International Journal of Electrical and Computer Engineering (IJECE)
Vol. 14, No. 2, April 2024, pp. 2274~2281
ISSN: 2088-8708, DOI: 10.11591/ijece.v14i2.pp2274-2281

Auto retry circuit breaker for enhanced performance in microservice applications

E. Punithavathy[1], N. Priya[2]
[1]Department of Computer Applications, Madras Christian College, Chennai, India
[2]Research Department of Computer Science, Shrimathi Devkunvar Nanalal Bhatt Vaishnav College for Women, Chennai, India

Article Info

Article history:

Received Jun 23, 2023
Revised Nov 5, 2023
Accepted Nov 29, 2023

Keywords:

Auto-retry circuit breaker
Resilience
Retry
Timeout
Resilience

ABSTRACT

Cloud-hosted distributed systems are much more resilient to failure. Resilience is the key factor in avoiding cascading failures in microservice architectures, where circuit-breaker, retry, bulkhead, and rate-limiter patterns are implemented. Through the implementation of these patterns, availability and reliability factors can be greatly improved. Circuit-breaker emphasizes a fail-fast mechanism, so it is highly recommended. Although they are beneficial, some factors, such as configuration, need to be improved to increase the system's throughput. In this paper, an approach called auto retry circuit breaker (ARCB) is implemented, which is a modified version of existing patterns of circuit breaker, and it has been found that the throughput of the system can be increased by 36% when compared to the existing manual circuit breaker pattern.

Corresponding Author:

E. Punithavathy
Department of Computer Applications, Madras Christian College
Chennai, India
Email: punithavathy@mcc.edu.in

1. INTRODUCTION

Cloud computing has become popular due to its vast performance in distributed systems. The most prominent factors of distributed systems are flexibility in expanding the operations, capability in withstanding the failures, and its distributed content delivery both for a nominal user and for an organization [1]. In order to satisfy the demands of cloud based distributed systems, there formed an architectural pattern called microservice, which is specialized for its decentralized nature [2]–[4]. Although it is quite useful in developing suitable applications, they need to be concentrated towards its resiliency nature [5].

One of the most demanding factors for applications to survive in the field of cloud is their resilience nature. Fault or failure might lead an application into an unreliable state, further evolving as cascading failure [6]. Failures are usually classified into two categories; they are transient faults and non-transient faults. The faults that could recover by themselves can be called transient faults, whereas faults that require external help to recover can be called non-transient faults. It is stated that cloud-based environments usually suffer from transient faults. All the cloud application designs should focus on these fault-tolerant features, which could avoid cascading outages. The end users might not be sure whether they are connected to the system all the time due to the connection problem that exists in the network [7]. Whenever there is a problem with the network or the services, it is notified by the error messages such as status 500-internal server error, status 503-service unavailable and status 504-gateway timeout errors [8]. The current developing trends of any application include this fault-tolerant model, stating that even though the application is successful it must be tolerant towards its failure cases. This feature decides the survival of the application in many cases [9].

To survive the failures of applications, which are hosted in cloud-distributed systems, resilience patterns were used. The term resilience can be defined as the ability of the system to regain its original condition after an unpredicted degradation [10]. Resilience characteristic is achieved in a microservice architecture by using the patterns such as circuit-breaker, retry, bulkhead, and rate limiter patterns [11]–[13]. Out of all the patterns, the circuit breaker is the most often used technique. It is a type of recovery technique, which is used at the time of repeated failure to avoid cascading of failures [14]. The essential components of circuit breakers are their parameters (timeout, number of failures, failure threshold, and number of calls), states (closed, half-open and open), and the most important error handling method or alternate actions [15]–[17]. As shown in Figure 1, this pattern is similar to the circuit breaker switch pattern used in electrical applications.

The second popular resiliency pattern is retry. This pattern does not include states such as circuit breakers. This pattern will exponentially or periodically bombard the application programming interface (API) with, request calls to the preferred service to get to know about the working of those services. The calls will be sent based on the number of retries specified in the configuration of the service [18]. As shown in Figure 2, this pattern is used for retrying the requests, based on the number of retries the user specifies which might lead to overhead issues, if not configured properly.

