



What Are Some
Characteristics of Cloud-
Native Applications?

A Partial List

- They scale
- 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.

CAP Theorem:

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


Consistency: Every read receives the most recent write or an error

Availability: Every request receives a (non-error) 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

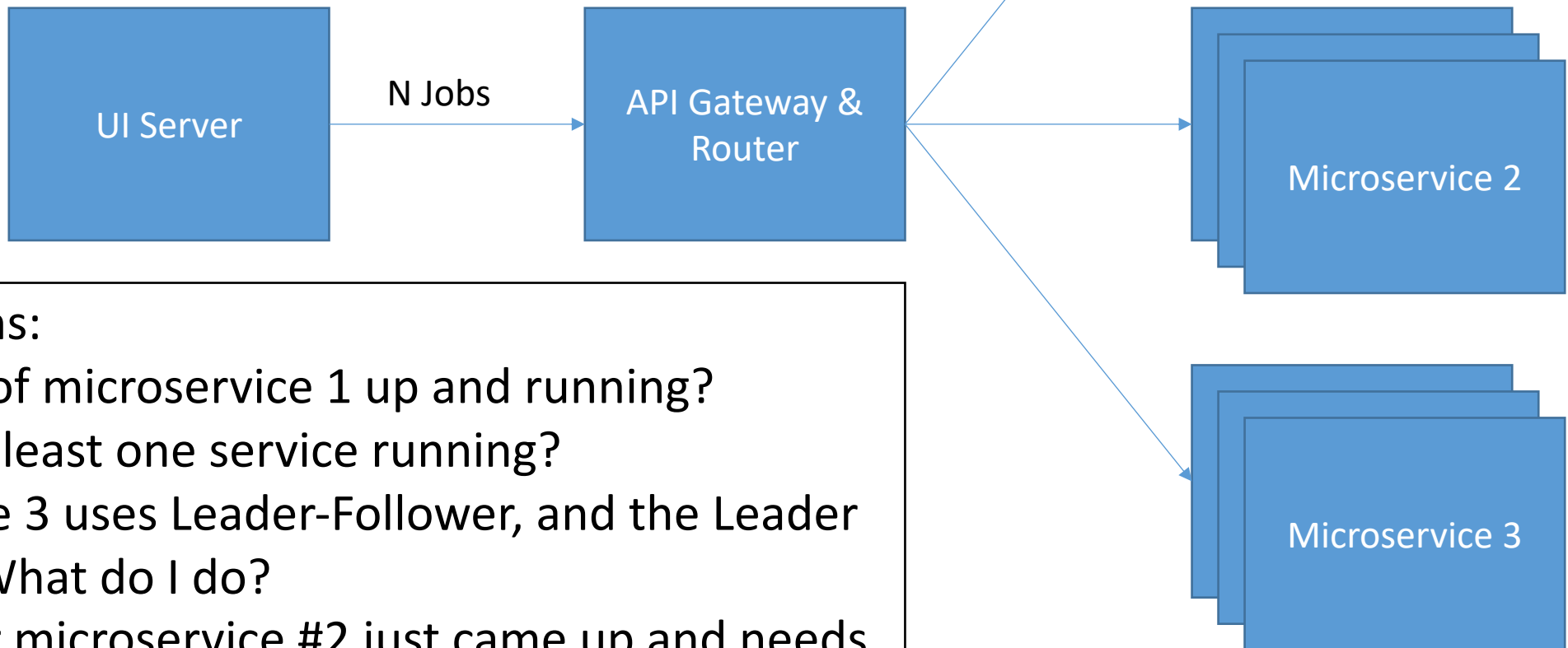
CAP and the Control Plane

- When you build a distributed system, you typically choose between availability and consistency
- RAFT is consistent: all READs go to the leader, but the leader can only handle so much traffic
- Other systems may loosen the consistency requirements to be more available
 - Ex: Higher READ throughput if eventual consistency is OK
 - Ex: Scaling across data centers
- It all depends on how consistent your distributed state needs to be



The Control Plane: Information Management and Coordination in Distributed Systems

