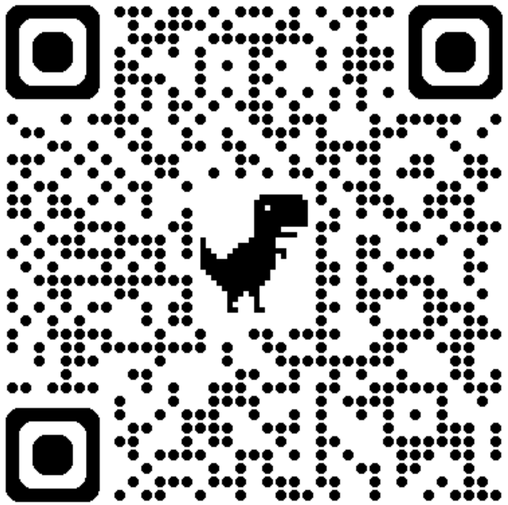
Architectural Patterns Focus: Fault Tolerance

Daniel Hinojosa

In this Presentation

- Circuit Breaker
- Throttling
- Retries
- Bulkhead
- Failover
- Fallback
- Timeout
- Leader Election
- Raft Protocol
- Competing Consumers
- Health Checks
- Replicator
- Idempotency
- Shadowing
- Graceful Degradation



Slides and Material: https://github.com/dhinojosa/nfjs-architectural-patterns-faulttolerance

Circuit Breaker

The Problem

- When a service synchronously requests from a remote call there is a big possibility of failure
- That remote call is that is a dependent may either be out or just taking too long
- Threads, since running synchronously will be taken up
- One service’s failures or latency can cause a cascading failure

The Solution

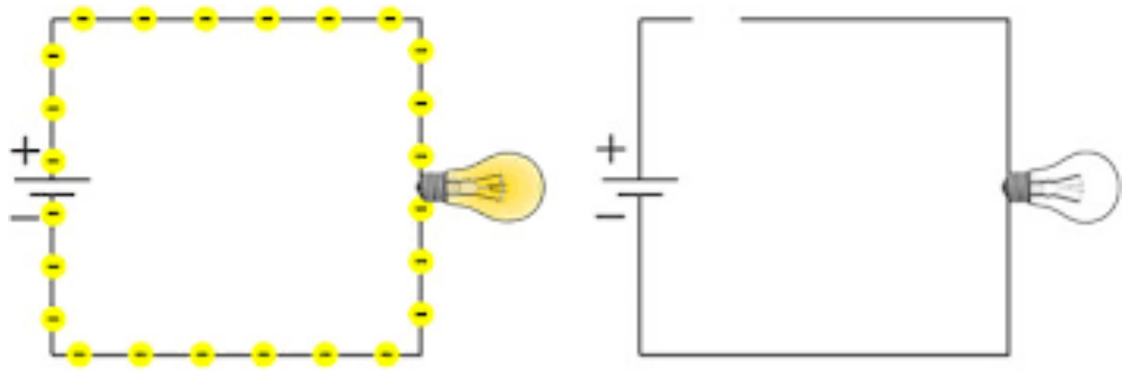
- The client should invoke the remote service by proxy
- When the number of consecutive failures crosses a threshold, the circuit trips
- All attempts to invoke the remote service will fail immediately
- After the timeout expires, the circuit breaker allows limited traffic to go through
- If it those test connections succeed, the circuit breaker resumes normal operation
- Otherwise, a new timeout is triggered

Closed Circuit Breaker

- The request from the application is routed to the operation. Proxy maintains a count of the number of recent failures
- If the call to the operation is unsuccessful the proxy increments this count
- If the number of recent failures exceeds a specified threshold within a given time period, the proxy is placed into the Open state.
- At this point the proxy starts a timeout timer, and when this timer expires the proxy is placed into the Half-Open state.
- The purpose of the timeout timer is to give the system time to fix the problem that caused the failure before allowing the application to try to perform the operation again.

Open Circuit Breaker

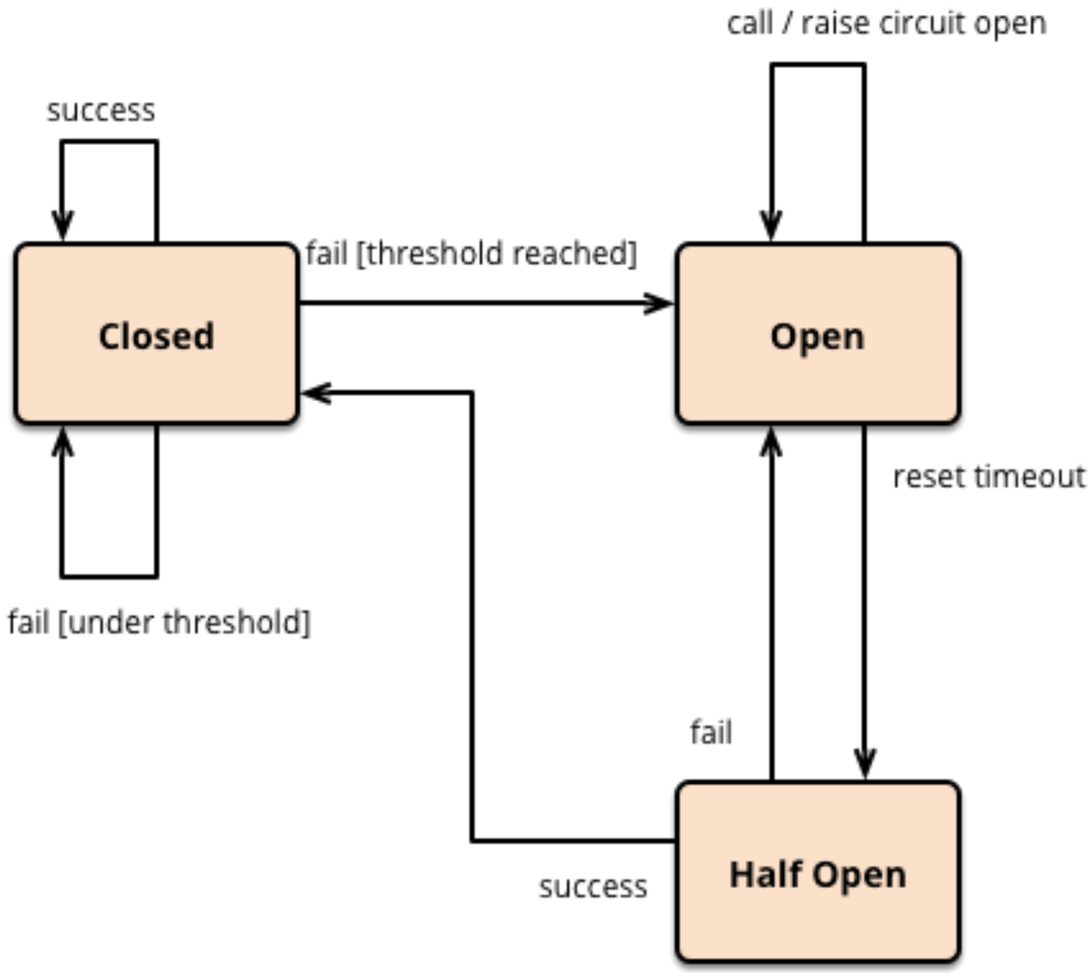
- The request from the application fails immediately and an exception is returned to the application
- Usually Open is a positive thing, but in this case it is not.



Half-Open Circuit Breaker

- A limited number of requests from the application are allowed to pass through and invoke the operation
- If these requests are successful, it's assumed that the fault that was previously causing the failure has been fixed and the circuit breaker switches to the **Closed** state
- If any request fails, the circuit breaker assumes that the fault is still present so it reverts back to the **Open** state and restarts the timeout timer to give the system a further period of time to recover from the failure

The Diagram



Source: https://martinfowler.com/bliki/CircuitBreaker.html

Notes on Circuit Breaker

- You must handle the exception thrown by the circuit breaker, or use Either types so that you can have the opportunity to degrade, or offer better exceptions
- Review the types of exceptions that are thrown, many are different. You may need to adjust your strategy based on the exception
- A circuit breaker can and should log all the requests for health monitoring

The Tradeoffs

- Not used for private owned resources like an in-memory database
- This would not be a substitute for handling exceptions for internal programmatic functionality, this is an architectural pattern

Demo: Resilience 4j

- Let’s show the Circuit Breaker using Resilience4j

Technology Stacks

- **Libraries:** Resilience4j, Polly for .NET
- **API Gateways:** Kong, AWS API Gateway, Apigee, KrakenD
- **Service Meshes:** Istio, Linkerd
- **Applications:** Envoy Proxy
- **Cloud Services:** AWS Lambda, Azure Functions, Google Cloud Functions

Throttling

The Problem

- Systems often face unpredictable spikes in traffic or usage.
- Backend resources like databases, APIs, and microservices may get overwhelmed.
- Unbounded traffic can lead to:
  - Degraded performance.
  - Service outages.
  - Poor user experience.

The Solution

- Throttling is a mechanism to control the rate at which requests are allowed to access system resources.
  - Limit the number of requests per unit of time per user, client, or system.
  - Implement quotas based on thresholds (e.g., per user, API, or region).
  - Employ mechanisms like rate limiting, token buckets, or sliding windows.

Rate Limiting

- A technique to control the number of requests a system processes in a specific time frame.
- Used to enforce usage policies and prevent resource overuse or abuse.
  - **Request Quotas:** Limits on the number of requests allowed per user, client, or system.
  - **Time Window:** Defines the period over which requests are counted (e.g., per second, minute, hour).
  - **Thresholds:** Configurable limits tailored to specific endpoints or clients.
- Rate Limiting is typically done as Fixed Window aggregates

Token Buckets

- A rate-limiting algorithm used to control request flow and manage resource usage.
- Based on a “bucket” metaphor where tokens represent permissions to process requests.
- Process:
  - **Token Generation:** Tokens are added to the bucket at a fixed rate.
  - **Request Processing:** Each request consumes one or more tokens.
  - **Capacity Limit:** The bucket has a maximum capacity, preventing token overflow.
  - **Request Handling:** If the bucket has tokens, the request is allowed; otherwise, it’s denied or queued.
- Anagogy: Tokens at a carnival

Sliding Windows

- A time-based mechanism to control request rates over a continuously updated time frame.
- Tracks the number of requests within a “sliding” or rolling window of time (e.g., last 60 seconds).
- Process:
  - **Dynamic Window:** The time frame continuously updates as new requests arrive.
  - **Request Tracking:** Keeps a record of timestamps for incoming requests.
  - **Evaluation:** Counts requests in the active window to decide whether to allow or block new ones.
  - **Decision Logic:** If the count exceeds the limit, subsequent requests are denied until older timestamps fall out of the window.

The Tradeoffs

- Advantages:
  - Protects critical resources and avoids system downtime.
  - Improves overall system resilience and availability.
  - Enables smoother user experiences during high demand by prioritizing critical traffic.
- Drawbacks:
  - Complexity: Adds implementation and maintenance overhead.
  - User Frustration: Legitimate users may be blocked if limits are too restrictive.
  - False Negatives: Dynamic traffic spikes might still overwhelm systems before throttling activates.
  - Overhead: Throttling itself consumes processing and memory resources.

Technology Stacks

- **Libraries:** Resilience4j, Polly for .NET
- **API Gateways:** Kong, AWS API Gateway, Apigee, KrakenD, NGINX
- **Service Meshes:** Istio, Linkerd
- **Applications:** Envoy Proxy
- **Load Balancers:** NGINX Plus, AWS ELB
- **Messaging:** Kafka, RabbitMQ, AWS SQS

Retries

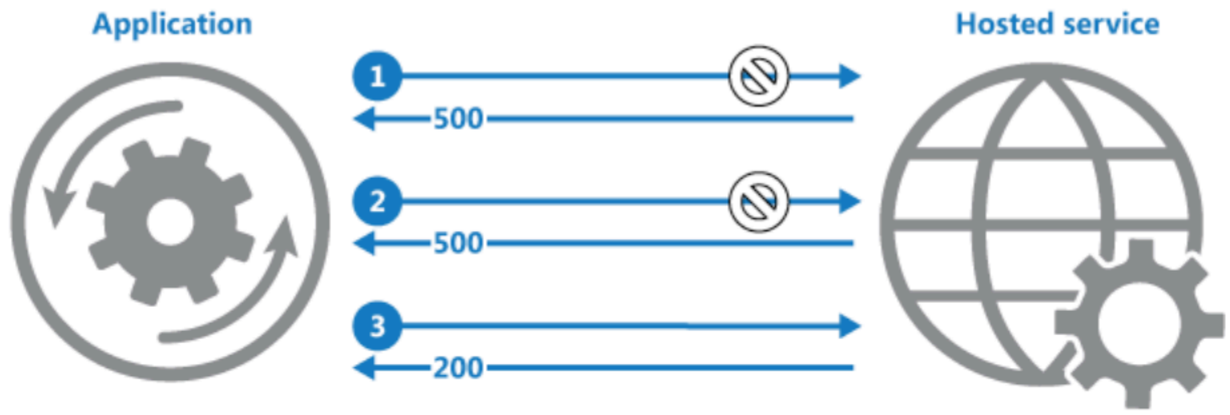
The Problem

- There can be transient faults when it comes to networking applications
  - Momentary loss of network
  - Unavailable Service
  - Service is busy
- Services may also throttle requests if the load is too high until resources start easing

The Solution

- Handle the failures with the following:
  - **Cancel**. If the fault indicates that the failure isn't transient or is unlikely to be successful if repeated, the application should cancel the operation and report an exception. For example, an authentication failure caused by providing invalid credentials is not likely to succeed no matter how many times it's attempted.
  - **Retry.** If the specific fault reported is unusual or rare, it might have been caused by unusual circumstances such as a network packet becoming corrupted while it was being transmitted. In this case, the application could retry the failing request again immediately because the same failure is unlikely to be repeated and the request will probably be successful.
  - **Retry after delay.** If the fault is caused by one of the more commonplace connectivity or busy failures, the network or service might need a short period while the connectivity issues are corrected or the backlog of work is cleared. The application should wait for a suitable time before retrying the request.

The Diagram



1: Application invokes operation on hosted service. The request fails, and the service host responds with HTTP response code 500 (internal server error).
2: Application waits for a short interval and tries again. The request still fails with HTTP response code 500.
3: Application waits for a longer interval and tries again. The request succeeds with HTTP response code 200 (OK).

Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/retry

Retry Strategies

- Retries can be done using a backoff time period. This can be incremental or exponential
- Retries should log all the attempts for diagnosis
- Exponential Backoff is useful if the service is frequently busy
- Retries are very common with messaging systems, but this will often end in an at least once semantic

Technology Stacks

- **Libraries:** Resilience4j, Polly for .NET
- **API Gateways:** Kong, AWS API Gateway, Apigee, KrakenD, NGINX
- **Service Meshes:** Istio, Linkerd
- **Applications:** Envoy Proxy
- **Load Balancers:** NGINX Plus, AWS ELB
- **Messaging:** Kafka, RabbitMQ, AWS SQS

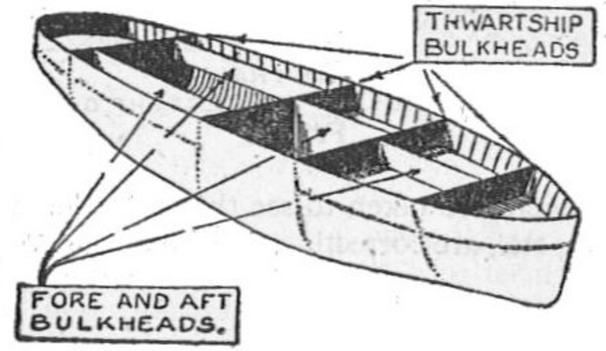
Bulkhead

The Problem

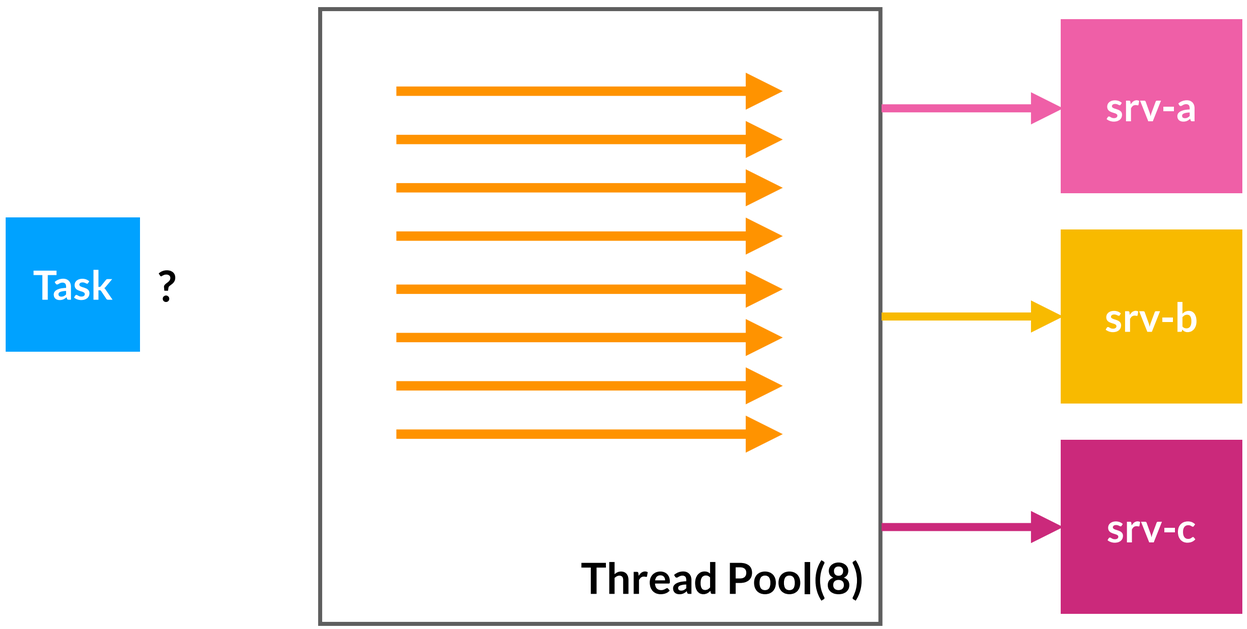
- Elements of an application are isolated into pools so that if one fails, the others will continue to function
- An application may constitute multiple services, with each service having one or more consumers
- Resources may become exhausted
- We will require that thread pools do not exceed a certain threshold

