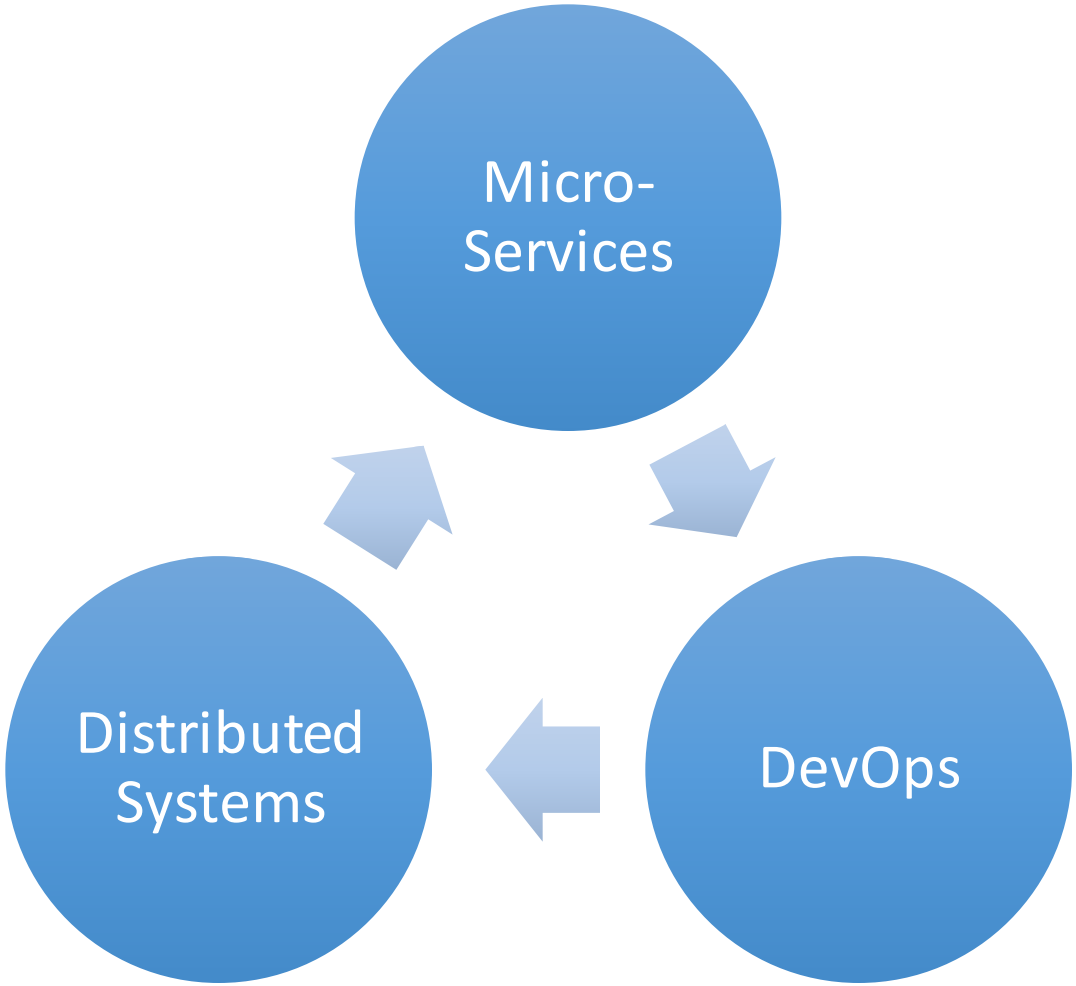

Micro-
Services
DevOps
Distributed
Systems

Microservices and Science Gateways

Martin Fowler's Definition

- Develop a single application as a suite of small services
- Each service runs in its own process
- Services communicate with lightweight mechanisms
  - "Often an HTTP resource API"
  - But that has some problems
  - Messaging is better.
- These services are built around business capabilities
- Independently deployable by fully automated deployment machinery.
- Minimum of centralized management of these services,
  - May be written in different programming languages
  - May use different data storage technologies.

http://martinfowler.com/articles/microservices.html

Monolithic Applications

Enterprise applications three tiered architecture

- a client-side user interface
- a database
- a server-side application.

Server-side Application

- handle HTTP requests,
- execute domain logic,
- retrieve and update data from the database,
- select and populate HTML views to be sent to the browser.

This server-side application is a *monolith*

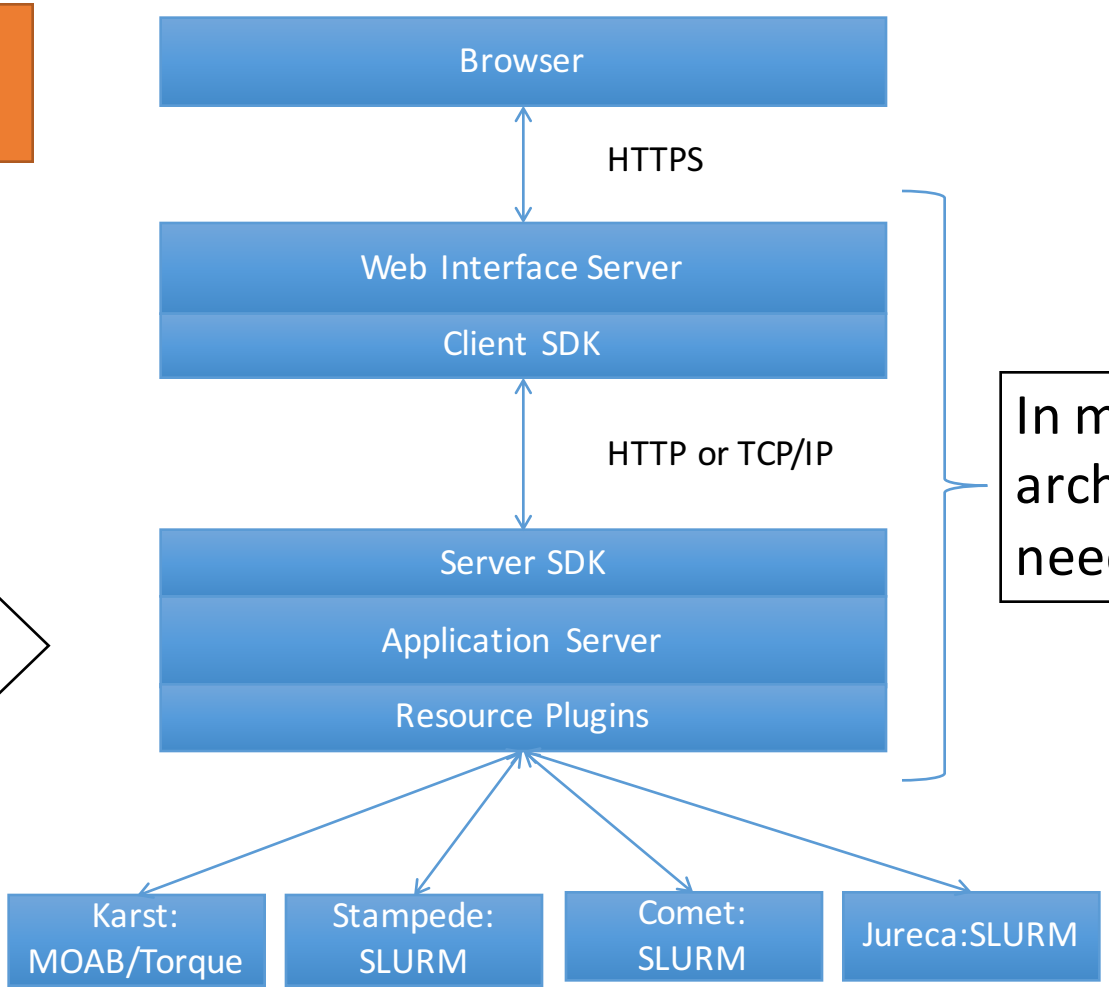
- A single logical executable.
- Changes require building and deploying a new version of the server-side application.

