



Designing Distributed Systems

Reading Notes by [Euccas](#), 2018

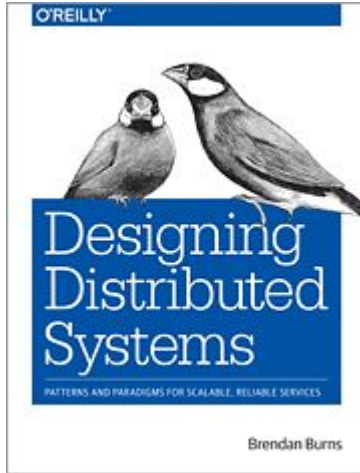


Reading Log

- 3/29/18 2H
- 3/30/18 1H
- 4/25/18 3H
- 4/26/18 3H
- 4/27/18 2H
- 5/9/18 2H
- 5/13/18 1H

Total: 14H

The Book



- Designing Distributed Systems
 - Patterns and Paradigms for Scalable, Reliable Services
 - 166 pages
- Author
 - Brendan Burns
 - Microsoft Azure Engineering
- Publisher
 - O'Reilly
 - Released in Feb 2018
- Materials
 - Example code
 - <https://github.com/brendandburns/designing-distributed-systems>



Contents

- Introduction
 - History of systems development, patterns in software development, and why patterns
- Single-Node Patterns
 - Sidecar
 - Ambassadors
 - Adapters
- Serving Patterns
 - Microservices
 - Replicated Load-Balanced Services
 - Sharded Services
 - Scatter/Gather
 - Functions and Event-Driven Processing
 - Ownership Election
- Batch Computational Patterns
 - Work Queue Systems
 - Event-Driven Batch Processing
 - Coordinated Batch Processing



Why do I read this book?

My Questions


- What's the most important difference between **designing distributed systems** and **single machine systems**?
- Why container technology (docker, kubernetes) is so popular? Is it really useful?
- What are the common patterns and how can they help (when do I need them)?

Some topics I haven't read thoroughly

- How to design a particular pattern (design guidelines)
- Hands on exercises


Q & A





What's the most important difference between designing distributed systems and single machine systems?

- Designing distributed systems can be significantly more complicated to design, build, and debug correctly
- Designing distributed systems need much more efforts in designing for scalability and reliability
- In a distributed system, tasks/data are spreaded to multiple workers. It requires techniques like containers and load balancing to utilize parallelisation



Why **container technology** (docker, kubernetes) is so popular? How could they be helpful?

- Containers are not only useful for applications which have components running on **multiple machines**, but also for **single machine** applications
- The goal of containerization is to **establish boundaries** around specific resources, team ownership, separation of concerns
- The benefits include **resource isolation**, **scaling teams**, **reuse components and modules**, **break big problems into smaller ones** (Small, focused applications are easier to understand, be tested, updated and deployed)

Reading Notes






What's going on in Software Development?

History of System Development

- Purpose-built machines for specific purposes
- Networked machines, client-server architectures
- Early 2000s, Internet, large-scale data centers, distributed systems, container technology

History of Software Development Patterns

- Formalization of algorithmic programming: 1962, Donald Knuth, *The Art of Computer Programming*
- Patterns for Object-Oriented programming: early to mid-1990s, *Design Patterns*
- Open Source Software: late 1990s and the 2000s, open source communities



Why need patterns in Software Development?

- General **blueprints** for system design, without mandating any specific technology or application choices
- Learn from the **mistakes of others**
- A shared language, vocabulary for discussing and understanding each other quickly
- Identify common components for easy reuse



Why need containers on a single node?

On a single machine, why do you might need break your application into different containers?

What's the purpose of containers?

- **Establish boundaries** around specific resources
 - this application needs two cores and 8GB of memory
- Establish boundary that delineates team ownership
 - this team owns this image, service
- Establish boundary that provide separation of concerns
 - this image does this one thing

Your considerations

- Resource isolation
 - separate a user-facing service and a background service
- Scale a team
 - have small, focused pieces for each team to own
- Reuse components, modules
 - a Git sync tool reused with PHP, HTML, JavaScript, Python
- Small, focused applications are easier to understand, be tested, updated and deployed



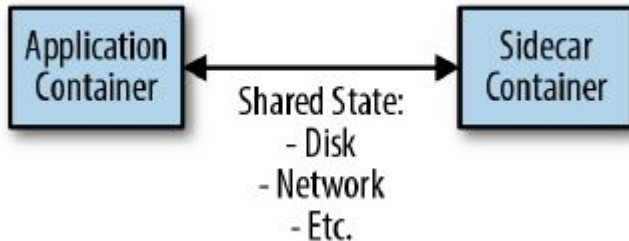
Single-Node Patterns

- Containers exist on the same machine
 - They depend on **local, shared resources** like disk, network interface, or interprocess communications
- Sidecar
 - Ambassadors
 - Adaptors

What is Sidecar pattern?

- a **single-node** pattern made up of **two containers**
 - *application container* core logic for the application
 - *sidecar container* augment and improve the application container
- used to **add functionality** to a container that might otherwise be difficult to improve
- on the same machine, share a number of resources (filesystem, hostname, network, etc.)

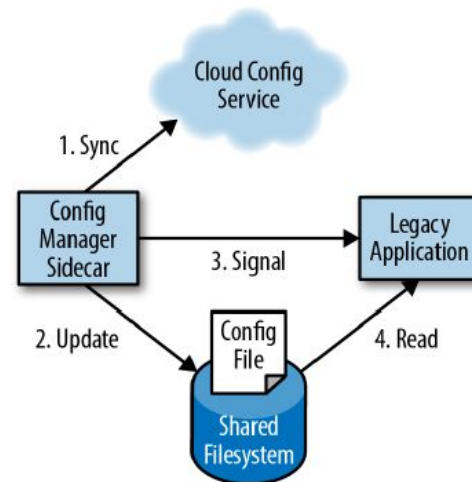
Container Group (aka Pod)



Why do I need use ?