The Solution

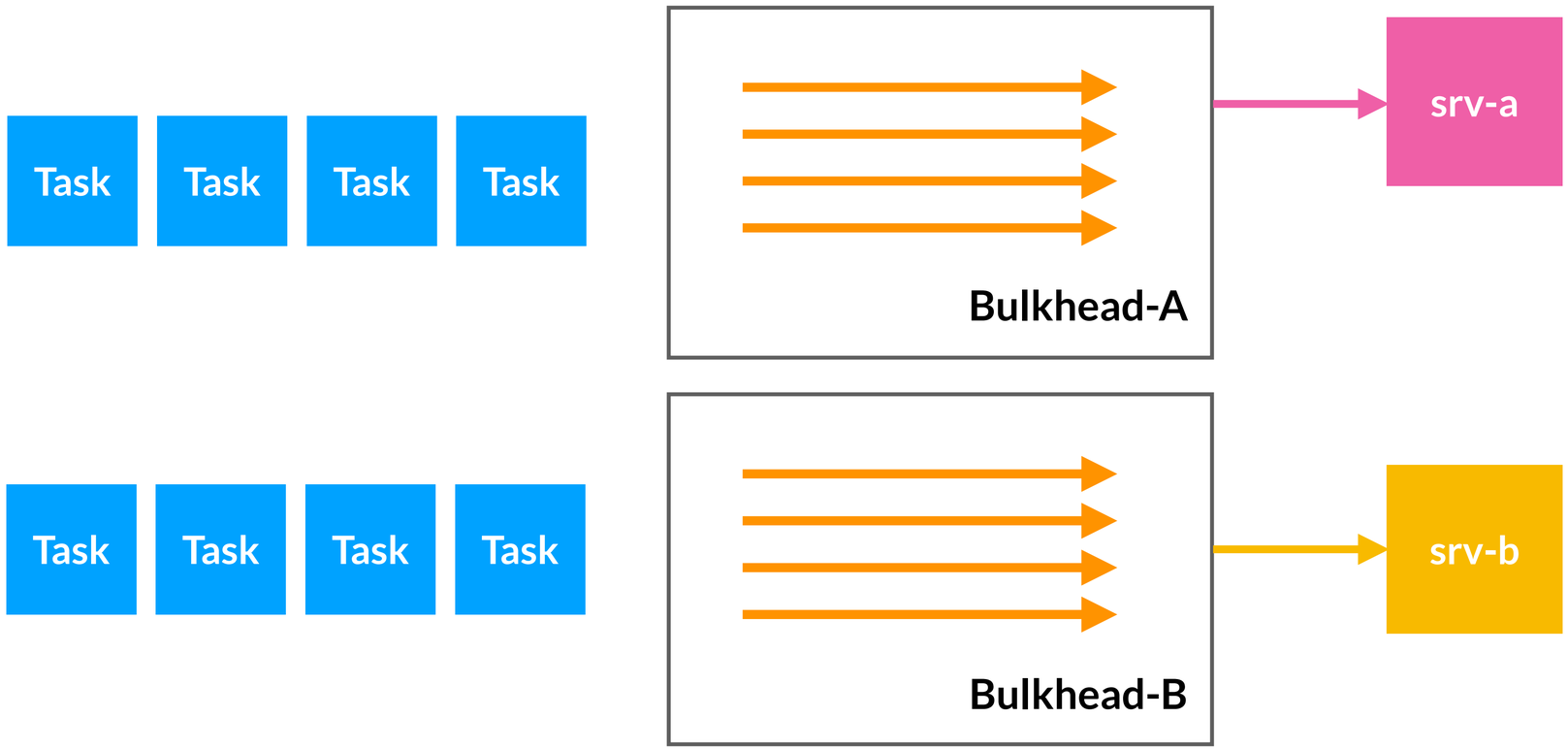
- Partition Service instances into groups based on consumer load and availability requirements
- A consumer that calls multiple services may be assigned a connection pool for each service, and that connection pool can either have a task on a queue, or backed by another thread
- If a service begins to fail, it only affects the connection pool assigned for that service, allowing the consumer to continue using the other services



The Diagram



The Diagram


Task
Task
Task
Task
Task
Task
Task
Task
Bulkhead-A
Bulkhead-B
srv-a
srv-b

The Tradeoffs

- Great for isolating resources used to consume a set of backend services, especially if the application can provide some level of functionality even when one of the services is not responding.
- Perfect for isolating critical consumers from standard consumers.
- It protects the application from cascading failures.
- It does add some complexity

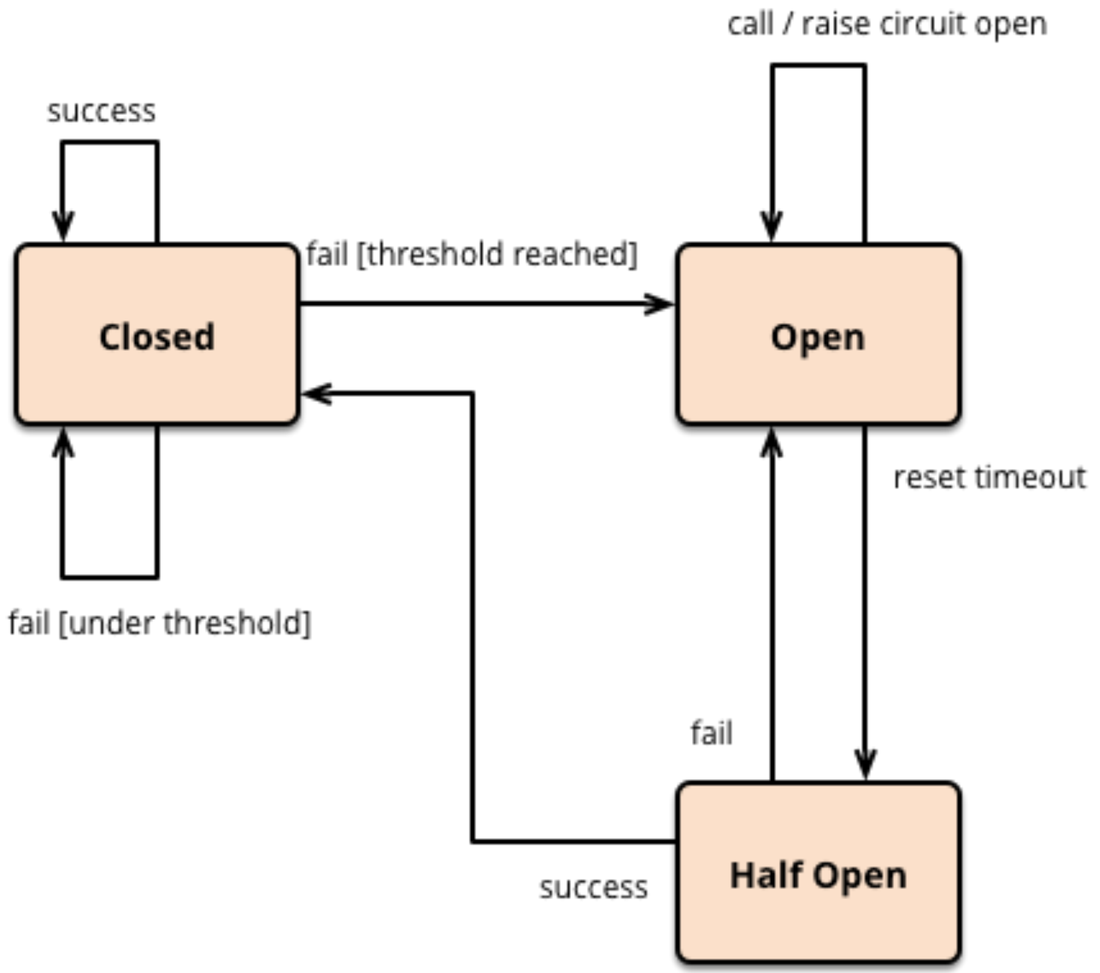
Demo: Resilience 4j

- Again we will turn our attention to Resilience4j
  - This time we will perform a bulkhead pattern so as to ensure that not all threads will be consumed for various purposes

Ambassador Pattern

The Problem

- Applications require a multitude of services like
  - Circuit breaking
  - Monitoring
  - Metering
  - Datasource Connectivity and Proxying
- There are risks when it comes to application you are unfamiliar with.


call / raise circuit open
success
fail [threshold reached]
Closed
Open
reset timeout
fail [under threshold]
fail
success
Half Open

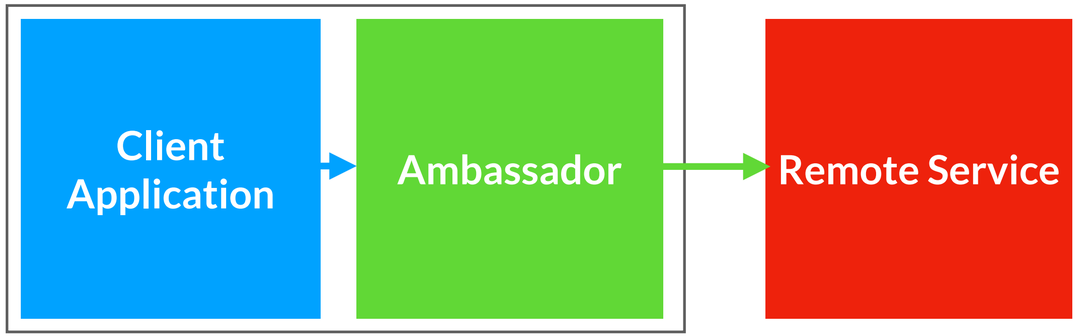
The Solution

- An Ambassador pattern is a proxy for your application to external services
- Deploy the proxy to the same environment, e.g. same pod
- The proxy can monitor performance metrics such as latency or resource usage, and this monitoring happens in the same host environment as the application
- You can make updates to the ambassador without affecting the legacy application
- This can be typically done as a sidecar in the same pod

The Solution

- There is minimal latency overhead
- The solution is better off to be integrated to the application itself
- When client connectivity features are consumed by a single language, a better option might be a client library that is distributed to the development teams as a package

The Diagram



The ambassador will be intercept traffic and provide services for caching, security, circuit breaking, and more

Ambassador vs Sidecar

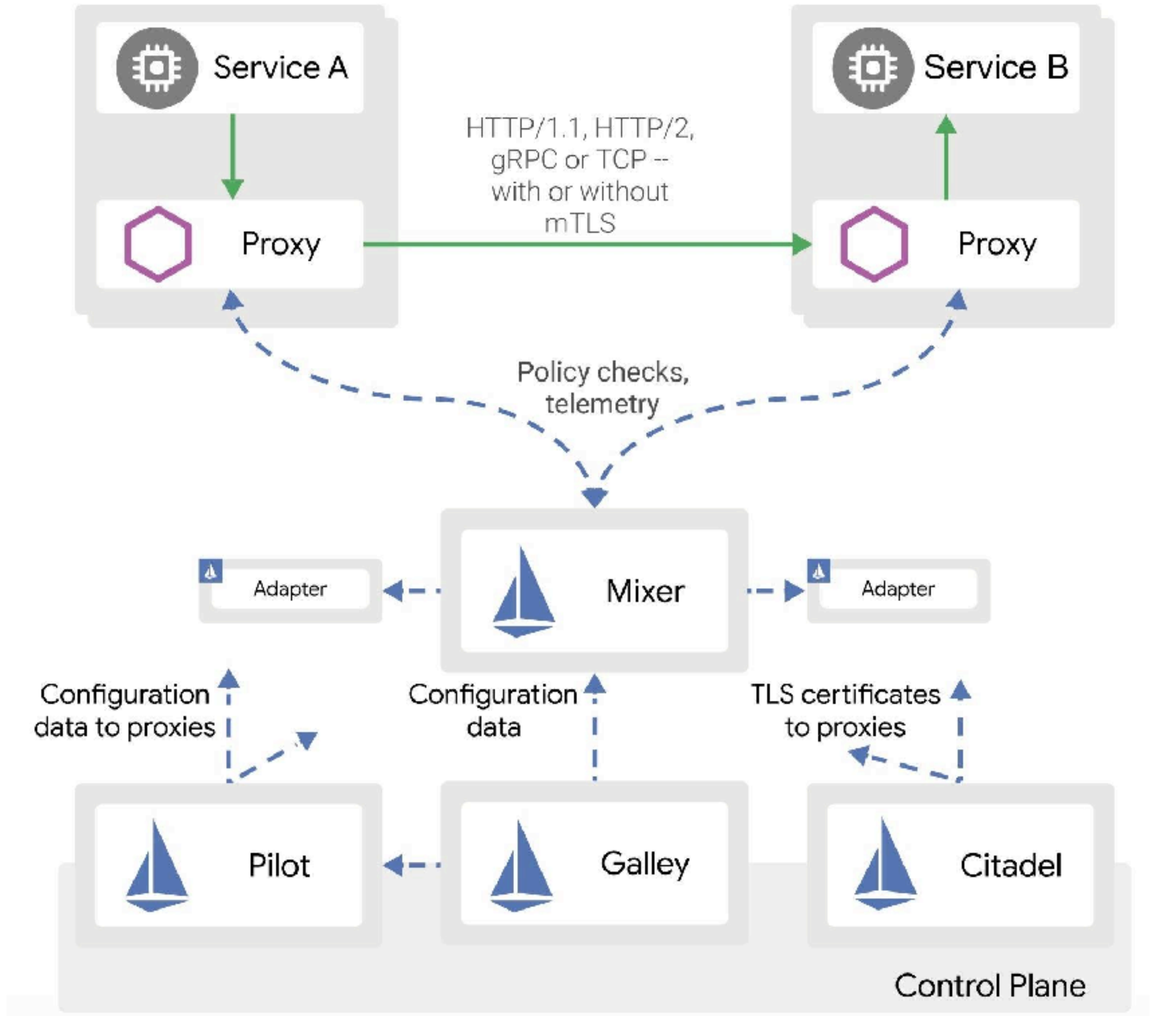
Ambassador:

- Used for monitoring, logging, routing, security (like TLS), all *networking based tasks*
- Used to provide circuit breaking
- Ambassador services can be deployed as a sidecar to accompany the lifecycle of a consuming application or service

Side Car:

- Additional container attached to a parent application and provides supporting features for the application
- If it includes networking it would be considered an ambassador as well
- If it purely logging, configuration it is typically just a sidecar
- It can get rather blurry when trying to distinct between the two

Istio


Service A
Service B
HTTP/1.1, HTTP/2,
gRPC or TCP --
with or without
mTLS
Proxy
Proxy
Policy checks,
telemetry
Adapter
Mixer
Adapter
Configuration
data to proxies
Configuration
data
TLS certificates
to proxies
Pilot
Galley
Citadel
Control Plane

Istio Components

- **Pilot** - Responsible for configuring the Envoy and Mixer at runtime.
- **Proxy / Envoy** - Sidecar proxies per microservice to handle ingress/egress traffic between services in the cluster and from a service to external services.
- **Mixer** - Create a portability layer on top of infrastructure backends. Enforce policies such as ACLs, rate limits, quotas, authentication, request tracing and telemetry collection at an infrastructure level.
- **Citadel / Istio CA -** Secures service to service communication over TLS. Providing a key management system to automate key and certificate generation, distribution, rotation, and revocation.
- **Ingress/Egress** - Configure path based routing for inbound and outbound external traffic.
- **Control Plane API** - Underlying Orchestrator such as Kubernetes or Hashicorp Nomad.

Demo: Envoy

- Envoy was the basis for Istio
- With envoy you can set up proxies to perform a variety of actions
- View the documentation here: https://www.envoyproxy.io/docs/envoy/v1.29.2/

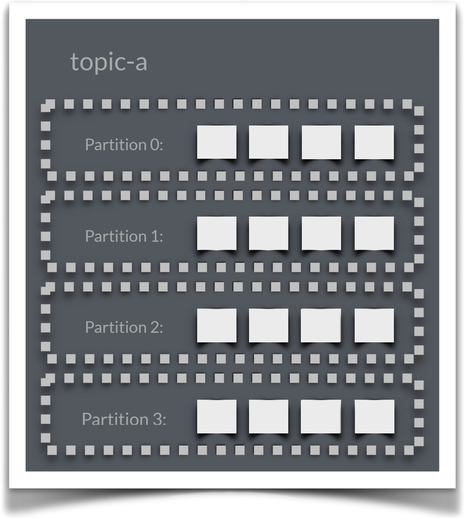
Failover

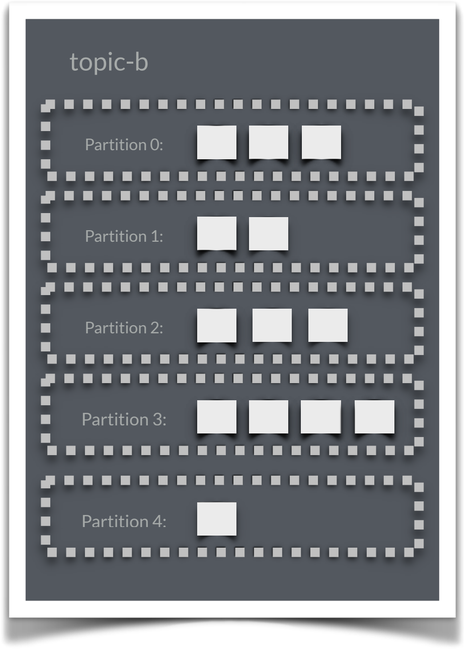
The Problem

- When a critical component fails, it can disrupt the entire system, particularly if it is a single point of failure.
- Unexpected failures can lead to downtime, impacting user experience, revenue, and reliability.
- Identifying failures quickly and accurately is challenging in complex systems.
- Switching to a backup or standby system often takes time, reducing system availability.

