

# Some Remarks on RabbitMQ Clustering

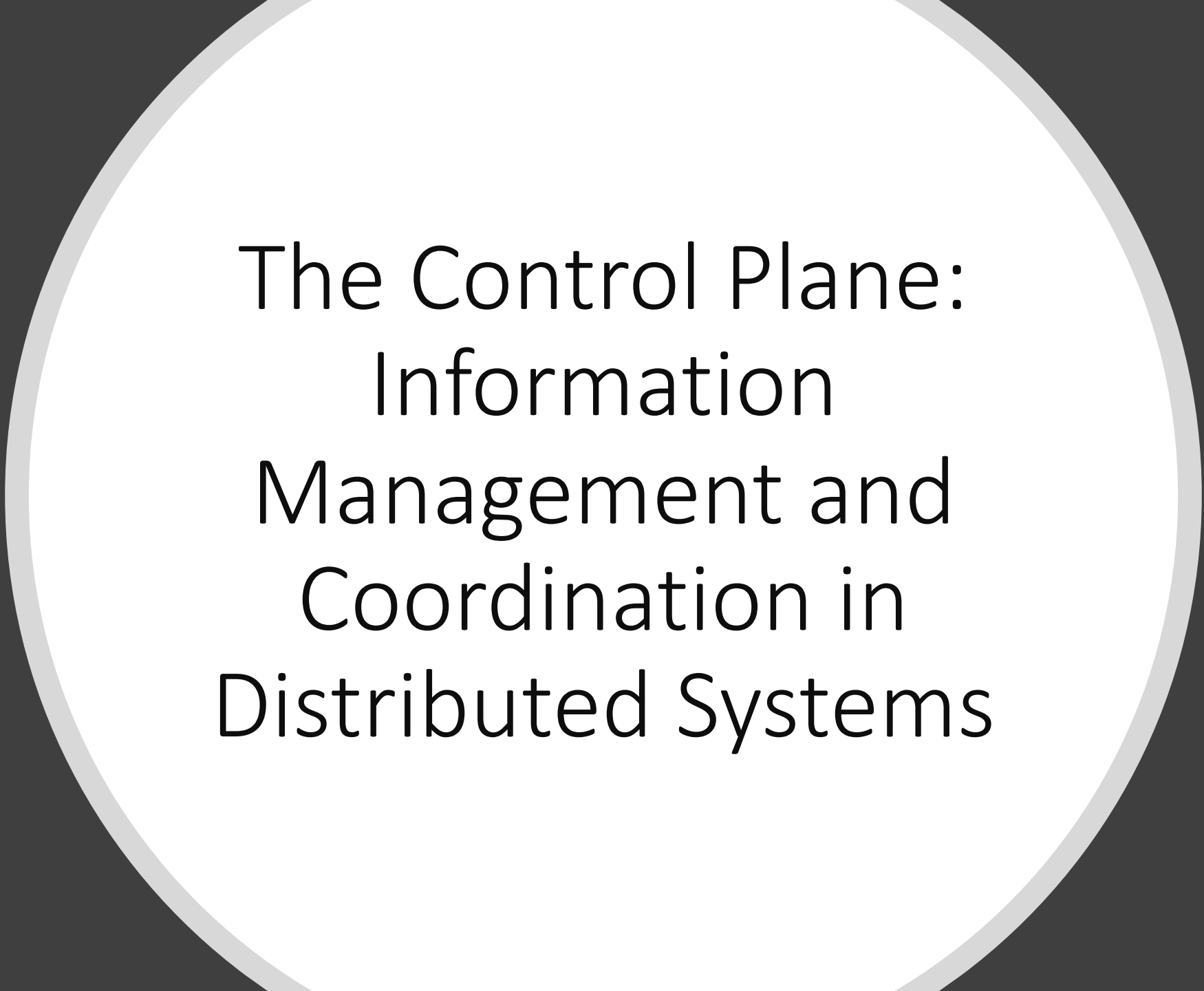
- <https://www.rabbitmq.com/clustering.html>
- Note use of Consul, RAFT, etc
- RabbitMQ always has good documentation, so I recommend reading



What Are Some  
Characteristics of Cloud-  
Native Applications?

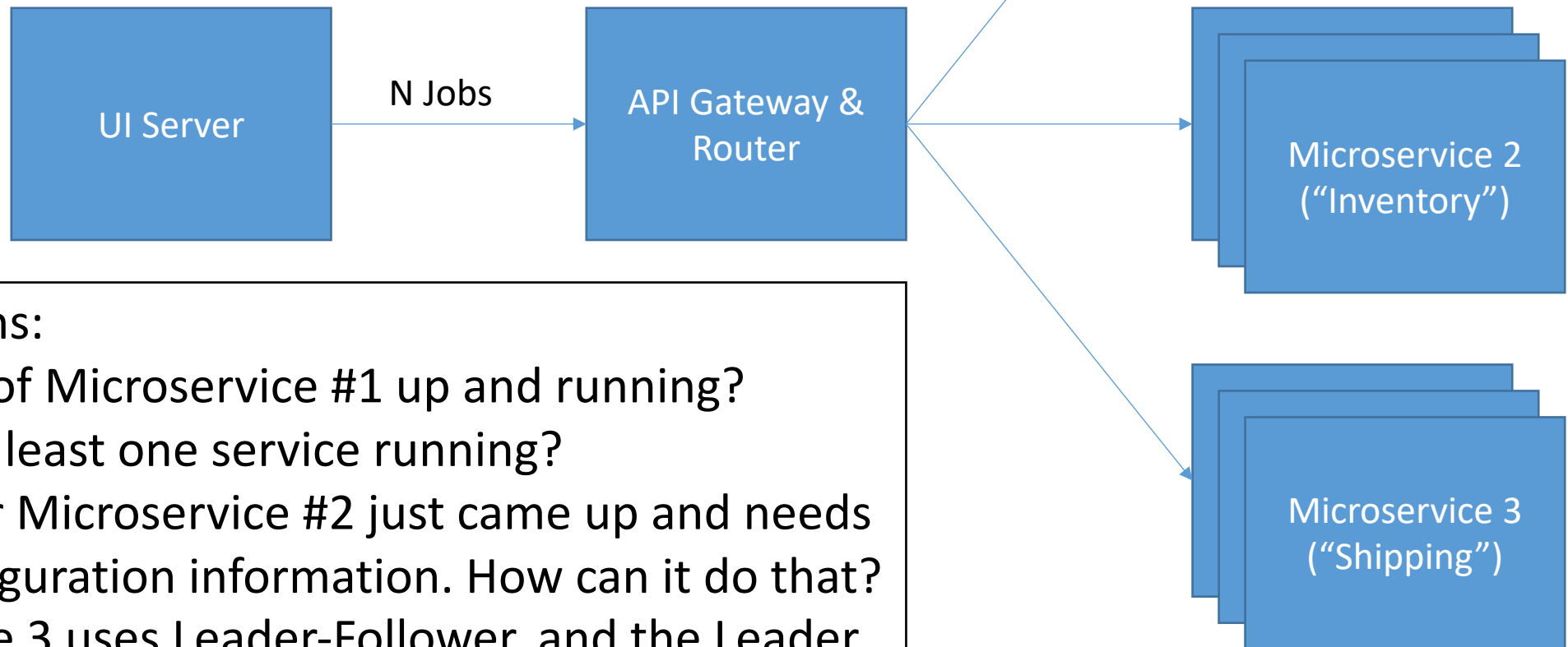
## Cloud-Native Applications: A Partial List

