Microservices Resiliency Patterns to instrument Enterprise Fault Tolerance.

Amit Sengupta (Independent Research)
Email – amits2913@gmail.com
Independent Researcher

Abstract

In a monolith application, a single error can bring down the entire system. This risk is reduced in a microservices architecture because it uses smaller, independently deployable units that don't affect the whole application or system.

Microservice application developers try to mitigate the impact of partial outages typically by implementing service-to-service interactions that use well-known resiliency patterns, such as Retry, Fail Fast, and Circuit Breaker. However, those resiliency patterns as well as their available open-source implementations are often documented informally, leaving it up to application developers to figure out when and how to use those patterns in the context of a particular microservice application.

Simply converting a monolith into microservices doesn't automatically fix all issues. Microservices heavily rely on distributed systems, making resiliency critical to their design and performance. When architecting distributed cloud applications, it's crucial to anticipate failures and design your applications with resiliency in mind. Microservices are likely to fail at some point, so it's essential to be prepared for failures and design your microservices to handle failures. Having said that, we'll discuss fault tolerance and failure recovery in microservices and how to achieve them.

Keywords: - Microservices, Resiliency and Reliability, Patterns and Anti-patterns, Retry Pattern, Circuit Breaker Pattern, Fallback Pattern, Bulkhead Pattern, Failover and Redundancy, Enhanced Customer Experience, Business Continuity,

---

Introduction: -

Resiliency is an integral aspect of designing, developing, and operating microservice architectures. Application-to-application communication over the network is inevitable in distributed systems. Anything can go wrong during app-to-app communication, like network glitches or timeouts, application unavailability, data center failovers, etc. Due to these events, applications may overload, stop responding, or even crash the entire system.

Lack of Resiliency with Traditional Systems: -

Traditional architectures and designs were not made for the complexity and distribution of microservices. And hence traditional resilience approaches, like redundancy in one application or relying on a single powerful server, might not be sufficient for microservices for several reasons:

Complexity: Microservices bring more complexity because they are distributed. Traditional methods that work in simpler architectures may struggle with microservices' complexities, such as managing service dependencies and handling network issues.

Single Point of Failure: Traditional methods often rely on one central system or server. If that fails, the whole application can go down. In microservices, the aim is to avoid this by having redundancy at different levels.

Resource Efficiency: Microservices allow for better resource use by scaling individual services independently. Traditional methods are less efficient because they scale entire applications, leading to unused resources.

Elasticity: Microservices can scale up and down quickly based on demand. Traditional systems may not be as elastic and can't adapt as fast.

Isolation and Containment: Microservices need to be isolated to prevent failures from spreading. Traditional methods might not have the right mechanisms for this.

Application Resiliency with Microservices: -

Resilience in microservices refers to an application's ability to withstand failures, stay available, and deliver consistent performance in distributed environments. Resilience patterns are established mechanisms that empower applications to handle failures gracefully, ensuring stability in complex, distributed systems. By using these patterns, developers can reduce the impact of unexpected errors or high loads, leading to less downtime and better overall performance.

In distributed systems, failures are unavoidable due to various factors like network issues, unresponsive services, or hardware problems. Hence, it's essential to acknowledge these uncertainties and develop strategies to manage them effectively. This is where resilience patterns come into the picture, helping create fault-tolerant systems that respond well to failures, ensuring the application remains available and functional. Implementing resilience patterns in microservices offers several key benefits:

- **Minimized Service Downtime:** These patterns help applications recover quickly from failures, minimizing disruptions and ensuring high availability for users.
- **Improved Fault Isolation:** By using resilience patterns, developers can isolate failures, preventing them from spreading and causing widespread issues.
- **Consistent System Performance:** A resilient microservices application can maintain consistent performance, even under high load or network issues.
- **Enhanced User Satisfaction:** Reliable performance improves user experience, building trust and loyalty.

Resiliency Patterns with Microservices –

Below are the commonly used resiliency patterns in microservices -

Resiliency Pattern	Short Description
Circuit Breaker Pattern	Fail fast in case of errors and enables you to perform the default or fallback operations.
Retry Pattern	Making several attempts to execute a failed remote operation before giving up and reporting it as an issue.
Timeouts/Time Limits	Set a time limit for a remote operation instead of indefinitely waiting for response.
Fallback Mechanism	Fallback mechanisms provide an alternative response or behaviour when a remote operation is failing. This can be like returning cached results or default values.
Bulkhead Pattern	The Bulkhead pattern involves isolating components of a system so that the failure of one component does not lead to the failure of the entire system.
Health Checks	Monitor the remote services and remove from the load balancer automatically or stop routing requests when it is unhealthy.
Failover and Redundancy	Redundancy and failover capabilities ensures that if one instance or component fails, another can take over.
Event/message-based communications	Adopt event/message-based communication wherever is possible during service-to-service communications. This decouples services and enables them to react to events at their own pace, improving overall resilience.