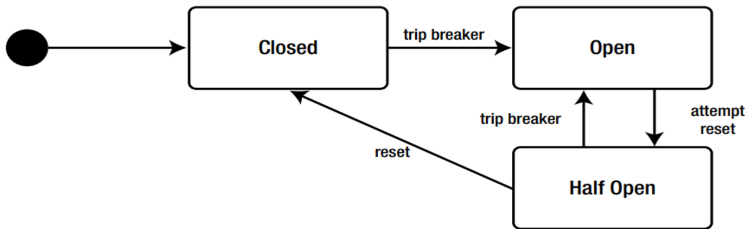


Figure 1. Circuit breaker diagram [18]

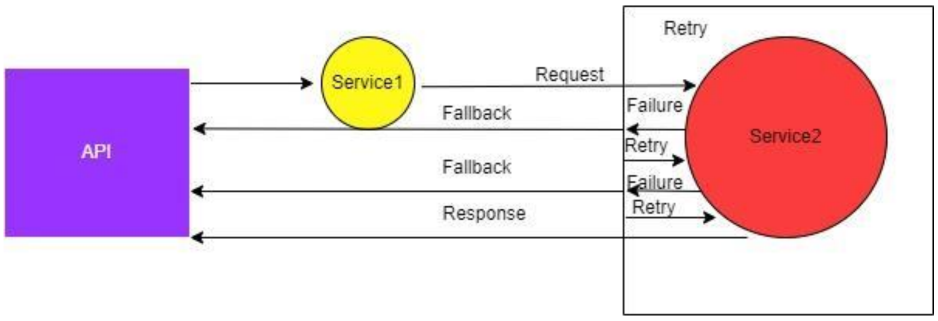


Figure 2. State diagram of retry pattern

Even though these patterns are widely used, all these patterns are implemented with manual configurations. The developer without knowing the nature of transient failure conditions, a random value was configured in circuit breaker and retry pattern properties. This causes the problem of accelerated breaking and recoverability issues while using the wrong values in the waiting or open state of the service, which leads to poor performance of the service. The concept of circuit breaker is to isolate the service for a period of time to recover whenever it is down. In other cases, if services are working fine and the problem exists with the network, using the manual circuit breaker might be a bad idea because it makes the working service idle for a long period of time [19]. The retry pattern is one of the most used patterns in distributed cloud applications, for repeating tries [20], [21]. But the retry pattern with repeated retries also fails to solve the problem of resiliency and give a complete solution. The main objective of resiliency is to assure fault tolerance by increasing the availability and redeemability towards its failure state, during its execution. But the existing patterns with misconfigured values, fail to recognize the nature of failures.

To overcome this problem, a novel approach called auto retry circuit breaker (ARCB) pattern is implemented. This pattern incorporates the states used in circuit breaker with a modification in addition to the timeout strategy. Here, the ARCB pattern incorporates an auto approach in spite of manual physical configuration and retries were done to check the functionality of the services. The objective of this pattern is to reduce the issues related to the throughput of the service, and the availability demand in the existing scenario.

2. PROPOSED ALGORITHM

In the proposed algorithm, circuit breakers are automatically switched between states. The properties of existing circuit breakers are used and along with that time taken for request is calculated. The manual configurations are modified with respect to the time-taken strategy. The parameters used in the proposed algorithm are *failure_count* to count the number of failures, States which consists of two states such as closed and open, an initial value is set as closed, and limit, which is used to check the threshold value of failures. This proposed model comes with two phases. The first phase is done if the limit value does not exceed the failure limit. A closed-state algorithm is the first phase here.

Algorithm 1. ARCB-closed state

```
Step 1: Set $failure\_count = 0$ (count number of failures)
Step 2: Set $State = closed$ (indicates whether it is closed or open)
Step 3: Set $limit$ as $avg\_limit$ (maximum failures allowed during closed state)
Step 4: if $response\_status == success$ then return $result\_of\_response$
            else if $response\_status == failure$ then $failure\_count = failure\_count + 1$ and
Step 5: execute fallback method
Step 6: if $failure\_count >= threshold\_limit$ then
          a) Set $State = open$
          b) Call ARCB-open
Step 7: For each client requests that is received during the open state
        execute fallback method
```

In the second phase of the algorithm, the State value is changed from closed to open, it uses an auto-retry concept based on request time taken. Based on the obtained response time, the waiting time is calculated and the next request is retried by the system. The main role of this proposed method is creating lightweight requests by the system and sending them to the service, to know the status of the circuit breaker.

