UDC 621.391 **doi: 10.20998/2522-9052.2018.4.13**

H. Molchanov, A. Zhmaiev

National Technical University "Kharkiv Polytechnic Institute", Kharkiv, Ukraine

CIRCUIT BREAKER IN SYSTEMS BASED ON MICROSERVICES ARCHITECTURE

The subject of the article is a review of Circuit Breaker pattern in systems based on microservices architecture. **The purpose** of the article is to analyze the advantages and disadvantages of Circuit Breaker for microservices. **Results.** The precise way that the circuit opening and closing occurs is as follows: assuming the volume across a circuit meets a certain threshold; and if the error percentage exceeds the threshold error percentage; then the circuit-breaker transitions from closed to open; while it is open, it short-circuits all requests made against that circuit-breaker. After some amount of time, the next single request is let through (this is the half-open state). If the request fails, the circuit-breaker returns to the open state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to closed and the logic in 1 takes over again. **Conclusions.** Circuit Breaker has been reviewed and explained. This pattern is emerging as essential for the reliability, ease of access, and flexibility of MSAs. Since microservices is in its early development, we can expect more patterns like this to appear in the future. It is interesting that it is structural, in the sense that they do not change the operations that services offer. Being of this nature, their implementations benefit from parametricity to achieve reusability. However, their adoption also makes MSAs more complicated, and they influence the communication structures that will be enacted in a system. This suggests that methods for the programming and verification of communications among services should keep patterns such as these into account.

Keywords: microservices; Circuit Breaker; Hystrix, software; hardware.

Introduction

Systems based on microservices architecture (MSA) are becoming more and more popular in modern IT environments. Integration of different components is an integral part of any system. Almost all systems, which perform anything useful for a given business, need to be integrated with one or more third party component. But this integration also presents huge challenges with respect to the performance of the overall integrated system. With MSA, where a number of services are broken down based on the services or functionality these microservices offer, count of touch points increase. While connecting to other microservices (within the same bounded context or of some remote, external system), a lot of things can go wrong. Microservices being connected to may be slow or down. If our system is not designed to handle this scenario gracefully, it can have an adverse impact on the performance and stability.

Base material

Even the most reliable services will eventually fail, if it given with enough number of incoming requests. What makes it even more complicated is that, in MSA [6], a failing service probably has other services that depend on it. If we do not properly manage this failure event, we have a risk of a cascading failure.

The Circuit Breaker pattern [1-5] is aimed at preventing the failure of a single component to cascade beyond its boundaries, and thereby bring the entire system down with it. So, when a service becomes unresponsive, its invokers should stop waiting for it and start dealing with the fact that the failing service may be unavailable. As a result, Circuit Breakers contribute to the stability and resilience of both clients and services: clients limit their waste of resources on trying to access unresponsive services, and overloaded services are given a chance to recover by finishing some of the tasks they are currently processing. Circuit Breaker works by wrapping calls towards a target service and monitoring their failure rates. The idea is that when the target service becomes too slow or replies too often with faults, the Circuit Breaker will trip and future invocations from the client will immediately return a fault. More specifically, the pattern can be implemented as a finite-state machine, depicted in Fig. 1.

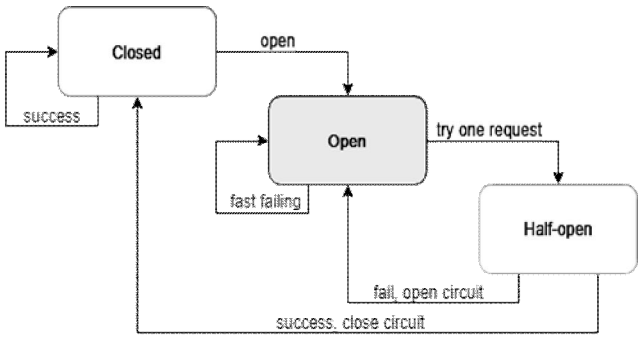

Fig. 1. Circuit Breaker State Diagram

We describe these states as the following.

Closed: Requests are passed to the target service. Faults caused by the requested operation such as exceptions or timeouts increase the Circuit Breaker's respective failure and timeout counters. When these counters exceed a specified threshold, or when another predefined criterion is met, the breaker is tripped and transitions into the open state.

