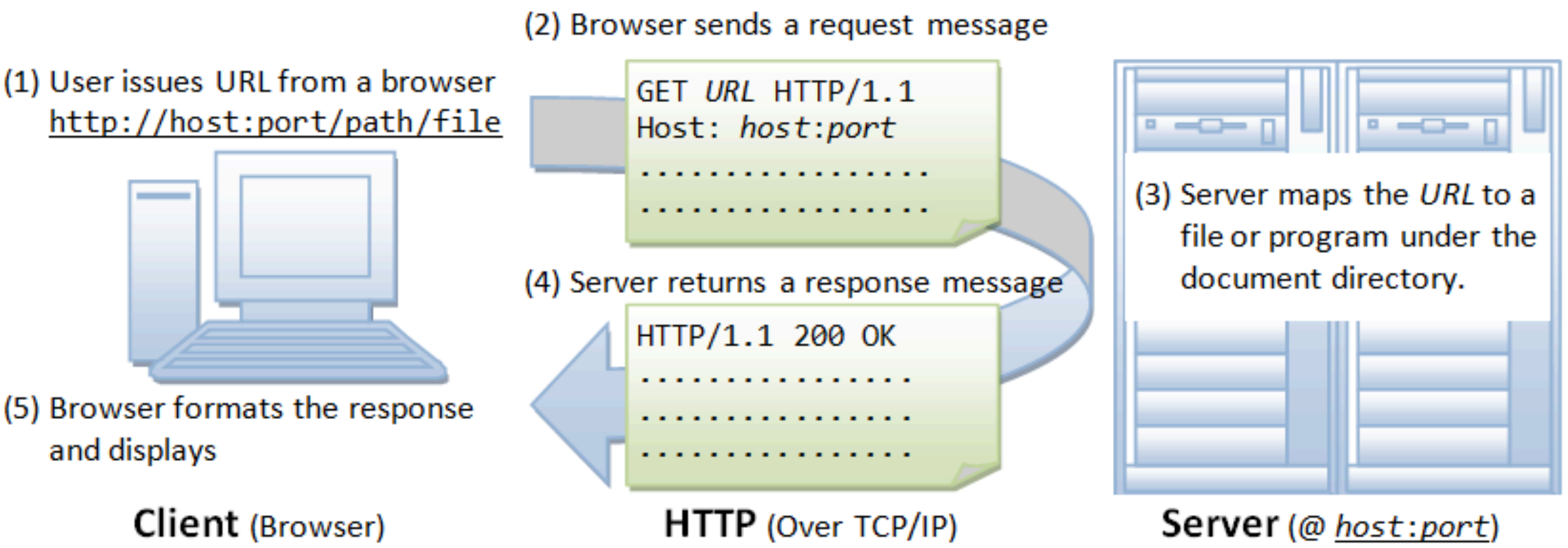
Microservices, Messaging and Science Gateways

Review microservices for science gateways and then discuss messaging systems.

What Is HTTP?

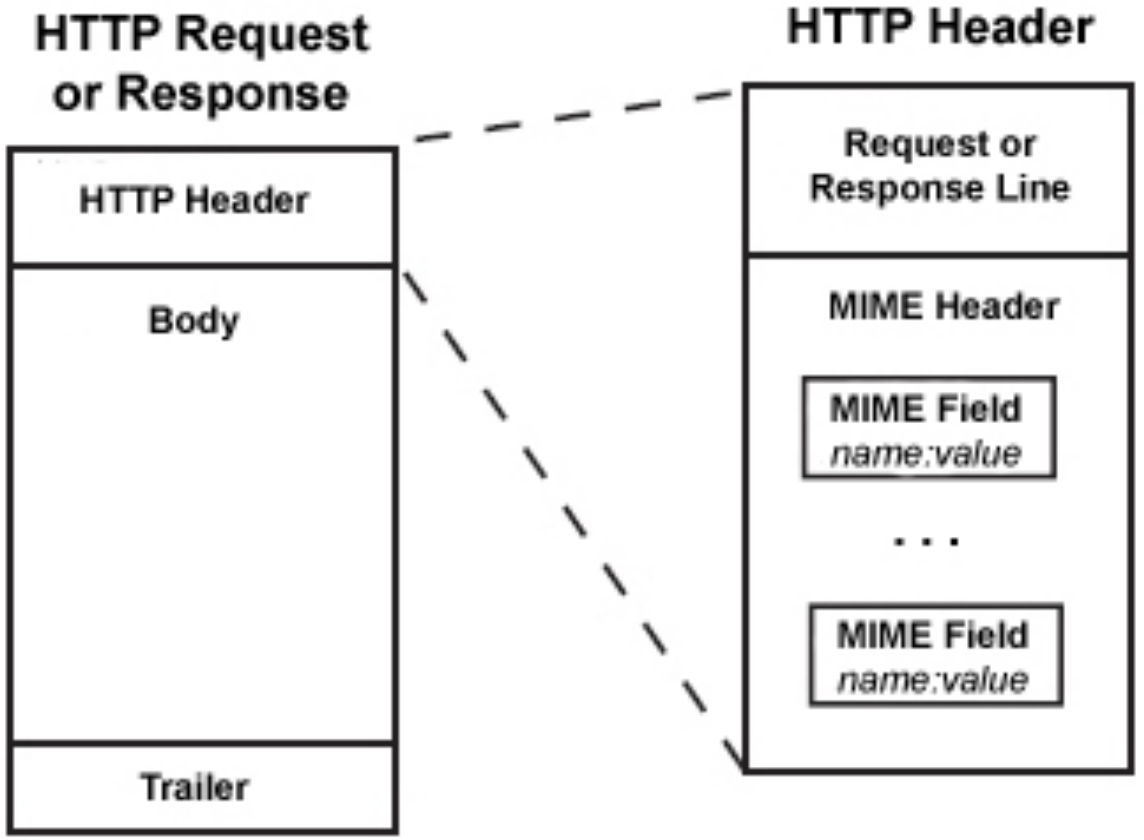
HTTP Summary



REST generalizes this for machine-to-machine communication

HTTP Features

- HTTP official specifications
  - https://tools.ietf.org/html/rfc2616
- Request-Response
- Uses URLs to identify and address resources.
- Stateless (but extendable)
- Limited set of operations
  - GET, PUT, POST, DELETE, HEAD, ...
- Transfers hypermedia in the body
  - HTML, XML, JSON, RSS, Atom, etc.
- Extendable by modifying its header
  - Security, etc.
- Point to point security
  - TLS: transport level
- Well defined error codes