Algorithm 2. ARCB-open

```
Step 1: if $State == open$, then find $request\_time$ of the last request.
Step 2: if $request\_time >= average\_limit$ then generate auto request after ($request\_time/2$)
        else if ($request\_time <= average\_limit$) then generate auto request after $\left(\frac{request_{time}}{2}\right) + 1\,ms$
Step 3: if $response\_status == success$ then
          a) $success\_count = success\_count + 1$
          b) return $result\_of\_response$
Step 4: if $success\_count$ is greater than $success\_limit$ then $state = closed$
```

Once the required service requests yield a successful response, the state value changes to closed. The algorithm for the closed state will perform its processing and reset all parameters to their initial values. Similarly, if the service yields an unsuccessful response or an error response, the state remains in an open state, and the auto-retry process will continue until it receives a successful response.

3. METHOD

This ARCB pattern was implemented in a microservice application, which includes two services such as product service and variant service. This microservice application was built using active server pages or ASP.Net core using C#. This is a simple microservice application, which gets a request from a client, querying the product details. Figure 3 explains the overall architecture of this microservice application.

The request is forwarded to the product service through the API, and consecutively, the request is forwarded to the variant service from the product service to get the external details of the product, such as description, and price. The product service contains information such as product ID and name. Whereas the variant service consists of external information about the product such as its product ID, price, description, and expiration. The product service sends the request to the variant service with the input parameter product ID. If the product ID of the product service and the product ID of the variant service gets matched, the corresponding information is posted back to the client as a response. The libraries used to implement the circuit breaker are Polly libraries. Resilience patterns are implemented using these libraries [22]. The applications are hosted and deployed on Azure websites. Using Postman, API requests are generated and sent to the services. Postman specializes in creating hypertext transfer protocol (HTTP) requests and provides an option to save the request and response [23]. To control the path of data, a single gateway is used for all services [24]. The API involves a request that uses parameters as a querying entity to get some results from the service [25]. Both the existing circuit breakers and ARCB are implemented, and their performance is observed with the execution of these services. When compared with the existing pattern, the ARCB pattern duration is much faster than that of the circuit breaker pattern.

In this proposed concept of ARCB, the three states of the circuit breaker are reduced to two states. They are the combination of circuit breaker and retry patterns, where the retry operation with a modified algorithm is implemented in an open state. Here no manual parameters are required instead the requests are sent automatically based on the analyzed time duration of failed requests. Normal circuit breakers use the concepts of timeout and parameters set; the circuit breaker will start to function. The timeouts were calculated for both success and failure calls. Any request, that takes a long duration will eventually be considered to be a failure. So, we measure the time taken for each request. An average time is set for each request, if the given request takes greater than the average time then it is considered to be a failure. Figure 4 explains the behavior of the ARCB pattern, which is modified from the existing circuit breaker pattern.

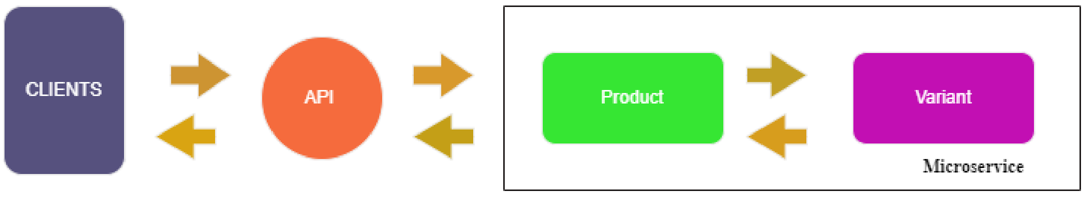


Figure 3. Microservice application architecture

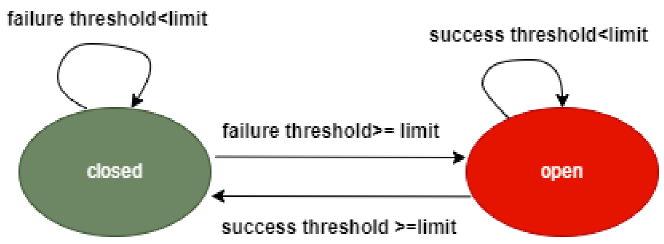


Figure 4. State diagram of ARCB

3.1. Closed to open state

Initially, ARCB will begin in a closed state. If all API request calls receive a successful response, the state will eventually be closed, as shown in Figure 5. If there is a problem with the duration of the response or failure response, the failure threshold will be increased. Once the failure threshold reaches its limits, the state is shifted to an Open state. In the Open state, the ARCB actual functioning is done by generating auto retries.

Figure 6 shows the response received for a given request during the working of a closed state in the ARCB pattern. As per the given request, the details regarding the product are retrieved from the variant service. Based on the successful response obtained from variant service, the state remains closed.

