From the decorator pattern to circuit breakers in microservices

Fabrizio Montesi
University of Southern Denmark
Odense, Denmark 5230
fmontesi@imada.sdu.dk

Janine Weber
Schneider Electric Denmark
Ballerup, Denmark
Janine.Weber@schneider-electric.com
University of Southern Denmark
Odense, Denmark 5230

ABSTRACT

We analyse different deployment setups for circuit breaker, a design pattern for preventing cascading failures by guarding calls towards a target service. Then, we define a unifying implementation strategy in the setting of microservices, by using the Jolie programming language. Our implementation captures all setups with a single program, by interpreting a circuit breaker as a decorator that is generic on the interface of its target service.

CCS CONCEPTS

• **Computer systems organization → Reliability**; • **Software and its engineering → Design patterns**; **Error handling and recovery**;

KEYWORDS

Microservices, Circuit Breaker, Decorator Pattern

ACM Reference format:
Fabrizio Montesi and Janine Weber. 2018. From the decorator pattern to circuit breakers in microservices. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 3 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Microservices is an emerging development paradigm where the components of an application are autonomous services that execute independently and communicate by message passing [7].

Dealing with communication failures is key to microservice programming, since all interactions among services happen through communications that, in general, may fail. Since services typically depend on other services, an error at a service may cause a cascading failure that influences a large part of a microservice system.

The circuit breaker design pattern is a specialisation of the decorator pattern that mitigates cascading failures [14]. The idea is to "fail fast": when a service becomes unresponsive, its invokers should stop waiting for it and assume the worst, i.e., act as if the service has become unavailable. Thus, 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.

In this paper, we present a systematic analysis of the different deployment strategies for circuit breaker. We then show how the Jolie programming language [11] can be used to define a generic circuit breaker implementation that can be used for all strategies, without requiring changes to its code. Our implementation relies on aggregation, a native feature of Jolie that allows to program decorators that work over communications without requiring changes to the code of either clients or services [6].

2 STRATEGIES AND IMPLEMENTATION

A circuit breaker intercepts and monitors calls towards a target service. When the target service becomes too slow or replies too often with faults, the breaker will trip and future client invocations will be rejected with a fault. The behaviour of the pattern is determined by a state machine. We briefly describe its states and transitions.
Closed: Requests are passed to the target service. Faults and timeouts caused by the requests increase internal failure and timeout counters. When these counters pass a threshold, or a critical fault is detected, the breaker "trips" and transitions into the open state.
Open: Requests are not passed to the target service. Instead, failures are sent to the client as replies. The circuit breaker can transition to the half-open state after some time, possibly by using periodic observation of the health of the target service (e.g., pinging).
Half-open: Requests are passed to the target service in limited number. After some requests are successful, the circuit breaker goes back into the closed state. Should any requests fail while in this state, the circuit breaker transitions back into the open state.

Hystrix [13] is a mainstream implementation of circuit breaker, which wraps invocation code at the client. However, we observe that in general circuit breakers can be usefully adopted in three different ways, depending on the scenario. We describe them below.

Client-side Circuit Breaker. Clients include a separate circuit breaker for intercepting calls to each external service. The main advantage of this strategy is that an open circuit breaker will prevent its target service from receiving further messages. However, this requires two strong assumptions: we can force clients to use our circuit breakers (e.g., access to the client source code); and all clients are not malicious (they will actually execute our code). Another disadvantage is that the knowledge about the availability of a service is local to the client. To counteract this issue, all clients might regularly ping each target service to inquire about its health (but this functionality should then be supported by services).

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Service-side Circuit Breaker. All client invocations received by a service are first processed by an internal circuit breaker, which decides whether the invocation should be processed or not. A benefit is that we do not need the assumptions for client-side circuit breakers that we described (e.g., clients can be malicious). However, the service now uses resources to run the circuit breaker and receive messages even when the circuit breaker is open. An interesting aspect is that the service can see aggregate information about its responsiveness that encompasses requests from all clients, which enables throttling requests based on overall performance.