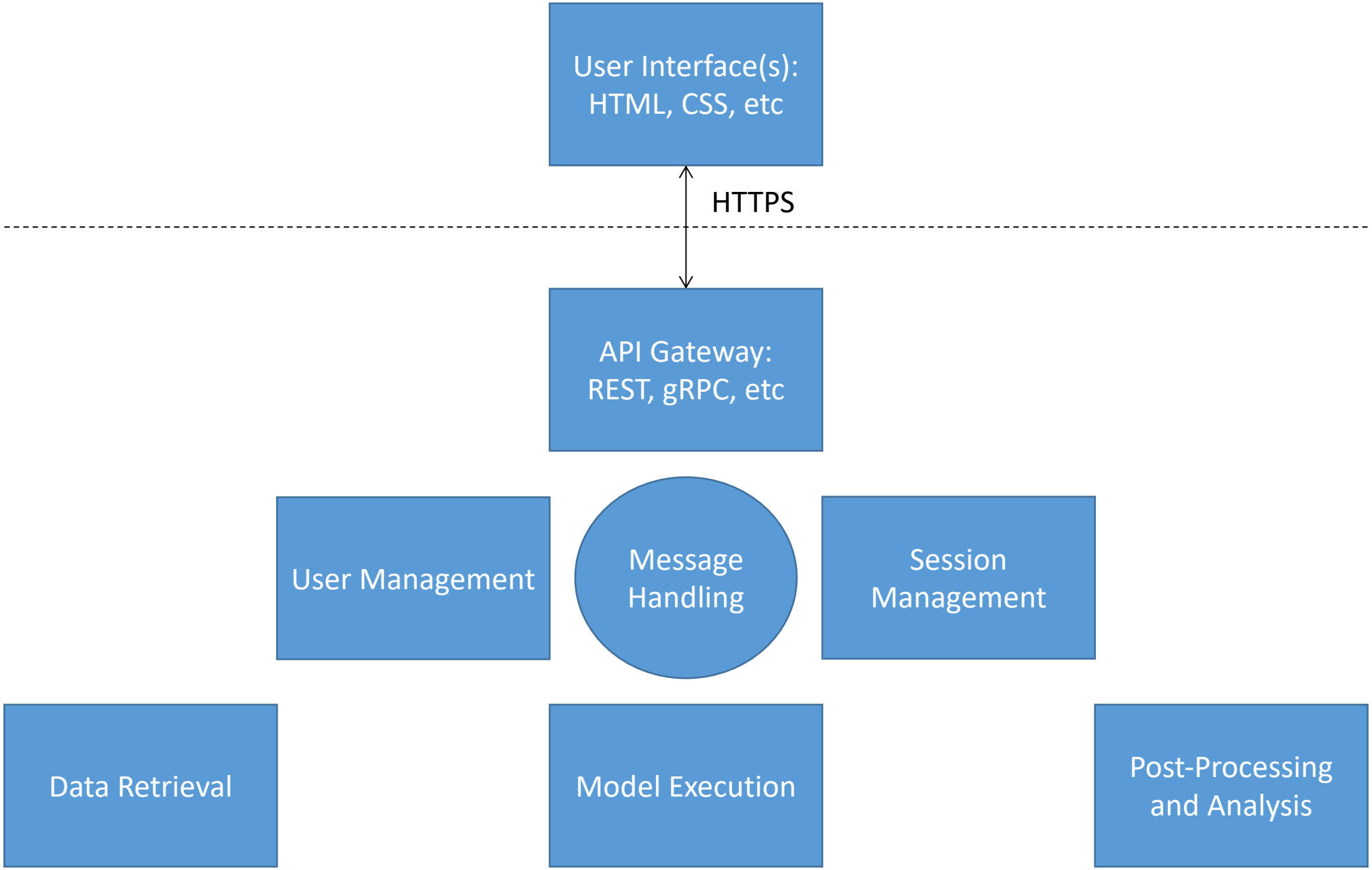
What Are Some Problems Using HTTP for Microservices?

Some Answers

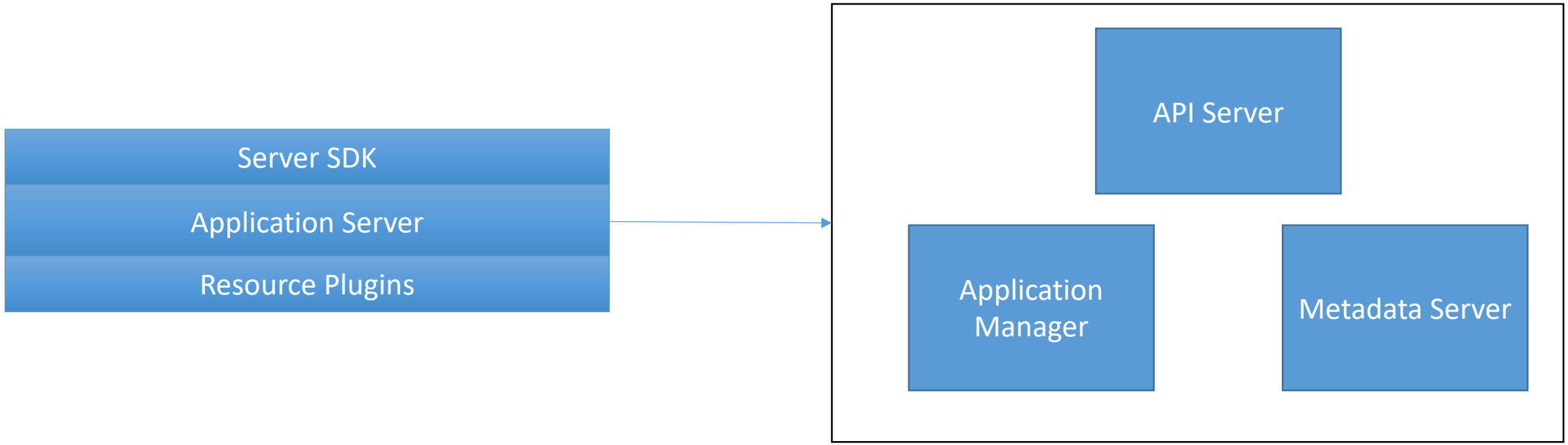
- HTTP is point-to-point
  - Sender and receiver are tightly coupled
  - How do you send messages to more than one recipient?
- Request-response
  - Block until you get a response
  - How do you push a message back later?
- Messages (HTML) are designed for human, not machine consumption
  - Hypermedia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

HTTP is not designed for more complicated messaging scenarios

What Do We Need?

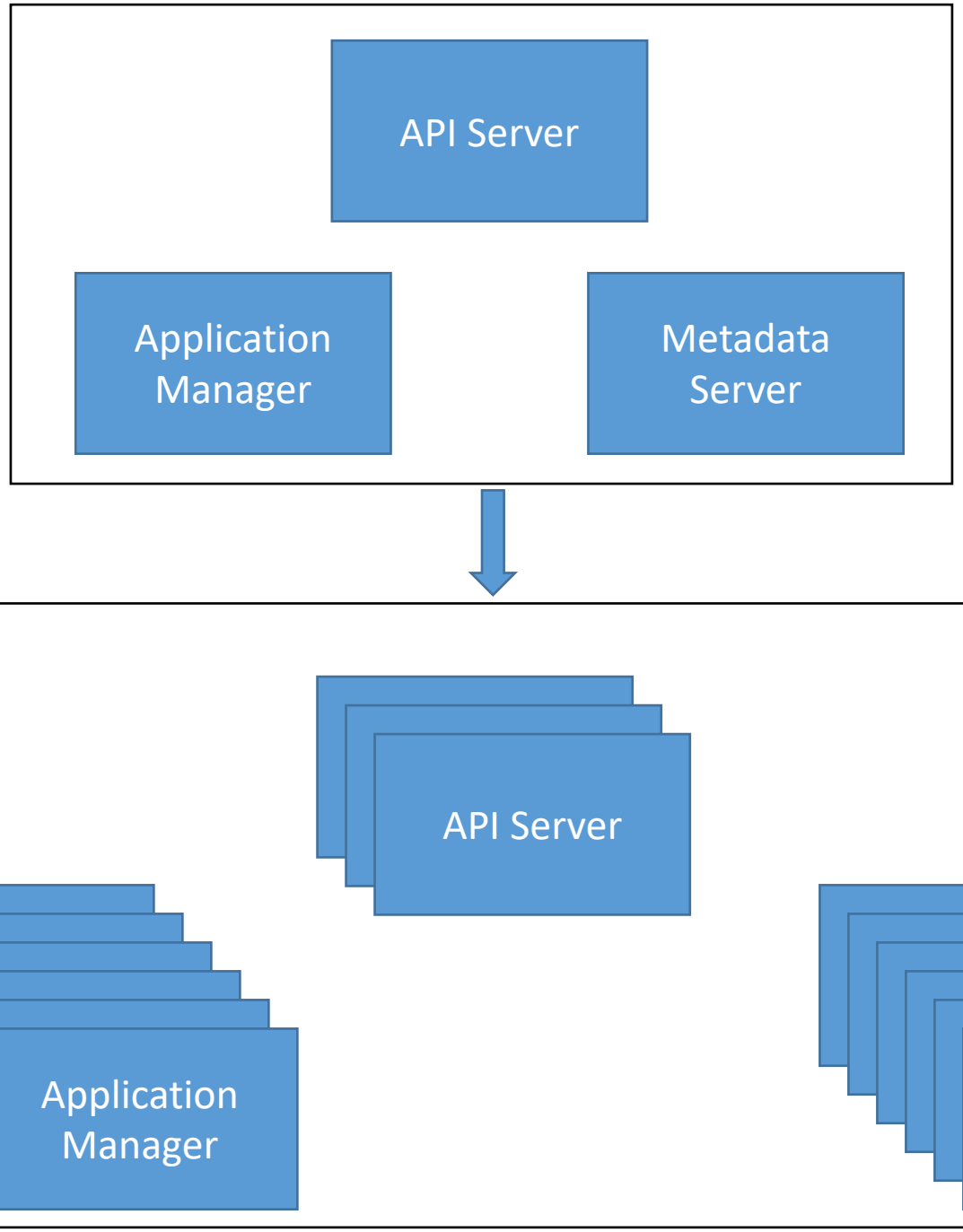

User Interface(s):
HTML, CSS, etc
HTTPS
API Gateway:
REST, gRPC, etc
User Management
Message
Handling
Session
Management
Data Retrieval
Model Execution
Post-Processing
and Analysis