Circuit Breaker Pattern:

The circuit breaker pattern is used to prevent repeated calls to a failing service, which can overload the system and worsen the situation. The circuit breaker monitors the status of the service and "opens" the circuit when it detects a failure. Subsequent calls are then "short-circuited" and fail immediately, without making a request to the service. The Circuit Breaker pattern with fallback is a strategy used in microservices architecture to boost system resilience. Picture it as a safety net for your services. When a critical service, like a product catalog, starts acting up due to high traffic or errors, the circuit breaker kicks in and temporarily redirects the system to a backup plan the fallback mechanism. This can be displaying cached data or a simple message to users, preventing the entire system from going haywire. The circuit breaker periodically checks if the troubled service is back on track, automatically switching back to normal operations when it's ready. Essentially, it's a smart way to handle hiccups in microservices, ensuring smoother user experiences even when certain services hit a rough patch.

Retry Patterns:

Retrying failed operations is a common strategy to improve the robustness of microservices. By retrying a failed operation, the service has another chance to succeed, especially in cases where the failure is transient.

Additionally, when implementing the retry pattern, developers need to be mindful of potential performance implications. Retrying requests too frequently or for too long can put unnecessary strain on external systems and lead to degraded performance or even service outages.

Retry policies come in a variety of forms, including fixed, exponential, and random. Finally, which retry policy to use will be determined by the specific needs and priorities of the system being designed. A good retry policy will strike a balance between reliability and speed, and it will be designed to adapt to changing network conditions over time.

Timeouts:

The Timeout pattern is a resiliency pattern that helps to prevent system overload and improve the overall reliability of a microservice architecture. In this pattern, a timeout is set for each operation or service call, and if a response is not received within that time limit, the operation is considered to have failed.

The Timeout pattern is designed to prevent a microservice from being stuck waiting for a response that may never come, which can lead to the system becoming unresponsive or even crashing. By setting a timeout for each operation, the microservice can move on to other tasks and prevent system overload. When implementing the Timeout pattern, it's important to choose an appropriate timeout value for each operation. The timeout value should be set based on the expected response time of the service or operation, and it should be long enough to allow for normal operation but short enough to prevent the system from becoming overloaded.

In addition, it's important to handle timeouts gracefully by providing appropriate error messages or fallback responses to the client. This can help to ensure that the client is aware of the timeout and can take appropriate action, such as retrying the operation or using an alternative service. The Timeout pattern can be used in conjunction with other resiliency patterns, such as the Circuit Breaker pattern, to provide even greater resiliency and reliability to a microservice architecture. By implementing the Timeout pattern, you can help to prevent system overload, improve reliability, and ensure the overall resiliency of your microservice system.

Fallback Mechanism:

Fallback provides an alternative solution during a service request failure. When the circuit breaker trips and the circuit is open, a fallback logic can be started instead. The fallback logic typically does little or no processing and return value. Fallback logic must have little chance of failing, because it is running as a result of a failure to begin with.

Each service that's wrapped by a circuit breaker can implements a fallback using one of the following three approaches:

- ***Custom fallback:*** In some cases, a service's client library provides a fallback method we can invoke, or in other cases we can use locally available data on an API server (eg, a cookie or local JVM cache) to generate a fallback response

- ***Fail silent:*** In this case the fallback method simply returns a null value, which is useful if the data provided by the service being invoked is optional for the response that will be sent back to the requesting client
- ***Fail fast:*** Used in cases where the data is required or there's no good fallback and results in a client getting a 5xx response. This can negatively affect the device UX, which is not ideal, but it keeps API servers healthy and allows the system to recover quickly when the failing service becomes available again.

Bulkhead Pattern:

The Bulkhead Pattern in software architecture divides a system's various parts or subsystems into isolated groups so that the other groups can continue to operate normally even if one group fails or becomes overburdened. In essence, the pattern makes sure that every group of components has access to exclusive resources and is not hampered by the failures or traffic of other groups.

The bulkhead pattern gets its name from a naval engineering practice where ships have internal chambers isolating their hull so that if a rock cracks it, water can't spread to the entire ship causing it to sink. If the hull of a ship is compromised, only the damaged section fills with water, which prevents the ship from sinking.

The Bulkhead pattern is designed in a way where elements of an application are isolated into pools so that if one fails, the others will continue to function. The ability of the system to recover from the failure and remain functional makes the system more resilient.

Health Checks:

A health check is a special REST API implementation that you can use to validate the status of a microservice and its dependencies. A health check can assess anything that a microservice needs, including dependencies, system properties, database connections, endpoint connections, and resource availability. The overall status of the microservice depends on whether all the configured health checks pass. A microservice is considered available and reports an UP status if all the configured health checks are successful. If one or more health checks fail, the microservice is considered unavailable and reports a DOWN status. Services can report their availability to a defined endpoint by implementing the API that is provided. A service orchestrator can use these status reports to decide how to manage and scale the microservices within an application.