- They can grow and shrink dynamically
- They are fault tolerant: a component crash doesn't bring down the entire system
- You don't need to restart the whole system to add, update, or remove individual parts
- They operate continuously and evolve without downtime over a wide variety of operating conditions.



The Control Plane:  
Information  
Management and  
Coordination in  
Distributed Systems

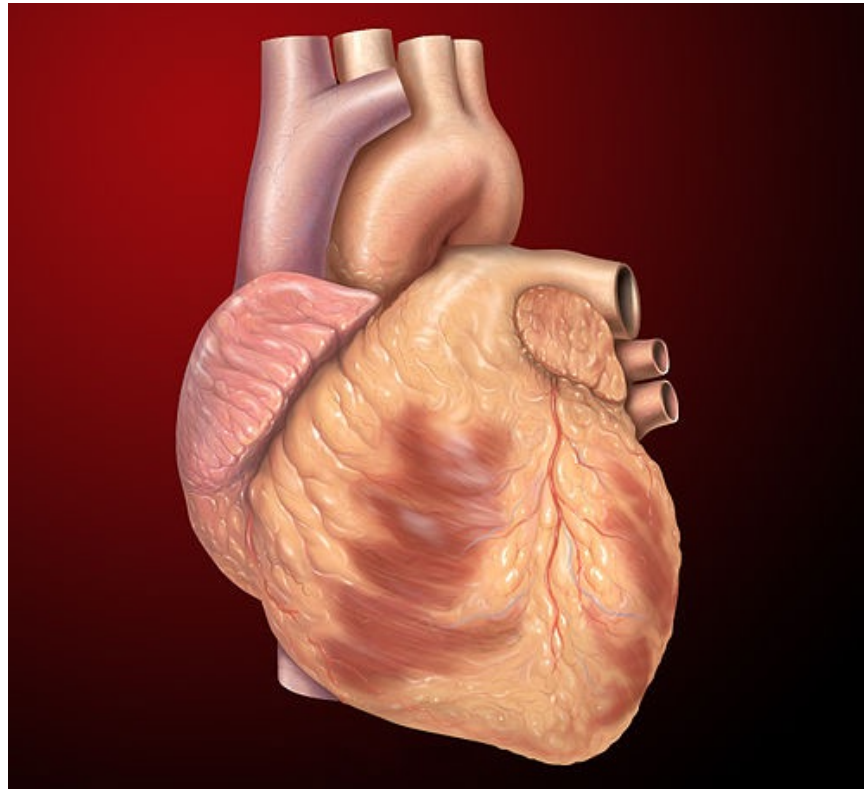
# Challenges with Microservices



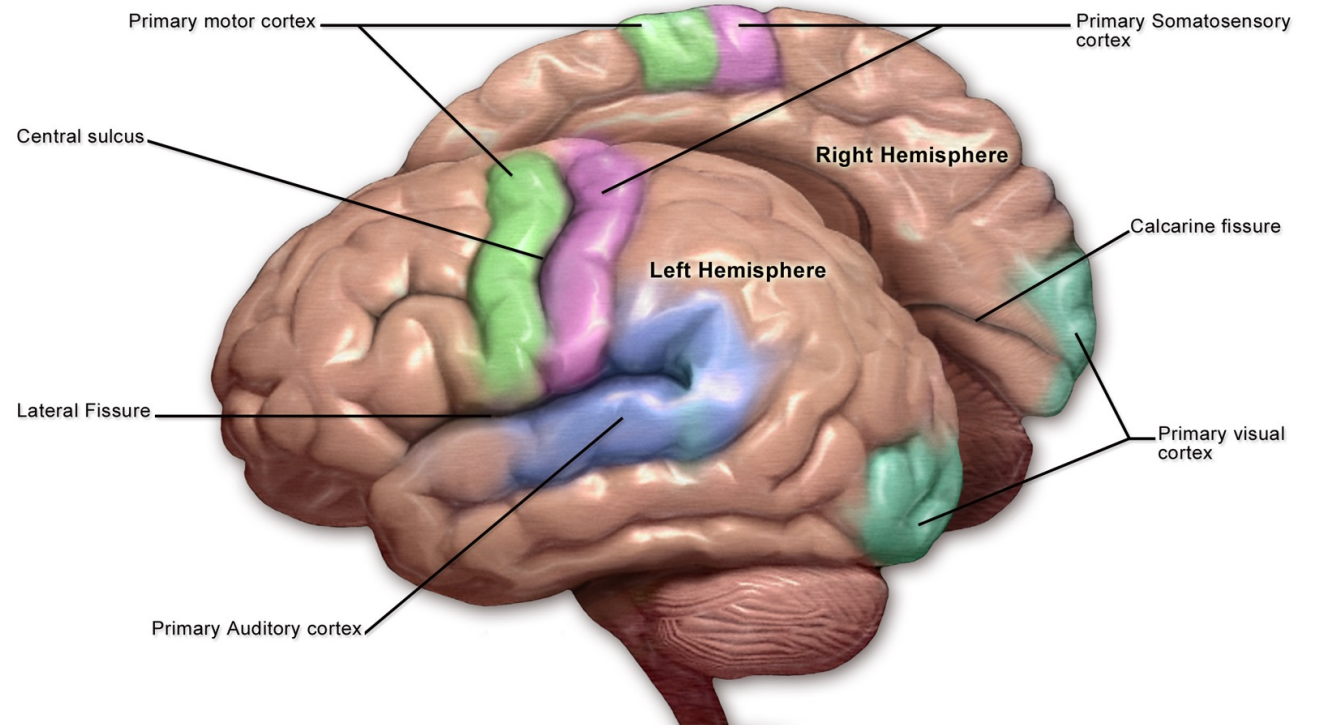
## Some questions:

- Is Replica 2 of Microservice #1 up and running?
- Do I have at least one service running?
- Replica 2 for Microservice #2 just came up and needs to find configuration information. How can it do that?
- Microservice 3 uses Leader-Follower, and the Leader just failed. What do I do?