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



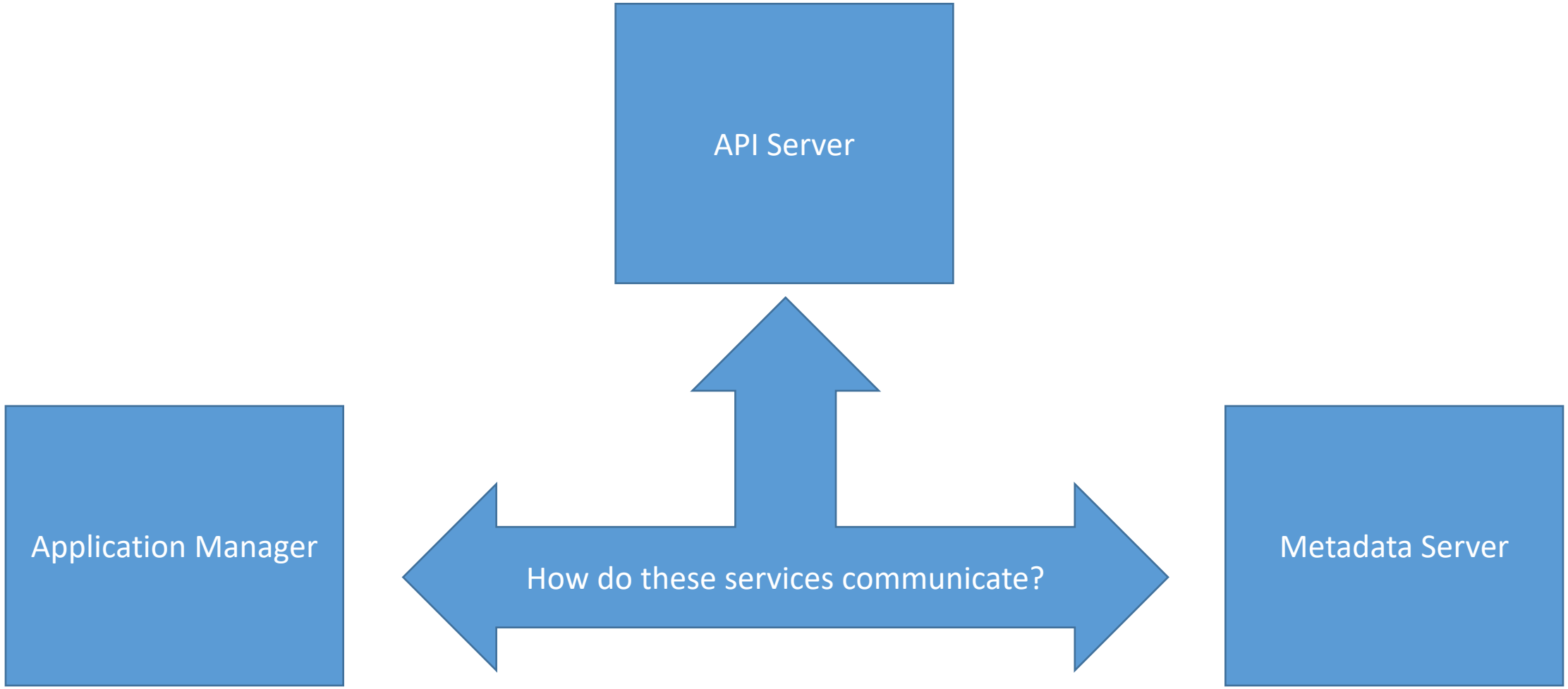
Communication patterns and system state are important

Messaging in Distributed Systems

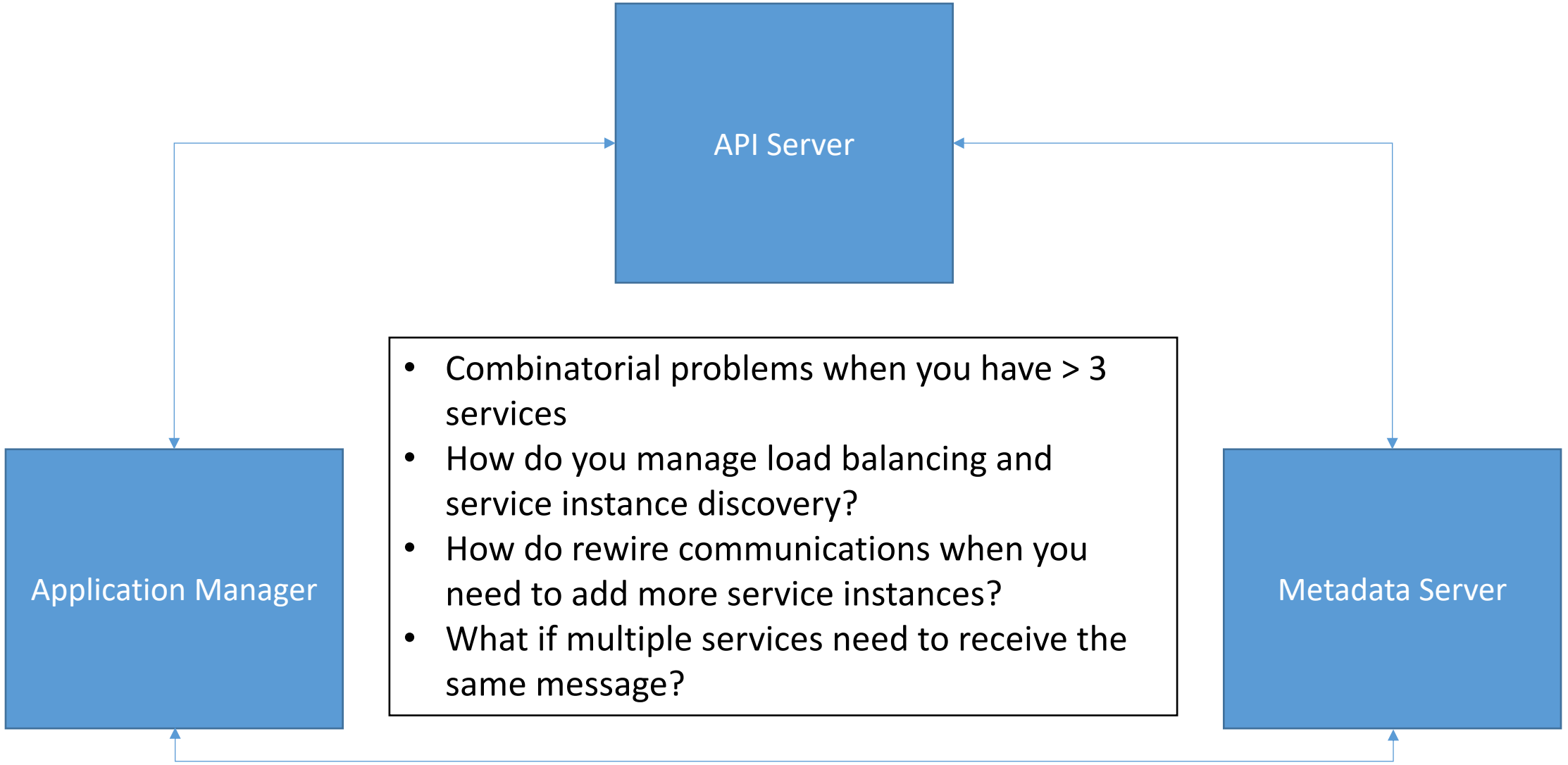
We'll use Advanced Message Queuing Protocol (AMQP) and RabbitMQ overviews as examples

This is not an endorsement of these for the assignments!

