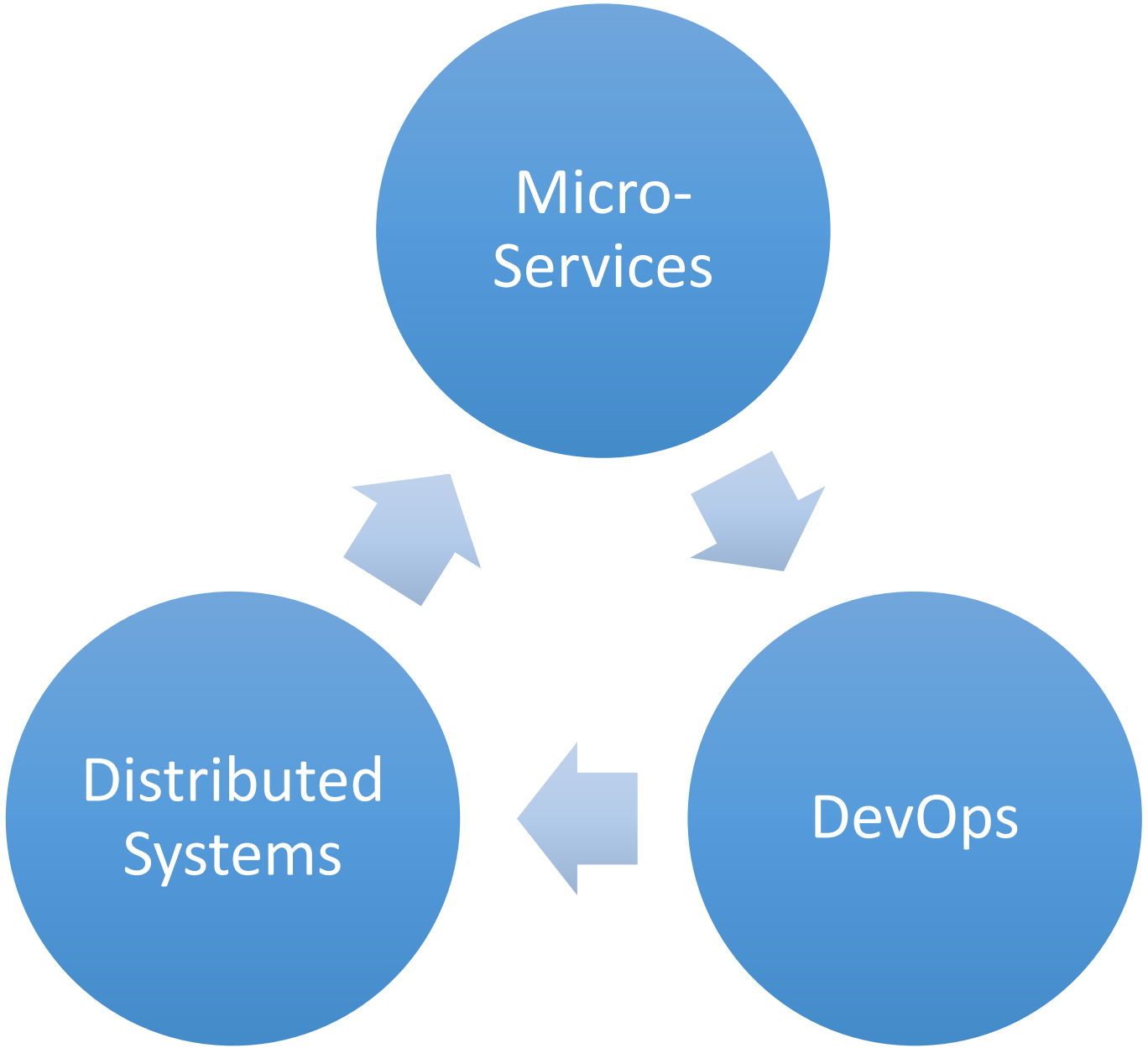
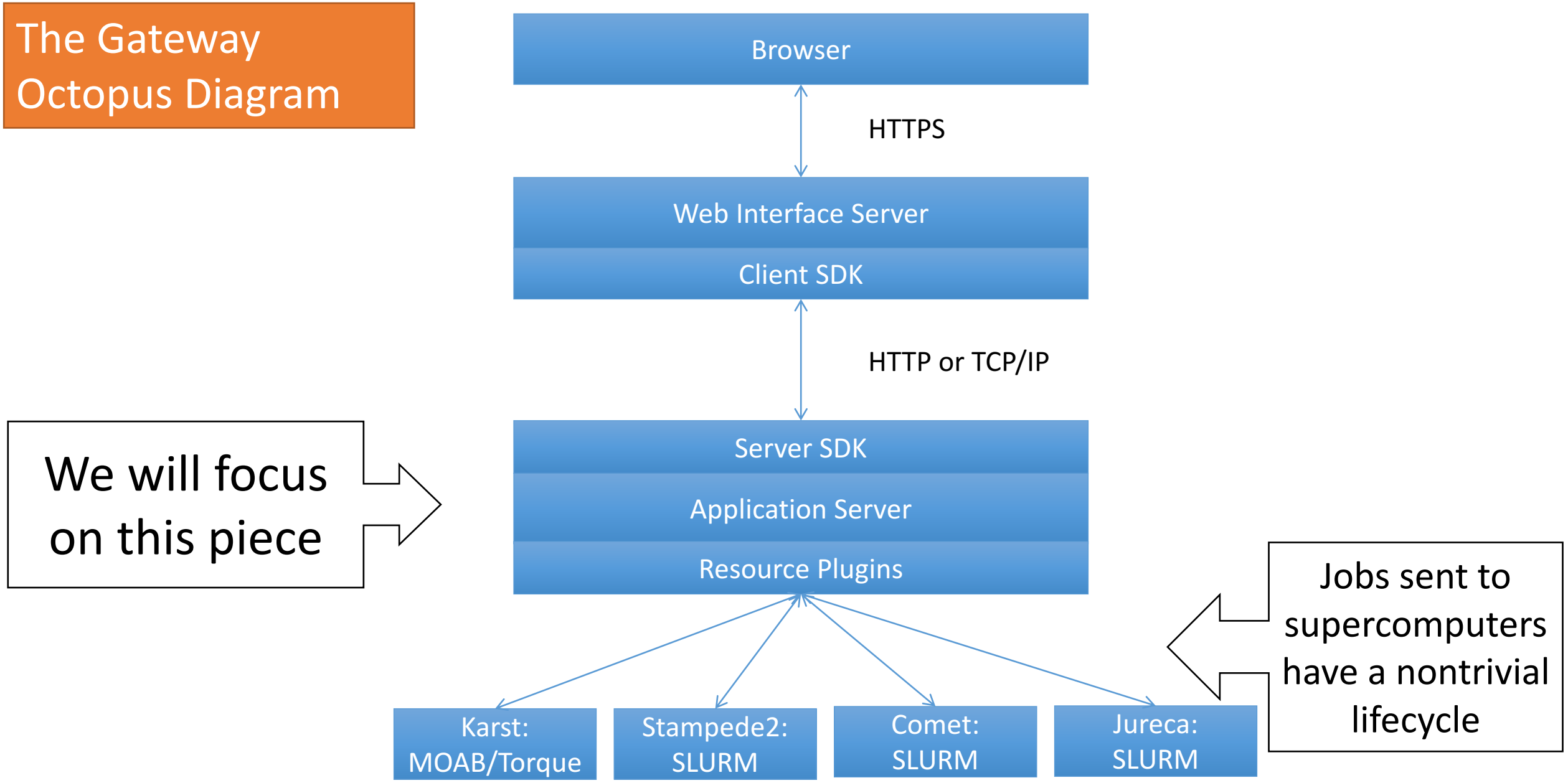
Microservices, Messaging and Science Gateways

Review microservices for science gateways and then discuss messaging systems.