# Messaging, Data Plane

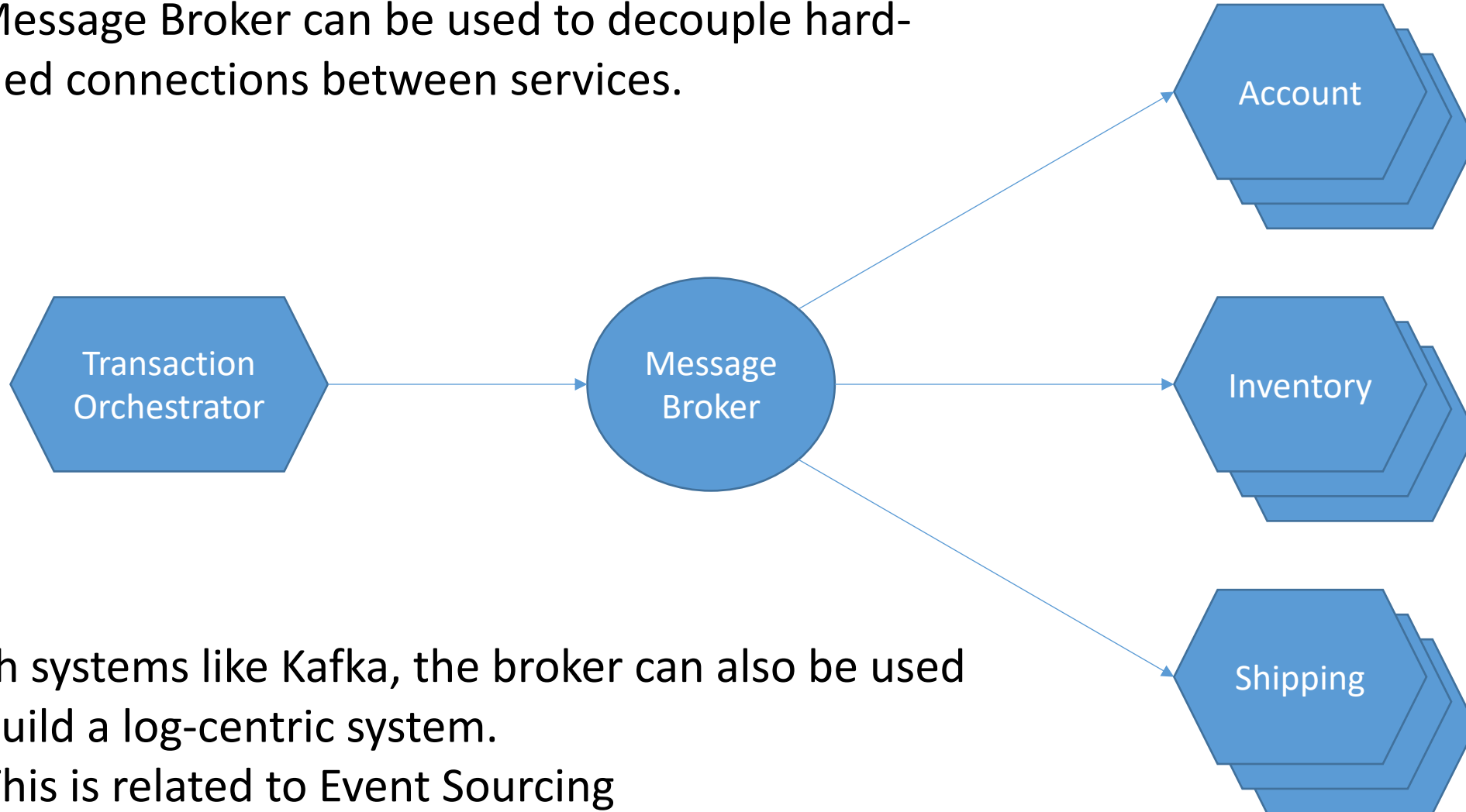


# Information, Control Plane



# Messaging and Log-Centric Approaches

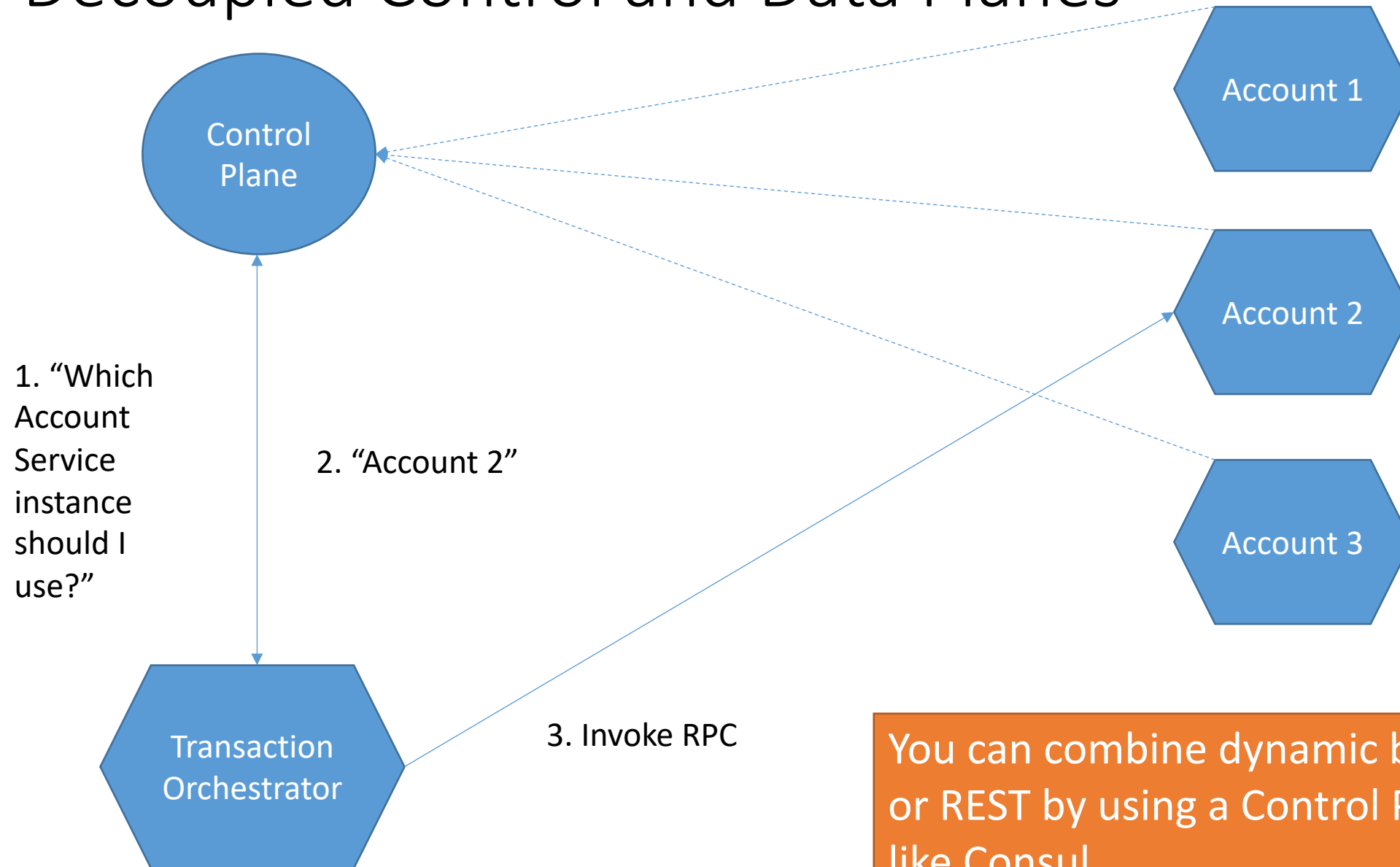
A Message Broker can be used to decouple hard-coded connections between services.



With systems like Kafka, the broker can also be used to build a log-centric system.

- This is related to Event Sourcing

# Decoupled Control and Data Planes



You can combine dynamic binding and RPC or REST by using a Control Plane system like Consul.





Distributed State and Coordination  
Management: Zookeeper, ETCD, Consul

# What Does the Control Plane Do?



Keeps track of services through heartbeats



Stores configuration information for services



Organizes services into useful groupings or collections (“Account Service” group, for example)



Helps services discover each other



Stores (usually) small pieces of metadata about services (port, IP, for example)