Other Questionable Monolithic Patterns

- The server-side application interacts with a single database
  - Not everything needs to fit into one DB system
  - Not just CAP choices: use the DB that fits each subsytem's problem and your development patterns
- Splitting teams into tiers
  - The DB team, the UX team, the Server App team
  - The deployment and operations team

CAP: Consistency-Availability-Partition Tolerance

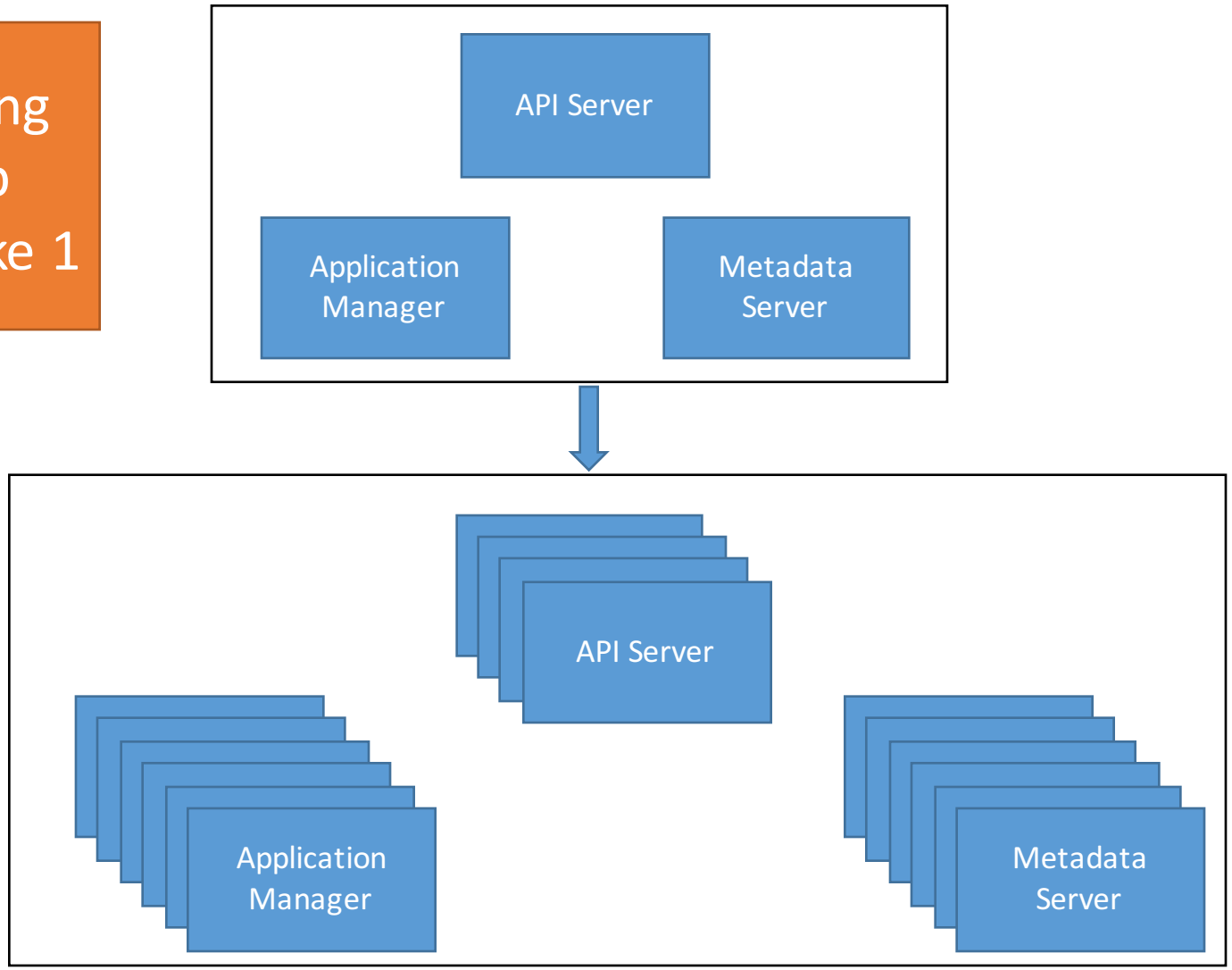
Recall the Gateway Octopus Diagram



"Super" Scheduling and Resource Management

In micro-service arch, these also need scheduling

Decoupling the App Server, Take 1



What Science Gateway Microservices Can We Identify?

My Suggested List (1/2)

Entry Point	• API Server • High availability, load-balanced
Orchestrator	• Scheduler • Highly available leader
Stateless Metadata Server	• Resources, Applications • Highly available?

