

# Microservices and Messaging

---

Communications in distributed applications

# Preview of Upcoming Lectures



Basics of Messaging in  
Microservices



Apache Kafka: Cloud  
Native Messaging



Control Plane  
Technologies: Consul  
and Zookeeper



RAFT: Building Fault  
Tolerant Microservice  
Systems



SWIM: Building  
Scalable Microservice  
Systems



Blockchain: Towards  
Unlimited Scaling



Service Meshes: Real,  
Scalable Systems

# From the Intro Lecture: Build on Foundations



**Network Messaging, Messaging  
Patterns, Protocols**



Algorithms



Design Patterns

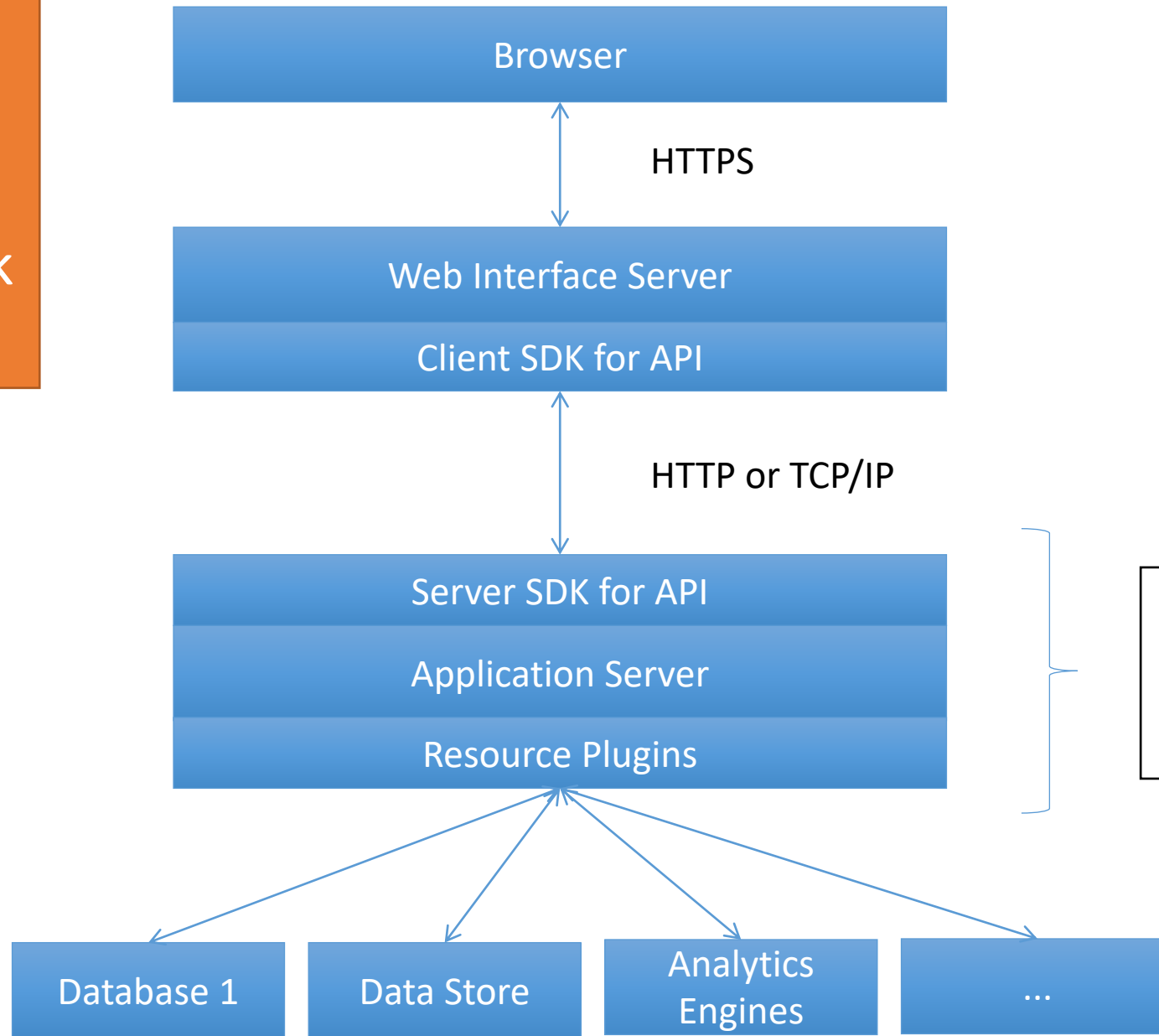


Engineering Practices



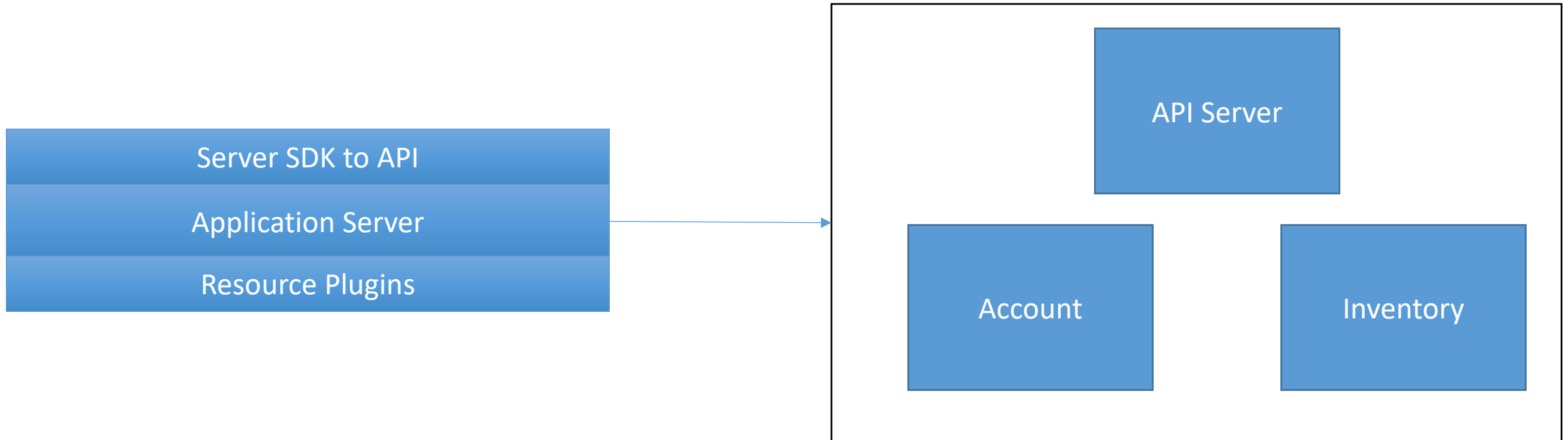
Tools

Three(+)-Tiered Architecture: this is a “north-south” monolithic full stack application.



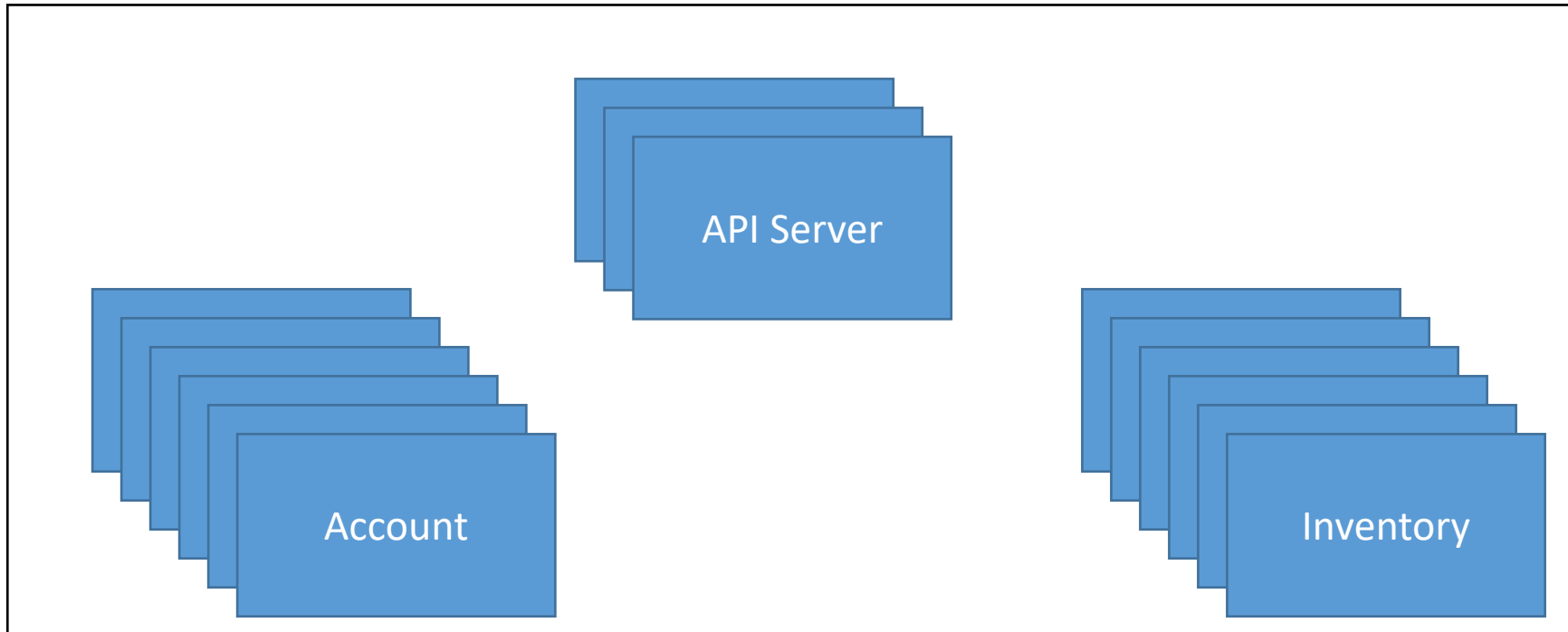
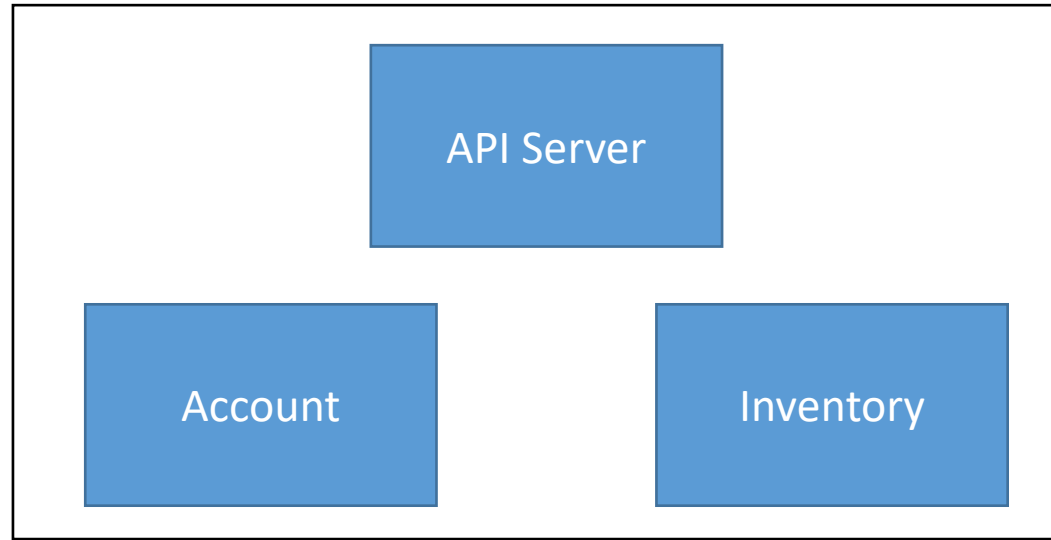
Let's decouple this into microservices