Helps services perform higher level coordination

# How Do Control Planes Do This?



Control planes store data using hierarchical key-value stores: trees, like a file system



Clients can create and delete nodes in these trees.



Clients can put, get, update, and delete information stored in a tree/leaf node

# One More Clever Thing: Notifications



Clients can request to get notified when changes occur in the Control Plane's data



A client can listen for the creation or removal of a node in a certain part of the tree



A client can listen for the writing or deletion of content within a node

# Control Plane Implementation



Control Planes must be very fault-tolerant



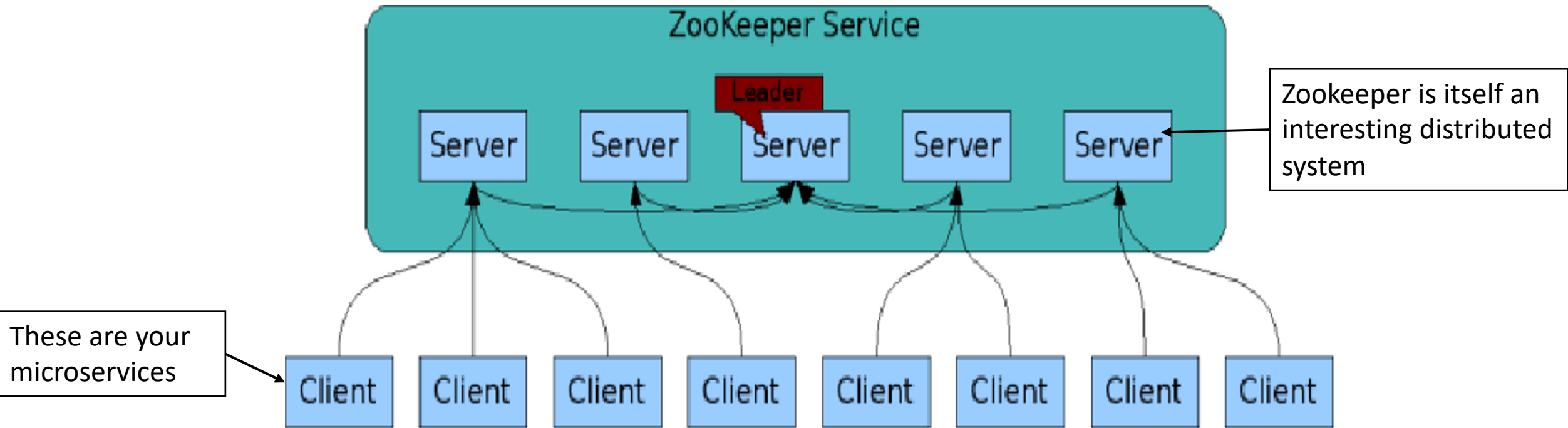
They replicate the tree structures and data across multiple servers



Control Planes are better suited for READ-heavy rather than WRITE-heavy applications



