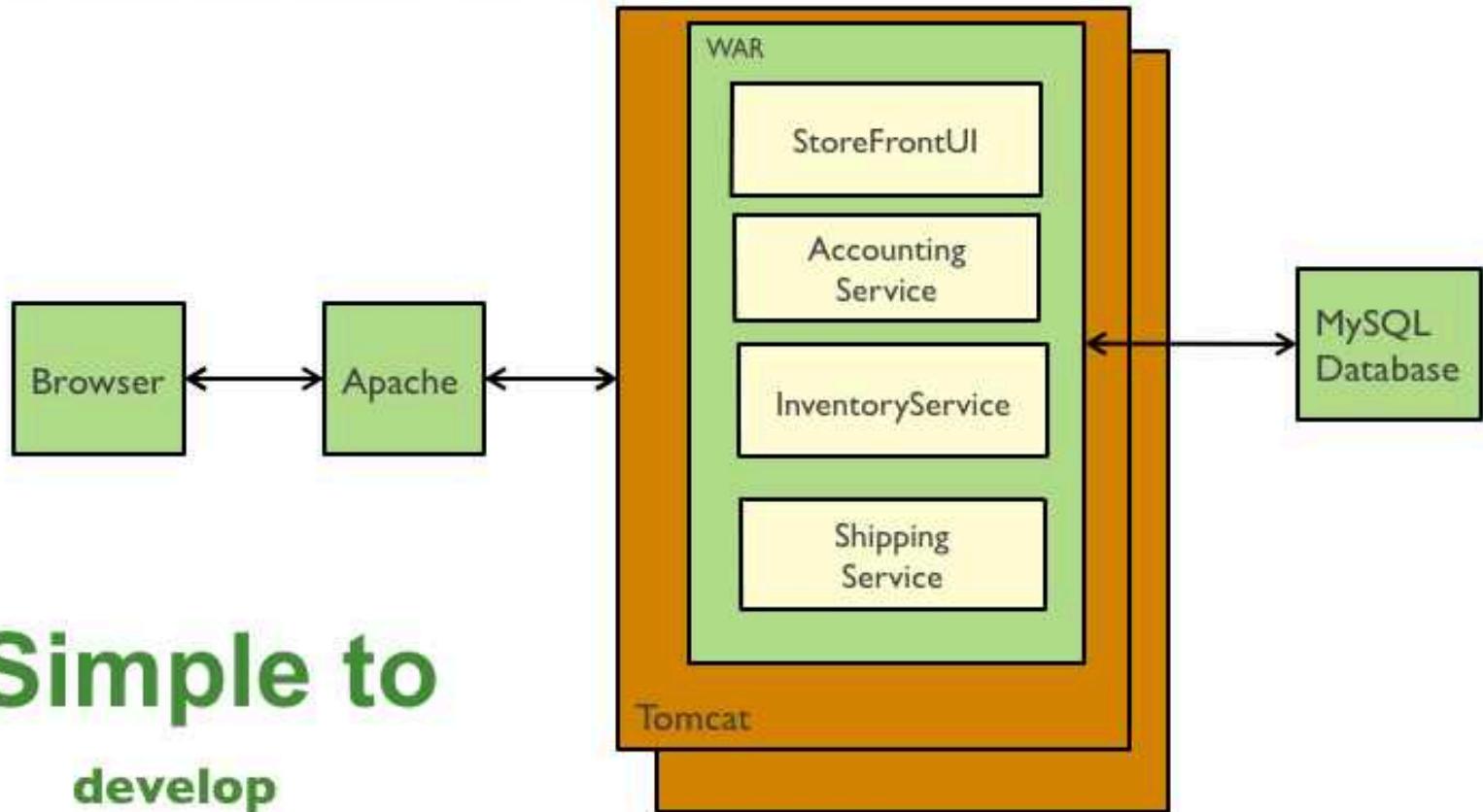


Microservices Review

Microservices use distributed systems concepts to build scalable, “cloud native” applications

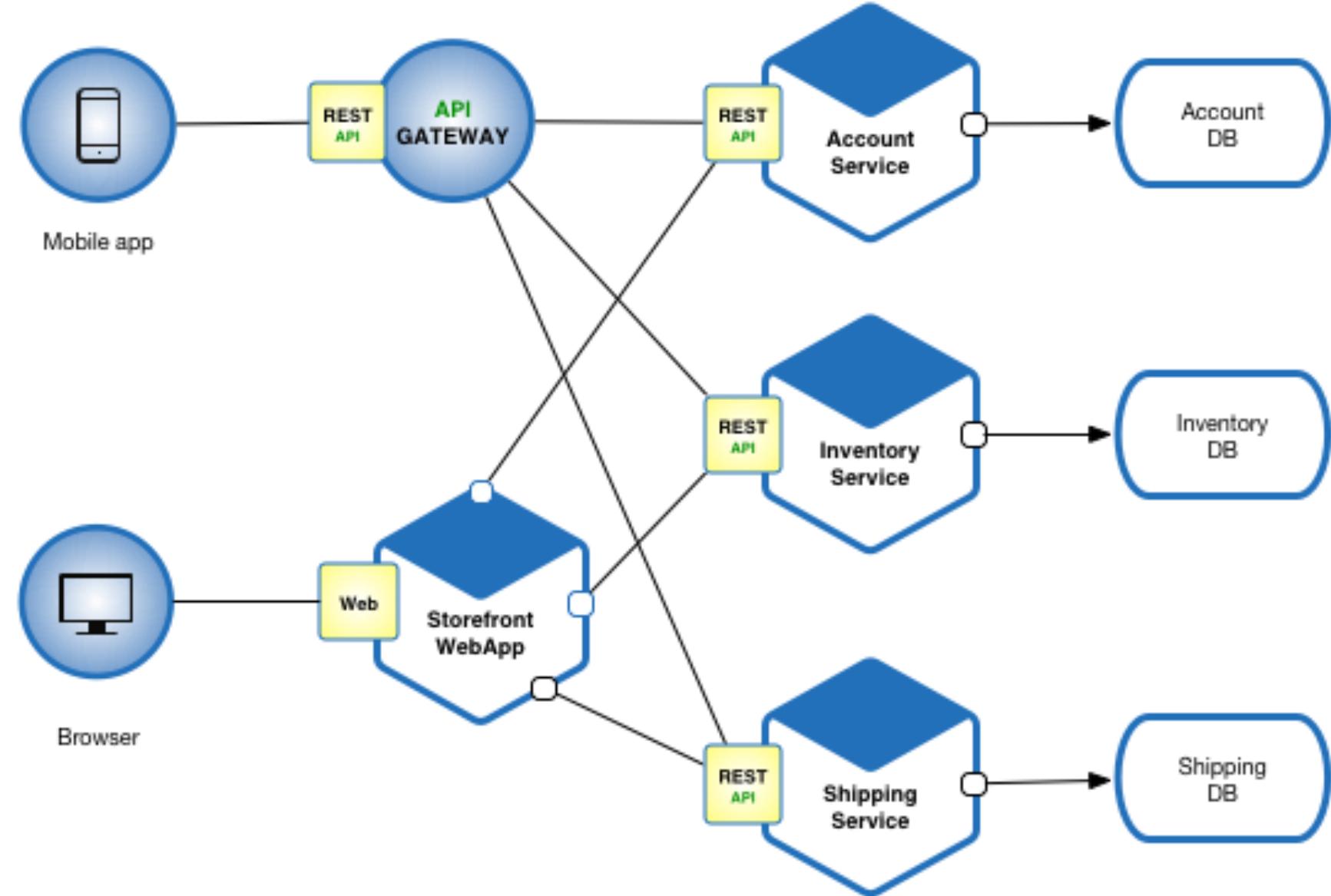
Monolithic Web Application

Traditional web application architecture



Microservice-Style Web Application: First Cut

What are the
shortcomings of this
simple REST approach?