Micro-
Services
DevOps
Distributed
Systems

The Gateway Octopus Diagram

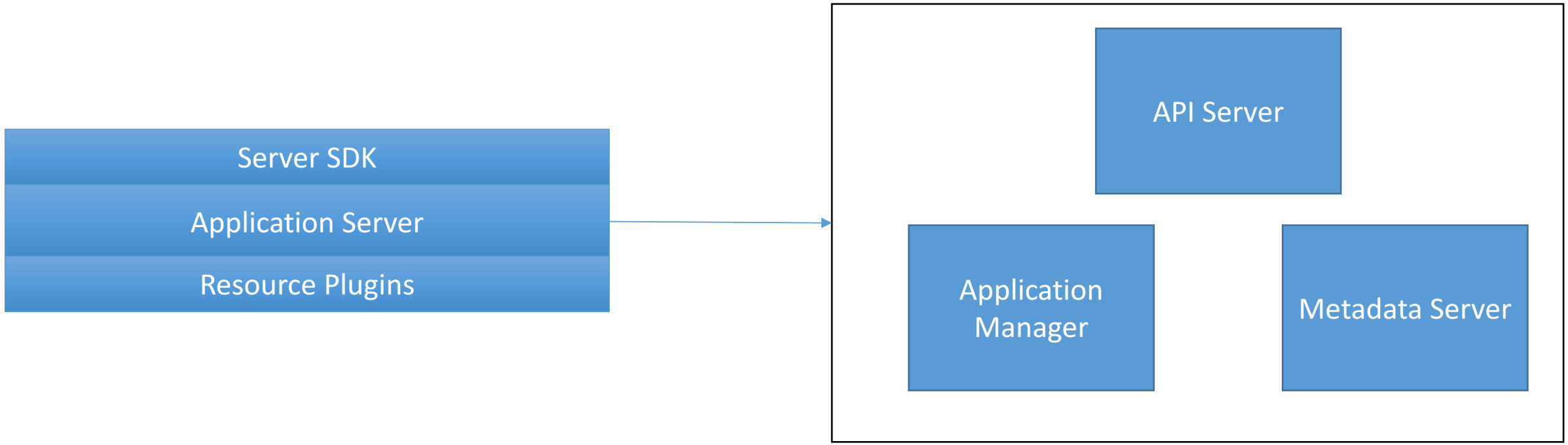


State in Science Gateways

- Scientific applications running on supercomputers have lifecycles
  - Queued
  - Executing
  - Completed
- Gateways may add additional states
  - Created (but not yet queued)
  - Archived
- The lifecycle of an executing job may be several minutes, hours, days, or even longer
  - Analogous to ordering a physical object delivered via Amazon.
- State management is an important concept in gateways.
  - It guides component (microservice) design and implementation
  - It governs messaging patterns

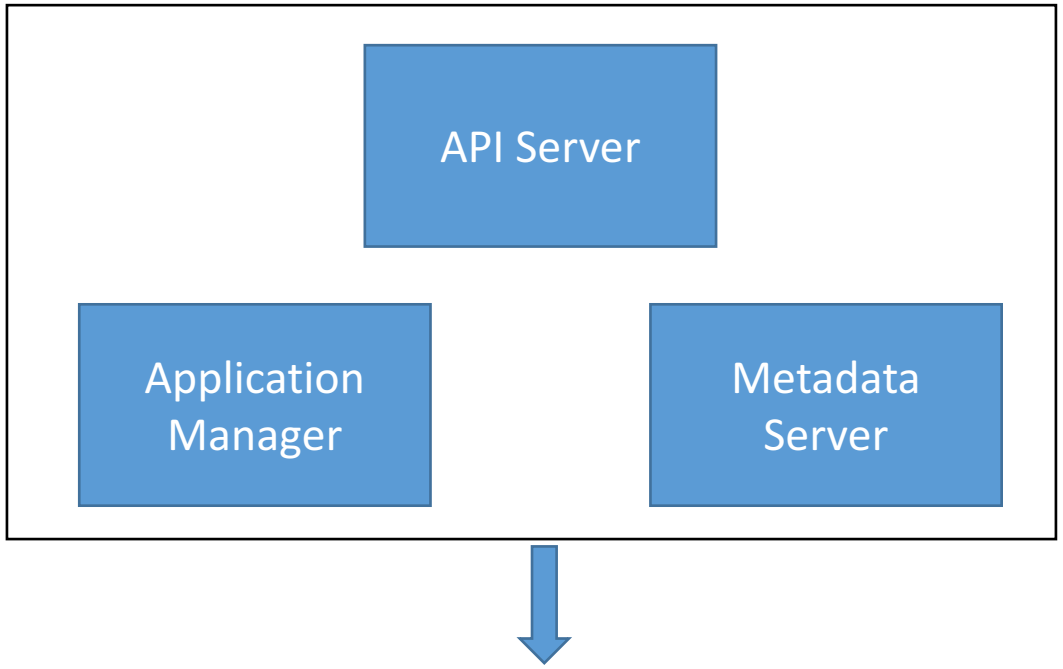
Simple Blocking Request-Response Mechanisms Are Not Adequate

Basic Components of the Gateway App Server Become Microservices



For now, we will not depict the communication connections between the microservices on the left. These are over-the-wire and need to be non-blocking in many cases.