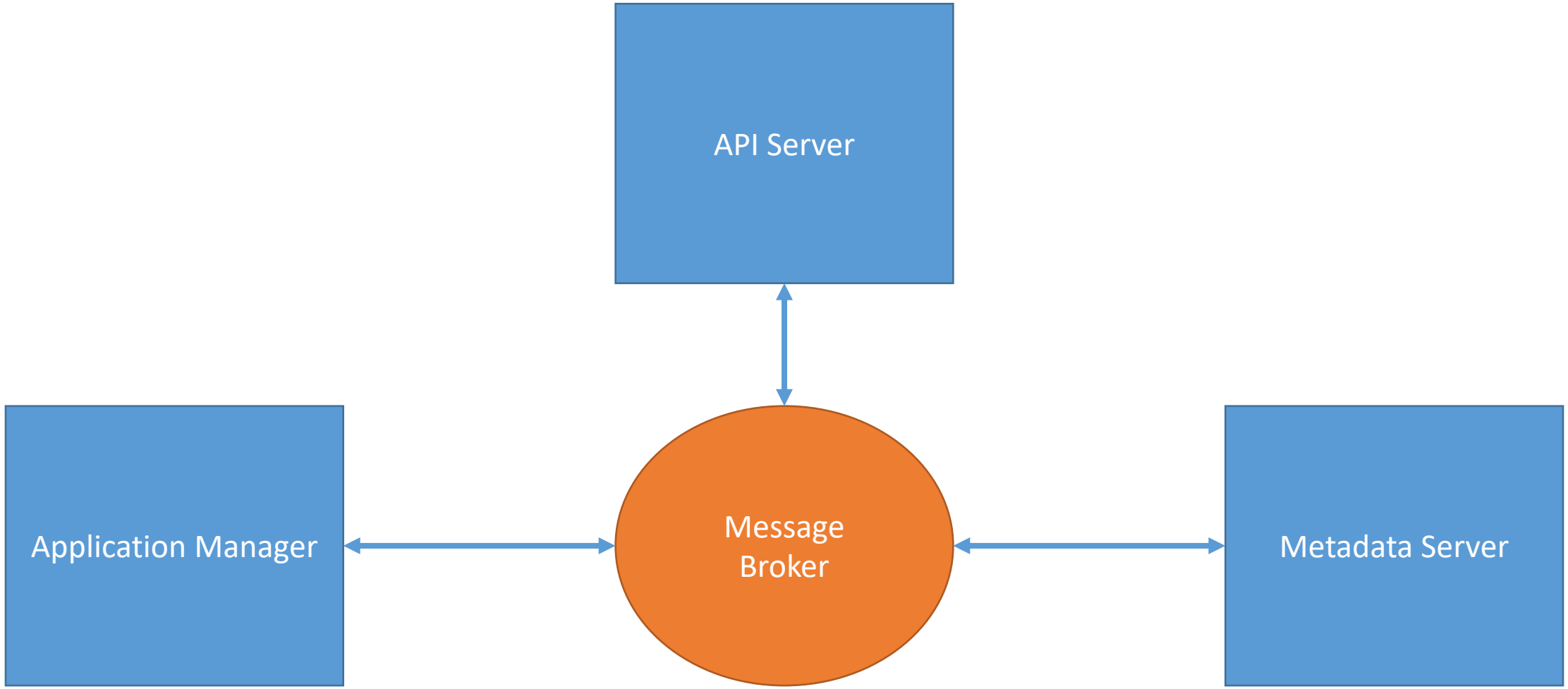
Basic Components of Science Gateway Microservices



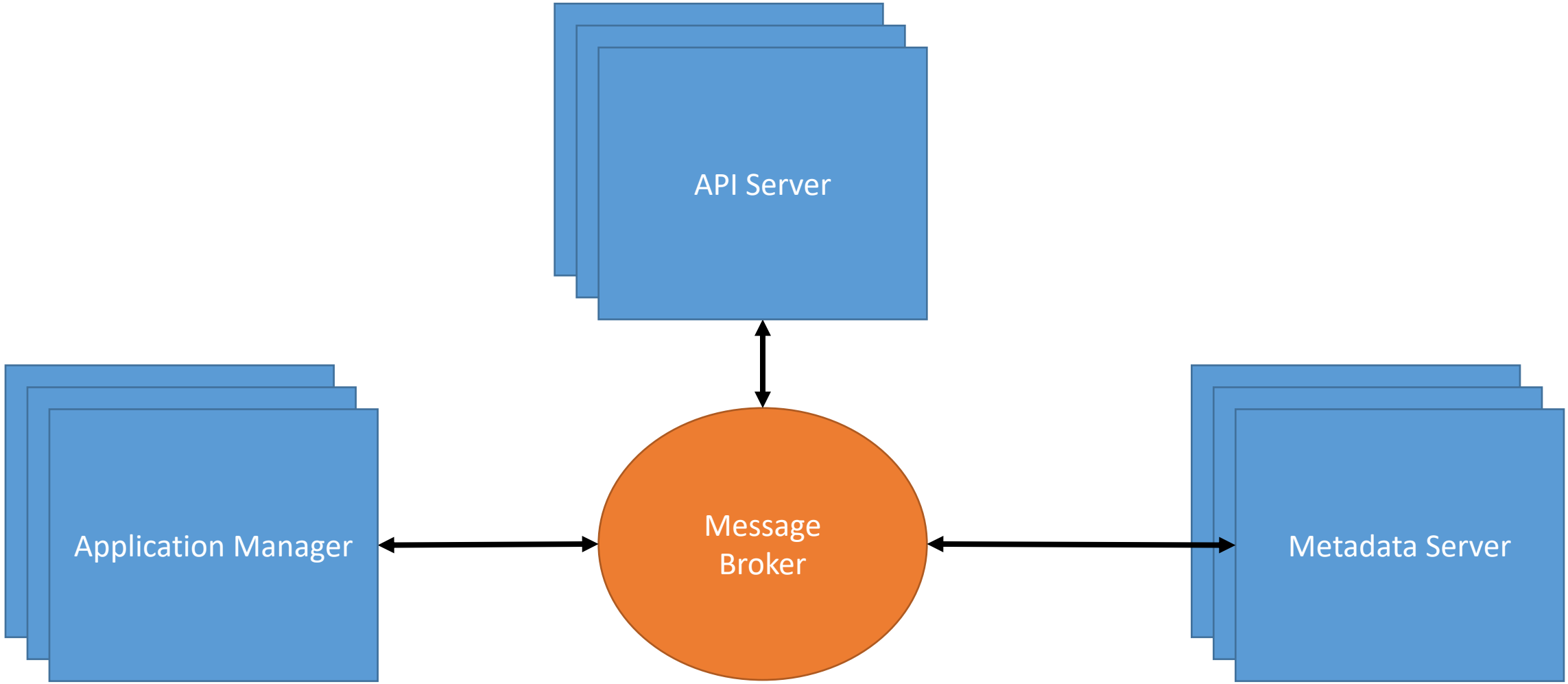
Point to Point Communication Is Brittle



Messaging Systems Solve Many of These Problems



They Work Well with Service Replication



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
  - Etc
- Many-to-many rather than just one-to-one
- Can support both push and pull messaging
- A bit more like email for machines