# Basic Components of the Application Server Become Microservices

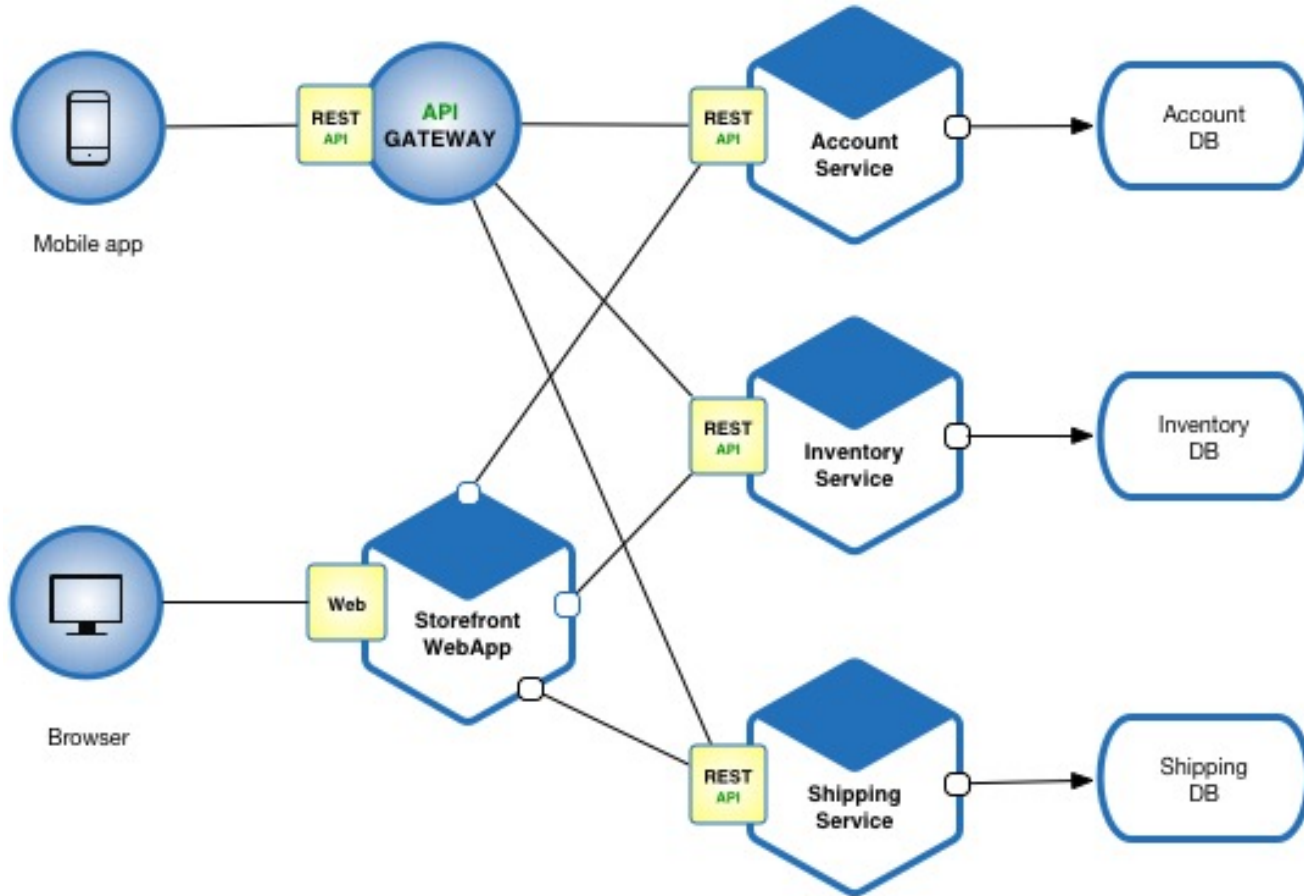


Communications between the microservices on the left are over-the-wire and need to be non-blocking in many cases.

# Replicate the Microservices

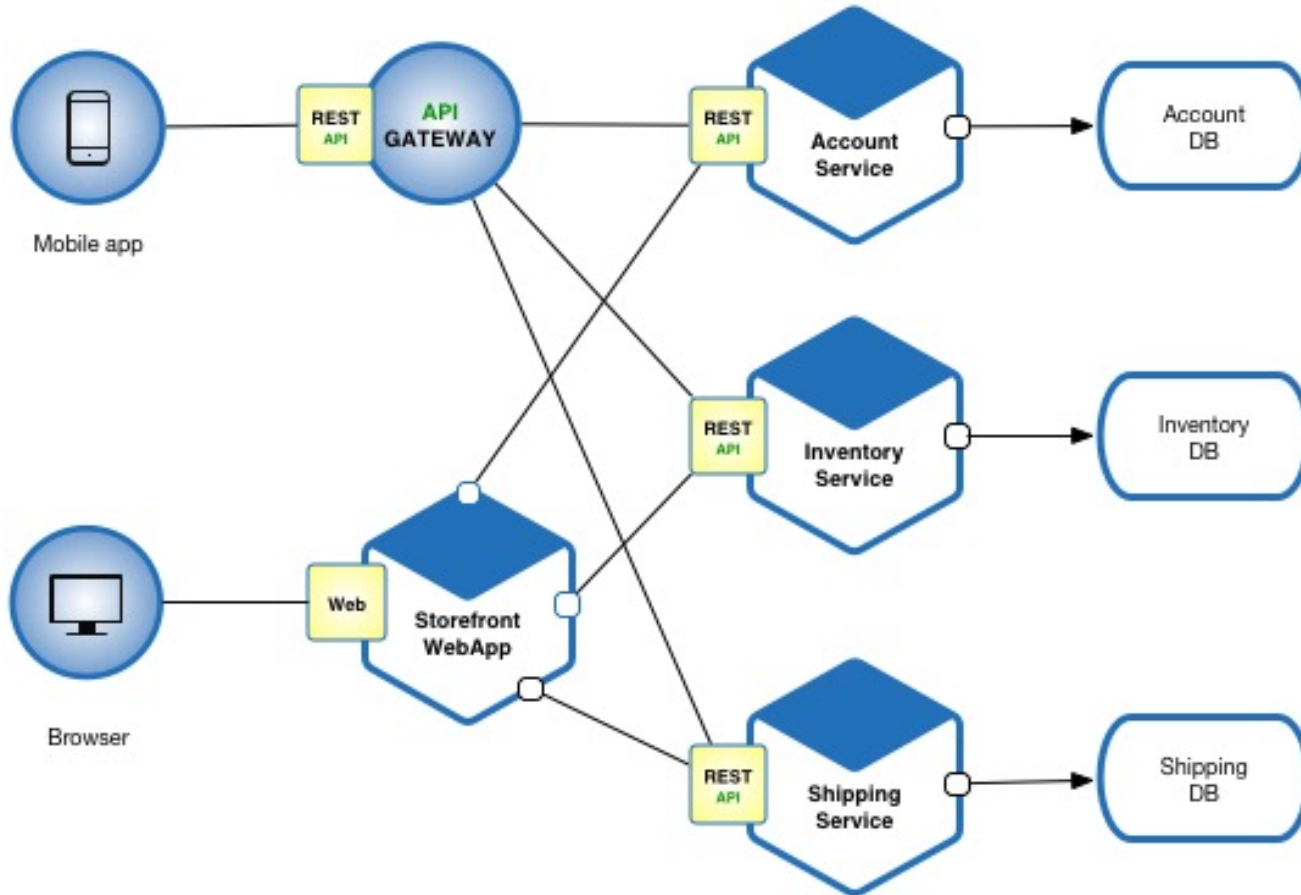


# Multiple Microservices Are Needed to Complete a Process



- Database Per Service Pattern
- Distributed Transactions
- NOTE: I think we can do better than this diagram. It doesn't scale.

# How Can We Make This Robust?



Services need to be replicated for load balancing and fault tolerance

Services need to be discoverable

We need a way to connect endpoints

We need to detect service crashes

Anything else?






How Can We Make This  
Work?

# Answer: Messaging and Coordination

Microservices communicate with other microservices via messaging over networks.

A light orange downward-pointing arrow indicating a flow from the first point to the second.

Microservices need a way to find each other **dynamically**: no hard-coded connections.

A light gray downward-pointing arrow indicating a flow from the second point to the third.

Microservices need to communicate with defined **message formats** or **data models**



# At Internet Scale

Services have IP addresses and domain names

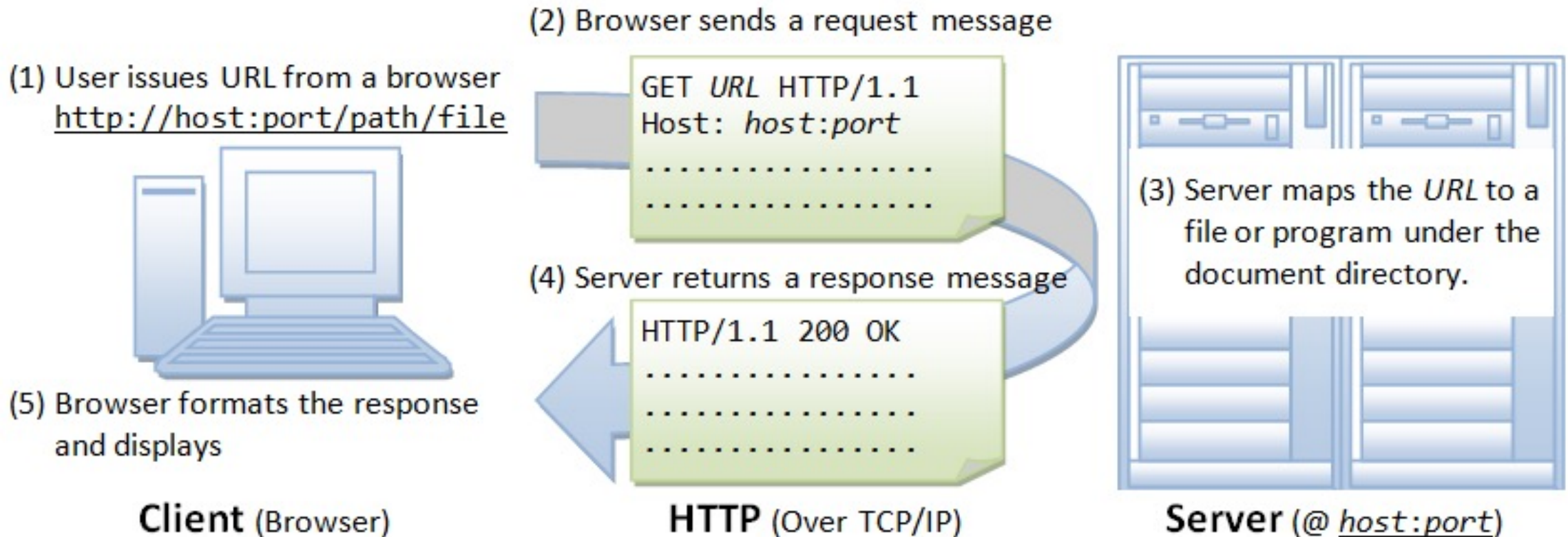
Domain Name Servers match services' domain names to IP addresses

Browsers and servers communicate over HTTP with HTML messages

Humans decide where to go

REST generalizes this for machine-to-machine communications

# HTTP Summary



REST generalizes this for machine-to-machine communication



# That's How the Internet Works

We need to scale this down  
to specific, coordinating  
distributed applications.