- #1 Improve/Adapt legacy applications where you no longer wanted to make modifications to the original source code
 - Add HTTPS to a legacy web service
 - Enable dynamic configuration in an existing application

Container Group (aka Pod)



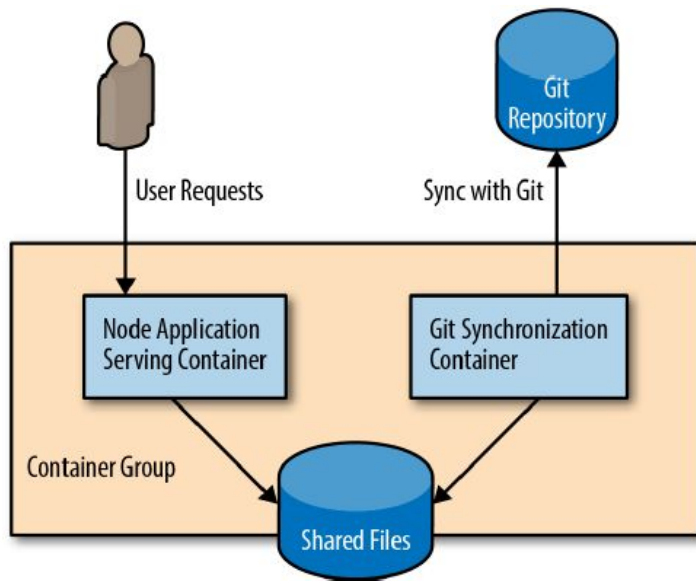


Why do I need use Sidecar?

- #2 Modularity and reuse of the components used as sidecars
 - Implement a topz that provides a readout of resource usage, for multiple languages
- Trade-offs between the modular container-based pattern (library-based approach) and rolling your own code into your application
 - Performance
 - API may require some adaptation

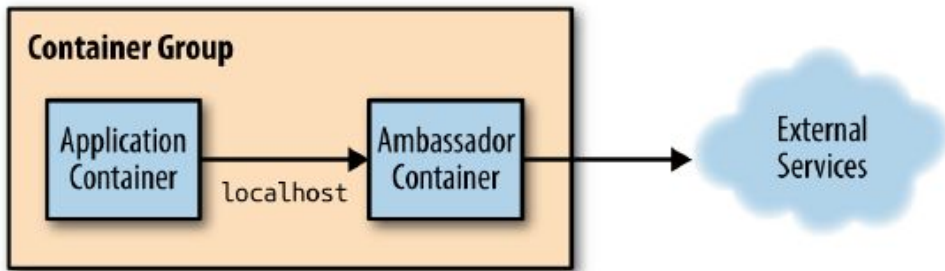
Why do I need use Sidecar?

- #3 More than adaptation and monitoring, implement the complete logic for your application in a simplified, modular manner
 - Build and deploy a simple PaaS built around the git workflow
 - main container: a Node.js server that implements a web server
 - sidecar container: shares a filesystem with the main application container, and runs a simple loop that synchronizes the filesystem with an existing Git repository



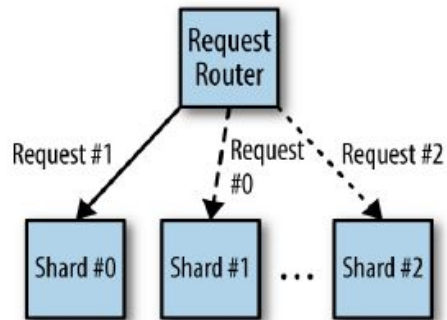
What is **Ambassadors** pattern?

- Two containers on a single node
- An ambassador container **brokers interactions (communications)** between the application containers and the rest of the world
 - Two containers **tightly linked** in a symbiotic pairing in a single machine



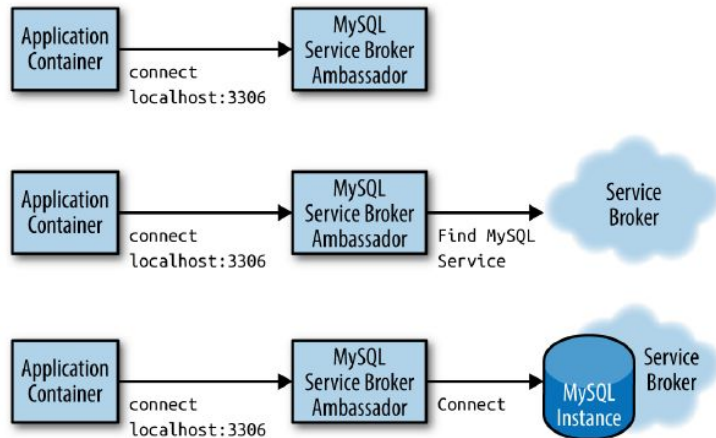
Why do I need use Ambassador?

- #1 Shard a service
 - Sharding: split into multiple disjoint pieces, each hosted by a separate machine
 - Shard your storage layer, when data you want to store becomes too big for a single machine to handle
- Approaches
 - Build sharding logic into the sharded service itself
 - Integrate a single-node ambassador on the client side to route traffic to the appropriate shard
 - Choose by making trade-offs



Why do I need use Ambassador?

- #2 Use an ambassador for service brokering
 - When trying to render an application portable across multiple environments (e.g., public cloud, physical datacenter, or private cloud), one of the primary challenges is service discovery and configuration
 - A portable application requires that the application know how to introspect its environment (eg. find the appropriate MySQL service to connect to). This process is called *service discovery*, and the system that performs this discovery and linking is commonly called a *service broker*
- Use ambassador pattern to separate the logic of the application container from the logic of the service broker ambassador
 - Ambassador: introspect the environment and find the MySQL service
 - Application: always connects to an instance of the MySQL service running on localhost





Why do I need use Ambassador?