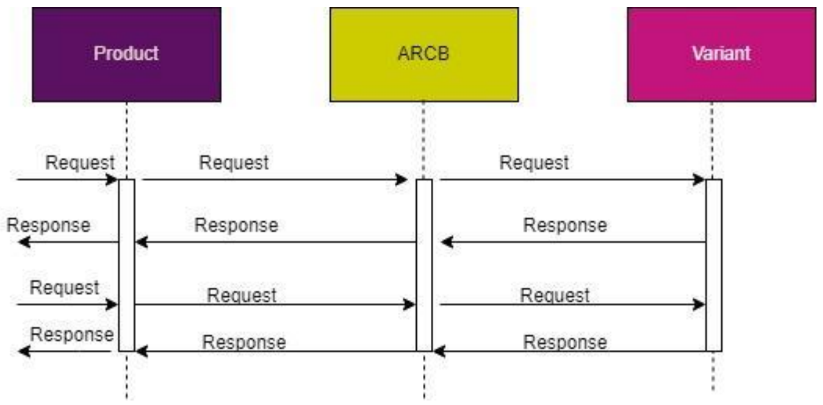
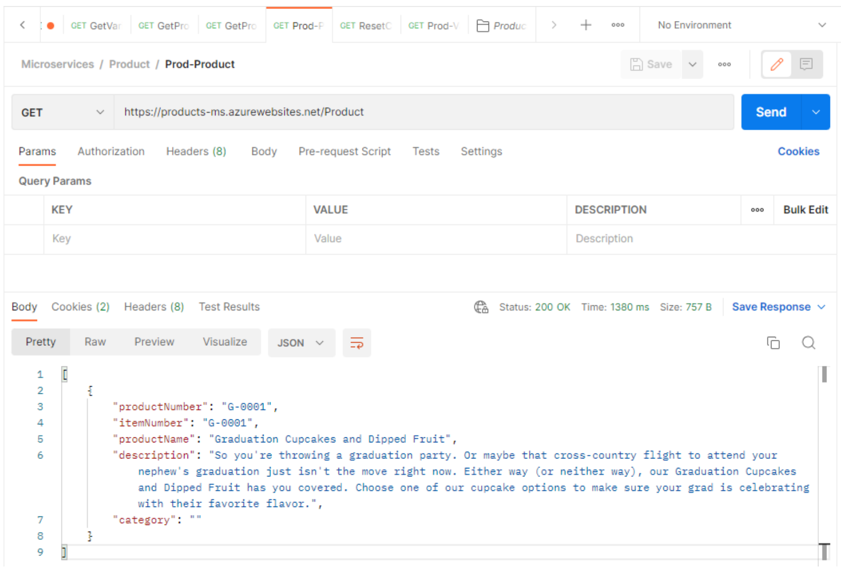


Figure 5. Sequential diagram of the closed state


GET GetVar | GET GetPro | GET GetPro | GET Prod-P | GET ResetC | GET Prod-V | Produc — No Environment

Microservices / Product / **Prod-Product** — Save

GET https://products-ms.azurewebsites.net/Product — Send

Params | Authorization | Headers (8) | Body | Pre-request Script | Tests | Settings — Cookies

Query Params

	KEY	VALUE	DESCRIPTION	Bulk Edit
	Key	Value	Description	

Body | Cookies (2) | Headers (8) | Test Results — Status: 200 OK Time: 1380 ms Size: 757 B — Save Response

Pretty | Raw | Preview | Visualize | JSON

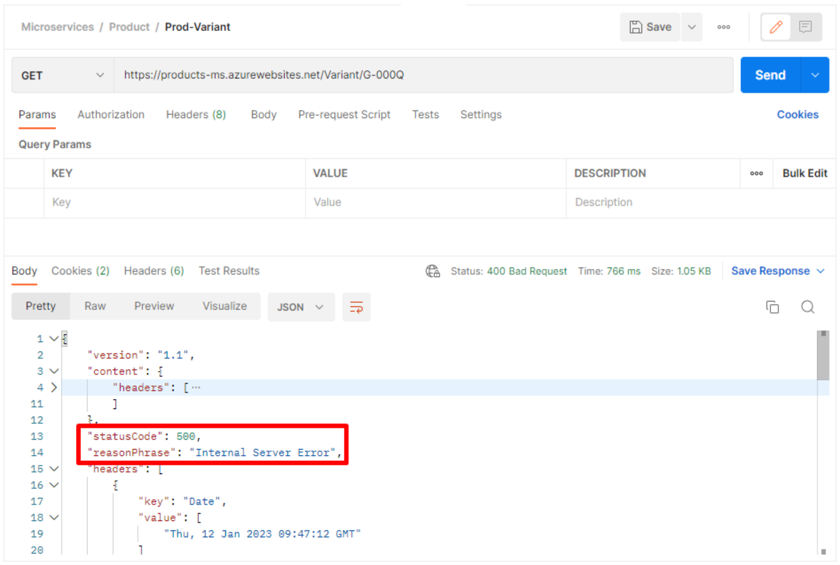
```
[
    {
        "productNumber": "G-0001",
        "itemNumber": "G-0001",
        "productName": "Graduation Cupcakes and Dipped Fruit",
        "description": "So you're throwing a graduation party. Or maybe that cross-country flight to attend your
            nephew's graduation just isn't the move right now. Either way (or neither way), our Graduation Cupcakes
            and Dipped Fruit has you covered. Choose one of our cupcake options to make sure your grad is celebrating
            with their favorite flavor.",
        "category": ""
    }
]
```