My Suggested List (2/2)

Stateful Metadata Server	• Experiment state
Application Manager	• Submits jobs • Fire and forget
Monitor Service	• Monitors jobs on remote resources
Logging	• Centralize your service logs

Draw and Think

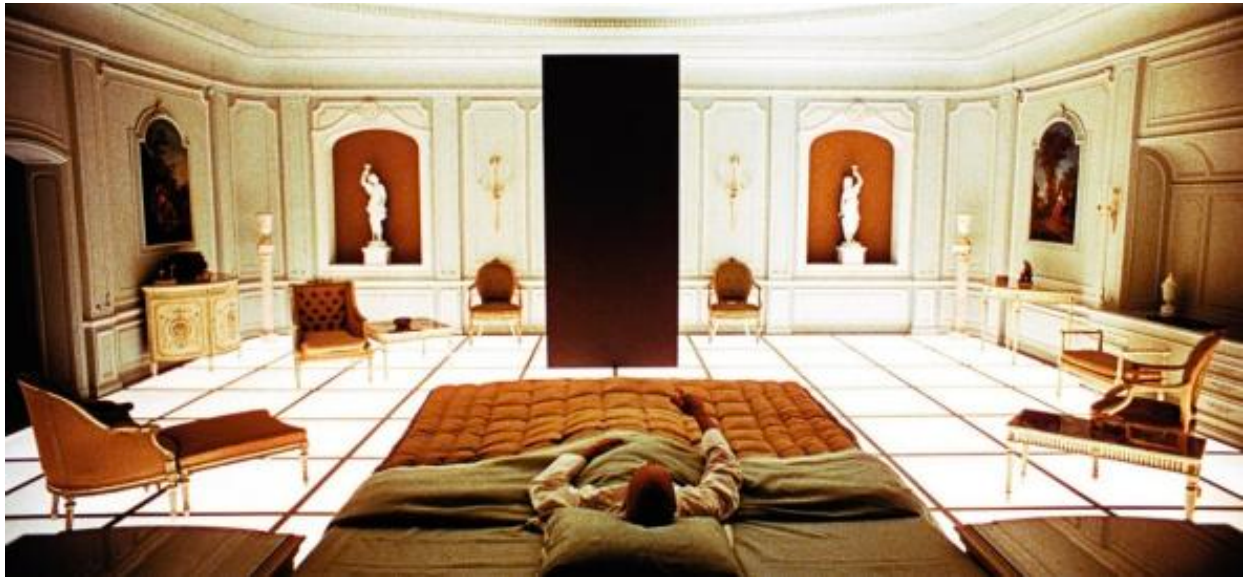
Thought Exercise

- Assume Amazon Cloud and a big budget
- How do these components interact?
- How can I distribute copies of components across multiple availability zones?
- What kinds of failures can occur?
- How can I handle failures of different components?

Last Thoughts: Are Monolithic Applications Bad?

- A monolithic server is a natural way to approach building many systems.
- Monolithic applications can and should still be modular
- All your logic for handling a request runs in a single process
  - You can use the basic features of your language to divide up the application into classes, functions, and namespaces.
- Monoliths can still use DevOps
  - You can run and test the application on a developer's laptop,
  - Use a deployment pipeline to ensure that changes are properly tested and deployed into production.
- You can horizontally scale the monolith by running many instances behind a load-balancer.
- DB CAP problems don’t hit most applications

The Dividing Line



When do you decouple your code? When is a monolithic application better?

Monolithic Applications: Traditional Software Releases

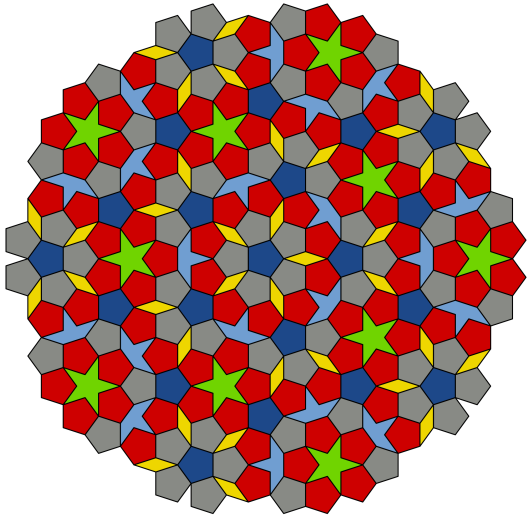
- Software releases occur in discrete increments
- Run on clients' systems
- Releases may be frequent but they are still distinct
  - Firefox
  - OS system upgrades
- Traditional release cycles
  - Extensive testing
  - Alpha, beta, release candidates, and full releases
- Extensive recompiling and testing required after code changes
- Code changes require the entire release cycle to be repeated

MicroServices: Software as a Service

- Does your software run as a service?
- Do you run this service yourself? Or does another part of your organization run the service that your team develops?
- Traditional release cycles don't work well
  - Test-release-deploy takes too long
  - May make releases many times per day
- You can be a little more tolerant of bugs discovered after release if you can fix quickly or roll back quickly.
- Get new features and improvements into production quickly.

Microservices: Start with Modularity

- Think about your current application.
- How can you break it into modular component parts?
- What are the interfaces? What are the messages that you need to send between the parts?
- Your code should be modular anyway.
  - OGSI and Spring do this for single JVM Java applications
  - You can maintain a module without touching other code.



Spring Coders, Speak Up!

What components, POJOs, etc do you have in your code now?

Should a person or group “own” a subset of modules or microservices?

For small teams, NO in my experience.

- Team members come and go
- They aren’t always available anyway
- Work-life balance will eventually become important to most
- No one should be indispensable

Module ownership leads to other problems

- “Possessive ownership” culture
- Opaque code
- “Not my problem” if it happens in another module

Ownership and Conway's Law

- *Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure. –Melvin Conway*

In Other Words

- **If** you separate your organization into
  - Systems specialists,
  - DB admins,
  - Software developers,
  - User experience specialists, etc,
- And ask them to build something,
- **Then**
  - the system will look like the org chart.
- This is the opposite of the DevOps approach
- Creates the bureaucratic infighting that plague organizations.

Products, Not Projects

- The alternative: organize around products
  - A product team has all the expertise it needs
  - Jim Gray interviews Werner Vogels: https://queue.acm.org/detail.cfm?id=1142065
  - “You build it, you own it”
- Problems with this approach:
  - Someone may work on multiple products
  - Who is that person’s boss?

My Advice: Own the Entire Product

Even if you aren't the manager. Own the product.

Do you see this in the way you work now?

Or in the way you worked before you came back to school?

How Big Is a Module?

How little is micro?

Rules of Thumb You Might Encounter

- Amazon's two-pizza rule
  - Teams shouldn't be larger than what two pizzas can feed
- "A good programmer should be able to rewrite a microservice in a couple of weeks"
- Bounded Contexts from Domain Driven Design

Can We Make This More Rigorous?