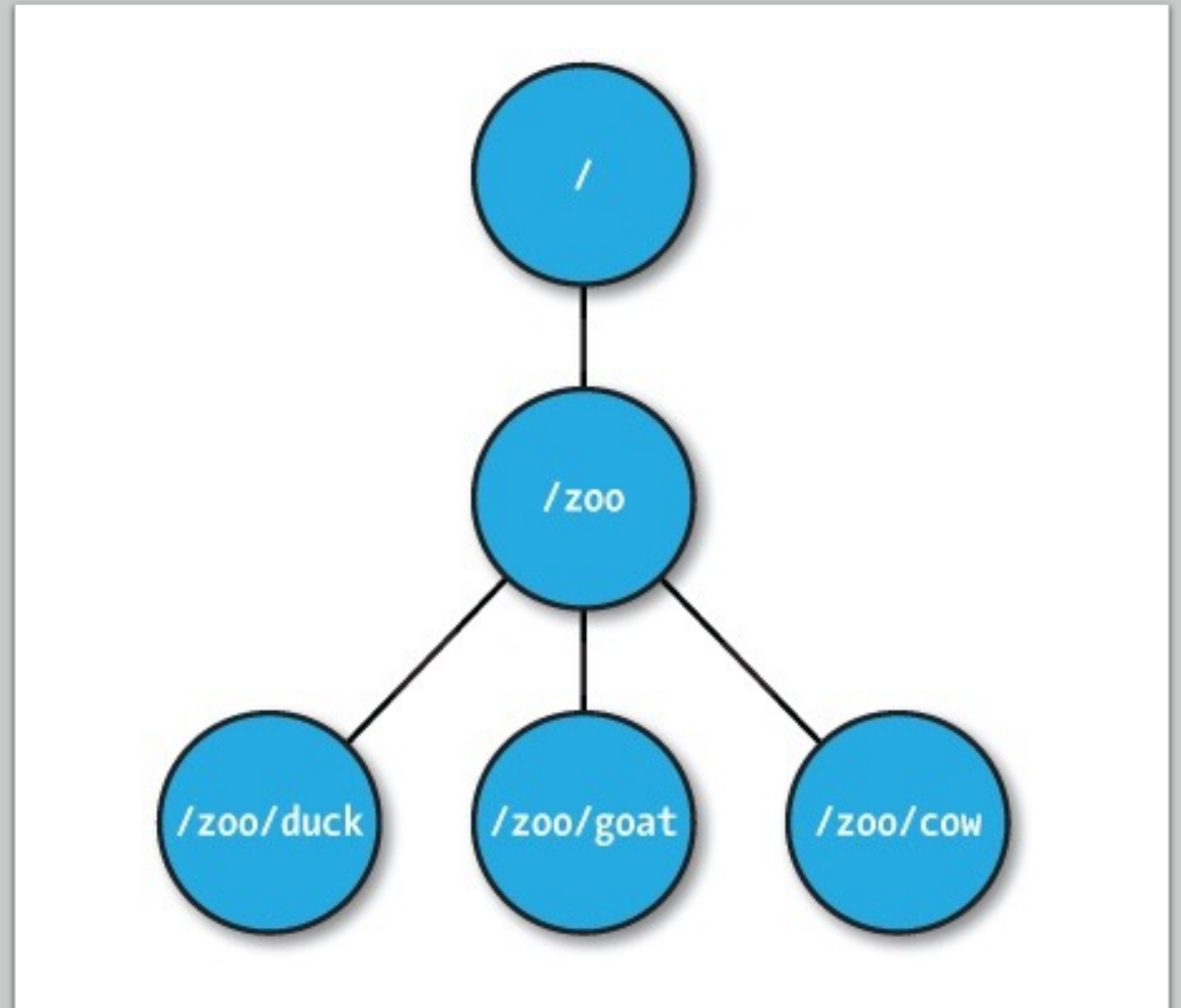
Zookeeper as an  
Example Control Plane  
Technology



- The ZooKeeper Service is replicated over a set of machines
- A leader is elected on service startup
- Clients can READ from any Zookeeper server.
- WRITES go through the leader & need majority consensus.

# Zookeeper Trees Consist of ZNodes

- Znodes maintain data with version numbers and timestamps.
- Version numbers increases with changes
- Data in a node are read and written in their entirety





# Example Structure

- ▼ test-cluster
  - PROPERTYSTORE
  - ▼ STATEMODELDEFS
    - STORAGE\_DEFAULT\_SM\_SCHEMATA
    - LeaderStandby
    - MasterSlave
    - OnlineOffline
  - ▼ INSTANCES
    - ▼ localhost\_8901
      - ▼ CURRENTSTATES
        - ▼ 139ff8ba6820064
          - test-db
          - ERRORS
          - STATUSUPDATES
          - MESSAGES
          - HEALTHREPORT
      - ▶ localhost\_8902
    - ▼ CONFIGS
      - ▼ RESOURCE
        - test-db
      - ▼ CLUSTER
        - test-cluster
      - ▼ PARTICIPANT
        - localhost\_8901
        - localhost\_8902

# Zookeeper, Briefly

- Zookeeper Clients (that is, your microservices) can create, read, update, and delete nodes on Zookeeper trees
- Clients can put small pieces of useful data into the nodes and get data out.
- Even the existence/non-existence of nodes can be useful information

You could build a small DNS or an LDAP server with Zookeeper

# ZNode Types

## Regular

- Clients create and delete explicitly
- Persistent

## Ephemeral

- Like regular znodes associated with sessions
- Deleted when session expires
- Clients can renew sessions

# Optional Node Property: Sequencing

- Both Regular and Ephemeral Nodes can be **Sequential**
- The ZNode name includes a universal, monotonically increasing counter
- You can use this to create unique node names

# Zookeeper API: Working with Trees

- **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
- **getChildren(path, watch)**: Returns the set of names of the children of a ZNode
  - More about *watch* in a minute

# Zookeeper API: Working with Node Data

- **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 named in the path if the version number is the current version of the ZNode

# Zookeeper API: Beyond CRUD

- **exists(path, watch):** Returns true if the ZNode with path name path exists and returns false otherwise.
  - If *watch* is true, the client will be notified if the ZNode is created.
- **sync(path):** Waits for all updates pending at the start of the operation to propagate to the server that the client is connected to.