Humans aren't around to  
make all the decisions

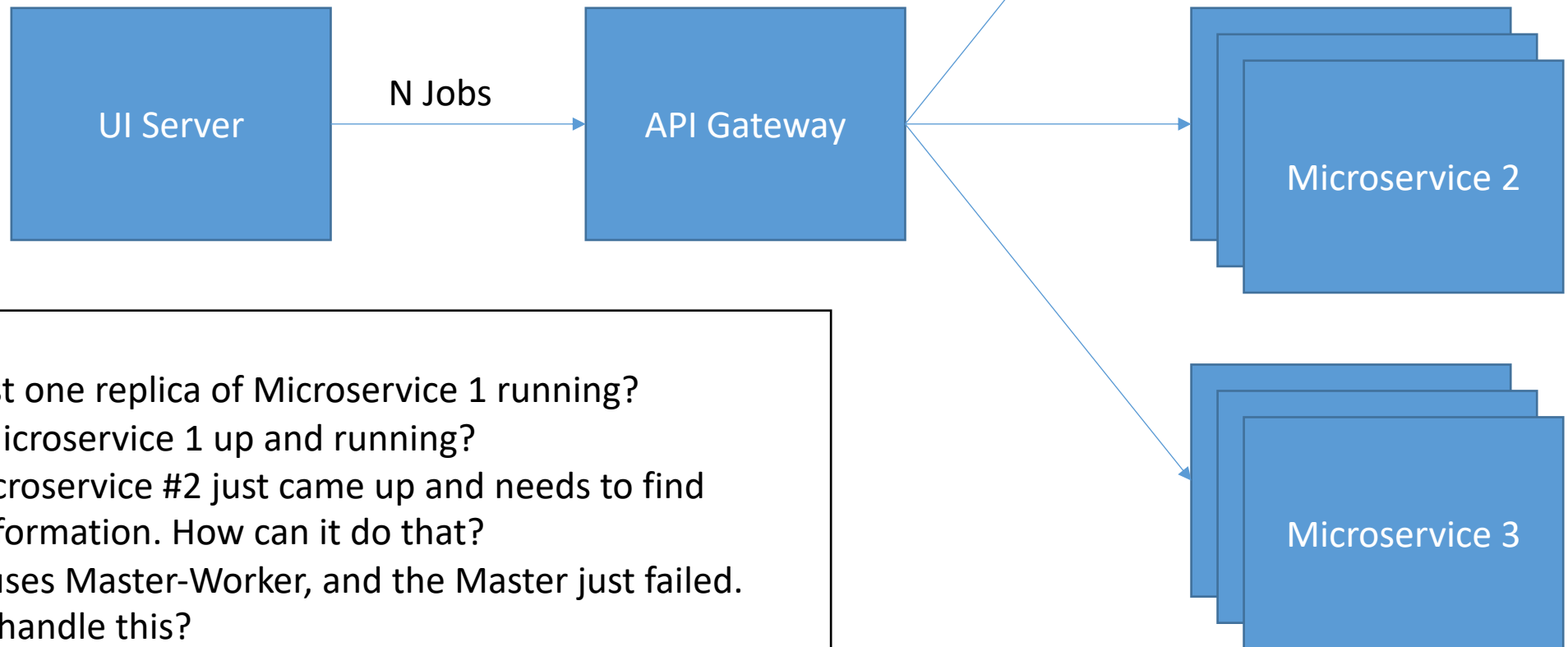
What Are Some  
Problems Using HTTP for  
Microservices?

# Some Answers

- HTTP is point-to-point
  - Sender and receiver are tightly coupled
  - How do you send messages to more than one recipient?
- Request-response
  - Block until you get a response
  - How do you push a message back later?
- Messages (HTML) are designed for human, not machine consumption
  - Hypermedia As The Engine Of Application State (HATEOAS), Swagger attempt to address this for REST

HTTP is not designed for more complicated messaging scenarios

# The Microservice Situation



## Some questions:

- Do I have at least one replica of Microservice 1 running?
- Is Replica 2 of Microservice 1 up and running?
- Replica 1 for Microservice #2 just came up and needs to find configuration information. How can it do that?
- Microservice 3 uses Master-Worker, and the Master just failed. How should we handle this?



# Two Approaches



**Service Mesh Approach:** Go with REST or RPC for communication but introduce a **control plane** information system and **sidecar proxies** to coordinate different end points.



**Message Oriented Middleware Approach:** Use a message broker to route messages AND manage information about endpoints.

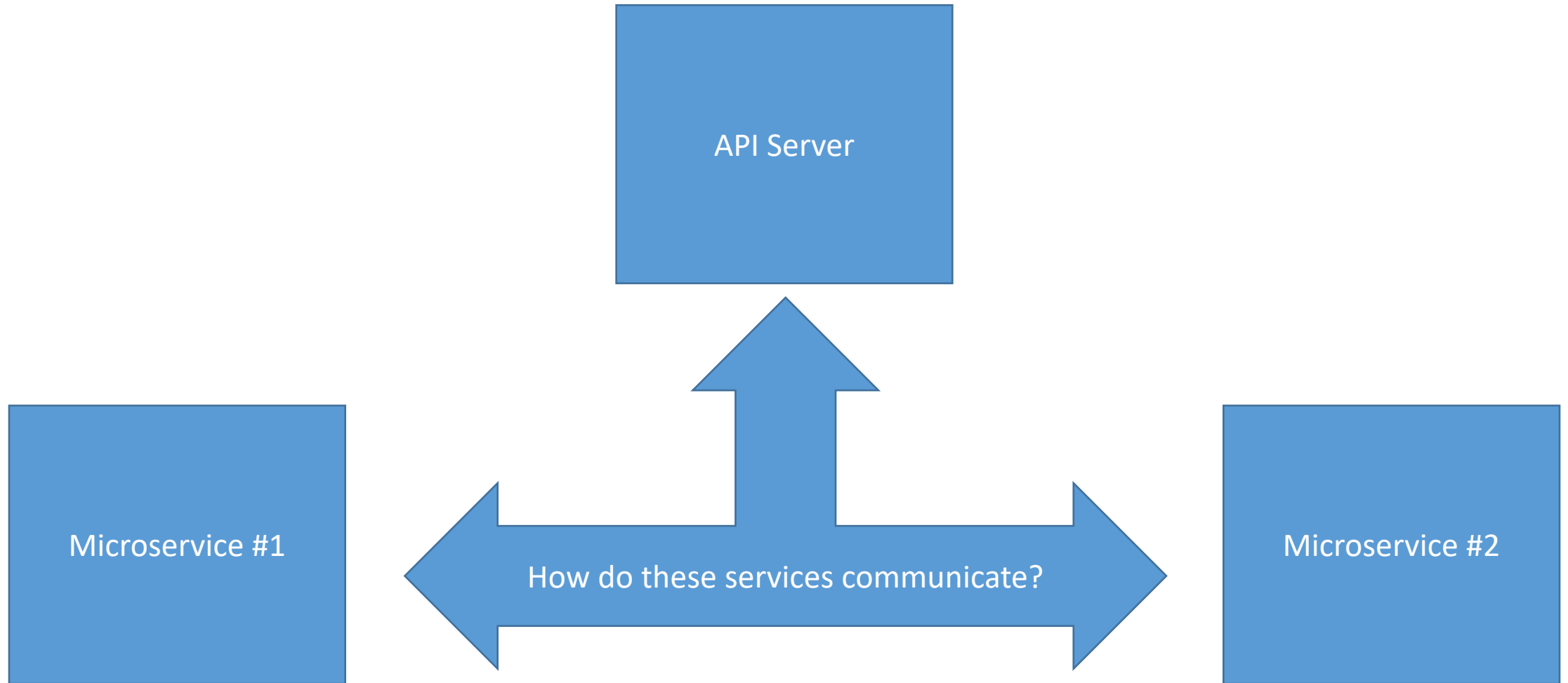
# Messaging in Distributed Systems

---

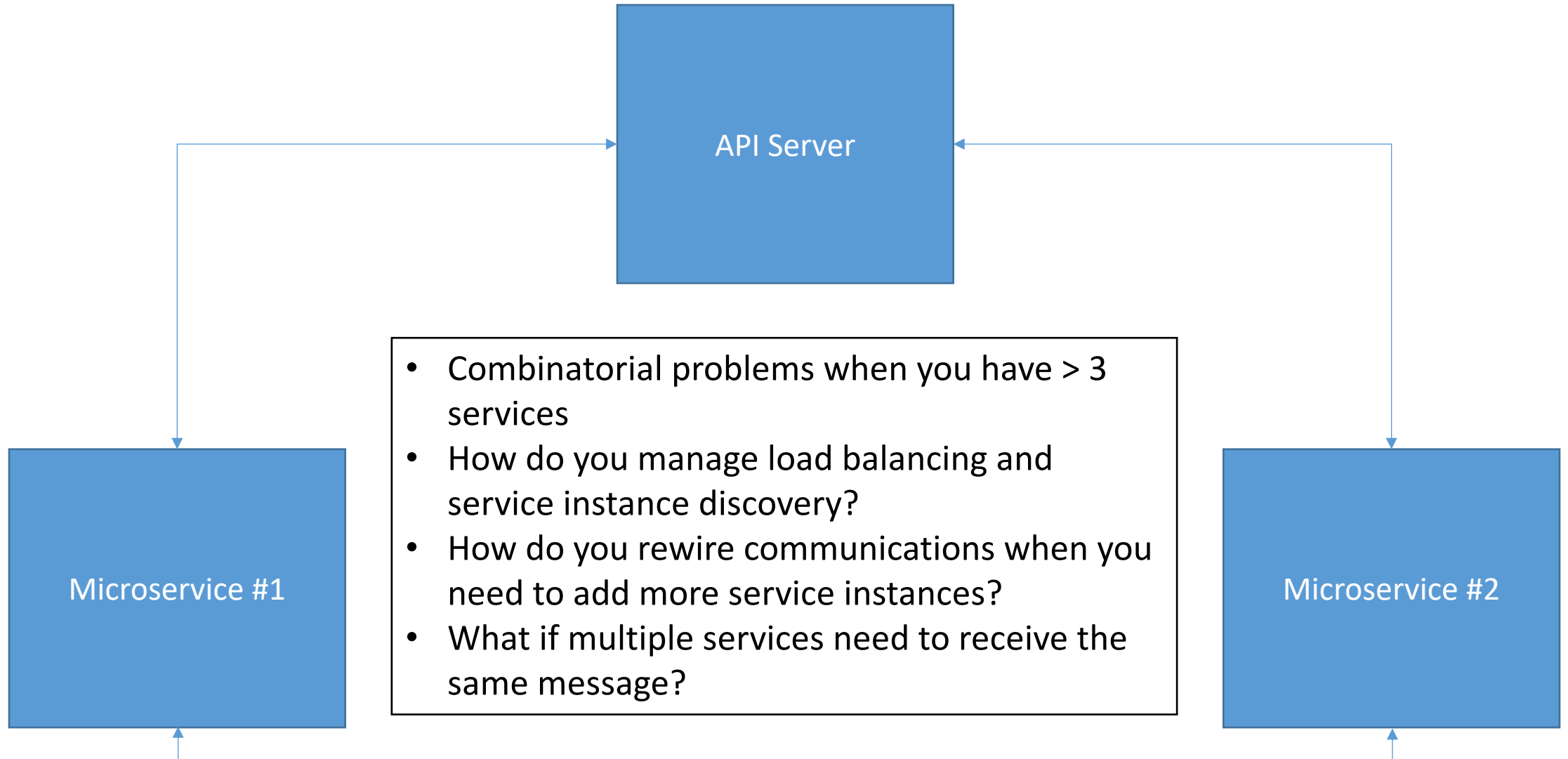
We'll use Advanced Message Queuing Protocol (AMQP) and RabbitMQ overviews as examples

This is not an endorsement of these for the assignments!