Messaging patterns between components

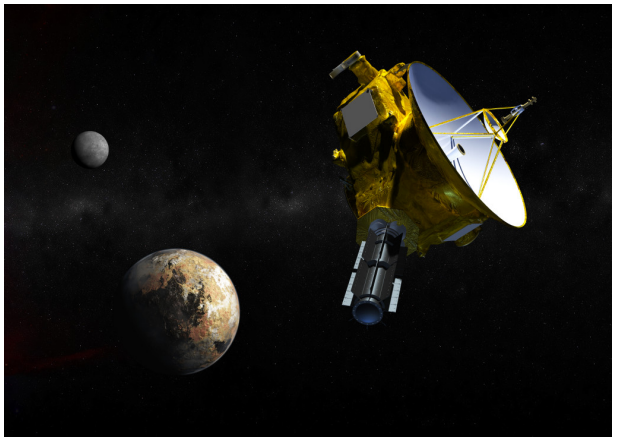
Request-response	•REST •Common in Internet applications but you would never do this within code in a single JVM
Push	•Common in single JVM application: Observer, listener patterns •But not so common in decoupled applications. •More generally, asynchronous messaging is a little scary.
One-to-one	•The component knows which component needs to get the new information.
Many-to-many	•Components don't need to know which other components needs the information. They just broadcast it.

How Big Is a Component?

- One answer: the optimal message pattern should not change when going from single JVM to distributed components.
  - Telescope in and telescope out
- Components that run in the same process can be chatty and send lots of messages.
  - Messages can be pointers to shared data objects
  - Messages are always received
  - This is true for both request and push messaging.

Pan-Galactic Components

- Components that run across the universe need different messages, either infrequent or terse.
  - No shared data objects to point to.
- Distributed components must also assume a wide variety of message failures
  - Lost connections
  - Recipients never received the message
  - Recipients received messages in the wrong order
  - Recipients crashed while processing
  - …
- Microservices are distributed systems
  - Should know and use DS ideas



Microservices Are Not Distributed Objects

- A component is not an object in the OO sense.
- A component should be a service in the SOA sense
- Recall REST architecture: services should be as idempotent as possible.
  - If you send the same message more than once, the server state after processing the message is the same.
  - Ex: If a component receives 10 identical message saying submit a job, the component should only submit the job once.
    - How do you know the message is really identical? Timestamps, other identifiers, etc
- "Shared nothing" between services
- Message processing should have no side effects that need to be exposed to other components
  - Anti-pattern: all those output files generated by scientific applications

Decentralized Data Management

This is an open challenge for Apache Airavata

- The Registry component, backed by a single DB system, holds all the data

Microservice Antipattern

- One database has all the information for the application
- Limits your choice of data stores: you have to use the same DB technology for everything.
- Microservices constantly need to make remote DB calls
- Performance and security

Setting Data Boundaries

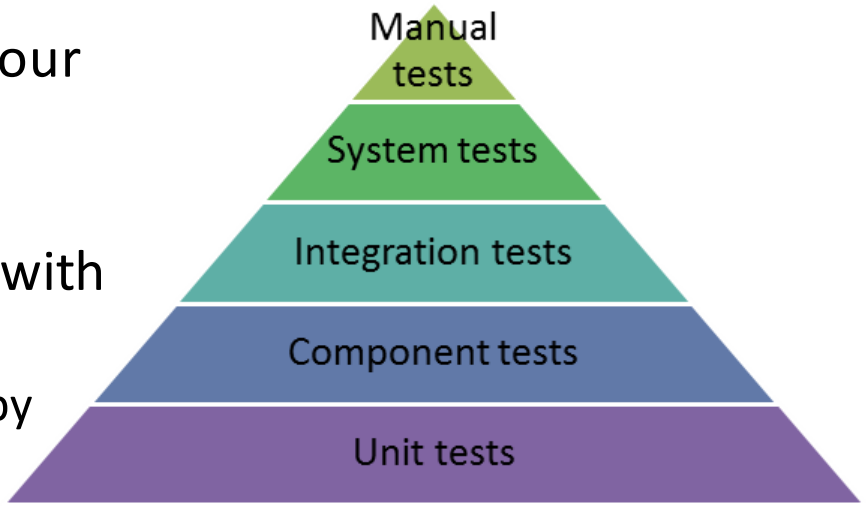
- This is easy when the data isn't connected or lightly connected.
- But not easy to split up a traditionally designed DB schema.
- Science gateway data boundaries
  - You may want to split stateless and stateful data
- If you split even lightly connected data across services, you will need to consider consistency
  - Strong: requires distributed transactions, which can be tricky
  - Weak: eventual consistency. Can you live with this?
- **What data boundaries do science gateways have?**

How do services communicate?

- Ideally, always do the same thing.
  - Your development environment should be a miniature version of your deployment environment: containerization
  - Encapsulate and abstract
  - This touches DevOps CI/CD, so those are separate lectures
- Keep the communication patterns the same
  - Push, pull
  - 1-1, or N-N
- We'll look at messaging systems in detail next
  - Push messaging, many-to-many, reliable delivery, replay, etc

If I make everything distributed, how can I test?

- Unit tests within components are simple
- But testing the entire system becomes an operational test
- This is not a bad thing if you can script your operation tests
  - And you should
- Monolith: test your code while building with Maven
  - Bring up simple test DBs like SQLite or Derby
  - One big build does everything