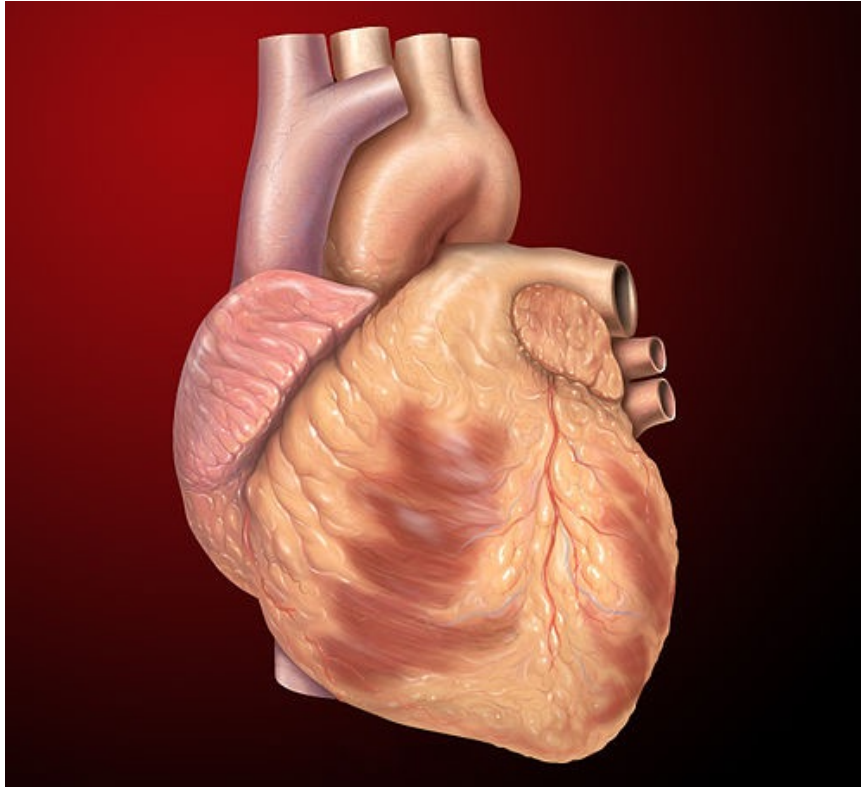
Challenges with Microservices



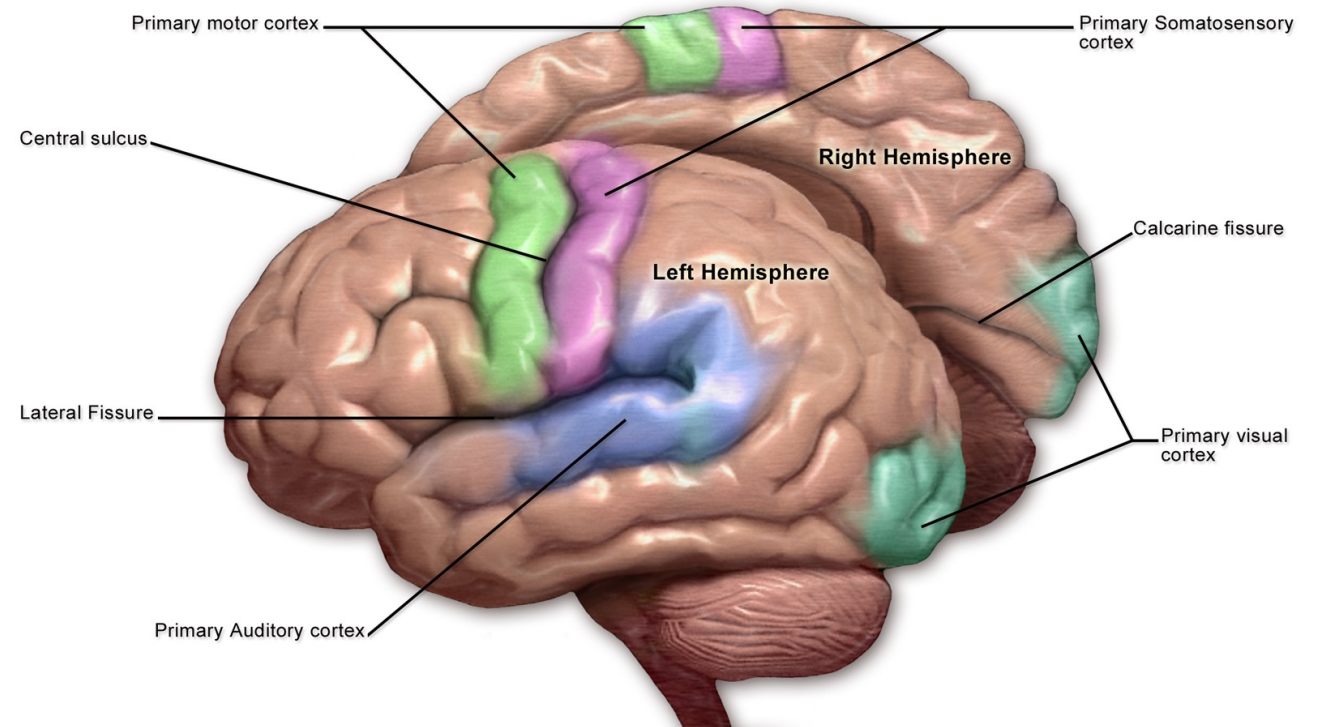
Some questions:

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

Messaging, Data Plane



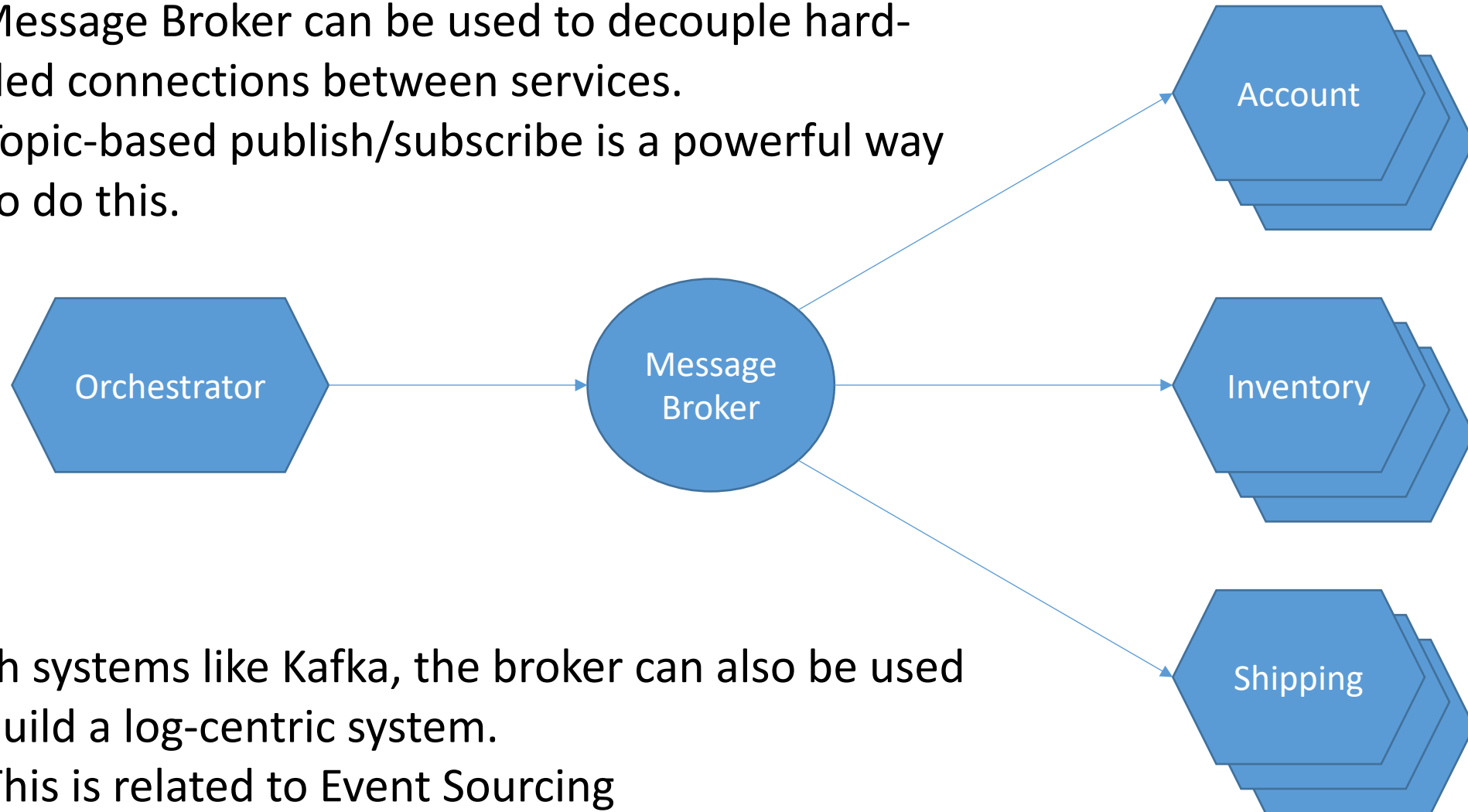
Information, Control Plane



Messaging and Log-Centric Approaches

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

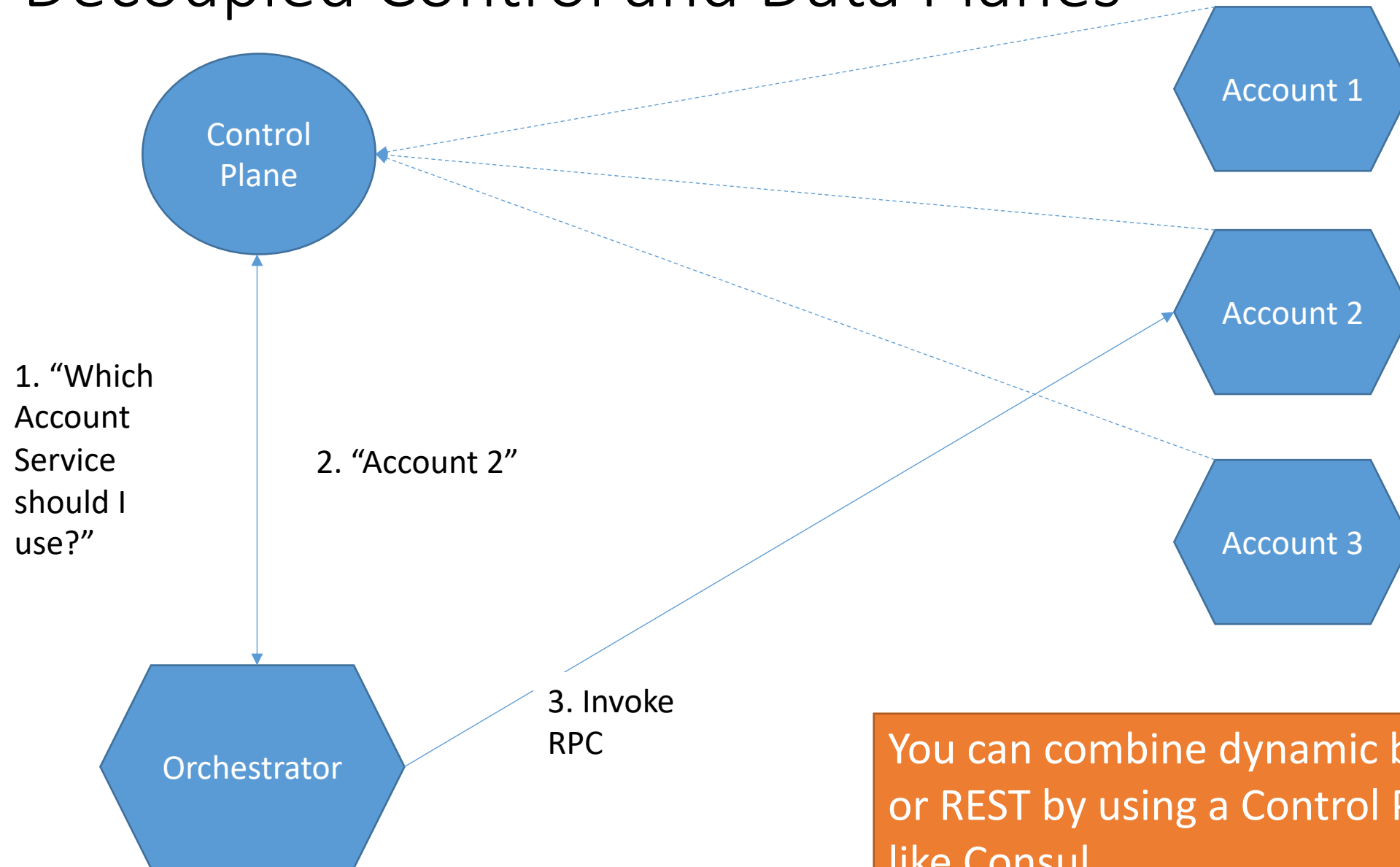
- Topic-based publish/subscribe is a powerful way to do this.



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 collections
("Account")



Helps services discover each other



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



Helps services perform higher level coordination

How Do Control Planes Do This?



Control planes store data using hierarchical key-value stores



Clients can create and delete nodes in trees.