Limitations of the Simple REST Approach



Each service is a single point of failure:
no redundancy.



All the connections and logic are hard-coded



Request-response may not be the right
message exchange pattern

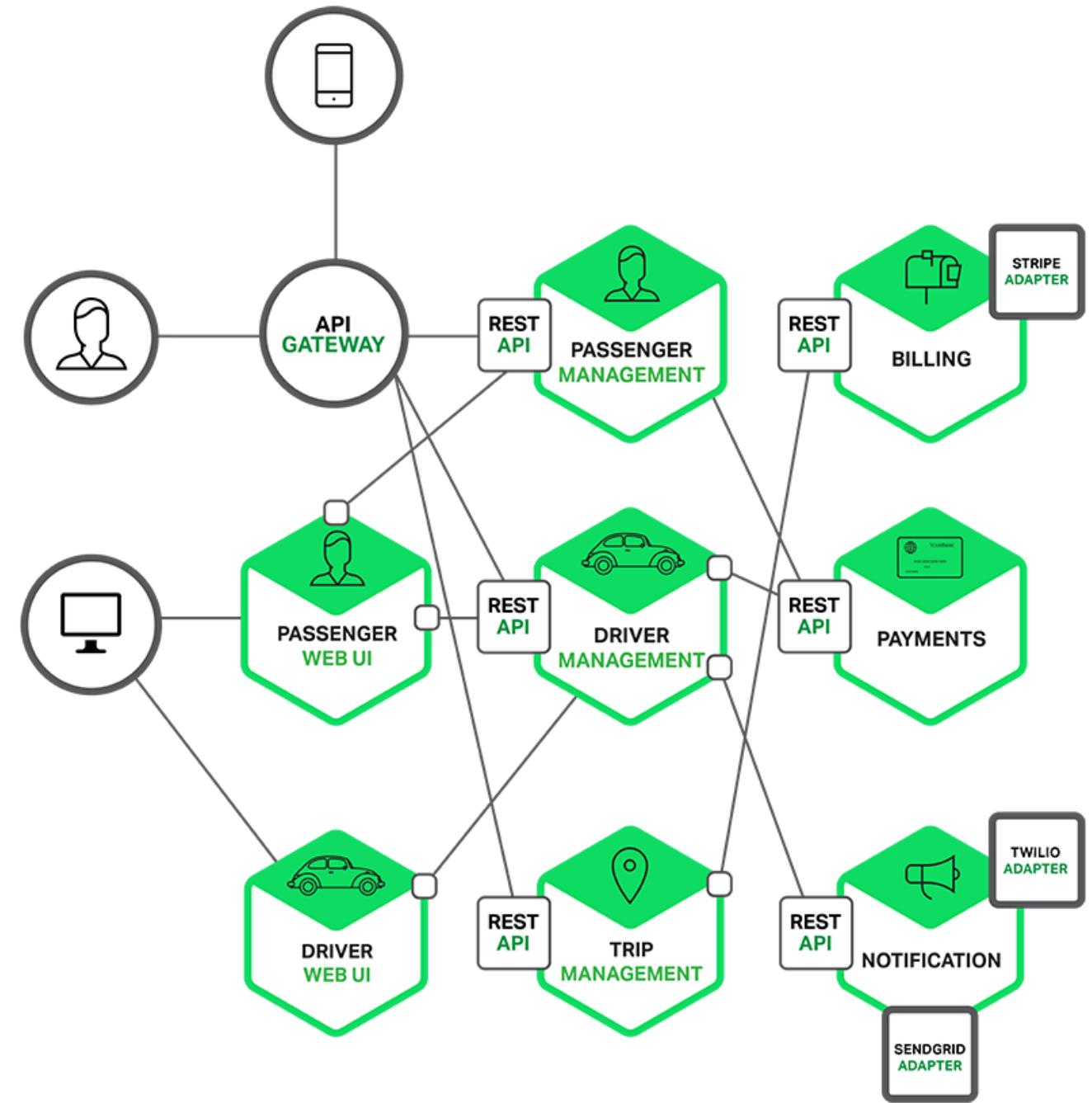


What do we do if we need to replay
messages?