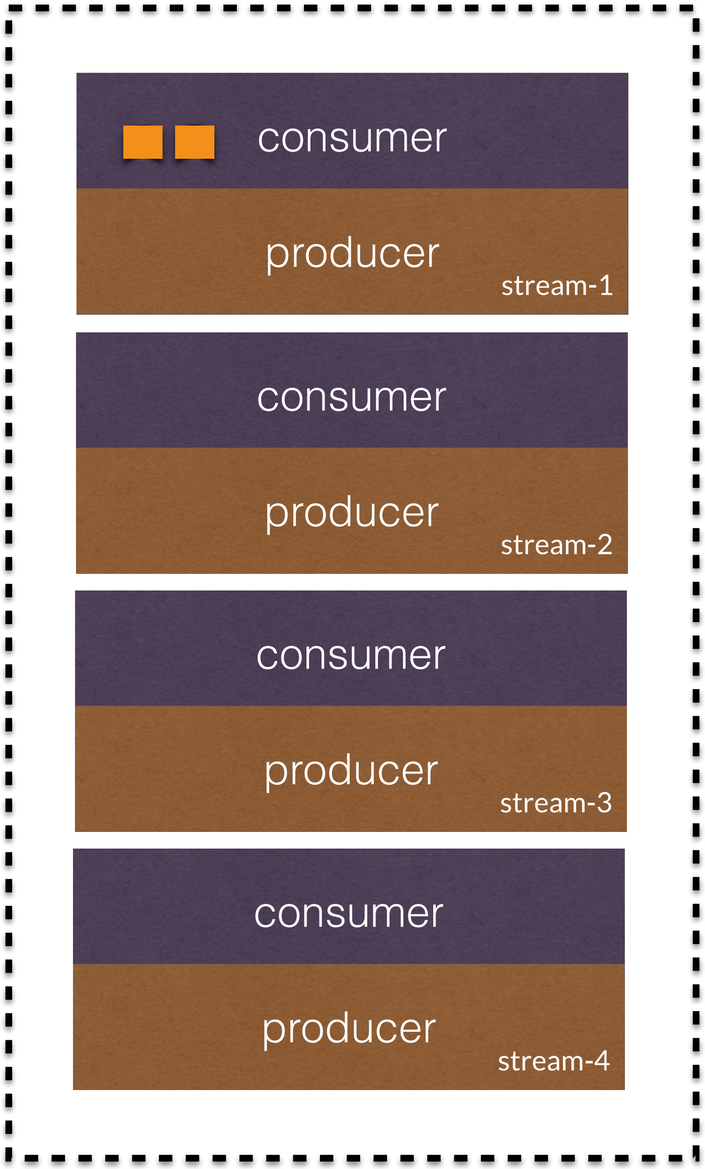
The Solution

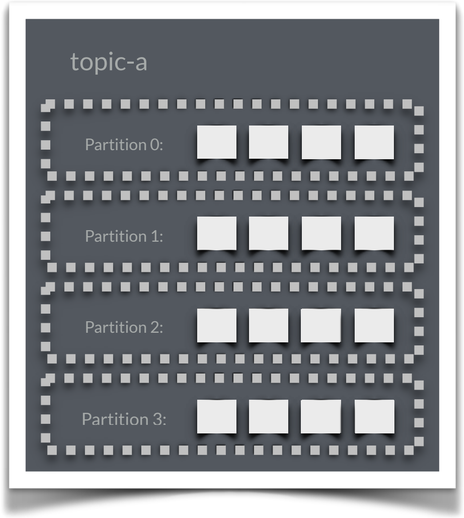
- Automatically or manually switching to a standby system or component when a primary one fails
- The failover can either be Active-Passive, Active-Active.
- Heartbeat monitoring is required to determine when the failover needs to happen
- Redundant infrastructure is also required to enable the swap. This infrastructure is typically situated on different availability zones or racks

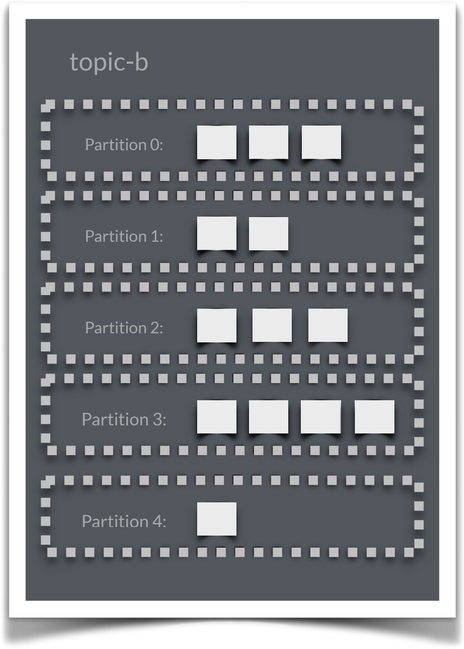




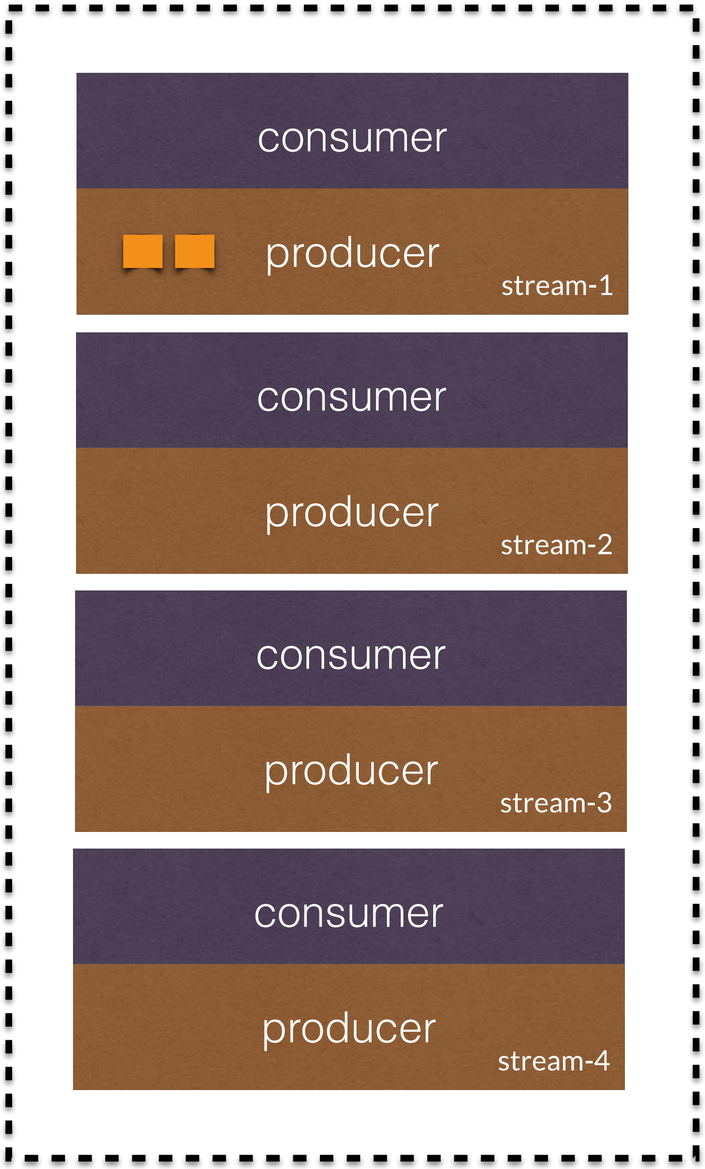
application.id = “my-stream”

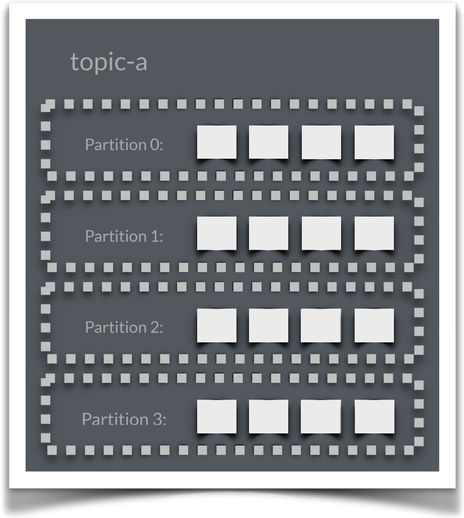


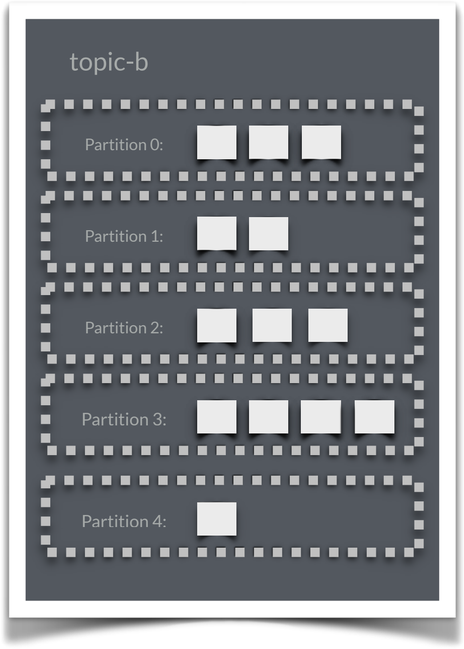


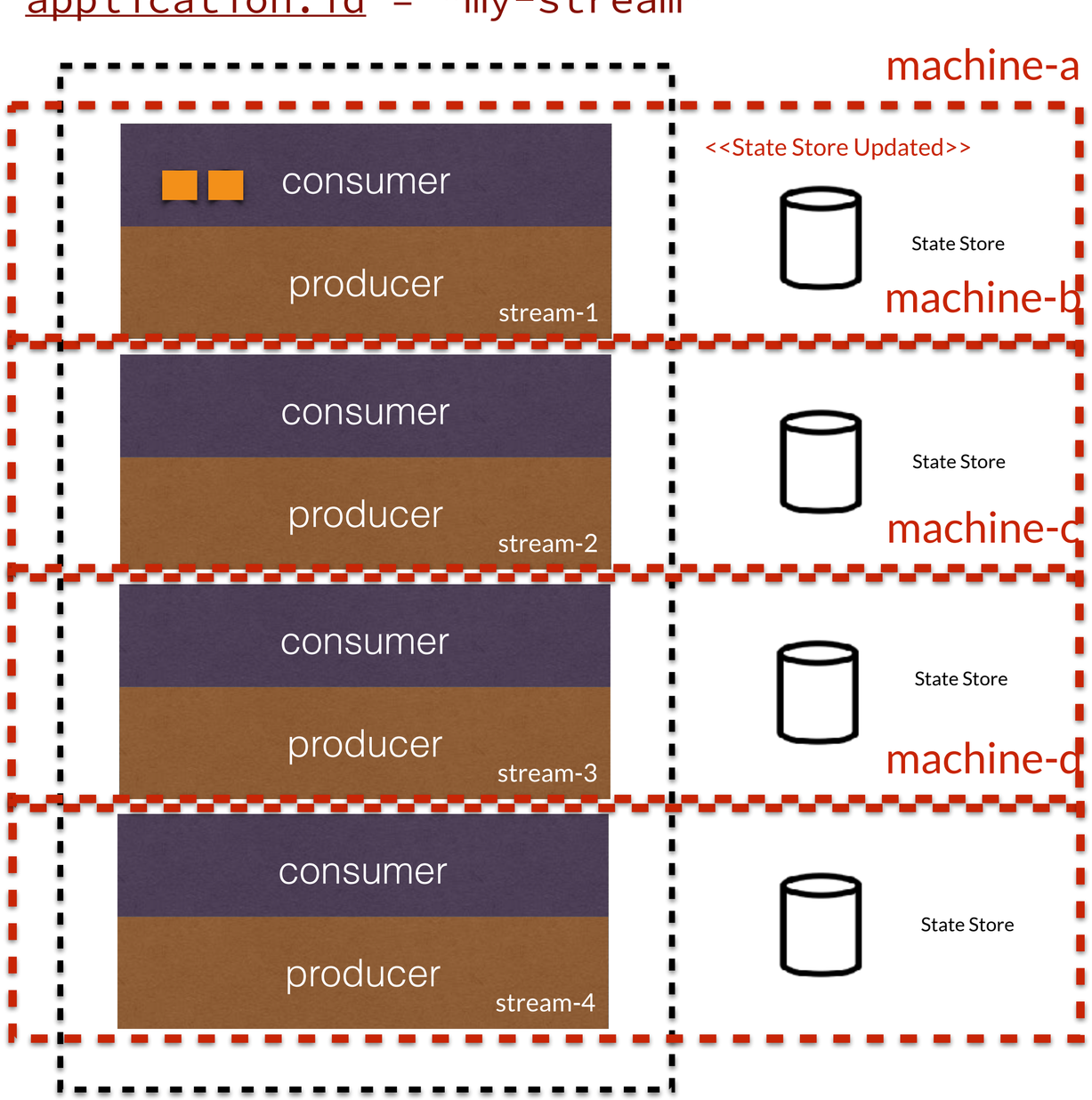


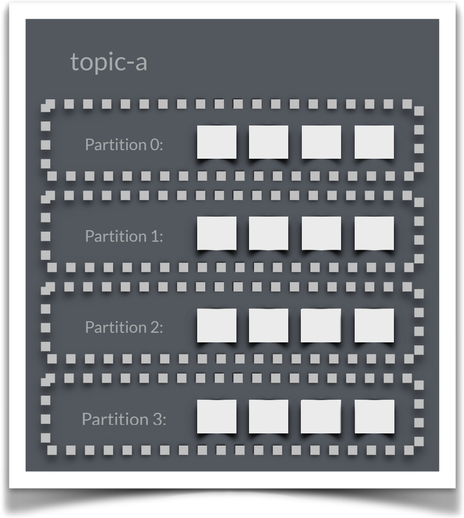
application.id = “my-stream”

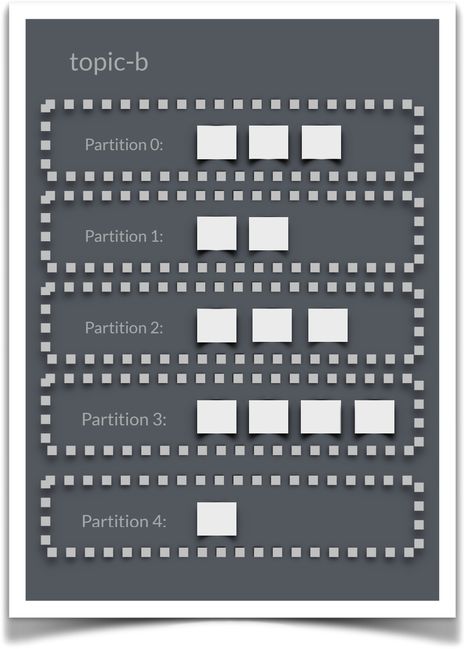



topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:

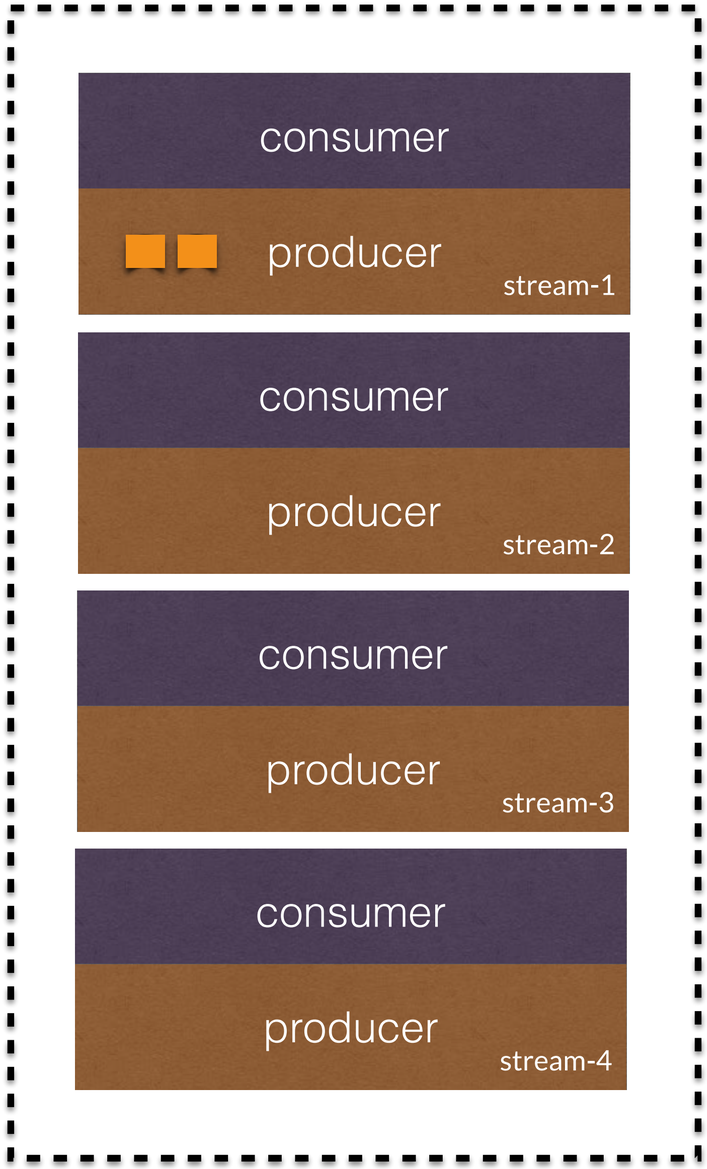

topic-b
Partition 0:
Partition 1:
Partition 2:
Partition 3:
Partition 4:

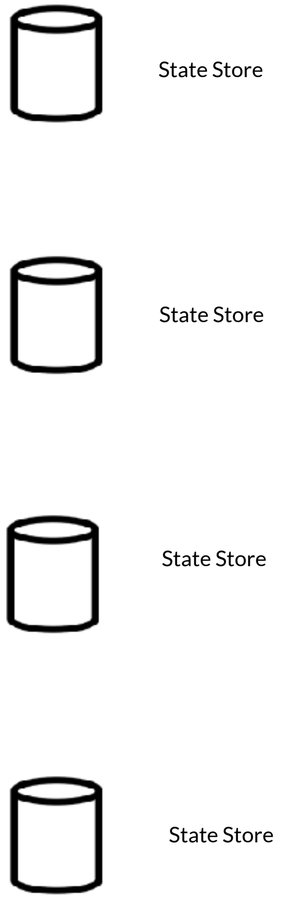

application.id = “my-stream”
machine-a
<<State Store Updated>>
consumer
producer
stream-1
State Store
machine-b
consumer
producer
stream-2
State Store
machine-c
consumer
producer
stream-3
State Store
machine-d
consumer
producer
stream-4
State Store



