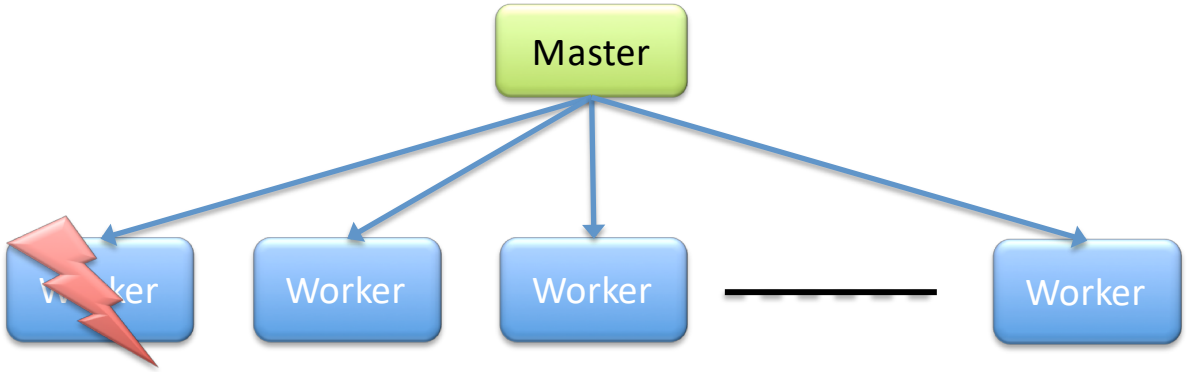
Master Crashes

- Single point of failure
- No work is assigned
- Need to select a new master



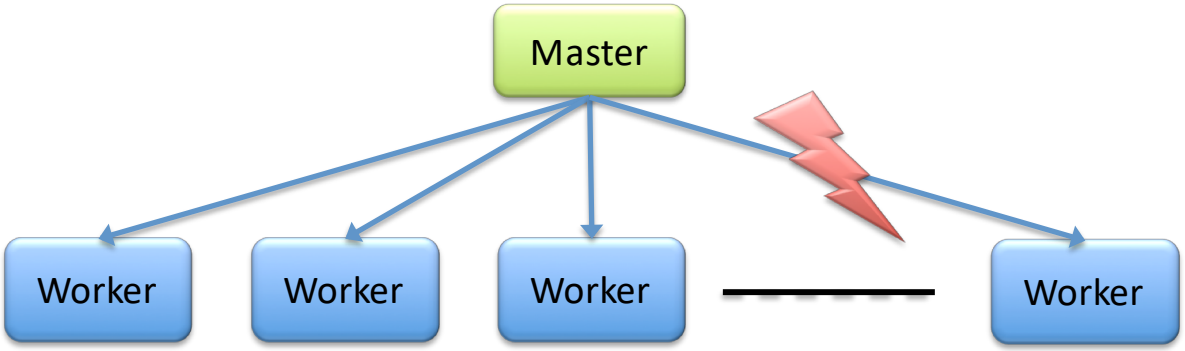
Worker Crashes

- Not as bad… Overall system still works
  - Does not work if there are dependencies
- Some tasks will never be executed
- Need to detect crashed workers

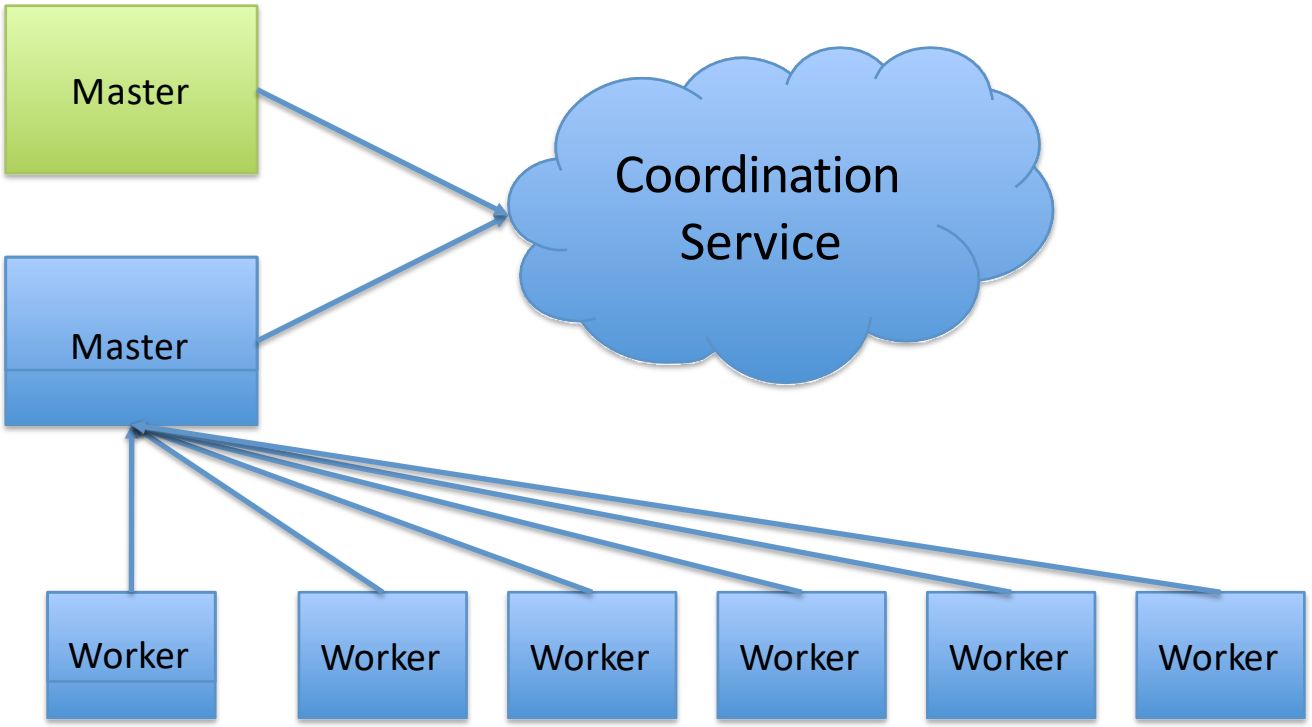


Worker Doesn't Receive Assignment

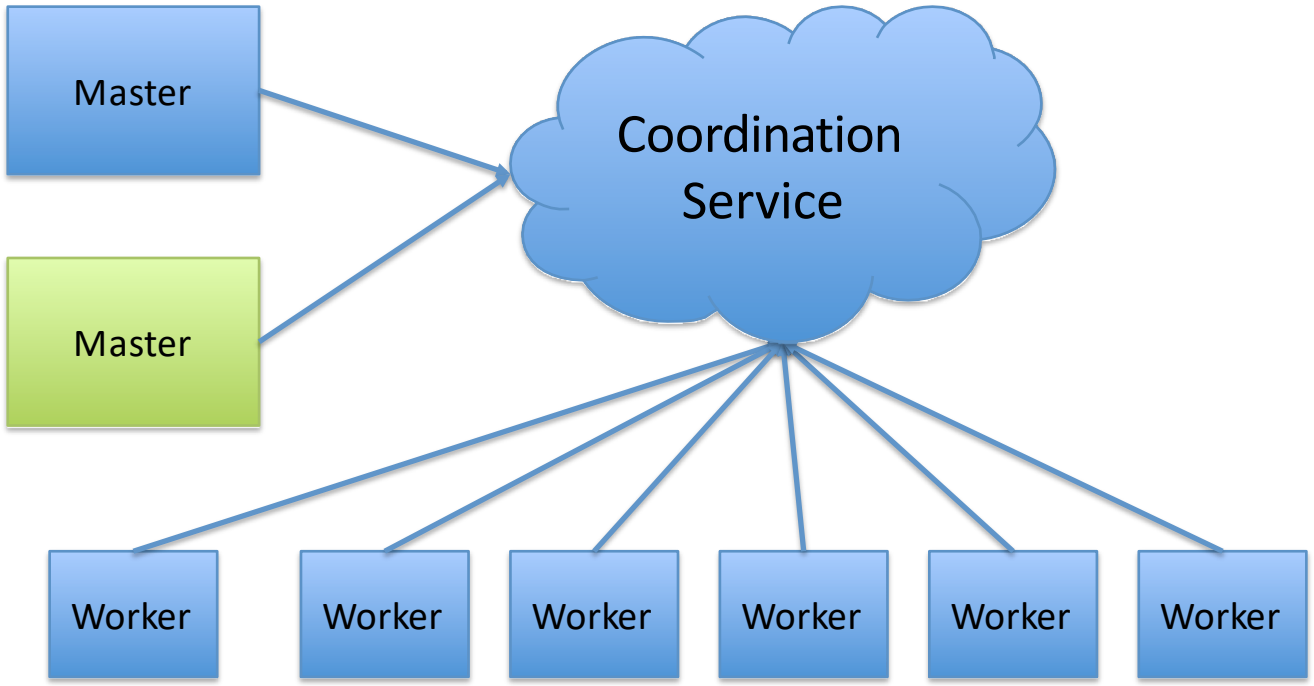
- Some tasks may not be executed
- Need to guarantee that worker receives assignment



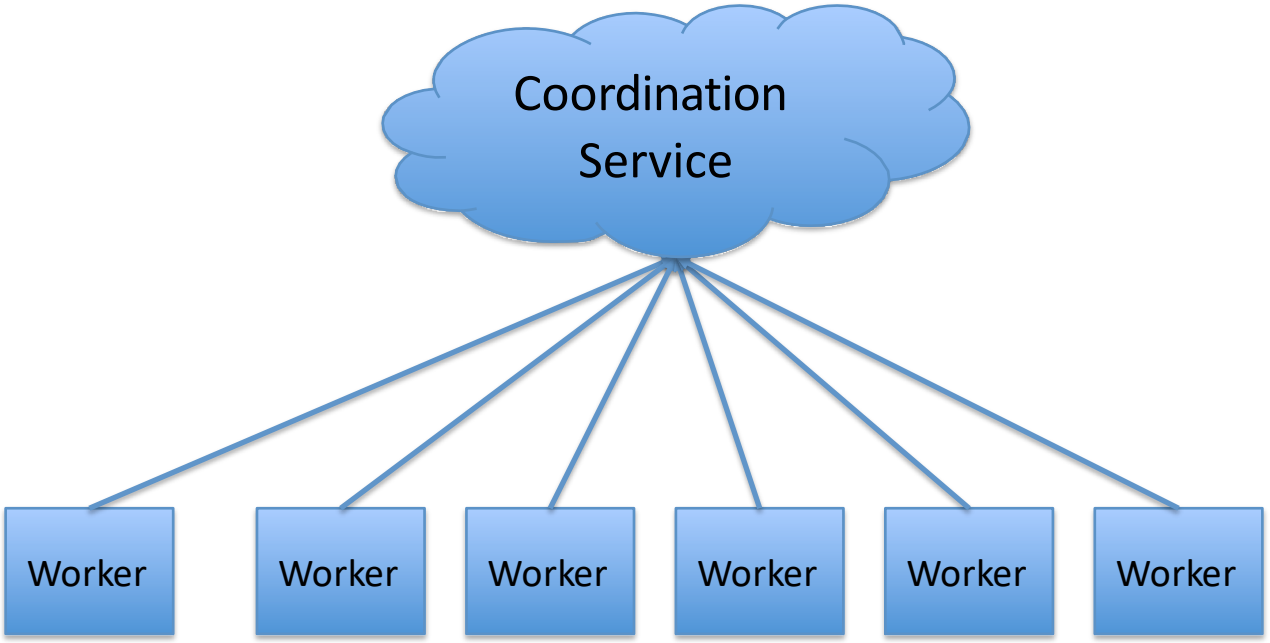
Fault-Tolerant Distributed System



Fault-Tolerant Distributed System



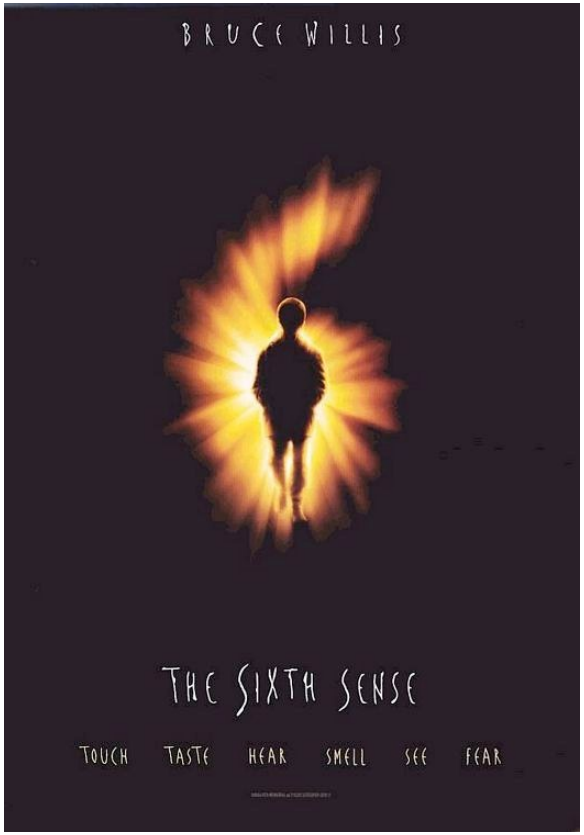
Fully Distributed System



Fallacies of Distributed Computing

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.
7. Transport cost is zero.
8. The network is homogeneous.

One More Fallacy



You know who is alive

Why Is This Difficult?

- FLP impossibility result
  - Consensus in asynchronous systems is impossible if a single process can crash
- CAP principle
  - Can’t obtain availability, consistency, and partition tolerance simultaneously

Why Need Coordination Systems?