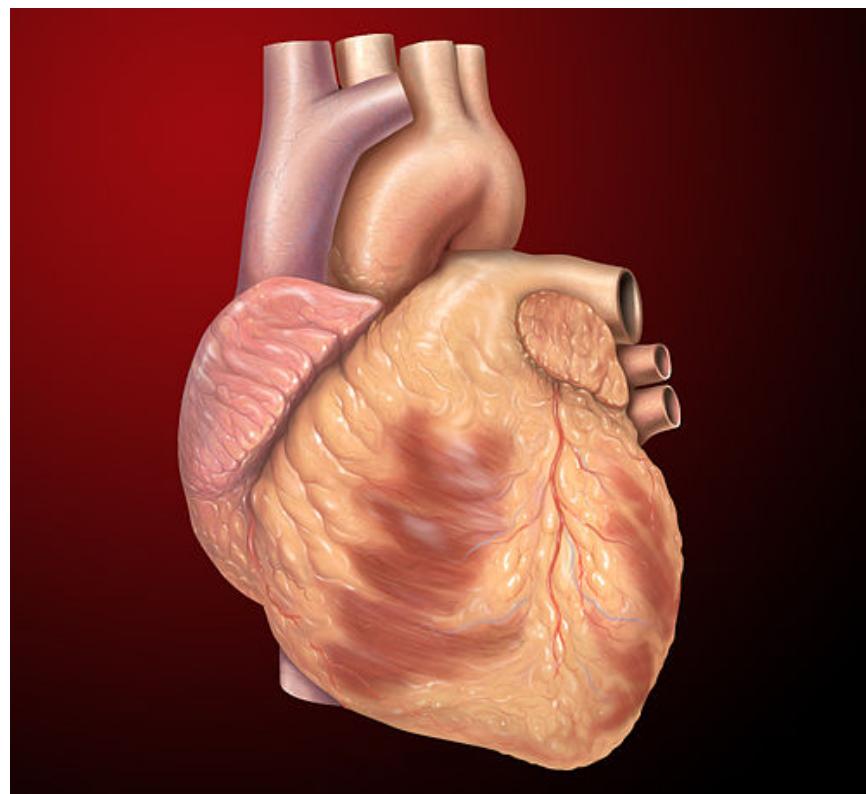


We only have a data plane, not a
control plane

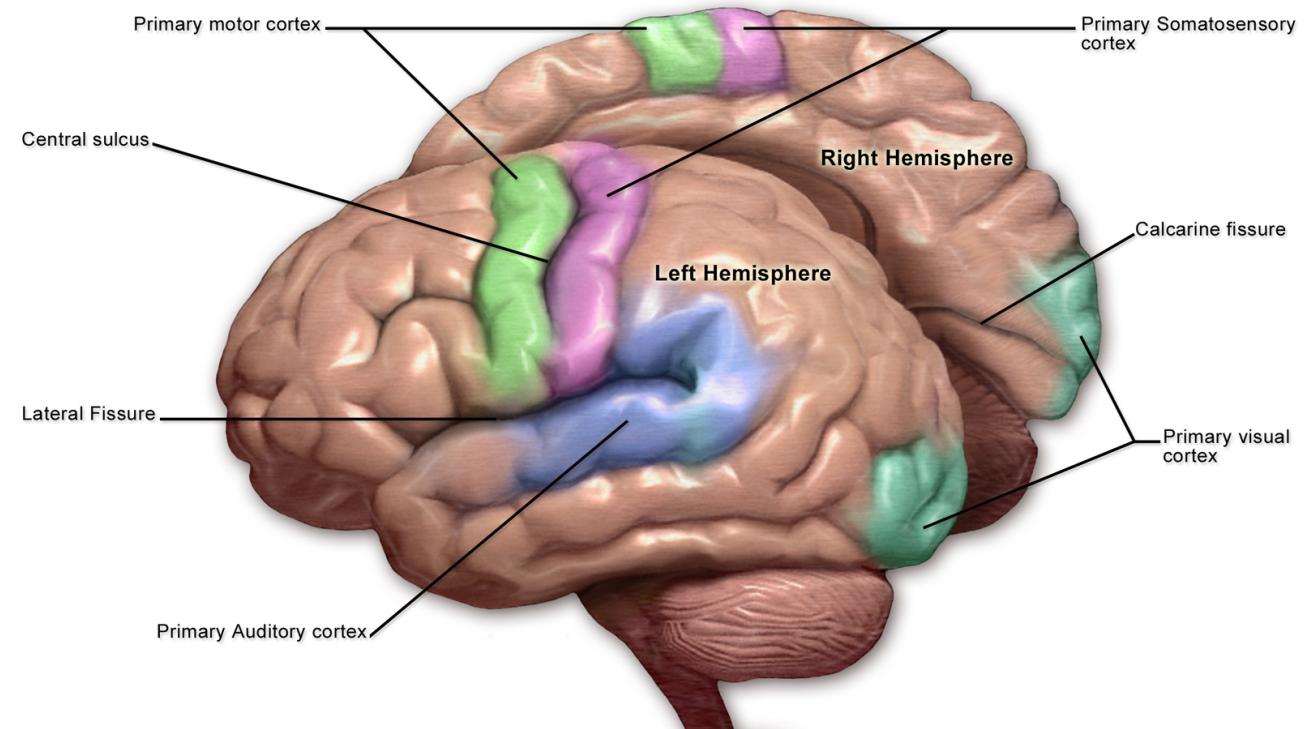
You get a combinatorial explosion problem as you increase the number of microservices



Messaging or Data Plane



Information or Control Plane



Messaging Systems Offer an Improvement

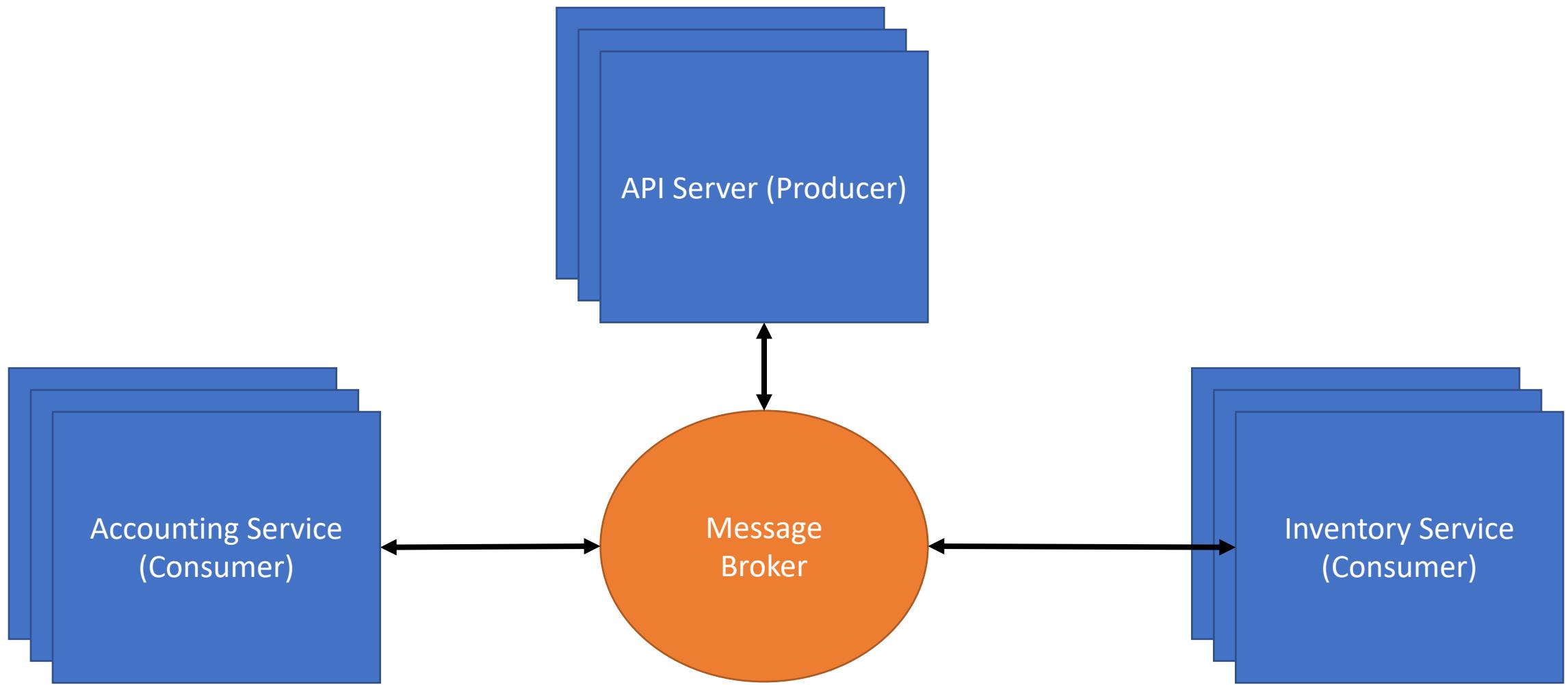
Last week, we looked at
RabbitMQ

You can use RabbitMQ to build a
more robust microservice system

But RabbitMQ itself is not “cloud
native”.

What does a “cloud native”
messaging system look like?

Messaging Systems



A Distributed System Case Study: Apache Kafka

High throughput messaging for diverse consumers

Lecture Sources

- Kreps, J., Narkhede, N. and Rao, J., 2011, June. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB* (pp. 1-7).
- Wang, G., Koshy, J., Subramanian, S., Paramasivam, K., Zadeh, M., Narkhede, N., Rao, J., Kreps, J. and Stein, J., 2015. Building a replicated logging system with Apache Kafka. *Proceedings of the VLDB Endowment*, 8(12), pp.1654-1655.
- <https://kafka.apache.org/documentation/>
- <https://sookocheff.com/post/kafka/kafka-in-a-nutshell/>

Why Look at Kafka?

You can use Kafka to build sophisticated, log-centric distributed systems

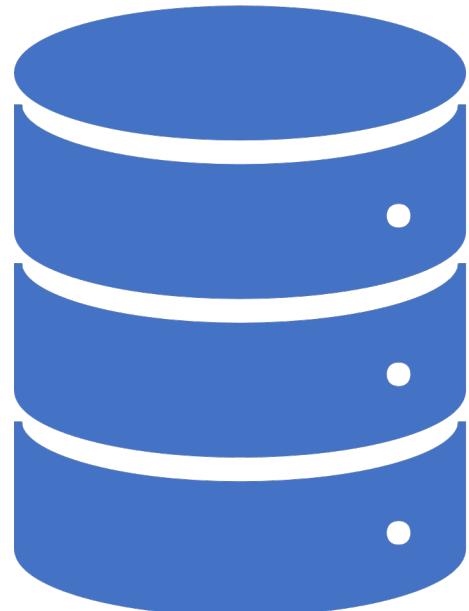
- LinkedIn does this at a huge scale

You can examine how Kafka works to understand how to build distributed systems generally

- Choices, tradeoffs, strategies, design patterns

Log-Centric Architecture

“The Log: What every software engineer should know about real-time data's unifying abstraction”, Jay Kreps,
<https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying>



Logs and State Machines: Databases

- Every time you do a Create, Update, or Delete, you change the DB state
- You need to store both the data and all the operations that got you to the current state.
- You can use this information to completely replicate the state of the DB on another server if you have a crash

Logs and State Machines: Git

- Git stores all the commits as diffs from the previous commit.
- You can restore any previous state of the code base

