The Scientific Temper (2024) Vol. 15 (3): 2564-2569
Doi: 10.58414/SCIENTIFICTEMPER.2024.15.3.23
E-ISSN: 2231-6396, ISSN: 0976-8653
https://scientifictemper.com/
RESEARCH ARTICLE

A resilience framework for fault-tolerance in cloud-based microservice applications

Punithavathy E.[1*], N. Priya[2]

Abstract

Cloud-distributed systems offer significant opportunities for fault-tolerant applications. Microservices have gained significant acceptance as a cloud-based architecture for building fault-tolerant cloud applications. The primary aim of this study is to develop a dependable resilience framework, incorporating appropriate design patterns, that can be applied to any cloud application. This framework combines a bulkhead utilizing a little law approach and an auto-retry circuit breaker, which can be seen as a fault tolerance pattern. This will eliminate the need for manual setting of design patterns, resulting in maximum throughput availability of resources and the performance can be increased up to 55.3% from the average execution duration.

Keywords: Bulkhead, Little law, Fault tolerance, Auto retry circuit breaker, Resilience, Framework, Microservices.

Introduction

The microservice architecture is ideal for developing cloud-based applications because of its high productivity, cost-effectiveness, and ease of deployment (Muzaffar *et al*., 2020). This architecture ensures the main factors of cloud-based applications, such as scalability and availability, are guaranteed (Hassan *et al*.,2023). Applications can be programmed using many programming languages and designed as services, providing an additional advantage by addressing the primary issue of application updates. Despite their benefits, fault-tolerant features in this domain are still in the process of improvement (Hylbovets and Paprotskyi, 2024). Applications hosted on distributed systems are prone to failure. Network failures can arise due to various factors, including the intricacy of the network, challenges in synchronization, and difficulties in managing partial failures.

If there is an overwhelming demand or a service failure, it will have an adverse effect on the entire system, causing many failures (Fu *et al*.,2023). When a service experiences a prolonged response time to a request, other requests will be unable to utilize that particular service during that period. In order to address the aforementioned issue, the microservice architecture employs a specific design pattern known as resilience pattern to safeguard all services from network outages. The resilience pattern is highly advantageous for shielding applications against the propagation of faults or cascading of failures.

Cascading failures occur when the failure of one service causes other unaffected services to also fail. These can result in significant repercussions and extensive disruptions across various industries. In order to prevent these issues, cloud applications prioritize resilience characteristics, which enable them to avoid failure scenarios and recover from them. These durable characteristics are only effective in the presence of temporary breakdowns.

Transient failures are failures that can automatically recover within a certain period of time. If these issues are not properly addressed it results in a complete application outage lasting many hours. Failures can occur as a result of a rapid surge in requests, which is also known as request overhead. In such instances, the service request can be temporarily halted, at which time the services will be provided an idle time and gradually restored. This is

[1]Department of Computer Applications, Madras Christian College, Affiliated to University of Madras, Chennai, India.

[2]PG Department of Computer Science, Shrimathi Devkunvar Nanalal Bhatt Vaishnav College for Women, Affiliated to University of Madras, Chennai, India.

***Corresponding Author:** Punithavathy E., Department of Computer Applications, Madras Christian College, Affiliated to University of Madras, Chennai, India., E-Mail: punithavathy@mcc.edu.in

How to cite this article: Punithavathy, E., Priya, N. (2024). A resilience framework for fault-tolerance in cloud-based microservice applications. The Scientific Temper, **15**(3):2564-2569.

Doi: 10.58414/SCIENTIFICTEMPER.2024.15.3.23

Source of support: Nil

Conflict of interest: None.

Received: 08/07/2024 **Accepted:** 08/08/2024 **Published** : 10/09/2024

accomplished by utilizing the existing resilience patterns of microservice architecture. These resilience patterns provide a solution for cloud services that experience these temporary failures.

The issue with the current resilience patterns is that they rely on static or manual setups, which limits their functionality. The static configuration may not provide a comprehensive solution for all the issues in a distributed system, as the failure types vary significantly from one another. The objective of this research is to facilitate the automation of commonly employed resilience patterns, with the value being determined by the type of failure that happens. The primary goal is to present a framework that can be universally utilized by any cloud-based microservice application, without any worries regarding its setups. This framework implements a modified bulkhead pattern using little law in combination with auto retry circuit breaker, to promote resiliency for all types of transient failures.