Failover and Redundancy:

Microservices Failover refers to the process of automatically resolving the failure of one or more microservices, ensuring the overall functionality, availability, and resilience of the application. Failover mechanisms are vital to maintain the uninterrupted flow of services in distributed systems, as they address the potential points of failure and enable seamless recovery from outages or errors.

As applications grow big and complex, the number of microservices increases, and so does the possibility of failures. Failures in microservices can be attributed to various reasons, such as hardware issues, network latency, software bugs, or even human errors. This is where Microservices Failover comes into play, offering a set of mechanisms that ensure the application continues to function and serve its users in the face of failures.

There are several strategies that can be employed for Microservices Failover, including:

- **Load balancing:** Distributing the workload across multiple instances of a microservice ensures that no single instance bears the burden of excessive traffic, reducing the risk of failure due to overload. This can be achieved through various algorithms such as Round Robin, Least Connections, or even custom heuristics.
- **Health monitoring and automated failover:** Regularly checking the health of individual microservice instances offers insights into their performance and load levels. By detecting failing instances early on, it is possible to prevent cascading failures and route traffic to healthy instances.
- **Automatic recovery and self-healing:** In case a microservice instance fails, the system should automatically provision new instances to maintain the desired level of redundancy and load distribution. Container

orchestration tools such as Kubernetes or Docker Swarm provide self-healing capabilities that manage the lifecycle of instances and ensure appropriate failover.

- **Failover through Circuit breaking:** Circuit breaking is a pattern that prevents overwhelming a failing microservice by temporarily limiting the traffic sent to that service. Service Mesh are specifically designed to provide circuit breaking functionality, allowing developers to define policies for gracefully handling failures and maintaining overall system stability.
- **Failover through Retry and timeout policies:** Implementing intelligent retry mechanisms and timeout policies can help alleviate the impact of transient failures in microservices. These policies should be defined depending on the specific requirements and characteristics of each microservice, taking into account factors such as response time, error rate, error budget etc.

Event Driven Patterns:

Event-driven architecture (EDA) is used in microservices where every microservice in the ecosystem can asynchronously publish and subscribe to events via an event broker.

EDA provides a flexible, scalable, and real-time approach to processing actions and responses quickly. It is ideal for managing high volumes and high-velocity data in real time with a minimum time lag. It can handle complex event processing, such as pattern matching or aggregation over time windows.

There are design patterns within EDA which can help you elevate scalability, resilience and maintainability of the systems. These patterns provide solutions to common challenges and guide the organization of components and their interactions. Below listed are three popular design patterns which is widely used -

- **Event Sourcing**
  Event Sourcing is a design pattern that involves capturing all changes to an application's state as a sequence of events. Instead of storing the current state of an entity, the system stores a log of events that represent state transitions.
- **Command Query Responsibility Segregation (CQRS)**
  CQRS is a design pattern that separates the read model (queries) and the write model (commands) of a system. By splitting these responsibilities, CQRS enables independent scaling and optimization of read and write operations.
- **Saga Pattern**
  The Saga Pattern is a design pattern that helps manage distributed transactions in a microservices or event-driven architecture. A saga is a sequence of local transactions, where each transaction is coordinated by an event. If one transaction fails, compensating transactions are executed to undo the previous steps and maintain data consistency.

Conclusion: -

Ensuring resilience in microservices architectures is key to maintaining high availability, performance, and customer satisfaction. By implementing comprehensive fault tolerance and failure recovery strategies, organizations can protect their systems against inevitable failures and minimize their impact. As technology evolves, so will the tools and techniques for building resilient systems, requiring ongoing attention and adaptation to best practices in microservices architecture.

Reference: -

1. B. Beyer et al., Site Reliability Engineering: How Google Runs Production Systems. O'Reilly, 2016.
2. J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014, [Online; last access on February 25, 2020].
3. P. Jamshidi et al., "Microservices: The journey so far and challenges ahead," IEEE Software, vol. 35, no. 3, pp. 24–35, 2018.

4. Microsoft Azure, "Resiliency patterns," https://docs.microsoft.com/enus/azure/architecture/patterns/category/resiliency, 2017, [Online; last accessed on February 25, 2020].
5. Microsoft Azure, "Retry Pattern," https://docs.microsoft.com/en-us/ azure/architecture/patterns/retry, 2017, [Online; last accessed on February 25, 2020].
6. F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," arXiv preprint arXiv:1609.05830, 2016.
7. D. Preuveneers and W. Joosen, "QoC2 Breaker: intelligent software circuit breakers for fault-tolerant distributed context-aware applications," Journal of Reliable Intelligent Environments, vol. 3, no. 1, pp. 5–20, 2017.
8. M. Nygard, Release It!: Design and Deploy Production-Ready Software. Pragmatic Bookshelf, 2007.