Proxy Circuit Breaker. Circuit breakers are deployed in a proxy service between clients and services. The proxy contains a circuit breaker for every client and every service within the system. For any request from a client to a service to be allowed to go through, the respective circuit breakers of both client and service must be closed. Observe that using a single proxy for multiple services introduces a network bottleneck, which in some cases plays well with the system (e.g., in case the proxy can be deployed at a routing point) and other times it does not. In the latter cases, it may be desirable to have one proxy for each target service. The proxy approach has two main benefits. First, no invasive changes are made at either clients or services (we simply need to update binding/discovery information). Second, clients and services are equally protected from each other: clients are made more resilient against faulty services, and services are shielded against cases in which a single client sends too many requests. This also enables using shared knowledge among the circuit breakers, for more refined strategies.

2.1 Implementation

We sketch an implementation of a circuit breaker using Jolie. The motivation for using Jolie is that all components in Jolie are (micro)services, and their definition is independent from their deployment. As such, our implementation can be adopted in all deployment strategies that we reported in the previous discussion, simply by loading it appropriately where desired. Jolie services can be deployed both as internal components that communicate using local memory or as distributed over a network. Our definition is also independent from the interface of the target service, and the transport used for communicating messages (we support all transports offered by Jolie, e.g., HTTP/JSON and the binary protocol SODEP).

The code (sketch) of our implementation is displayed in Figure 1. We describe its functioning, and also the necessary Jolie concepts as we encounter them. Line 1 declares an interface `TargetIface`, which is to be bound at deployment time through a separate configuration file (see [12] for details). Our implementation is thus generic on this interface. The **`outputPort`** `TargetSrv` (Lines 2–3), which is also configured at deployment time, represents the target service of the circuit breaker. In Lines 4–5, the **`inputPort`** `CB` defines the input endpoint that will receive client messages. The **`Aggregates`** part means that all client messages received by the circuit breaker (on input port `CB`) for an operation declared by the target service (`TargetSrv`) will be forwarded to the latter—**`with`** `CBFault` declares that the circuit breaker may decide to send clients faults of type `CBFault`. Lines 27–38 define a **`courier`** behaviour [6] for decorating input port `CB`, which intercepts and manages messages in transit from clients to the target service. In Lines 28–29, we intercept all messages for an operation defined in the interface of our target service (`TargetIface`); `request` is the variable that stores the client message, and `response` is the variable that will be used at the end to send the response to the client. We then implement the circuit breaker state machine, storing the current state in variable `state`. We use a `Stats` component (omitted here) to store and compute the typical statistics used in the logic of a circuit breaker, e.g., taking into consideration fault thresholds and rolling windows. We describe how each state works in the following.

Closed (Line 30). We call procedure `forwardMsg`, defined in Lines 14–23. The procedure first calls `callTimer` (omitted here), which starts a "call" timer. We then install a fault handler, which will be executed in case invoking the target service raises an error. In case of error, the handler would cancel the call timer and register the failure in `Stats` (Line 18), and check whether we should change state by invoking procedure `checkErrorRate` (Line 19). In this procedure (Lines 8–13): if we are in a closed state, we ask `Stats` whether we should trip the circuit breaker, based on the data accumulated so far about successes, timeouts, and failures; if, instead, we are in a half-open state, then we trip the circuit breaker immediately. In Line 21, we **`forward`** the message from the client to the target service. If we are successful, we register the success in `Stats` and cancel the call timer (Line 22).

If the call timer expires, a message for operation `callTimeout` is sent to the circuit breaker. This is handled in the **`main`** procedure of the service, Lines 41–43, where we update the internal statistics by registering that a timeout occurred and then check the error rate of the service (which may trip the circuit breaker).

Open (Line 31). We do not forward client requests and instead reply directly to clients with a message containing fault `CBFault`.

Half-Open (Line 32–36). We ask `Stats` whether the message can pass (Line 33). If so, we proceed with `forwardMsg` as in the closed state (Line 34). Otherwise, we send back to the client a fault `CBFault` as in the open state (Line 35).

Procedure `trip` trips the circuit breaker (Line 7), and also starts a reset timer. When this timer expires, operation `resetTimeout` is invoked, triggering a transition to the half-open state (Lines 44–47).