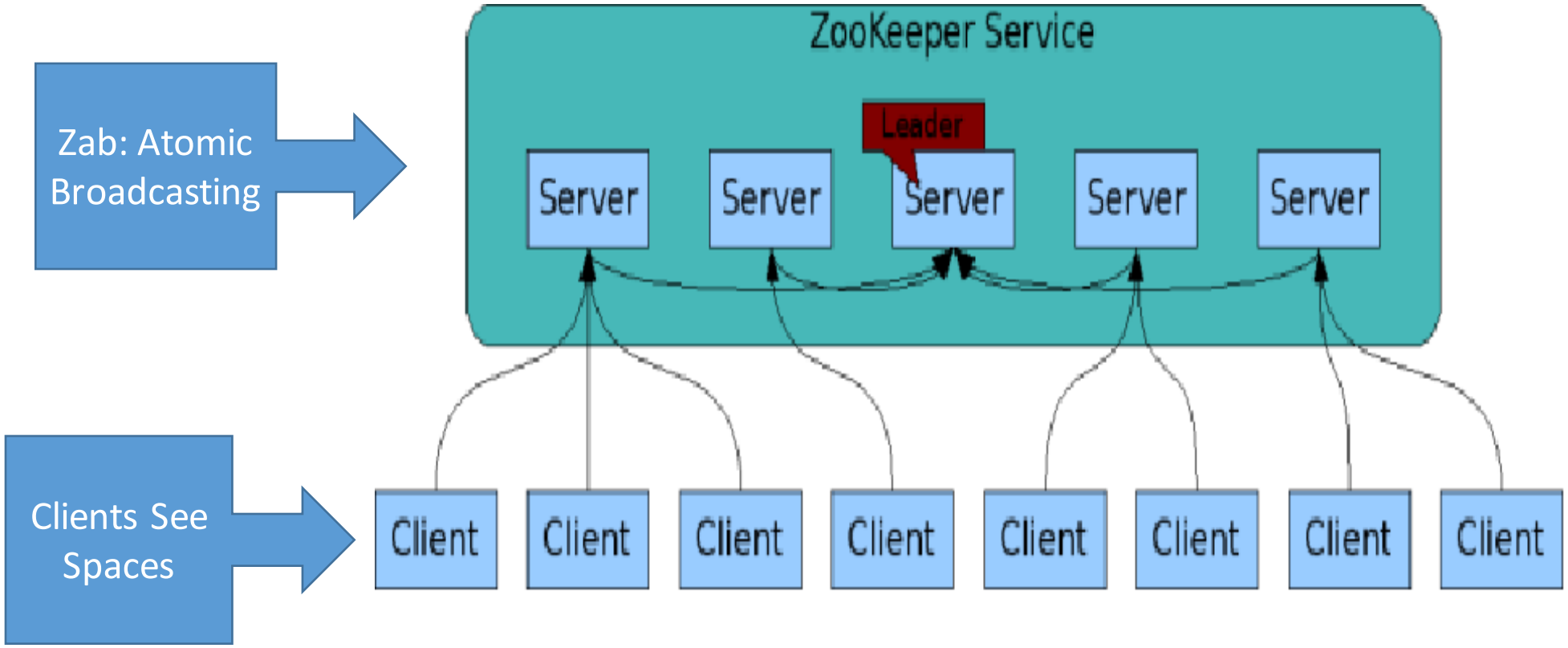
Microservice Testing

- Microservice: test your code in an operation-like setting
  - Don’t build the entire code every time
  - Build only the microservice you are working on.
  - So your build should be modular
- So maybe you don’t need one big build system after all
  - And maybe you don’t need one git repo for everything...
  - And maybe you don’t need one version # that covers all your code...
- So releases become incremental
- Testing, monitoring, and logging become more continuous
  - This is a touch point between Microservices and DevOps

Designing for Failure

- Microservices need to be built to tolerate errors.
- What kinds of errors will science gateway microservices encounter?
- How do you detect these failures?
- What tools do you need?
- What strategies can you use to do system testing?

Messaging in Distributed Systems



Zookeeper's client API resembles a structured version of the Tuple Spaces concept of parallel programming.

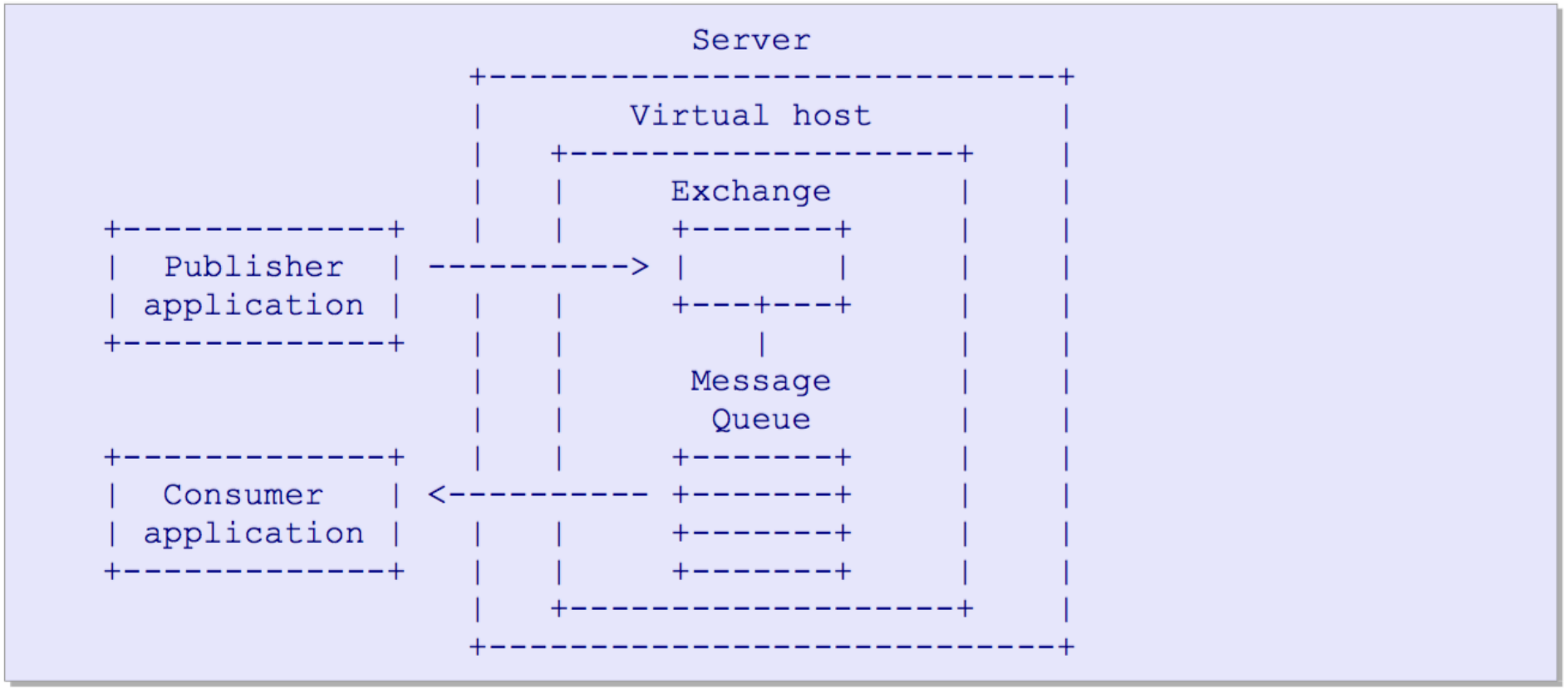
Advanced Message Queuing Protocol Overview

More Distributed Service Concepts

AMQP: Network Protocol and Architecture, Not An API

- Many Implementations
  - RabbitMQ
  - Apache ActiveMQ
  - Apache Qpid
  - SwiftMQ
  - And others:
  - This contradicts my earlier distinction between horizontal and vertical standardization efforts
    - At least for Version 0-9-1
- Each implementation can have its own API
- I'll focus on Version 0-9-1
- This is not an AMQP tutorial

Basic Concepts



An AMQP Server (or Broker)

Two main parts	• Exchange • Message Queue • Exchanges can interact with multiple message queues
Exchange	• Accepts producer messages • Sends to the Message Queue
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

Message Queue

- Types of queue properties:
  - Private or shared
  - Durable or temporary
  - Client-named or server- named, etc.
- **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.