Replicate the Microservices

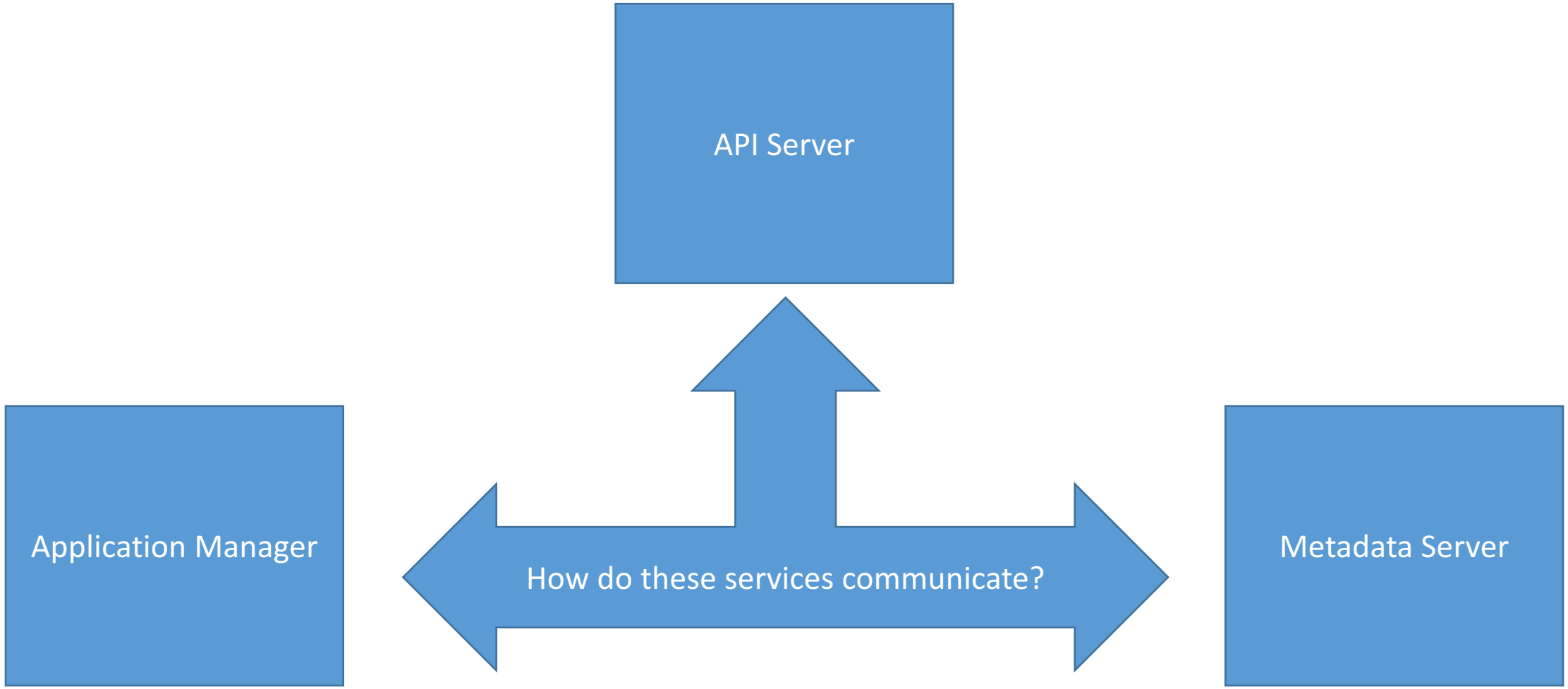


Communication patterns and system state are important

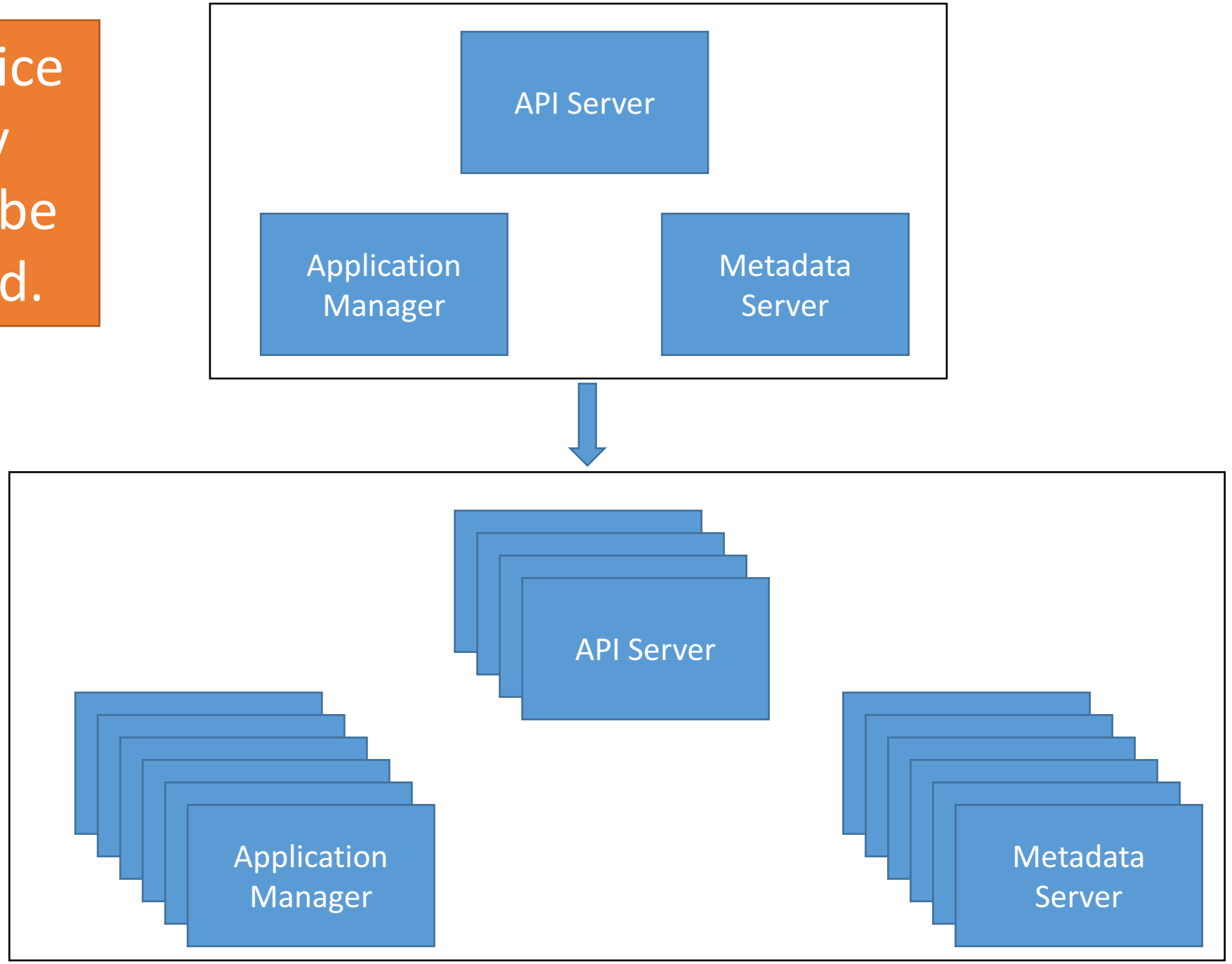
Messaging in Distributed Systems

Examining messaging in distributed systems though Advanced Message Queuing Protocol (AMQP) and RabbitMQ overviews