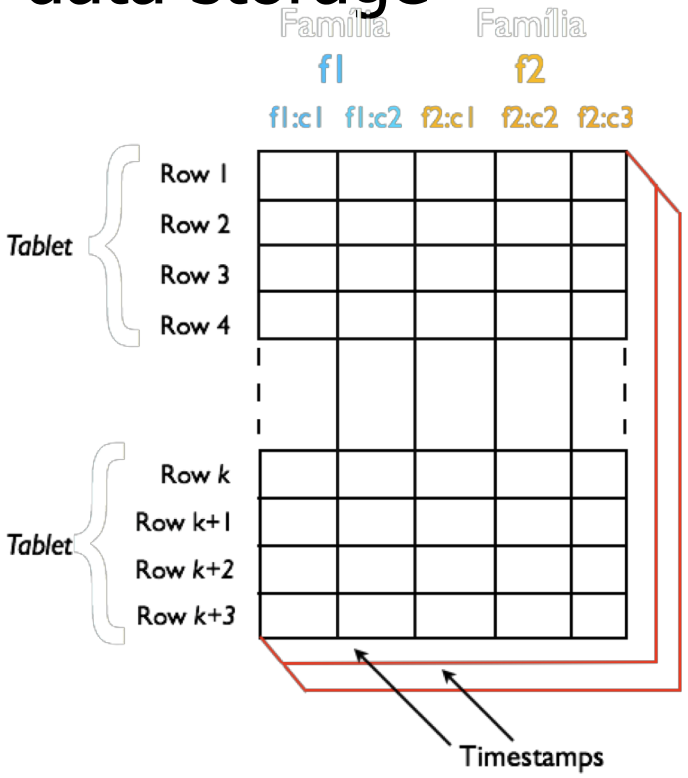
- Many impossibility results
- Many fallacies to stumble upon
- Several common requirements across applications
  - Duplicating is bad
  - Duplicating poorly is even worse
- Coordination service
  - Implement it once and well
  - Share by a number of applications

Existing Systems

- Chubby (Google)
  - Lock service
- Centrifuge (Microsoft)
  - Lease service
- Zookeeper (Yahoo!, Apache since 2008)
  - Coordination kernel

Example: Bigtable, HBase

- Sparse column-oriented data storage
  - Tablet: range of rows
  - Unit of distribution
- Architecture
  - Master
  - Tablet servers

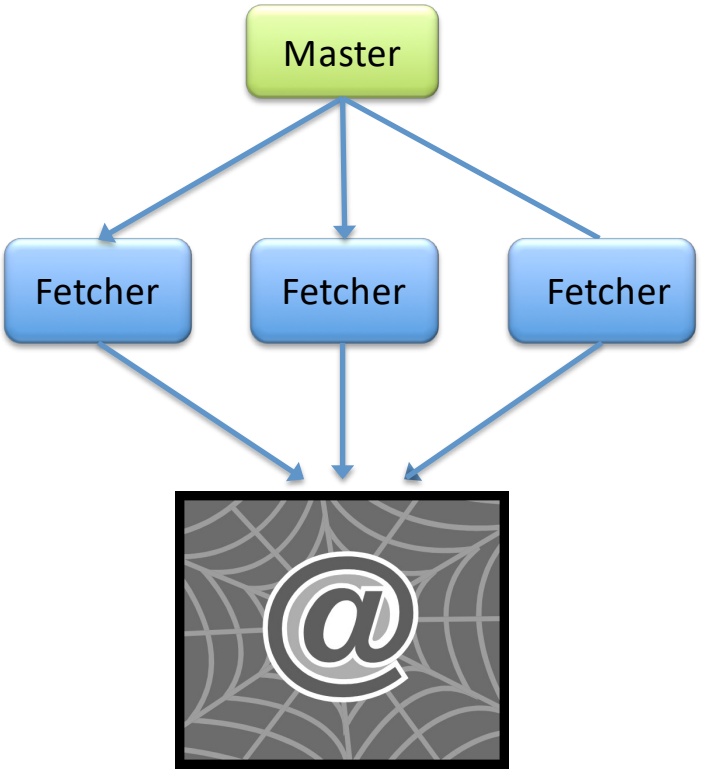


Bigtable, HBase Requirements

- Master election
  - Tolerate master crashes
- Metadata management
  - ACLs, Tablet metadata
- Rendezvous
  - Find tablet server
- Crash detection
  - Live tablet servers

Example: Web Crawling

- Master election
  - Assign work
- Metadata management
  - Politeness constraints
  - Shards
- Crash detection
  - Live workers



More Examples

- GFS – Google File System
  - Master election
  - File system metadata
- KaCa – document indexing system
  - Shard information
  - Index version coordination
- Hedwig – Pub-Sub system
  - Topic metadata
  - Topic assignment

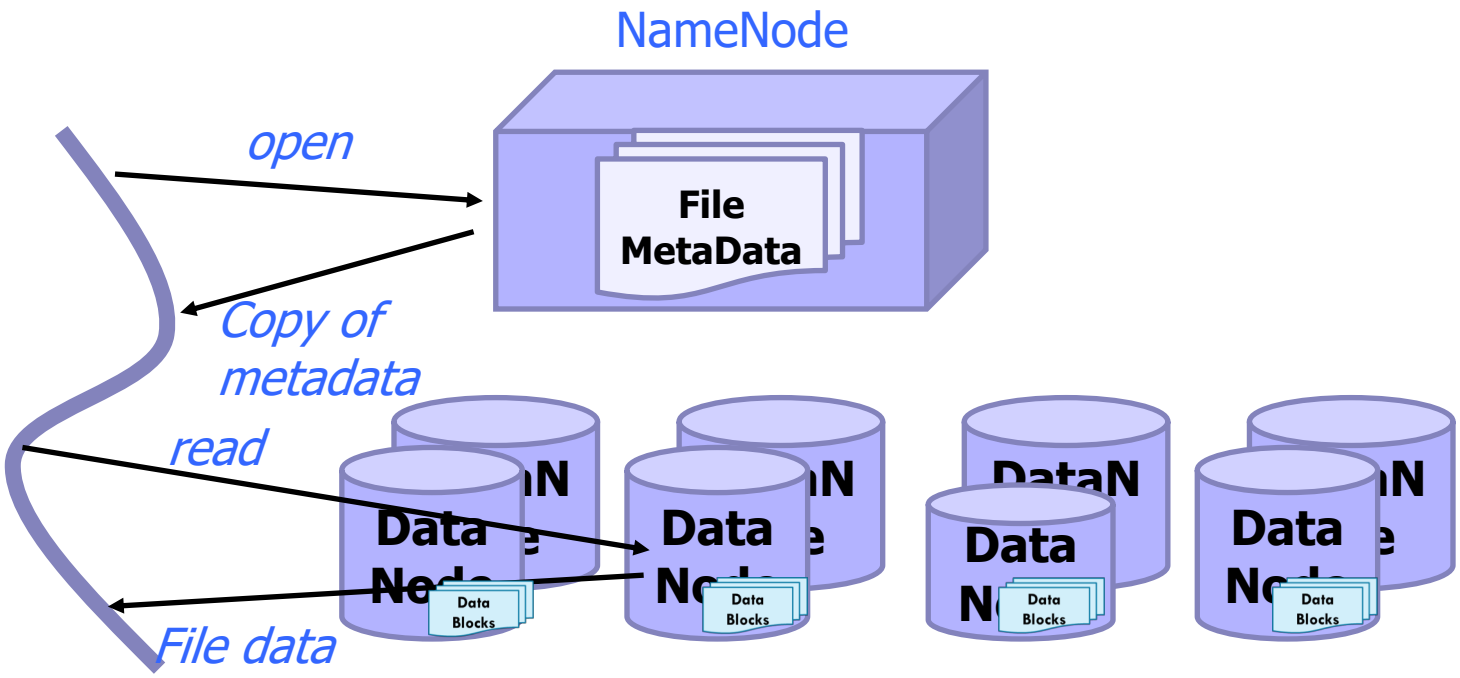
File Systems for the Cloud

- "Global" file systems for bulk storage
  - Google's GFS
  - Amazon S3
  - Azure storage fabric
- Typically offer built-in block replication using a Linux feature but guarantees somewhat weak

How They Work

- A Name Node fault-tolerant service tracks file metadata (like a Linux inode)
  - Name, create/update time, size, seek pointer, etc.
- Name Node tells which data nodes hold each file
- Very common to use a simple DHT scheme to fragment the Name Node into subsets to spread the load, data nodes are hashed at block level
- Some form of primary/backup scheme for fault-tolerance
  - Writes are automatically forwarded from the primary to the backup

How They Work

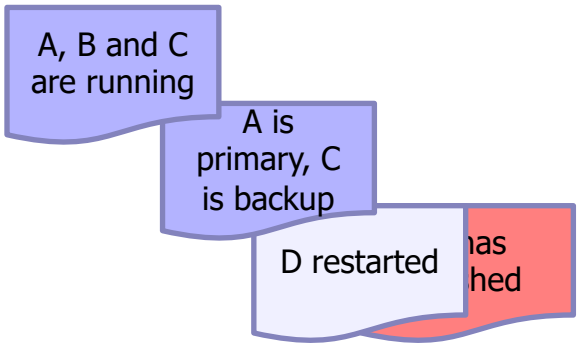


Global File Systems: Pros and Cons

Pros	Cons
1. Scales well even for massive objects 2. Works well for large sequential reads/writes, 3. Provides high performance (massive throughput) 4. Simple but robust reliability model	1. NameNode (Master) can become overloaded, especially if individual files become extremely popular 2. … a single point of failure 3. … if slow, can impact the whole data center 4. Concurrent writes to the same file can interfere with one another

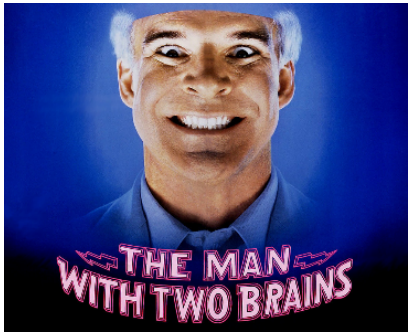
Limitations of Logging

- Maintain a log of computer status events: “crashed”, “recovered”…
- The log is append-only: when you sense something, write to the end of the log
- Issue: If two events occur more or less at the same time, one can overwrite the other, hence one might be lost



Overwrites Cause Inconsistency

- If we are logging status of machines, some machines may have seen the overwritten update and think that C crashed, but others never saw this message
  - Worst case: maybe C really is up, and the original log report was due to a transient timeout… but now half the system thinks C is up, and half thinks C is down
- The system has a "split brain"



Root Issue (1)

- The quality of failure sensing is limited
- If we use timeouts to sense faults, we might make mistakes. Then if we reassign the role but the "faulty" machine is really still running and was just transiently inaccessible, we have two controllers!
- This problem is unavoidable in a distributed system, so we have to use agreement on membership, not "apparent timeout". The log plays this role.

Root Issue (2)

- In many systems two or more programs can try to write to the same file at the same time or to create the same file
- The normal Linux file system will work correctly if the programs and the files are all on one machine. Writes to the local file system won't interfere.
- Distributed, global file systems lack this property!

Consistent Logging

- If you trust the log, just read log records from top to bottom and get an unambiguous way to track membership
  - Trust in the log depends on the file system
- Even if a logged record is "wrong", e.g. "node 6 has crashed" but it hasn't, we are forced to agree to use that record

With S3 and GFS, Can't Trust Log

- These file systems don't guarantee consistency, they are unsafe with concurrent writes
- Concurrent log appends can…
  - Overwrite each other, so one is lost
  - Be briefly perceived out of order, or some machine might glimpse a written record that will be overwritten a moment later
  - Sometimes two conflicting versions can even linger for extended periods

How Things Goes Wrong

"Append-only log" behavior

1. Machines A, B and C are running

2-a. Machine D is launched

2-b. Concurrently, B thinks A crashed

3. 2-b is overwritten by 2-a

4. A turns out to be fine, after all

In an application using it

- A is selected to be the primary controller for some functionality, C is assigned as backup.
- C notices 2-b, and takes over. But A wasn't really crashed – B was wrong!
- **Log entry 2-b is gone.**

  A keeps running.
- Now we have A and C both in the controller role – a split brain!

Fine for Non-Concurrent Writes

- S3, GFS, and similar systems are perfectly fine for object storage by a single, non-concurrent writer
- The reason that they don't handle concurrent writes well is that the required protocol is slower than the current weak consistency model

Zookeeper

- A system for solving distributed coordination problems
- Many systems need a place to store configuration, parameters, lists of which machines are running, which nodes are “primary” or “backup”, etc.
- File-system interface with strong, fault-tolerant semantics
- Stronger guarantees than GFS, but…
  - Data lives in small files
  - Slow and not very scalable

When To Use Zookeeper

- For small objects used to do distributed coordination, synchronization or configuration
- Not for persistent data
  - If shut down, loses state
  - Checkpointing every 5s by default, but recent updates will be lost
  - Most applications simply leave Zookeeper running and if it shuts down, the whole application shuts down and restarts

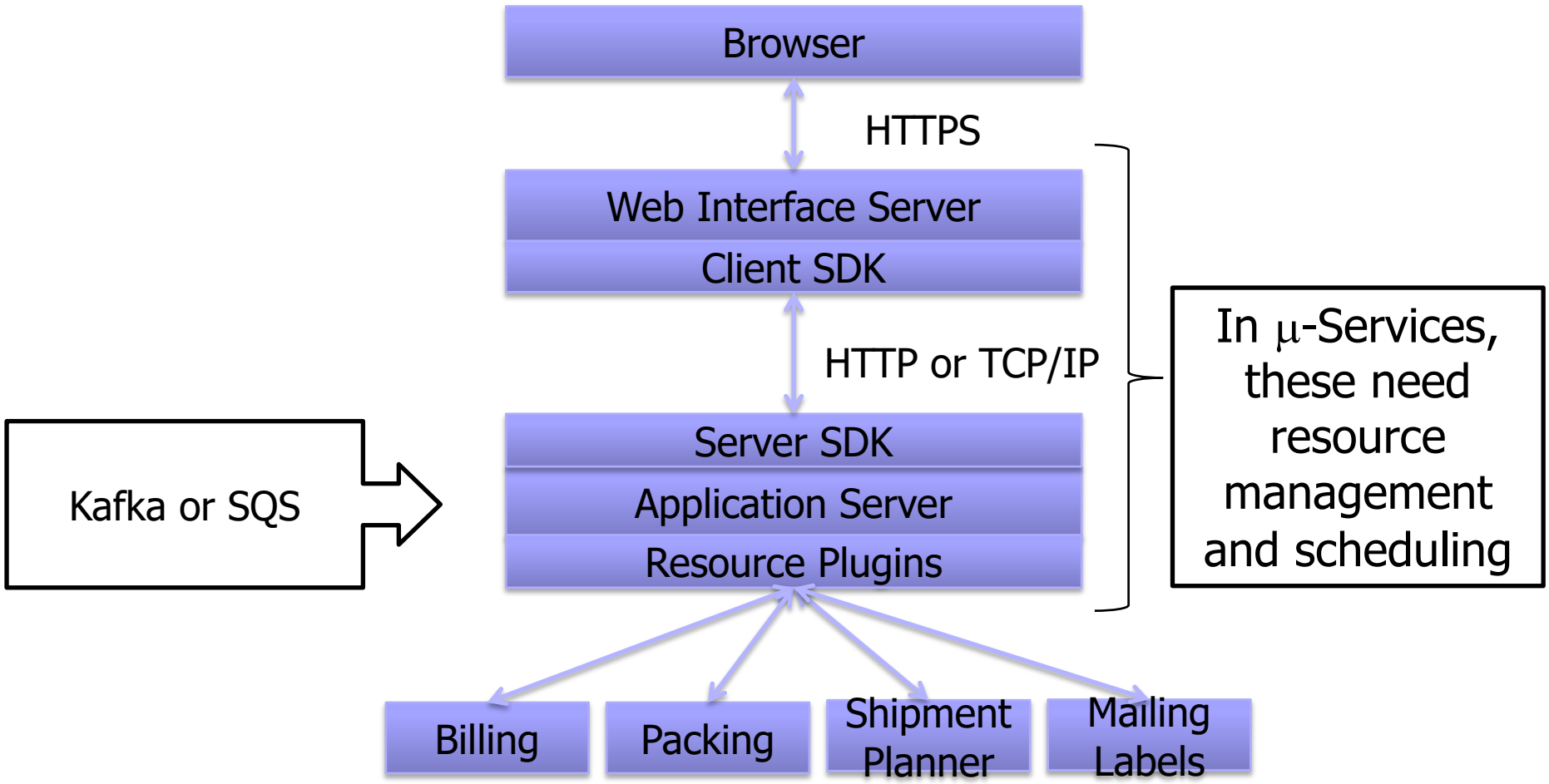
Uses of Zookeeper

- Naming service
  - Identifying nodes in a cluster by name ("DNS" for nodes)
- Configuration management
  - Up-to-date system config info for a joining node
- Cluster management
  - Joining / leaving of nodes, real-time node status
- Leader election
  - Electing a node as leader for coordination purpose
- Locking and synchronization service
- Highly reliable data registry