Background

Resilience is the capacity to endure setbacks and bounce back from them. The primary objective is to facilitate uninterrupted operations and minimize downtime and disturbances for users of cloud applications. Microservice architectures provide resilience through the use of specific design patterns, including the circuit breaker, retry, and bulkhead patterns (Norton & Shoney,2023). These patterns are utilized in many areas, including fault detection and fault isolation. The retry pattern is the simplest among all these patterns (Frank *et al.*,2021). The circuit breaker is a fault detection pattern that simplifies the handling of breakdowns in a system when they occur. At the user's end, the fallback technique is employed without any notification of failure (Rahman,2022).

Circuit breaker

This design pattern is identical to the one used in electrical applications. This design pattern is employed for service monitoring and maintaining a record of successful and unsuccessful requests. When the number of failures reaches the specified count, an action is triggered. The service experiencing a problem is contained and the incoming requests are being responded to with temporary alternative statements (Norton & Shoney, 2023). As shown in Figure 1, this pattern consists of three states: open, half-open, and closed. The open state will not redirect the request to the designated service. The half-open state will initiate a limited number of queries to the service in order to verify if the service becomes operational. The closed state is similar to the state of sleep. This is the primary state of the circuit breaker.

Retry

One of the most often employed patterns in networking is the retry pattern. The system addresses service outages by automatically attempting to resend the request. The method of retrying the request can be classified into three distinct types: fixed retry, retry after exponential backoff, and retry with randomized interval. The fixed retry function will attempt again after a predetermined duration of fixed waiting time. The retry mechanism using exponential backoff will attempt to retry an operation after increasing delays, whereas the randomized retry approach will retry with a randomly determined waiting time.

Figure 2, explains the process of retrying the requests in response to the error.

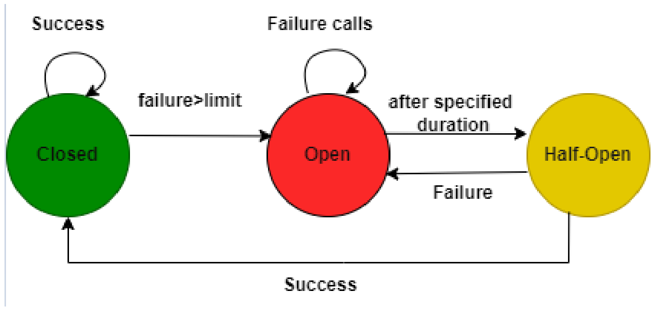

Figure 1: Process flow of circuit breaker pattern

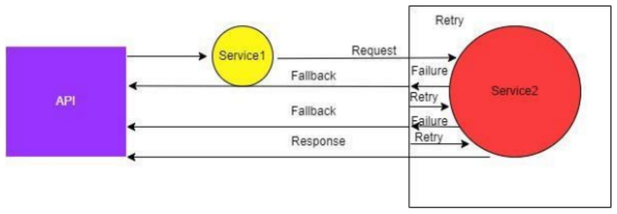

Figure 2: Retry pattern (Punithavathy & Priya,2023)

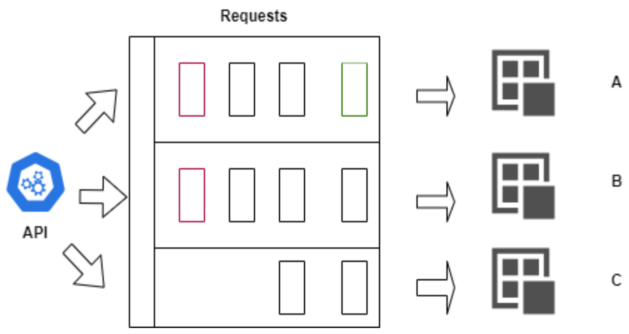

Figure 3: Bulkhead pattern

Bulkhead pattern

The purpose of this pattern is to safeguard the application by preventing failures in one service from impacting other services and triggering a chain reaction of failures. The requests for services are restricted to the relevant service, which prevents excessive requests as the number of requests increases.

Figure 3 illustrates the procedure of the bulkhead pattern. Each service, A, B, and C, is allotted a different pool.

Each service is assigned a certain pool, and as a result, a limit is established for each pool. It permits a restricted quantity of requests to be made to a certain service for processing. By using this approach, a separate pool is created for any sensitive service that is likely to get a higher volume of requests. This prevents overloading from impacting the subsequent service.

Methodology

The order and inventory services are the two services that make up the microservice application. By sending the request to the inventory service, the order service will obtain the product's inventory details. The product's details are contained in the inventory service. Intellij Idea framework and Spring Boot were used in the creation of the services. The maven dependency was added for dependency files. The order service and the inventory service are designed to operate on ports 8081 and 8080, respectively.