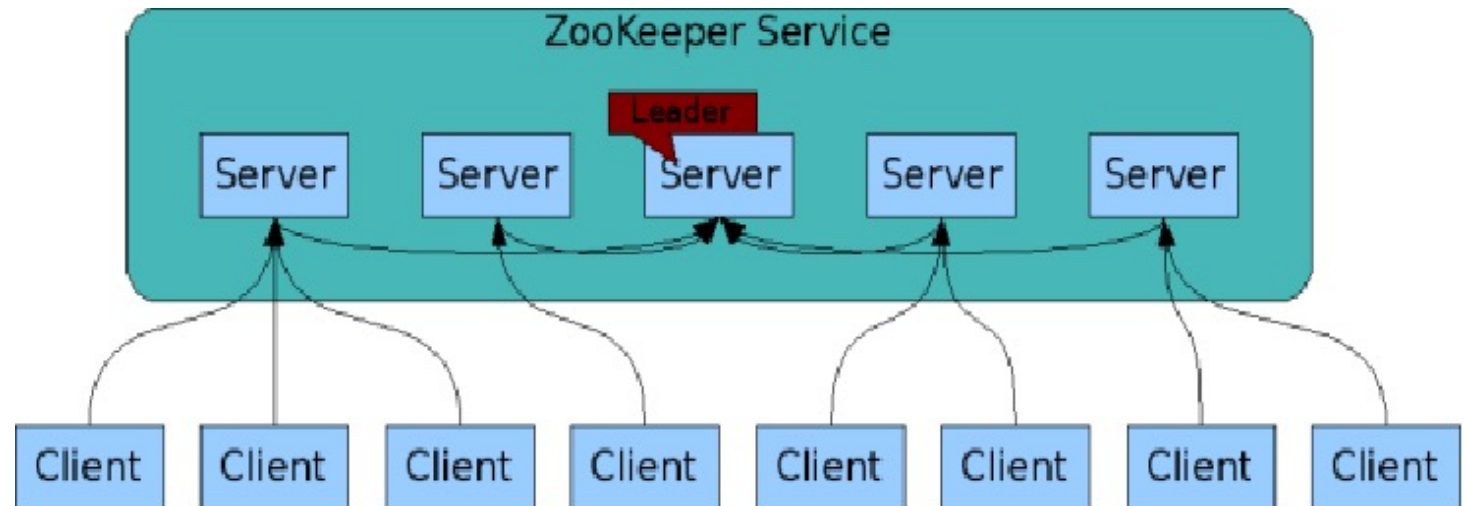
# Let's Look at Implementing Basic Control Plane Operations





# Terminology Reminder

- **Client:** this is one of are your microservices
- **Server:** this is an internal Zookeeper server



# Client Discovery by Other Clients

- Have each new instance create an ephemeral node when it joins
  - */root/services/newService*
- The content of the node can contain useful information about the service
  - For example, IP address and port
- Querying the children of */root/services* will return a service list.
- You can group them.
  - */root/services/accounts/newService*

# System Health

- Builds on Service Discovery
- Clients register themselves as ephemeral nodes
- Ephemeral nodes get deleted if the client that creates them fails.
- Other clients can put *watches* on these nodes.
  - If the node is deleted, a notification is fired to the other watching clients

# Group Membership for Clients

- Groups are just child nodes  
*/root/services/dataStaging/members/newService*
- Path names are just conventions
- You may want to define a few bootstrapping paths in static configuration files.
- Need unique names? Create new nodes with sequential flag
  - */root/services/dataStaging/members/newService\_3*

# Runtime Server Configuration

- Assume services need runtime configuration that they read after they come up
  - For example, which RabbitMQ queues does the “Account” service read from or write to?
- Put this info into Zookeeper as a standard node
  - /root/services/accounts/config
- Services in /root/services/accounts then read from config node
- If config is empty or doesn't exist, clients set *watch* flags for it.
  - Receive notices if the config is created, gets populated, or changes

# Advanced Coordination Scenarios



## Rendezvous Problem

What happens if essential configuration information isn't available to a client when it starts?

For example, which member of a client group is the leader?

# Solving the Rendezvous Problem

- Use a node named (for example) **/root/accounts/rendezvous**
- All the members of the **/root/accounts/services** group watch for the creation of the rendezvous node
- Example: **exists(path=/root/accounts/rendezvous, watch=true)**
- When the rendezvous node is created and filled in (by the leader, for example), the watching clients are notified, and everyone reads the information in the rendezvous node.



# Resource Locking in Distributed Systems



Locks allow multiple client instances and client types to modify shared resources, like files or configuration information.



Locks can also be used to implement a simple leader management scheme

# Solving the Locks Problem

Assume a client wants to lock a system resource so that it can make changes to it

The client locks the resource by creating a sequential ephemeral node:  
`/root/accounts/resources/lock_1`

The locking client can delete the *lock\_1* node, or it can just let the node session expire

If the client with the lock fails, the lock will eventually get released

# Locks, Continued: Wait Your Turn

If a second client wants to modify a locked resource, what does the second client do?

First, Client #2 creates its own lock as a sequential, ephemeral node: `/root/dataStaging/resources/lock_2`

Next, put a watch on `/root/dataStaging/resources/lock_1`.

When `lock_1` is released, the client who created `lock_2` gets notified and now has the lock

Client 2 should renew the `lock_2` node's session until Client 1's lock is released

If Client 2 fails before Client 1 releases the lock, the lock passes to the owner of `lock_3`

# Work Queues

You can implement work queues by having clients create ephemeral nodes indicating they are available to do work.

When a client gets work, it deletes its node

After the client completes a job, it creates a new node

# Work Partitioning

- Imagine you have a large amount of data to process
- Partition the data into 100 parts
- Create 100 ZNodes corresponding to each data partition
  - Each ZNode includes instructions, such as the location, size, and offset of the data to be processed
- If a client wants to process partition #12, it creates an ephemeral child node on the Partition #12 node to lock it.
- When the processing is done, the client deletes the node for Partition #12.
- If the client fails, the lock elapses, and another client can take Partition #12