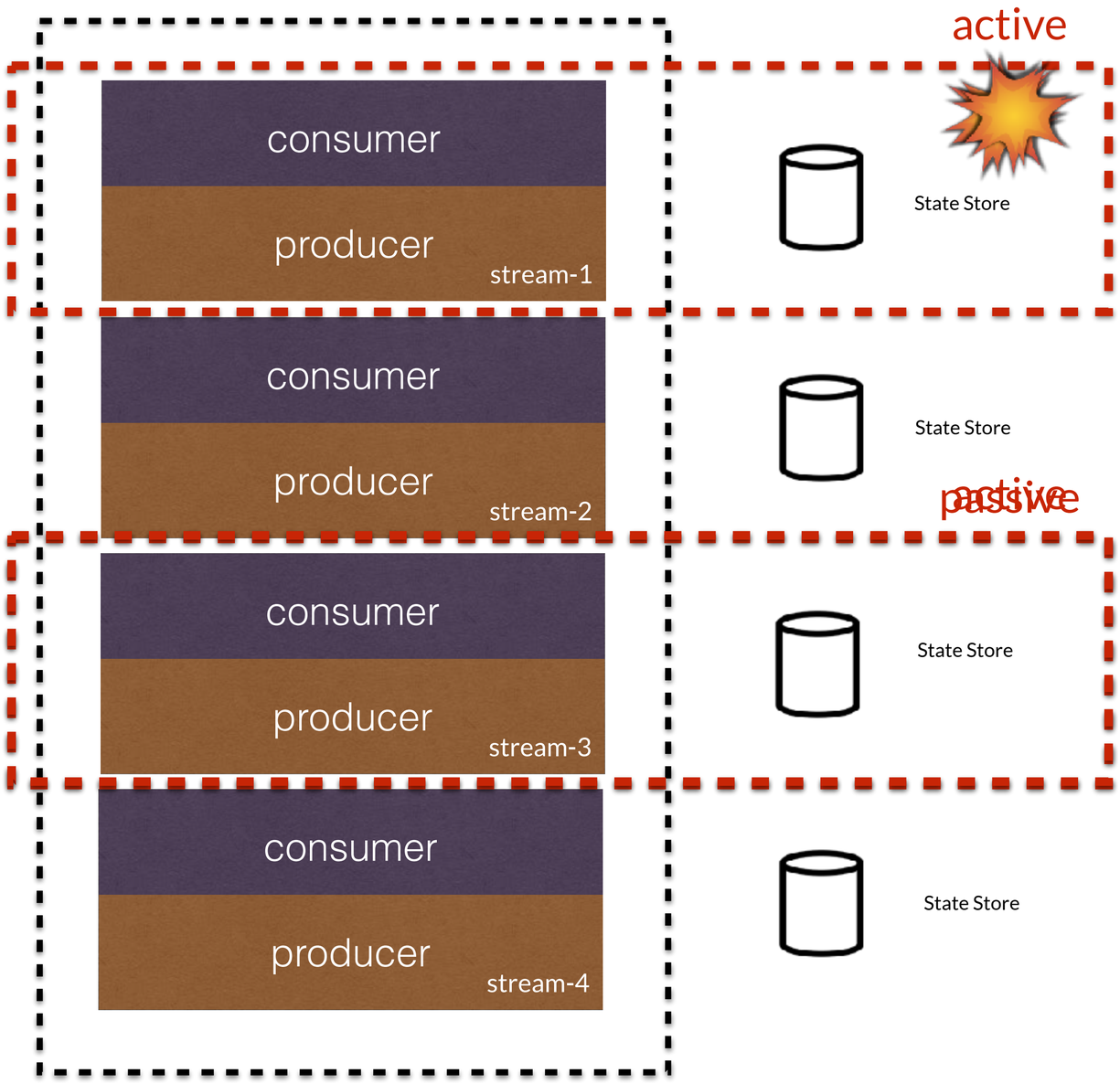


application.id = “my-stream”

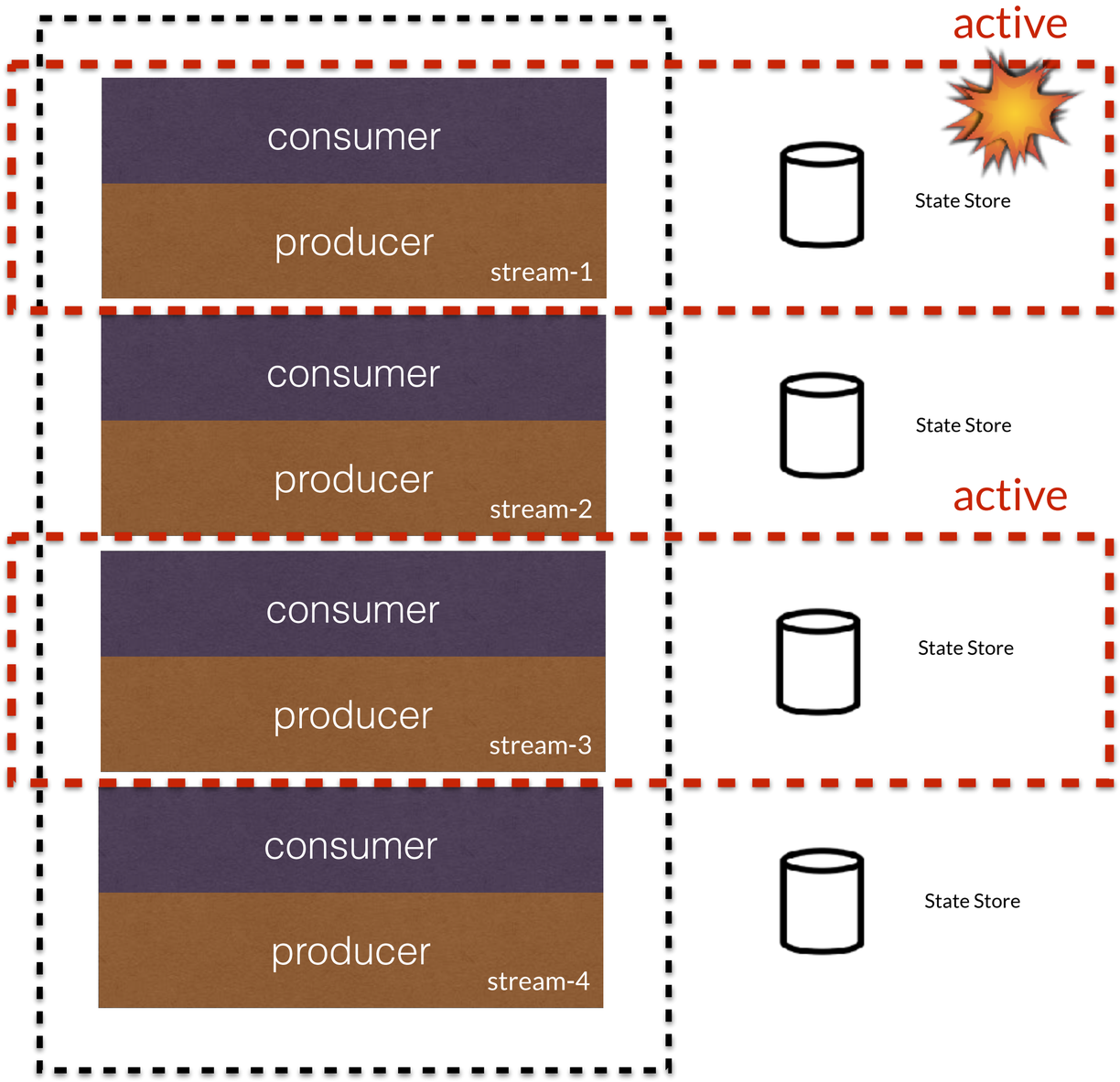




application.id = “my-stream”



application.id = “my-stream”



The Tradeoffs

- Maintaining redundant systems increases infrastructure and operational costs
- Implementing and testing failover mechanisms can complicate system design and maintenance.
- Active-Passive may experience downtime during the switch.
- Active-Active minimizes downtime but requires synchronization overhead.
- In Active-Active setups, inconsistent state synchronization can lead to data divergence.
- False positives can trigger unnecessary failovers, while false negatives delay recovery.

Technology Stacks

- **Libraries:** Resilience4j, Polly for .NET
- **API Gateways:** Kong, AWS API Gateway, Apigee, KrakenD, NGINX
- **Service Meshes:** Istio, Linkerd
- **Applications:** Envoy Proxy
- **Load Balancers:** NGINX Plus, AWS ELB, HA Proxy
- **Cloud Providers:** • AWS Route 53, Cloudflare, Google Cloud DNS:
- **Messaging:** Kafka, RabbitMQ, AWS SQS

Fallback

The Problem

- Modern systems rely heavily on external services (e.g., APIs, databases, third-party providers).
- If a dependent service becomes unavailable or slow:
  - It may cascade failures to upstream services.
  - Degrades user experience or causes complete outages.
  - How can a system remain functional or partially operational when dependent services fail?

The Solution

- A *fallback* is a predefined, alternate response or behavior triggered when a primary service or dependency fails.
- How It Works:
  - Detect failure or timeout from the dependent service.
  - *Switch to an alternative action or response (e.g., cached data, default values).*
  - Continue operations with reduced or partial functionality.
- Maintain system availability and ensure minimal disruption to users.

Fallback Strategies

- **Default Response:** Return a static or predefined response (e.g., “Service temporarily unavailable”).
- **Cached Data:** Use previously cached responses or data.
- **Alternative Service:** Redirect requests to a backup or secondary service.
- **Degraded Functionality:** Offer partial functionality (e.g., read-only mode).
- **Replaying Data Missed:** Use EDA and Message Brokers to replay missed data

Technology Stacks

- **Libraries:** Resilience4j, Polly for .NET
- **API Gateways:** Kong, AWS API Gateway, Apigee, KrakenD
- **Service Meshes:** Istio, Linkerd
- **Cloud Services:** AWS Lambda, Azure Functions, Google Cloud Functions

Timeout

The Problem

- Distributed systems often rely on external services, APIs, or databases.
- If a dependency becomes slow or unresponsive:
  - Requests may hang indefinitely, blocking resources.
  - System throughput degrades, leading to cascading failures.
  - User experience suffers due to delays or timeouts.
- How do you prevent one unresponsive component from impacting the entire system?

The Solution

- A timeout is a mechanism that limits the maximum time a system waits for a response from a dependency.
- Process:
  - Define a time threshold for each request.
  - If the dependency doesn’t respond within the threshold, terminate the request.
  - Handle the timeout gracefully by retrying, triggering fallbacks, or returning meaningful errors.
- Prevent unresponsive components from blocking resources and impacting system availability.

Timeout Types

- **Request Timeout:** Maximum time to wait for an HTTP request.
- **Connection Timeout:** Maximum time to establish a connection to a service.
- **Read Timeout:** Maximum time to wait for data during a read operation.
- **Write Timeout:** Maximum time to wait for data during a write operation.

Application and Pairings

- Can be applied at the application level (e.g., API calls) or infrastructure level (e.g., database connections, service mesh).
- Combine timeouts with *retries*, *fallbacks*, and *circuit breakers* for resilience.

The Tradeoffs

- **Advantages:**
  - **Resource Protection:** Frees up resources by terminating unresponsive requests.
  - **System Stability:** Prevents cascading failures and ensures system responsiveness.
  - **Improved User Experience:** Provides predictable responses even during failures.
- **Drawbacks:**
  - **False Positives:** Legitimate requests may be terminated if the timeout is too short.
  - **Complex Configuration:** Requires careful tuning to balance performance and fault tolerance.
  - **Increased Retries:** May amplify traffic during temporary slowdowns if retries are not limited.
  - **Error Propagation:** Improperly handled timeouts can lead to cascading error responses.

Technology Stacks

- **Languages:**
  - **Java:** HTTPClient, OkHTTP, Spring WebClient, RestTemplate, Resilience 4j
  - **Python:** `requests`, `Asyncio`
  - **.NET:** `HttpClient`, Polly
- **Infrastructure:** Istio, LinkerD for service to service communication
- **API Gateways:** AWS API Gateway, Kong, and Apigee support timeout settings
- **Load Balancers:** NGINX and HAProxy provide timeout options for connections and responses.
- **Databases:** Most database clients support query and connection timeouts

Leader Elections

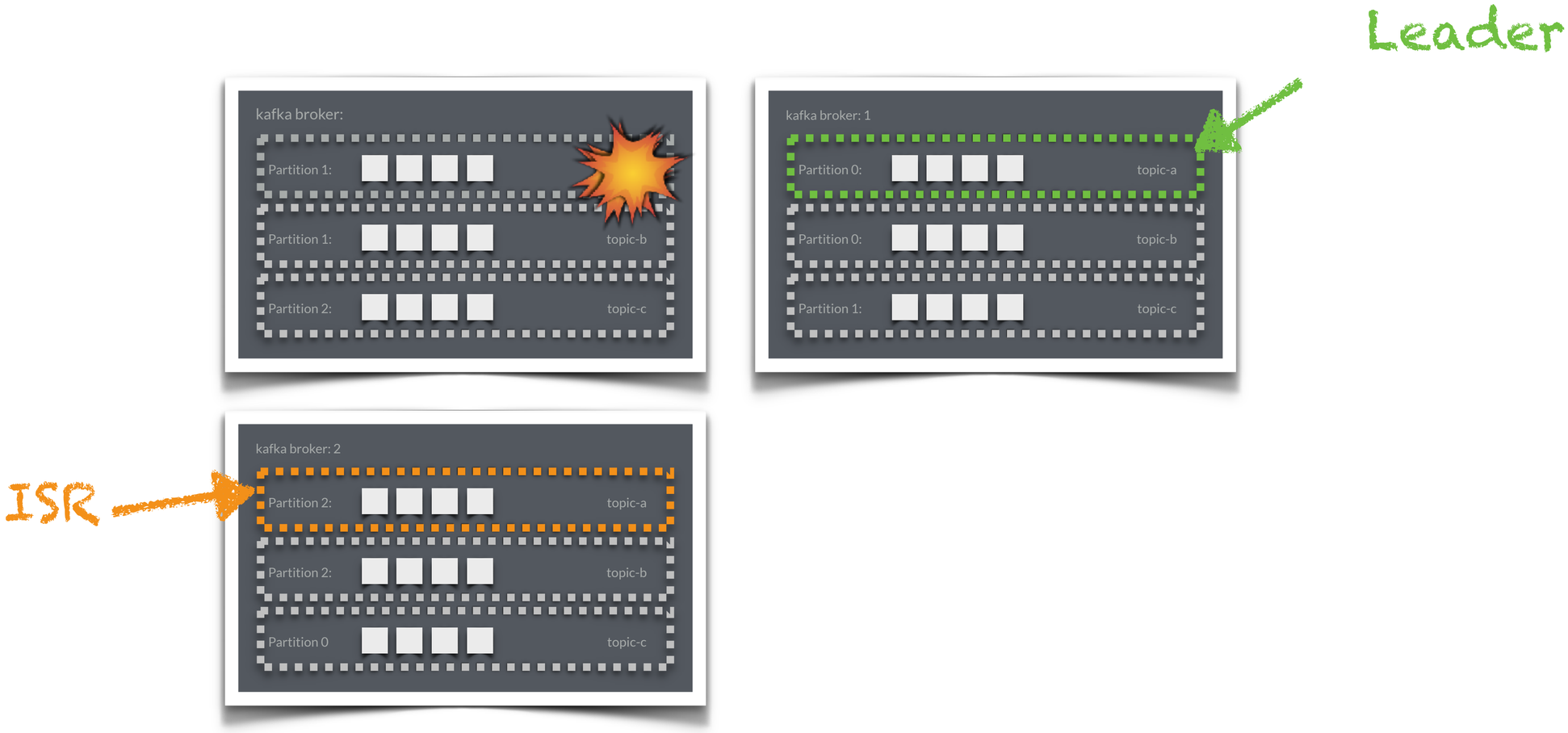
The Problem

- Distributed systems often require coordination for tasks like assigning responsibilities, managing shared resources, or handling failures.
- Without a clear leader, systems can suffer from:
  - Conflicting decisions.
  - Inefficient resource usage.
  - Inconsistent states across nodes.

The Solution

- Leader election is a process where nodes in a distributed system dynamically select a single node as the “leader” to coordinate tasks and manage resources.
  - Nodes compete or vote to elect a leader using a consensus algorithm.
  - Once elected, the leader handles responsibilities such as task delegation, resource allocation, or failure recovery.
  - If the leader fails, the system initiates a new election.
- Ensure a single source of truth or coordinator while maintaining fault tolerance and high availability.


Leader
kafka broker:
Partition 1:
Partition 1:
topic-b
Partition 2:
topic-c
ISR
kafka broker: 1
Partition 0:
topic-a
Partition 0:
topic-b
Partition 1:
topic-c
ISR
kafka broker: 2
Partition 2:
topic-a
Partition 2:
topic-b
Partition 0
topic-c


Leader
kafka broker:
Partition 1:
Partition 1:
topic-b
Partition 2:
topic-c
kafka broker: 1
Partition 0:
topic-a
Partition 0:
topic-b
Partition 1:
topic-c
ISR
kafka broker: 2
Partition 2:
topic-a
Partition 2:
topic-b
Partition 0
topic-c

Key Features

- **Consensus Mechanisms:**
  - Ensure agreement among nodes on who the leader is.
  - Algorithms like Raft, Paxos, or ZooKeeper handle leader elections.
- **Election Triggers:**
  - No current leader (initial state).
  - Leader failure detected (e.g., heartbeat timeout).
  - Network partition or recovery.
- **Requirements for Effectiveness:**
  - Uniqueness: Only one leader at a time.
  - Fault Tolerance: System can elect a new leader if the current one fails.
  - Agreement: All nodes recognize the same leader.

Tradeoffs

- **Advantages:**
  - **Coordination:** Simplifies decision-making in distributed systems.
  - **Consistency:** Ensures one authoritative source for critical tasks.
  - **Fault Tolerance:** Automatically recovers leadership after failures.
- **Drawbacks:**
  - **Latency:** Elections can delay operations, especially during network partitions or frequent leader failures.
  - **Complexity:** Requires robust algorithms to avoid split-brain scenarios.
  - **Single Point of Bottleneck:** The leader might become a performance bottleneck for high-traffic operations.
  - **Edge Cases:** Handling network partitions or simultaneous leader candidates can be challenging.