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

One More Clever Thing: Notifications



Clients can request to get notified when changes occur



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



“State changes”

Control Plane Implementation



Control Planes must be very fault-tolerant



They use a protocol like RAFT to distribute the tree structures and the data stored in the trees across multiple servers

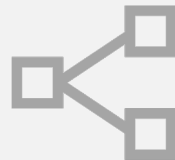


Consequently, Control Planes are better suited for READ-heavy rather than WRITE-heavy applications

What Does the Control Plane Not Do?



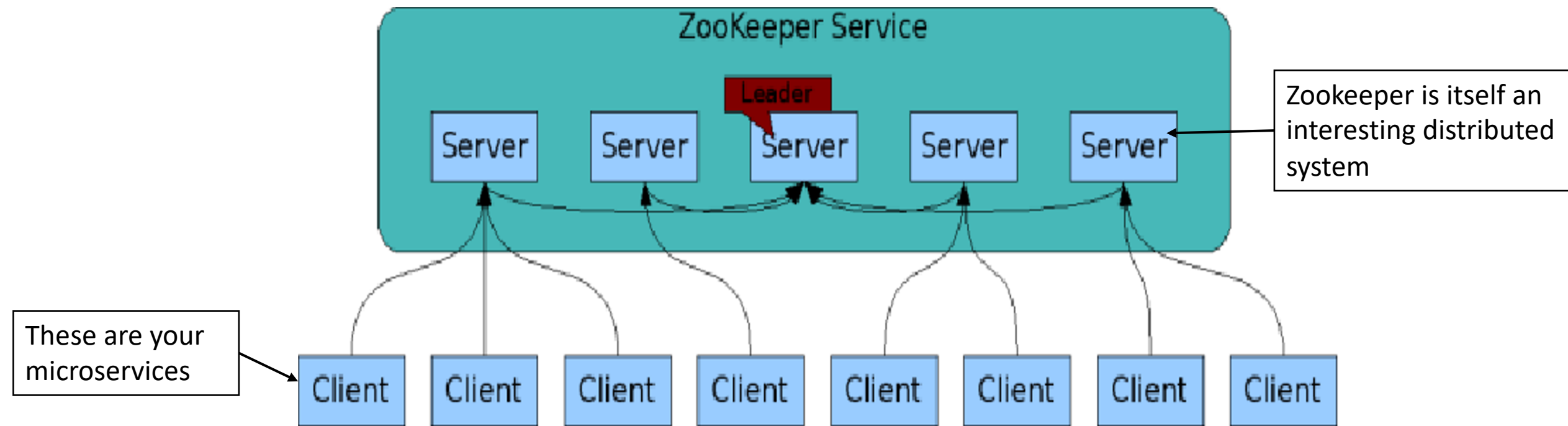
Control planes do not route messages between other services