Postman, an application programming interface, was utilized to handle this request. The aforementioned application was used to implement this framework. Resilience 4j was used to add the current resilience design patterns. Method-based implementation was used to put these design patterns into practice. These services are created using the Java programming language. Jmeter was used to generate requests at random. This microservice application was developed as a prototype, and its errors were intentionally introduced to explore the pattern's operational model.

Figure 4, explains the process flow of the proposed fault tolerance framework. As shown in the figure, once the request reaches the pool, it checks whether the thread pool is full. The request is routed to the little law bulkhead pattern as soon as the program begins to receive it. The request is sent for processing and the response is returned if there is sufficient space. Requests are sent to sleep, where they wait for a short while if the queue is full. If the requests are still active after the waiting period, they will keep trying to reach the thread pool. Requests are sent for execution if there is sufficient space in the thread pool. If the requests are successfully processed for execution, the response will be returned. Requests that fail are routed

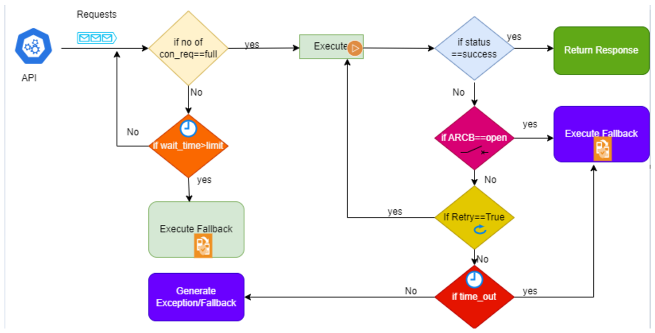

Figure 4: Proposed fault tolerance framework

to the ARCB pattern, which logs the error and tries the request again after the predetermined amount of time. Every incoming request will get a fallback response, even in case of an error. If the requests are blocked from processing by the timeout feature, the appropriate exception or fallback will be utilized.

This framework encompasses the proposed little law bulkhead feature with an auto retry circuit breaker (ARCB) to handle the failure of applications dynamically.

This framework was framed using two stages

- Proposed bulkhead with law
- Proposed bulkhead with law & ARCB pattern.

Stage: 1 Proposed Bulkhead with Law

Existing Bulkhead

The bulkhead is one of the resilience patterns that is used to enhance the fault-tolerant capability of the applications by limiting the request thread from obtaining the services for a long duration (Kapikul *et al.*,2024). This feature is usually deployed in applications and used along with circuit breaker patterns for providing a suitable framework of resilience. These patterns are usually available from libraries such as Hystrix, Resilience4j, Polly, etc. These available patterns are static configurations. The list of parameters required for setting this bulkhead parameters are:

Table 1 displays the set of parameters that are required for implementing this bulkhead pattern. The max concurrent calls display the number of concurrent calls that can be allowed in the service for processing. The max_wait_duration specifies the waiting time of the request if the number of concurrent requests limit is full. These two parameters were part of a semaphore-based bulkhead. The max_threadpool_size specifies the maximum size that the threads can accommodate. The core thread pool

Table 1: Parametric configurations of bulkhead

Parameters	*Description*	*Default values*
max_concurrent_calls	The count of requests that can be performed simultaneously	25
max_wait_duration	Maximum waiting time of a newly arriving request if concurrent calls count is full	0
maxThreadPoolSize	Maximum number of threads available	Available runtime processors
coreThreadPoolSize	Minimum number of threads available	Available runtime processors
queueCapacity	Maximum number of tasks that can wait	100
keepAliveDuration	Wait time of newly arriving threads	20ms
writableStackTrace Enabled	Details of error information	True

size displays the number of threads that are available for processing. The queue capacity specifies the number of tasks that can be held in the queue. The keep alive duration specifies the time limit that the thread can hold on. In case of an error, the writable stack trace would leave an exception indicating the type of error.

The use of bulkheads is gaining popularity as a means to safeguard applications from excessive requests. However, that does not address the issue of the cascade of failures. In order to enhance the application with robust and resilient capabilities, the isolation strategy is integrated with a fault detection technique. The fault isolation approach serves as the outer layer of protection, while the circuit breaker functions as the interior layer.