- #3 Perform experimentation or other forms of request splitting
 - Perform experiments with new, beta versions of the service to determine if the new version of the software is reliable
 - [euccas] Such as 2% -> 10% -> 50% -> 100% release process
 - Example
 - Kubernetes: define an “experiment” service, and a “web” service
 - config Nginx, redirect 10% of the traffic to the experiment, send 90% of the traffic to the main existing application
 - deploy nginx itself as the ambassador container within a *pod* (a resource unit in Kubernetes)

```
worker_processes 5;
error_log error.log;
pid nginx.pid;
worker_rlimit_nofile 8192;

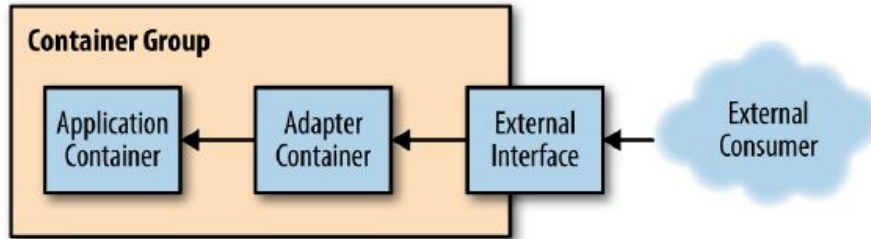
events {
    worker_connections 1024;
}

http {
    upstream backend {
        ip_hash;
        server web weight=9;
        server experiment;
    }

    server {
        listen localhost:80;
        location / {
            proxy_pass http://backend;
        }
    }
}
```

What is **Adapters** pattern?

- Two containers on a single node
- Adapter container is used to **modify the interface** of the application container so that it conforms to some predefined interface that is expected of all applications.
 - For example, a consistent monitoring interface





When do I need use Adapters Pattern?

- Monitoring
 - The adapter container contains the tools for transforming the monitoring interface exposed by the application container into the interface expected by the general purpose monitoring system
- Logging
 - Redirect outputs, transform data format
 - Why not simply modify the application container itself?
 - in many cases we are reusing a container produced by another party

Serving Patterns

- Microservices
- Replicated Load-Balanced Services
- Sharded Services
- Scatter/Gather
- Event Driven FaaS Patterns



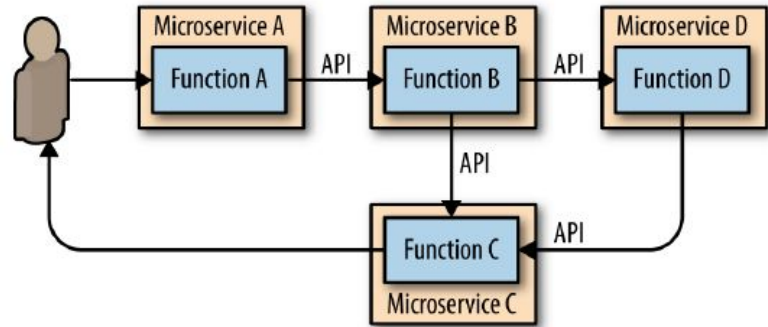
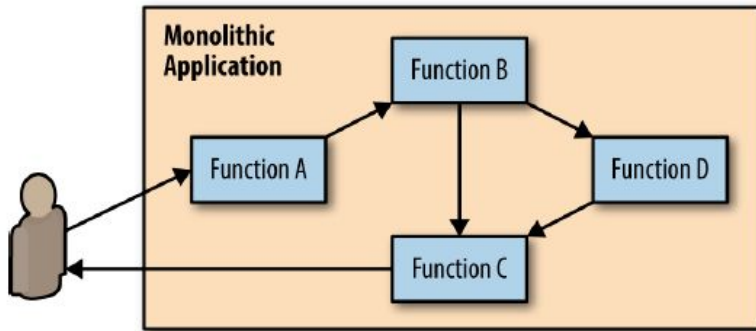



Serving Patterns

- **Multi-node** distributed patterns are more **loosely** coupled
- Components communication is based on **network calls**
- Furthermore, many calls are issued in **parallel**, and systems coordinate via loose synchronization rather than tight constraints

Microservices

- Microservices: a buzzword for describing **multi-node distributed software architecture**
- It describes a system built out of **many different components** running in different **processes** and communicating over **defined APIs**.
- In contrast to: Monolithic systems, which tend to place all of the functionality for a service within a single, tightly coordinated application



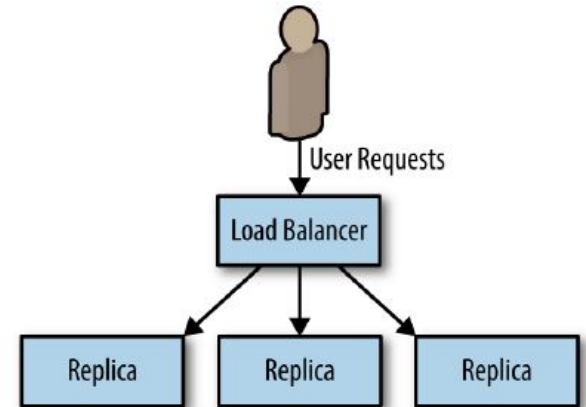


What's the benefits to the microservices approach?

- Benefits
 - Reliability
 - Agility
 - Reduced scope, reduced team size, reduced overhead
 - Decouple the teams, enables teams to independently manage their code and release schedules, which in turn improves each team's ability to iterate and improve their code
 - Scaling
 - Each component is in its own service, and can be scaled independently
- Downsides
 - Difficult to Debug
 - Difficult to Design and Architect
 - Use well-known design patterns to make the design easier

Replicated Load-Balanced Services

- A replicated load-balanced service is the simplest distributed pattern
 - Every server is **identical** to every other server and all are capable of supporting traffic
 - The pattern consists of a scalable number of servers with a load balancer in front of them
 - Load balancer: typically **round-robin** or use some form of **session stickiness**
- Stateless services
 - Systems are replicated to provide redundancy and scale
 - Need at least **two replicas** to provide a service with a “**highly available**” service level agreement
 - Build and deploy a **readiness probe** to inform the load balancer when the container is **ready** to serve



Replicated Load-Balanced Services, cont'd

- Session tracked services

- Session tracked services ensure that all requests for a single user map to the same replica. It's an adaption of the stateless replicated service pattern.
- Ensure a particular user's requests always end up on the same machine when you (examples)
 - Cache that user's data in memory
 - Interaction is long-running and some states is maintained between requests