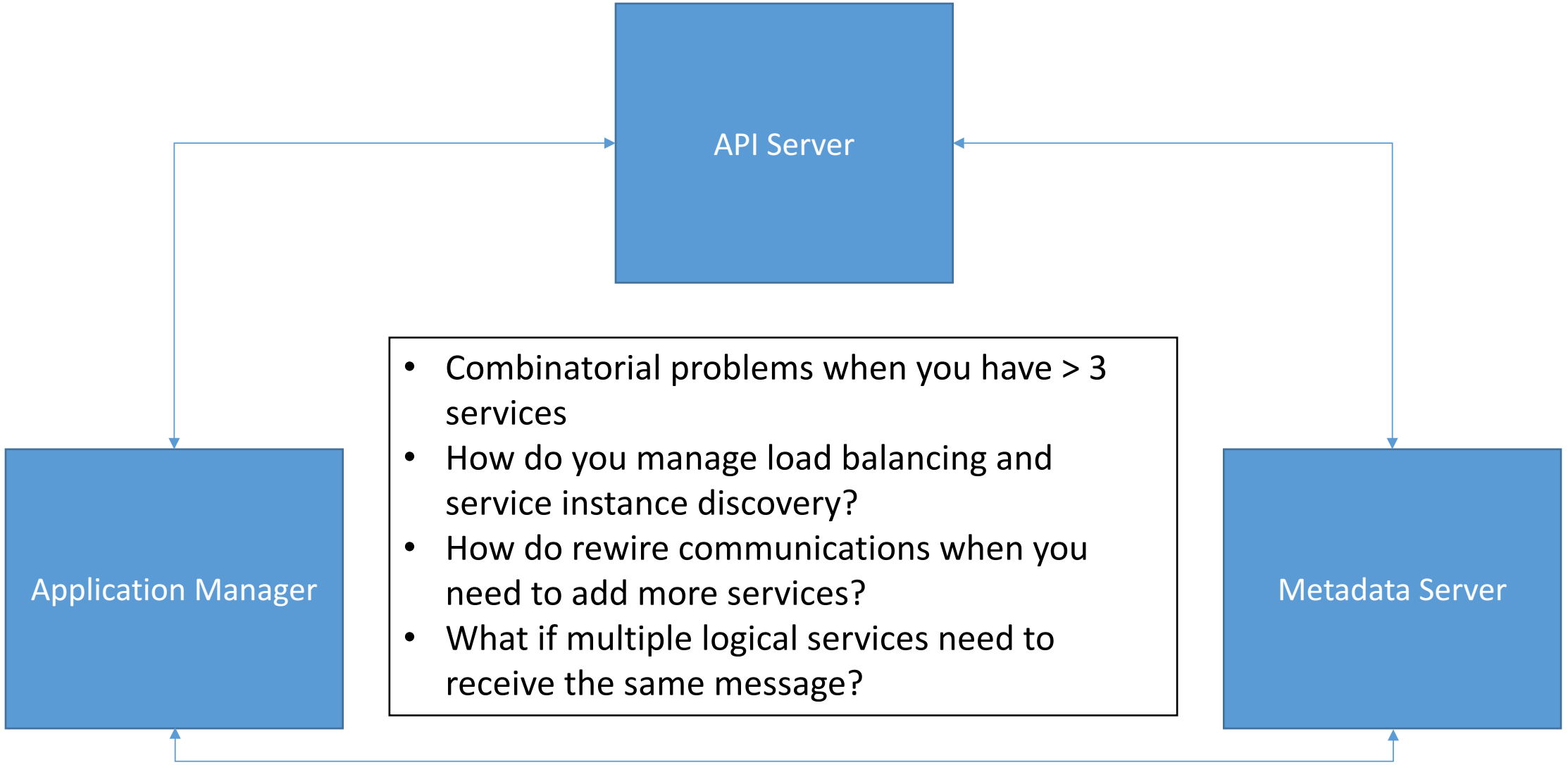
Basic Components of the Gateway App Server



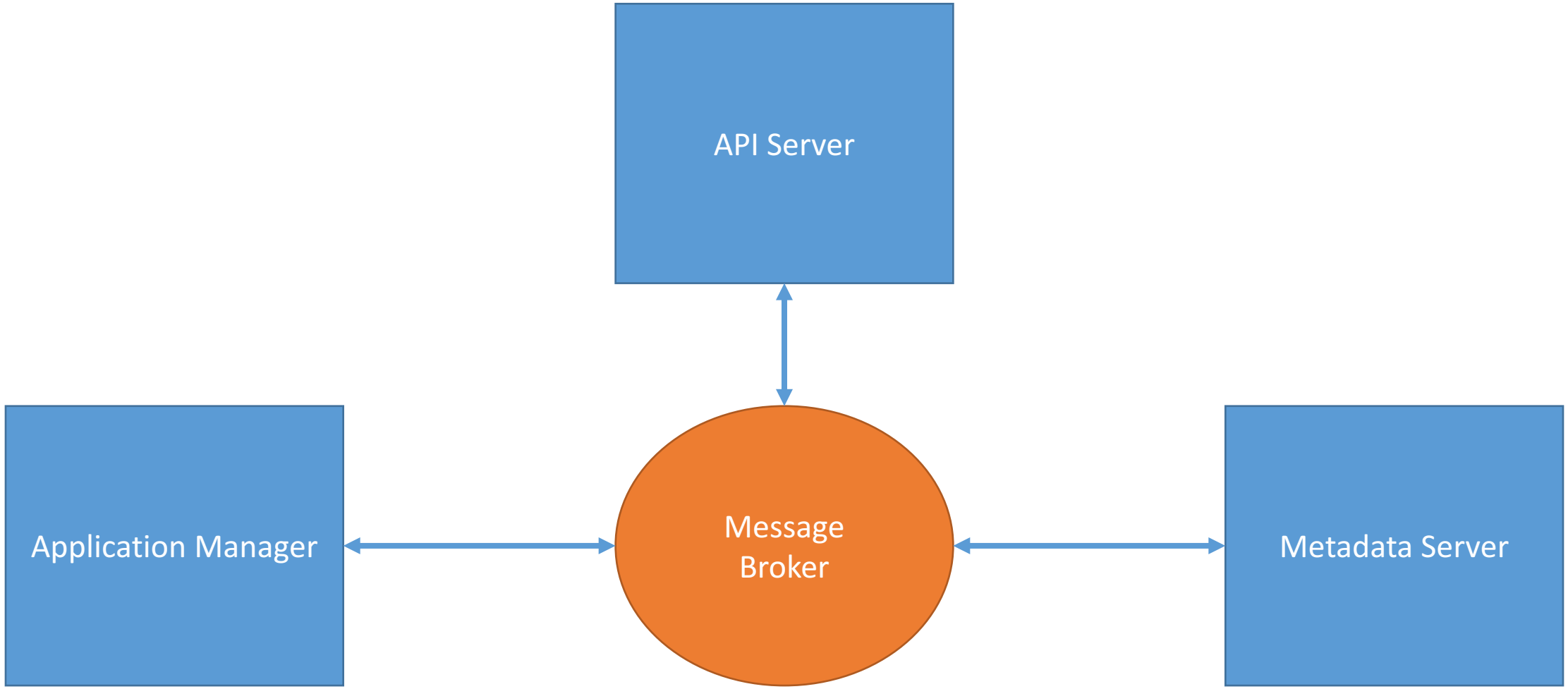
Each service actually needs to be replicated.