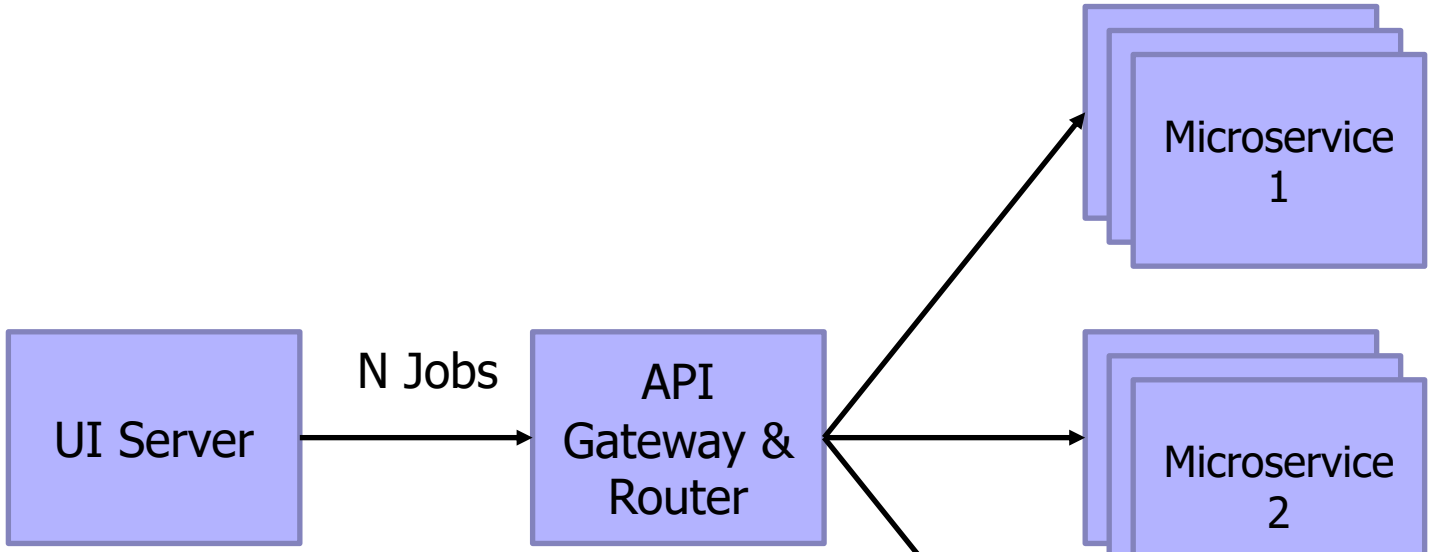
Zookeeper Implementation

- Layered implementation
- Health of all components is tracked, so that we know who is up and who has crashed ("membership status")
- Metadata layer is a single program
  - Consistent by design
- Data replication layer uses atomic multicast to ensure that all replicas are in the same state

Example: Amazon Micro-services



A Simple Micro-service

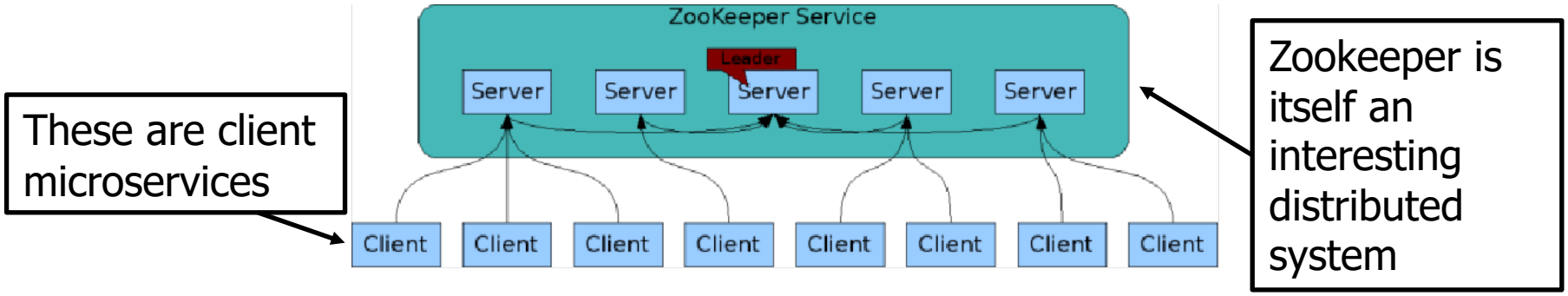


- Is Replica 2 of micro-service 3 up and running?
- Do I have at least one service running?
- Micro-service 3 uses Master-Worker, and the Master just failed. What do I do?
- Replica 2 needs to find configuration information. How can it do that?

Zookeeper Can Manage…

- IP addresses, version numbers, and other configuration information of microservices
- The health of the microservices
- The state of a particular calculation
- Group membership

Zookeeper Architecture



- ZooKeeper Service is replicated over a set of machines
- All machines store a copy of the data **in memory** (!), optional checkpointing to disk
- A leader is elected on service startup
- Client only connects to a single ZooKeeper server & maintains a TCP connection
- Client can read from any Zookeeper server
- Writes go through the leader & need majority consensus.

Data Management in Zookeeper

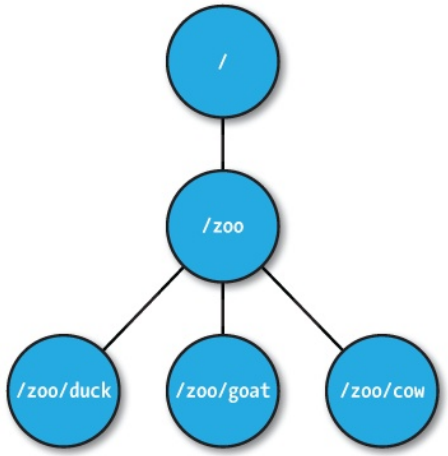
- Tree model for organizing information into nodes
  - Node names may be all you need
  - Lightly structured metadata stored in the nodes
- Wait-free aspects of shared registers with an event-driven mechanism similar to cache invalidations of distributed file systems
- Targets simple metadata systems that read more than they write
  - Small total storage

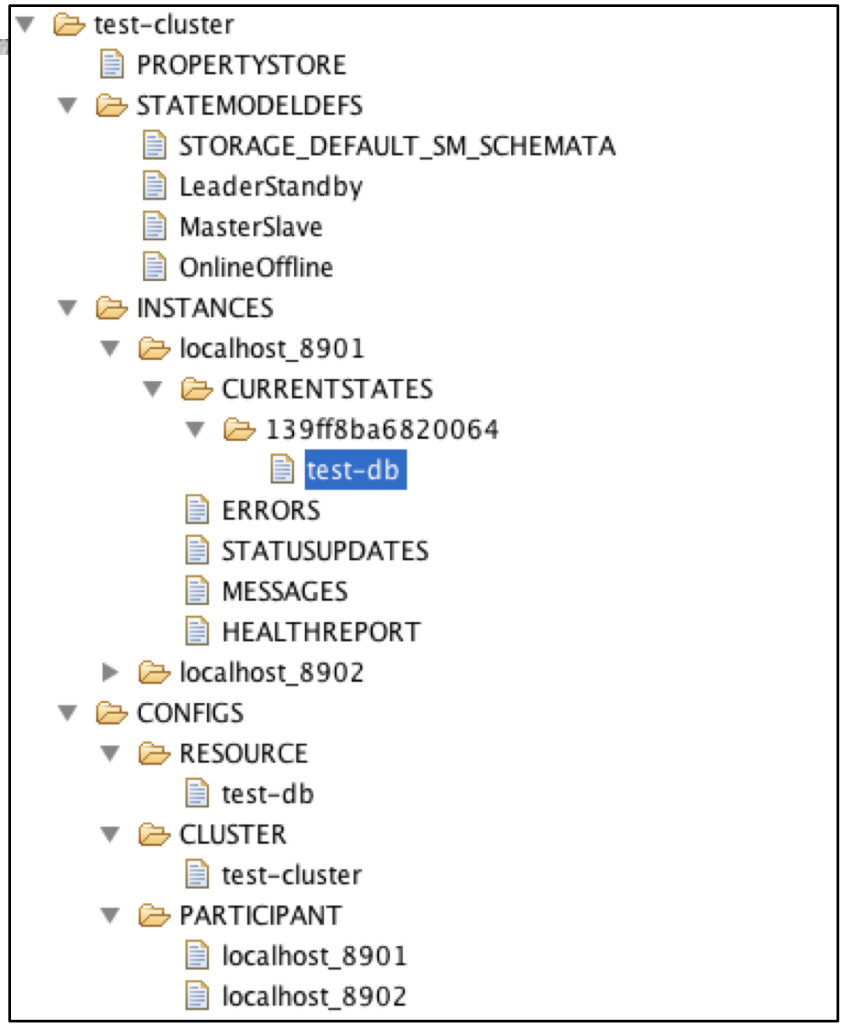
Data Management in Zookeeper

- Clients (i.e., applications) can create and discover nodes on Zookeeper trees
- Clients can put small pieces of data into the nodes and get small pieces out
  - 1 MB max for all data per server by default
  - Each node also has built-in metadata like its version number
- Simple analogy: lock files and .pid files on Linux systems

ZNodes

- Maintain file meta-data with version numbers for data changes, ACL changes and timestamps.
- Version numbers increases with changes
- Data is read and written in its entirety





Znode Types

Regular

- Clients create and delete explicitly

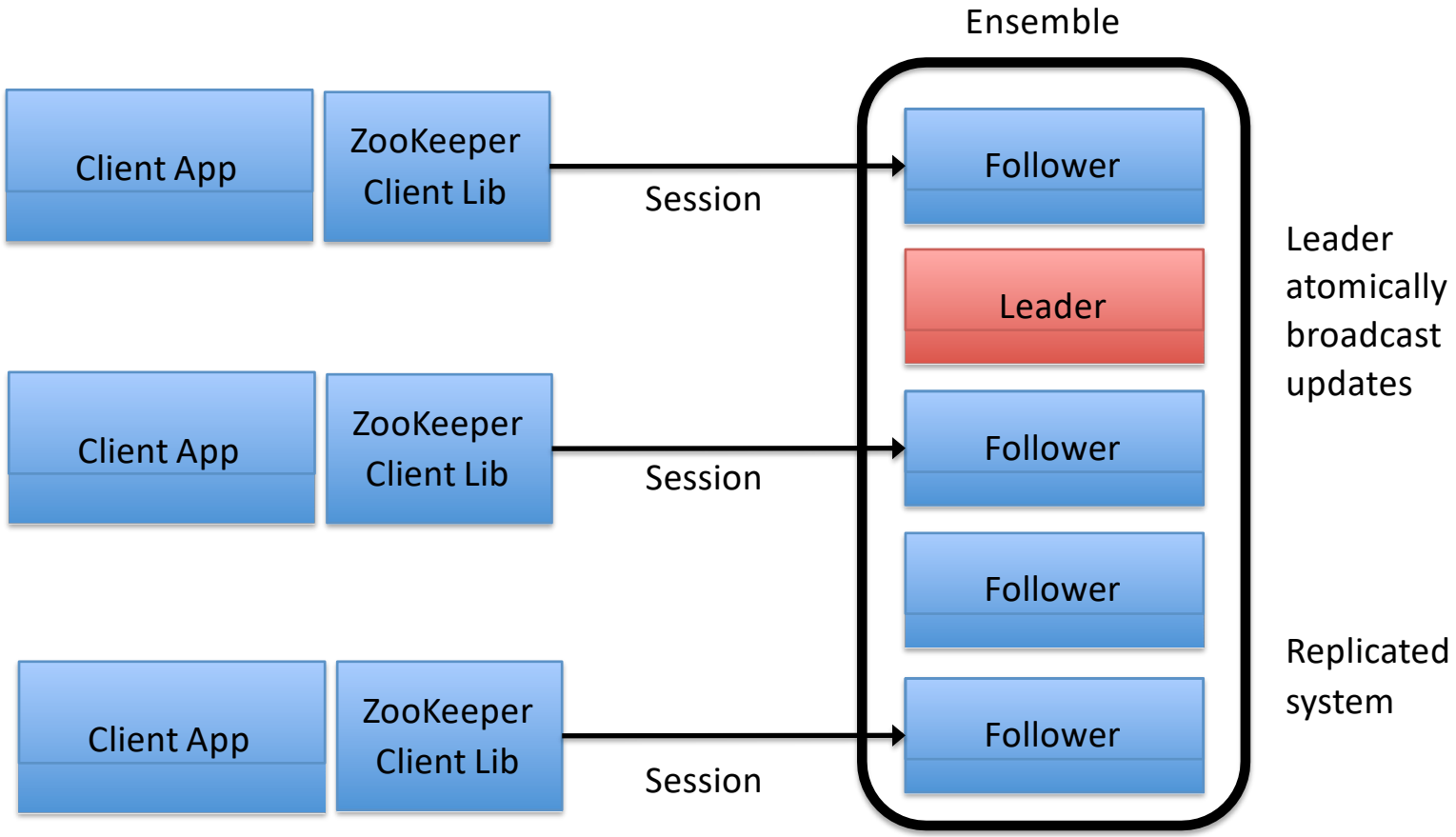
Ephemeral

- Like regular znodes associated with sessions
- Deleted when session expires

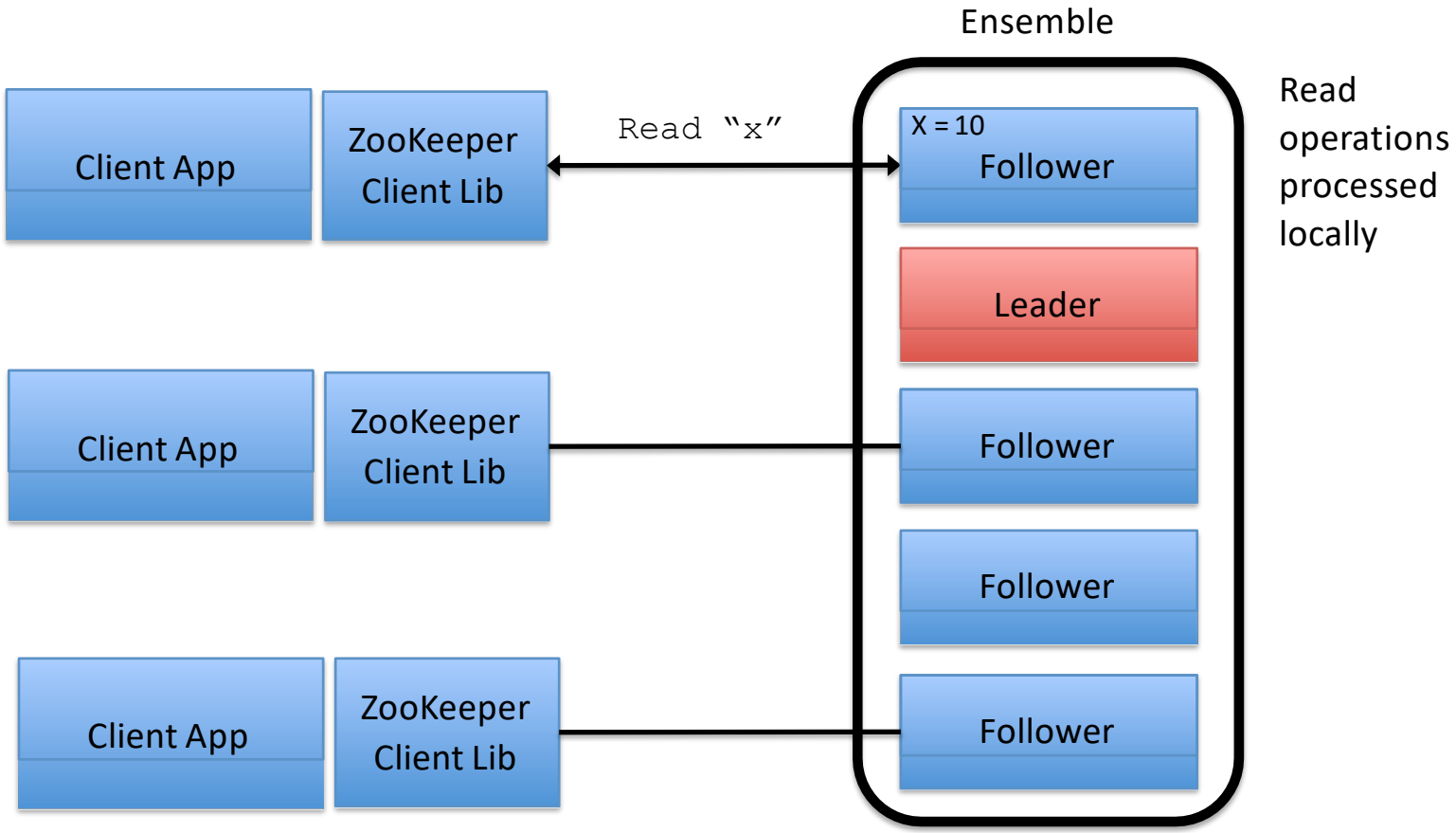
Sequential

- Property of regular and ephemeral znodes
- Has a universal, monotonically increasing counter appended to the name