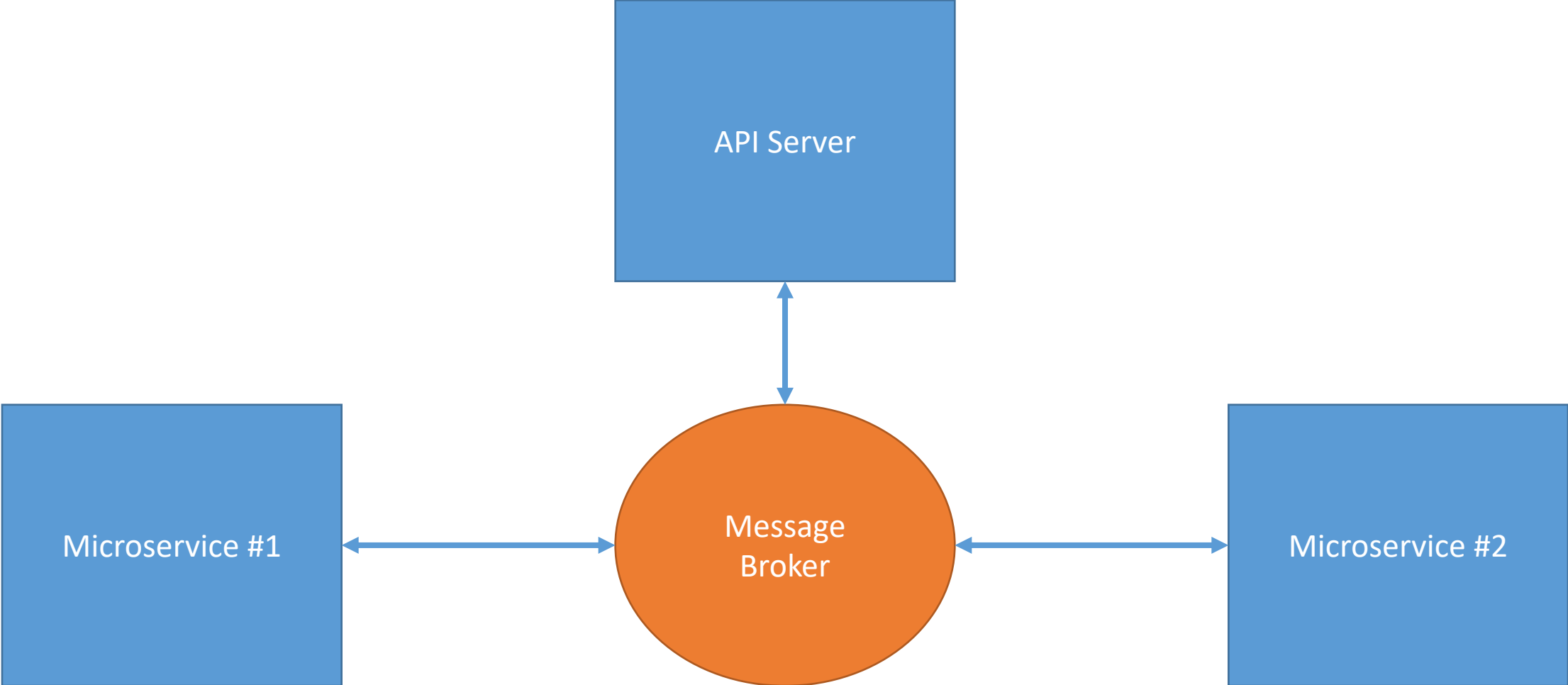
# A Simple Microservice Collection



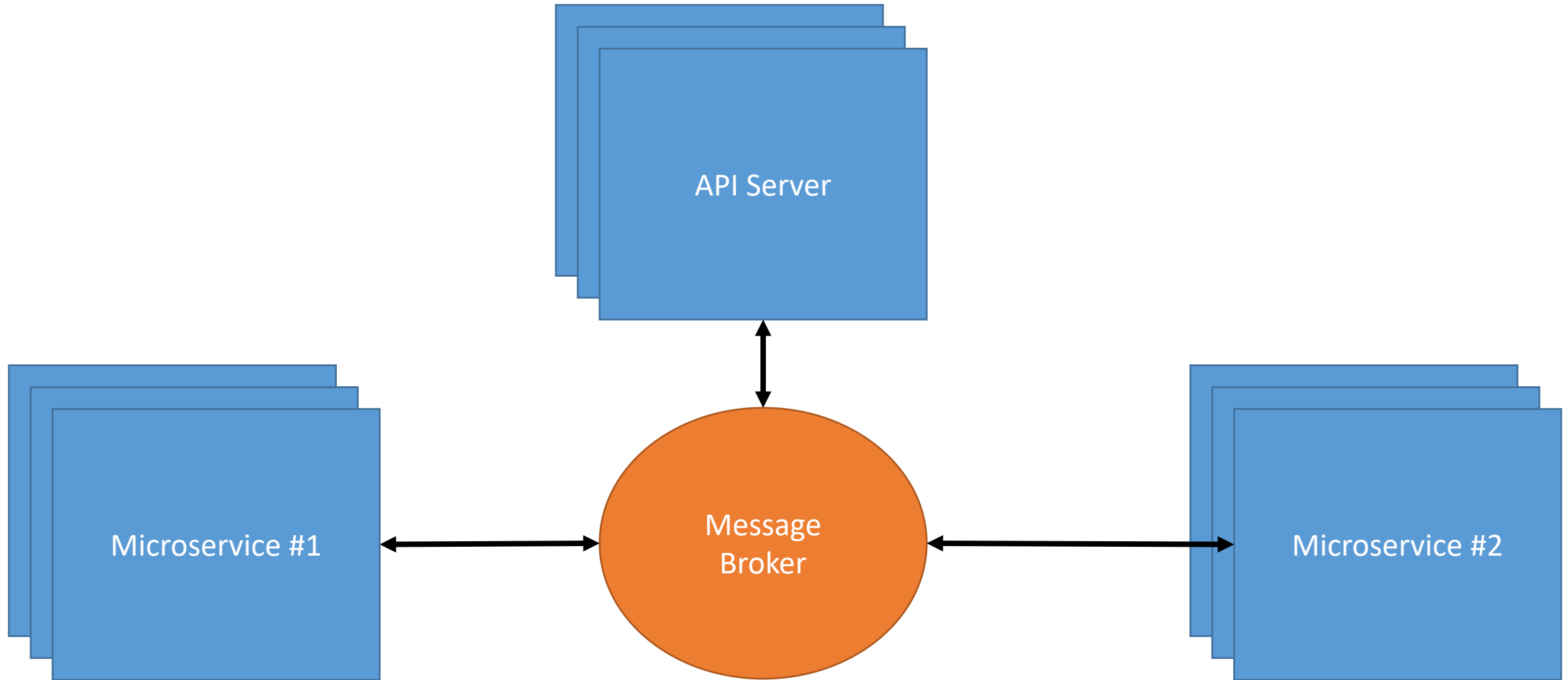
# Point to Point Communication Is Brittle



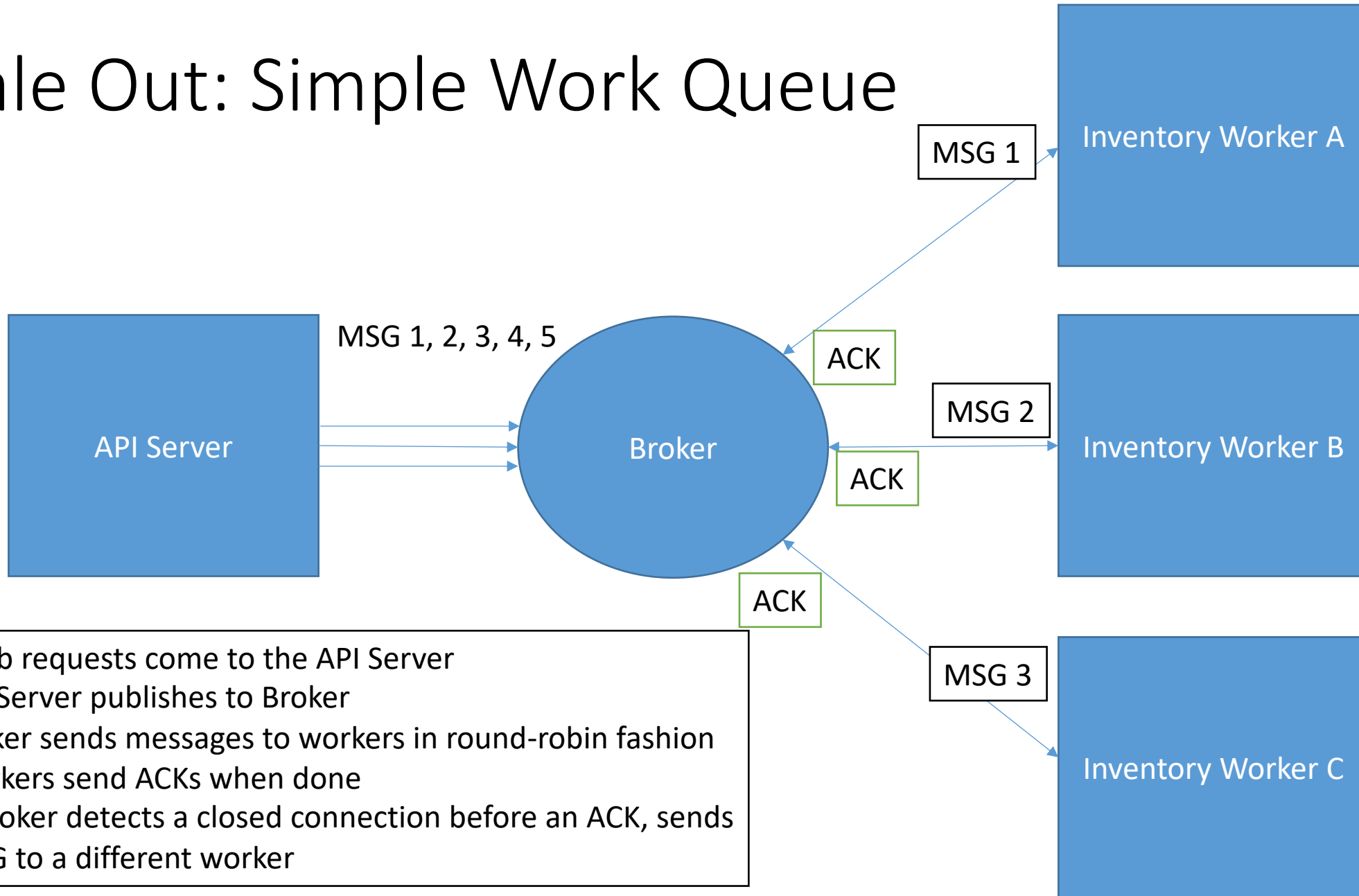
# Messaging Systems Solve Many of These Problems