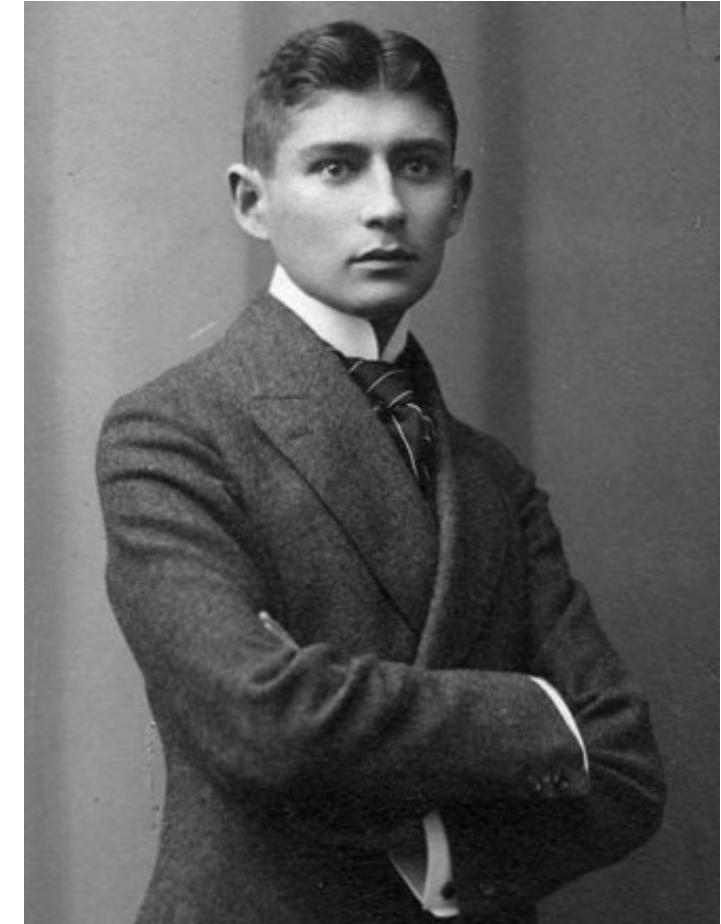
Event Sourcing

- <https://martinfowler.com/eaaDev/EventSourcing.html>
- “Event Sourcing ensures that **all changes to application state are stored as a sequence of events**. Not just can we query these events, **we can also use the event log to reconstruct past states**, and as a foundation to automatically adjust the state to cope with retroactive changes.”
- Fowler thought about this for single applications. What about distributed applications with distributed state?

RabbitMQ versus Apache Kafka



OR



PUSH



VS

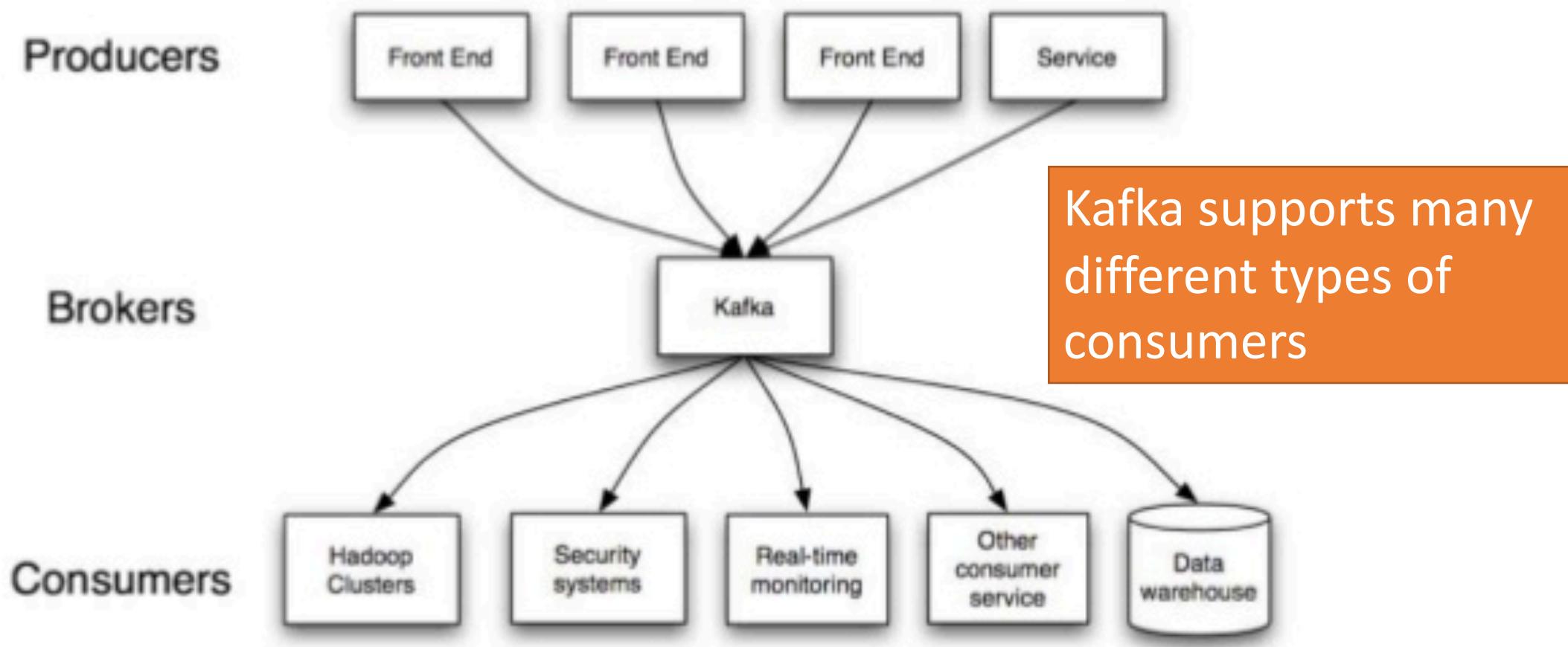
PULL



LinkedIn's Requirements

- Volume: LinkedIn needs to push billions of messages per day to a wide variety of consumers
- Real-time processing applied to activity streams
- Asynchronous processing for log analysis
- Need to support both user-facing applications and system applications
- And they needed a way to recover a large-scale infrastructure in cases of failures at all scales.

Kafka decouples data-pipelines



Consequences of Use Case Requirements

- Consumers must decide when to pull the data
 - Fast or slow, small or large
- This means that the messaging system needs to store a lot of data (TBs)
 - This is not what traditional message systems are designed to do.
 - Kafka will need an efficient way to find the requested message
- Virtue from Necessity: support message rewind and replay
 - This is not a normal operation for a queue, which removes messages after they are delivered.
 - **Treat the accumulated, ordered messages as input for a state machine**

Cloud-Native Messaging: Kafka	Enterprise Messaging: AMQP
Brokers are stateless.	Brokers are state-full.
Messages can be delivered in batches	Messages are individually delivered
“At least once” delivery	“Exactly once” delivery
Eventual consistency across multiple brokers	Strong consistency
Optimized for highly variable latency, large message throughput	Optimized for low latency delivery of smaller messages
The system is designed to replay messages.	Replay is an add-on

Apache Kafka resembles in some ways the REST architecture (idempotency)