Control planes do not expose their inner workings to clients



Zookeeper as an Example Control Plane Technology



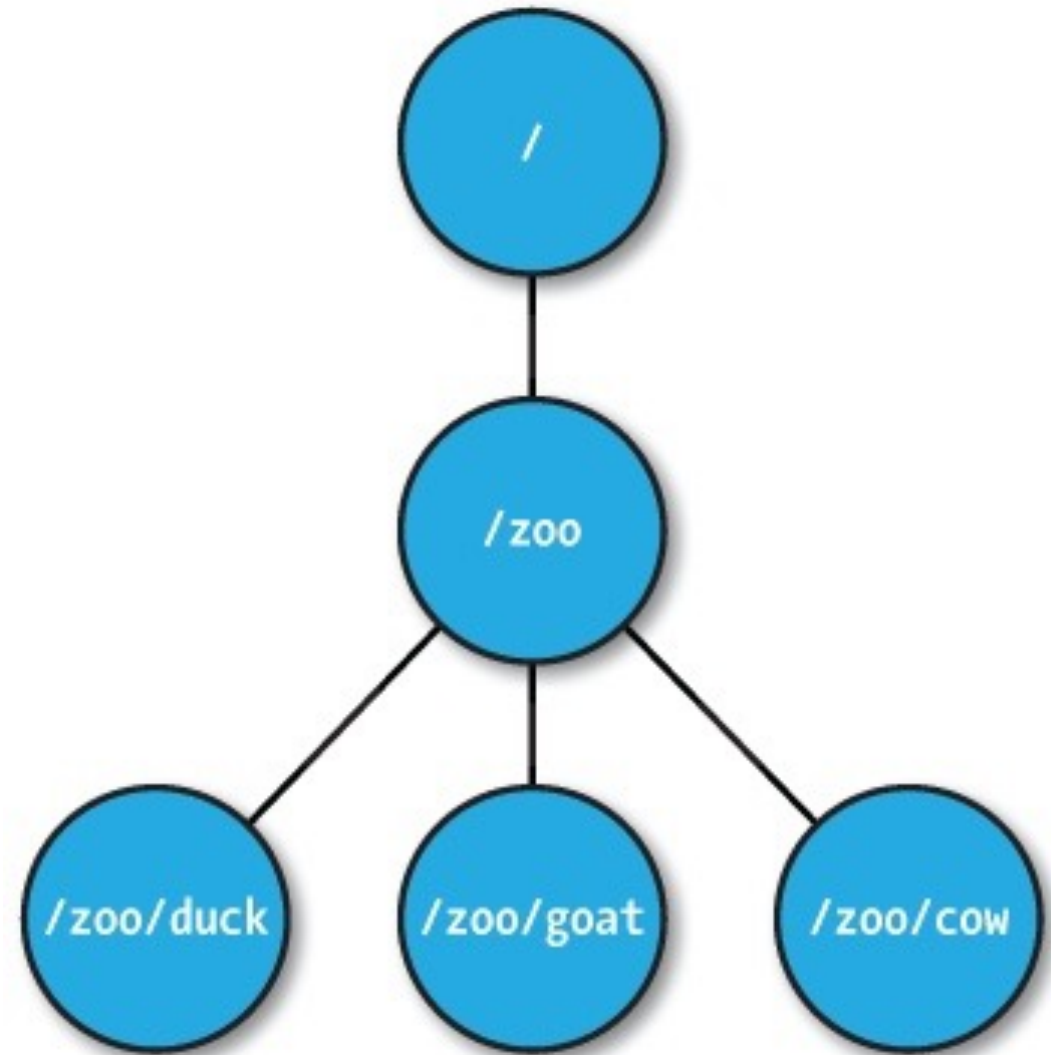
- The ZooKeeper Service is replicated over a set of machines
- A leader is elected on service startup
- WRITES go through the leader & need majority consensus.
- Clients can READ from any Zookeeper server.