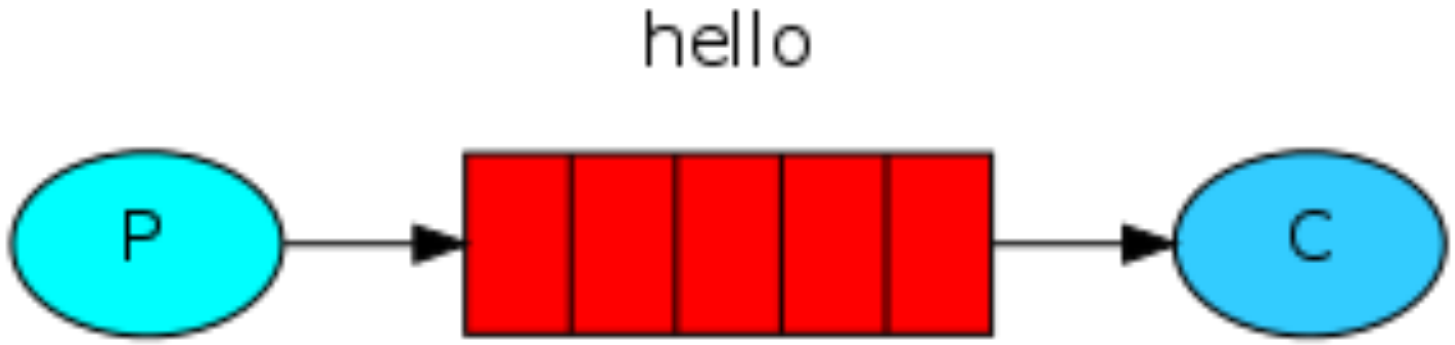
Value of Message Systems, Continued

- Messaging systems remove the need for publishers and consumers to know too much about each other
  - 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.
  - Decouple the logical system from the physical system
- Synchronous and asynchronous messages are supported.
- More efficient:
  - RabittMQ can multiplex multiple channels of communication over a single TCP/IP connection
  - Kafka can support high throughput

Message Exchange Patterns

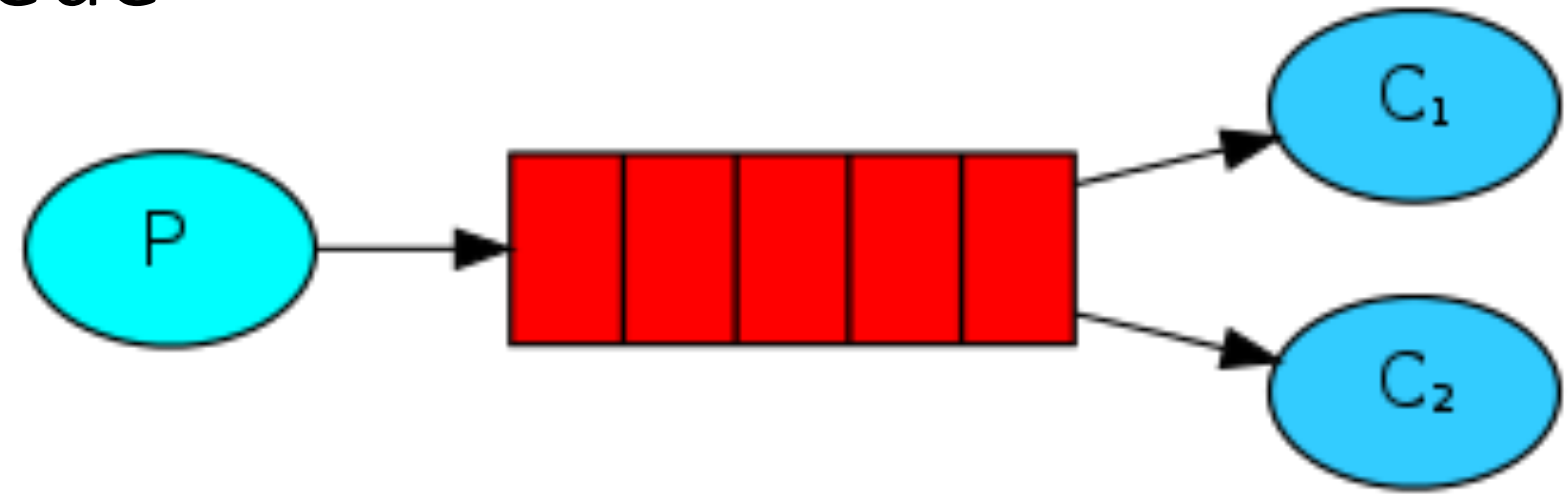
See the RabbitMQ tutorial for more examples

Point to Point Communication



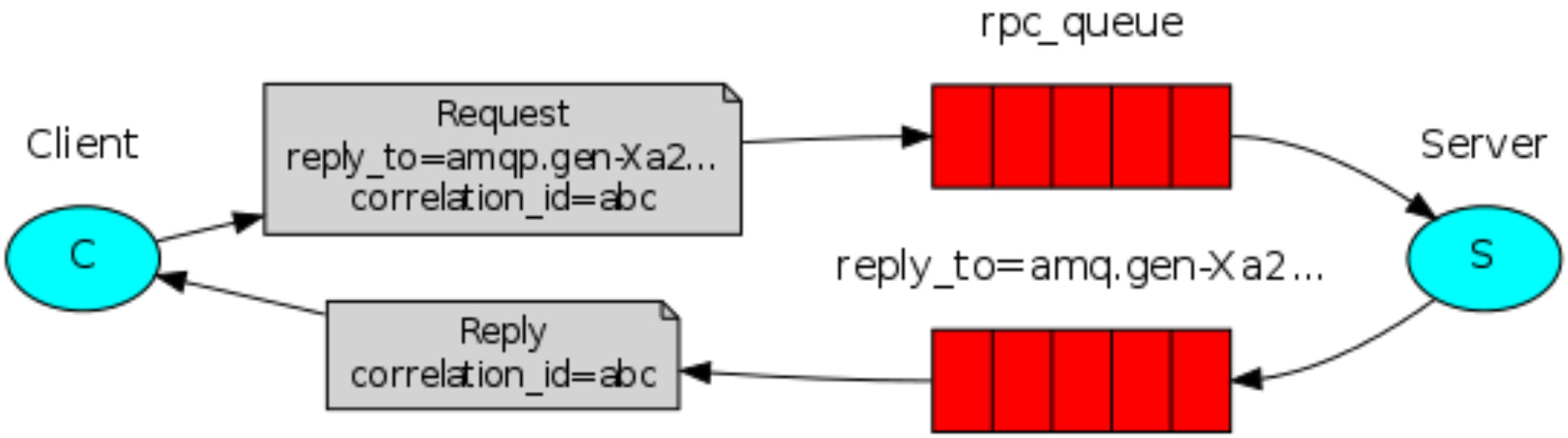
- Publisher (P) sends a message to the broker, which puts it in queue
- The broker routes to the appropriate Consumer (C) using routing keys
- Messages are *pushed* to the consumer
- The publisher doesn't explicitly know how to connect to the consumer.
  - The broker handles this
- What are some useful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

Work Queue



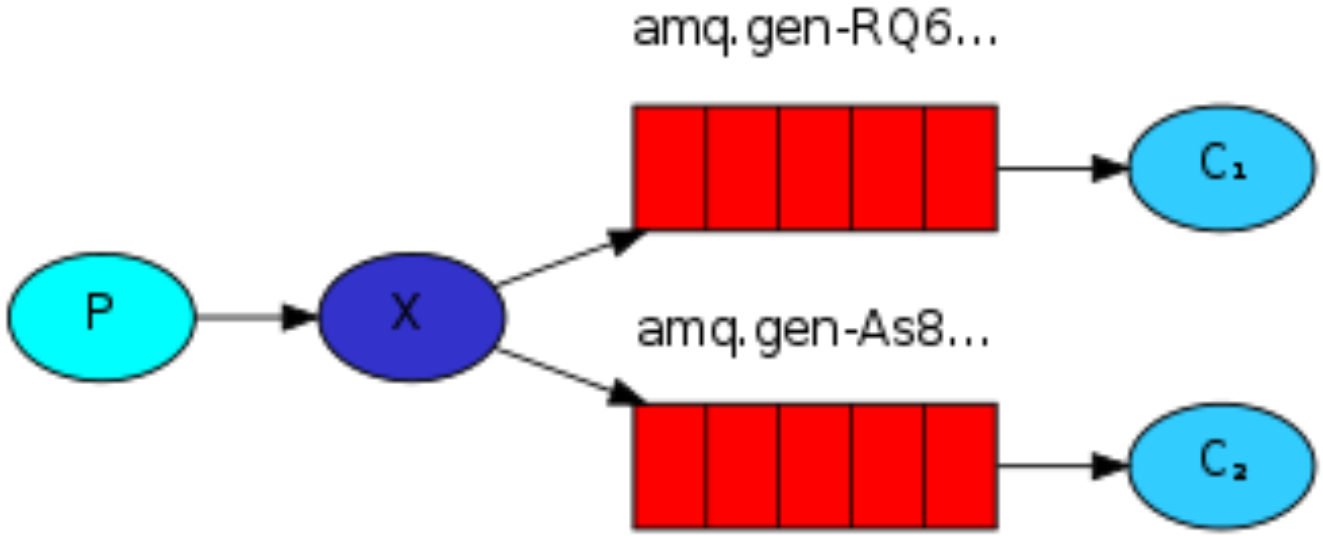
- Consumers that have capacity to work take messages from the queue
- Multiple consumers listen to the same queue
- Each message is delivered to only one consumer
- The broker can use round robin or other scheduling strategies for routing messages to specific consumers
- You can program/configure consumers 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.