Technology Stacks

- **Consensus Protocols:**
  - **Raft:** Simplifies leader election with understandable, fault-tolerant consensus.
  - **Paxos:** A robust but complex protocol for distributed consensus and leader election.
- **Coordination Tools:**
  - **Apache ZooKeeper:** Provides leader election and distributed coordination.
  - **Etcd:** A distributed key-value store with built-in leader election capabilities.
- **Distributed Systems Platforms:**
  - **Kubernetes:** Uses leader election for controllers and schedulers.
  - **Consul:** Offers leader election for service discovery and distributed coordination.
- **Frameworks:**
  - **Hazelcast:** Supports leader election in distributed data grids.
  - **Akka Cluster:** Implements leader election for actor-based distributed systems.

Raft Protocol

The Problem

- Ensuring consistency across nodes in a distributed system is challenging, especially during failures.
- Determining a leader in a reliable and efficient way under potential network partitions.
- Preventing data divergence between nodes while maintaining high availability.
- We want to handle node crashes and resynchronization without significant downtime.

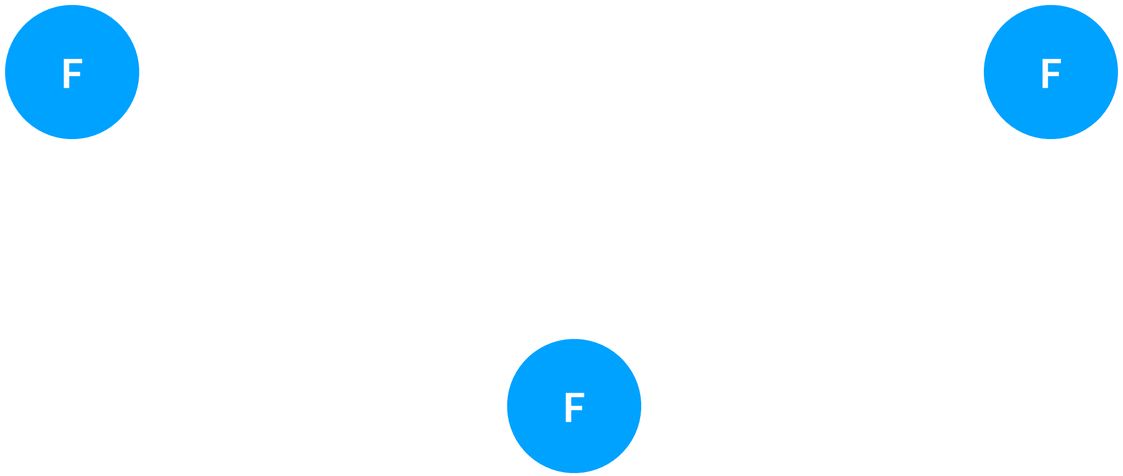
The Solution

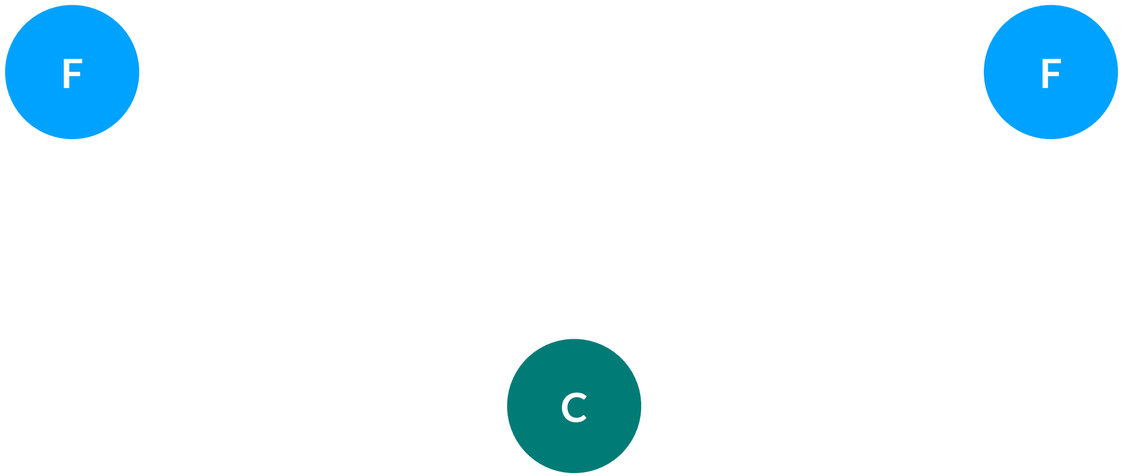
- **Raft Protocol:** A consensus algorithm designed to be understandable and robust for distributed systems. It is also easier than Paxos another architectural pattern to communicate data in a fault-tolerant way.
- Raft ensures a single leader is responsible for log replication.
- Commands are committed only after a majority consensus, ensuring consistency.
- Uses randomized timeouts to efficiently elect a leader during failures.
- Guarantees that all servers agree on the same sequence of commands.

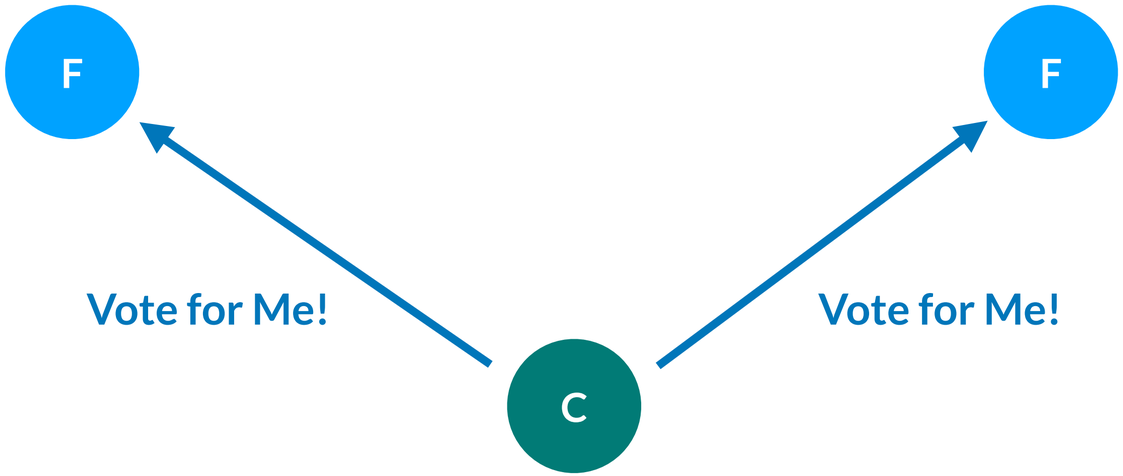
Raft is Majority Based

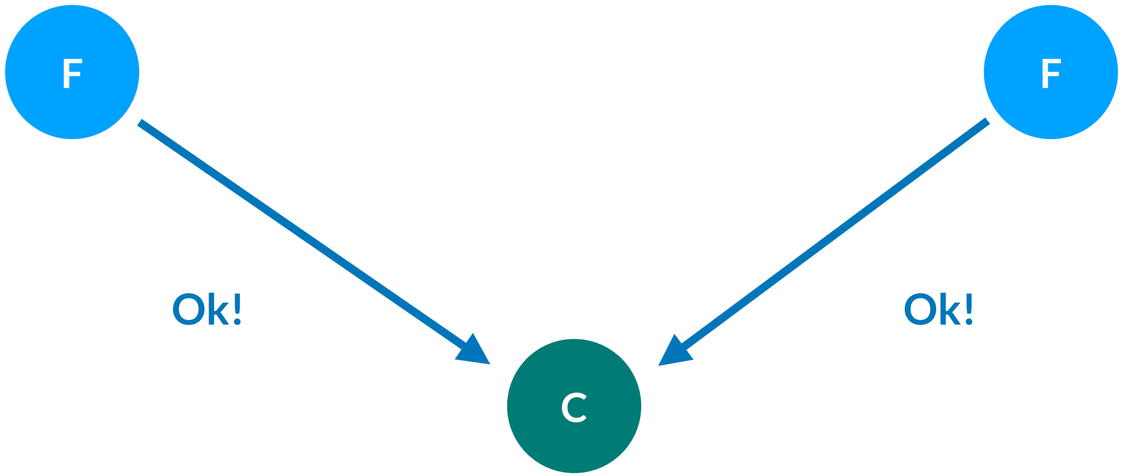
- Raft requires multiple systems
- If you have 3 systems and 2 goes down, you are out!
- If you have 5 systems and 3 goes down, you are out!