# Brokers Also Work Well with Service Replication



# Scale Out: Simple Work Queue



- 5 Job requests come to the API Server
- API Server publishes to Broker
- Broker sends messages to workers in round-robin fashion
- Workers send ACKs when done
- If Broker detects a closed connection before an ACK, sends MSG to a different worker

# Value of Message Queuing Systems

- They are queues for messages....
- You can put lots of messages in a queue , which the message broker will deliver with various qualities of service
  - In order
  - Exactly once
  - At most once
  - At least once
- Many-to-many rather than just one-to-one
- Can support both push and pull messaging
- A bit like email for machines



# Value of Message Systems, Continued

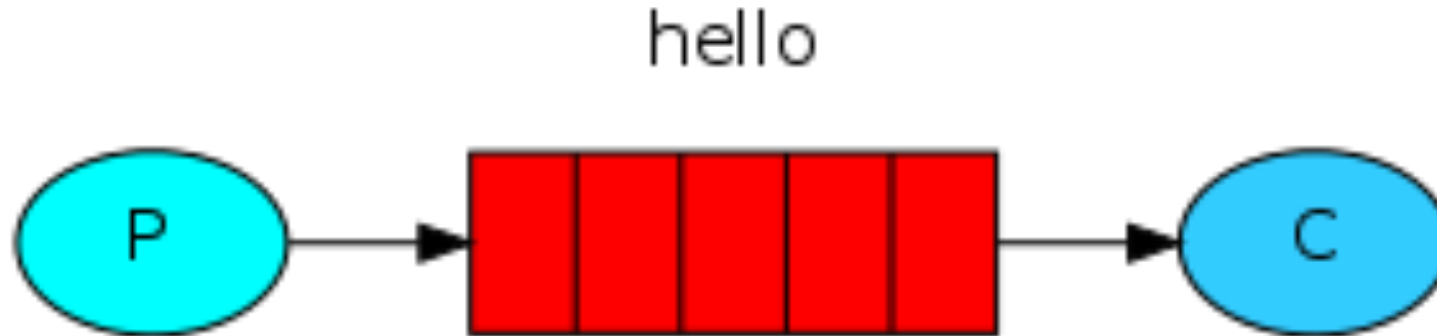
- Messaging systems remove the need for microservices to know too much about each other
  - Services just need to know how to connect to the message broker.
  - The network locations and specific instances of the services can change over time.
- Messages can be synchronous or asynchronous
- Possibly better performing than HTTP:
  - RabbitMQ can multiplex multiple channels of communication over a single TCP/IP connection
  - Apache Kafka can support high throughput

# Message Exchange Patterns

---

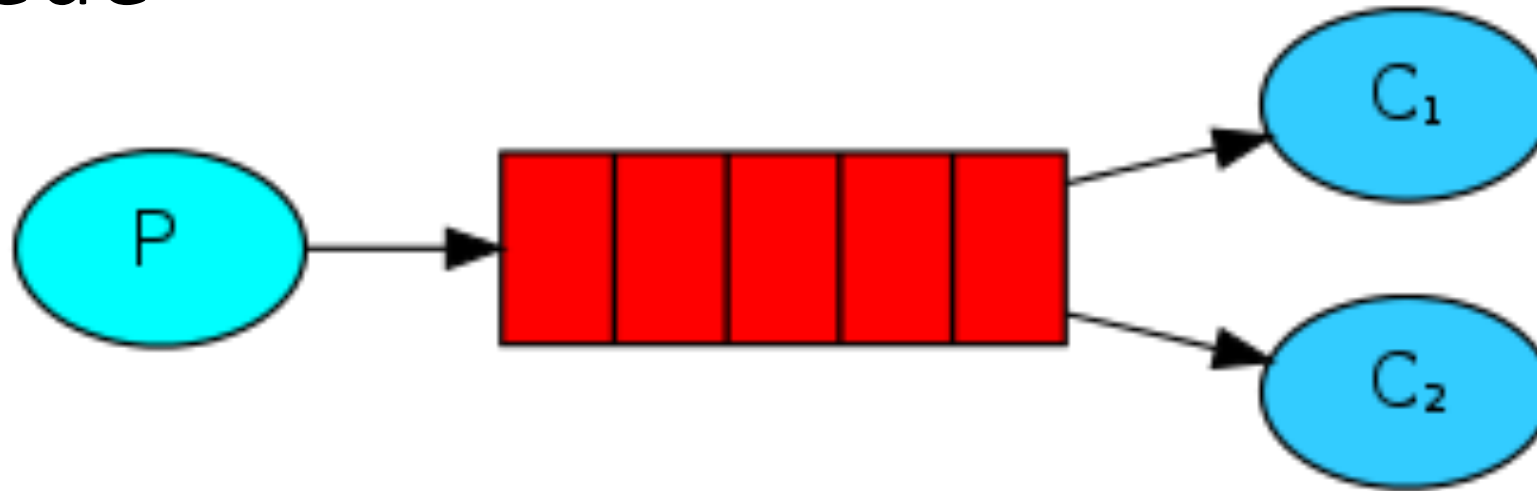
See the RabbitMQ tutorial for more examples

# Point to Point Communication



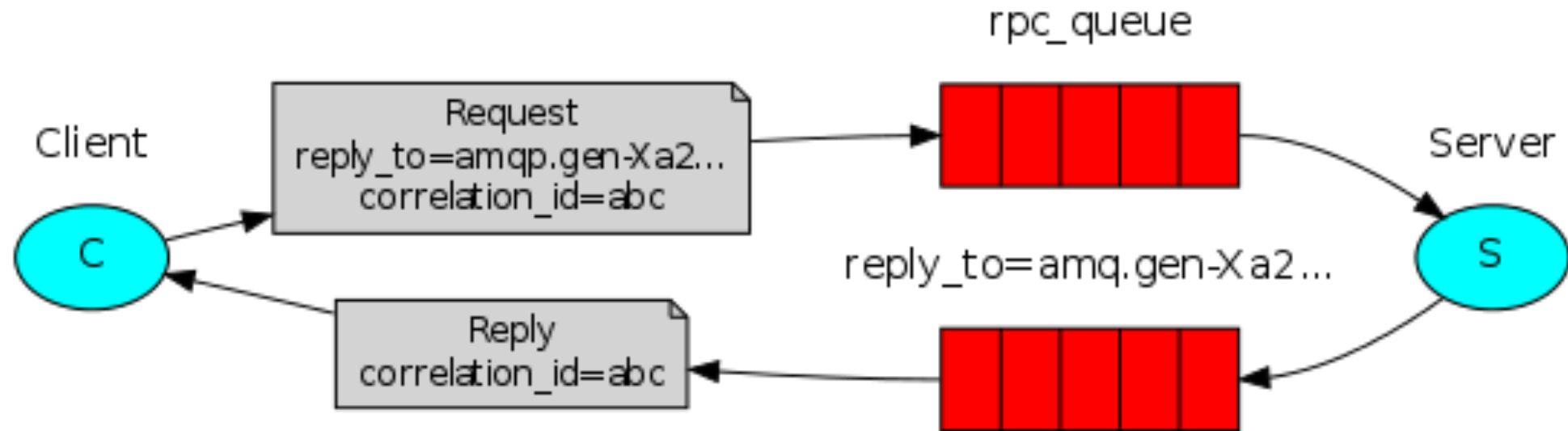
- Publisher (P) sends a message to the broker, which puts it in queue
- The broker routes to the appropriate Consumer (C) using routing keys
- Messages are *pushed* to the consumer
- The publisher doesn't explicitly know how to connect to the consumer.
  - The broker handles this

# Work Queue



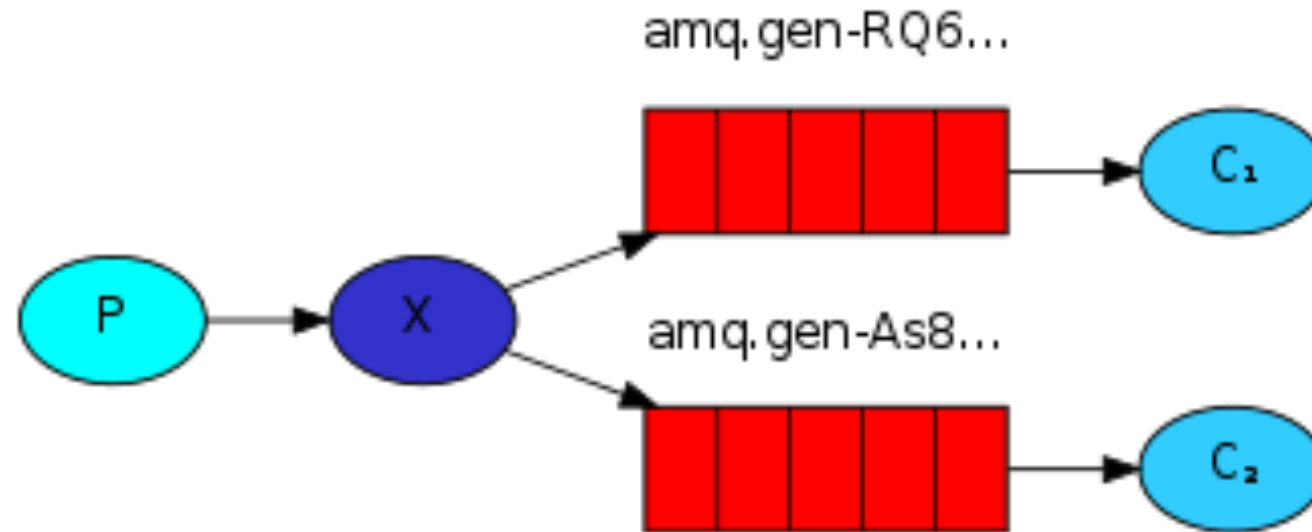
- Multiple consumers listen to the same queue
- Each message is delivered to only one consumer
- The broker can use round robin or other scheduling strategies for routing messages to individual consumers
- You can program/configure consumers to ACK messages and block until they are finished processing a message
- If a consumer crashes, the broker can resend a message
- Consumers can join or leave the pool dynamically

# Remote Procedure Calls with Messaging



- We can make more sophisticated Work Queues by adding a second queue (callback or reply) to the broker.
- This allows the server to send back complicated replies to the client or to another recipient
  - Not just ACKs to the broker
- Now both end points act as both publishers and consumers.

# Publish/Subscribe



- Publisher sends a message to multiple queues (same broker) via an Exchange (X)
- Consumers attach to different queues
- This allows multiple types of consumers to receive the same message
- Important Variation: Topic-Based Pub/Sub
- When should you use this?