Consumers and Message Queues

- AMQP Consumers can create their own queues and bind them to Exchanges
  - This has an interesting implication
- 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.
  - This greatly simplifies things
- If you need > 1 consumer to receive a message, you can give each consumer their own queue.
  - Use topic matching to route messages
- Compare this to Zab and atomic broadcast protocols, generally.
  - AMQP makes some simplifications

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

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.*

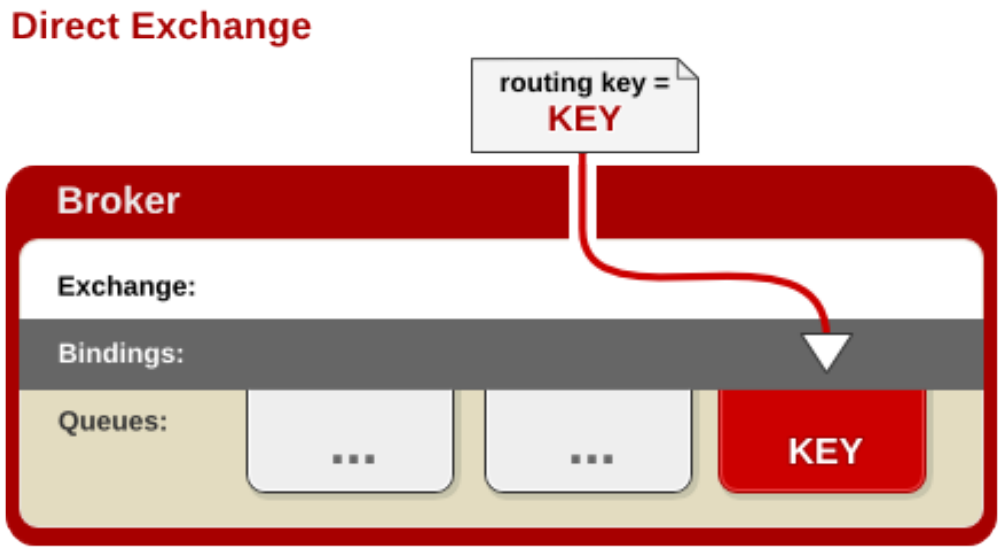
The Message Payload

- Read the specification for more details.
- In general AMQP follows the head-body format
- The message body payload is binary
- AMQP assumes the content is handled by consumers
  - It is the consumer's job to do something with the content.
  - Unlike JMS
- You could serialize your content with JSON or Thrift and deserialize it to directly send objects.

Message Exchange Patterns

Direct Exchange

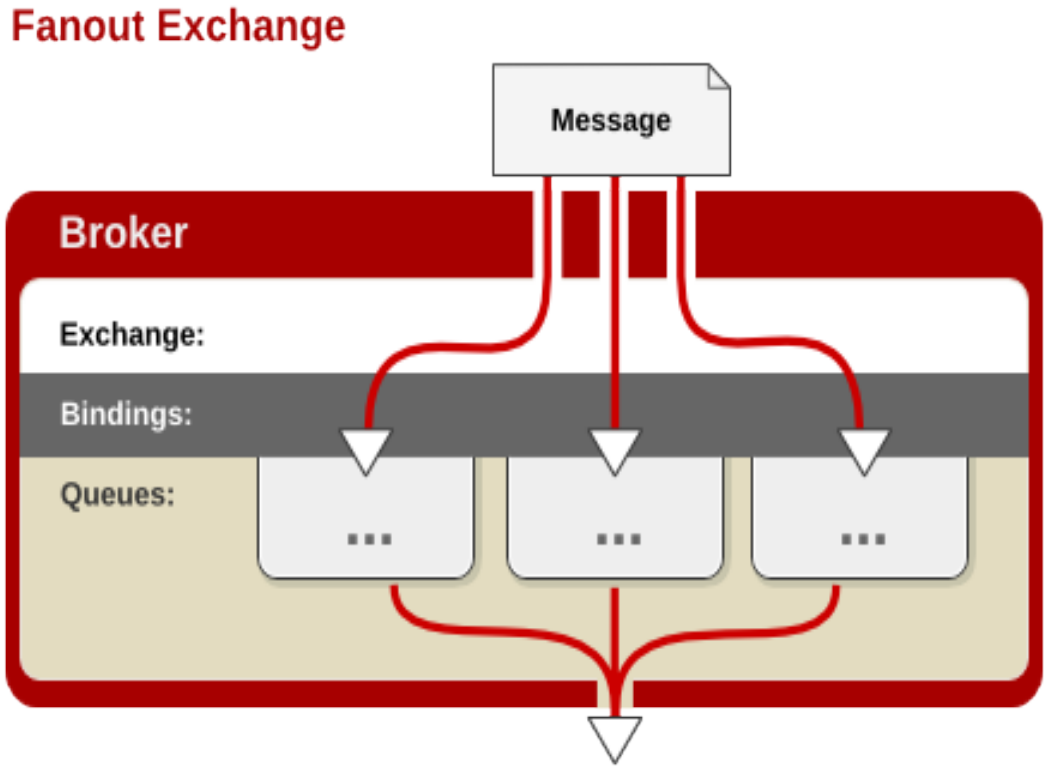
- A publisher sends a message to an exchange with a specific routing key.
- The exchange routes this to the message queue bound to the routing key.
- One or more consumers receive messages if listening to the queue.
- Default: round-robin queuing to deliver to multiple subscribers of same queue