Figure 6. Screenshot of the response received in a closed state

3.2. Open state

During the open state, the API request from the client will not be forwarded to the required service. Instead, an error handling method or a fallback method is returned as a response to the client. Meanwhile, the ARCB will send auto requests to the required service, on the basis of the last request duration. The ARCB will check for the response, if they reach the average time or failure response again, the request is again retried after the specified number of seconds. In order to avoid the overloading of requests, the duration is measured. If the duration is greater than the average time then the next auto request will be forwarded at half the duration. If the duration of a request is less than the average time, but the response is a failure then the next auto request is forwarded at half plus the duration. While the state is open, the user's requests are ignored and in response to that, a fallback method will be executed for each request. Whereas, lightweight request calls are generated by the system and sent to the service.

Figure 7 shows an error message from the screenshot, which is normally used at times of failure conditions when a circuit breaker is not used. It also noted that the state value remains closed, so the ARCB has not started its action, since the *failure_counter* has not reached its limits. If the *failure_counter* value exceeds the limit, state value is changed to open and ARCB comes in to action.


Microservices / Product / **Prod-Variant** — Save

GET https://products-ms.azurewebsites.net/Variant/G-000Q — Send

Params | Authorization | Headers (8) | Body | Pre-request Script | Tests | Settings — Cookies

Query Params

	KEY	VALUE	DESCRIPTION	Bulk Edit
	Key	Value	Description	

Body | Cookies (2) | Headers (6) | Test Results — Status: 400 Bad Request Time: 766 ms Size: 1.05 KB — Save Response

Pretty | Raw | Preview | Visualize | JSON

```
{
    "version": "1.1",
    "content": {
        "headers": [⋯
        ]
    },
    "statusCode": 500,
    "reasonPhrase": "Internal Server Error",
    "headers": [
        {
            "key": "Date",
            "value": [
                "Thu, 12 Jan 2023 09:47:12 GMT"
            ]
```

Figure 7. Error screenshot using postman API

3.3. Open to closed

Once the auto requests are sent to the service, it returns either a successful response or a failed response. If the response status is a success, then the success counter is incremented. If the *success_Counter* values are greater than the prescribed limit, the state of the ARCB is changed to closed. When the ARCB is in a closed state, it behaves like an existing circuit breaker until the response time is within the specified time. During this state all the counter values are reset to zero.

Till they get a successful response, the state will function in open mode, if they get a successful response then the success threshold will get increased. If they reach their success limits, then eventually they are moved to the closed state. And the values of the parameter will be reset to the initial value. The diagram from Figure 8, explains the processing of requests in the open state of ARCB. Similarly, Figure 9 presents a screenshot of the working state of the ARCB pattern in the closed state.