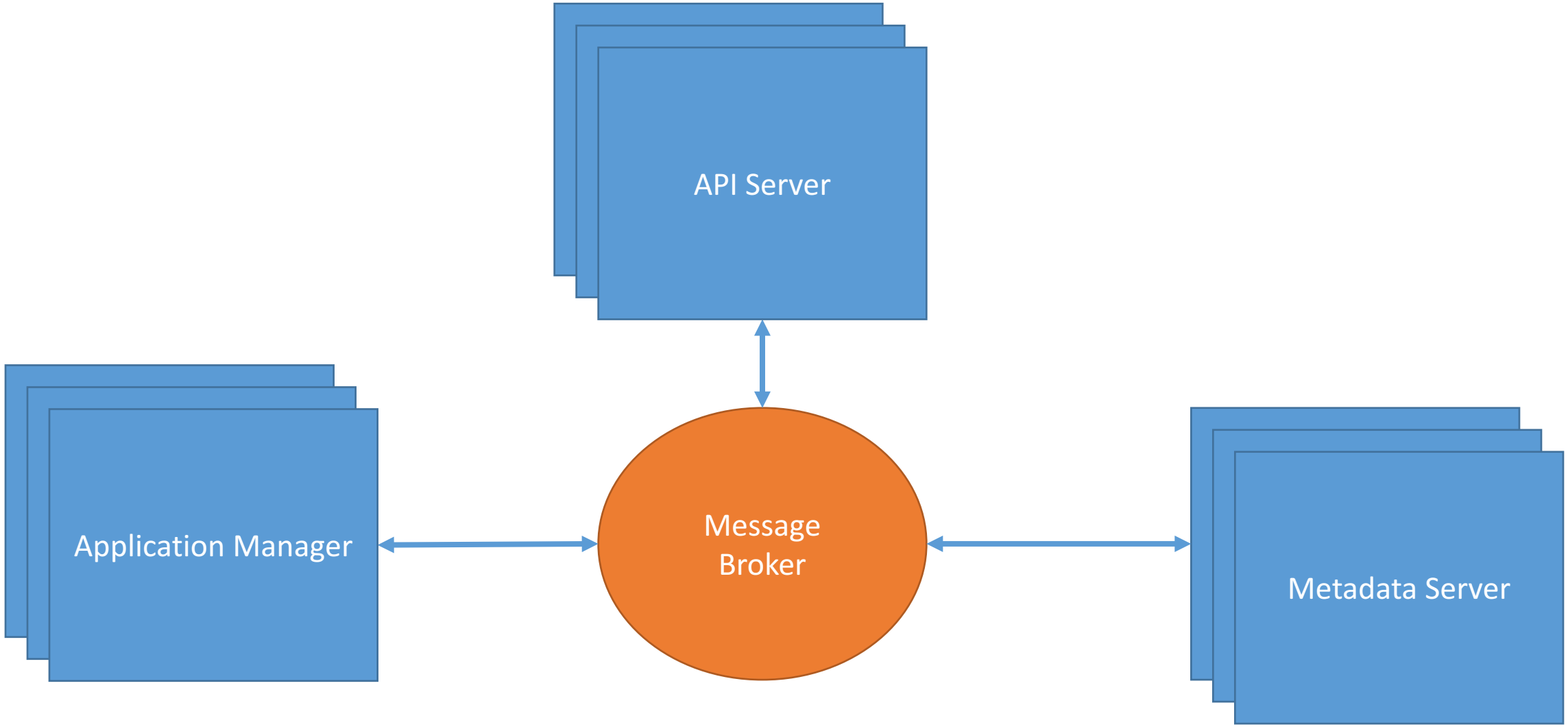
Point to Point Communication Is Brittle



Messaging Systems Solve Many of These Problems



They Work Well with Service Replication



Messaging Summary

- Review RabbitMQ's wonderful set of tutorials for several programming languages.
  - https://www.rabbitmq.com/getstarted.html
- RabbitMQ implements some basic AMQP patterns out of the box.
  - You don't need to understand the full power and flexibility of AMQP-based systems to use messaging.
- We'll introduce some of the main concepts

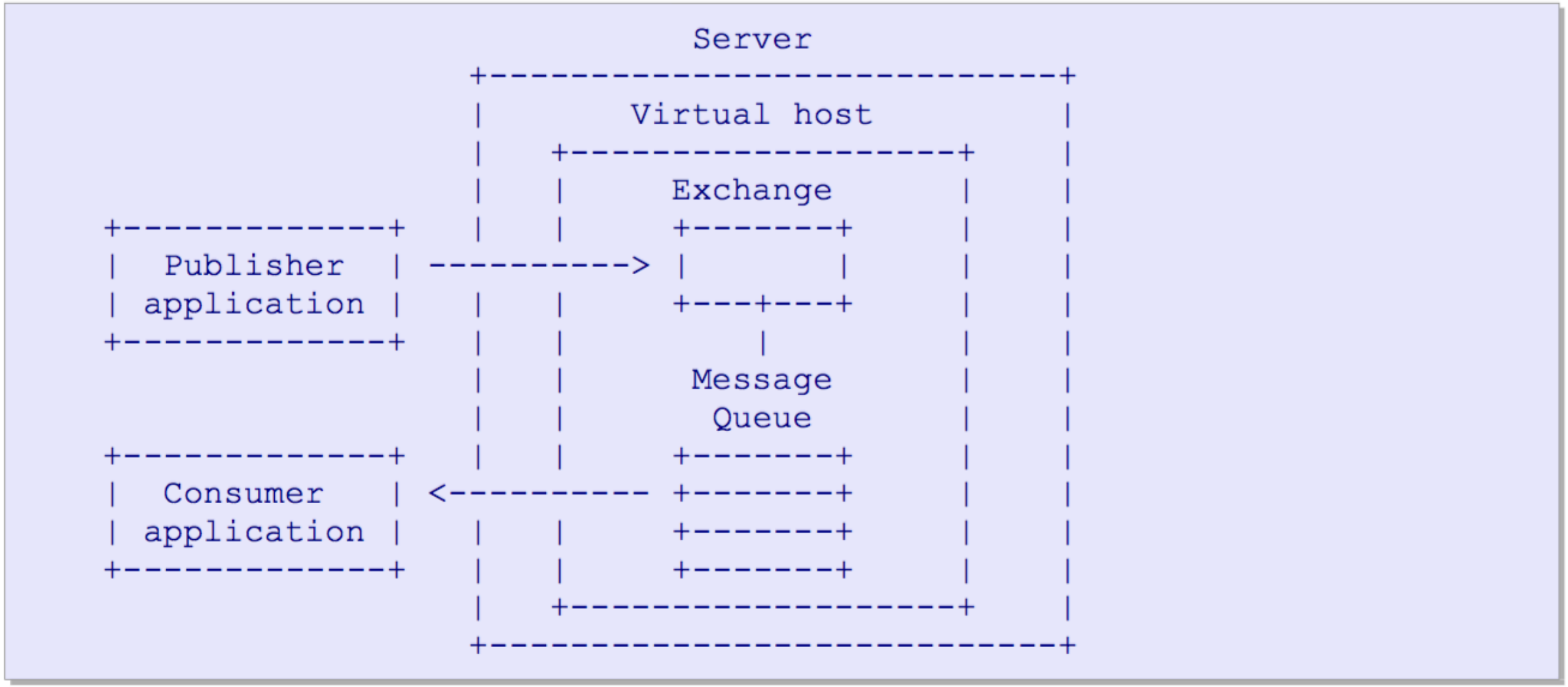
AMQP: Network Protocol and Architecture, Not An API

- "Programmable" by endpoints
  - Can create and manage their own queues, exchanges, etc.
  - The broker manager does not need to configure these things.
- Many Implementations
  - RabbitMQ, Apache ActiveMQ, Apache Qpid, SwiftMQ, ...
  - I'll focus on Version 0-9-1 and RabbitMQ
- RabbitMQ and Apache Zookeeper
  - Similar philosophy: provide "primitive" operations that can be combined to support more complicated scenarios.
- This is not an AMQP tutorial

Value of Message Queuing Systems, Generally

- They are queues for messages....
- Even if you are doing point-to-point routing, messaging systems remove the need for publishers and consumers to know each other's IP addresses.
  - Publishers and consumers just need to know how to connect to the message broker.
  - The network locations and specific instances of the publishers and consumers can change over time.
  - But logical instances persist
- Synchronous and asynchronous messages natively supported.
- Multiplex multiple channels of communication over a single TCP/IP connection