Open: Requests are not passed to the target service. Instead, a failure message is immediately given to the client as reply. Potential fallback mechanisms can be called to handle the failure. The Circuit Breaker can transition to the half-open state, either by periodically pinging the service to check for when it becomes responsive again, or after a specified amount of time.

Half-Open: While in this state, a limited number of requests are allowed through to the service. If the target

service sends back successful replies, the Circuit Breaker is reset back into the closed state as well as its failure and timeout counters. However, if any of the requests fail while in the half-open state, the Circuit Breaker transitions back into the open state. The state transitions for Circuit Breakers are generally controlled by a set of parameters, which typically includes those described in Table 1.

Table 1 – **Circuit Breaker Parameters**

Parameter	Description
callTimeOut	timeout the client request after N seconds without a response from the server
rollingWindow	monitor errors over a rolling window of N seconds
tripTreshold	open the circuit if the error rate gets ≥ N%
resetTimeOut	attempt to reset the circuit after N seconds of opening the circuit

One of the most famous implementations of Circuit Breakers is provided by the Hystrix [2]. It is a library that helps you control the interactions between distributed services by adding latency tolerance and fault tolerance logic. Hystrix does this by isolating points of access between the services, stopping cascading failures across them, and providing fallback options, all of which improve your system's overall resiliency. Hystrix evolved out of resilience [11] engineering work that the Netflix API team began in 2011. In 2012, Hystrix continued to evolve and mature, and many teams within Netflix adopted it. Today tens of billions of thread-isolated, and hundreds of billions of semaphore-isolated calls are executed via Hystrix every day at Netflix. This has resulted in a dramatic improvement in uptime and resilience.

The Fig. 2 shows what happens when you make a request to a service dependency by means of Hystrix.

The first step is to construct a command object to represent the request you are making to the dependency. Pass the constructor any arguments that will be needed when the request is made.

There are four ways you can execute the command, by using one of the following four methods of your Hystrix command object (the first two are only applicable to simple Hystrix Command objects and are not available for the HystrixObservableCommand):

- execute() – blocks, then returns the single response received from the dependency (or throws an exception in case of an error):
- queue() – returns a Future with which you can obtain the single response from the dependency
- observe() – subscribes to the Observable that represents the response(s) from the dependency and returns an Observable that replicates that source Observable
- toObservable() – returns an Observable that, when you subscribe to it, will execute the Hystrix command and emit its responses

If request caching is enabled for this command, and if the response to the request is available in the cache, this cached response will be immediately returned in the form of an Observable. When you execute the command [12], Hystrix checks with the circuit-breaker to see if the circuit is open.

If the circuit is open (or "tripped") then Hystrix will not execute the command but will route the flow to (8) Get the Fallback.

If the circuit is closed, then the flow proceeds to (5) to check if there is capacity available to run the command.

If the thread-pool and queue (or semaphore, if not running in a thread) that are associated with the command are full then Hystrix will not execute the com-