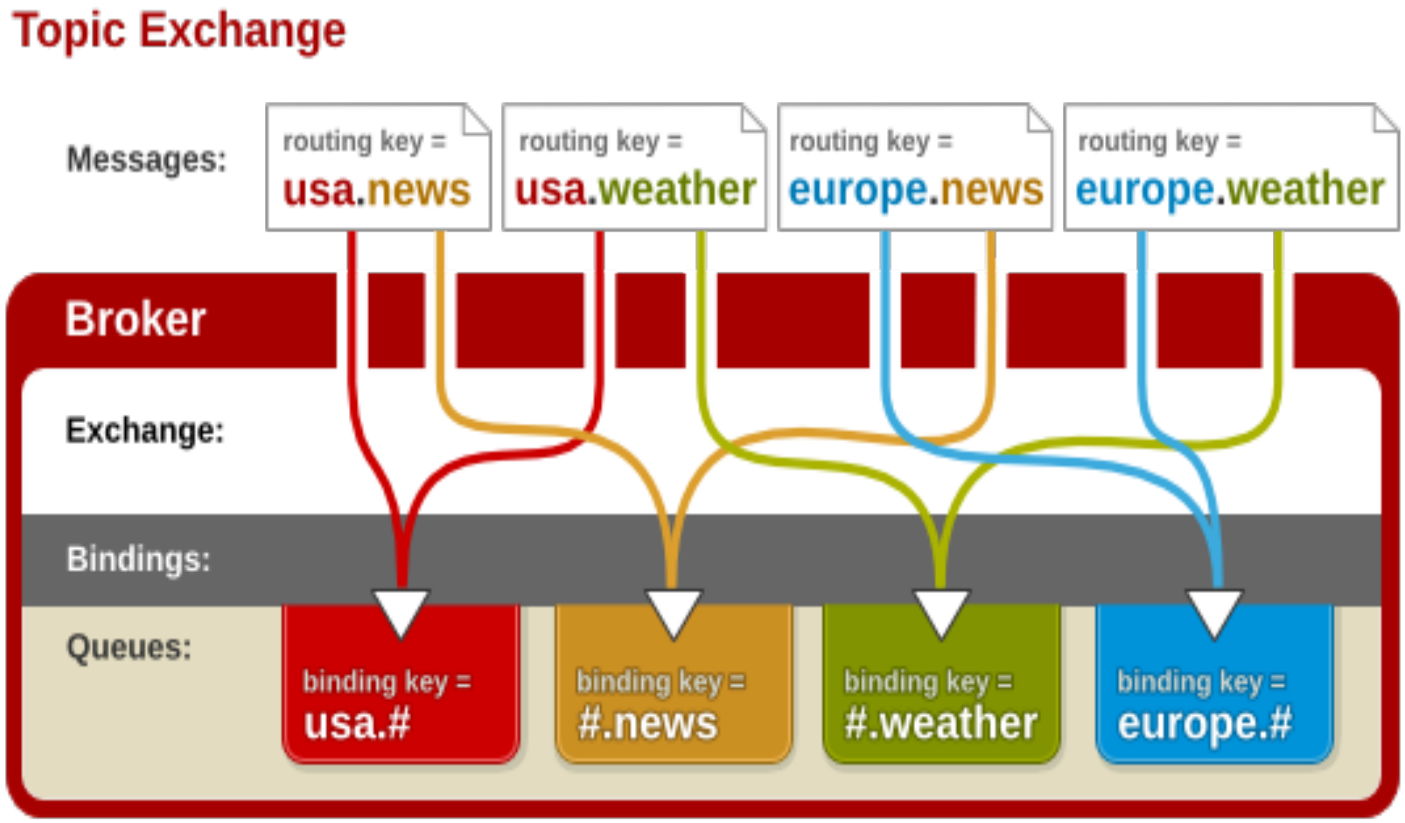
Publish/Subscribe



- Publisher sends a message to multiple queues (same broker) via an Exchange (X)
  - X and the queues are all in the same broker process
- Consumers attach to different queues
- This allows multiple consumers to receive the same message
- Variations: Fanout, Routing, and Topic-Based Pub/Sub