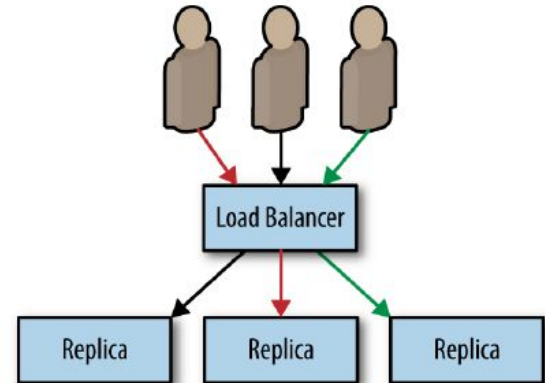
IP-based session tracking:

Session tracking is performed by **hashing the source**

and **destination IP addresses**

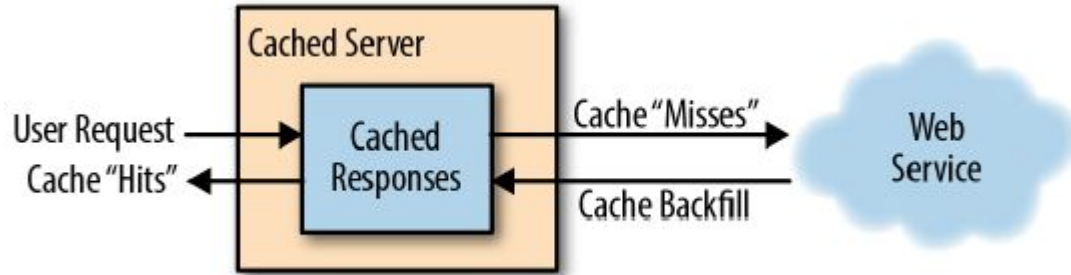
and using that key to identify the server

- Application-level tracking:
eg. via cookies



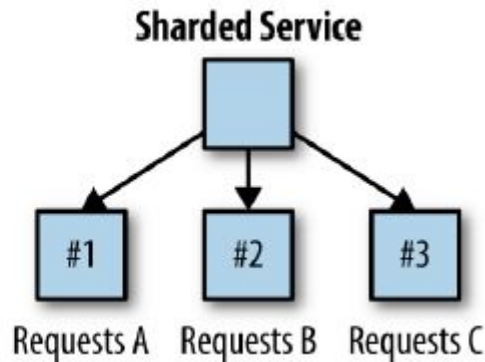
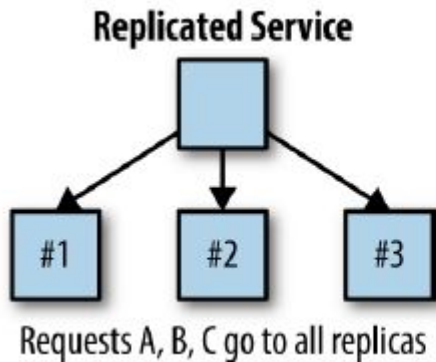
Replicated Load-Balanced Services, cont'd

- Caching layer
 - Eg. use Varnish, an open source web cache



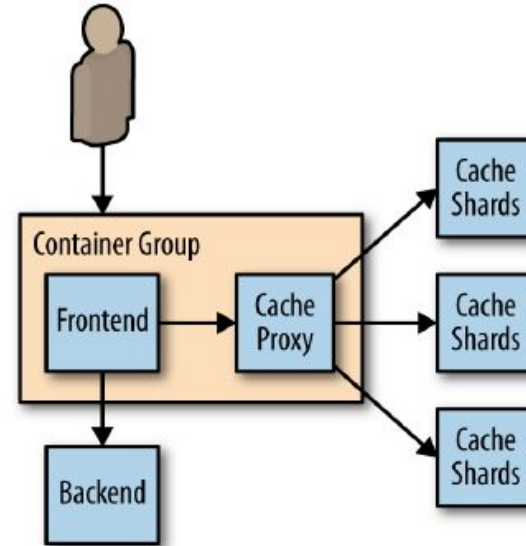
Sharded Services

- Sharding is useful when considering any sort of service where there is **more data than can fit on a single machine**.
- A shard is only capable of serving a **subset** of all requests
- A **load-balancing node**, or *root*, is responsible for examining each request and distributing each request to the appropriate shard or shards for processing
- Sharded services are generally used for building **stateful** services
 - The primary reason for sharding the data is because the **size of the state is too large** to be served by a single machine



Sharded Caching

- A sharded cache is a cache that sits **between the user requests** and the actually **frontend implementation**
- **How to evaluate the cache performance, and system performance with/without the cache**





Sharding Functions

- How traffic is routed to different shards? e.g. Given a user request *Req*, how do you determine which shard *S* in the range from 0 to 9 should be used for this request?
- A Sharding Functions relates this *mapping*
 - a sharding function is very similar to a *hashing function*
 - $Shard = ShardingFunction(req)$
 - Commonly, the sharding function is defined using a *hashing function* and the *modulo % operator*
 - The most important characteristics
 - *Determinism*
 - Ensures a particular request *R* always goes to the same shard in the service
 - *Uniformity*
 - Ensures the load is evenly spread between the different shards
 - Important: use *Consistent Hashing Functions* to resolve the problem *caused by #shards changes*
 - Consistent hashing functions are special hash functions that are guaranteed to *only remap # keys / # shards, when being resized to # shards*

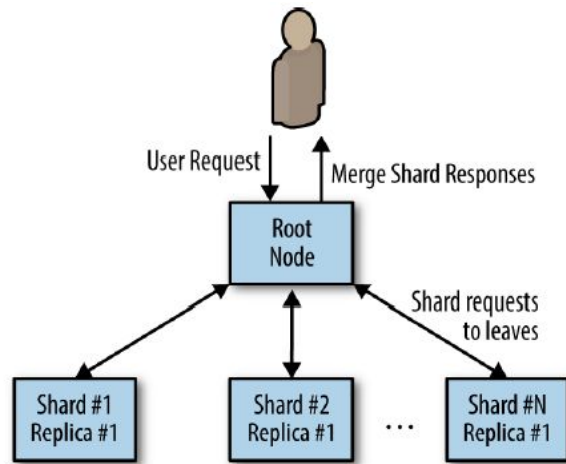


Hot Sharding Systems

- Ideally the load on a sharded cache will be perfectly even, but in many cases this isn't true and “hot shards” appear because organic load patterns drive more traffic to one particular shard.
- As an example of this, consider a sharded cache for a user's photos; when a particular photo goes viral and suddenly receives a disproportionate amount of traffic, the cache shard containing that photo will become “hot.” When this happens, with a replicated, sharded cache, you can scale the cache shard to respond to the increased load.