Some Applications

# Simple Microservice Work Queue with RabbitMQ



Publisher (API Server): pushes a request for work into the queue



Message Queue should implement “only deliver message once to one consumer”, round-robin



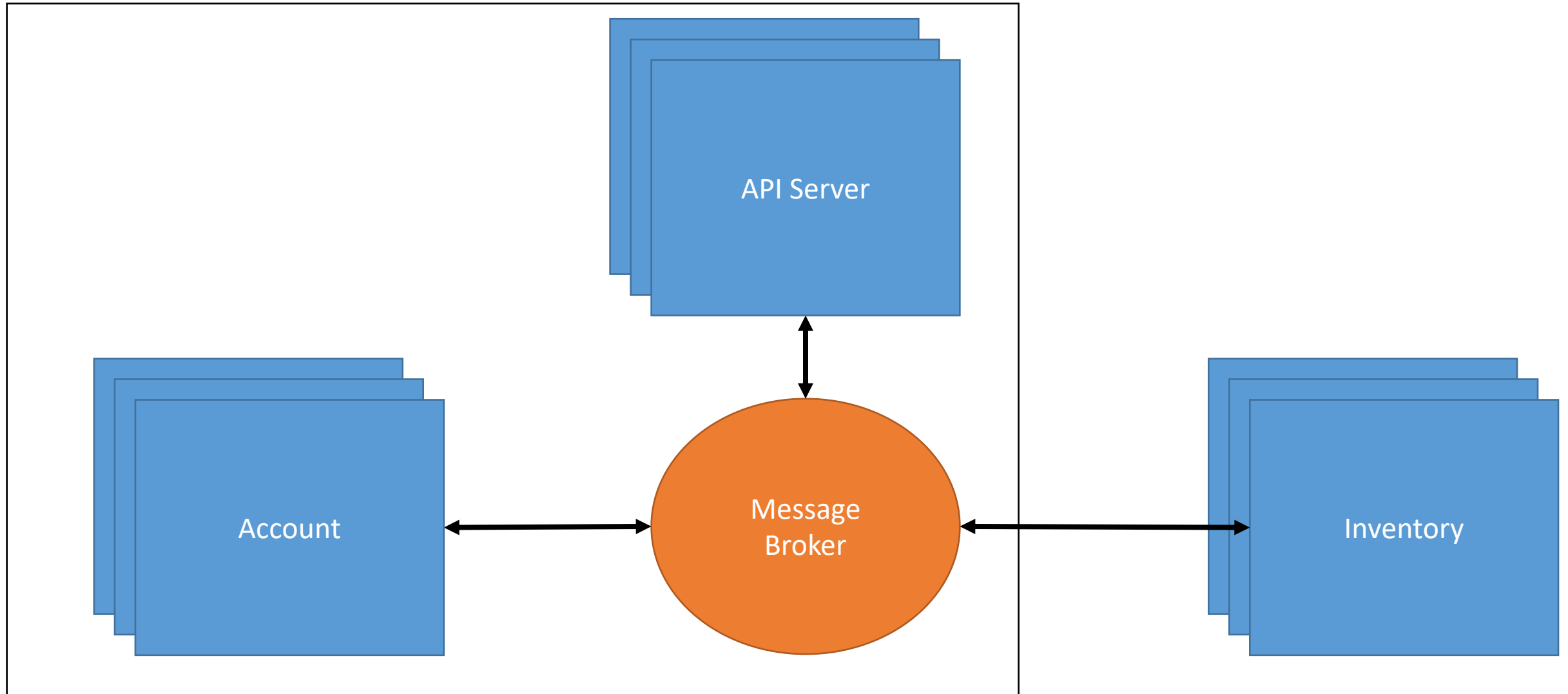
Consumer: Sends an ACK after completing the task.



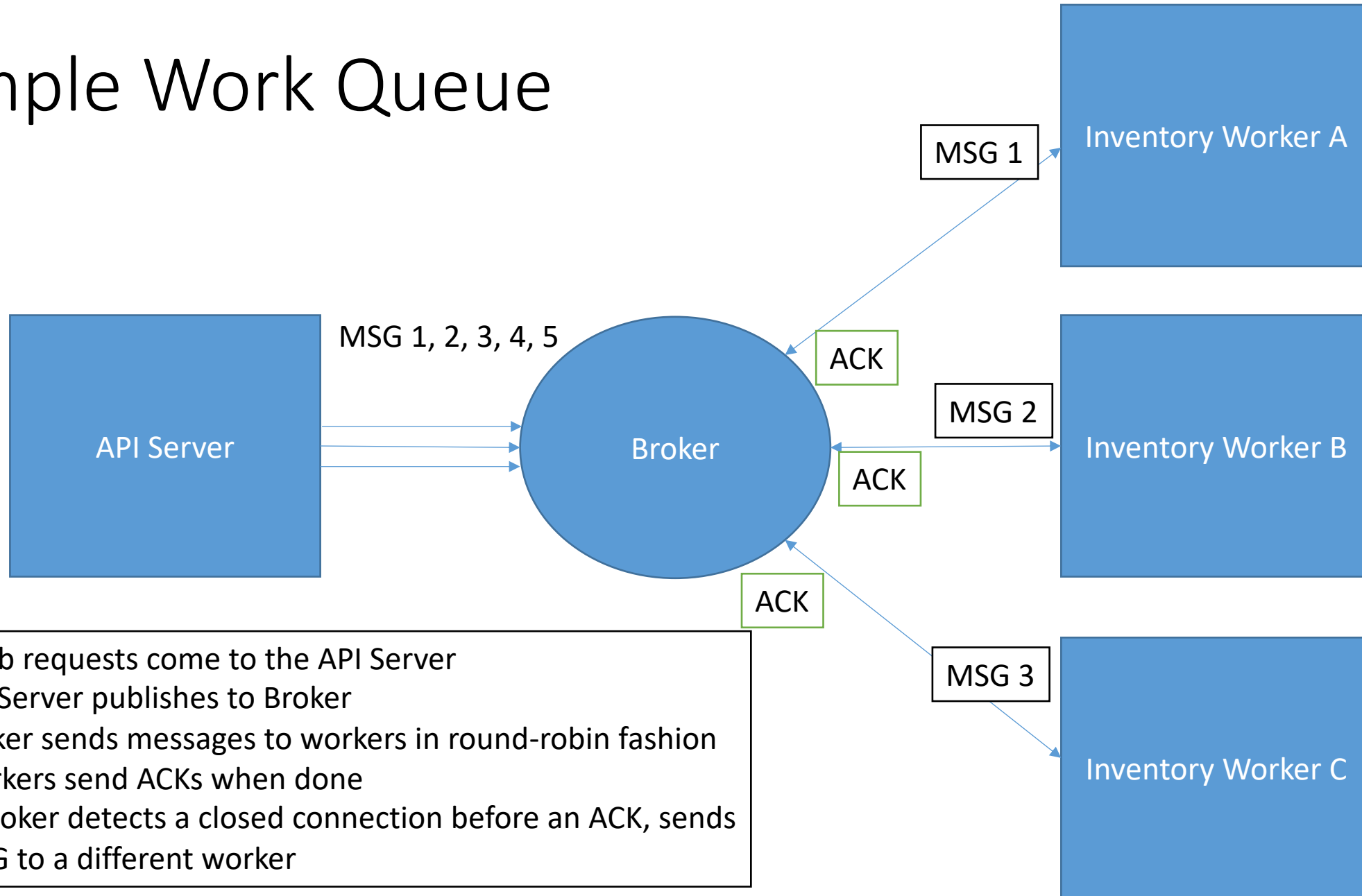
If a Consumer closes before it send the ACK, the broker resends the message to a new consumer.



# Let's Replicate the Account Manager

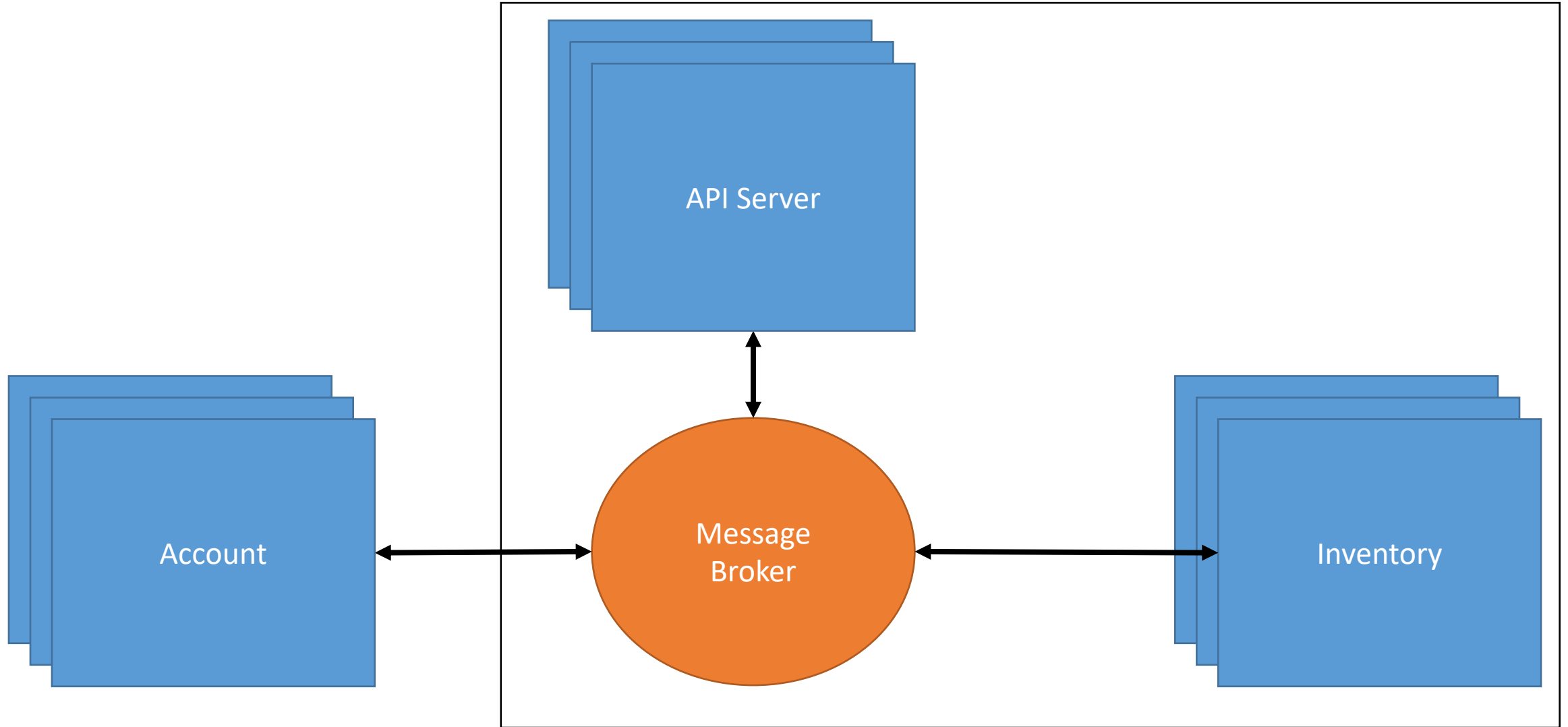


# Simple Work Queue



- 5 Job requests come to the API Server
- API Server publishes to Broker
- Broker sends messages to workers in round-robin fashion
- Workers send ACKs when done
- If Broker detects a closed connection before an ACK, sends MSG to a different worker

# What About the Next Step in the Transaction?



# Coordinating Transactions: Some Choices

You can hard-code it all in the API server

If using a message broker, you can use an RPC pattern to signal to the next service

You can use a transaction manager as a separate service

You can define your process flows in a workflow language that is used by the API Server or Transaction Manager

# Coordinating Transactions: Some Microservice Patterns for Further Reading

SAGA

Event Sourcing

Command Query Responsibility Segregation (CQRS)

# Summary

Message systems route communications between distributed entities.