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

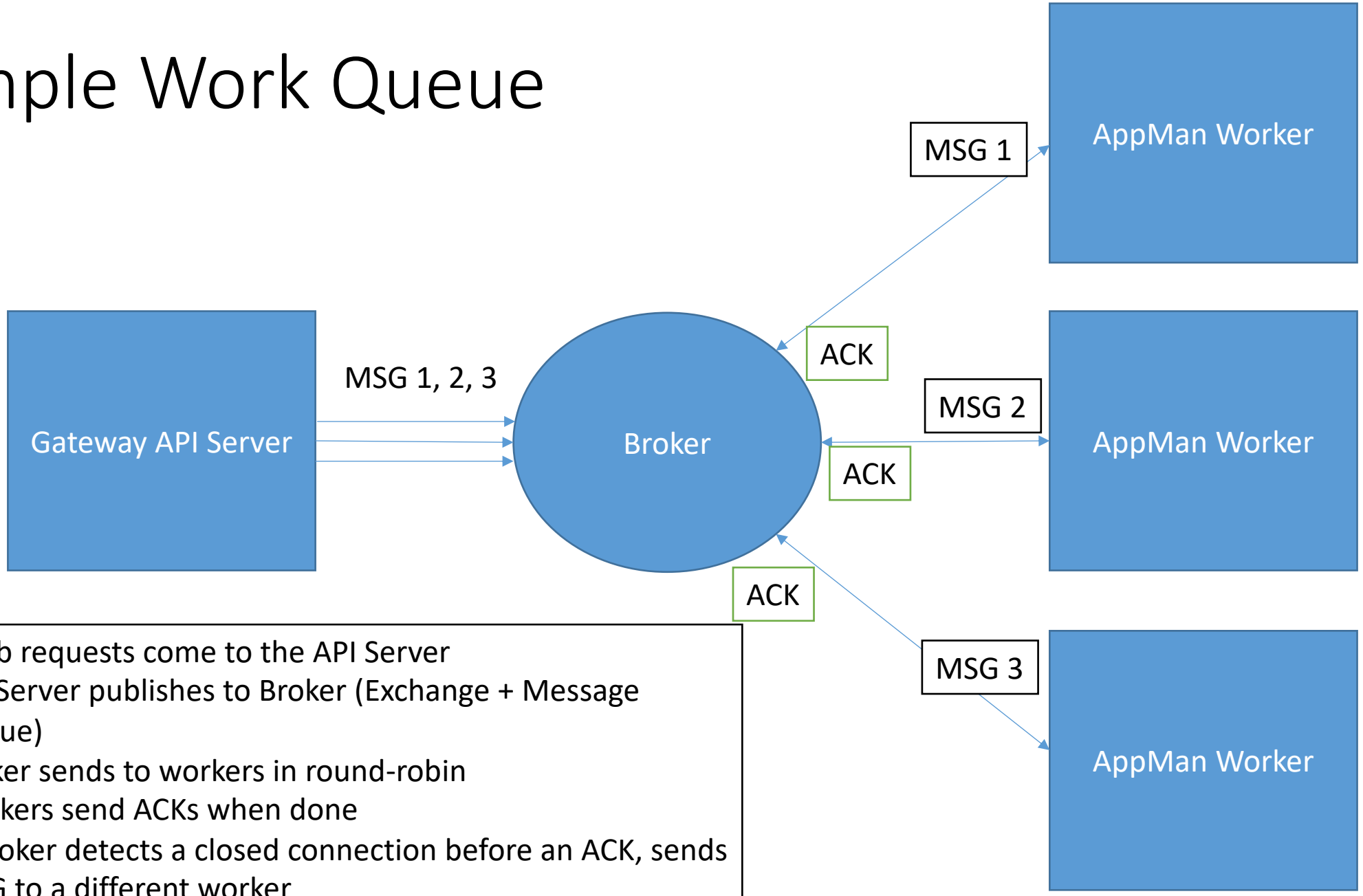


Some Applications

Simple Gateway Work Queue with RabbitMQ

- 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
  - When is this a failure condition?
- Jobs may get launched on an external resource (supercomputer or cloud), so you don’t want to launch twice just because of a missing ACK
- Clients must implement their own queues
  - Could get another work request while doing work.

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

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 necessarily cloud-ready

- 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.

Summary

Message systems provide an abstract system for routing communications between distributed entities.

- You don’t need to know the physical addresses

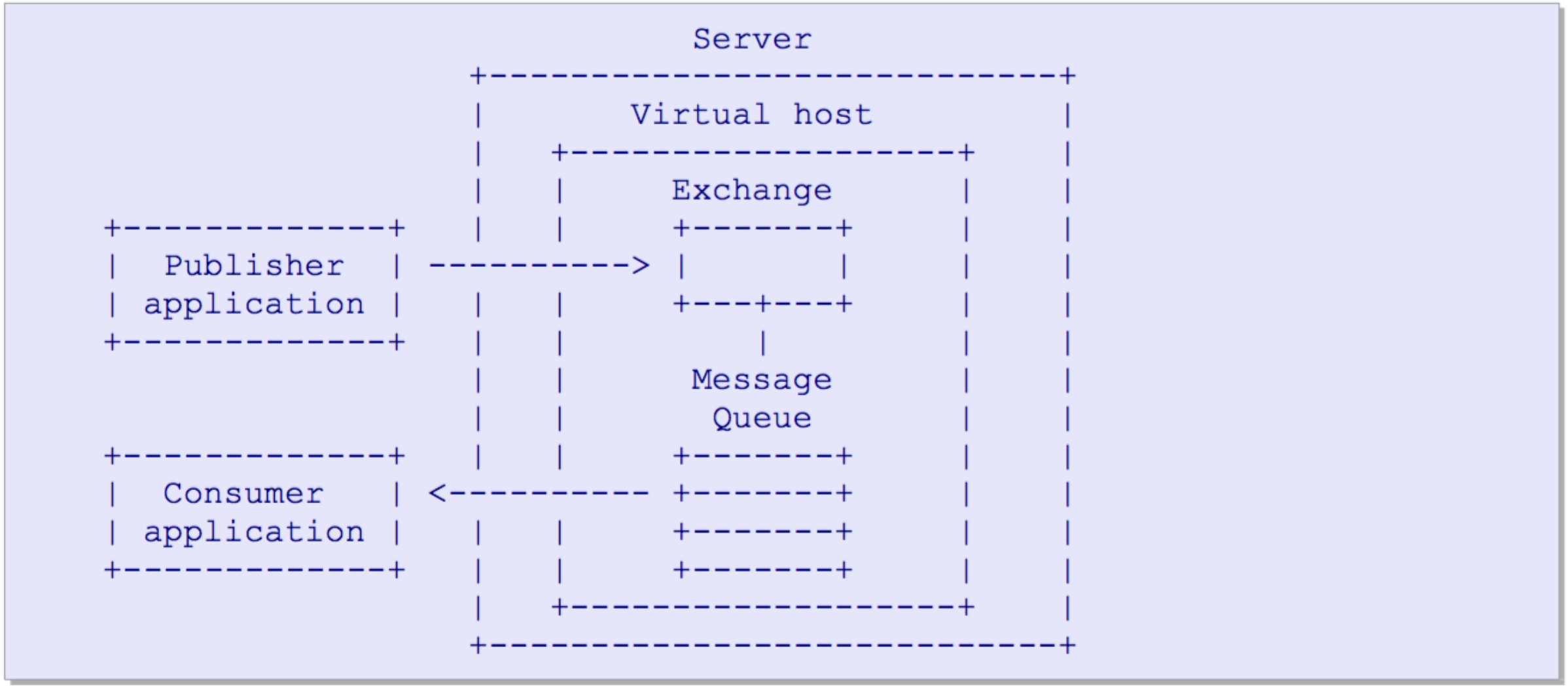
Support multiple messaging patterns out of the box.

- 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.

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

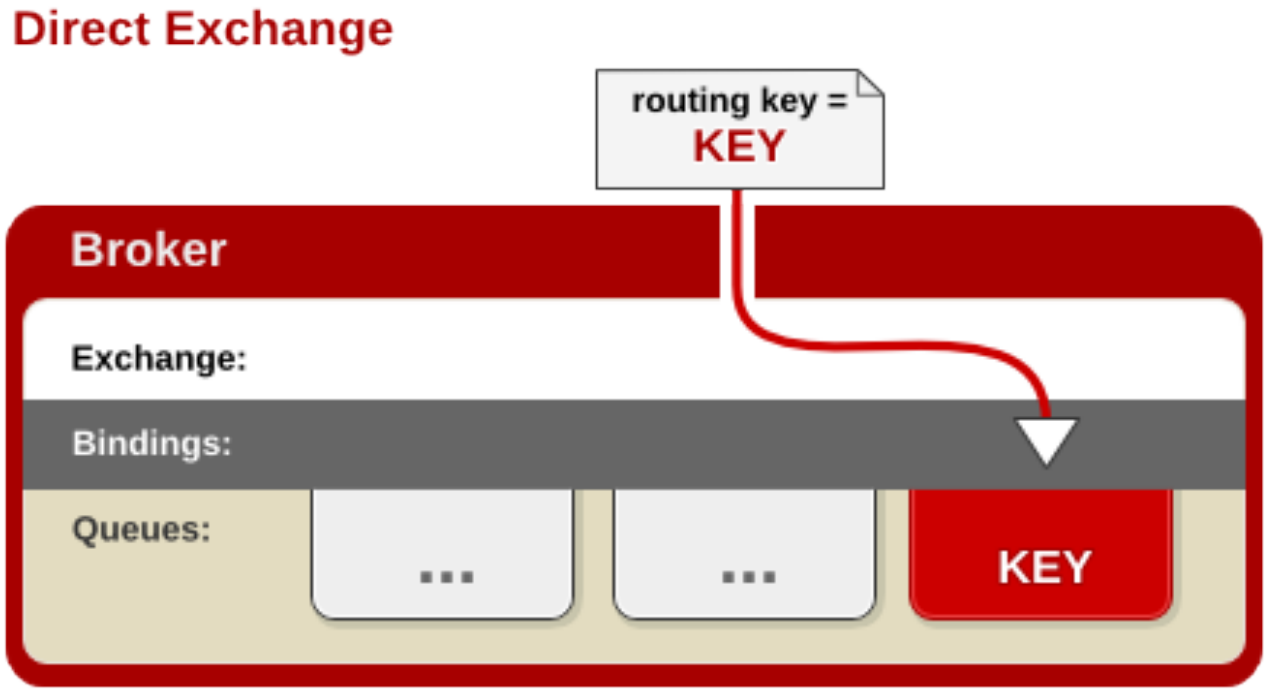
- Useful for many-to-many messaging
- In microservice-based systems, several different types of components may want to receive the same message
  - But take different actions
  - Ex: you can always add a logger service
- You can always do this with explicitly named routing keys.
- You may also want to 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