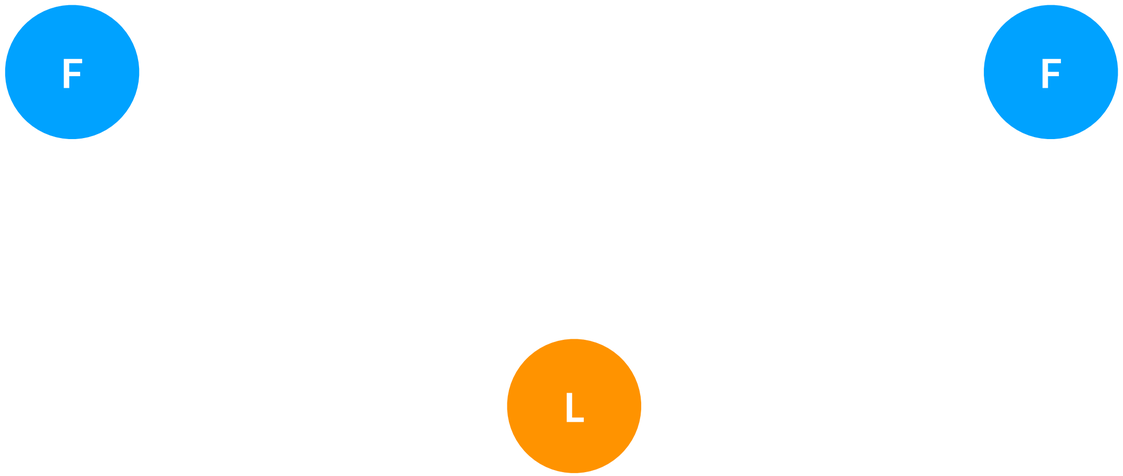
Happy Path

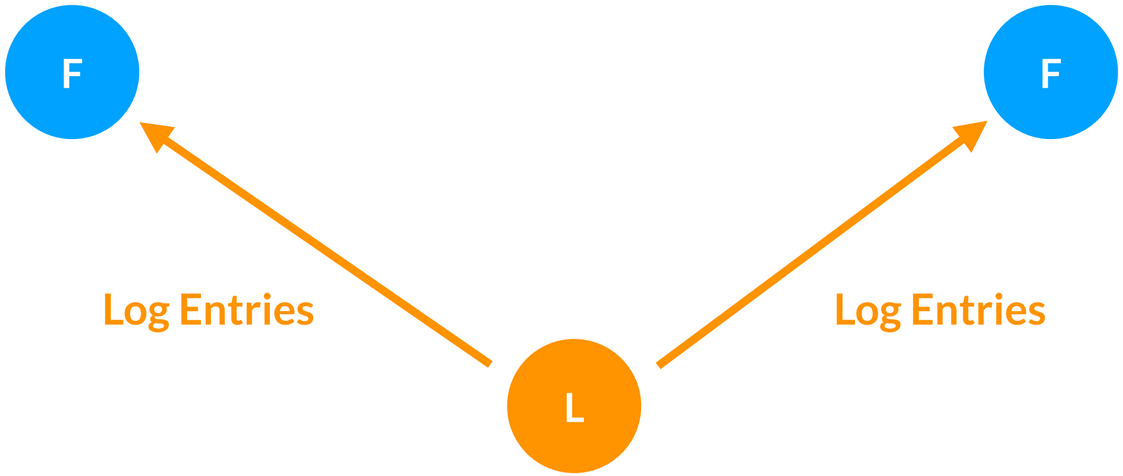

F
F
F


F
F
C

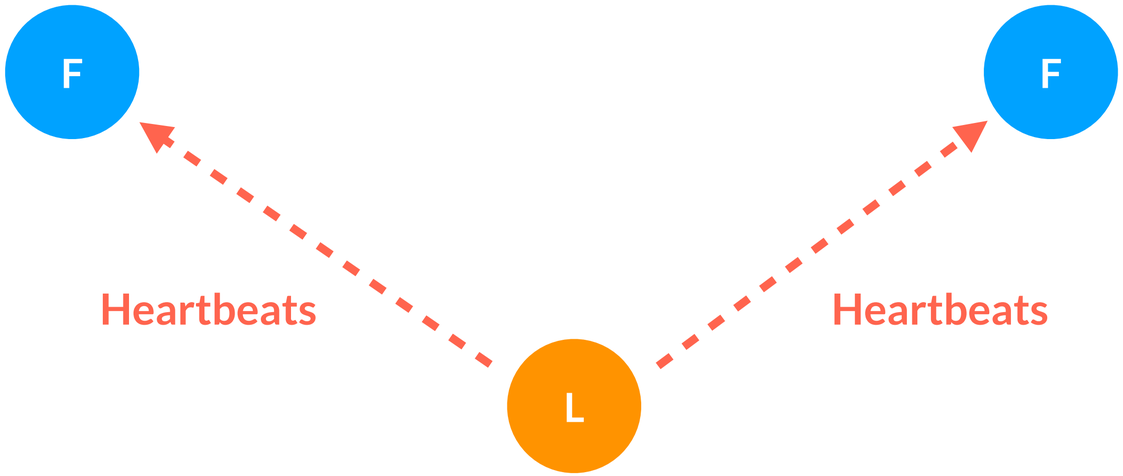

F
F
Vote for Me!
Vote for Me!
C


F
F
Ok!
Ok!
C


F
F
L

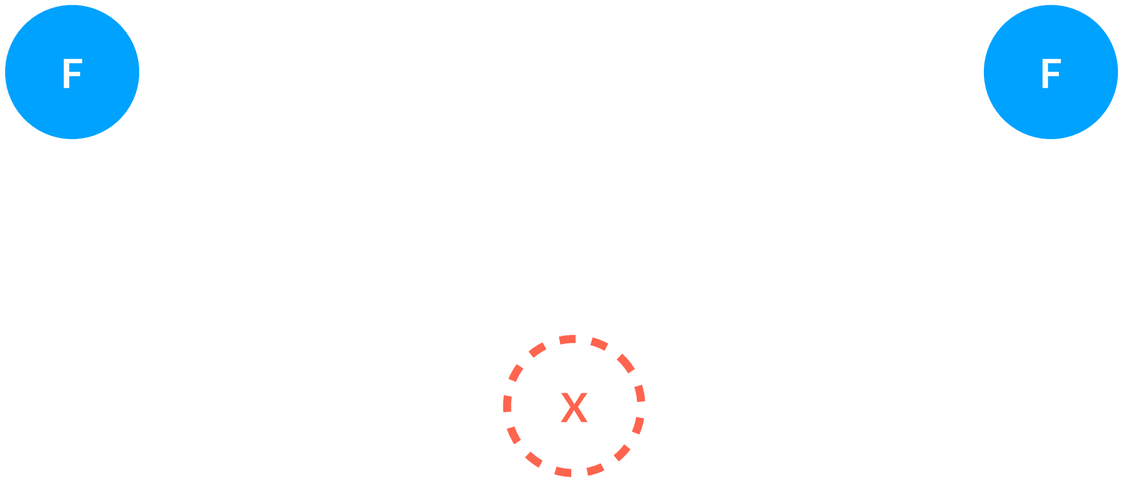


State changes are actively communicated to the followers

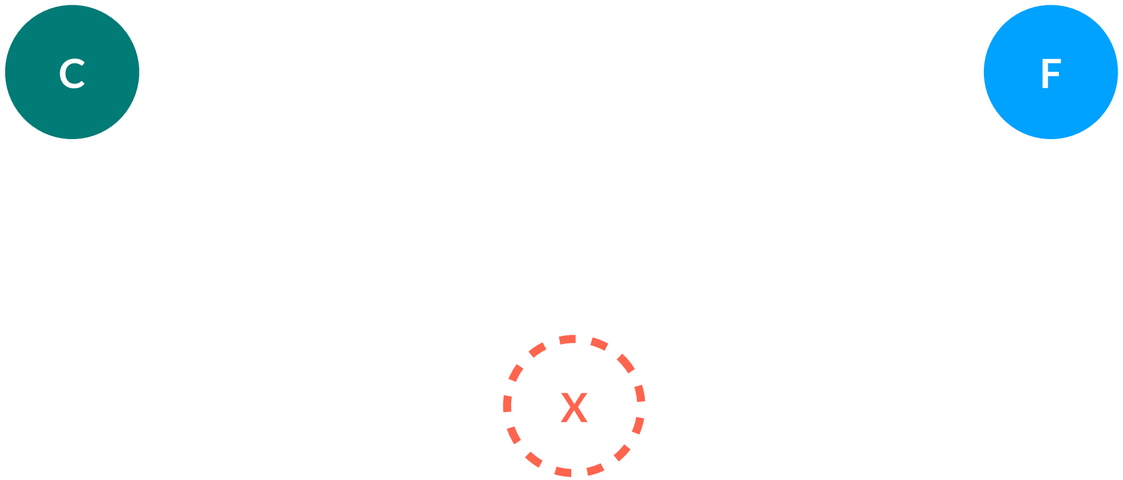


If no active changes to state are being done, then heartbeats are sent

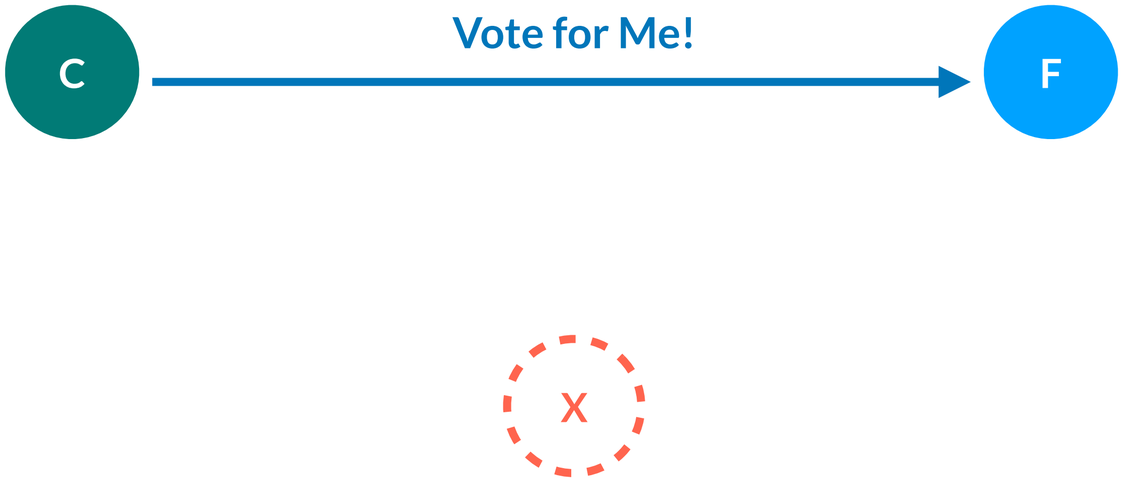
Leader Failure



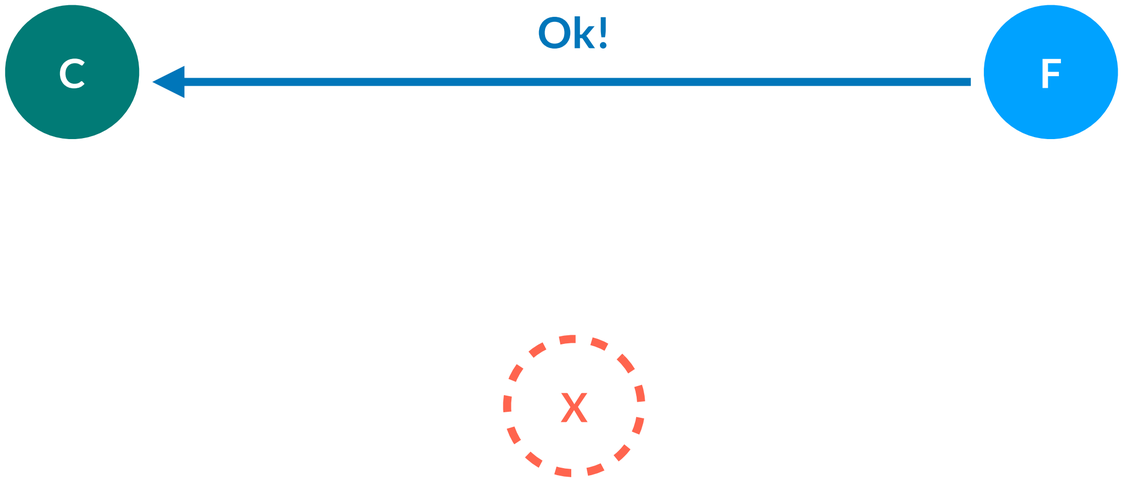
Leader has failed



Since each clock is randomized, one will trip. It becomes a candidate



New candidate requests that the followers vote for it

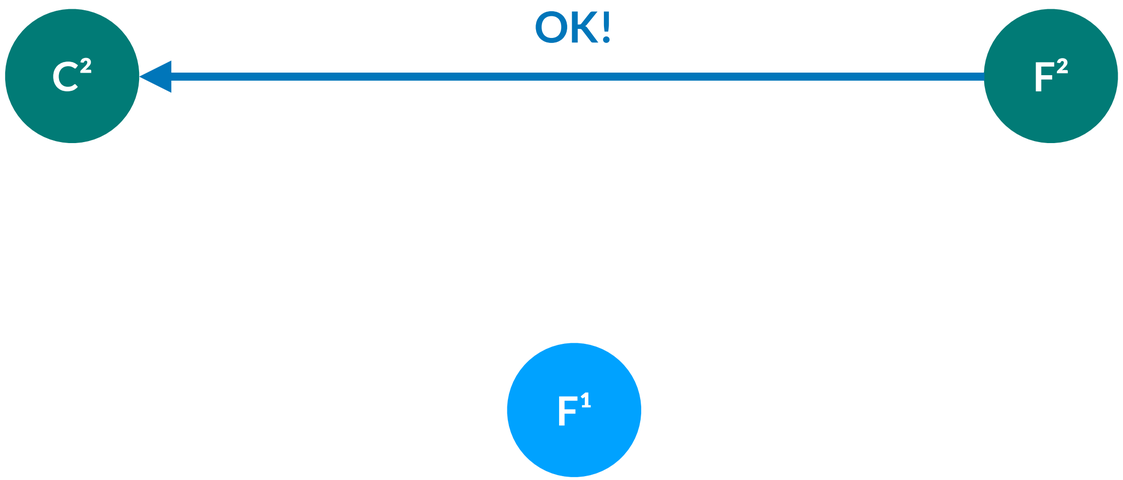


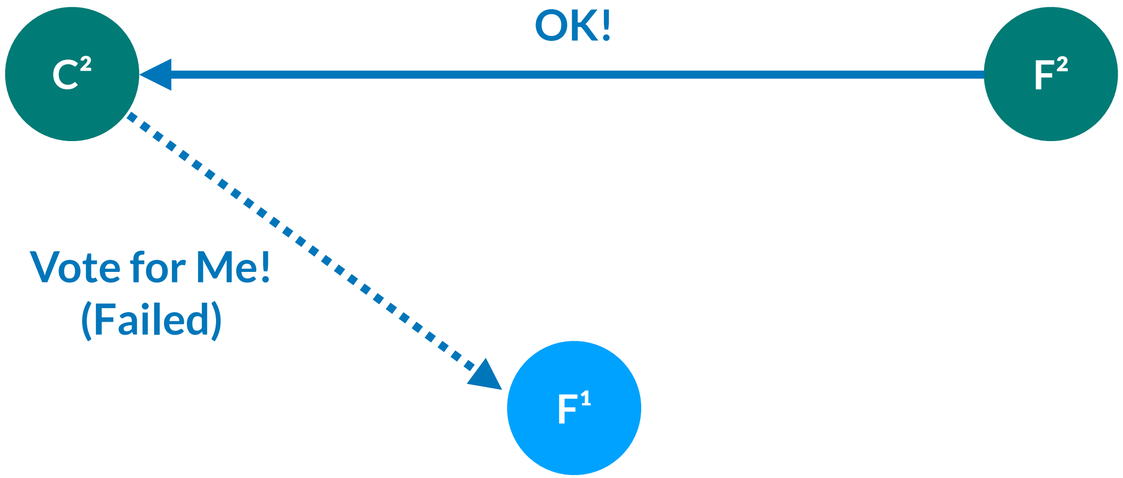
Follower votes for the candidate

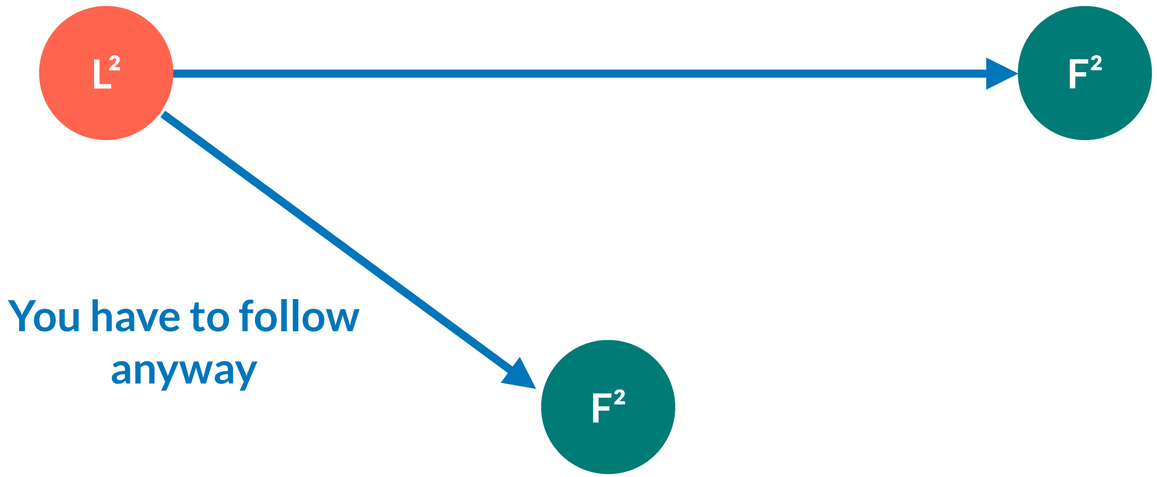
New Leader takes control. It had two votes, itself and the follower, it sends state changes and heartbeats

Term Elections

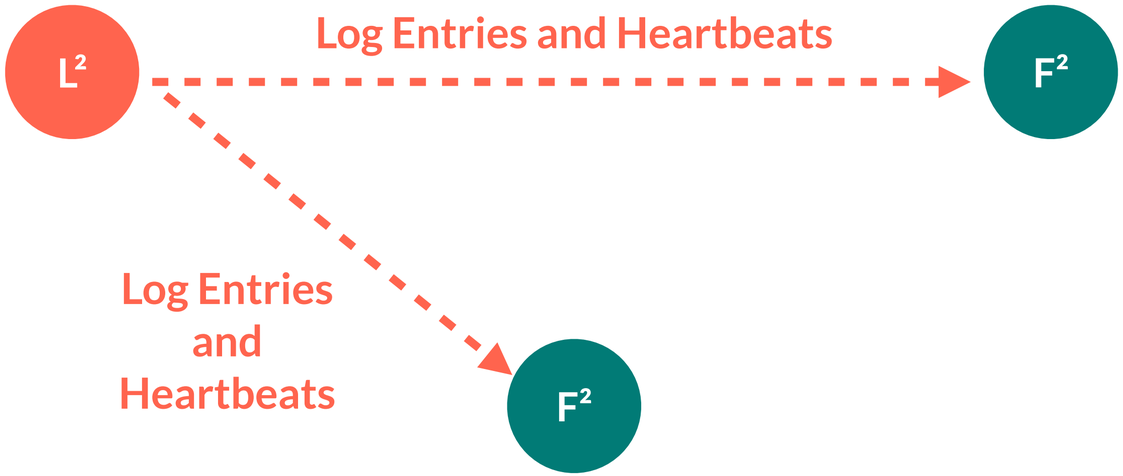

Vote for Me!
C2
F1
F1


OK!
C²
F²
F¹


OK!
C²
F²
Vote for Me!
(Failed)
F¹

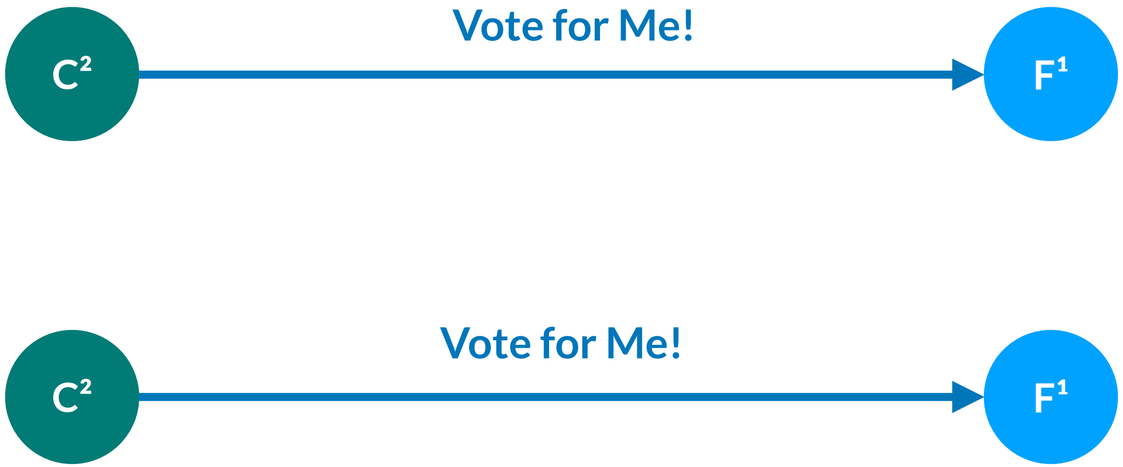


Both are now at Term 2

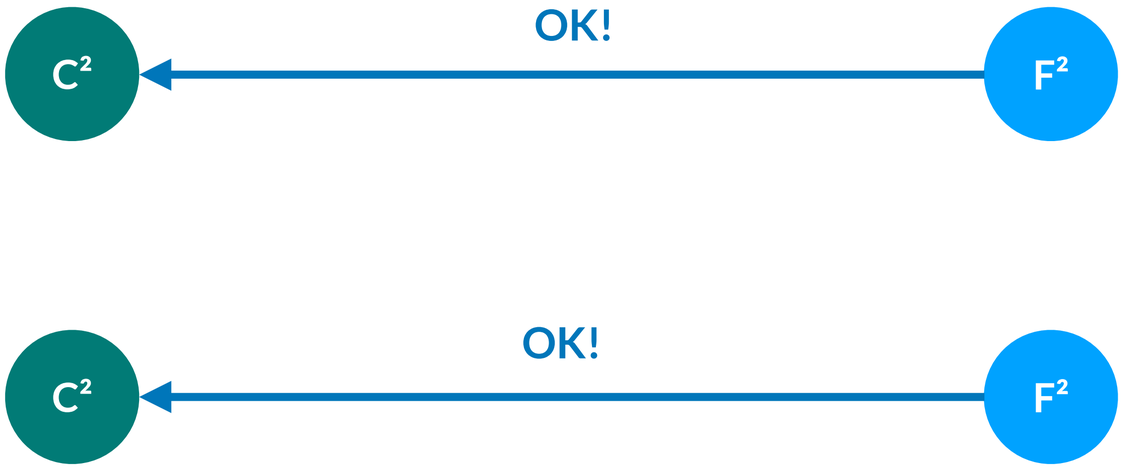

L²
Log Entries and Heartbeats
F²
Log Entries and Heartbeats
F²

Split Votes

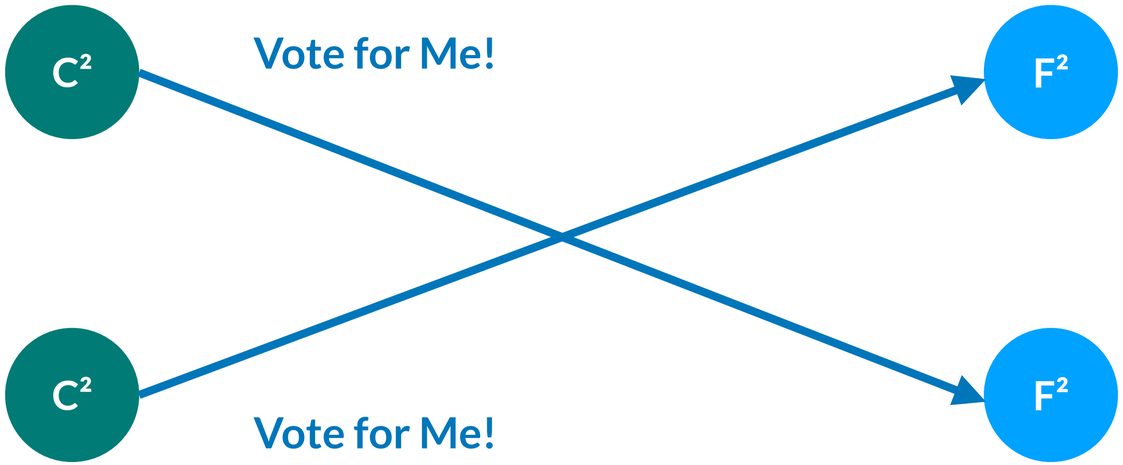
Two followers become candidates and being requesting votes



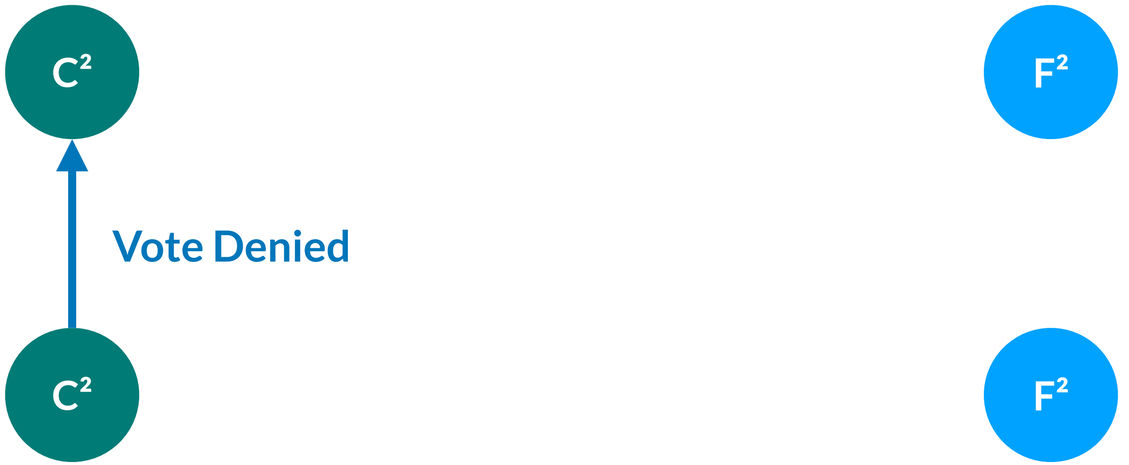
Two followers become candidates and being requesting votes



Each candidate receives a vote from themselves and one follower peer, but that is not enough