Scatter/Gather


- Scatter/Gather pattern uses replication for scalability in terms of **time**
 - Replicated Load-Balance and Sharding uses replicate for scalability in terms of **the number of requests** processed per second
- Scatter/Gather pattern allows you to achieve **parallelism** in servicing requests
 - Enabling you to service them significantly faster than you could if you had to service them sequentially
- A tree pattern with a **root** that **distributes** requests, and **leaves** that **process** those requests
- Requests are **simultaneously** farmed out to all the replicas in the system
 - Each replica does a small amount of processing and then returns a fraction of the result to the root
 - The root server combines the various partial results to form a single complete response to the request
 - Root server then sends the request back out to the client
- Useful for processing a large amount of mostly **independent processing** that is needed to handle a particular request





Functions and Event-Driven Processing

- Emerging architectures for event-driven applications, such as FaaS (Function-as-a-service) services and products
 - For example, applications that might only need to temporarily come into existence to handle a single request
 - FaaS: an event-based application model
 - FaaS: oftentimes is referred to as *serverless* computing. But FaaS is more about event-driven, and it is different from the broader notion of serverless computing.
- The Benefits of FaaS
 - Primary benefits are for the developer. It simplifies the distance from code to running service. FaaS makes it simple to go from code on a laptop or web browser to running code in the cloud
 - The deployed code is managed and scaled automatically. If a function fails due to application or machine failures, it is automatically restarted on some other machine.
 - Functions, like containers, are even more granular building block for designing distributed systems
 - Functions are stateless and thus any system you build on top of functions is inherently more modular and decoupled than a similar system built into a single binary
 - The decoupling is both a strength and a weakness (the challenges of FaaS)

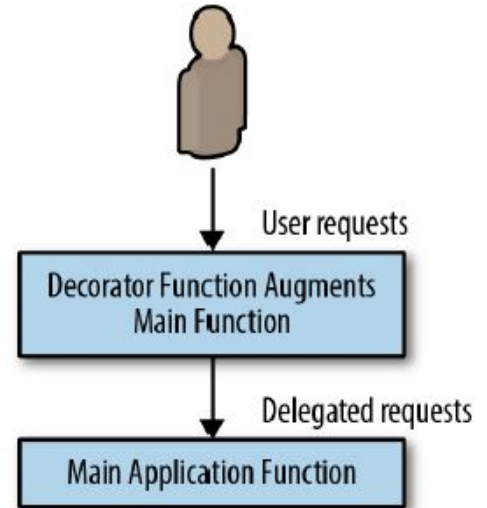


Functions and Event-Driven Processing, cont'd

- Challenges of FaaS
 - Extra complications caused by forced Decoupling
 - Communication between functions
 - Each function instance cannot have local memory, requiring all states to be stored in a storage service
 - Difficult to obtain a comprehensive view of your service
 - Difficult to debug

Patterns for FaaS - Decorator

- **Decorator** Pattern: Request or Response Transformation
- Take an input, transform it into an output, then pass it on to a different service
 - Eg. decorate HTTP requests to or from a different service
- A close analogue: the *decorator* pattern for Python





Patterns for FaaS - Handling Events

- Most systems are **request driven**, handling a steady stream of user and API requests
- Many other systems are more **event-driven** in nature
- The **differentiation** have to do with the notion of **session**
 - Requests are part of a larger series of interactions or sessions; generally each user request is part of a larger interaction with a complete web application or API.
 - Events instead tend to be single-instance and asynchronous in nature.

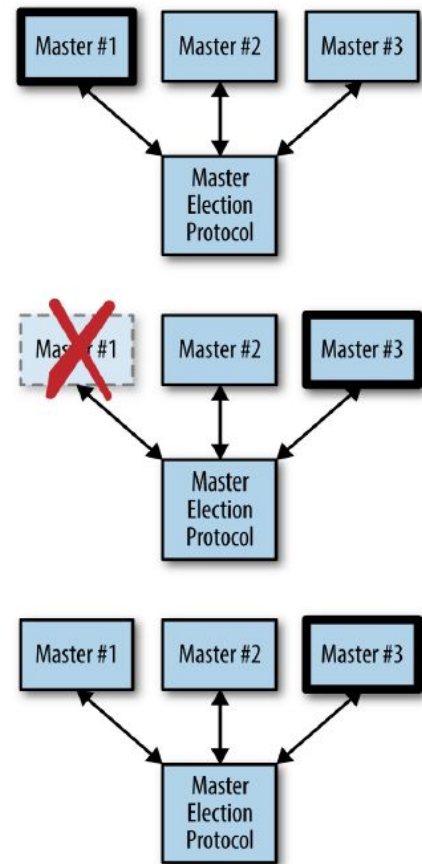


Patterns for FaaS - Event-Based Pipelines

- Event-Based pipelines can be represented as a **directed graph** of **connected event syncs**
- In the pipeline, each **node** is a different **function** or **webhook**, and the **edges** linking the graph together are **HTTP** or **other network calls** to the function/webhook.