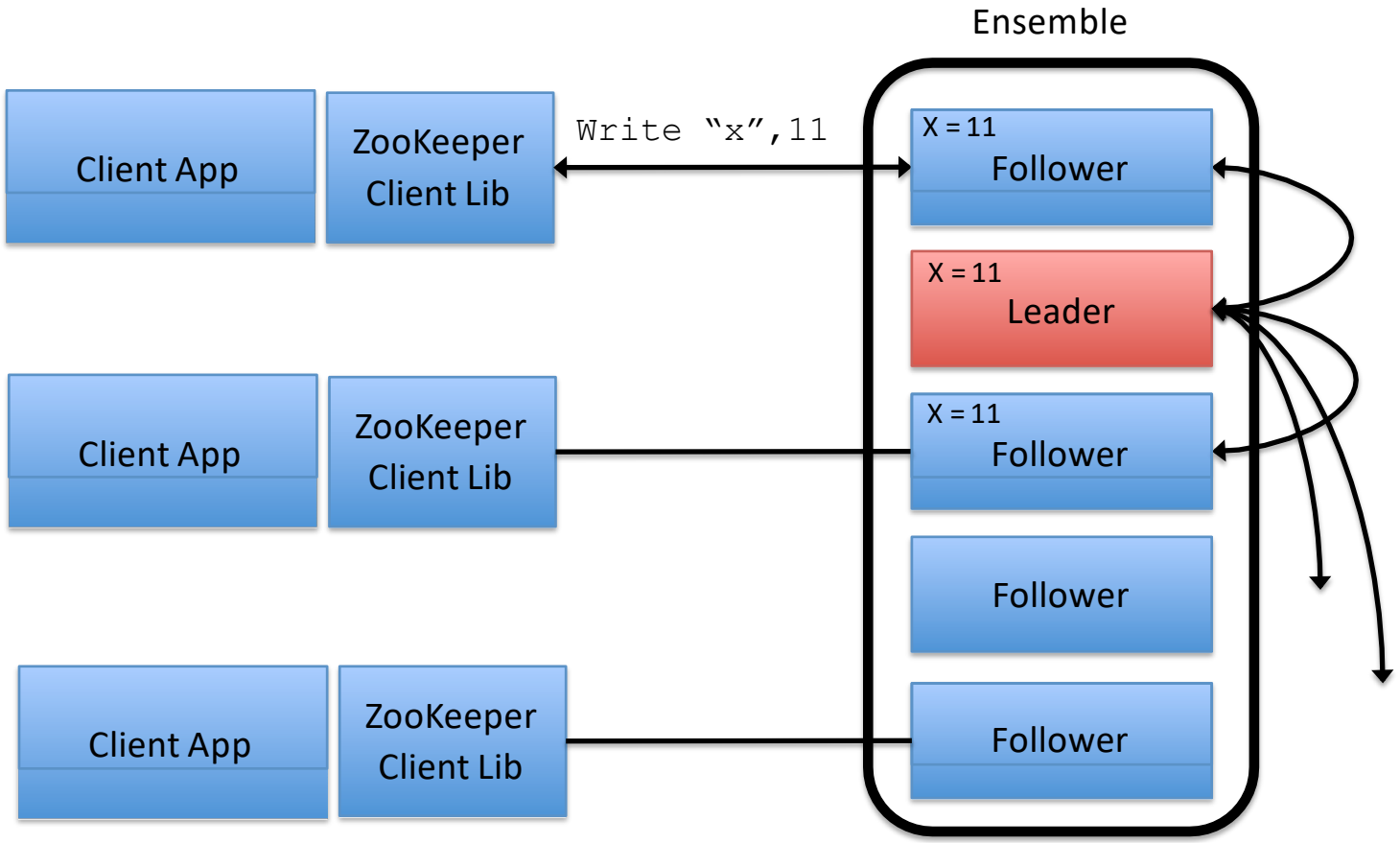
Overview of Zookeeper API



Zookeeper Reads



Zookeeper Writes



Replicates across a quorum

Zookeeper API (1)

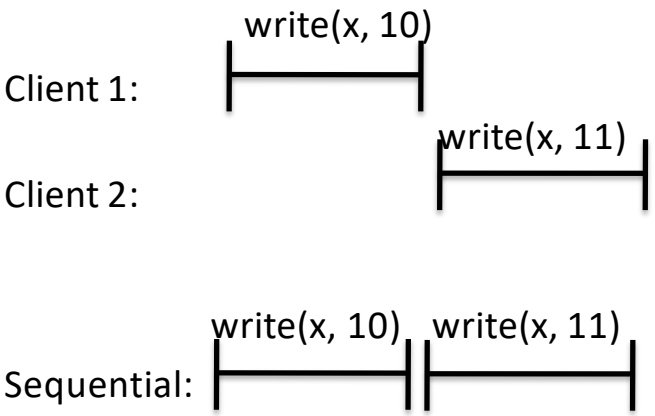
- **create(path, data, flags):** Creates a znode with path name path, stores data[] in it, and returns the name of the new znode
  - *flags* enables a client to select the type of znode: regular or ephemeral, and set the sequential flag;
- **delete(path, version):** Deletes the znode path if that znode is at the expected version
- **exists(path, watch):** Returns true if the znode with path name path exists, false otherwise

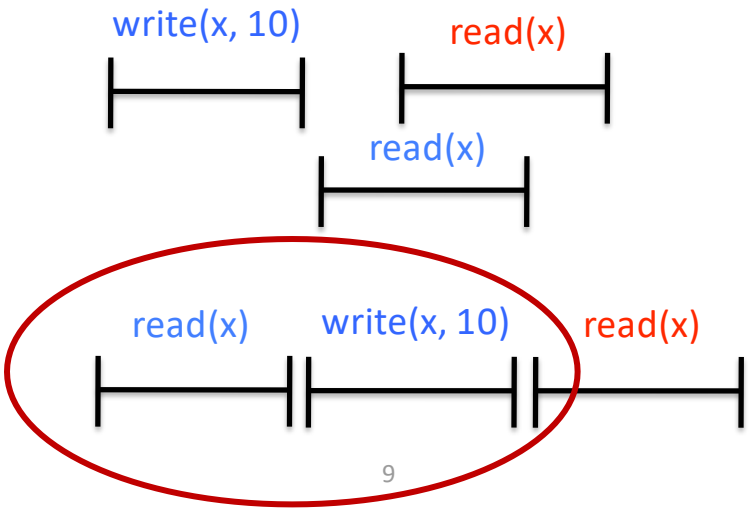
Zookeeper API (2)

- **getData(path, watch):** Returns the data and meta-data, such as version information, associated with the znode
- **setData(path, data, version):** Writes data[] to znode path if the version number is the current version of the znode
- **getChildren(path, watch):** Returns the set of names of the children of a znode
- **sync(path):** Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to