Let's Look at Some Details

Apache Kafka Terminology: Topic-Based Publish-Subscribe

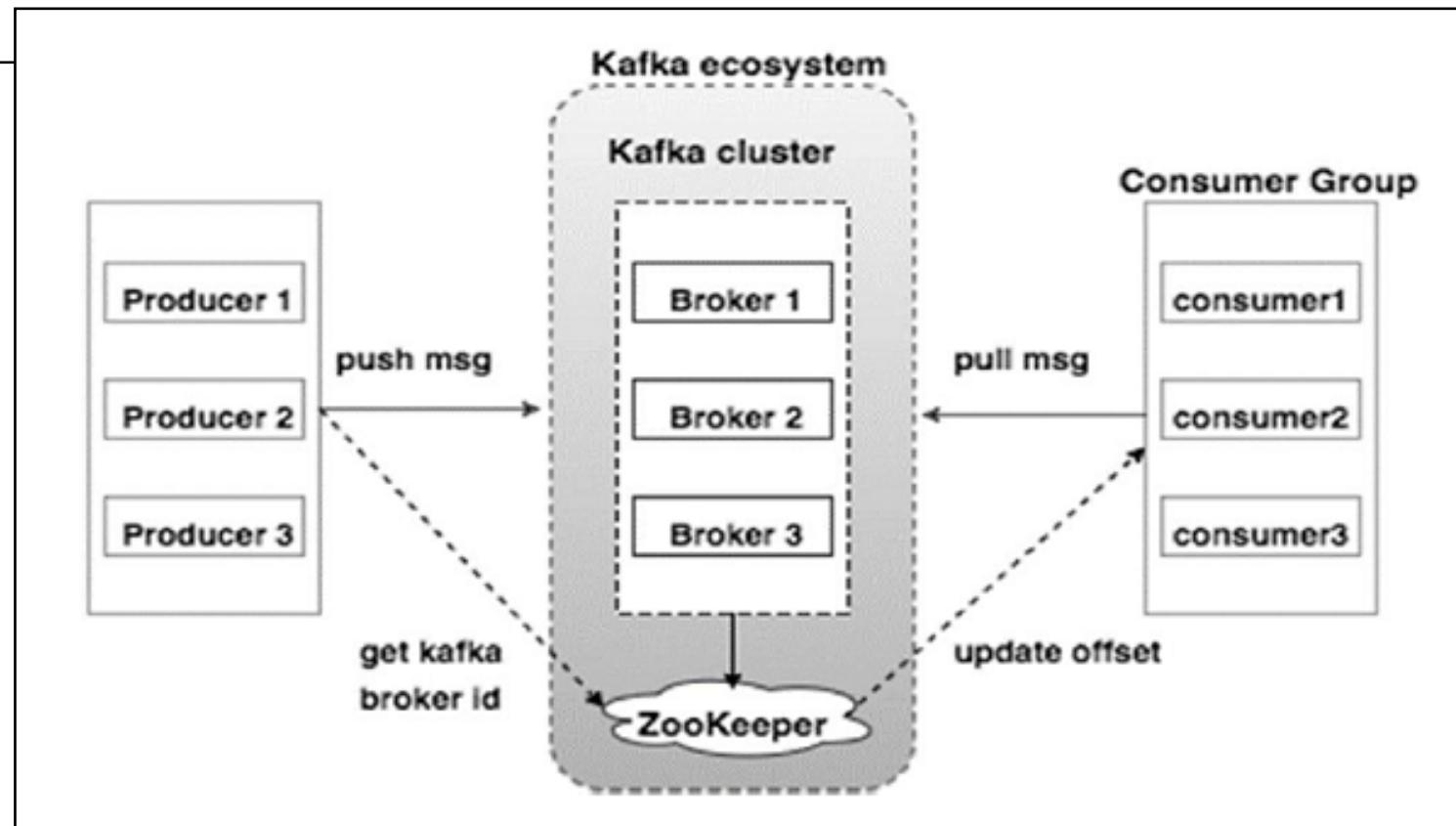
Component	Description
Topic	The label for a stream of messages of a particular type. Kafka further divides topics into partitions .
Producer	An entity that publishes to a topic by sending messages to a broker
Broker	An entity on a network that receives, stores, and routes messages.
Consumer	An entity that subscribes to one or more topics. Kafka generalizes this to Consumer Groups

Kafka Brokers

Connecting message producers to consumers

Kafka Uses Clusters of Brokers

- Kafka is run as a cluster of brokers on one or more servers.



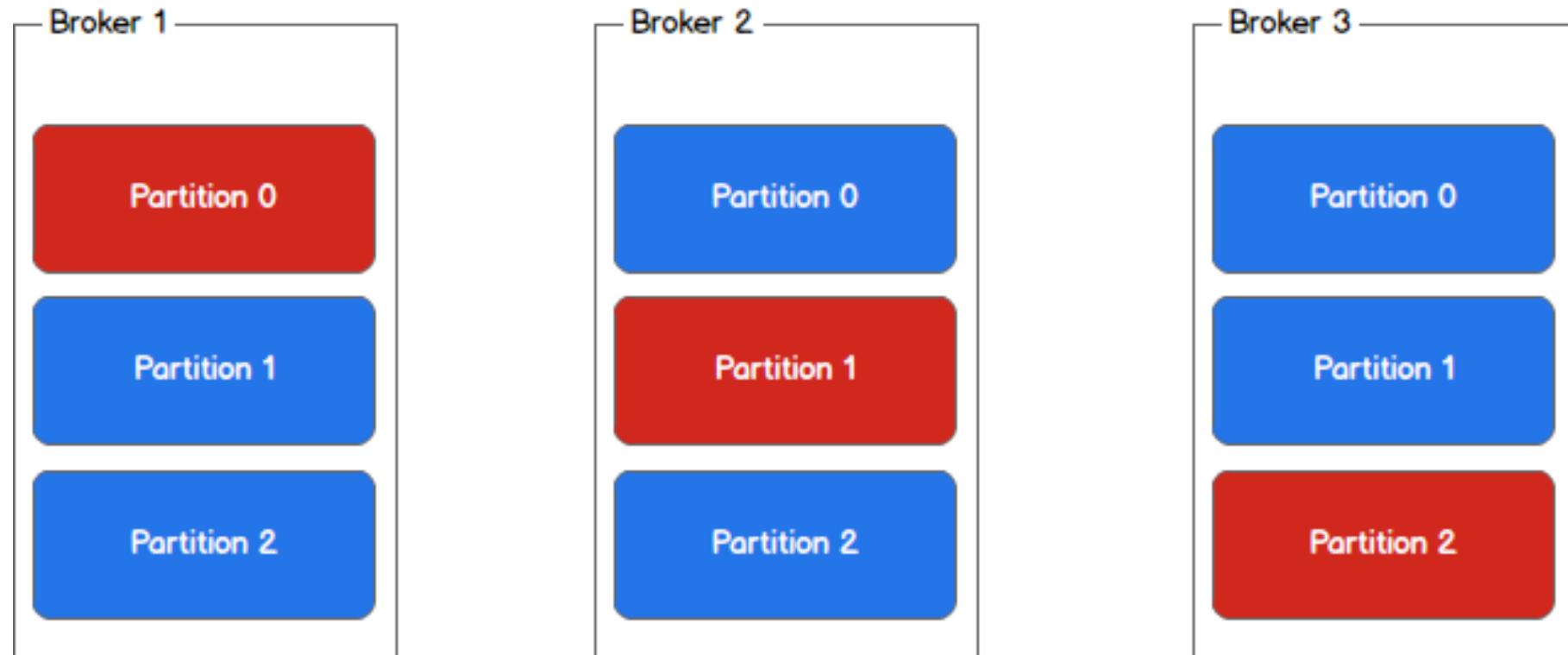
Kafka uses Zookeeper to coordinate it all. Zookeeper manages the control plane

Topics and Partitions

Topics are broken up into partitions that span multiple brokers

Kafka Partitions Are Replicated

Leader (red) and replicas (blue)



If a leader for a partition replica fails, a follower becomes the new leader

Aside: Write-Ahead Logging

- Write-Ahead Logging: this is a technique of writing your file first and then having your broker read the file.
 - This is the reverse of the way logging normally works
 - Why? If the broker needs to be restarted, it reads its log to recover its state.
 - The way a broker works in recovery mode is the same as the way it works ordinarily.
- Kafka uses the file system
 - Linux file systems already have many sophisticated features for balancing in-memory versus on-disk files

Aside: Write-Ahead Logging

- “Kafka acts as a kind of write-ahead log that records messages to a persistent store and allows subscribers to read and apply these changes to their own stores in a system appropriate time-frame.”
 - Goodhope et al

Producers

Kafka Producers

Producers publish data to the topics of their choice.