Ownership Election

- How you scale assignment in multi-node serving pattern
- Often, **establishing distributed ownership** is both the most complicated and most important part of designing a reliable distributed system
- First, determine if you even need master election
- Master election protocol example





Master Election

- Master election: the process of how the master is selected as well as how a new master is selected if that master fails
- Two ways to implement master election
 - Implement a distributed consensus algorithm (like Paxos, RAFT). But this approach is too complex and not worthwhile to implement
 - Use a distributed key-value store system, such as *etcd*, *ZooKeeper*
 - These systems provide the ability to perform a compare-and-swap operation for a particular key



Implementing Locks

- The simplest form of synchronization is the **mutual exclusion lock** (aka **Mutex**)
- The concept is the same as Mutex used in concurrent programming on a single machine
- Distributed locks can be implemented in terms of the **distributed key-value stores**
- Many key-value stores let you watch for changes instead of polling. You don't have to wait at least a second after the lock is released before you acquire the lock - With traditional lock implementations you have to do this
- **TTL: Time-to-live**
 - Used for a key in the key-value stores. Once the TTL expires, the key is set back to empty
 - When a lock function writes with a TTL, if we don't unlock within a given time, the lock will automatically unlock

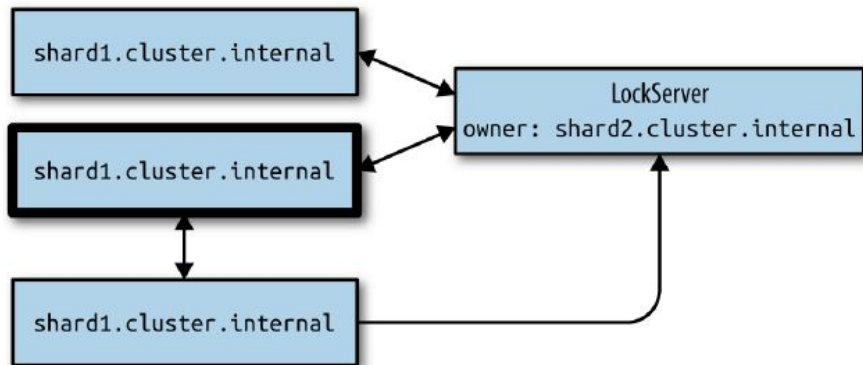


Implementing Ownership

- Locks are not suitable to handle the case when you want to take ownership for the duration of the time that the component is running. For example when the ownership need exist for very long time (say a week or longer)
- **Renewable Lock**: a lock that can be **periodically renewed** by the owner so that the lock can be **retained** for **an arbitrary period** of time
 - eg. renew the lock every $t_{tl}/2$ seconds
 - Also need to implement a *handleLockLost()* function so that it terminates all activity that required the lock in the first place

Handling Concurrent Data Manipulation

- Possible occurrence: Two replicas simultaneously believe they hold the lock for a very brief period of time
- The system that is being called from the replica needs to validate that the replica sending a request is actually still the master
 - To do this, the **hostname** of the replica holding the lock is **stored** in the key-value store in addition to the **state of the lock**



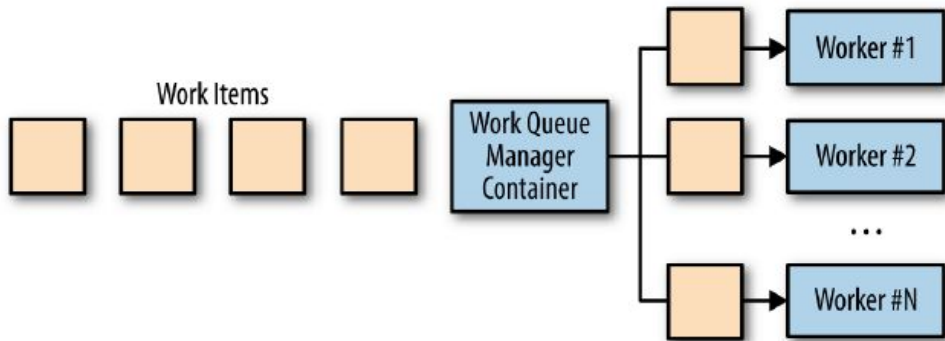


Batch Computational Patterns

- Work Queue Systems
- Event-Driven Batch Processing
- Coordinated Batch Processing

Work Queue Systems

- Work queue: the simplest form of batch processing
 - In a work queue system, there is a batch of work to be performed. Each piece of work is wholly independent of the other and can be processed without any interactions.
 - The goals of the work queue system: ensure each piece of work is processed within a certain amount of time
 - Workers are scaled up or scaled down to ensure that the work can be handled



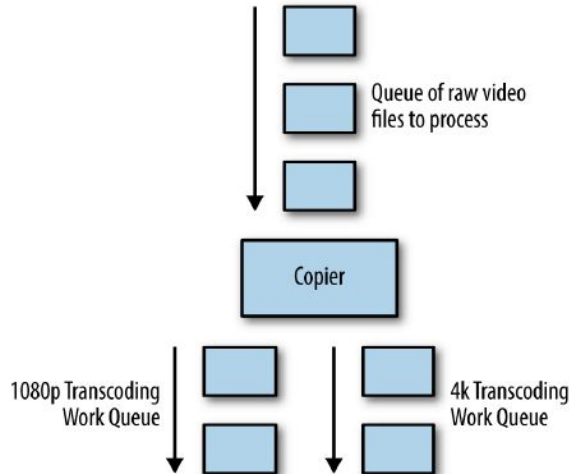


Event-Driven Batch Processing

- Event-driven processing systems: often called *workflow* systems
 - Suitable for performing multiple actions, or generate multiple different outputs from a single data input
 - *Linking work queues together*, the output of one workqueue becomes the input to *one or more* other work queues
- Patterns
 - Copier
 - Filter
 - Splitter
 - Sharder
 - Merger

Event-Driven Batch Processing Patterns

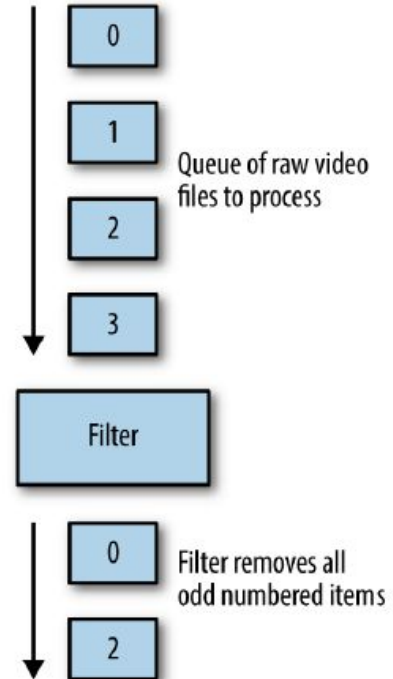
- Copier
 - The job of a copier is to take a single stream of work items and **duplicate** it out into **two or more identical** streams
 - Useful when there are multiple different pieces of work to be done **on the same work item**
 - Example: rendering a video - a variety of different formats as the output, with the same input