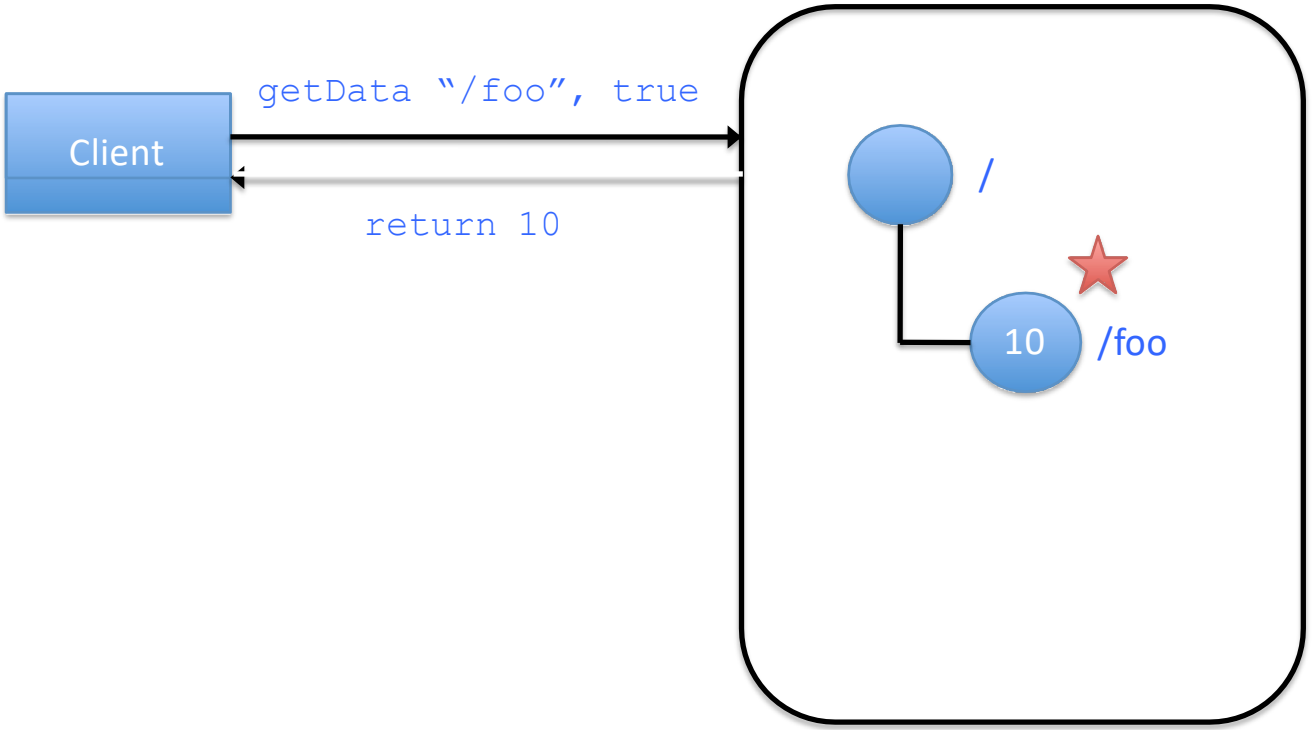
FIFO Order for Client Operations

- Updates: totally ordered, linearizable
- Reads: sequentially ordered

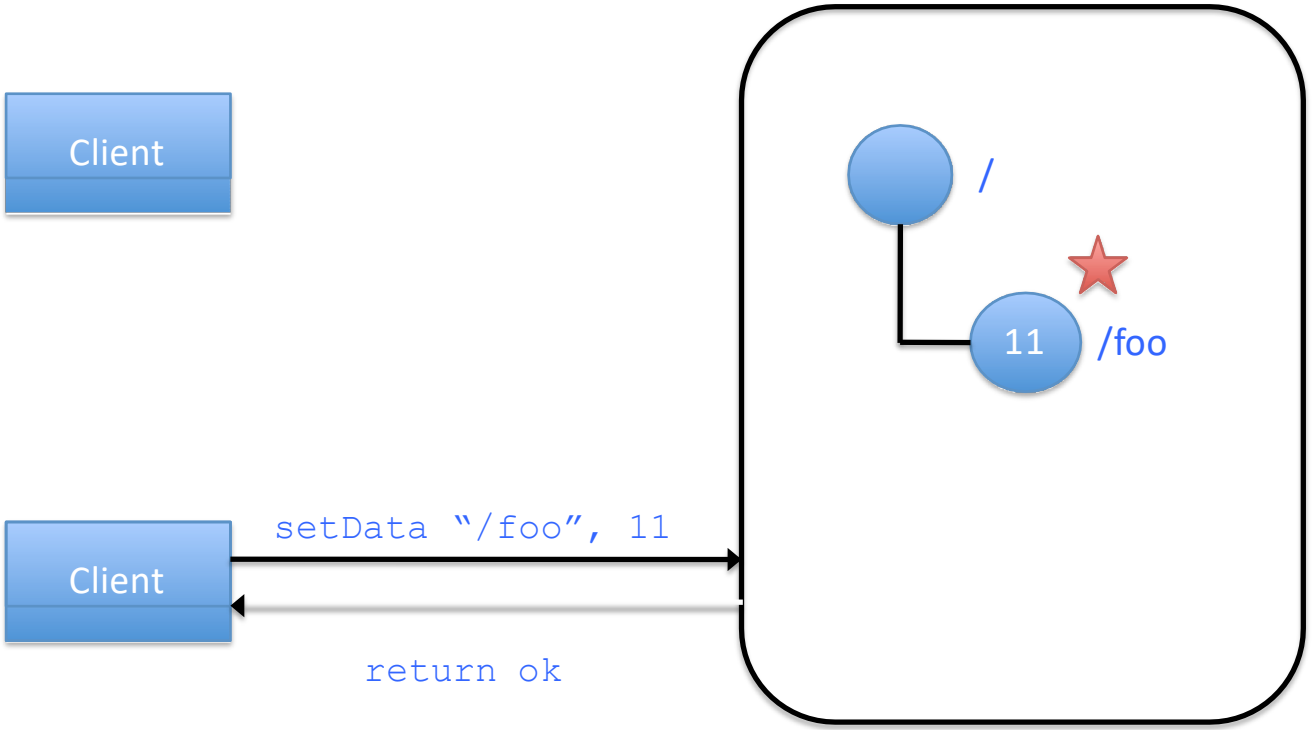




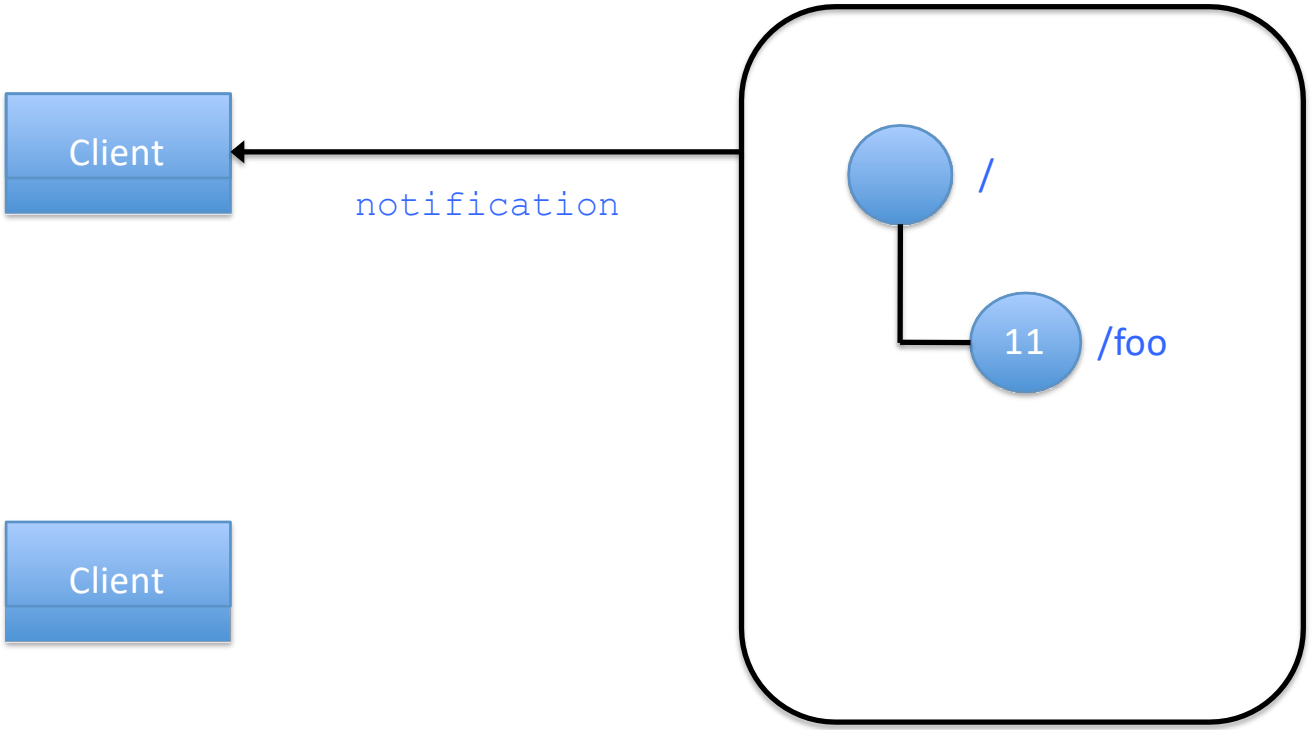
Zookeeper Watch



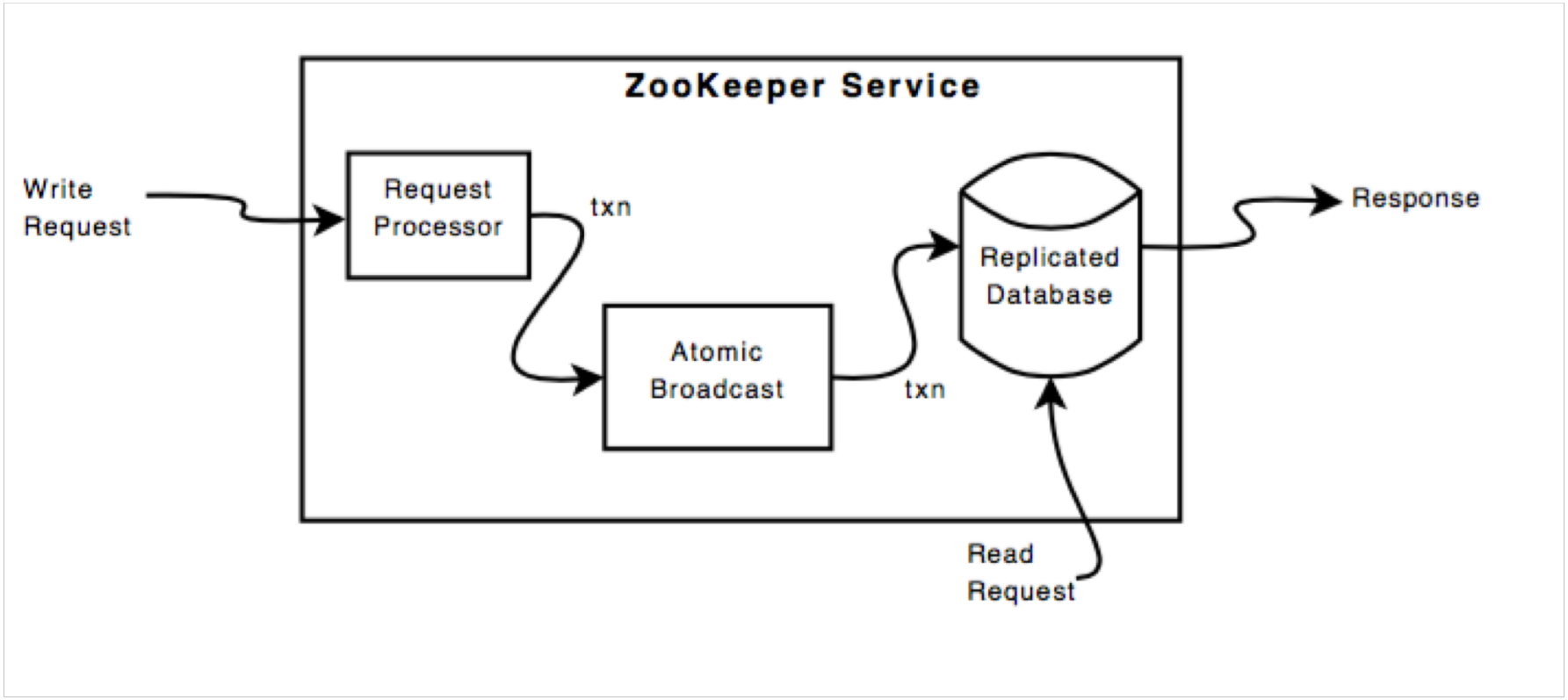
Zookeeper Watch



Zookeeper Watch



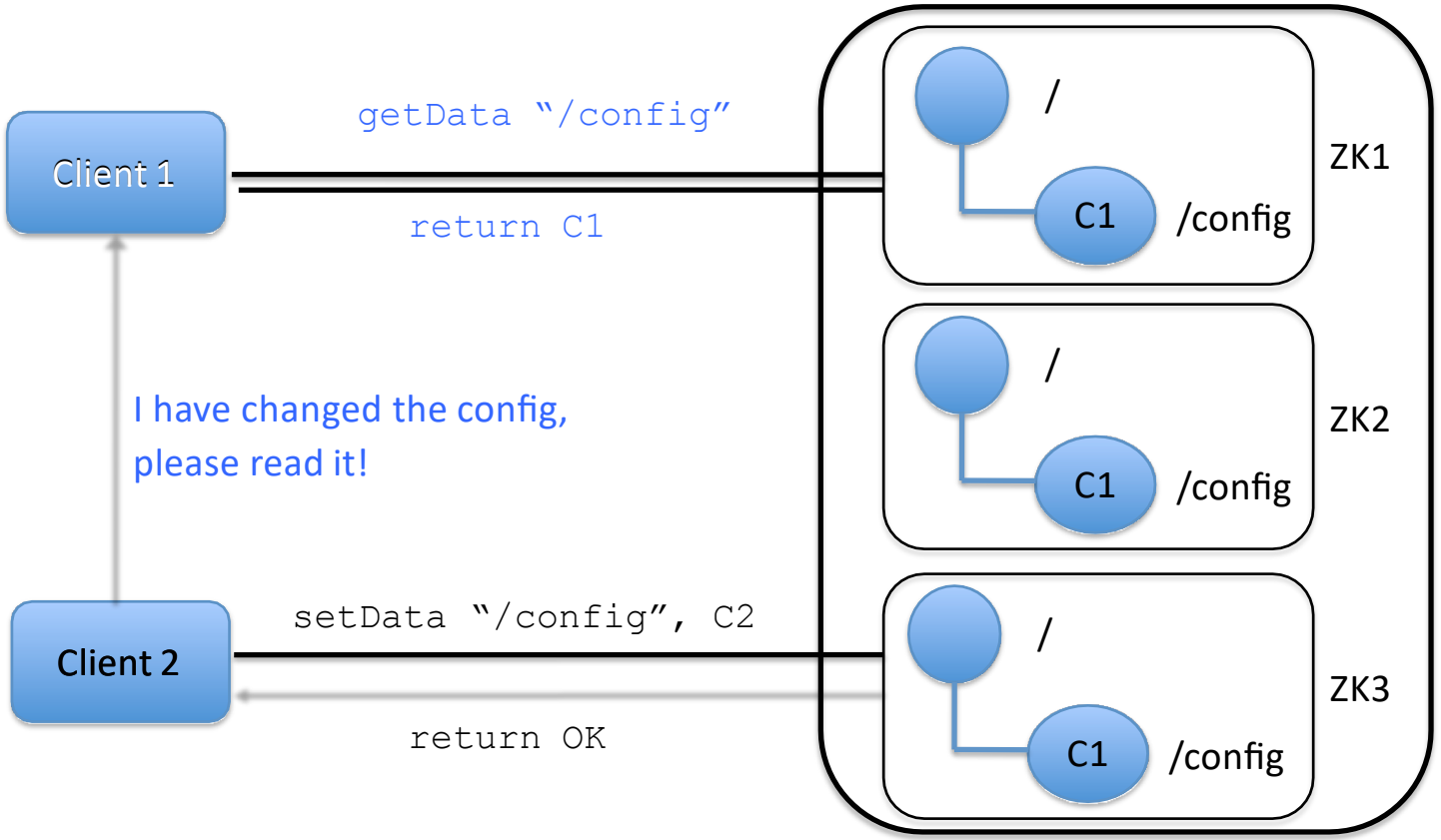
Under the Hood: Zab Protocol



Handling Writes

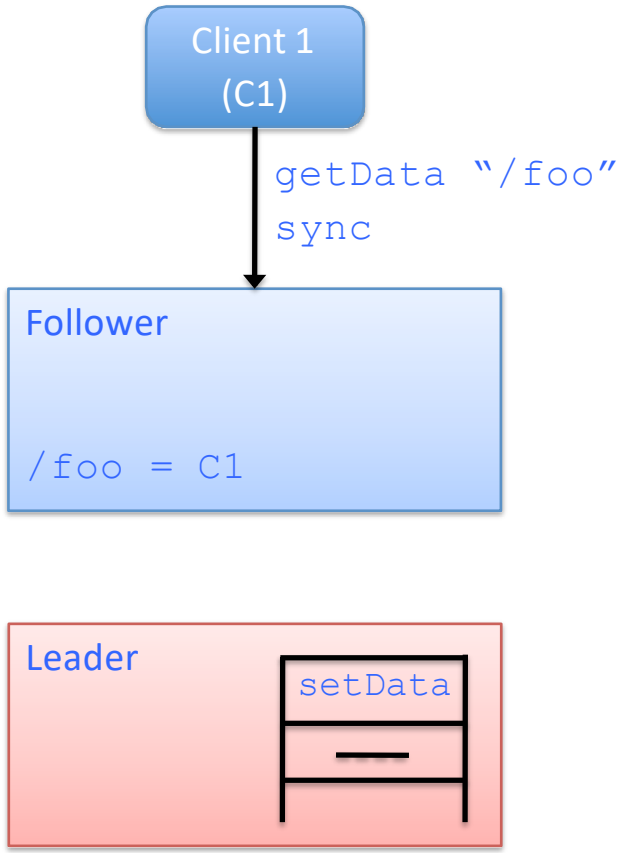
- READ requests are served by any Zookeeper server
  - Scales linearly, although information can be stale
- WRITE requests change state so handled differently (a kind of "consensus")
  - One Zookeeper server acts as the leader
  - The leader executes all write requests forwarded by followers
  - The leader then broadcasts the changes
  - The update is successful if a majority of Zookeeper servers have correct state at the end of the protocol

Hidden Channels



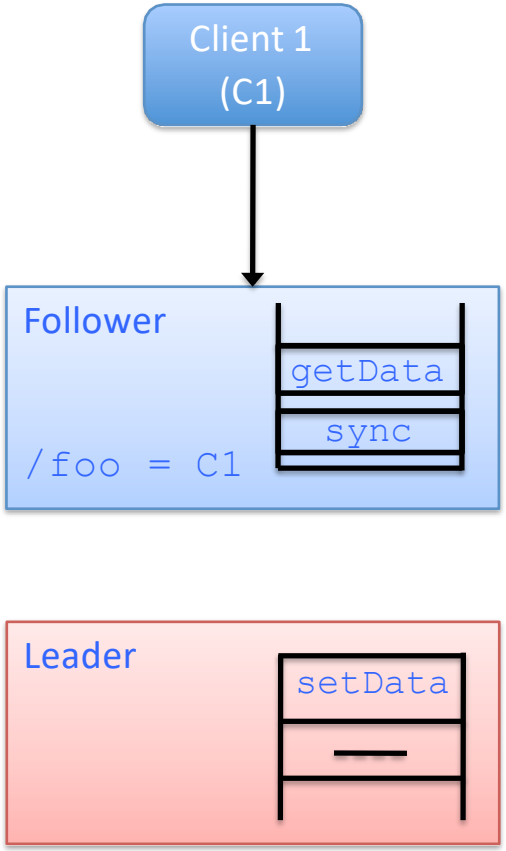
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



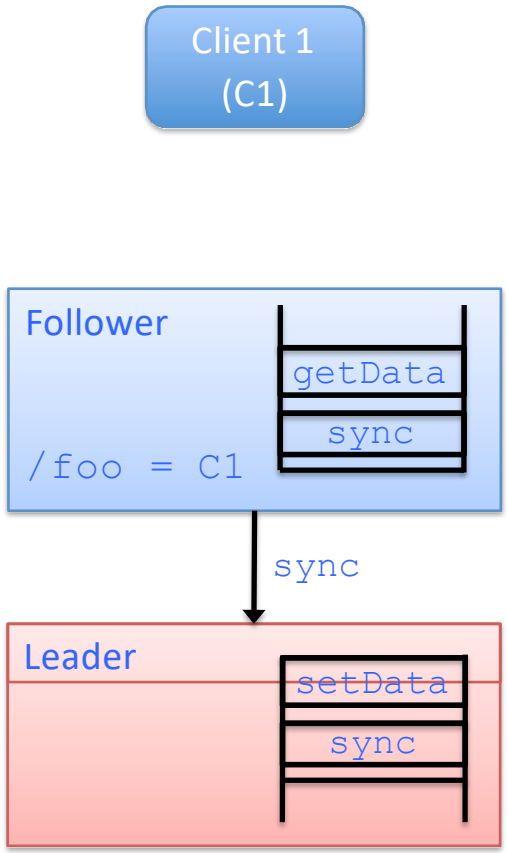
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