Zookeeper, Briefly

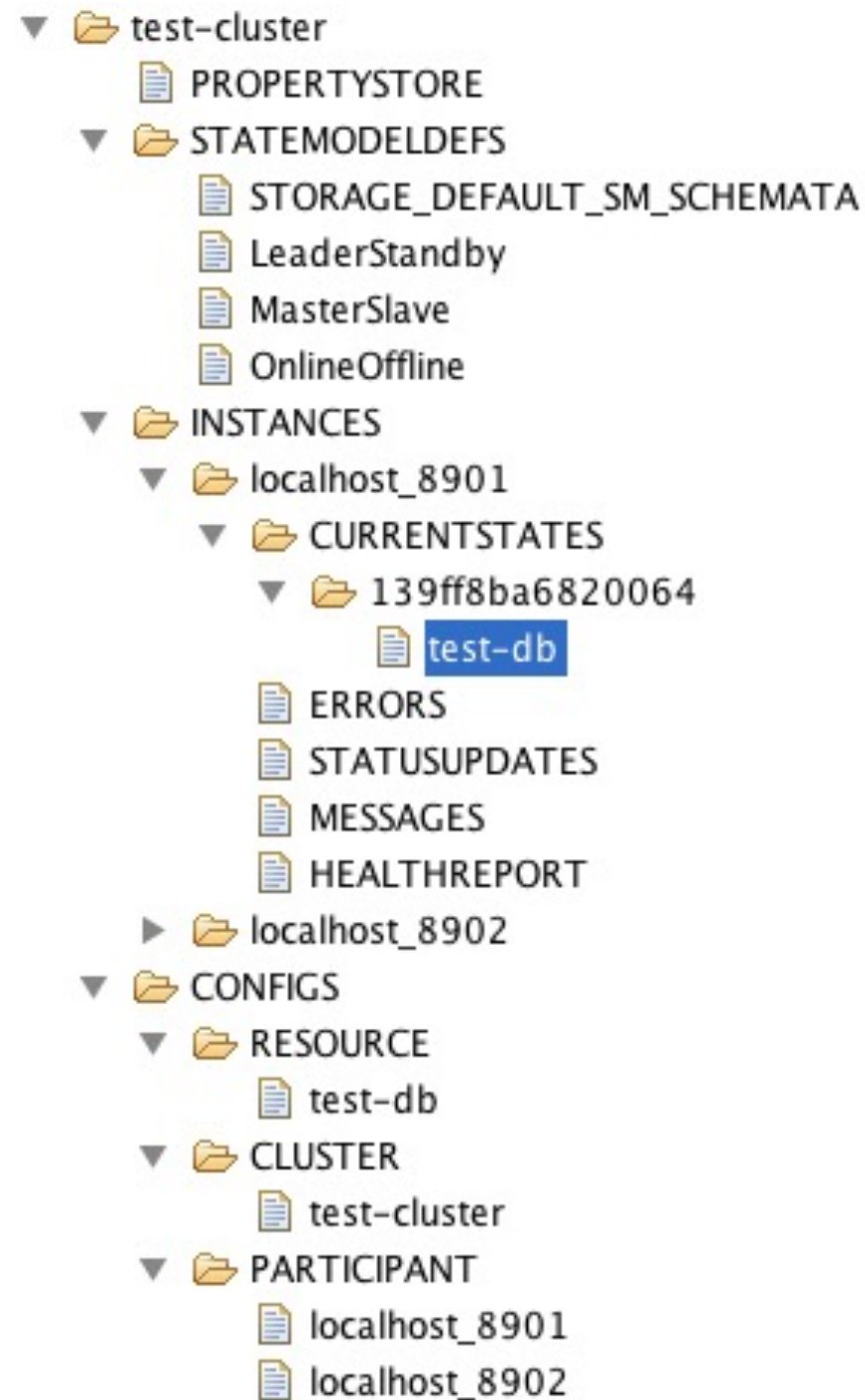
- Zookeeper Clients (that is, your microservices) can create and discover nodes on Zookeeper trees
- Clients can put data into the nodes and get data out.
 - Each node also has built-in metadata like its version number.
- Even the existence/non-existence of nodes can be useful information
- You could build a small DNS or an LDAP server with Zookeeper.
- It can support several other interesting uses

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



ZNode Types

Regular

- Clients create and delete explicitly

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, 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

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 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.
 - Note the *watch* flag
- **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

Service Discovery

- 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
 - Its IP address
- Querying the children of */root/services* will get a service list.
- You should also group them.
 - */root/services/accounts/newService*

System Health

- Builds on Service Discovery
- Ephemeral nodes get deleted if the client that creates them fails.
 - A service can also delete these nodes by choice
- Other clients can put *watches* on these nodes.
 - If the node is deleted, a watch notification is fired

Group Membership for Services

- 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 read from config
- If config is empty or doesn't exist, services watch() for it.
 - Receive notices if the config is created or gets populated

Advanced Coordination

Control planes can be used to support more complicated operations

Before Zookeeper, some of these were done by dedicated systems like Google's Chubby lock management system

Zookeeper's innovation was to design a system that could be used to build many different systems

Others have subsequently done it better

Rendezvous Problem

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

For example, where is the leader of a service group?

Solving the Rendezvous Problem

- Use a node named `/root/dataStaging/rendezvous`
- All the members of `/root/dataStaging/services` watch for the creation of the rendezvous node
 - `exists(path=
/root/dataStaging/rendezvous,
watch=true)`
- When the node is created and filled in (by the leader, for example), the watchers are notified and everyone reads the information in the rendezvous node.

Resource Locking in Distributed Systems



Locks allow multiple service instances and service 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/dataStaging/resources/lock_1`

The locking client can delete the `lock_1` node or it can just let its session expire

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

Locks: Wait Your Turn

If you want to modify a resource but another process has locked it, what do you do?

First, create your own lock as a sequential, ephemeral node: `/root/dataStaging/resources/lock_2`

Next, put a watch on `lock_1`.