Little law rule

The little law concept is one of the laws that is most frequently applied in networking systems. In the queueing system, it is frequently utilized. It is said that when a thread is present in the system, it could be actively running or it could be waiting to run. It is stated using the formula:

$$L=\lambda W \quad (1)$$

where,

L=number of concurrent tasks in the system

W=transaction rate

λ=response time (Little & Graves, 2008)

The aforementioned equation can be used to dynamically calculate the maximum number of concurrent requests, hence resolving the issues with statically specified bulkheads. In this case, the average number of transactions can be used to establish the transaction rate. For example, if there are 5000 transactions per hour than the average transaction rate is 16000/ 3600 =1.66 request per second (RPS)

The amount of time it takes for a certain service to respond is known as the response time. Response times are obtained from the application programming interface (API) in this instance. In the end, the concurrent requests can be computed using the formula below:

concurrent_requests= transaction_rate * response_time

Based on the limitation, only limited number of requests are permitted and the remaining requests will be ignored and a fallback mechanism will be implemented. To prevent displaying errors, the fallback approach is employed as a stopgap measure. When this technique is used, a notice such as "out of stock" or "try again after some time" is displayed. As a result, this pattern will process every request, decreasing the error rate and raising the number of requests that are fulfilled.

Stage 2: Proposed Bulkhead using Little Law & Auto Retry Circuit Breaker

Auto retry circuit breaker

Figure 5 illustrates the process diagram of the auto-repeat circuit breaker, which is an adapted form of a static circuit breaker. This modified circuit breaker operates with only two states and is designed to automatically repeat calls based on the request time. The determination of when to retry the calls is determined by the request time attribute (Punithavathy and Priya, 2023).

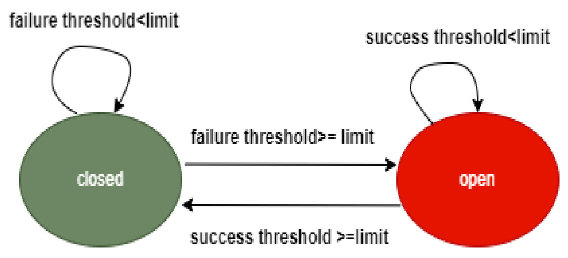

Figure 5: Auto retry circuit breaker (ARCB) pattern (Punithavathy & Priya,2023)

Table 2: Request arrival, allowed and completed

Elapsed time (s)	*Request arrival*	*Request completed*	*Maximum request allowed*
1	10	0	10
5	10	10	20
10	10	20	20
15	10	30	20

An auto retry circuit breaker (ARCB) is implemented, which functions similarly to a circuit breaker. The ARCB is a dynamic pattern that calculates the failure counter based on the response time and success of the request. The state and service idle time is then determined based on the value of the failure counter. Furthermore, the request calls are generated using a retry-based pattern, following the specified waiting duration. In addition to this implementation, other patterns such as timeout and fallback patterns, were incorporated.

Results and Discussion

Upon executing the service and implementing all the previously described patterns, issues arose either within the service itself or in the networking. Efforts were made to escalate the number of queries to the extent of overflowing the system with requests. These services are specifically designed to do tasks, and the length of time it takes for them to respond has been recorded.

The values in Table 2 presents the request arrival rate every five seconds, together with the number of arrived and completed requests, as well as the maximum allowable concurrent requests. It also shows the determination of the maximum concurrent requests, the arrival of more requests, and the remaining requests permitted in the queue. It displays the maximum number of requests that can be processed simultaneously. The maximum number of requests allowed and the finished amount of requests are determined based on the limit.

Table 3: Execution duration of bulkhead, bulkhead using little law and the combination of bulkhead using little law and ARCB pattern

No of requests	*Existing bulkhead (Miraj & Fajar,2022)*			*Bulkhead using little law (ms)*			*Bulkhead using little law and ARCB (ms)*		
	Min (ms)	*Max (ms)*	*Avg (ms)*	*Min (ms)*	*Max (ms)*	*Avg (ms)*	*Min (ms)*	*Max (ms)*	*Avg (ms)*
100	492	2308	1258	564	1076	820	400	724	562
200	1404	4665	2822	794	1258	1026	510	760	635
300	315	7558	5820	456	1992	1224	350	890	621
400	4146	11424	8880	843	1276	1060	568	1054	811
500	491	9084	5658	921	1732	1327	496	616	556