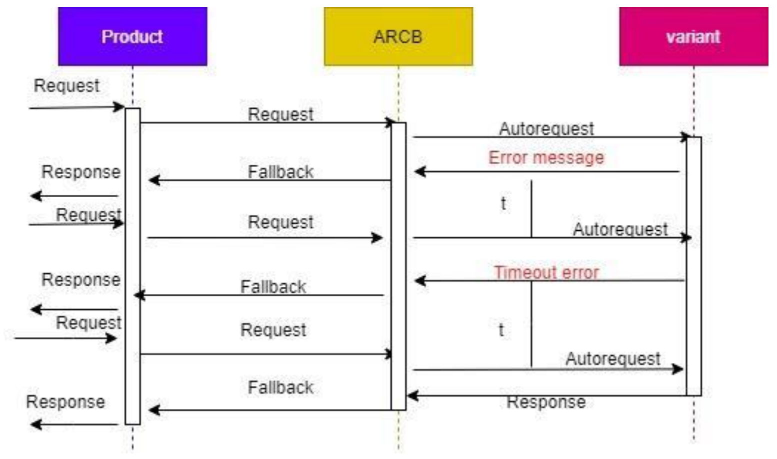
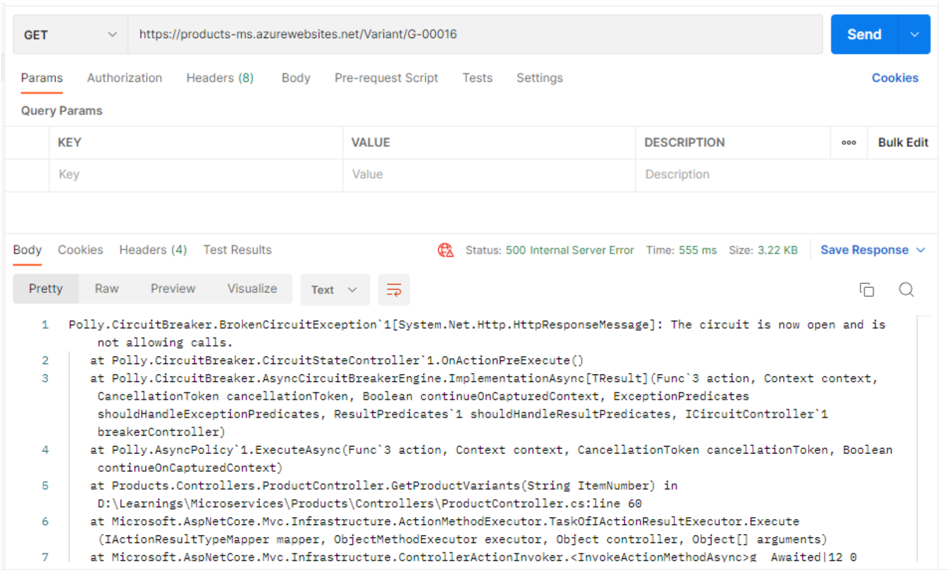


Figure 8. Sequential diagram of an open state in ARCB



```
Polly.CircuitBreaker.BrokenCircuitException`1[System.Net.Http.HttpResponseMessage]: The circuit is now open and is
    not allowing calls.
   at Polly.CircuitBreaker.CircuitStateController`1.OnActionPreExecute()
   at Polly.CircuitBreaker.AsyncCircuitBreakerEngine.ImplementationAsync[TResult](Func`3 action, Context context,
    CancellationToken cancellationToken, Boolean continueOnCapturedContext, ExceptionPredicates
    shouldHandleExceptionPredicates, ResultPredicates`1 shouldHandleResultPredicates, ICircuitController`1
    breakerController)
   at Polly.AsyncPolicy`1.ExecuteAsync(Func`3 action, Context context, CancellationToken cancellationToken, Boolean
    continueOnCapturedContext)
   at Products.Controllers.ProductController.GetProductVariants(String ItemNumber) in
    D:\Learnings\Microservices\Products\Controllers\ProductController.cs:line 60
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.TaskOfIActionResultExecutor.Execute
    (IActionResultTypeMapper mapper, ObjectMethodExecutor executor, Object controller, Object[] arguments)
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.<InvokeActionMethodAsync>g  Awaited|12 0
```

Figure 9. Working of open state in ARCB

4. RESULTS AND DISCUSSION

The existing patterns of circuit breakers and the ARCB pattern were implemented in a microservice project, and the results were analyzed. When using circuit breaker patterns in the application, all the parameters are assigned initial values, and the response time is recorded during their failure states. Similarly, the ARCB was implemented without any manual configurations, and the responses were recorded.

In Table 1, an average of ten requests duration and their timings were noted in their failure states. The timing value proves that while using ARCB the duration of execution can be highly reduced due to this automation. The average value of the ARCB timings is 36% less when compared with the time duration obtained from manual service side circuit breaker. Similarly, Figure 10 represents an analysis of both circuit breaker and ARCB, which proves that execution duration of ARCB is faster when compared with the manual configuration of circuit breaker.