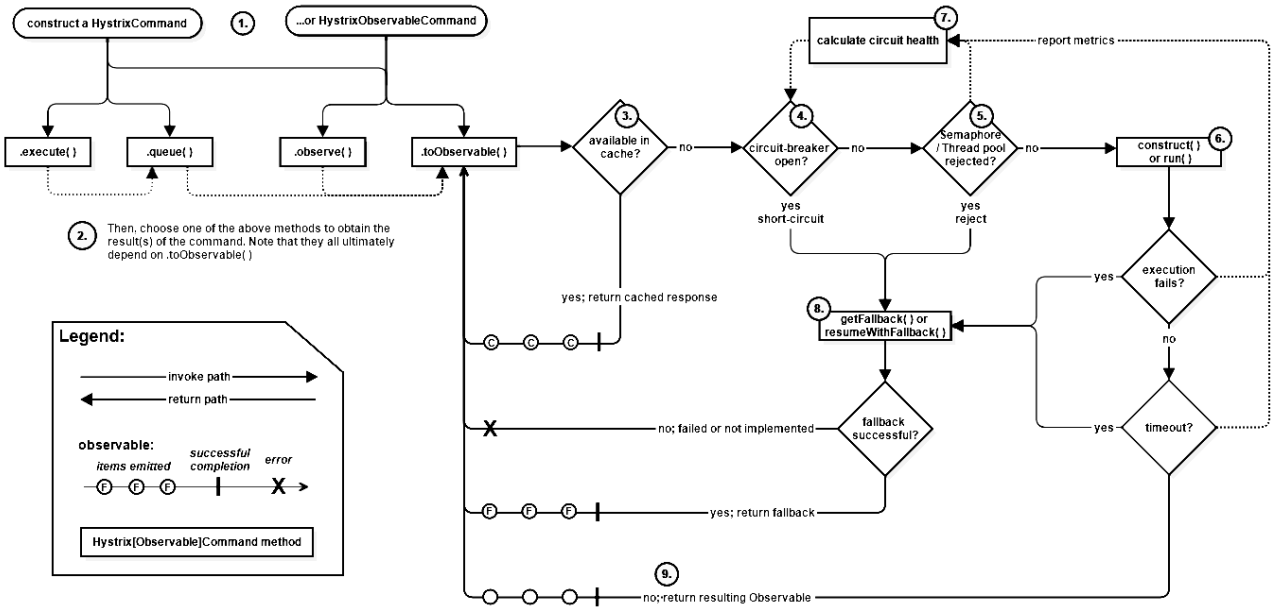

Fig. 2. Hystrix Flow chart

mand but will immediately route the flow to (8) Get the Fallback.

Hystrix reports successes, failures, rejections, and timeouts to the Circuit Breaker, which maintains a rolling set of counters that calculate statistics.

It uses these stats to determine when the circuit should "trip", at which point it short-circuits any subsequent requests until a recovery period elapses, upon which it closes the circuit again after first checking certain health checks.

Hystrix tried to revert to your fallback whenever a command execution fails: when an exception is thrown by construct() or run() (6), when the command is short-circuited because the circuit is open (4), when the command's thread pool and queue or semaphore are at capacity (5), or when the command has exceeded its timeout length.

Write your fallback to provide a generic response, without any network dependency, from an in-memory cache or by means of other static logic.

If you must use a network call in the fallback, you should do so by means of another Hystrix command.

If the Hystrix command succeeds, it will return the response or responses to the caller in the form of an Observable.

Interaction of Hystrix commands with Hystrix Circuit Breaker and its flow of logic and decision-making is shown in the Figure 3.