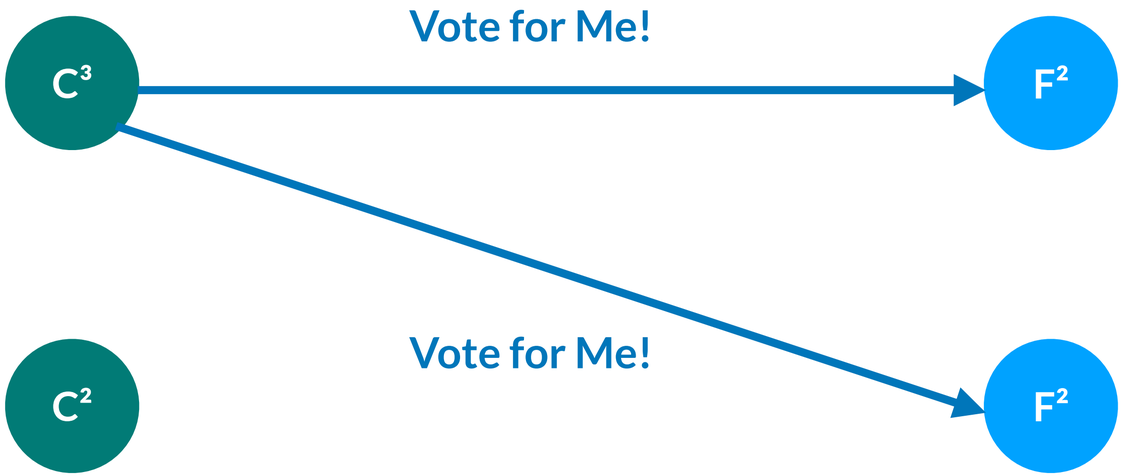


Each candidate now tries to ask for votes from a follower who has already voted

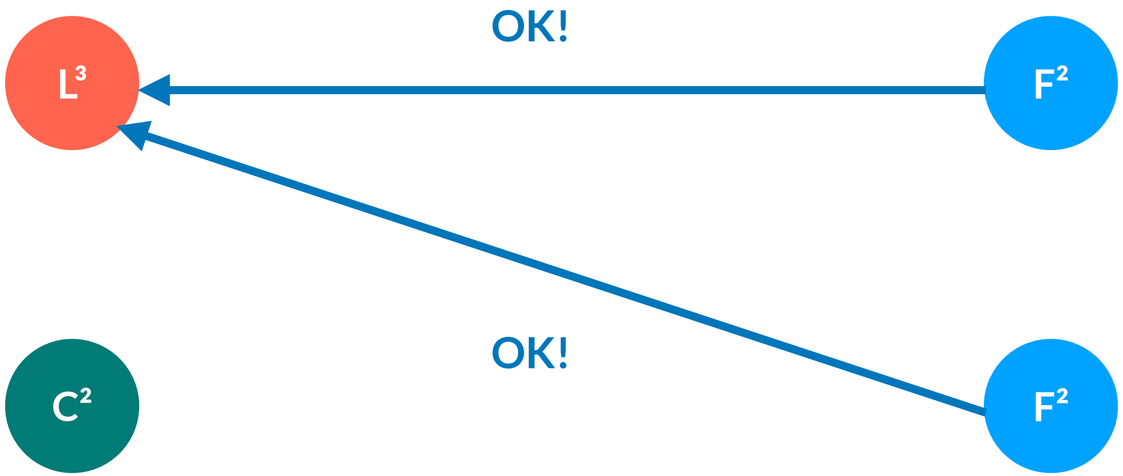


Vote Requests are denied since they already voted for themselves

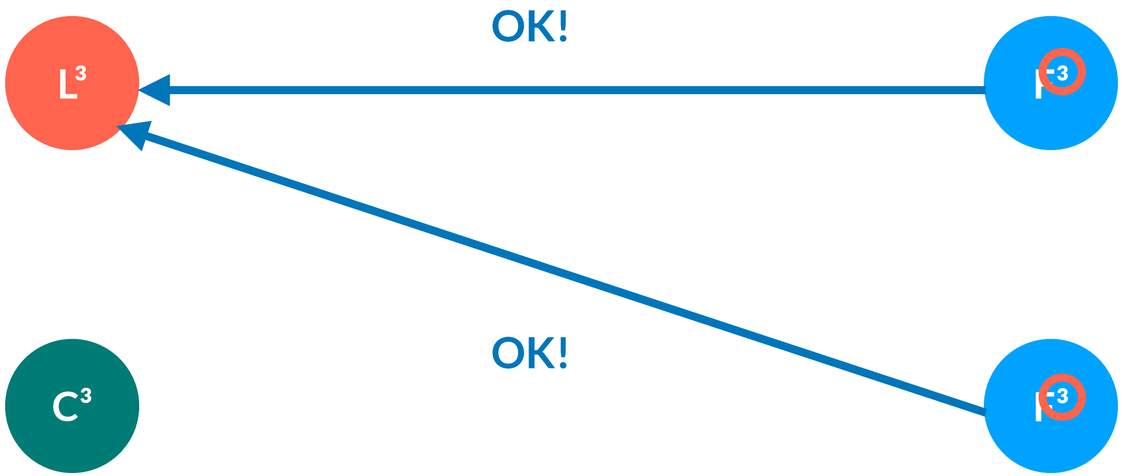
Stalemate. Candidates wait for another random election timeout to occur (150ms - 300ms)



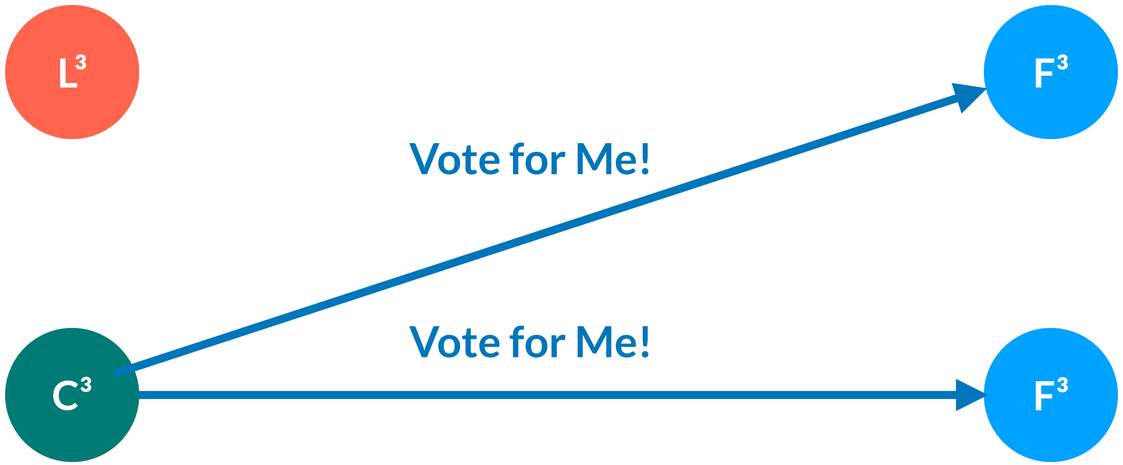
Let’s try again. We are now in Term 3



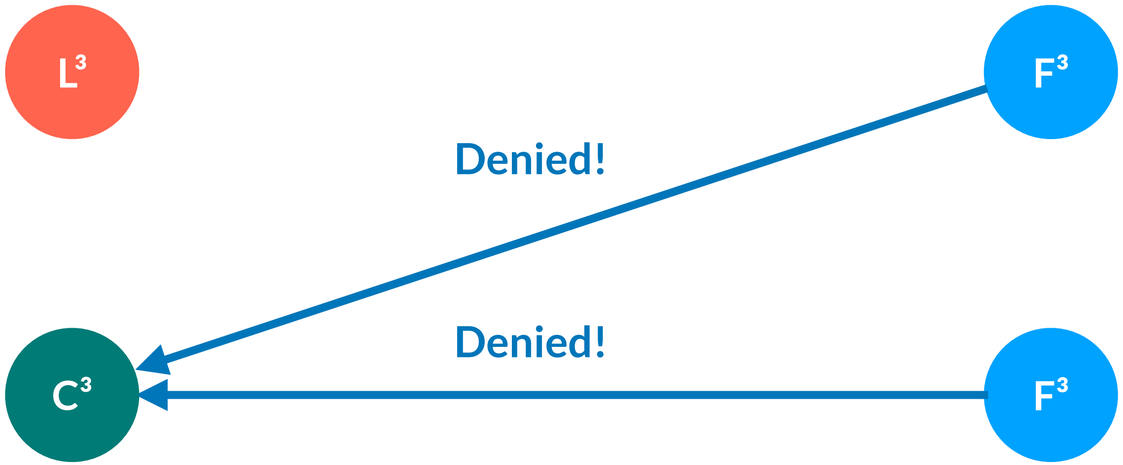
Leader becomes elected, but there is still one candidate!



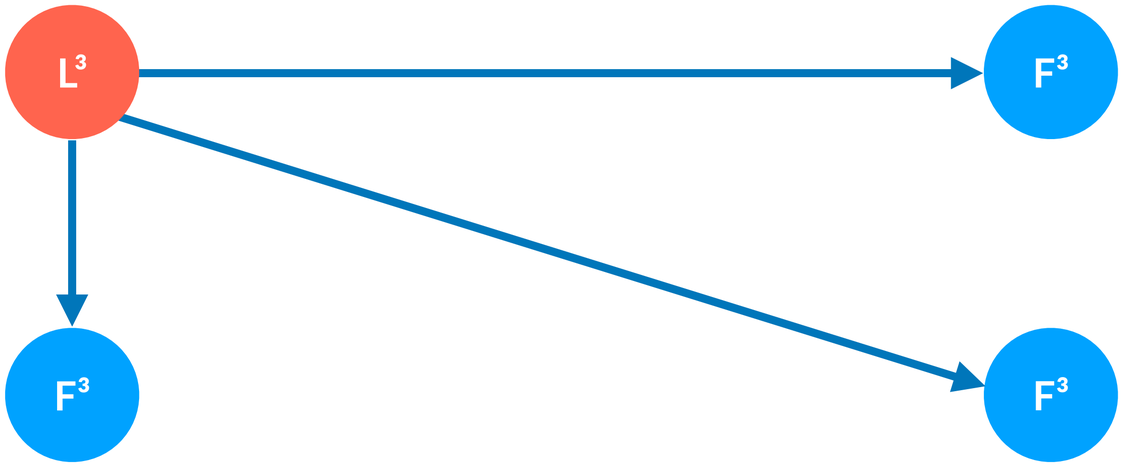
Leader becomes elected, but there is still one candidate!



It will not try to increase its term and run for leader



It will not try to increase its term and run for leader



Leader 3 will assert itself as the leader

The Raft Consensus Algorithm

What is Raft?

Raft is a consensus algorithm that is designed to be easy to understand. It's equivalent to Paxos in fault-tolerance and performance. The difference is that it's decomposed into relatively independent subproblems, and it cleanly addresses all major pieces needed for practical systems. We hope Raft will make consensus available to a wider audience, and that this wider audience will be able to develop a variety of higher quality consensus-based systems than are available today.

Quick Links

- Raft paper
- raft-dev mailing list
- Raft implementations

Hold on—what is consensus?

Consensus is a fundamental problem in fault-tolerant distributed systems. Consensus involves multiple servers agreeing on values. Once they reach a decision on a value, that decision is final. Typical consensus algorithms make progress when any majority of their servers is available; for example, a cluster of 5 servers can continue to operate even if 2 servers fail. If more servers fail, they stop making progress (but will never return an incorrect result).

Consensus typically arises in the context of replicated state machines, a general approach to building fault-tolerant systems. Each server has a state machine and a log. The state machine is the component that we want to make fault-tolerant, such as a hash table. It will appear to clients that they are interacting with a single, reliable state machine, even if a minority of the servers in the cluster fail. Each state machine takes as input commands from its log. In our hash table example, the log would include commands like *set x to 3*. A consensus algorithm is used to agree on the commands in the servers' logs. The consensus algorithm must ensure that if any state machine applies *set x to 3* as the n^{th} command, no other state machine will ever apply a different n^{th} command. As a result, each state machine processes the same series of commands and thus produces the same series of results and arrives at the same series of states.

Raft Visualization

Here's a Raft cluster running in your browser. You can interact with it to see Raft in action. Five servers are shown on the left, and their logs are shown on the right. We hope to create a screencast soon to explain what's going on. This visualization (RaftScope) is still pretty rough around the edges; pull requests would be very welcome.

https://raft.github.io/

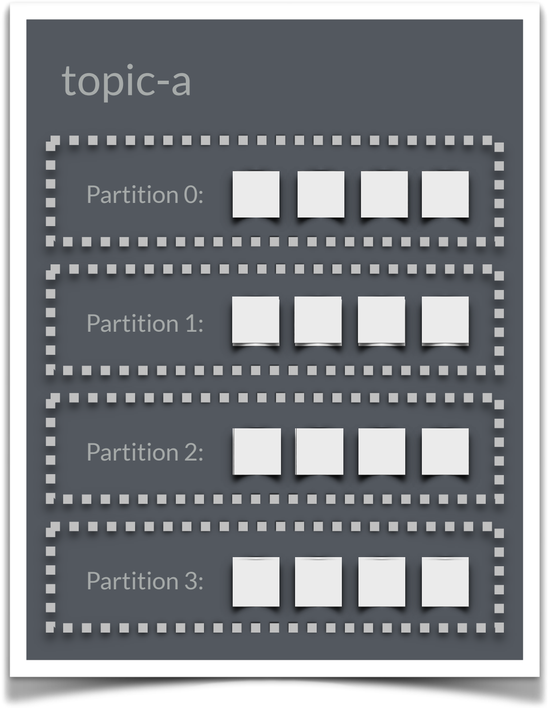
Competing Consumers

The Problem

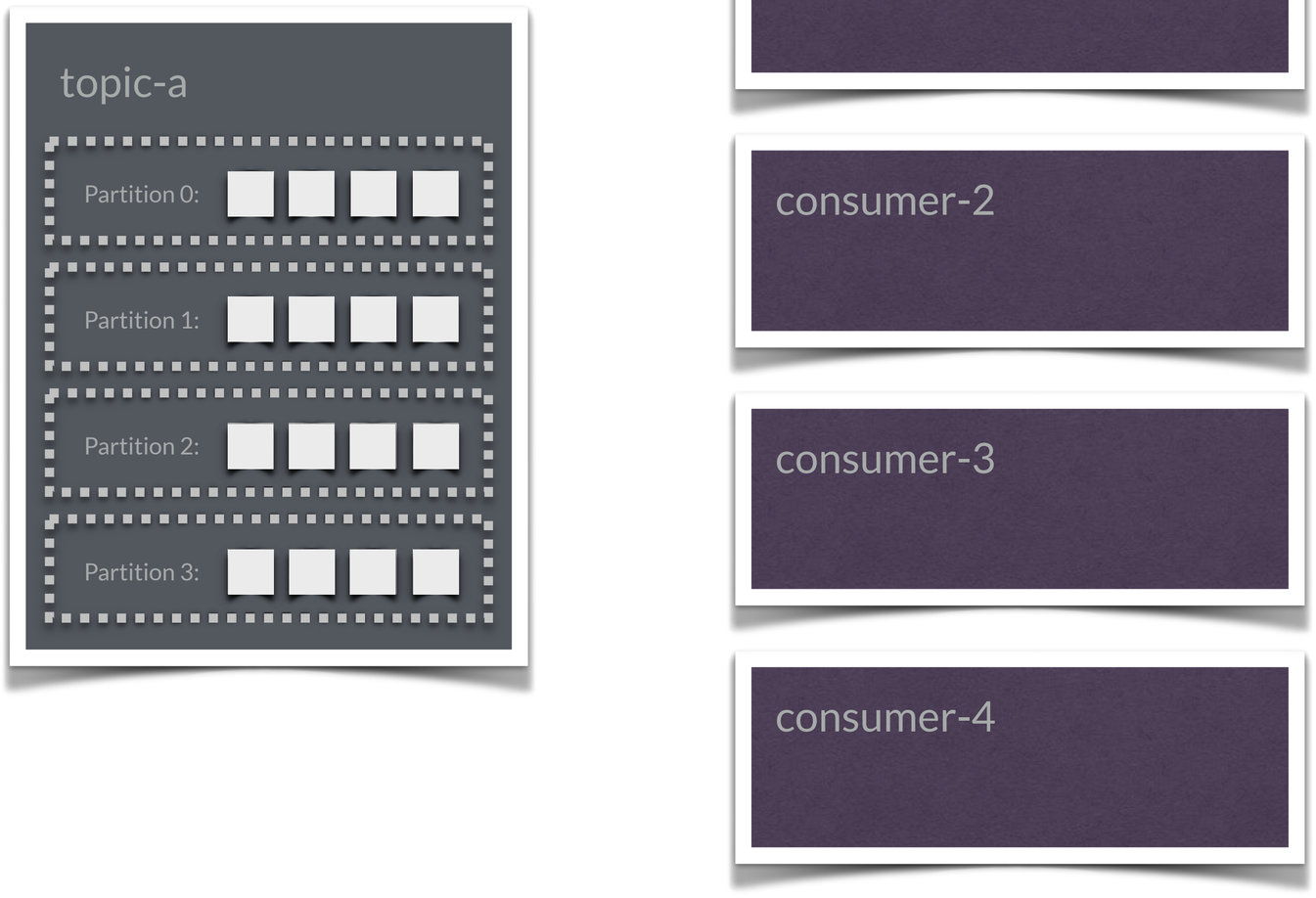
- Systems often need to process large volumes of tasks or messages from a queue.
- Single-threaded or single-consumer systems struggle with:
- High latency due to task backlog.
- Lack of fault tolerance if the consumer fails.
- How can systems scale task processing efficiently while maintaining reliability and fault tolerance?