- You don't need to know the physical addresses of the recipients

Messaging systems support multiple messaging patterns out of the box.

- You don't have to implement them.

Queues are a powerful concept within distributed systems.

- Entities can save messages in order and deliver/accept them at a desirable rate.
- Queues are a “primitive” (foundational) concept that you can use to build more sophisticated systems.

# Which Messaging Software to Choose?

## You have many choices

- RabbitMQ, Kafka, NATS, Apache Pulsar, ZeroMQ...
- This is a team issue: consider options, make a choice, and go with it.

## AMQP messaging system implementations are not cloud-native

- They must be configured as highly available services.
  - Primary + failover
- No fancy leader elections, etc as used in Zookeeper + Zab or Apache Kafka
- Have scaling limitations, although these may not matter at our scales.

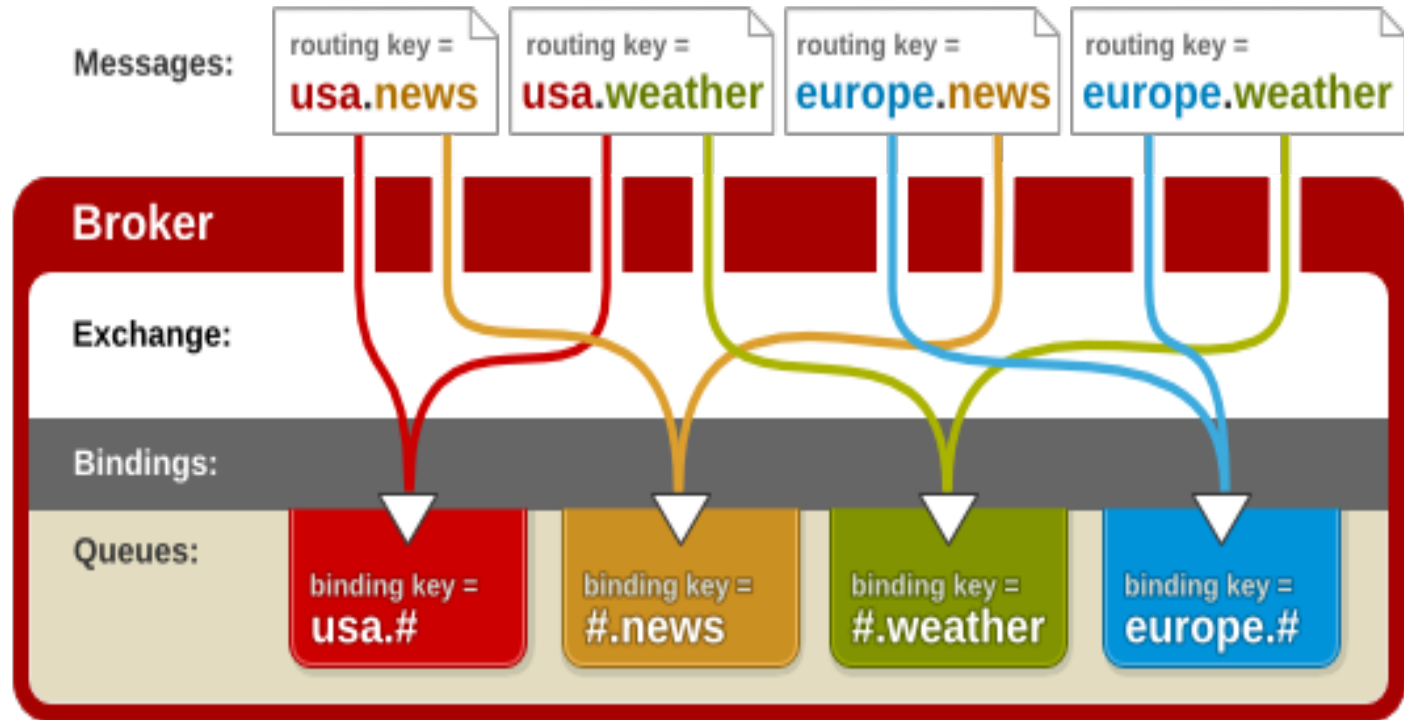




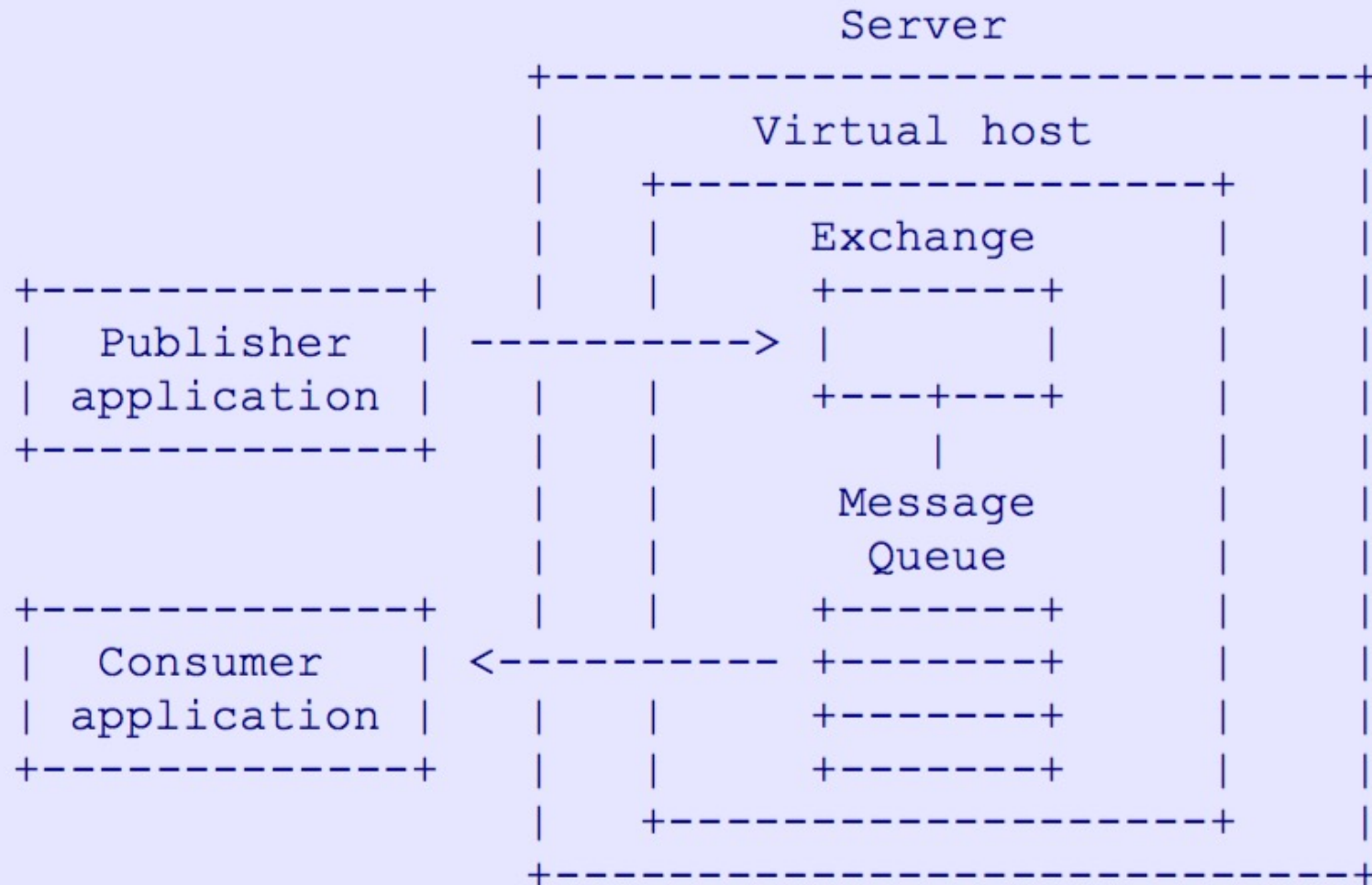
# Topic Exchange

- Message Queues bind using *routing patterns* instead of routing keys.
- A Publisher sends a message with a routing key.
- Exchange will route to all Message Queues that match the routing key's pattern

## Topic Exchange



# Basic Concepts



# An AMQP Server (or Broker)

## Exchange

- Accepts producer messages
- Sends to 0 or more Message Queues using routing keys

## Message Queue

- Routes messages to different consumers depending on arbitrary criteria
- Buffers messages when consumers are not able to accept them fast enough.

# Producers and Consumers

- Producers only interact with Exchanges
- Consumers interact with Message Queues
- Consumers aren't passive
  - Can create and destroy message queues
- The same application can act as both a publisher and a consumer
  - You can implement Request-Response with AMQP
  - Except the publisher doesn't block
- Ex: your application may want an ACK or NACK when it publishes
  - This is a reply queue

# The Exchange

- Receives messages
- Inspects a message header, body, and properties
- Routes messages to appropriate message queues
- Routing usually done with **routing keys** in the message payload
  - For point-to-point messages, the routing key is the name of the message queue
  - For pub-sub routing, the routing key is the name of the topic
    - Topics can be hierarchical

# Message Queue Properties and Examples

- Basic queue properties:
  - Private or shared
  - Durable or temporary
  - Client-named or server-named, etc.
- Combine these to make all kinds of queues, such as
- **Store-and-forward queue:** holds messages and distributes these between consumers on a round-robin basis.
  - Durable and shared between multiple consumers.
- **Private reply queue:** holds messages and forwards these to a single consumer.
  - Reply queues are typically temporary, server-named, and private to one consumer.
- **Private subscription queue:** holds messages collected from various "subscribed" sources, and forwards these to a single consumer.
  - Temporary, server-named, and private

# Consumers and Message Queues

- AMQP Consumers can create their own queues and bind them to Exchanges
- Queues can have more than one attached consumer
- AMQP queues are FIFO
  - AMQP allows only one consumer per queue to receive the message.
  - Use round-robin delivery if  $> 1$  attached consumer.
- If you need  $> 1$  consumer to receive a message, you can give each consumer their own queue.
  - Each Queue can attach to the same Exchange, or you can use topic matching.