Event-Driven Batch Processing Patterns

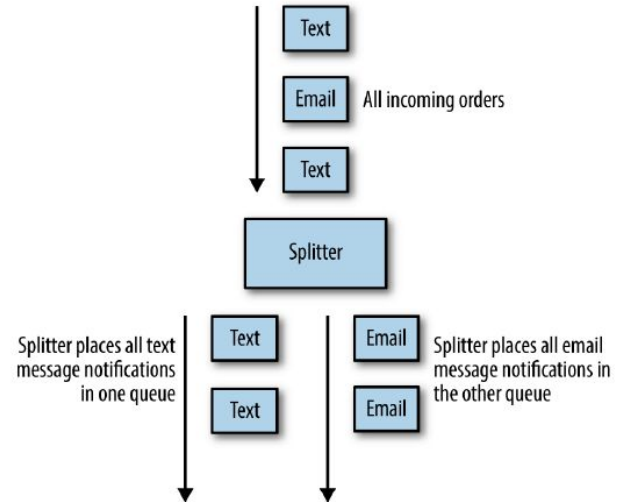
- Filter

- The role of a filter is to **reduce a stream** of work items to a **smaller stream** of work items by filtering out work items that don't meet particular criteria
- Example: Setting up a batch workflow that handles new users signing up for a service.
 - Some set of those users will have ticked the checkbox that indicates that they wish to be contacted via email for promotions.
 - In such a workflow, you can **filter the set of newly signed-up users** to only be those who have explicitly **opted into** being contacted.



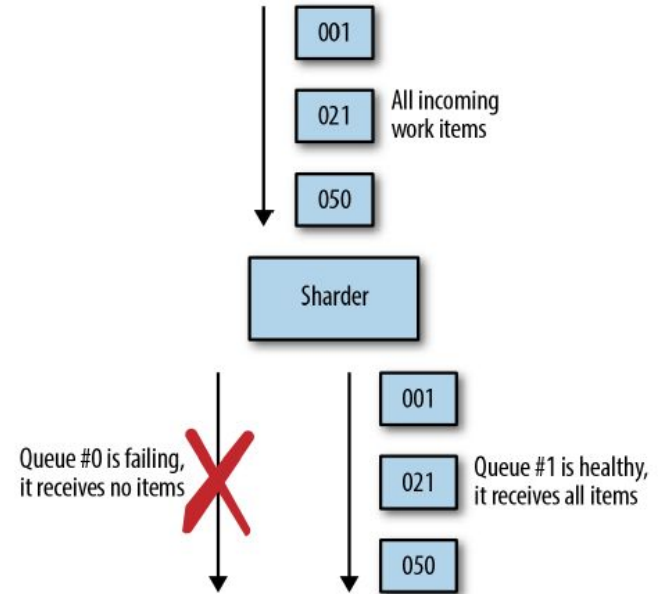
Event-Driven Batch Processing Patterns

- Splitter
 - Use it when you want to **divide** work items to **multiple work queues** without dropping any of them
 - The role of a splitter is to **evaluate some criteria**, but instead of eliminating input, the splitter **sends different inputs to different queues** based on that criteria
 - Example: Processing online orders where people can receive shipping notifications either by email or text message
 - A splitter can also be a copier if it sends the same output to multiple queues



Event-Driven Batch Processing Patterns

- Sharder
 - Sharder is a more generic form of splitter
 - The role of a sharder in a workflow is to **divide up a single queue** into a **evenly divided collection of work items** based upon some sort of **sharding function**
 - Benefits of using a sharder
 - Reliability: The **failure** of a **single** workflow only affects a **fraction** or your service
 - To **more evenly distribute work across different resources**: eg. Use a sharder to evenly spread work across multiple datacenters to even out utilization of all datacenters/regions



Event-Driven Batch Processing Patterns

- Merger

A merger is **the opposite of a copier**

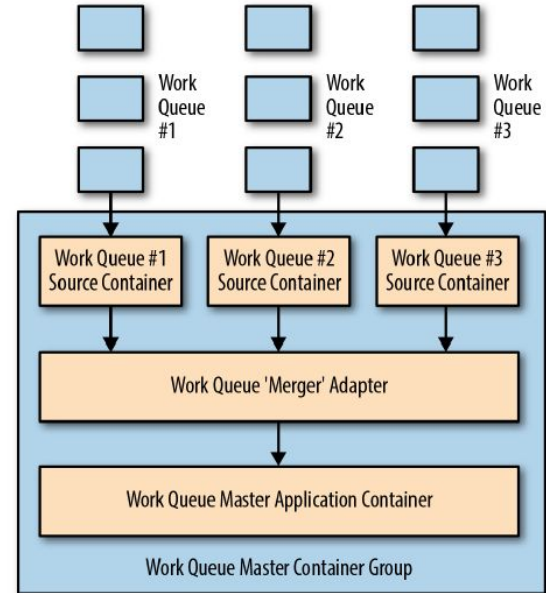
A merger is a great example of the **adapter pattern**

The job of a merger is to **take two (multiple) different work queues** and **turn them into a single work queue**

- Example: *You have a large number of different source repositories all adding new commits at the same time. You want to take each of these commits and perform a build-and-test for it.*

A scalable solution: model each of the different source repositories as **a separate work queue source** that provides a set of commits

Transform all of these different work queue inputs into a single merged set of inputs using **a merger adapter**