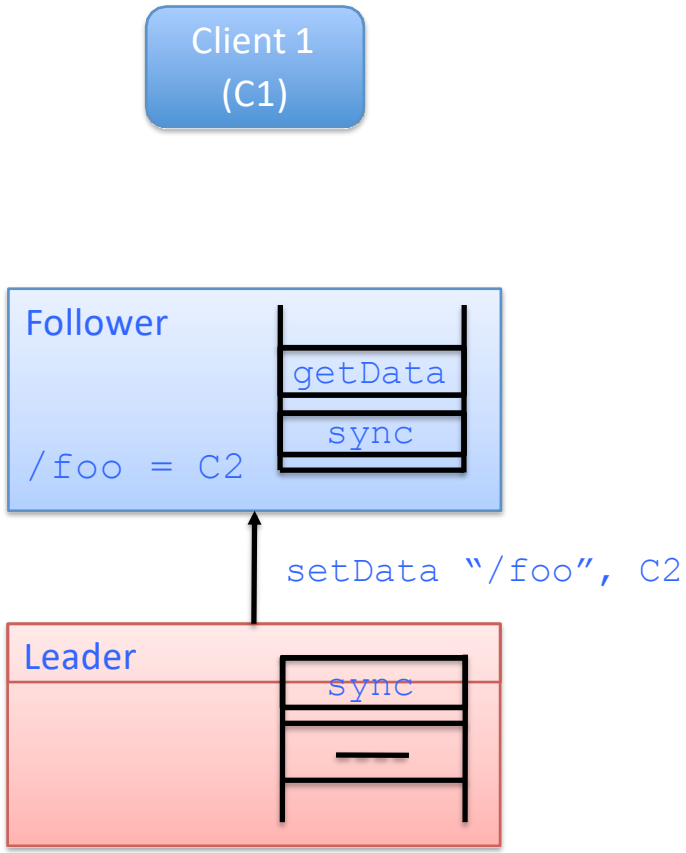
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



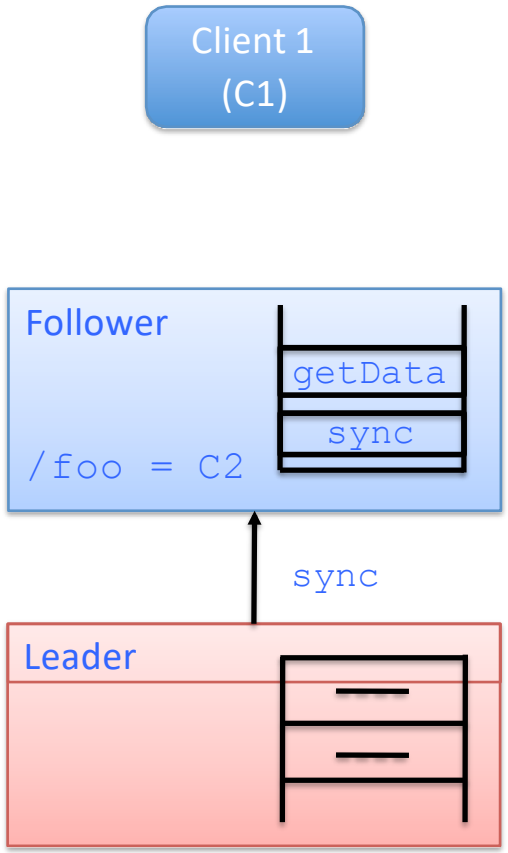
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



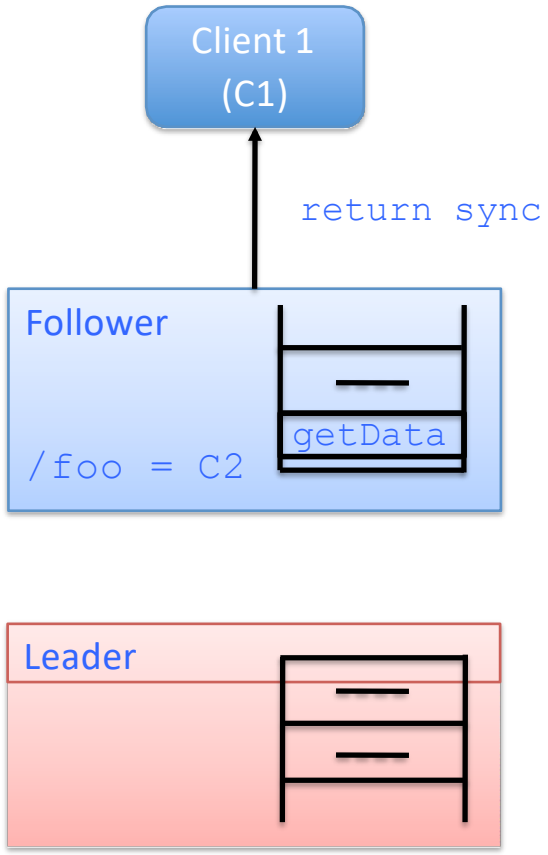
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



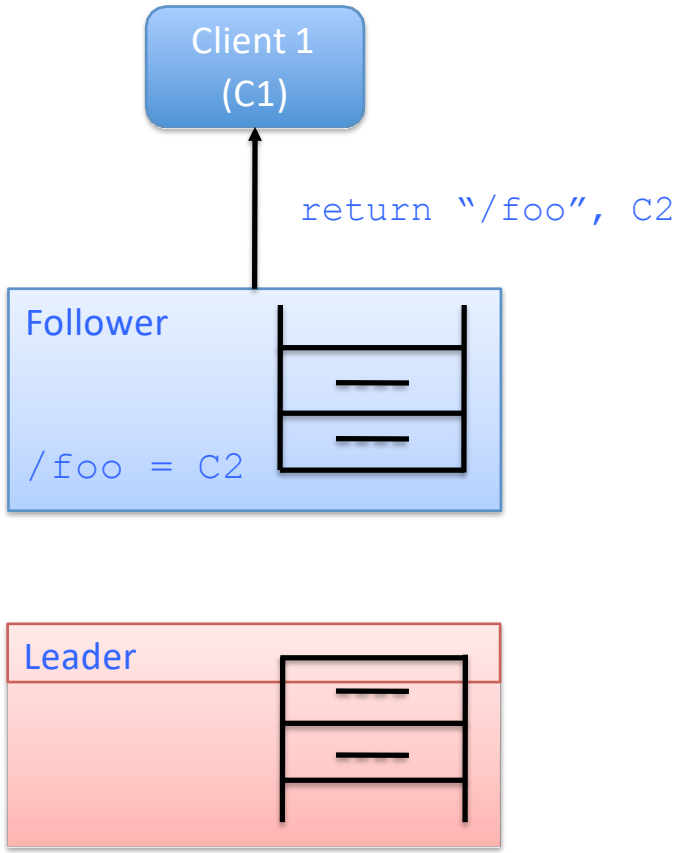
sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



sync

- Asynchronous operation
- Before read operations
- Flushes the channel between follower and leader
- Makes operations linearizable



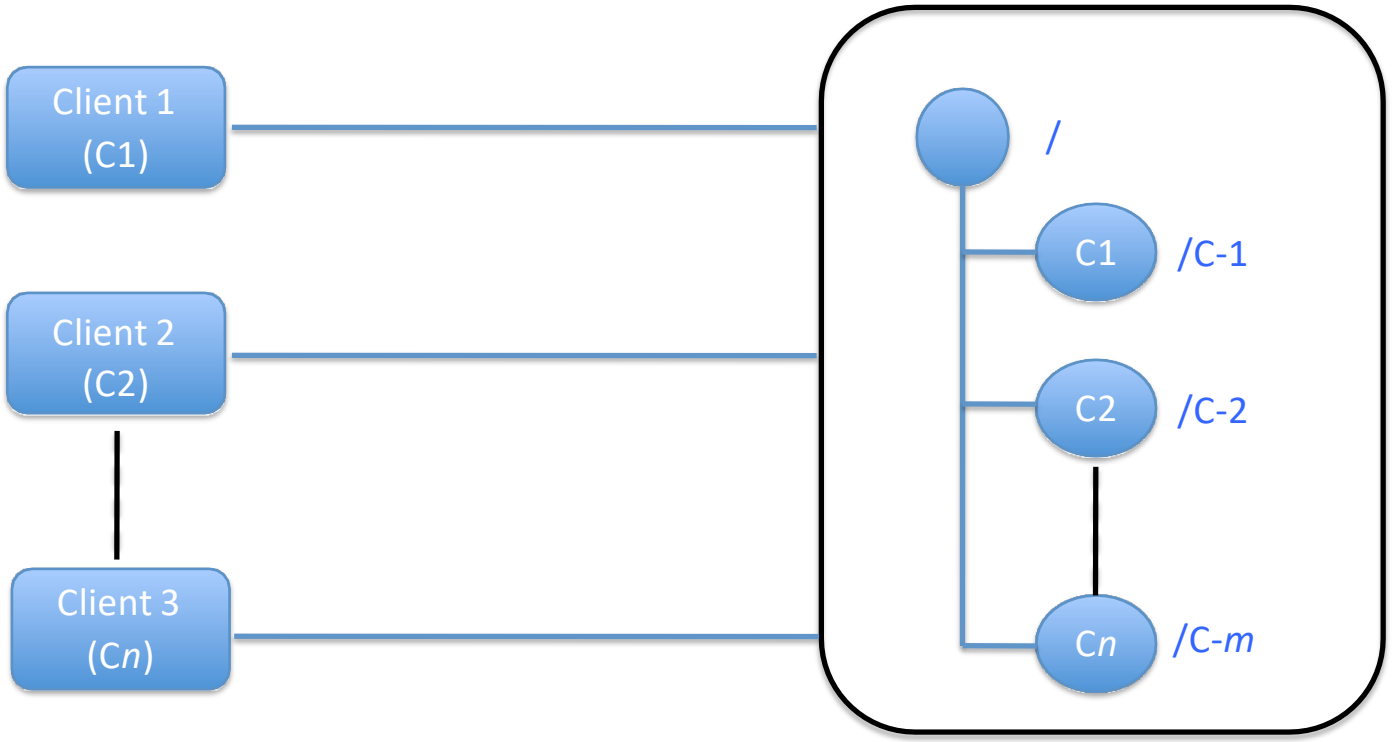
Implementation Simplifications

- Uses TCP for its transport layer
  - Message order is maintained by the (reliable?) network
- Assumes reliable file system
  - Logging and DB checkpointing
- Write-ahead logging
  - Requests are first written to the log
  - The Zookeeper DB is updated from the log
  - Zookeeper servers can acquire correct state by reading the logs from the file system
    - With checkpoints, need not reread the entire history
- Assumes a single administrator so no deep security

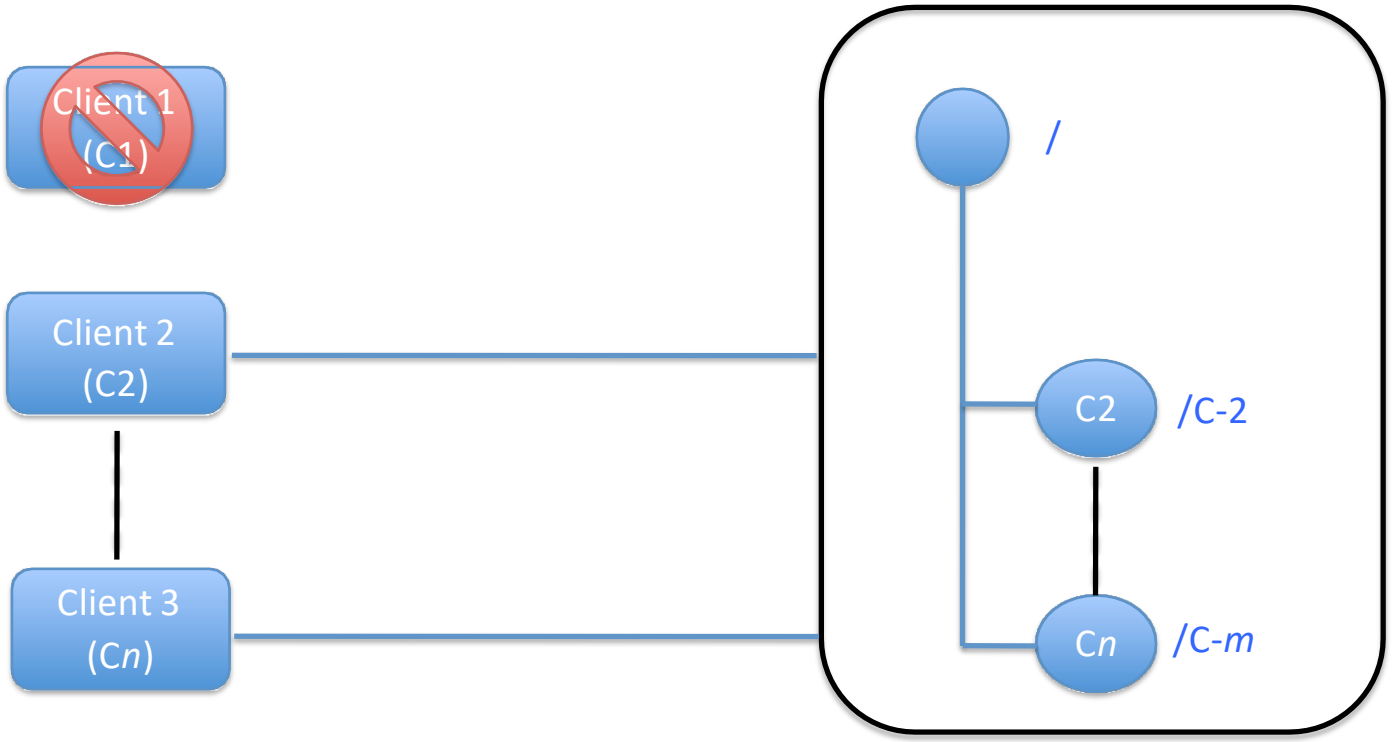
Locks

- Zookeeper example: who is the leader with primary copy of data?
- Implementation:
  - Leader creates an ephemeral file: /root/leader/lockfile
  - Other would-be leaders place watches on the lock file
  - If the leader client dies or doesn't renew the lease, clients can attempt to create a replacement lock file
- Use SEQUENTIAL to solve the herd effect problem
  - Create a sequence of ephemeral child nodes
  - Clients only watch the node immediately ahead of them in the sequence