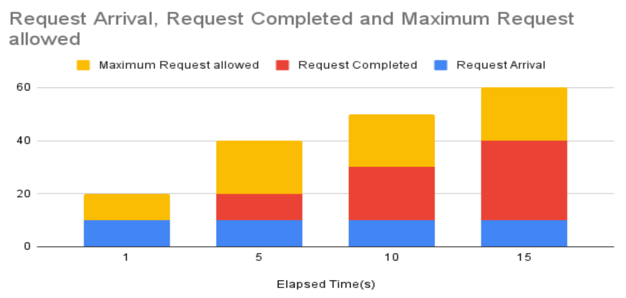

Figure 6: Arrival, processing and completion of requests

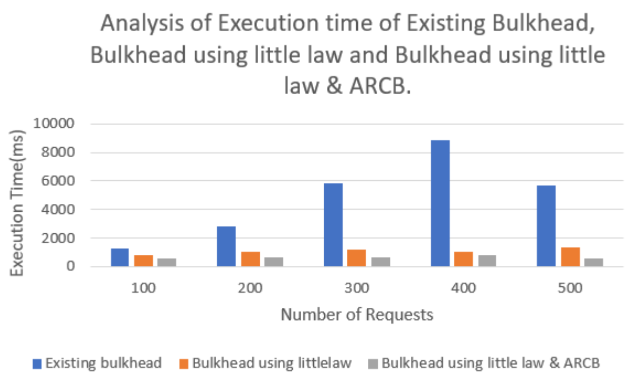

Figure 7: Analysis of execution time of existing bulkhead, bulkhead using little law and bulkhead using little law and ARCB

Figure 6 illustrates the process of request arrival after determining the number of concurrent requests and setting the maximum request limit in the queue. The current status of the remaining permitted requests is updated every five seconds. The graph presents the sample flow of requests.

Table 3 displays the response time of the services while implementing several strategies: existing bulkhead (Miraj and Fajar, 2022), bulkhead using Little's law, and a combination of bulkhead using Little's law with ARCB pattern. The maximum and minimum durations were recorded, and the average duration was calculated. It is observed that when comparing the values of these patterns. The modified version of the bulkhead using Little's law and ARCB obtains the minimized response time of the request. The performance is increased up to 55.3% when compared with that of the existing results and the proposed framework is 2.2 times faster than the existing bulkhead.

Figure 7 presents the analysis of execution time obtained while implementing all the above patterns. It can be observed that the static configuration involves a great amount of time for execution when compared with the dynamically formed patterns.

Conclusion

Distributed systems that prioritize fault tolerance and are hosted in the cloud are popular applications. The microservice design, despite being quite popular, relies on static resilience patterns that have not proven to be effective in addressing all forms of failures. Therefore, the framework created by the dynamic patterns excels in comparison to applications with static setups. The implementation of the suggested framework significantly decreases the time it takes to execute the request and relieves the resources from being overwhelmed by several requests. By alleviating the resource from excessive demands, the services are safeguarded. Therefore, the attainment of availability and throughput is significantly enhanced.

Acknowledgments

None

References

Ali, A., Iqbal, M.M., Jamil, H., Qayyum, F., Jabbar, S., Cheikhrouhou, O., Baz, M., & Jamil, F. (2021). An Efficient Dynamic-Decision Based Task Scheduler for Task Offloading Optimization and Energy Management in Mobile Cloud Computing. *Sensors (Basel, Switzerland), 21*.

Ali, M., Ali, S., & Jilani, A. (2020). Architecture for microservice based system. a report. *A report*.

Frank, S., Hakamian, A., Wagner, L., Kesim, D., Zorn, C., von Kistowski, J., & van Hoorn, A. (2021, September). Interactive elicitation of resilience scenarios based on hazard analysis techniques. In *European Conference on Software Architecture* (pp. 229-253). Cham: Springer International Publishing.

Fu, X., Li, Q., & Li, W. (2023). Modeling and analysis of industrial IoT reliability to cascade failures: An information-service coupling perspective. *Reliability Engineering & System Safety, 239*, 109517.

Hlybovets, A., & Paprotskyi, I. (2024). Increasing the Fault Tolerance in Microservice Architecture. *Cybernetics and Systems*

Analysis, 1-9.

Kapikul, A., Savić, D., Milić, M., & Antović, I. (2024, February). Application Development From Monolithic to Microservice Architecture. In *2024 28th International Conference on Information Technology (IT)* (pp. 1-4). IEEE.

Little, J. D., & Graves, S. C. (2008). Little's law. *Building intuition: insights from basic operations management models and principles*, 81-100.

MIRAJ, M., & FAJAR, A. N. (2022). MODELBASED RESILIENCE PATTERN ANALYSIS FOR FAULT TOLERANCE IN REACTIVE MICROSERVICE. *Journal of Theoretical and Applied Information Technology*, *100*(9).

Punithavathy, E., & Priya, N. (2024). Auto retry circuit breaker for enhanced performance in microservice applications. *International Journal of Electrical & Computer Engineering (2088-8708)*, *14*(2).

Rahman, M. I. (2022). *Analysis of Microservice Coupling Measures* (Master's thesis).

SA, N. S., & Sebastian, S. (2023, December). Circuit Breaker: A Resilience Mechanism for Cloud Native Architecture. In *2023 Global Conference on Information Technologies and Communications (GCITC)* (pp. 1-8). IEEE.