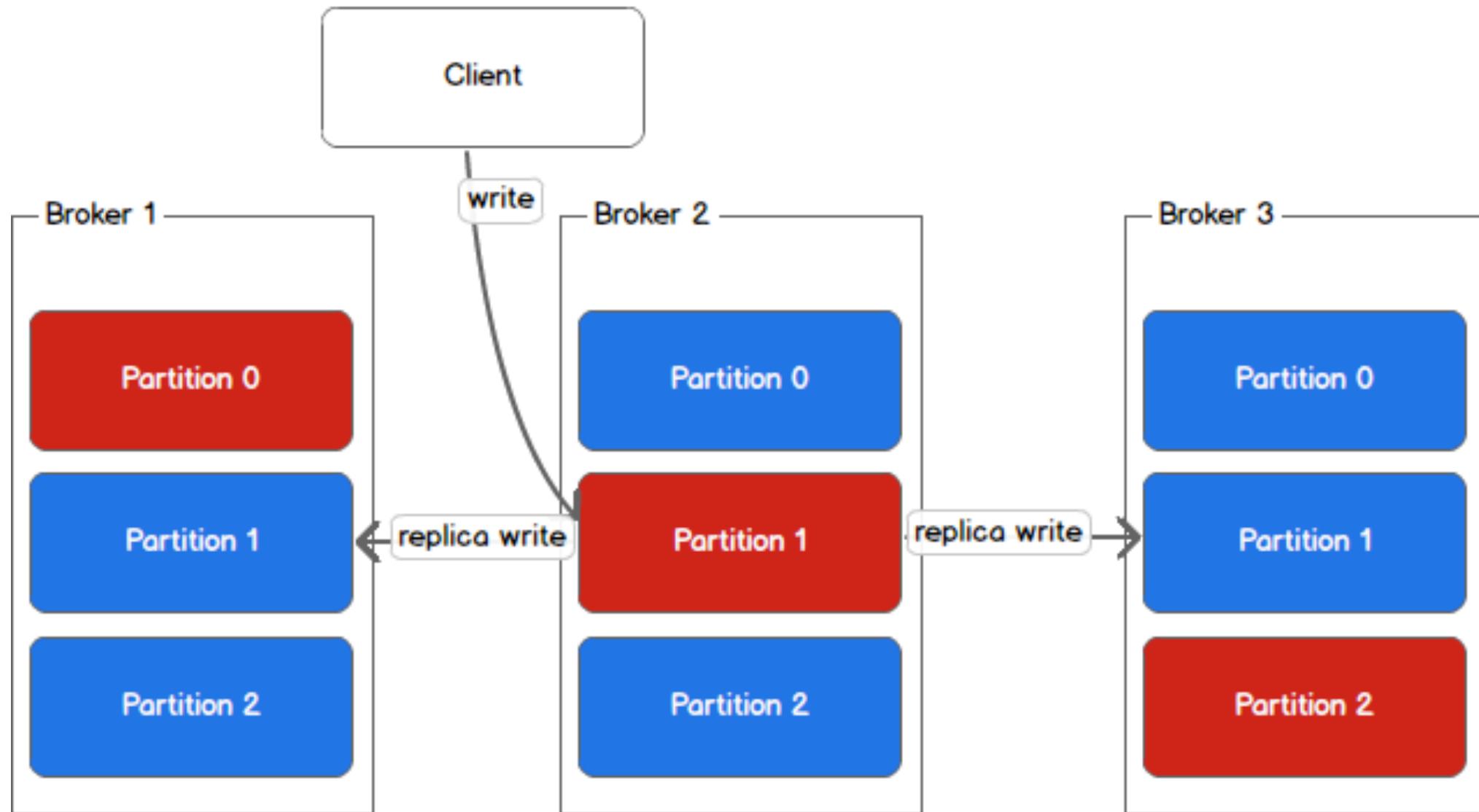
The producer is responsible for choosing which record to assign to which partition within the topic.

This can be done in a round-robin fashion

Producers write to the partition's leader.

The broker acting as lead for that partition replicates it to other brokers

Leader (red) and replicas (blue)



A producer writes to Partition 1 of a topic. Broker 2 is the leader. It writes replicas to Brokers 1 and 3

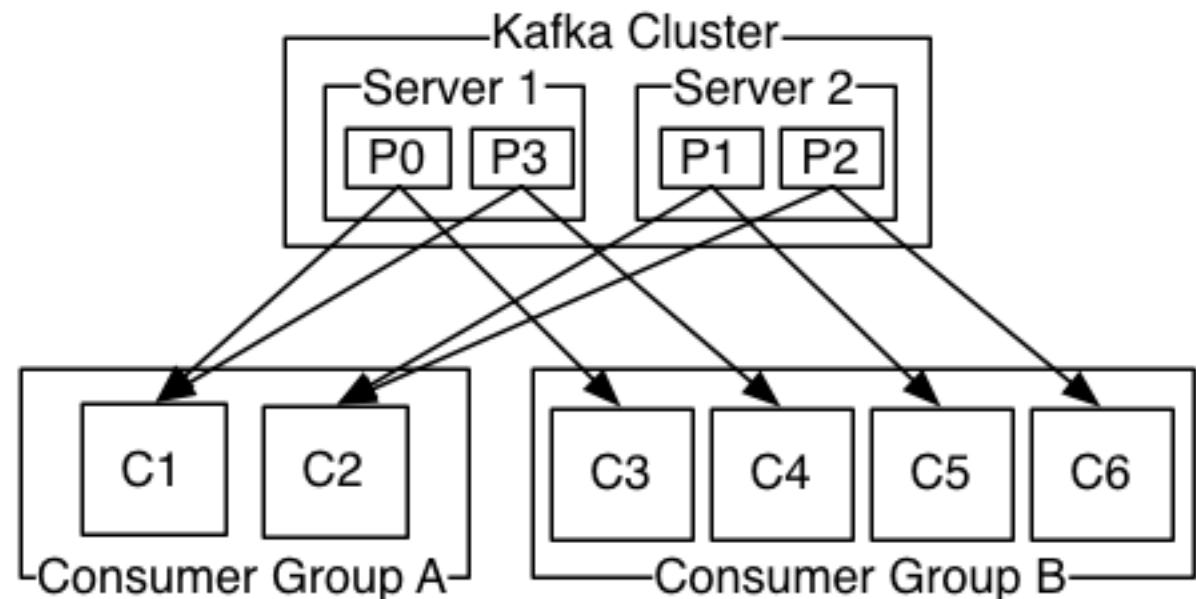
Consumer Groups

Kafka Consumer Groups

- Consumer groups contain one or more consumers of a given topic.
 - Many consumer groups can subscribe to the same topic.
- A consumer group is whatever is useful for a given consuming application
 - You can use consumer groups to implement work queues
- Only one member of a consumer group consumes the messages on a given partition
 - Avoid locking and other complicated state management issues

Consumers and Consumer Groups

- In a consumer group, each member is associated with a specific partition.
- Collectively, a consumer group receives all messages on a topic.
- If the group expands or contracts, it can rebalance using Kafka's built-in Zookeeper



Consumer Group Scenarios

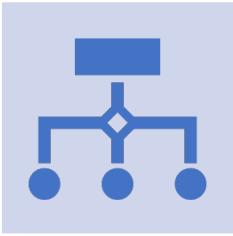
- Consumer Load Balancing: All consumers are in one group
 - Consumer Group B, previous slide
- Broadcast: each consumer is its own group
 - Each consumer receives messages from all partitions
- Partition Load Balancing: $N(\text{Partitions}) > N(\text{Consumers in Group})$
 - Consumer Group A, previous slide
 - But messages between partitions may not be ordered
- Consumer ordering: $N(\text{Partitions}) == N(\text{Consumers in a Group})$
 - Each member gets messages from only one partition

We assume round-robin message distribution to partitions. Note message order is preserved only within a partition

Rewinding and Replaying Messages

- Kafka persistently stores messages much longer than conventional messaging systems
 - Doesn't assume low-latency delivery.
- The state of a topic is the message order, stored in partition files.
- A consumer can request the same messages many times if it needs to.
 - Why? Rollback. A consumer may have had a bug, so fix the bug and consume the message again with the corrected code.
 - Recall Blue-Green deployments
 - Or the consumer may have crashed before processing the message
 - This is NOT a typical queue pattern.
- Rewinding is much more straightforward in a pull-based architecture.