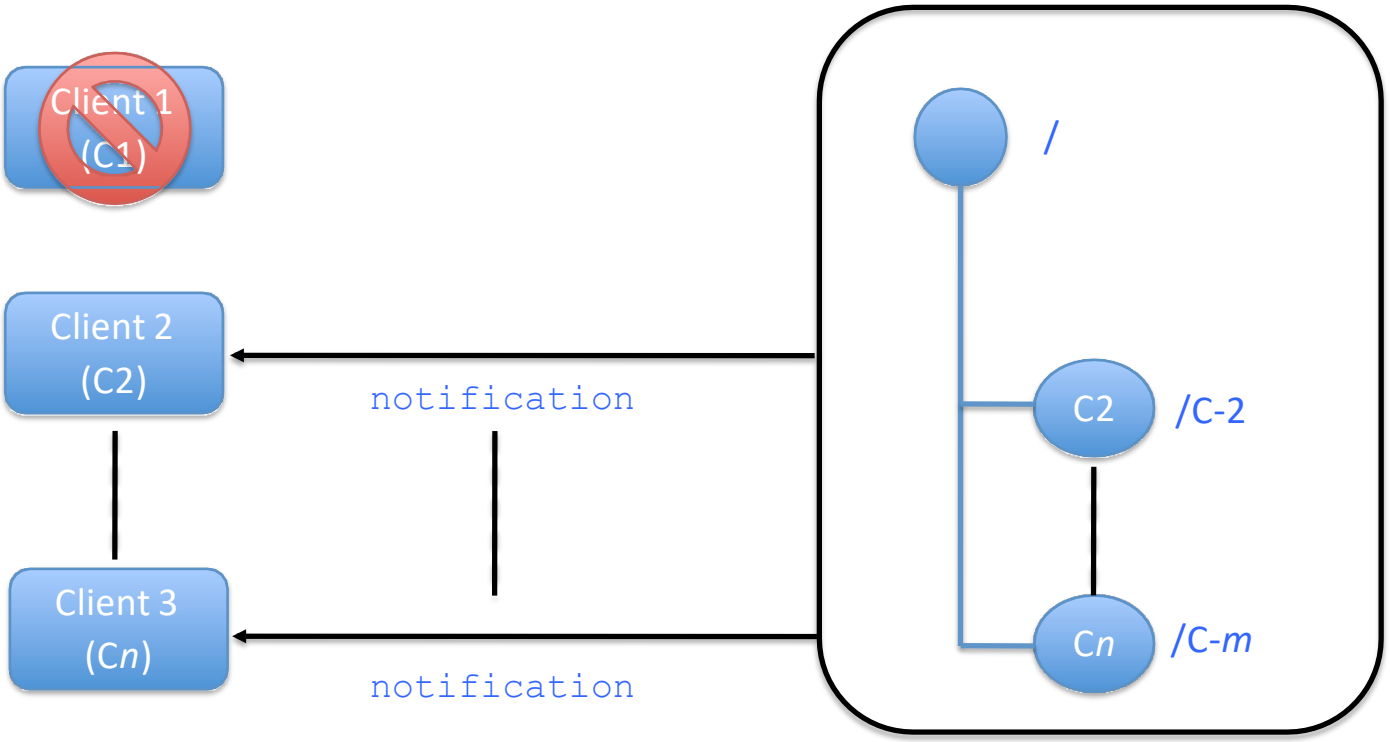
Herd Effect



Herd Effect



Herd Effect

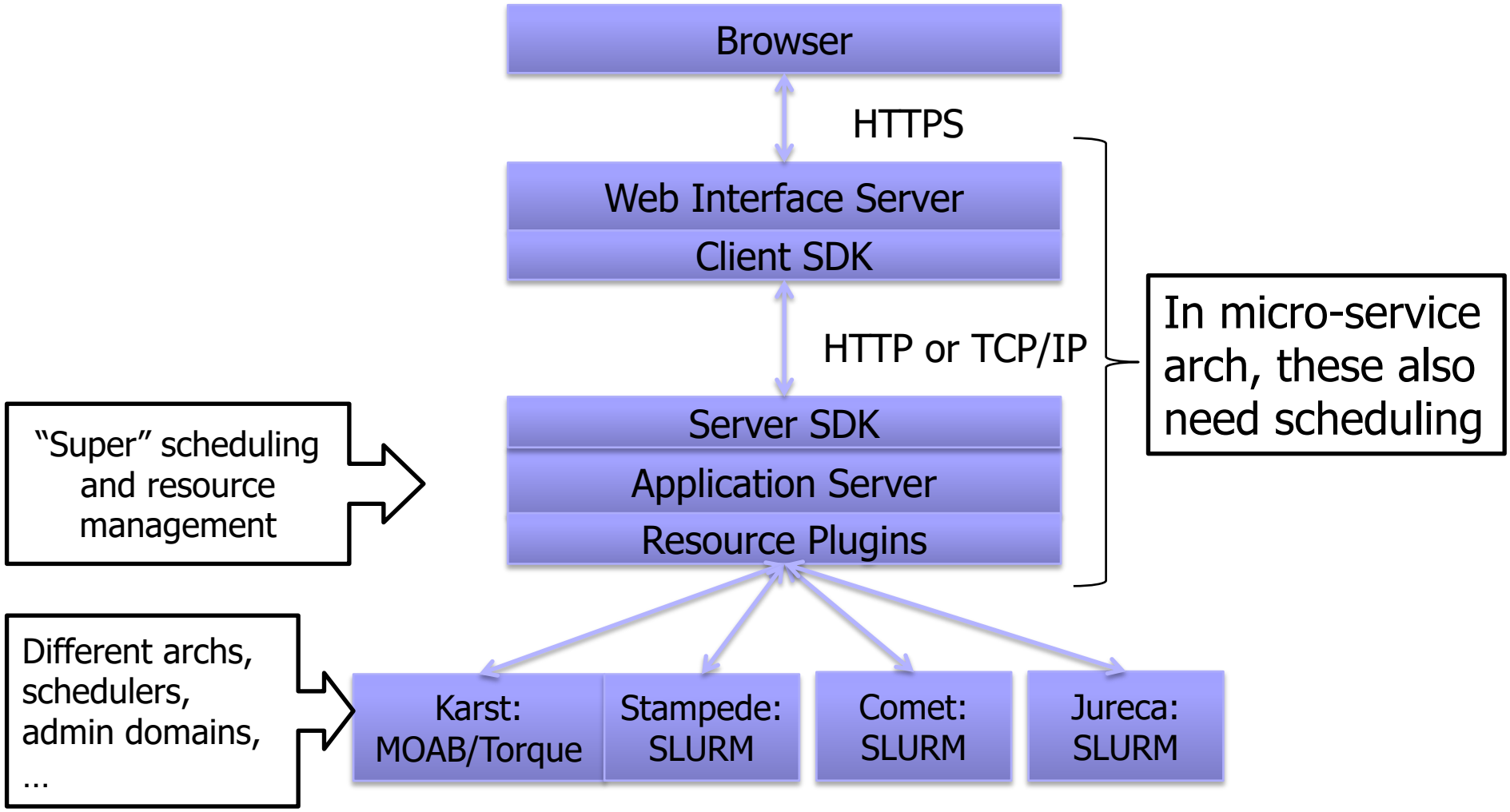


Load spike

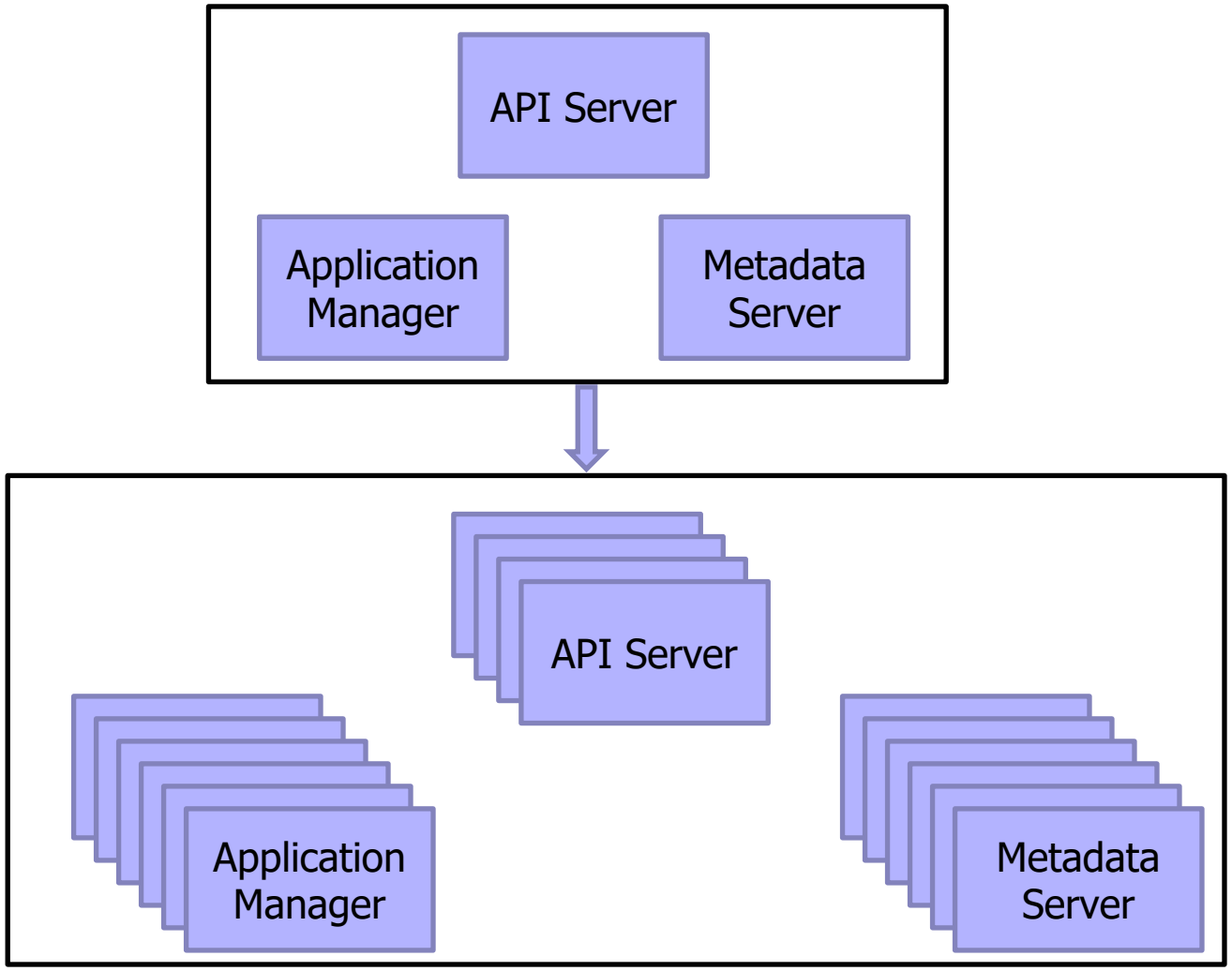
Solving the Herd Effect

- Use order of clients
- Each client:
  - Determines the znode z preceding its own znode in the sequential order
  - Watches z
- A single notification is generated upon a crash
- Disadvantage for leader election
  - One client is notified of a leader change

Zookeeper in Scientific Computing



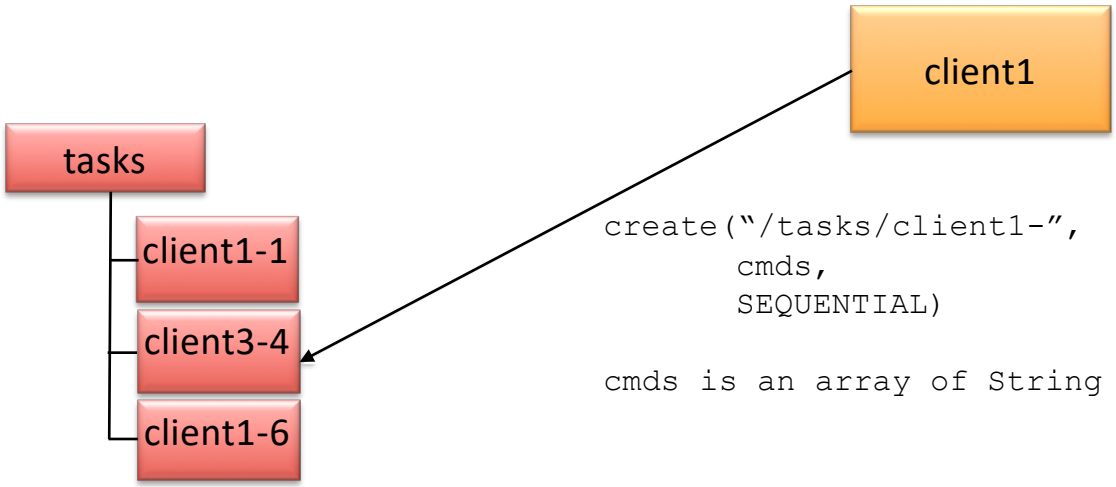
Replicating Components



Why Replication?

- Fault tolerance
- Increased throughput, load balancing
- Component versions
  - Not all components of the same type need to be on the same version
  - Backward compatibility checking
- Component flavors
  - Application managers can serve different types of resources
  - Useful to separate them into separate processes if libraries conflict.

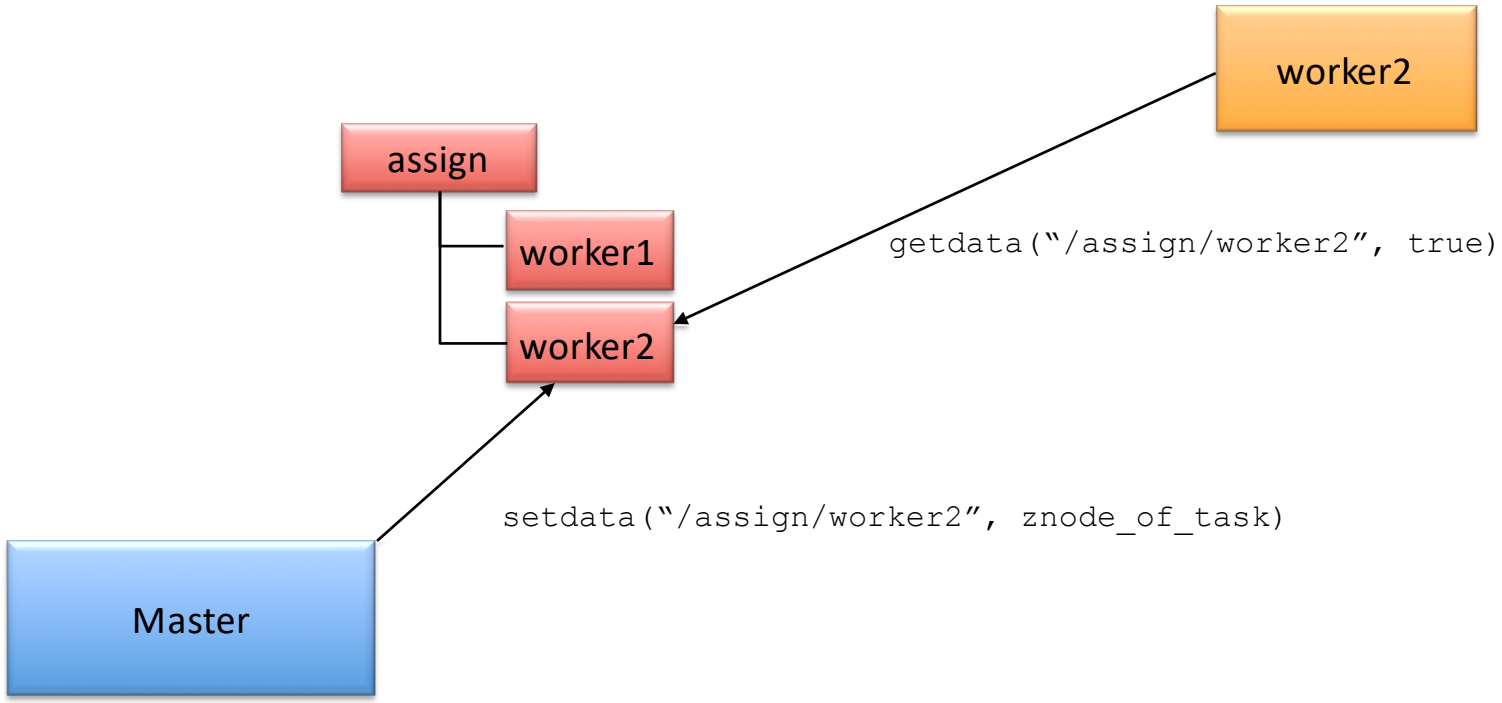
Task Queue


client1
tasks
client1-1
client3-4
client1-6
create("/tasks/client1-",
cmds,
SEQUENTIAL)
cmds is an array of String

Configuration Management

- Problem: gateway components in a distributed system need to get the correct configuration file
- Solution: Components contact Zookeeper to get configuration metadata
  - This includes the component's own configuration file as well as configurations for other components
  - Rendezvous problem

Configuration Management



Configuration Management

- All clients get their configuration information from a named znode
  - /root/config-me
- Example: build a public key store with Zookeeper
- Clients set watches to see if configurations change
- Zookeeper doesn't explicitly decide which clients are allowed to update the configuration
  - That would be an implementation choice
  - Zookeeper uses leader-follower model internally, so you could model your own implementation after this