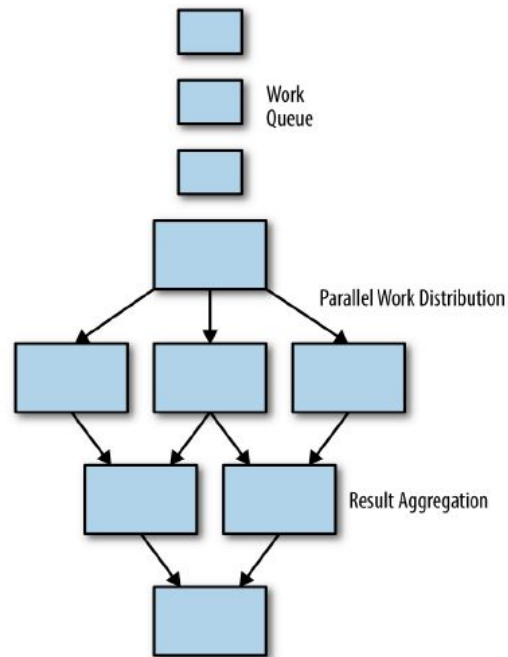


Event-Driven Batch Processing

- Publisher/Subscriber Infrastructure
 - Question: How to manage the stream of data that passes through the event-driven workflow
 - A popular approach: Use a **publisher/subscriber (pub/sub) API** or **service**
 - A pub/sub API allows a user to **define a collection of queues** (sometimes called **topics**)
 - One or more **publishers** **publishes messages** to these queues
 - Likewise, one or more **subscribers** **is listening to these queues** for new messages
 - When a message is published, it is reliably stored by the queue and subsequently delivered to subscribers in a reliable manner
 - Other inefficient alternatives if not using pub/sub API
 - Write each element in the work queue to a particular **directory** on a **local filesystem**, and then have each stage monitor that directory for input - Limited to a single node
 - Introduce a **network filesystem** to distribute files to multiple nodes - Introduces increasing complexity both in our code and in the deployment of the batch workflow
 - Most **public clouds** feature a pub/sub API
 - Azure's EventGrid
 - Amazon's Simple Queue Service
 - Kafka provides a very popular pub/sub implementation that you can run on your own hardware, or on cloud virtual machines

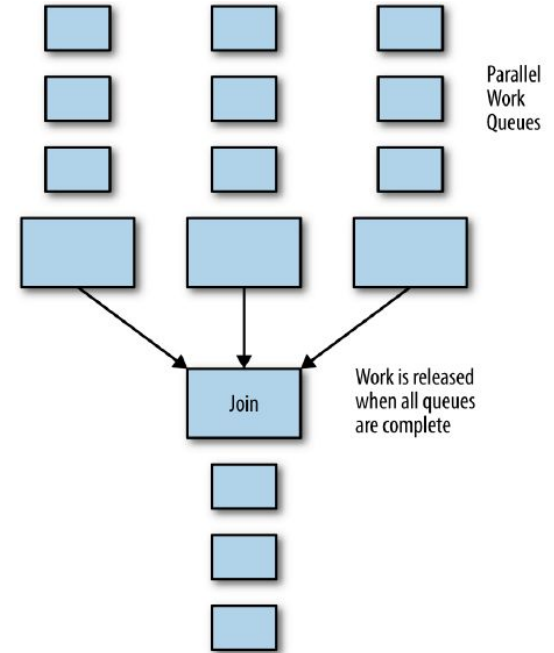
Coordinated Batch Processing

- A pattern for aggregation: pulling **multiple outputs back together** in order to generate some sort of **aggregate** output
- Probably the most canonical example of this aggregation is the *reduce* part of the **MapReduce pattern**
- Other different aggregate patterns
 - Join (or Barrier Synchronization)
 - Reduce
 - Sum
 - Histogram



Coordinated Batch Processing - Join

- When processing a workflow, sometimes it is necessary to have the complete set of work available to you before you move on to the next stage of the workflow (for example, use a sharded work queue to distribute work in parallel)
- Join pattern
 - Similar to joining a thread. Basic idea is that all of the work is happening in parallel, but work items aren't released out of the join until all of the work items that are processed in parallel are completed
 - This above is also known as **barrier synchronization** in concurrent programming
 - Difference between **Merger** pattern: Join is a stronger, **coordinated** primitive for batch data processing
 - Merger doesn't ensure that a complete dataset is present prior to the beginning of process - there can be **no guarantees about the completeness of the processing being performed**
 - Join ensures that **no data is missing before aggregation phase** is performed through coordination





Coordinated Batch Processing - Reduce

- Reduce is an example of a coordinated batch processing pattern because it can happen regardless of how the input is split up, and it is used *similar to join*; that is, to **group together the parallel output of a number of different batch operations on different pieces of data**
- Different from *join* pattern, the goal of reduce is **not to wait** until all data has been processed, but rather to **optimistically merge** together all of the parallel data items into a single comprehensive representation of the full set
- With the reduce pattern, each step in the reduce merges several different outputs into a single output. This stage is called “reduce” because
 - It reduces the **total number of outputs**
 - It reduces the data **from a complete data item to simply the representative data necessary** for producing the answer to a specific batch computation
 - The reduce phase can be repeated **as many** or **as few** times as necessary in order to successfully reduce the output down to a single output for the entire data set
 - The ability to begin data processing early means that the batch computation executes more quickly overall



Coordinated Batch Processing - Sum

- Sum is a similar but slightly different form of **reduction**: summation of a collection of different values
- Compare with **Counting**: add together a value that is present in the original output data, rather than simply counting one for every value
- Example: Measure the population of the United States
 - First, shard the work into work queues of towns, sharded by state
 - A second sharding to another set of work queues by county
 - Each work queue in each county produces a stream of outputs of (town, population) tuples
 - Sum two or more output together to produce a new output
 - This reduction can be performed an arbitrary number of times, in parallel



Coordinated Batch Processing - Histogram

- Another example of the reduce pattern
- Consider while counting the population of the United States, we also want to build a model of the average American family - Want to develop a *histogram* of family size
 - Use a *histogram model* that estimates the total number of families with zero to 10 children
 - The output of the data collection phase is a histogram per town
 - Use Histogram pattern to merge these histogram outputs: need multiply each histogram by its relative population, then divide this new total by the sum of the merged populations



Conclusion

- Modern softwares are required to **deliver APIs and services** to be consumed by mobile applications, devices in the internet of things (IoT), or even autonomous vehicles and systems. The increasing criticality of these systems means that it is necessary for these online systems to be built for **redundancy**, **fault tolerance**, and **high availability**
- Patterns like sidecars, ambassadors, sharded services, FaaS, work queues, and more can form the foundation on which modern distributed systems are built
- Distributed system developers should no longer be building their systems from scratch as individuals but rather **collaborating together on reusable, shared implementations of canonical patterns**