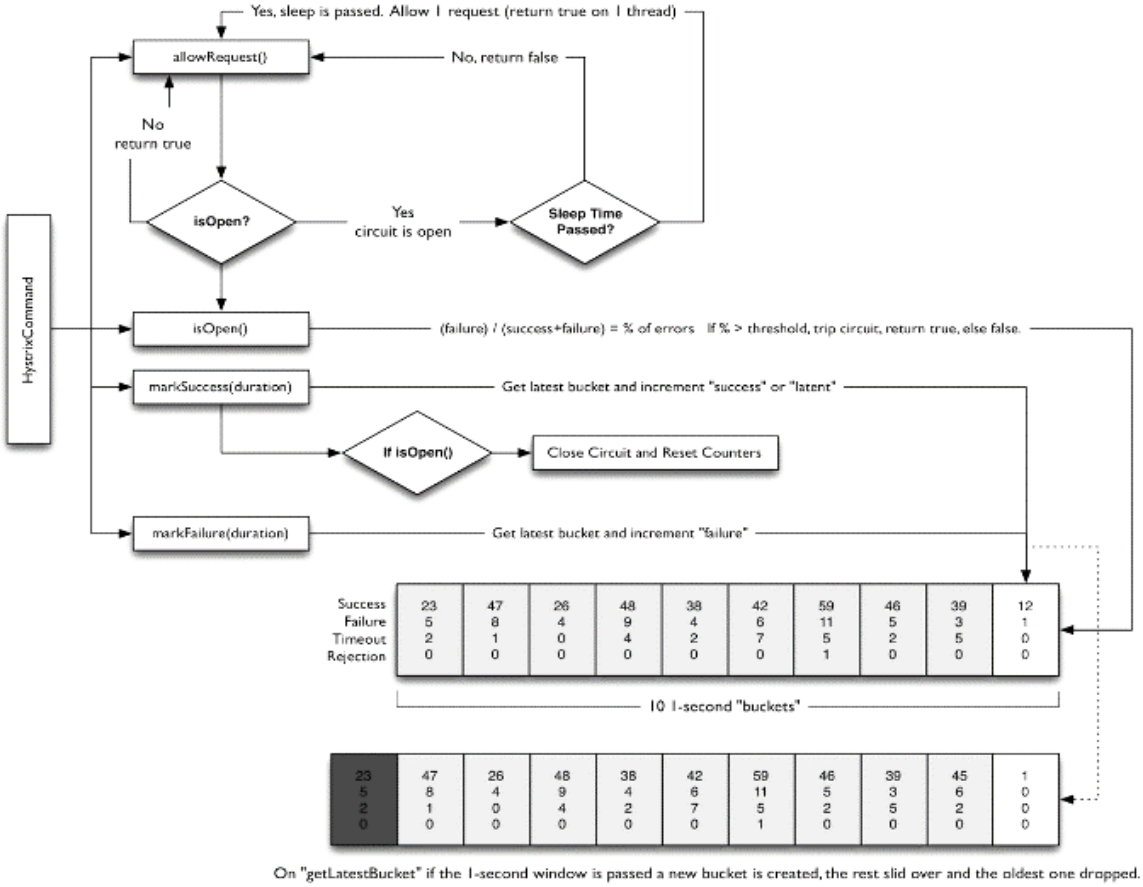

Fig. 3. Interaction with Hystrix Circuit Breaker

The precise way that the circuit opening and closing occurs is as follows:

1. Assuming the volume across a circuit meets a certain threshold
2. And if the error percentage exceeds the threshold error percentage
3. Then the circuit-breaker transitions from *closed* to *open*.
4. While it is open, it short-circuits all requests made against that circuit-breaker.
5. After some amount of time, the next single request is let through (this is the *half-open* state). If the request fails, the circuit-breaker returns to the *open* state for the duration of the sleep window. If the request succeeds, the circuit-breaker transitions to *closed* and the logic in 1 takes over again.

Conclusions

Circuit Breaker has been reviewed and explained. This pattern is emerging as essential for the reliability, ease of access, and flexibility of MSAs. Since microservices is in its early development, we can expect more patterns like this to appear in the future. It is interesting that it is structural [13], in the sense that they do not change the operations that services offer. Being of this nature, their implementations benefit from parametricity to achieve reusability. However, their adoption also makes MSAs more complicated, and they influence the communication structures that will be enacted in a system. This suggests that methods for the programming and verification of communications among services should keep patterns such as these into account.

REFERENCES

1. Lightbend. Akka's Circuit Breaker Pattern (2018), vailable at: http://doc.akka.io/docs (last accessed August 20, 2018).
2. Netflix Hystrix (2018), available at: https://github.com/Netflix/Hystrix/wiki/How-it-Works (last accessed August 21, 2018).
3. Martin Fowler. Circuit Breaker (2018), available at: https://martinfowler.com/bliki/CircuitBreaker.html (last accessed August 21, 2018).
4. Michael T. Nygard (2007), *Release It!* , 326 p., ISBN: 978-0-9787-3921-8.
5. Cloud design patterns by Microsoft (2018), available at: https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker (last accessed August 30, 2018).
6. Sam, Newman (2015), *Building Microservices*, O'Reilly Media, Inc., 282 p.
7. Matthias, K. and Kane, S.P. (2015), *Docker: Up & Running*, O'Reilly Media, Inc., 230 p.
8. Nicola, Dragoni, Saverio, Giallorenzo, Alberto, Lluch-Lafuente, Manuel, Mazzara, Fabrizio, Montesi, Ruslan, Mustafin and Larisa, Safina (2016), *Microservices: yesterday, today, and tomorrow*. CoRR, abs/1606.04036, 322 p..
9. Fowler, M. and Lewis J. (2014), *Microservices*, ThoughtWorks, 185 p.
10. Little M. *SOA versus microservices?* (2018), available at: http://www.infoq.com/news/2015/02/special-microservices-mark-litle (last accessed at August 21, 2018).
11. Hystrix and resilience (2018), available at: http://callistaenterprise.se/blogg/teknik/2017/09/11/go-blog-series-part11/ (last accessed August 21, 2018).
12. Gentle introduction to Hystrix (2018), available at: https://dzone.com/articles/gentle-introduction-to-hystrix-hello-world (last accessed August 21, 2018).
13. Structural design patterns (2018), available at: https://sourcemaking.com/design_patterns/structural_patterns (last accessed August 21, 2018).