# Apache Curator



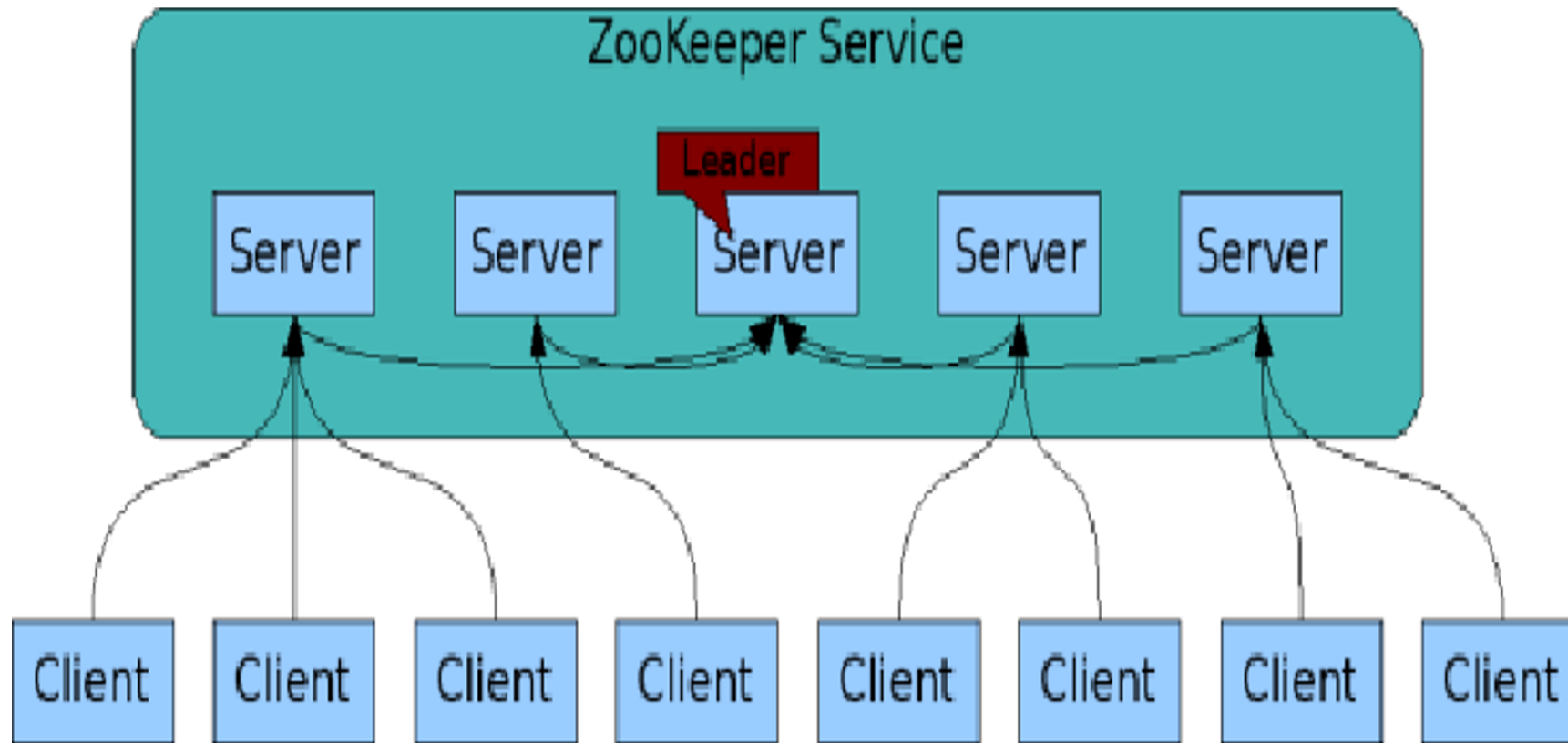
Curator is a Java library implemented on top of Zookeeper



It has many “out of the box” recipes (patterns) for many common distributed system problems

<https://curator.apache.org/curator-recipes/index.html>

# How Does Zookeeper Work Internally?



# Zookeeper Internals, Briefly

Zookeeper has one server that acts as the leader

The leader keeps the primary copy of the data (the node tree and its contents)

Client write requests are passed by a subordinate server to the leader

The leader provisionally applies the update to its data

It then notifies the other servers about the change

When a majority of the other servers confirm that they can apply the change, the leader commits the change



# Zookeeper Internals, Continued

---

This is called “quorum consensus” or “majority consensus”

---

A minority of subordinate servers may be behind the current state of the leader

---

Zookeeper allows clients to still read the obsolete information from lagging clients