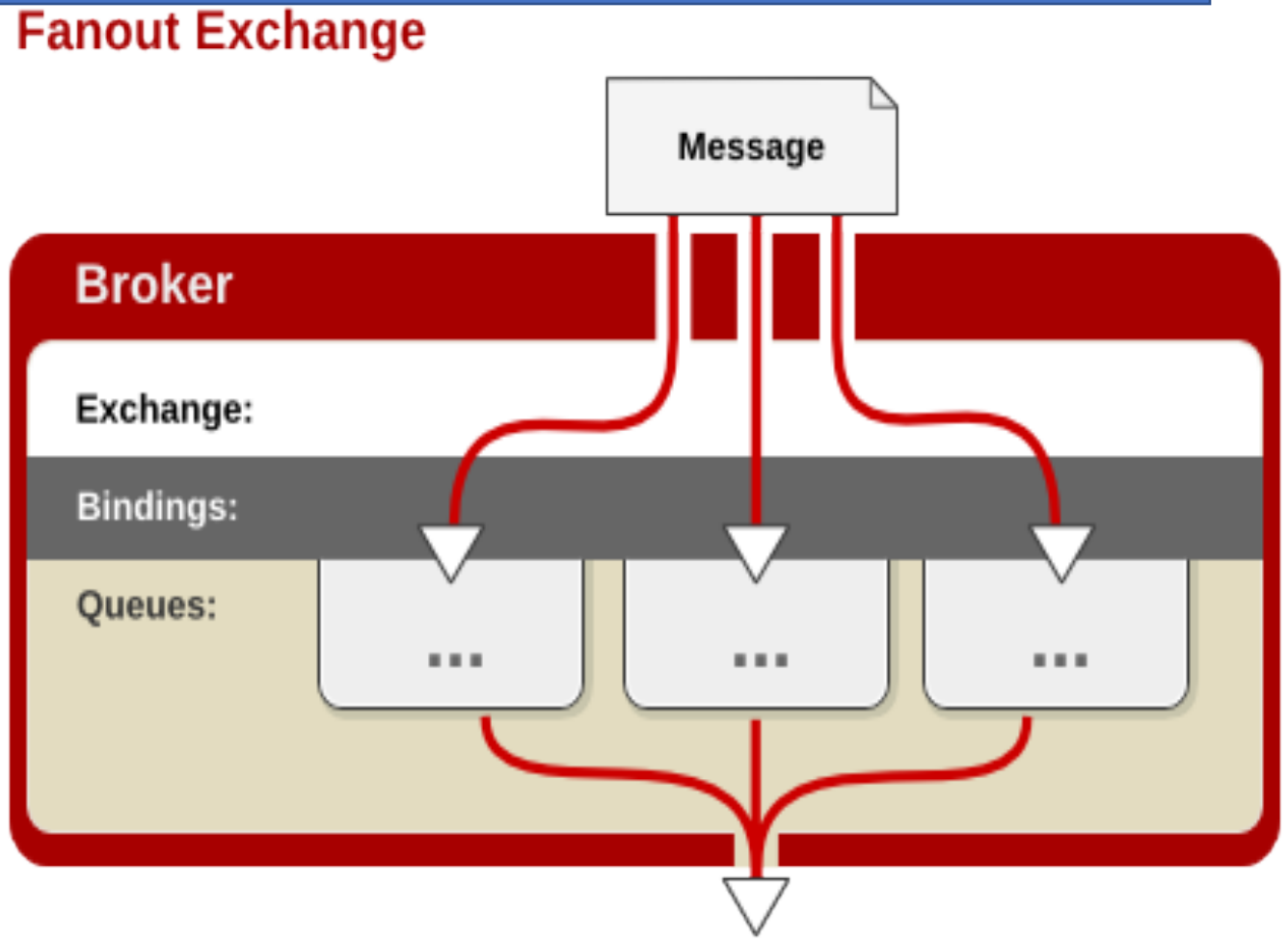
- A publisher sends a message to an exchange with a specific routing key.
- The exchange routes this to the message queue bound to the routing key.
- A consumer receives the messages if listening to the queue.
- Default: round-robin queuing to deliver to multiple subscribers of same queue




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

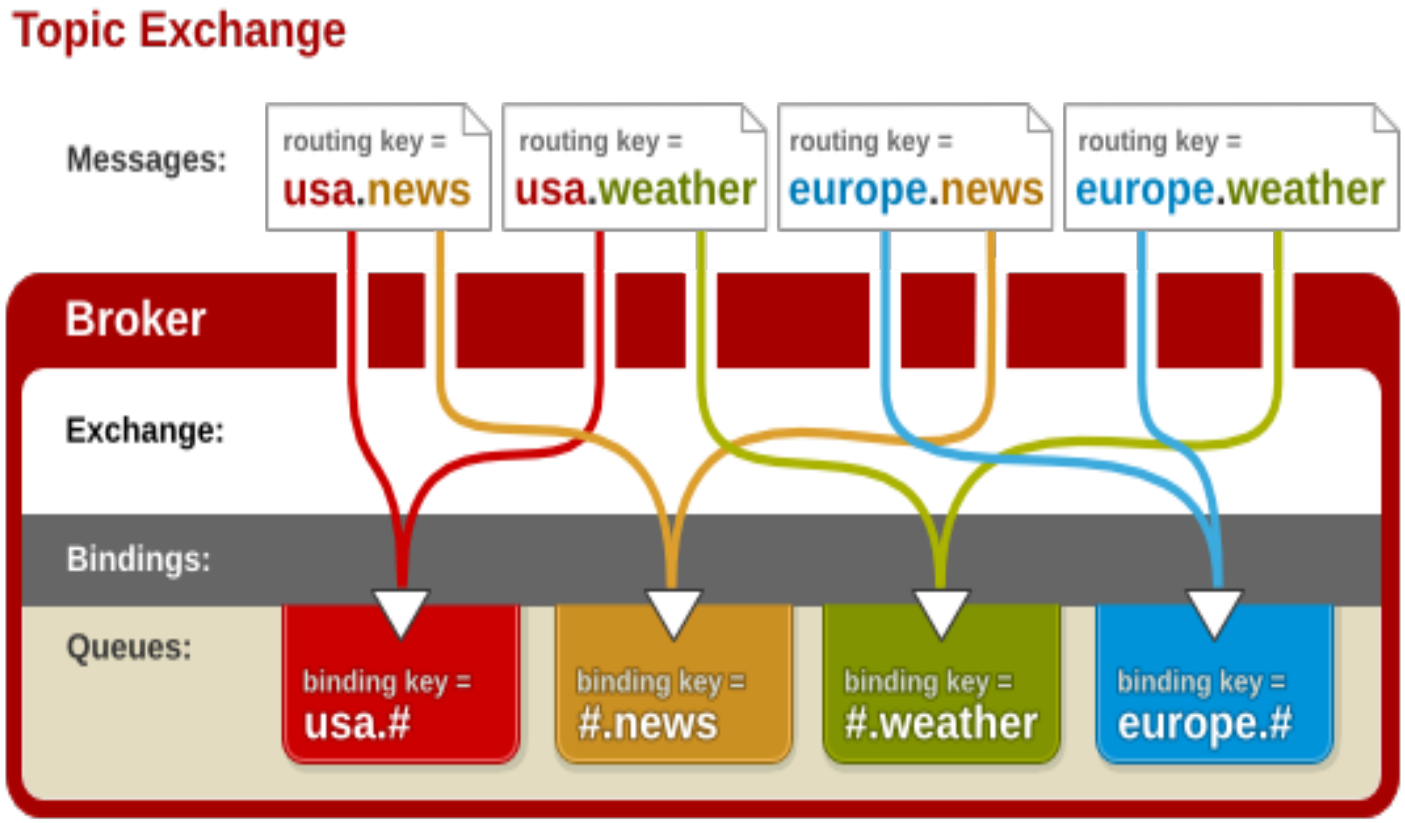
Fanout Exchange

- 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

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