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

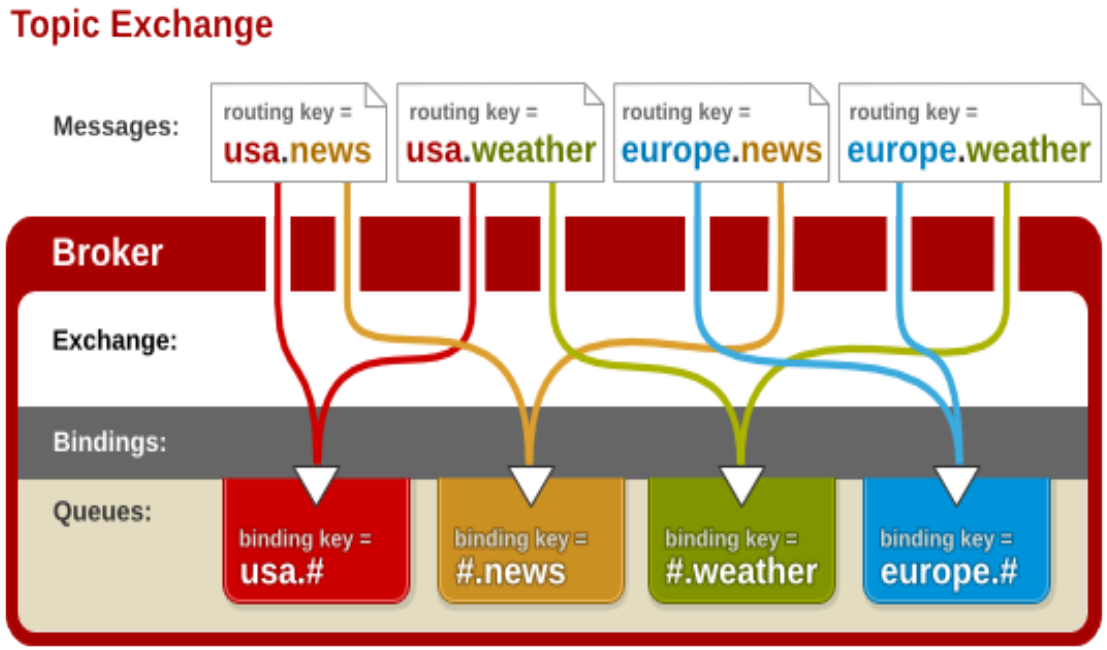
Fanout Exchange

- 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

- The RabbitMQ tutorial page has several nice examples of classic message exchange patterns.
- https://www.rabbitmq.com/getstarted.html

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
  - Multiple recipients OK
- 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?

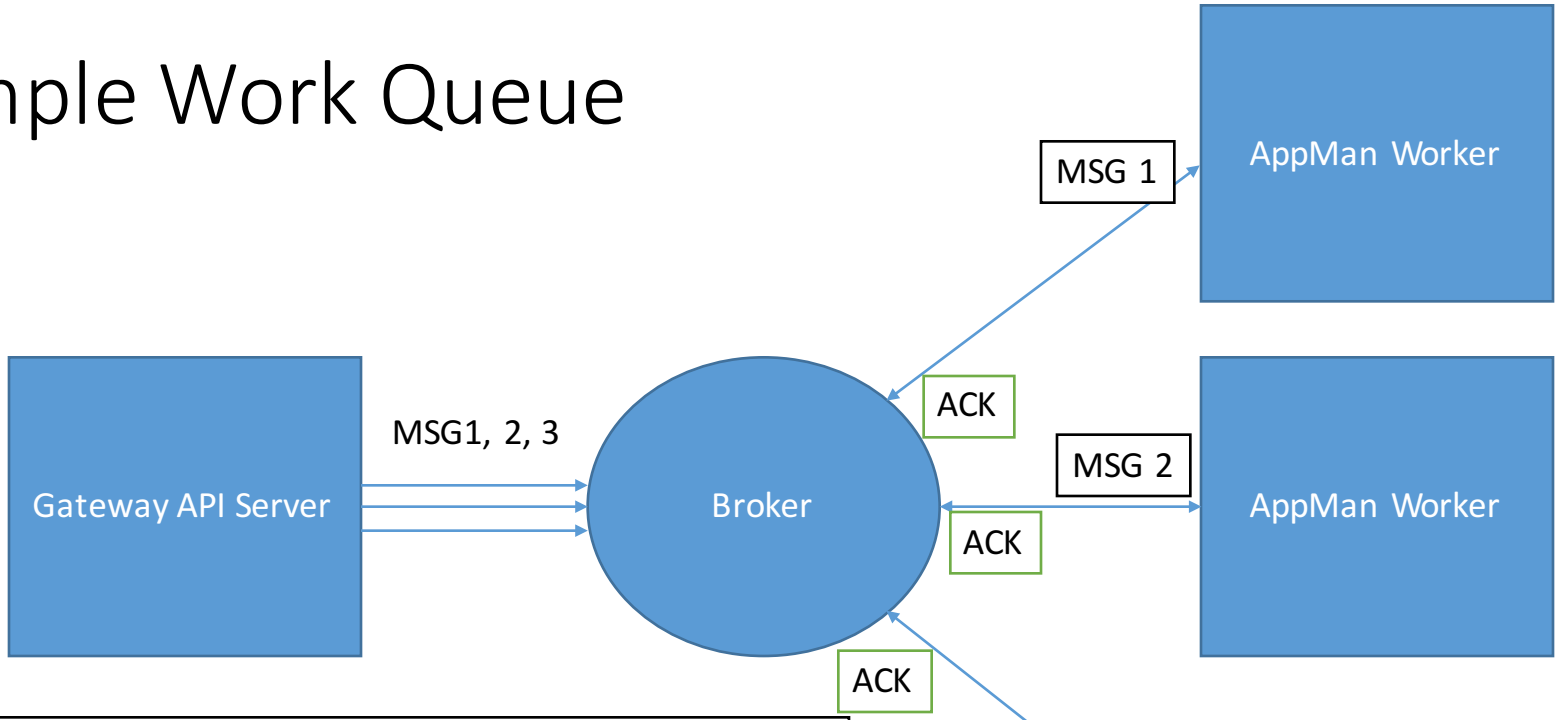
- AMQP does not cover all the capabilities listed above.
  - It can be extended to cover these in many cases
- AMQP messaging system implementations are not necessarily cloud-ready
  - They have to be configured as highly available services.
    - Primary + failover
  - No fancy leader elections, etc as used in Zookeeper + Zab
  - 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