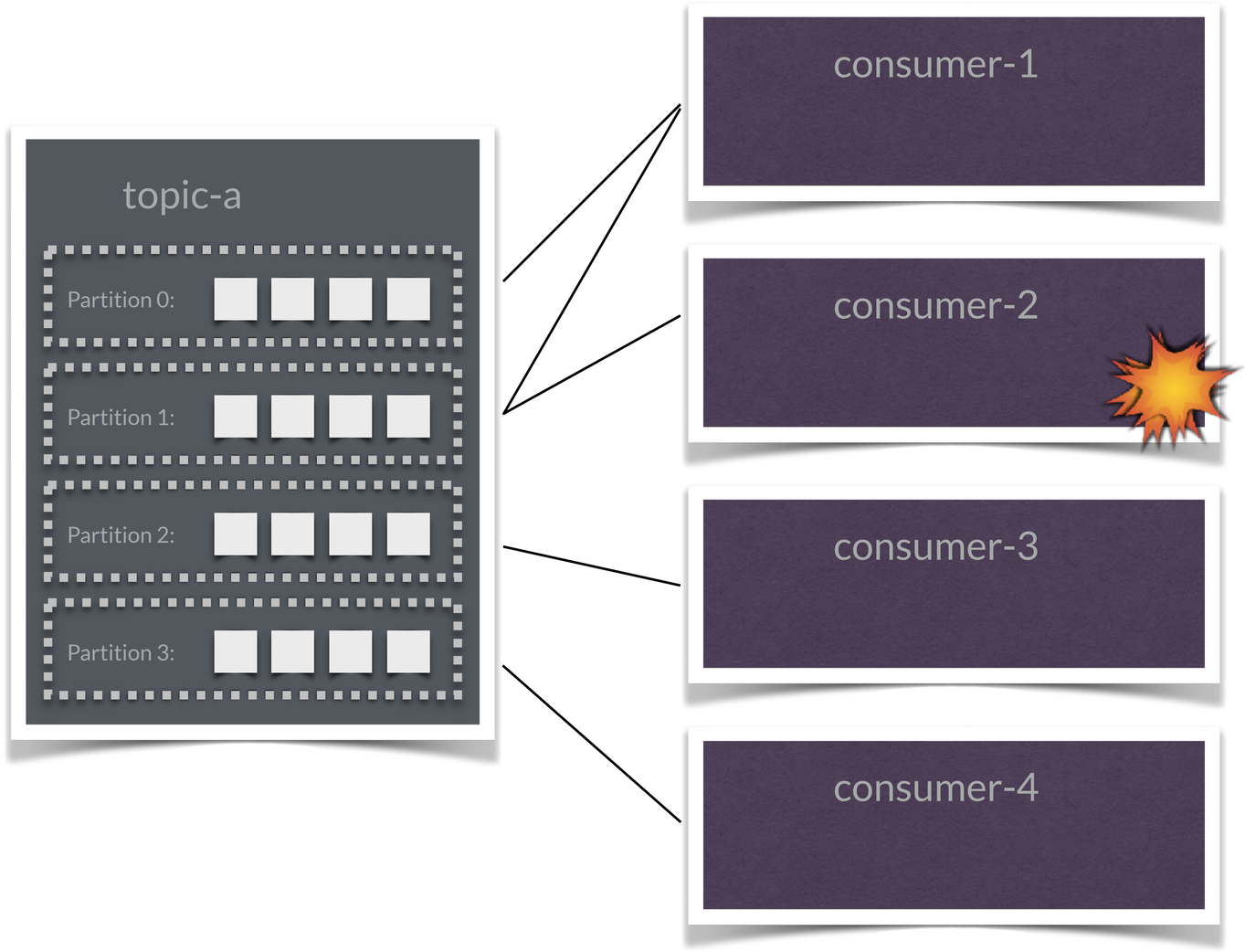
The Solution

- Competing Consumers is a pattern where multiple consumer instances process messages or tasks from a shared queue to balance the workload and increase throughput.
  - Tasks are published to a queue by producers.
  - Multiple consumer instances pull tasks from the queue concurrently.
  - Each task is processed by a single consumer, ensuring no duplication.
  - Distribute workloads efficiently across consumers to handle high traffic and achieve horizontal scalability.


topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:


consumer-1


topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:
consumer-1
consumer-2
consumer-3
consumer-4


topic-a
Partition 0:
Partition 1:
Partition 2:
Partition 3:
consumer-1
consumer-2
consumer-3
consumer-4

The Tradeoffs

- **Advantages:**
  - **Increased Throughput:** Multiple consumers process tasks concurrently, reducing bottlenecks.
  - **Fault Tolerance:** If a consumer fails, the queue redistributes tasks to other consumers.
  - **Scalability:** Horizontal scaling allows handling of larger workloads by adding more consumers.
  - **Decoupling:** Producers and consumers operate independently, improving flexibility.
- **Drawbacks:**
  - **Complexity:** Managing multiple consumers and ensuring message processing consistency can be challenging.
  - **Ordering Issues:** Task order may not be preserved unless explicitly handled.
  - **Resource Contention:** Consumers may compete for limited system resources, leading to potential inefficiencies.
  - **Dead Letter Queues:** Requires handling for unprocessable tasks to avoid queue clogging.

Technology Stacks

- **Message Brokers and Queues:**
  - **RabbitMQ:** Built-in support for multiple consumers and message acknowledgment.
  - **Apache Kafka:** Supports multiple consumer groups for distributed task processing.
  - **Amazon SQS:** A scalable queue service with competing consumer capabilities.
  - **Google Pub/Sub:** Ensures reliable message delivery and supports competing consumers.
- **Frameworks and Libraries:**
  - **Celery (Python):** Task queue framework that supports competing consumers.
  - **Spring Cloud Stream (Java):** Integrates with message brokers to implement competing consumers.
  - **Akka Streams (Scala/Java):** Allows distributed task processing in actor-based systems.
- **Serverless Platforms:**
  - **AWS Lambda:** Competing Lambda functions process events from SQS or EventBridge.
  - **Azure Functions:** Can consume messages from Service Bus queues in parallel.

Replicator

The Problem

- Distributed systems require data to be available across multiple nodes for:
  - Fault tolerance.
  - High availability.
  - Load balancing.
- Without replication:
  - Data loss can occur if a single node fails.
  - Read-heavy systems face bottlenecks.
  - System downtime increases during failures.

The Solution

- The Replicator pattern involves duplicating data or processes across multiple nodes or systems to ensure redundancy,
- This will improve read performance, and enhance system resilience.
- **Process:**
  - Data is copied from a primary node (or source) to one or more secondary nodes (or replicas).
  - Replication can be synchronous (real-time consistency) or asynchronous (eventual consistency).
  - Replicated nodes can handle read traffic, act as backups, or share the workload.

Replication Strategies

- **Primary-Replica:** A designated primary node handles write operations, while replicas handle read operations.
- **Multi-Primary:** Multiple nodes can perform read and write operations, resolving conflicts dynamically.
- **Peer-to-Peer:** Each node acts as both a source and destination for replication, ensuring equality among nodes.

Tradeoffs

- **Advantages:**
  - High Availability: Replicas ensure continuous service even if some nodes fail.
  - Improved Read Performance: Distributes read traffic across replicas, reducing load on the primary node.
  - Fault Tolerance: Protects against data loss with redundant copies.
  - Scalability: Supports horizontal scaling for read-heavy workloads.
- **Drawbacks:**
  - Consistency Challenges: Maintaining data consistency across replicas can be complex.
  - Latency: Synchronous replication introduces delays for write operations.
  - Conflict Resolution: Multi-master replication requires handling conflicts effectively.
  - Storage Overhead: Replication increases data storage requirements.

Technology Stacks

- **Databases:**
  - **Relational Databases:** MySQL, PostgreSQL.
  - **NoSQL Databases:** MongoDB, Cassandra
- **Message Brokers:**
  - **Apache Kafka:** Supports topic replication to ensure fault tolerance and data durability.
  - **RabbitMQ:** Allows queue replication across nodes.
- **Storage Systems:**
  - **HDFS (Hadoop Distributed File System):** Ensures file replication across clusters for fault tolerance.
  - **Ceph:** Provides data replication for distributed storage.
- **Service Meshes and Middleware:**
  - **Etcd:** Replicates key-value pairs using Raft consensus.
  - **Redis Sentinel:** Manages Redis replication for high availability.

Idempotency

The Problem

- Distributed systems and networks often face scenarios where:
- Requests are unintentionally repeated (e.g., due to retries or user errors).
- Duplicate operations lead to inconsistent states or unintended side effects.
- Issue can include:
  - Double payments in financial systems.
  - Redundant database writes causing data corruption.
  - Reprocessing an already fulfilled order.

The Solution

- Idempotency ensures that performing the same operation multiple times produces the same result as performing it once.
  - Define operations in a way that they are safe to repeat.
  - Use unique identifiers (e.g., request IDs) or state-checking mechanisms to detect duplicates.
  - Handle repeated requests without causing changes beyond the initial execution.
- Ensures system reliability, consistency, and correctness in the presence of retries or duplicate requests

Idempotency in HTTP

- Common HTTP methods like `GET`, `PUT`, and `DELETE` are naturally idempotent:
  - `GET`: Fetching the same resource repeatedly does not change its state.
  - `PUT`: Updating a resource with the same data multiple times results in no additional changes.
  - `DELETE`: Deleting the same resource repeatedly is harmless.

Tradeoffs

- **Advantages:**
  - **Reliability:** Handles retries gracefully, ensuring system consistency.
  - **Correctness:** Prevents unintended side effects from duplicate operations.
  - **User Experience:** Avoids errors caused by repeated actions, like double payments.
- **Drawbacks:**
  - **Increased Complexity:** Requires additional logic to track and handle duplicates.
  - **State Dependency:** Systems must often maintain resource state or request history, increasing overhead.
  - **Storage Overhead:** Tracking unique identifiers or operation states can increase resource usage.
  - **Latency:** Validation of state or request uniqueness can introduce delays.

Technology Stacks

- **API Gateways and Frameworks:**
  - **AWS API Gateway**: Supports idempotency keys for safe retries.
  - **Kong Gateway:** Custom plugins can enforce idempotency.
- **Databases:**
  - **PostgreSQL:** Use UPSERTs (INSERT ON CONFLICT) to handle duplicate writes safely.
  - **Redis:** Use as a store for request IDs or tokens to detect duplicates.
- **Message Brokers:**
  - **Apache Kafka:** Provides message offsets to ensure messages are processed exactly once.
  - **RabbitMQ:** Use acknowledgment and deduplication strategies to avoid duplicate processing.
- **Application Frameworks:**
  - **Spring Boot (Java):** Support for idempotency with request tracking and conditional updates.
  - **FastAPI/Django (Python):** Custom middleware can enforce idempotent operations.

Shadowing

The Problem

- Introducing new features, code, or configurations into a production environment poses risks:
  - Unforeseen bugs or regressions.
  - Performance degradation under real-world loads.
  - Poor user experience if errors occur.
- Examples include:
  - A new payment gateway integration causing transaction failures.
  - Unexpected resource contention due to unoptimized queries.
- How can you safely test new changes in a production-like environment without affecting live users?

The Solution

- Shadowing, also called “Shadow Testing” or “Dark Launching”, involves running a copy of live production traffic against a new version of the system or feature without affecting the actual production environment.
- Process:
  - Production traffic is mirrored to a shadow environment containing the new version.
  - Results from the shadow environment are observed but not returned to live users.
  - Issues are identified and resolved before full deployment.
- Test new systems, features, or configurations with real-world data and workloads without impacting production.

Shadowing Strategies

- **Traffic Mirroring:**
  - Duplicate live user requests and send them to the shadow environment.
- **Shadow Environment:**
  - A replica of the production system running the new version.
- **Result Monitoring:**
  - Compare the shadow environment’s responses, performance, and errors against the production environment.

Mirroring in NGINX

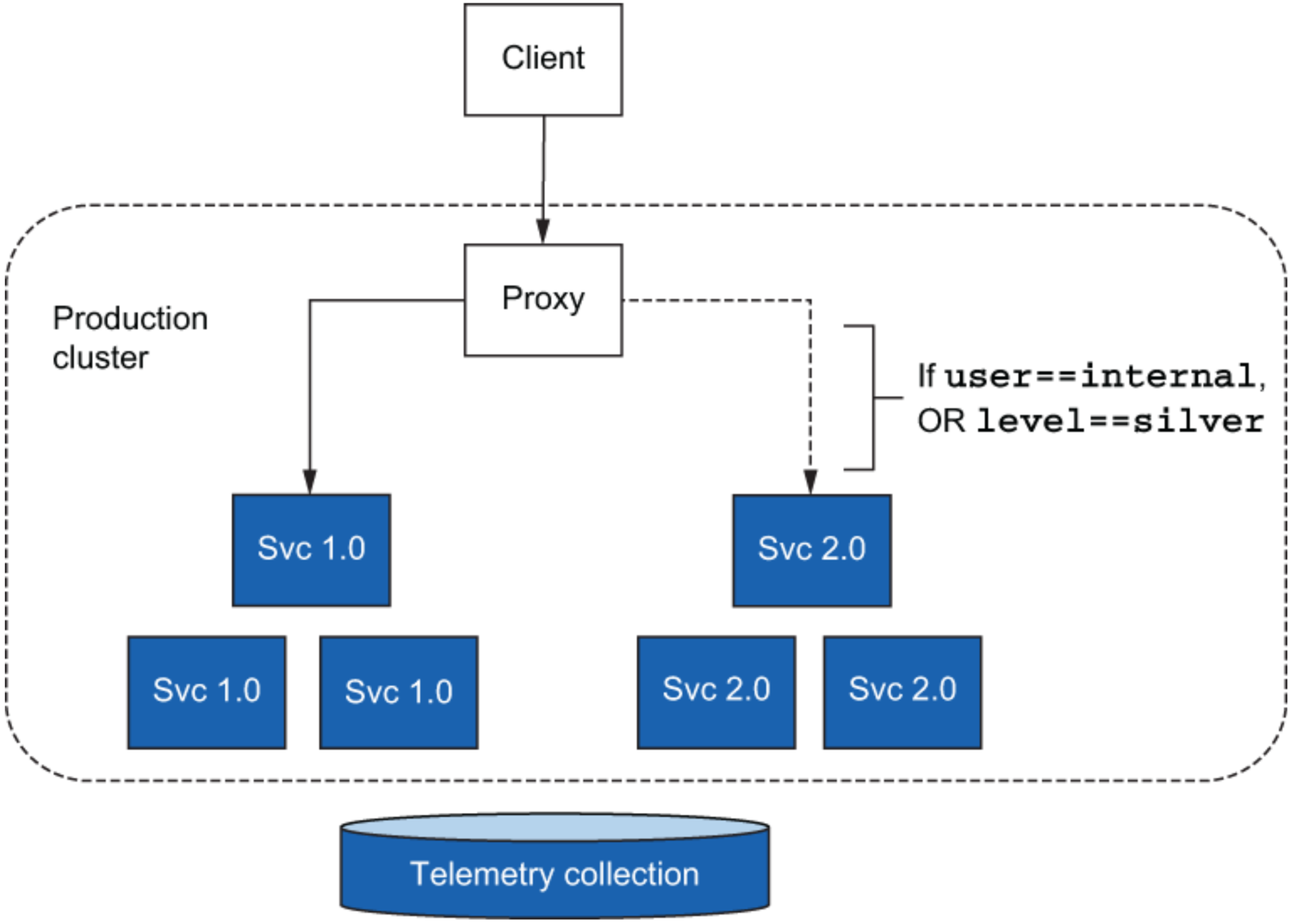
```
location / {
    mirror /mirror;
    proxy_pass http://backend;
}

location = /mirror {
    internal;
    proxy_pass http://test_backend$request_uri;
}
```

https://nginx.org/en/docs/http/ngx_http_mirror_module.html

Mirroring in Istio

```
spec:
  hosts:
- httpbin
  http:
  - route:
- destination:
    host: httpbin
    subset: v1
  weight: 100
mirror:
  host: httpbin
  subset: v2
mirrorPercentage:
  value: 100.0
```

https://istio.io/latest/docs/tasks/traffic-management/mirroring/



Source: Istio in Action Rinor Maloku, Christian E. Posta Published by Manning Publications

The Tradeoffs

- **Advantages:**
  - **Risk-Free Testing:** Validate changes without affecting live users.
  - **Realistic Workloads:** Test with production-like traffic and data for accurate results.
  - **Early Detection:** Identify bugs, regressions, or performance issues before full rollout.
- **Drawbacks:**
  - **Infrastructure Overhead:** Requires maintaining a shadow environment with resources mirroring production.
  - **Traffic Duplication Complexity:** Mirroring traffic without introducing latency or inconsistency can be challenging.
  - **Data Sensitivity:** Sensitive production data needs to be handled securely in the shadow environment.
  - **Cost:** Running shadow systems incurs additional expenses.

Technology Stacks

- **Traffic Mirroring Tools:**
  - **Envoy Proxy**: Supports traffic mirroring to shadow services.
  - **NGINX:** Can duplicate incoming requests to shadow environments.
  - **AWS Load Balancers:** Offer built-in traffic mirroring for testing.
- **Monitoring and Observability Tools:**
  - **Prometheus/Grafana:** Monitor shadow environment performance and behavior.
  - **Datadog/New Relic:** Compare shadow and production environment metrics.
  - **OpenTelemetry:** Trace requests across both environments for detailed insights.
- **Testing Platforms:**
  - **Gremlin:** Simulates traffic patterns and chaos scenarios in shadow systems.
  - **TestContainers (Java):** Create shadow environments for microservices testing.
- **Cloud Platforms:**
  - **AWS Lambda & S3:** Mirror traffic to serverless environments for testing.
  - **Google Cloud Traffic Director:** Manage mirrored traffic distribution.

Graceful Degradation

The Problem

- Systems inevitably face partial failures due to:
  - Network disruptions.
  - Service outages.
  - Overloaded components.
- Complete failures or unresponsive behavior can lead to:
  - Poor user experience.
  - Loss of trust.
  - Business impact.

The Solution

- Graceful degradation ensures that when parts of a system fail, the remaining parts continue to operate, offering limited or reduced functionality instead of a complete shutdown.
- The Process:
  - Design the system to identify and isolate failing components.
  - Provide fallback or alternative functionality for critical features.
  - Communicate transparently with users about the reduced capabilities.
- Maintain a baseline level of service to minimize user impact and ensure system resilience.

Graceful Degradation Strategies

- **Service Prioritization:** Focus on keeping critical services operational while deprioritizing less important ones.
- **Fallback Mechanisms:** Provide cached data or defaults when real-time data isn’t available.
- **Component Isolation:** Use modular designs to prevent cascading failures.
- **Error Communication:** Notify users of partial failures with clear and actionable messages.

Examples of Degradation

- A streaming platform offering a reduced-quality video stream during network issues.
- E-commerce sites displaying cached product data when the database is down.
- A payment system allowing transactions to queue during gateway downtime.
- Switch to local processing when cloud connectivity is lost.

Tradeoffs

- **Advantages:**
  - **Improved User Experience:** Avoids complete outages by maintaining partial functionality.
  - **Resilience:** Limits the impact of failures on critical features.
  - **Trust Retention:** Transparent communication builds user confidence.
  - **Incremental Recovery:** Easier to bring services back online step-by-step.
- **Drawbacks:**
  - **Increased Complexity:** Requires additional design and testing for fallback mechanisms.
  - **User Expectations:** Limited functionality may not meet user needs or lead to frustration.
  - **Cost Overhead:** Maintaining fallback infrastructure can be resource-intensive.
  - **Not all Domains:** May not work for critical or high-stakes systems (e.g., healthcare or financial services).

Thank You

- Email: dhinojosa@evolutionnext.com
- Github: https://www.github.com/dhinojosa
- Mastodon: https://mastodon.social/@dhinojosa
- Linked In: http://www.linkedin.com/in/dhevolutionnext