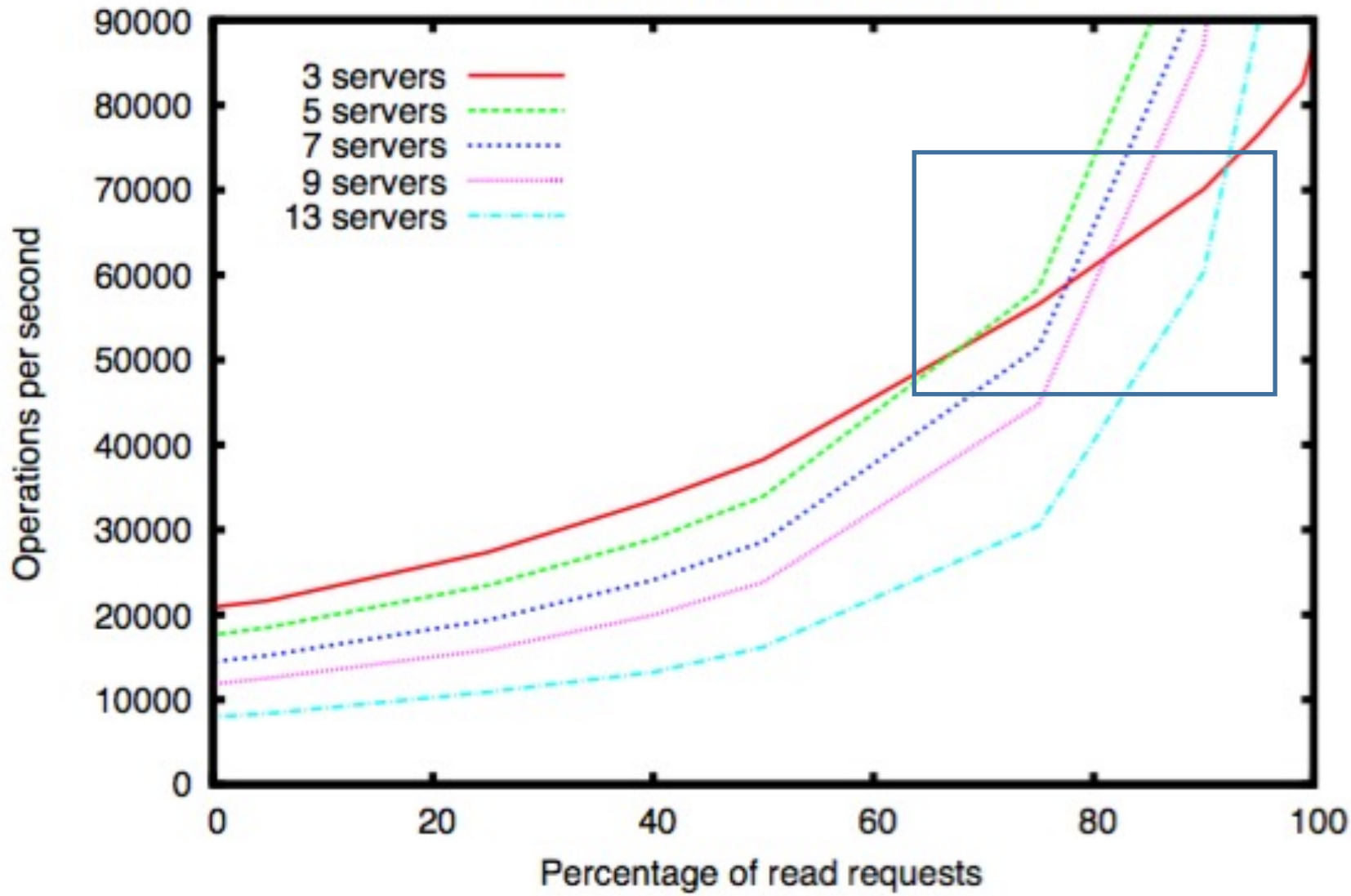
---

RAFT-based systems like Consul and ETCD are stricter: leader handles all reads

# Questions

- What are the limits of this version of the quorum consensus approach?
- What happens if the leader fails?

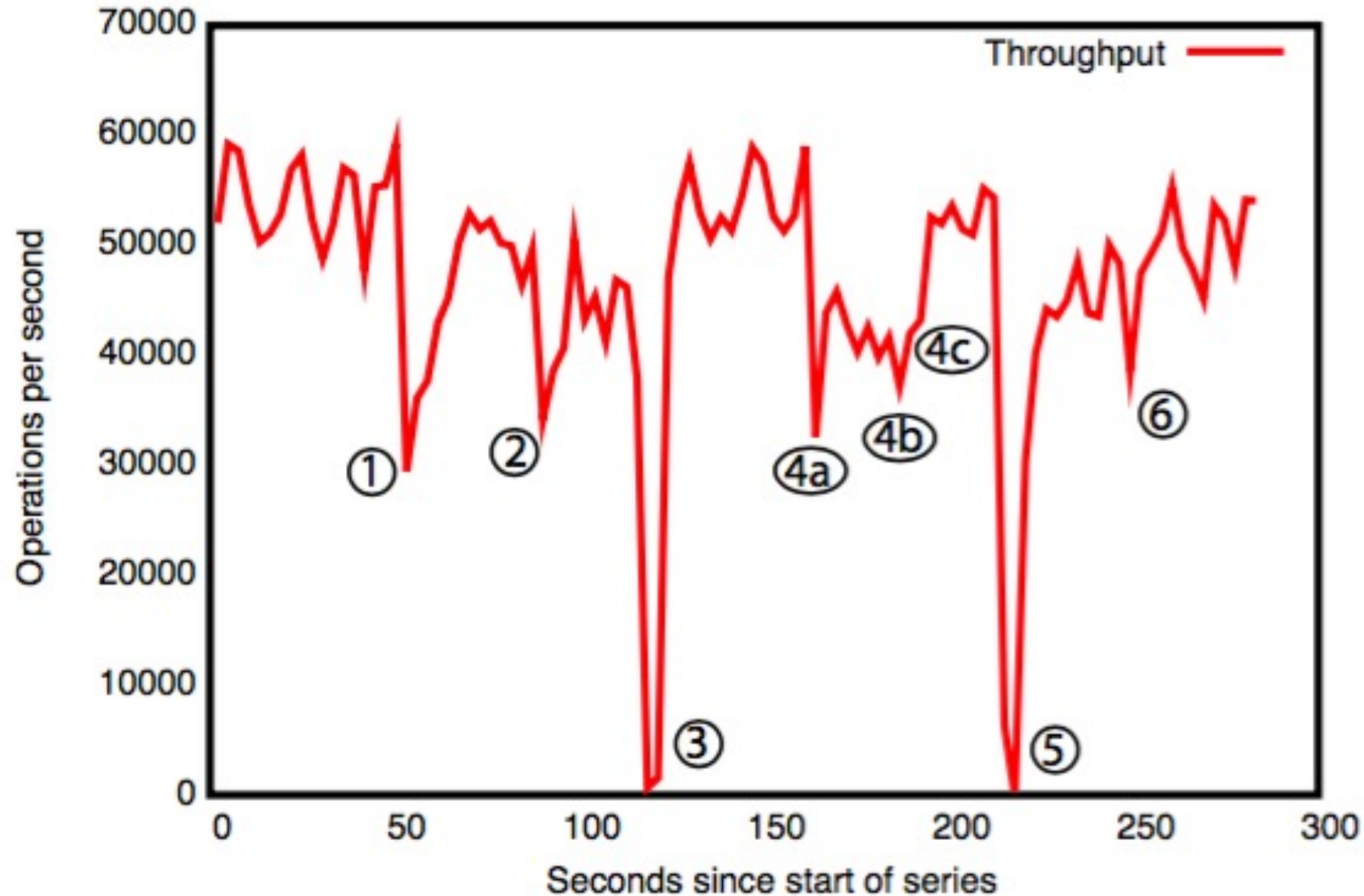
Throughput of saturated system



Zookeeper Clusters

- Small clusters handle writes more efficiently
- Large clusters handle larger read volumes better.

Time series with failures



- 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

A cluster of 5 zookeeper servers responds to manually injected failures.

## CAP Theorem:

A distributed data store can simultaneously provide at most two of the following three guarantees

***Consistency:*** Clients receive the most recent write for every request

***Availability:*** Clients receive a response without the guarantee that it contains the most recent write

***Partition tolerance:*** The system continues to operate despite an arbitrary number of messages being dropped or delayed by the network between nodes

# CAP and the Control Plane

- When you build a distributed system, you typically choose between **availability** and **consistency**
- RAFT-based systems like Consul are **consistent**: all READs go to the leader, but the leader can only handle so much traffic
- Zookeeper choose **availability** over consistency
  - Gives higher READ throughput if eventual consistency is OK
- It all depends on how consistent your distributed state needs to be