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.*

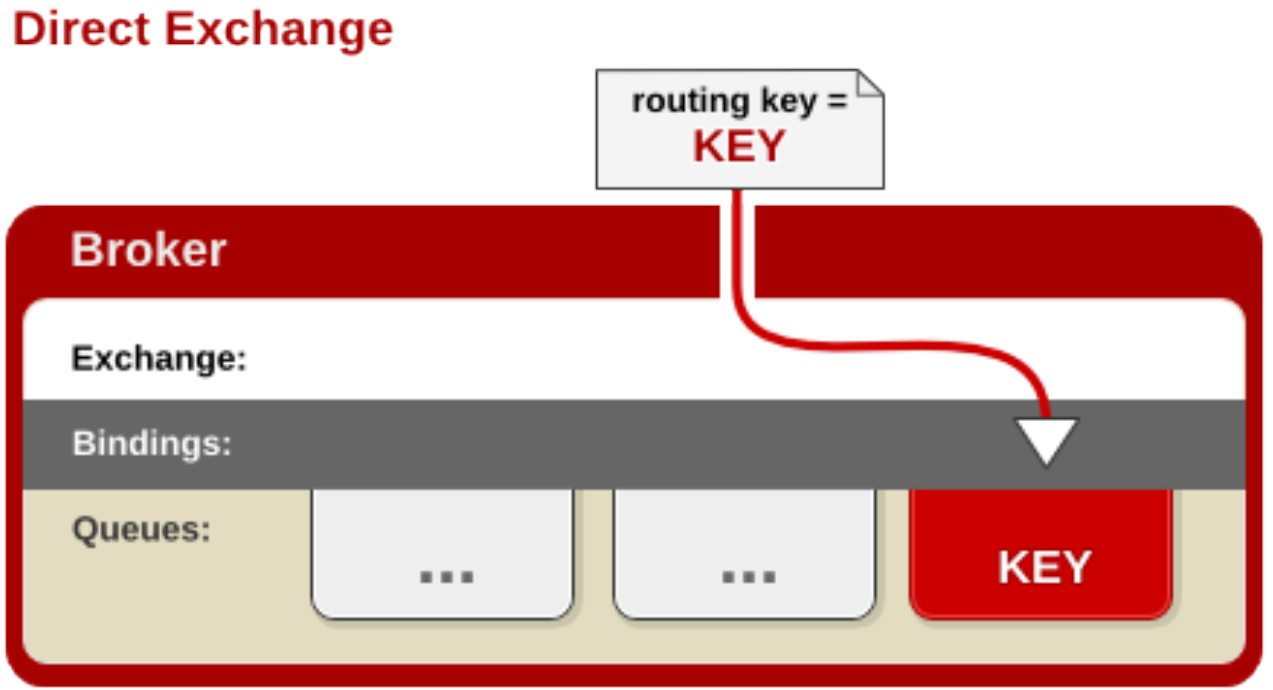
The Message Payload

- 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.

Message Exchange Patterns

Direct Exchange

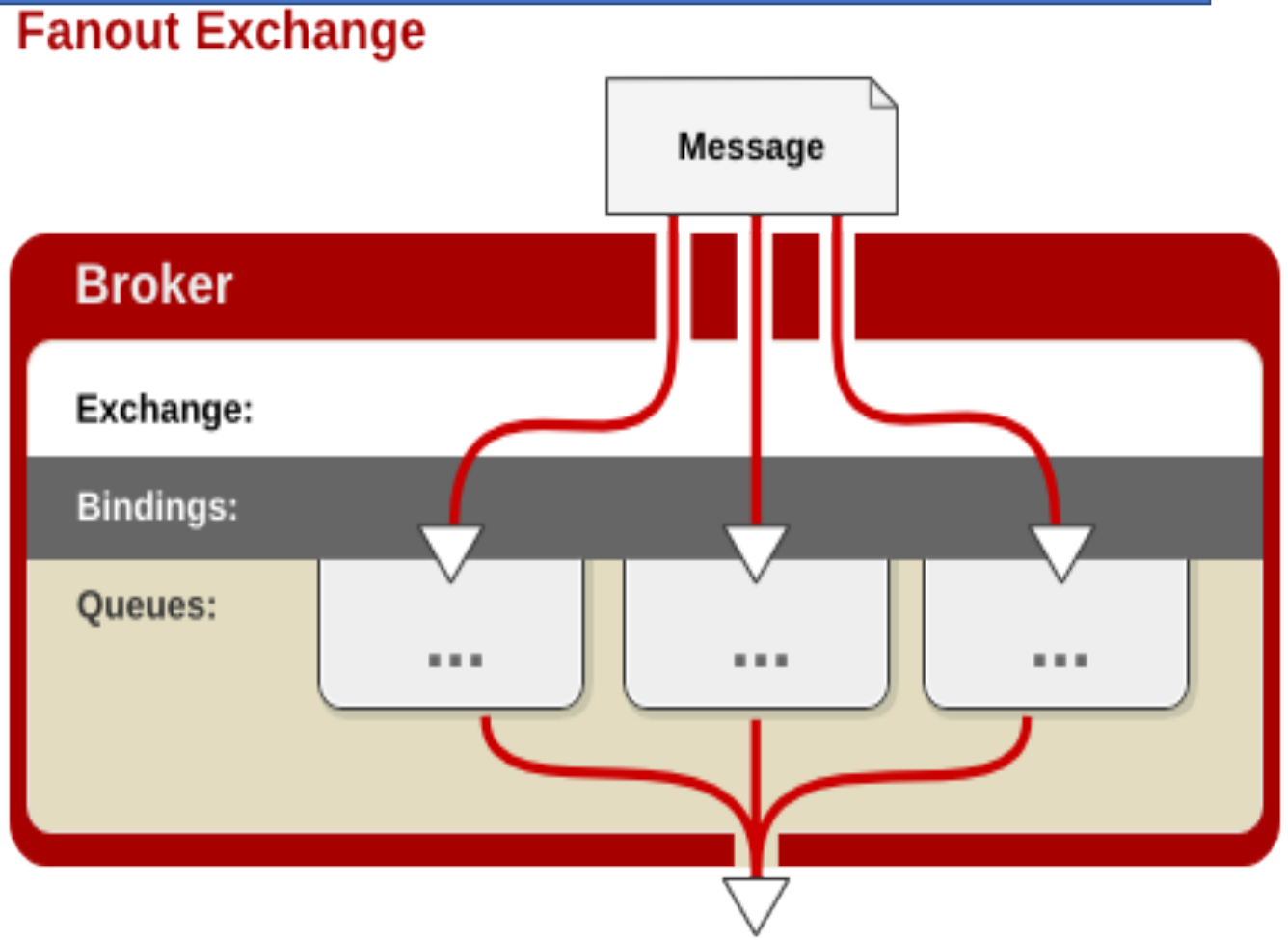
- A publisher sends a message to an exchange with a specific routing key.
- The exchange routes this to the message queue bound to the routing key.
- A consumer receives the messages if listening to the queue.
- Default: round-robin queuing to deliver to multiple subscribers of same queue