Is Kafka a Good Choice for You?

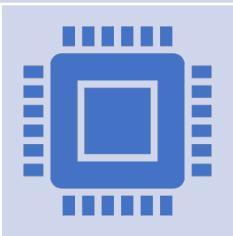


Think about your requirements for routing messages to replicated services.

Can you map your system to topics and partitions?

Do you need a log-centric system?

Kafka combines the data plane and control plane

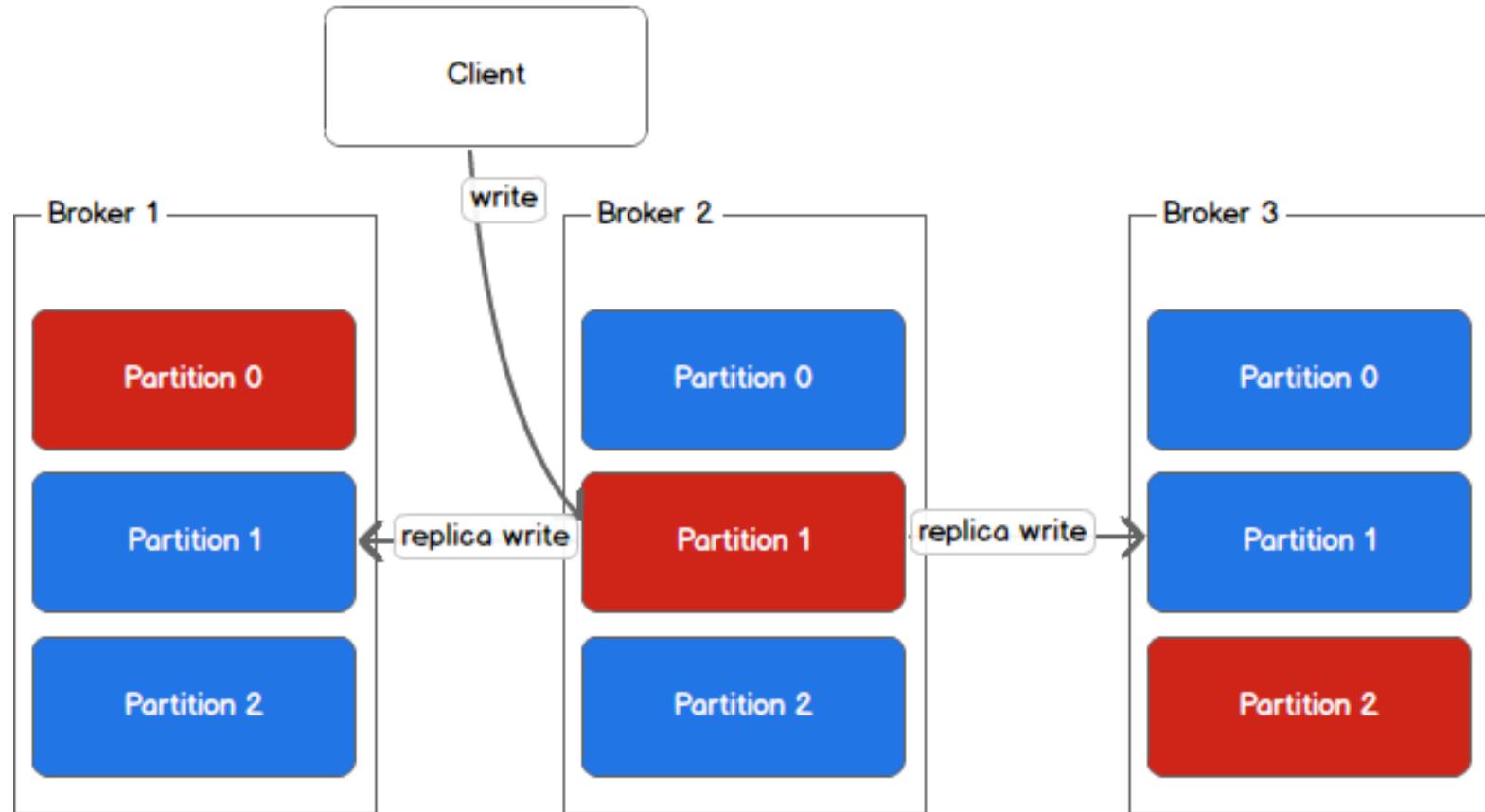


You can still build microservices using RPC or REST style systems

But you'll need to expose the control plane more directly

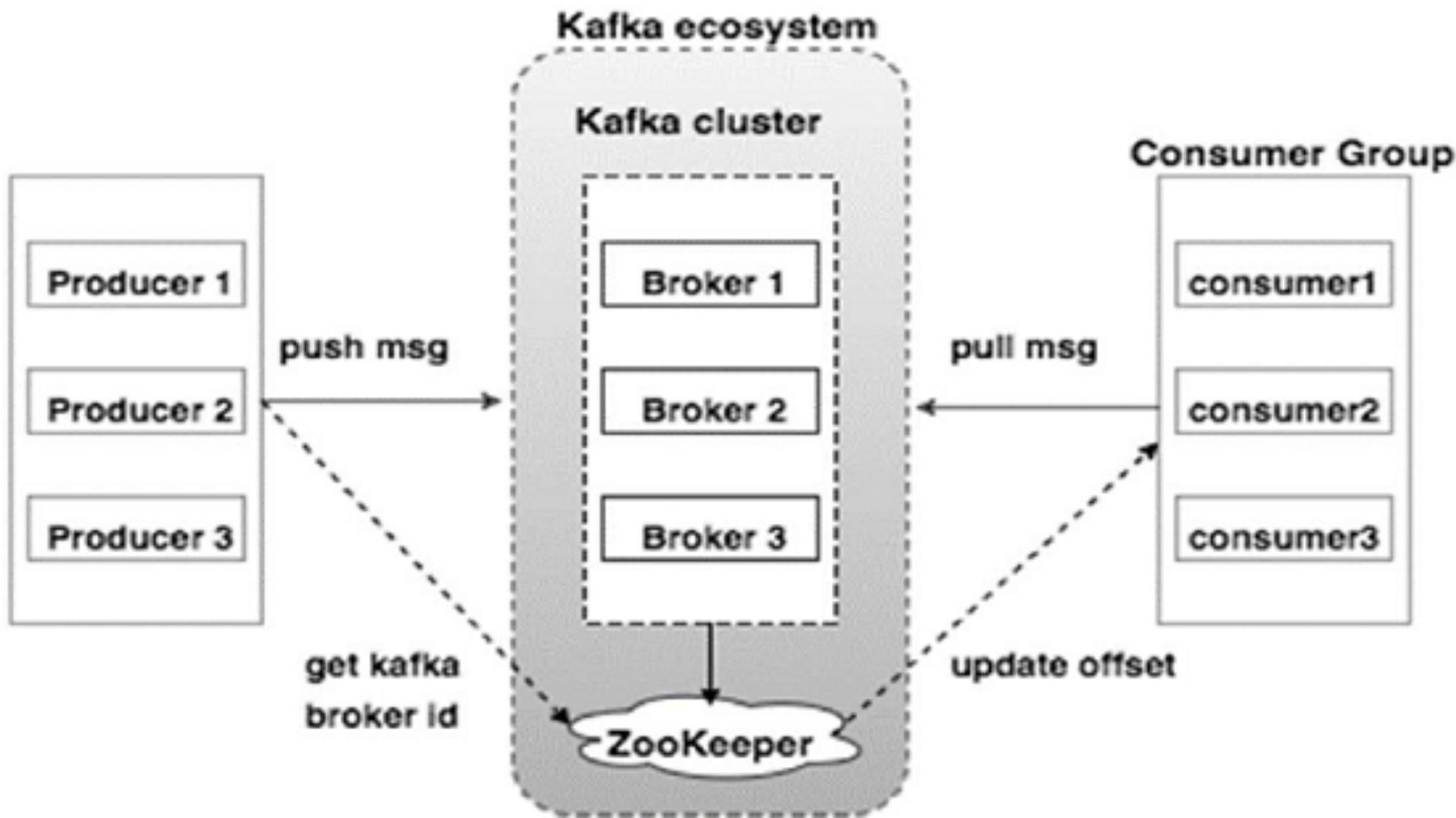
Next Time:
Message
ordering and
leader election
across brokers in
detail