When `lock_1` is released, Client 2 gets the lock

Client 2 should renew the node's session until the 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`

Simple Leader Election

Have all would-be leaders create ephemeral nodes

Each node contains metadata about the potential leader's qualifications, as in the RAFT protocol

The Control Plane acts as an independent source of information and helps coordinate the election

This simplifies the election process

The Control Plane itself of course still needs the full RAFT protocol

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 a node corresponding to each partition
 - The node 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 the node for partition 12.
- 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 it.

Power versus Simplicity

Control planes are powerful because they can be used to implement so many different distributed system patterns

But this can also make them complicated

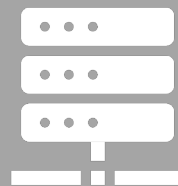
And what if you don't implement a pattern very well?

As we saw, leader election can be complicated

Apache Curator



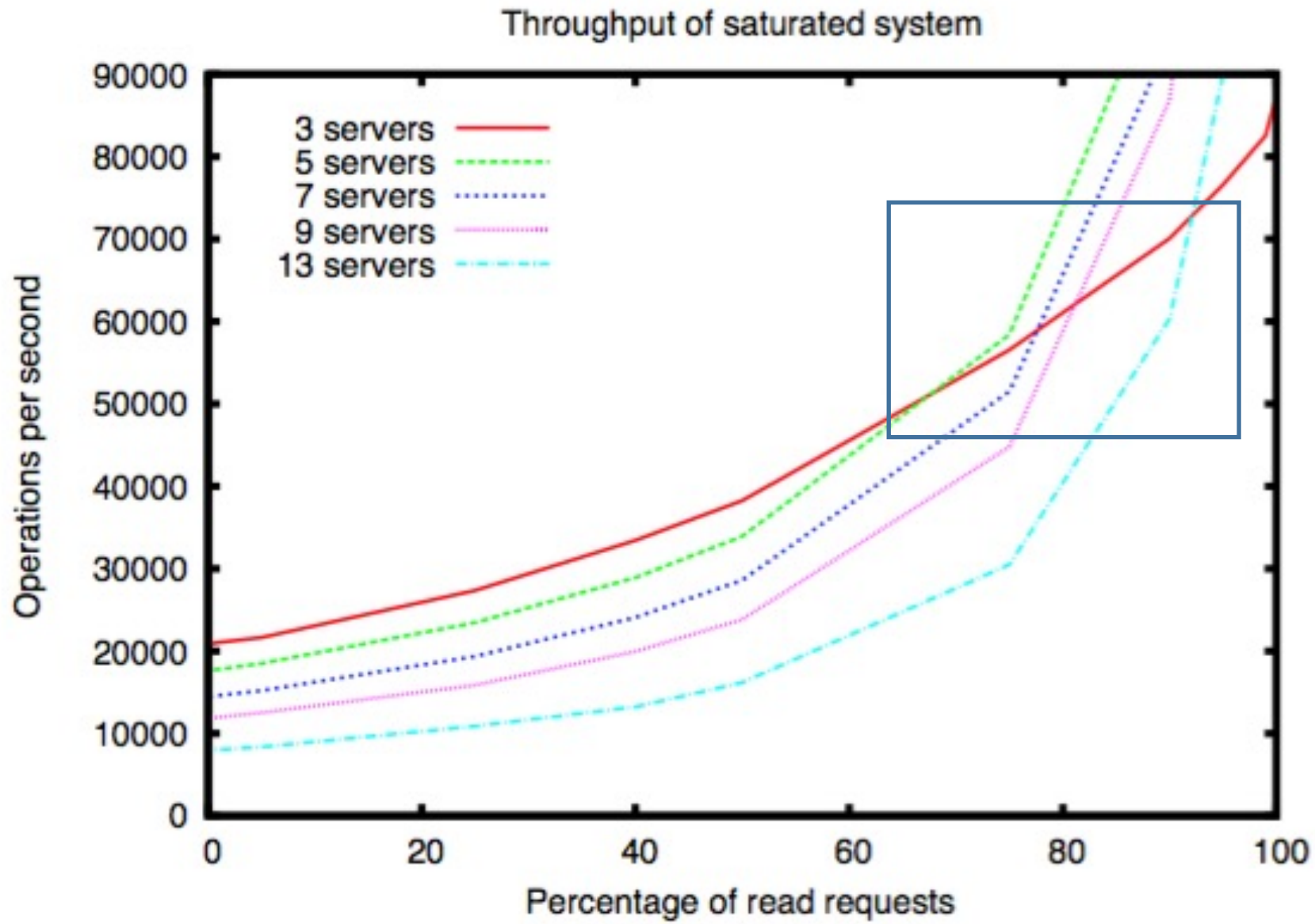
Curator is a Java library implemented on top of Zookeeper