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

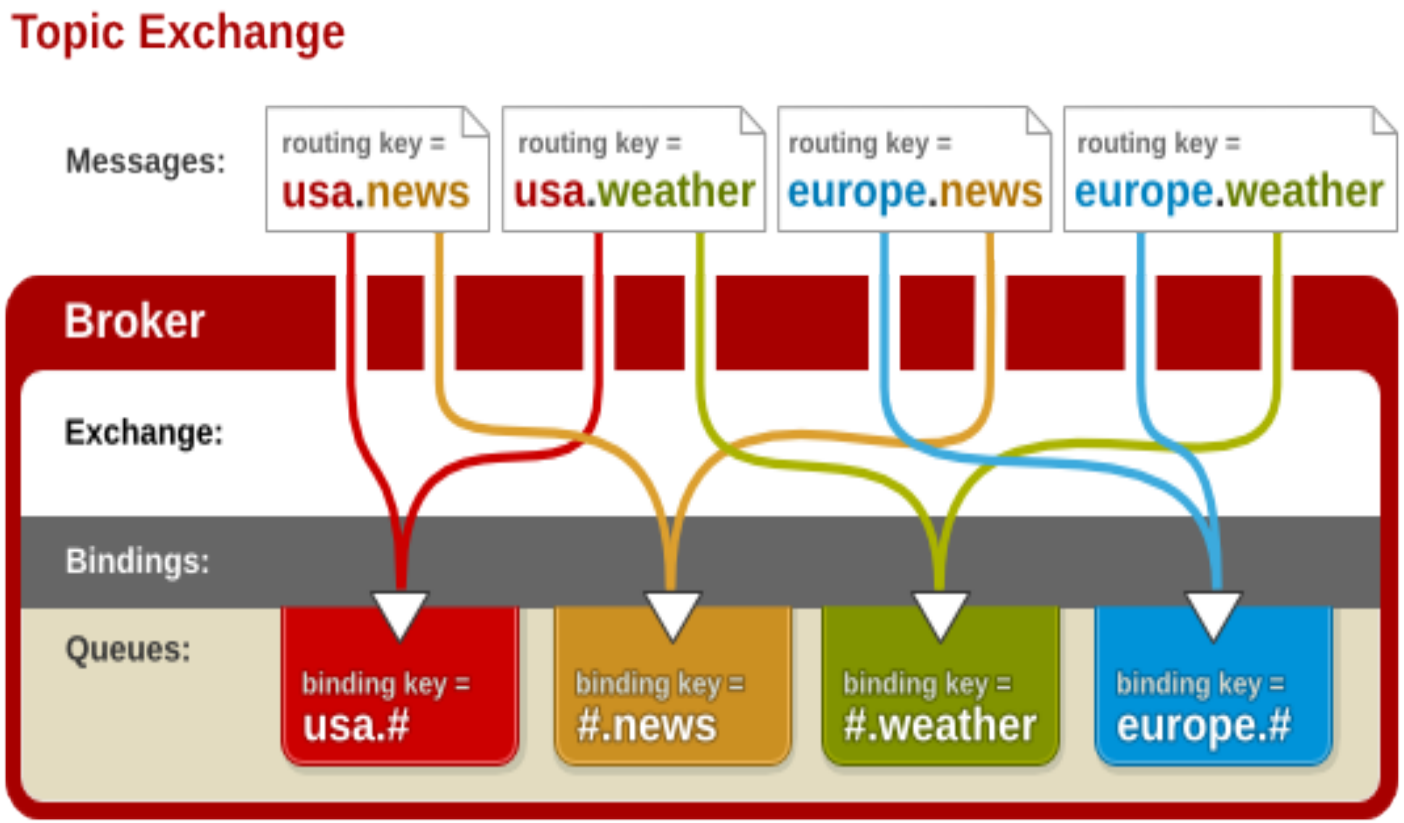
Fanout Exchange

- 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



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



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

Some Useful Capabilities of Messaging Systems for Microservices

Overarching Requirement: It should support your system's distributed state machine

Let's brainstorm some

Useful Capabilities: My List (1/2)

- Supports both push and pull messaging
- Deliver messages in order
- Successfully delivered messages are delivered exactly once
  - OK to have multiple recipients
- Deliver messages to one or more listeners based on pre-defined topics.
- Store messages persistently
  - There are no active recipients.
  - All recipients are busy
- Determine if critical messages were delivered correctly

Useful Capabilities: My List (2/2)

- Redeliver messages that weren't correctly received
  - Corrupted messages, no recipients, etc
  - Recipient can change
- Redeliver messages on request
  - Helps clients resynch their states
- Allow other components to inspect message delivery metadata.
  - Supports elasticity, fault tolerance
- Priority messaging?
- Qualities of Service
  - Security, fault tolerance

Which Messaging Software to Choose?

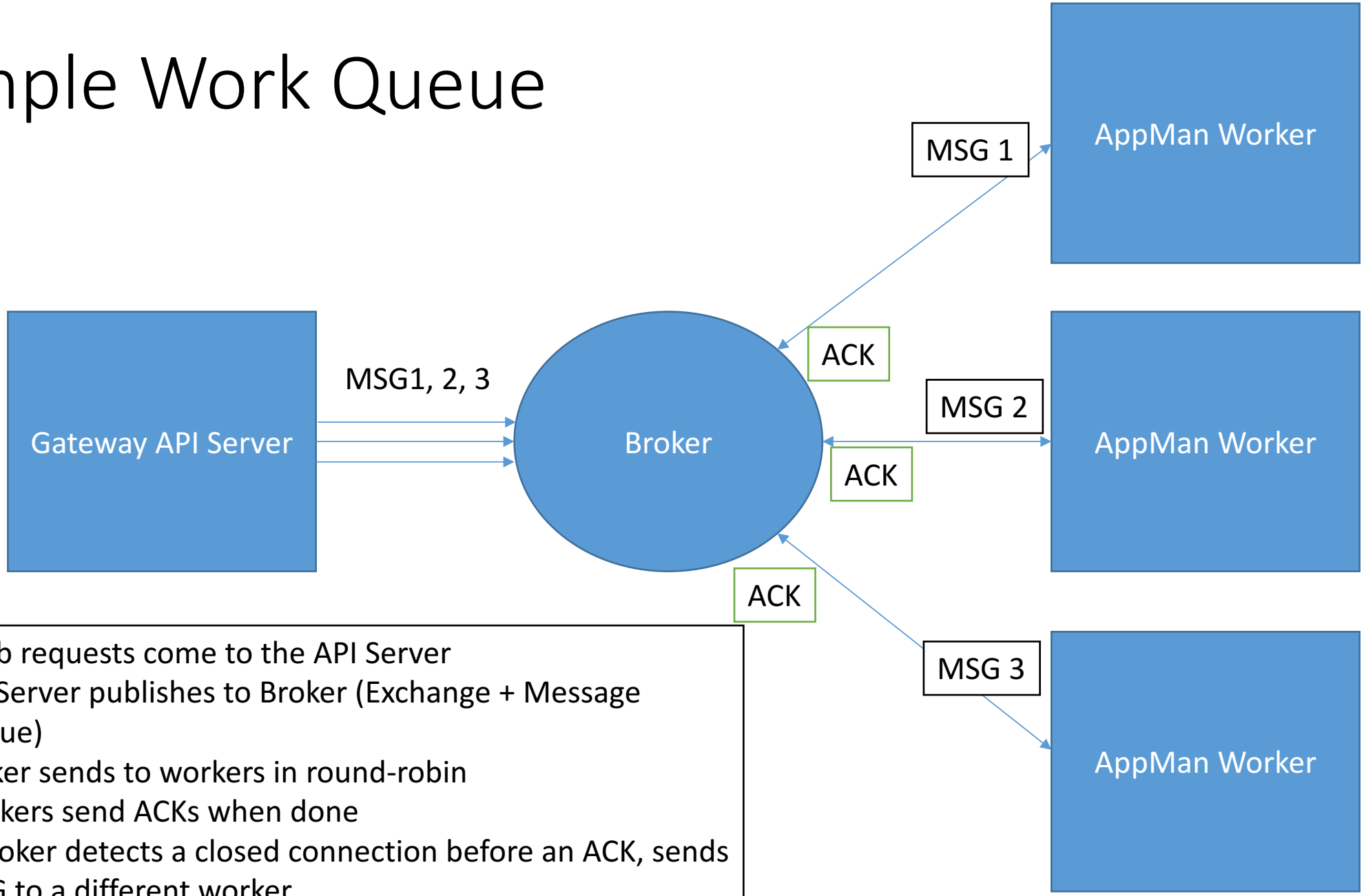
- AMQP does not cover all the capabilities listed above.
  - It can be extended to cover these in many cases
- AMQP messaging system implementations are not necessarily cloud-ready
  - They have to be configured as highly available services.
    - Primary + failover
  - No fancy leader elections, etc as used in Zookeeper + Zab or Apache Kafka
  - Have scaling limitations, although these may not matter at our scales.
- Other messaging systems (Kafka, HedWig) are alternatives