# Publish-Subscribe Patterns

- Useful for many-to-many messaging
- In microservice-based systems, several different types of components may want to receive the same message
  - But take different actions
  - Ex: you can always add a logger service
- You can always do this with explicitly named routing keys.
- You may also want to use hierarchical (name space) key names and pattern matching.
  - gateway.jobs.jobtype.gromacs
  - gateway.jobs.jobtype.\*



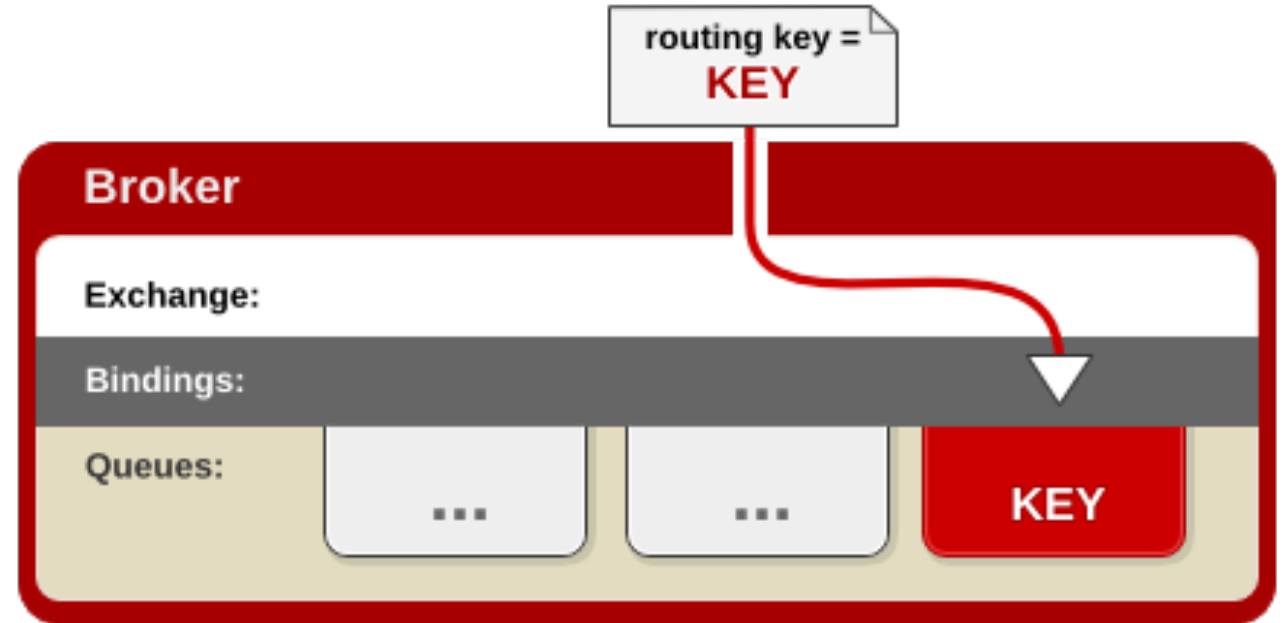
# The Message Payload

- Read the specification for more details.
- In general AMQP follows the header-body format
- The message body payload is binary
- AMQP assumes the body content is handled by consumers
  - The message body is opaque to the AMQP server.
- You could serialize your content with JSON or Thrift and deserialize it to directly send objects.

# Direct Exchange

- A publisher sends a message to an exchange with a specific routing key.
- The exchange routes this to the message queue bound to the routing key.
- A consumer receives the messages if listening to the queue.
- Default: round-robin queuing to deliver to multiple subscribers of same queue

## Direct Exchange

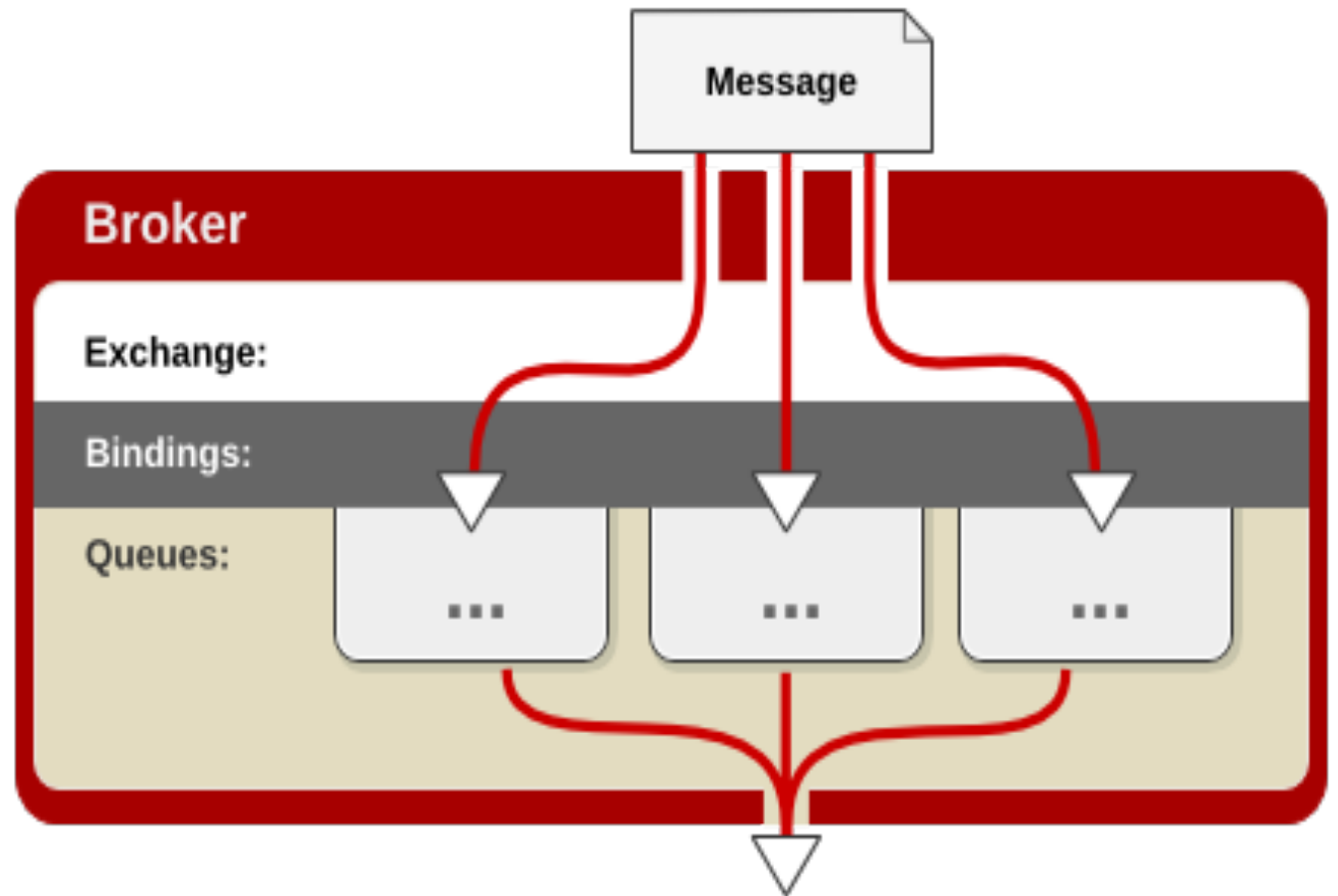


```
Queue.Declare queue=app.svc01
Basic.Consume queue=app.svc01
Basic.Publish routing-
key=app.svc01
```

# Fanout Exchange

- Message Queue binds to an Exchange with no argument
- Publisher sends a message to the Exchange
- The Exchange sends the message to the Message Queue
- All consumers listening to all Message Queues associated with an Exchange get the message

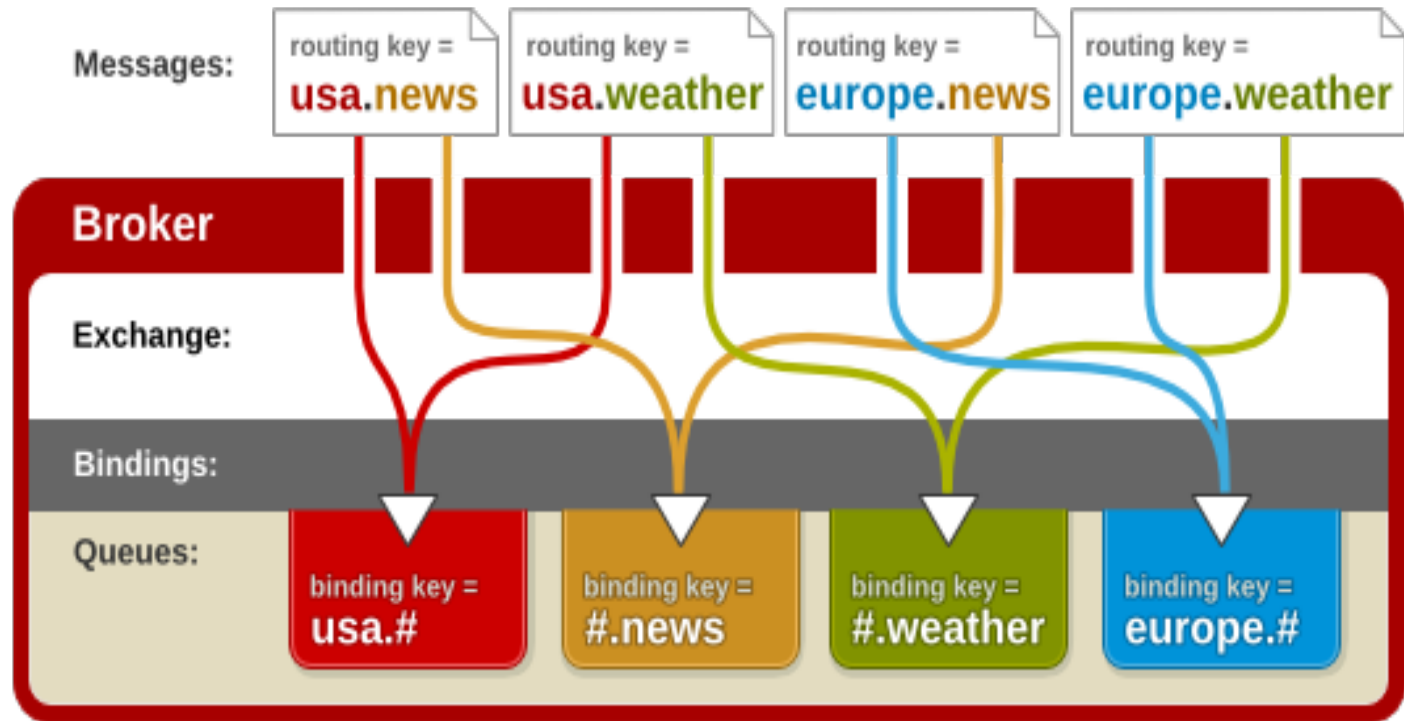
## Fanout Exchange



# Topic Exchange

- Message Queues bind using *routing patterns* instead of routing keys.
- A Publisher sends a message with a routing key.
- Exchange will route to all Message Queues that match the routing key's pattern

## Topic Exchange



# More Examples

## RabbitMQ Tutorial

- Has several nice examples of classic message exchange patterns.
- <https://www.rabbitmq.com/getstarted.html>

## What It Omits

- Many publishers
- Absolute and partial event ordering are hard problems
- Broker failure and recovery

# Work Queue, Take Two

- Orchestrator pushes work into a queue.
- Have workers request work when they are not busy.
  - RabbitMQ supports this as “prefetchCount”
  - Use round-robin but don’t send work to busy workers with outstanding ACKs.
  - Workers do not receive work requests when they are busy.
  - Messages wait in queue...
- Worker sends ACK after successfully submitting the job to an external resource.
  - This only means the job has been submitted
  - Worker can take more work
- A Monitor application handles the state changes on the supercomputer
  - Publishes “queued”, “executing”, “completed” or “failed” messages
- When job is done, Orchestrator creates a “cleanup” job
- Any worker available can take this.

# Work Queue, Take 2