Leader (red) and replicas (blue)



Kafka and Zookeeper

Managing brokers, consumers, and producers



Zookeeper Registry	Description	Node Type
Broker Registry	Contains brokers' host names, ports, topics, and partitions. Used by the brokers to coordinate themselves. Ex: deal with a broker failure.	EPHEMERAL
Consumer Registry	Contains the consumer groups and their constituent consumers.	EPHEMERAL
Ownership Registry	Contains the ID of the consumer of a particular consumer group that is reading all the messages. This is the “owner”.	EPHEMERAL
Offset Registry	Stores the last consumed message in a partition for a particular consumer group.	PERSISTENT

Each consumer places a watch on the broker registry and the consumer registry and will be notified if anything changes.

Delivery Guarantees

- Kafka chooses “at least once” delivery.
 - It is up to the consuming application to know what to do with duplicates
 - Duplicates are rare, occur when an “owning” consumer crashes and is replaced
 - **Two-phase commits** are the classic way to ensure “exactly once” delivery.
- Messages from a specific partition are guaranteed to come in order.
- Kafka stores a **CRC** (a hash) for each message in the log to check for I/O errors

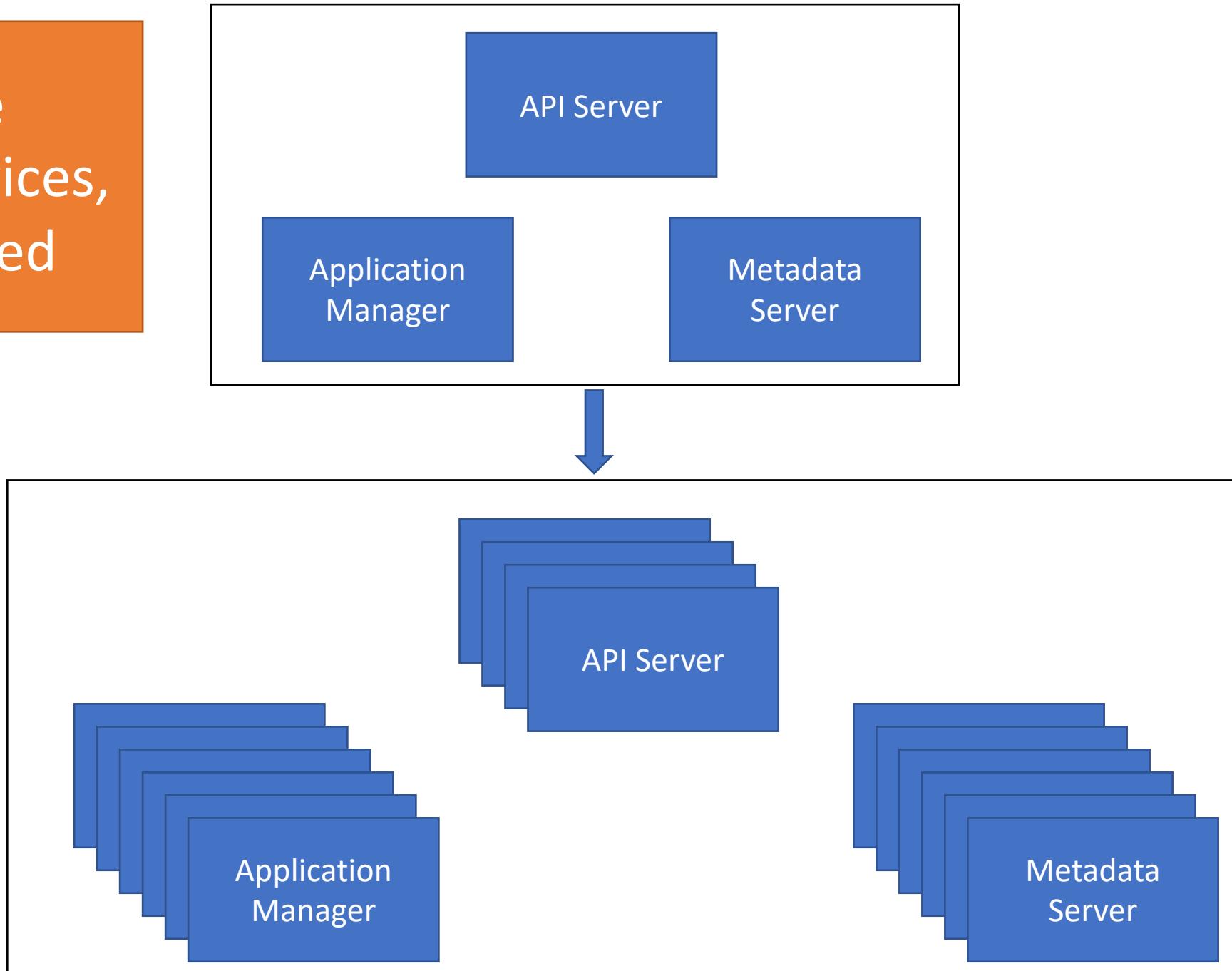
What About the Message Payload?

- Apache Kafka supports clients in multiple programming languages.
- This means that the message must be serialized in a programming language-neutral format.
- You can make your own with JSON or XML
- Kafka also supports Apache Avro, which is a schema-based binary serialization format.
 - Compare Avro with Apache Thrift and Protobuf
- Efficient message formats are essential for high throughput systems

Kafka, Airavata, and Microservices

Some thought exercises

Three
microservices,
replicated



Some General Distributed Systems Principals

Kafka, logs, and REST

Log-Centric Architecture

- Distributed servers use a replicated log to maintain a consistent state
- The log records system states as sequential messages.
- New servers can be added to expand the system or replace malfunctioning servers by reading the log
 - No in-memory state needs to be preserved
- The server just needs to know that it has an uncorrupted (not necessarily latest) version of the log.
- You can use this approach for both highly consistent and highly available systems (CAP)

Kafka is a log-oriented system that can be used to build other log-oriented systems

This just leaves one little problem...

How do you keep the log replicas up to date?

Primary-Backup Replication	Quorum-Based Replication
1 leader has the master copy and followers have backups	1 leader has the master copy and followers have backups
On WRITE, the master awaits the appending to all backup for acknowledging the client	On WRITE, the master waits on only a majority of the followers to confirm backups before it returns
Supports strong consistency for distributed READS, but doesn't scale easily and has lower throughput	Supports eventual consistency and higher throughput; doesn't require good networking between leader and followers
If the master is lost, restore from a backup	If a master is lost, elect a new leader from the replicas that have the latest data
F+1 replicas can tolerate F failures	2F+1 replicas can tolerate F failures

Kafka State Management

- Kafka brokers are stateless
 - They don't track which messages a client has consumed or not.
 - This is the client's job
 - Brokers simply send whatever the client requests
 - Compare to REST
- Brokers eventually must delete data
 - How does a broker know if all consumers have retrieved data?
 - It doesn't. Kafka has a Service Level Agreement:
 - “Delete all data older than N days” for example