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

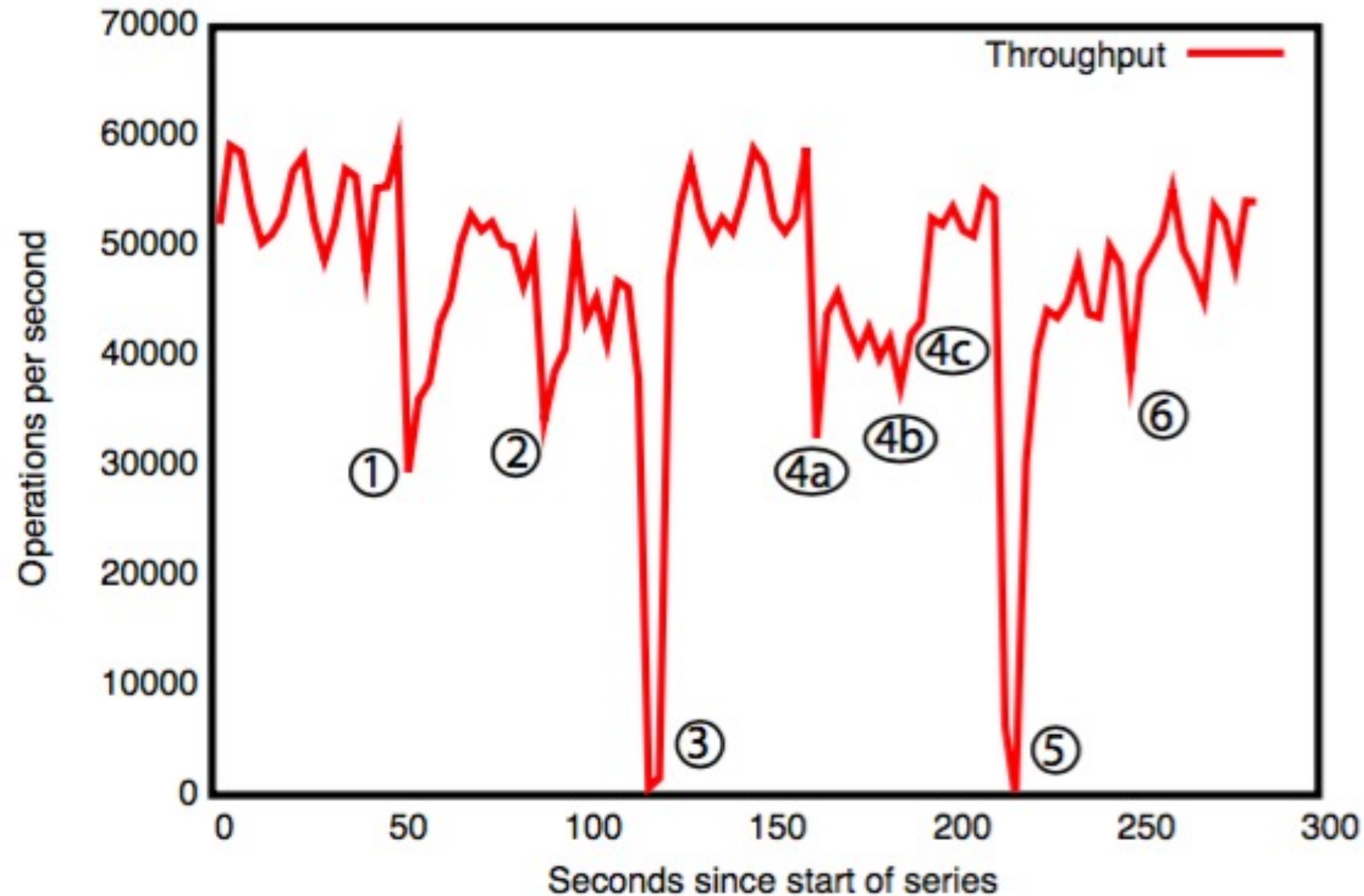
“I am an experimentalist. I do not hope.
I measure.”

J. S. Hangst, <https://www.nature.com/articles/d41586-021-00884-5>



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

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
6. Recovery of leader (?)

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

Zookeeper: Wait-Free Data Objects

Key design choice: wait-free data objects

- Locks are not a Zookeeper primitive
- You can use Zookeeper to build lock-based systems

Resembles distributed file systems

- Smaller data

FIFO client ordering of messages

- Asynchronous messaging
- Assumes you can order messages globally

Basic idea: idempotent state changes

- Components can figure out the state by looking at the change log
- Operations incompatible with state throw exceptions

A good approach when systems can tolerate inconsistent data

- DNS for example
- But not E-Commerce, which needs stronger guarantees

Coordination Examples in Distributed Systems

Configuration

Basic systems just need lists of operational parameters for the system processes: IP addresses of other members
Sophisticated systems have dynamic configuration parameters.

Group Membership, Leader Election

- Processes need to know which other processes are alive
- Which processes are definitive sources for data (leaders)?
- What happens if the leader is faulty?

Locks

- Implement mutually exclusive access to critical resources.

Zookeeper Caches and Watches

Zookeeper clients cache data

- Reads go to the cache

Watches: notify listening clients when cache has changed

- Watches don't convey the content of the change.
- Only work once; clients decide on the action

Why?

- Networks aren't uniform or reliable (fallacy)
- Centralized, top-down management doesn't scale

Suitable for read-dominated systems

- If you can tolerate inconsistencies