Table 1. Average processing duration in circuit breaker and auto-retry circuit breaker

Requests	Manual CB average timings in ms	ARCB average timings in ms
1-10	172	55
10-20	64	50
20-30	13	13
30-40	15	10
40-50	38	18
50-60	84	39
60-70	67	35
70-80	29	11
80-90	77	40
90-100	12	11

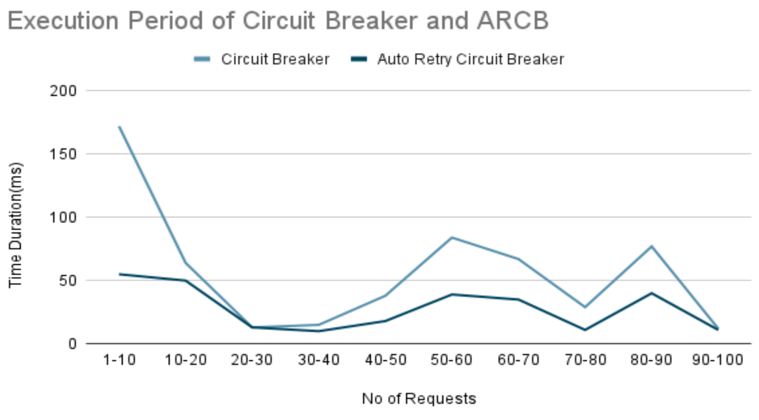


Figure 10. Analysis of ARCB and circuit breaker time duration in ms

5. CONCLUSION

This paper proposes a new algorithm for improvising the fail-fast mechanism in existing circuit breaker. It also proposes the changes of manual parameters to auto generate requests in an open state. The main goal of this work is to reduce the concept of manual parameter values in the property section. Analysis is done by implementing ARCB and existing circuit breaker patterns in a microservice application. Results proved the efficiency of ARCB is 36% faster, when compared with the existing circuit breaker. In the future, the ARCB is planned to be expanded for both service mesh concept with side car policy, which is widely used in complex applications.

REFERENCES

[1] S. Shetty, C. Kamhoua, L. Njilla, "Front matter," in *Blockchain for Distributed Systems Security*, Wiley, 2019.

[2] G. Blinowski, A. Ojdowska, and A. Przybylek, "Monolithic vs. microservice architecture: a performance and scalability evaluation," *IEEE Access*, vol. 10, pp. 20357–20374, 2022, doi: 10.1109/ACCESS.2022.3152803.

[3] M. Baboi, A. Iftene, and D. Gîfu, "Dynamic microservices to create scalable and fault tolerance architecture," *Procedia Computer Science*, vol. 159, pp. 1035–1044, 2019, doi: 10.1016/j.procs.2019.09.271.

[4] M. Villamizar *et al.*, "Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures," in *Proceedings - 2016 16th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing, CCGrid 2016*, May 2016, pp. 179–182, doi: 10.1109/CCGrid.2016.37.

[5] D. Richter, M. Konrad, K. Utecht, and A. Polze, "Highly-available applications on unreliable infrastructure: microservice architectures in practice," in *2017 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, Jul. 2017, pp. 130–137, doi: 10.1109/QRS-C.2017.28.

[6] L. J. Jagadeesan and V. B. Mendiratta, "When failure is (not) an option: reliability models for microservices architectures," in *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2020, pp. 19–24, doi: 10.1109/ISSREW51248.2020.00031.

[7] Z. Li, Q. Lu, L. Zhu, X. Xu, Y. Liu, and W. Zhang, "An empirical study of cloud API issues," *IEEE Cloud Computing*, vol. 5, no. 2, pp. 58–72, Mar. 2018, doi: 10.1109/MCC.2018.022171668.

[8] M. I. P. Salas and E. Martins, "Security testing methodology for vulnerabilities detection of XSS in web services and WS-security," *Electronic Notes in Theoretical Computer Science*, vol. 302, pp. 133–154, Feb. 2014, doi: 10.1016/j.entcs.2014.01.024.

[9] A. van Hoorn, A. Aleti, T. F. Dullmann, and T. Pitakrat, "ORCAS: efficient resilience benchmarking of microservice architectures," in *2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, Oct. 2018, pp. 146–147, doi: 10.1109/ISSREW.2018.00-10.

[10] K. Yin and Q. Du, "On representing resilience requirements of microservice architecture systems," *International Journal of Software Engineering and Knowledge Engineering*, vol. 31, no. 6, pp. 863–888, Jun. 2021, doi: 10.1142/S0218194021500261.

[11] M. R. S. Sedghpour and P. Townend, "Service mesh and eBPF-powered microservices: a survey and future directions," in *2022 IEEE International Conference on Service-Oriented System Engineering (SOSE)*, Aug. 2022, pp. 176–184, doi: 10.1109/SOSE55356.2022.00027.