Some Applications

Simple Work Queue

- Queue up work to be done.
- Publisher: pushes a request for work into the queue
  - Queue should be a simple Direct Exchange
- Message Queue should implement “only deliver message once to once consumer”.
  - Round-robin scheduling.
  - RabbitMQ does this out of the box
- Consumer: Sends an ACK after completing the task
- If a Queue-Client closes before an ACK, resend message to a new consumer.
  - RabbitMQ detects these types of failures.

Simple Work Queue



- 3 Job requests come to the API Server
- API Server publishes to Broker (Exchange + Message Queue)
- Broker sends to workers in round-robin
- Workers send ACKs when done
- If Broker detects a closed connection before an ACK, sends MSG to a different worker

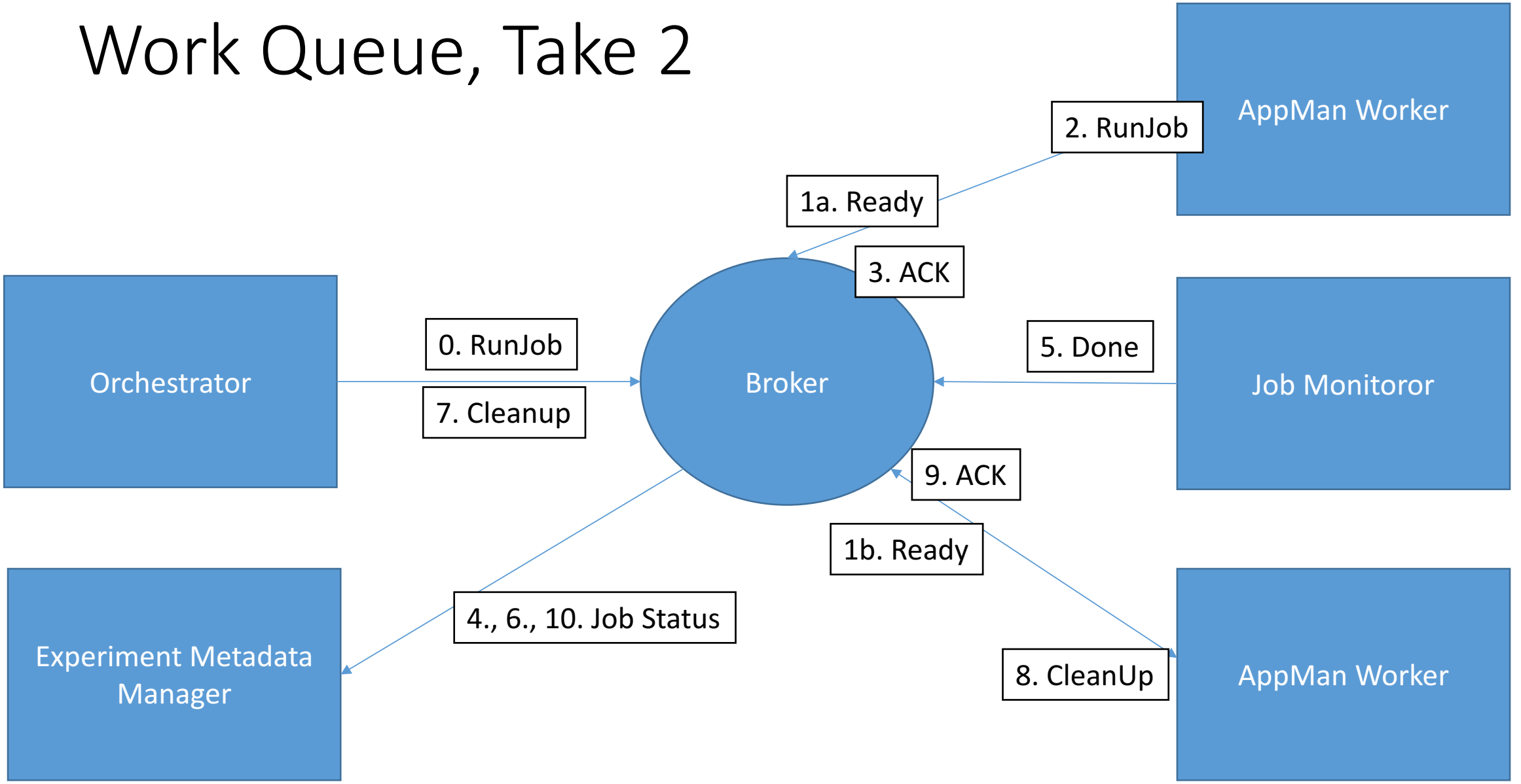
What Could Possibly Go Wrong?

- Jobs take a long time to finish, so ACKs may not come for hours.
  - Durable connections needed between Consumers and Message Queues
  - Alternatively, the ACK could come from a different process
- Jobs may actually get launched on the external supercomputer, so you don’t want to launch twice just because of a missing ACK
- Clients have to implement their own queues
  - Could get another work request while doing work.

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.
- Worker sends ACK after successfully submitting the job.
  - This only means the job has been submitted
  - Worker can take more work
- A separate process 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



What Could Possibly Go Wrong?

- A Worker may not be able to submit the job
  - Remote supercomputer is unreachable, for example
  - We need a NACK
- The Orchestrator and Experiment Metadata components are also consumers.
  - Should send ACKs to make sure messages are delivered.
- Orchestrator and Experiment Metadata Manager may also die and get replaced.
  - Unlike AppMan workers, Orchestrator and EMM may need a leader-follower implementation
- Broker crashes
  - RabbitMQ provides some durability for restarting
  - Possible to lose cached messages that haven’t gone to persistent storage

What Else Could Go Wrong?

- Lots of things.
- How do you debug unexpected errors?
  - Logs
- A logger like LogStash should be one of your consumers
- No one-to-one messages any more.
- Everything has at least 2 subscribers
  - Your log service
  - The main target
- Or you could use Fanout

Summary

- Message systems provide an abstract system for routing communications between distributed entities.
  - You don’t need to know the physical addresses
- 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.