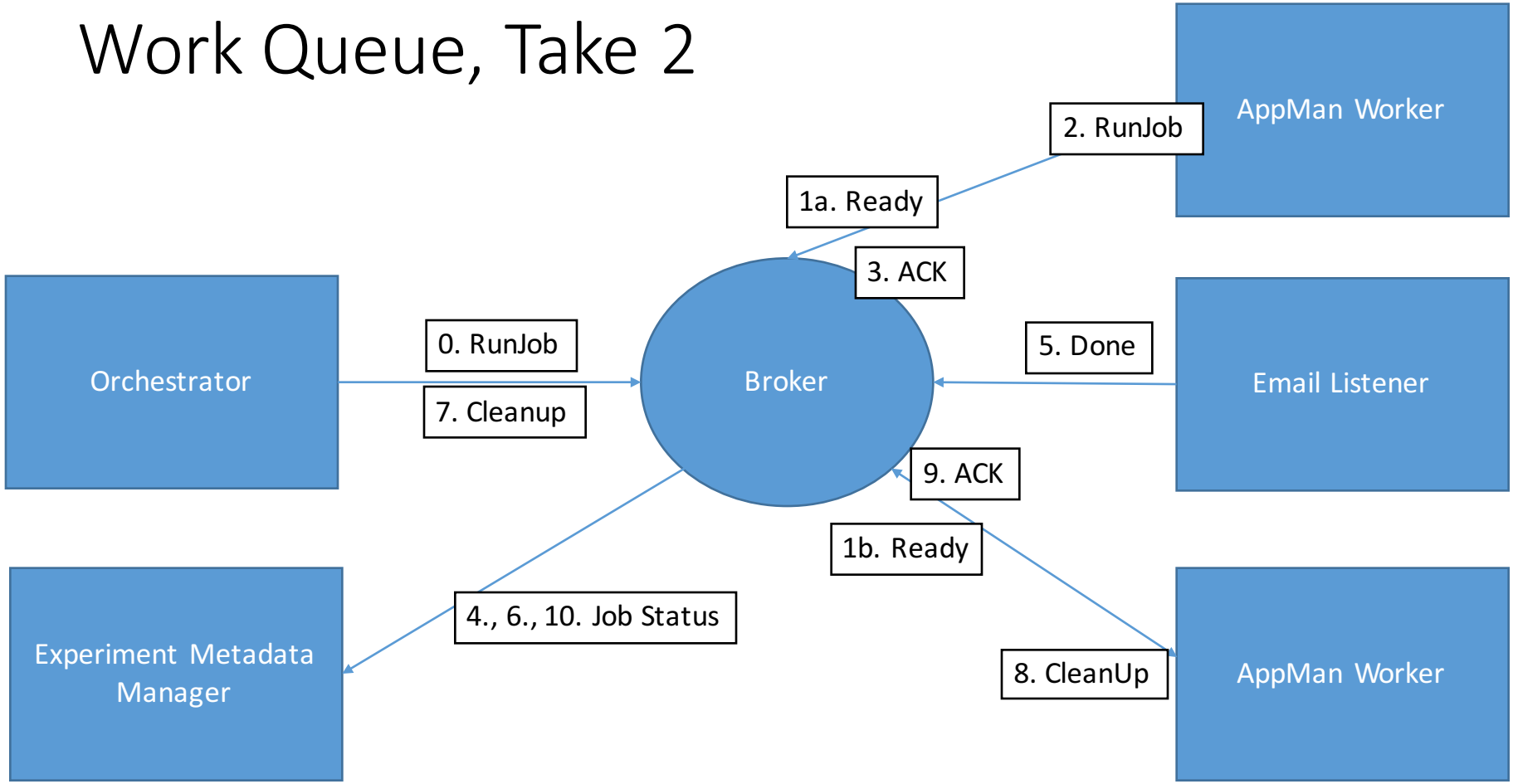
What Could Possibly Go Wrong?

- 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?

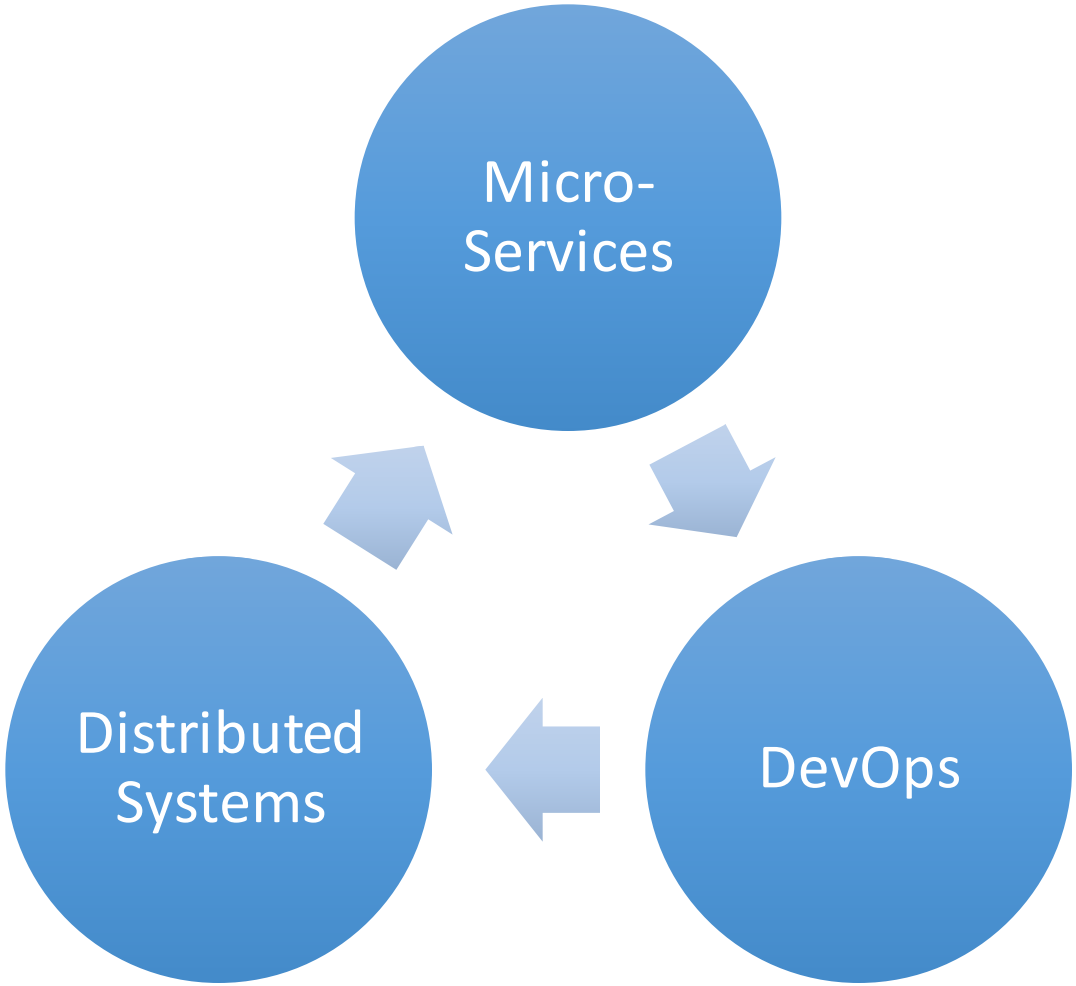
- A Worker may not be able to submit the job
  - Remote supercomputer is unreachable, for example
  - We need a NACK
- The Orchestrator and Experiment Metadata components are also consumers.
  - Should send ACKs to make sure messages are delivered.
- Orchestrator and Experiment Metadata Manager may also 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

- Microservices are applications built out of distributed components
- Important if you run Software as a Service
- Connection to DevOps:
  - Accelerate the develop-test-deploy cycle
  - Continuous Integration and Continuous Delivery are essential
- Services need to communicate
  - Asynchronous messaging is powerful mechanism
  - Push, many-to-many
  - Can easily extend to add more components


Micro-
Services
DevOps
Distributed
Systems