[12] C. Santana, L. Andrade, F. C. Delicato, and C. Prazeres, "Increasing the availability of IoT applications with reactive microservices," *Service Oriented Computing and Applications*, vol. 15, no. 2, pp. 109–126, 2021, doi: 10.1007/s11761-020-00308-8.

[13] A. Palliwar and S. Pinisetty, "Using gossip enabled distributed circuit breaking for improving resiliency of distributed systems," in *2022 IEEE 19th International Conference on Software Architecture (ICSA)*, 2022, pp. 13–23, doi: 10.1109/ICSA53651.2022.00010.

[14] H. H. Addeen, "A dynamic fault tolerance model for microservices architecture," Electronic Theses and Dissertations, South Dakota State University, 2019.

[15] Falahah, K. Surendro, and W. D. Sunindyo, "Circuit breaker in microservices: state of the art and future prospects," *IOP Conference Series: Materials Science and Engineering*, vol. 1077, no. 1, Feb. 2021, doi: 10.1088/1757-899X/1077/1/012065.

[16] S. Vergara, L. Gonzalez, and R. Ruggia, "Towards formalizing microservices architectural patterns with event-B," in *Proceedings - 2020 IEEE International Conference on Software Architecture Companion, ICSA-C 2020*, Mar. 2020, pp. 71–74, doi: 10.1109/ICSA-C50368.2020.00022.

[17] J. Carnell and I. H. Sánchez, *Spring microservices in action*, 2nd ed. 2017.

[18] F. Gutierrez, "Microservices," in *Spring Boot Messaging*, Apress, 2017, pp. 179–192.

[19] S. Beck, "Simulation-based evaluation of resilience antipatterns in microservice architectures," University of Stuttgart, 2018.

[20] M. R. Saleh Sedghpour, C. Klein, and J. Tordsson, "An empirical study of service mesh traffic management policies for microservices," in *Proceedings of the 2022 ACM/SPEC on International Conference on Performance Engineering*, Apr. 2022, pp. 17–27, doi: 10.1145/3489525.3511686.

[21] S. Frank, L. Wagner, A. Hakamian, M. Straesser, and A. van Hoorn, "MiSim: a simulator for resilience assessment of microservice-based architectures," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2022, pp. 1014–1025, doi: 10.1109/QRS57517.2022.00105.

[22] Christian Horsdal Gammelgaard, "Building applications," in *Microservices in .NET Core: with examples in Nancy*, Simon and Schuster, 2017, pp. 263–285.

[23] M. Shershneu and A. Oskin, "Postman platform for API development in the mobile application 'musicians of Russia'," in *Materials of XII Junior Researchers' Conference*, 2020, pp. 128–131.

[24] H. F. Martinez, O. H. Mondragon, H. A. Rubio, and J. Marquez, "Computational and communication infrastructure challenges for resilient cloud services," *Computers*, vol. 11, no. 8, Jul. 2022, doi: 10.3390/computers11080118.

[25] K. CEBECİ and Ö. KORÇAK, "Design of an enterprise level architecture based on microservices," *Bilişim Teknolojileri Dergisi*, vol. 13, no. 4, pp. 357–371, Oct. 2020, doi: 10.17671/gazibtd.558392.

BIOGRAPHIES OF AUTHORS

E. Punithavathy received B.Sc. degrees in computer science in the year 2009 and MCA degree from Anna University, in 2012 respectively. She has presented papers in international conferences. She has been an assistant professor in Madras Christian College, Chennai since 2013. Her research interests include cloud computing, software engineering. She can be contacted email: Punithavathy@mcc.edu.in.

N. Priya a technically skilled academician having more than 15 years of teaching experience. She completed both her UG and PG degrees with distinction. She was awarded Ph.D from Bharathiar University Coimbatore. She has been serving as the head of UG Department of Computer Application from 2004 to 2019 in SDNB Vaishnav College for Women, Chennai. She has also been a member of the Academic Audit and review panel in other institutions. She has chaired several sessions in International Conferences. She has published and presented more than 20 research articles and completed many NPTEL courses. She is a recognized research supervisor under the University of Madras. Her research areas include data mining, image processing, neural networks, network programming and fuzzy logic. Currently she is working as associate professor in the PG Department of Computer Science, SDNB Vaishnav College for Women, Chennai. She can be contacted at email: drnpriya2015@gmail.com.