The Rendezvous Problem

- Classic distributed computing algorithm
  - Consider master-worker: specific configurations (IP addrs, port numbers) may not be known until runtime
  - Workers and master may start in any order
- Zookeeper implementation
  - Create a rendezvous node: /root/rendezvous
  - Workers read /root/rendezvous and set a watch
    - If empty, use watch to detect when master posts its configuration information
  - Master fills in its configuration information (host, port)
  - Workers are notified of content change and get the configuration information

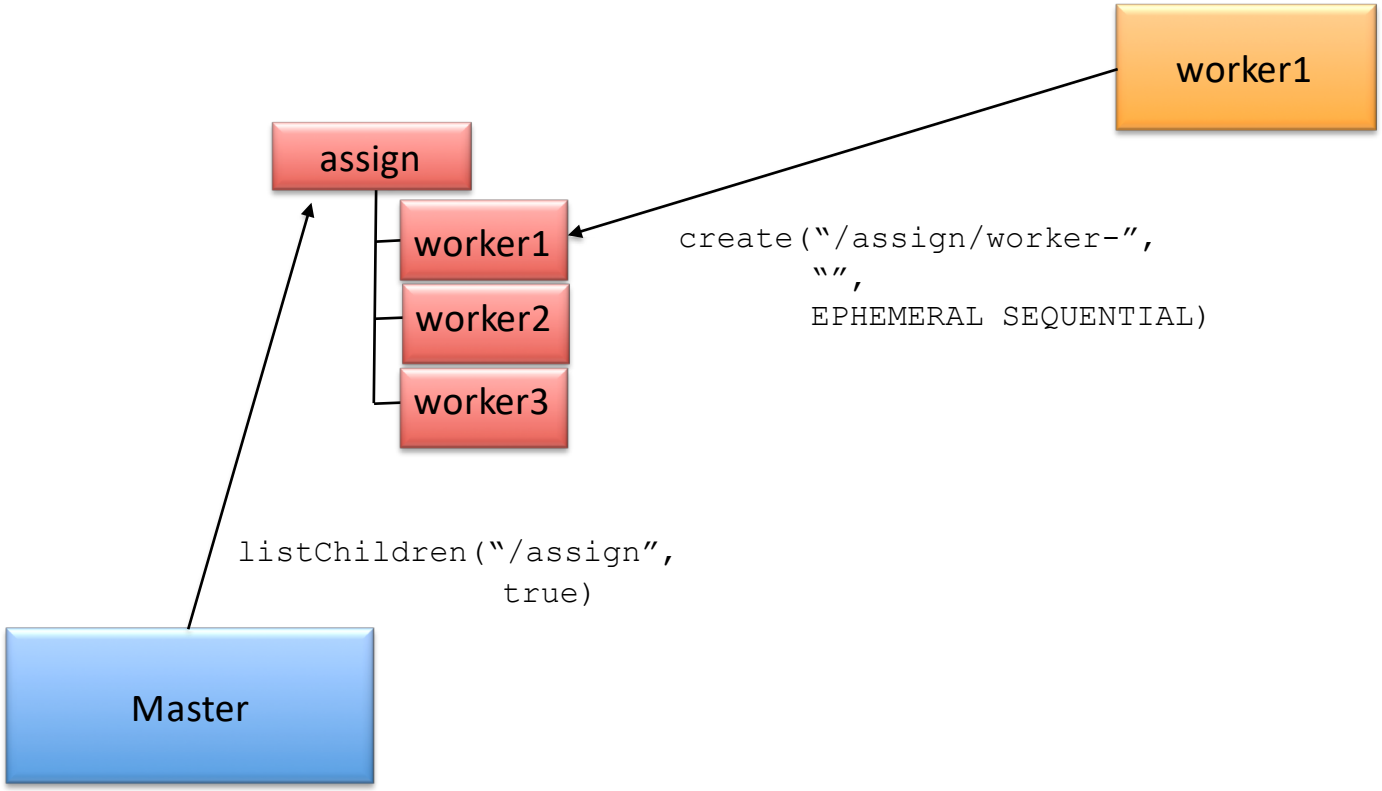
Service Discovery

- Problem: Component A needs to find instances of Component B
- Solution: Use Zookeeper to find available group members instances of Component B
  - More: get useful metadata about Component B instances like version, domain name, port #, flavor
- Useful for components that need to directly communicate but not for asynchronous communication (message queues)

Group Membership

- Problem: a job needs to go to a specific flavor of application manager. How can this be located?
- Solution: have application managers join the appropriate Zookeeper managed group when they come up
- Useful to support scheduling

Group Membership



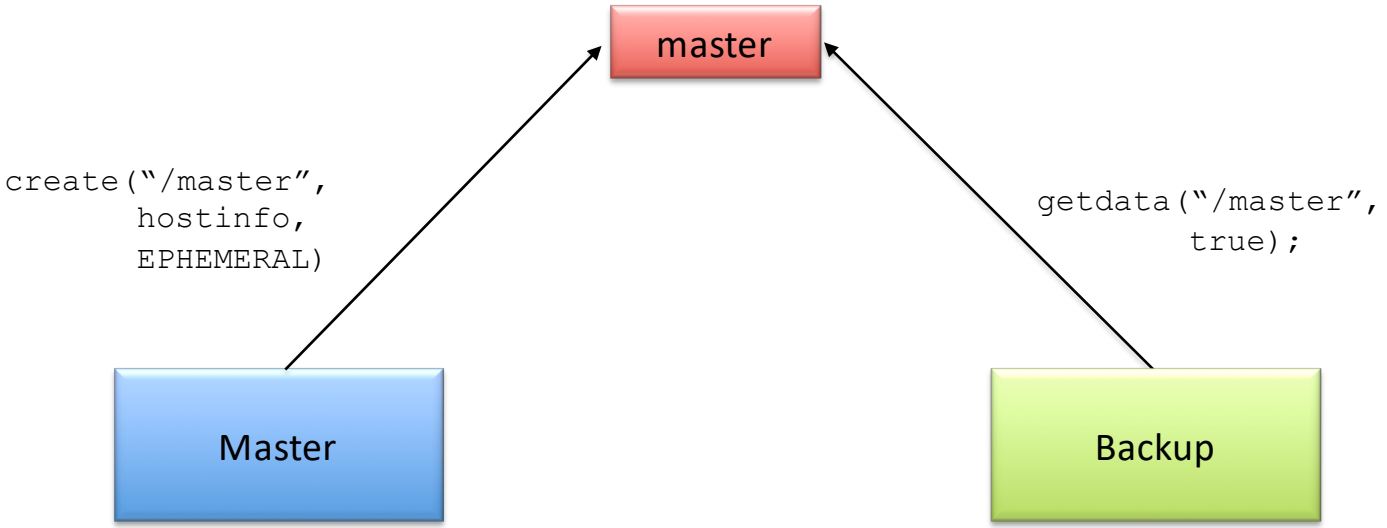
System State for Distributed Systems

- Which servers are up and running? What versions?
- Services that run for long periods could use Zookeeper to indicate if they are busy (or under heavy load) or not

Leader Election

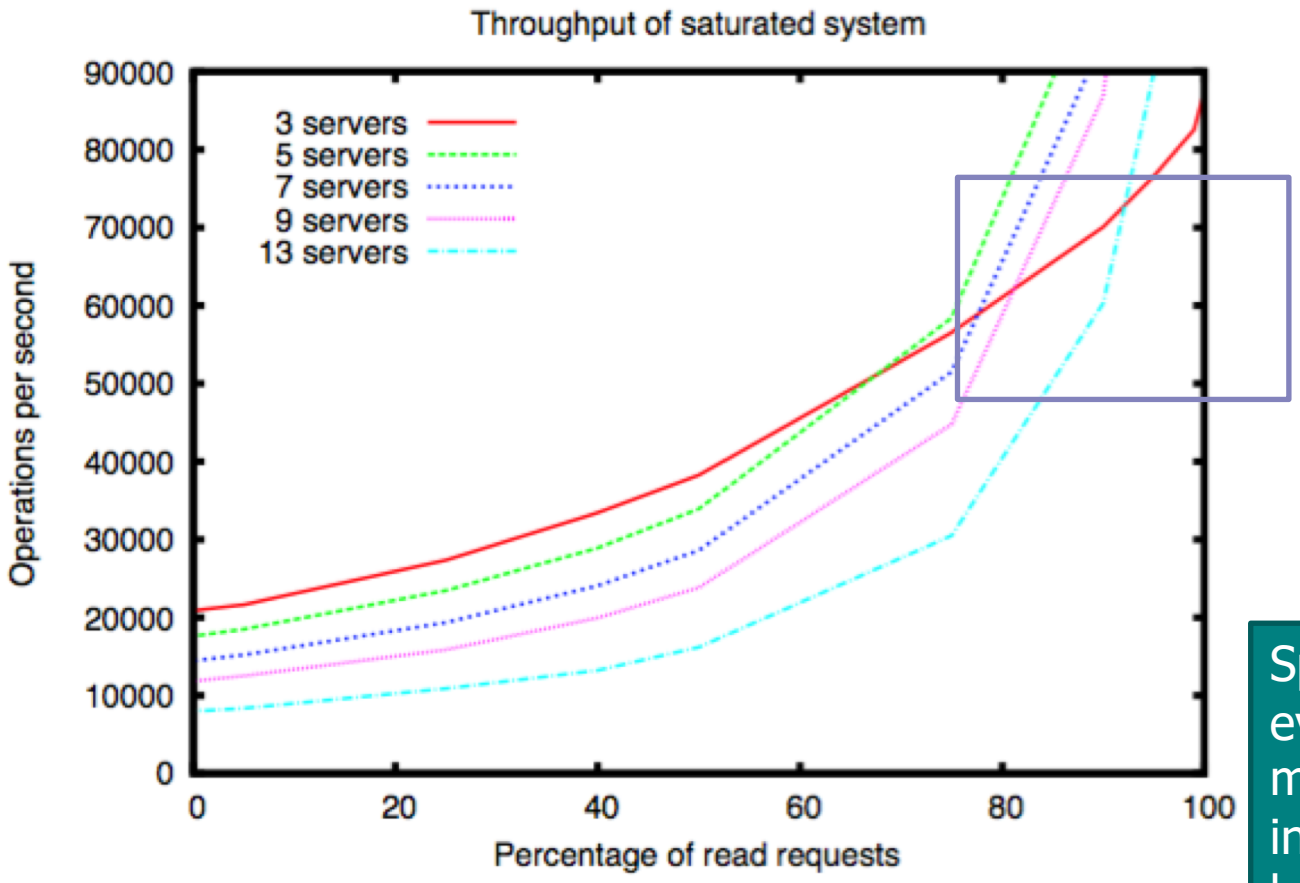
- Problem: metadata servers are replicated for read access but only the master has write privileges. The master crashes.
- Solution: Use Zookeeper to elect a new metadata server leader
- May not be the best way to do this…

Leader Election

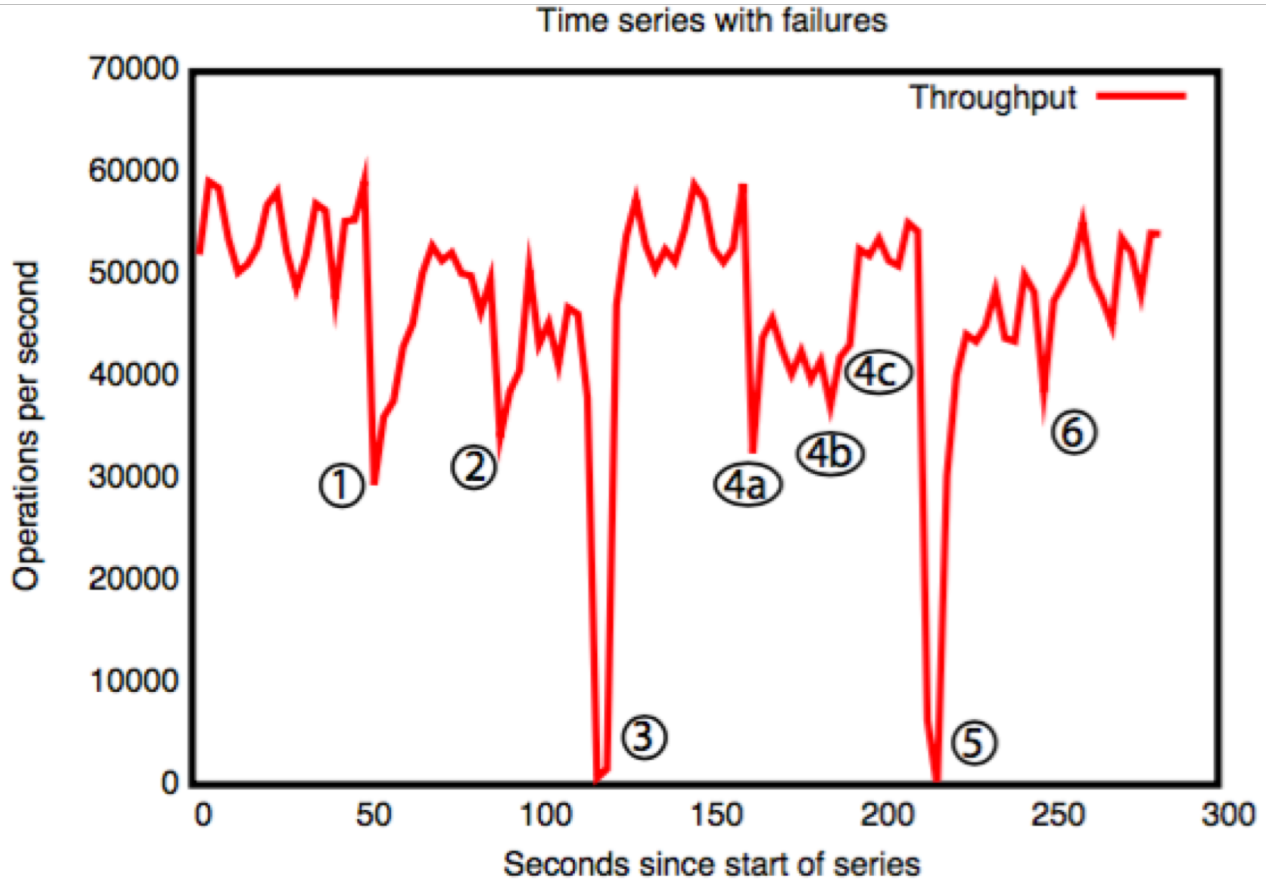


Implementing Consensus

- Each process p proposes then decides
- Propose(v)
  - setData “/c/proposal-”, “v”, sequential
- Decide()
  - getChildren “/c”
  - Select znode z with smallest sequence number
  - v’ = getData “/c/z”
  - Decide upon v’



Speed isn't everything. Having many servers increases reliability but decreases throughput as # of writes increases.



1. Failure and recovery of follower.
2. Failure and recovery of follower.
3. Failure of leader (200 ms to recover).
4. Failure of two followers (4a and 4b), recovery at 4c.
5. Failure of leader
6. Recovery of leader (?)

A cluster of 5 Zookeeper instances responds to manually injected failures.

Zookeeper vs. Paxos

- Zookeeper is solving the state machine replication problem – similar to Paxos
- Zookeeper's Zab is similar to the Paxos concept of an "Atomic Multicast" (aka "Vertical Paxos")
- Checkpointing every 5s is not the same as the true durable Paxos. Durable Paxos is like checkpointing on every operation.