Received (Надійшла) 21.09.2018
Accepted for publication (Прийнята до друку) 14.11.2018

Circuit Breaker у системах, що базуються на мікросервісній архітектурі

Г. І. Молчанов, А. Ю. Жмаєв

Предмет статті – дослідження можливості використання паттерну Circuit Breaker у системах, що базуються на мікросервісній архітектурі. **Метою** є аналіз переваг та недоліків паттерну Circuit Breaker для мікросервісів. **Результати.** Спосіб застосування Circuit Breaker полягає в наступному: припускаємо, що гучність в ланцюзі відповідає певному порогу; якщо відсоток помилок перевищує відсоток порогової помилки, то автоматичний вимикач переходить із замкнутого в розімкнутий режим. Поки він розімкнутий, він замикає всі запити, зроблені до цього автоматичного вимикача. Через деякий час наступний одиночний повторний квест пропускається (це напіввідкритий стан). Якщо запит не виконується, автоматичний вимикач вертається в розімкнутий стан на час очікування. Якщо запит виконується успішно, то автоматичний вимикач переходить в замкнутий режим. **Висновок.** Досліджена та обґрунтована можливість використання паттерну Circuit Breaker при розробці програмних комплексів з використанням мікросервісної архітектури. Цей паттерн є надзвичайно важливим для забезпечення надійності, легкості доступу та гнучкості мікросервісів, що розробляються. Важливо, що це структурний паттерн, адже він не впливає на функціональні можливості, що надають сервіси. Відповідно до своєї природи він дозволяє досягти багаторазового використання завдяки параметризації. В той же час, це робить MSA більш складною, а комунікаційні структури, які будуть введені до системи, можуть підлягати деяким змінам. Це свідчить про те, що, обираючи методи програмування та перевірки комунікацій між сервісами, на етапі дизайну рішення необхідно брати до уваги, що використовується саме такий паттерн, як Crcuit Breaker.

Ключові слова: мікросервіс; Circuit Breaker; Hystrix, програмне забезпечення; апаратне забезпечення.

Circuit Breaker в системах на основе микросервисной архитектуры

Г. И. Молчанов, А. Ю. Жмаев

Предмет статьи – исследование возможности использования паттерна Circuit Breaker в системах, основанных на микросервисной архитектуре. **Целью** является анализ преимуществ и недостатков паттерна Circuit Breaker для микросервисов. Результаты. Способ применения Circuit Breaker заключается в следующем: предполагаем, что громкость в цепи соответствует определенному порогу; если процент ошибок превышает процент пороговой ошибки, то автоматический выключатель переходит из замкнутого в разомкнутый режим. Пока он разомкнут, он замыкает все запросы, сделанные к этому автоматическому выключателю. Через некоторое время следующий одиночный повторный квест пропускается (это полуоткрытое состояние). Если запрос не выполняется, автоматический выключатель возвращается в разомкнутое состояние на время ожидания. Если запрос выполняется успешно, то автоматический выключатель переходит в замкнутый режим. **Вывод.** Исследована и обоснована возможность использования паттерна Circuit Breaker при разработке программных комплексов с использованием микросервисной архитектуры. Этот паттерн является чрезвычайно важным для обеспечения надежности, легкости доступа и гибкости разрабатываемых микросервисов. Важно, что Circuit Breaker является структурным паттерном и не влияет на функциональные возможности, которые предоставляют сервисы. В соответствии со своей сущностью, он позволяет достичь многократного использования благодаря параметризации. В то же время, это делает MSA более сложной, а коммуникационные структуры, которые будут введены в систему, могут подлежать некоторым изменениям. Это свидетельствует о том, что, выбирая методы программирования и проверки коммуникаций между сервисами, на этапе дизайна решения необходимо принимать во внимание, что используется именно такой паттерн, как Crcuit Breaker.

Ключевые слова: микросервис; Circuit Breaker; Hystrix; программное обеспечение; оборудование.