3 RELATED WORK AND CONCLUSIONS

Akka [9] provides a circuit breaker implementation that supports basic configuration parameters, such as call timeout, failure threshold and reset threshold. Hystrix [13], another implementation, is currently more flexible: it supports rolling statistics, fallback mechanisms, resource control, and control over the states and transitions of circuit breakers. Differently from these solutions, our circuit breaker is non-invasive, meaning that adopting it does not require internal code changes at clients or services. Furthermore, our implementation is parametric on the interface of the target service. Therefore, if such interface changes over time (which happens often in microservices, due to the practice of continuous integration and deployment), then the circuit breaker can be reused without any changes. Hystrix does not support this capability: supporting a new operation requires writing an additional implementation of a `HystrixCommand`. Importing this feature to mainstream circuit breaker frameworks could be an interesting future development.

```
interface TargetIface
outputPort TargetSrv
  { Interfaces: TargetIface }
inputPort CB
  { Aggregates: TargetSrv with CBFault }

define trip { state = Open; resetTimer }
define checkErrorRate {
  if ( state == Closed ) {
    shouldTrip@Stats()( shouldTrip );
    if ( shouldTrip ) trip
  } else if ( state == HalfOpen ) trip
}
define forwardMsg {
  callTimer;
  install(
    default =>
      cancelCallTimer; failure@Stats();
      checkErrorRate
  );
  forward( request )( response );
  success@Stats(); cancelCallTimer
}
courier CB {
  [ interface TargetIface
              ( request )( response ) ] {
    if ( state == Closed ) forwardMsg
    else if ( state == Open ) throw( CBFault )
    else if ( state == HalfOpen ) {
      checkRate@Stats()( canPass );
      if ( canPass ) forwardMsg
      else throw( CBFault )
    }
  }
}

main {
  [ callTimeout() ] {
    timeout@Stats(); checkErrorRate
  }
  [ resetTimeout() ] {
    if ( state == Open )
      { reset@Stats(); state = HalfOpen }
  }
}
```

Figure 1: Circuit Breaker Implementation (Sketch).

Looking at future research, we observe that it is not obvious how current methods and theories for the description of interaction protocols among services—*choreographies* for short—can be applied to systems with circuit breakers. Formal methods and languages based on choreographies have been created for different purposes, including: system specification [15, 17]; synthesis of service implementations [5, 10]; and, verification of safety properties (e.g., deadlock-freedom) [8, 18]. Unfortunately, circuit breaker cannot be readily implemented in these models. The latter require extensions to deal with some necessary features, e.g., timeouts, faults, and interface parametricity. There are promising works that deal with timeouts [1], faults [2, 3], and parametric behaviour [4, 16]. However, these works are not integrated with one another and some still need to be applied. A coherent choreography language that can capture circuit breakers still has to appear.

ACKNOWLEDGMENTS

This work was partially supported by the Open Data Framework project at the University of Southern Denmark, and by the Independent Research Fund Denmark, grant DFF-7014-00041.

REFERENCES

[1] Laura Bocchi, Julien Lange, and Nobuko Yoshida. 2015. Meeting Deadlines Together. In *CONCUR*. 283–296. https://doi.org/10.4230/LIPIcs.CONCUR.2015.283
[2] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. 2016. Global escape in multiparty sessions. *Mathematical Structures in Computer Science* 26, 2 (2016), 156–205. https://doi.org/10.1017/S0960129514000164
[3] Marco Carbone. 2009. Session-based Choreography with Exceptions. *Electr. Notes Theor. Comput. Sci.* 241 (2009), 35–55. https://doi.org/10.1016/j.entcs.2009.06.003
[4] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. 2016. Coherence Generalises Duality: A Logical Explanation of Multiparty Session Types. In *CONCUR (LIPIcs)*, Vol. 59. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 33:1–33:15.
[5] Marco Carbone and Fabrizio Montesi. 2013. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*. 263–274.
[6] Mila Dalla Preda, Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. 2012. Interface-Based Service Composition with Aggregation. In *ESOCC*. 48–63. https://doi.org/10.1007/978-3-642-33427-6_4
[7] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. 2017. Microservices: Yesterday, Today, and Tomorrow. In *Present and Ulterior Software Engineering*. Springer, 195–216.
[8] Kohei Honda, Nobuko Yoshida, and Marco Carbone. 2016. Multiparty Asynchronous Session Types. 63, 1 (2016), 9.
[9] Lightbend. 2017. Akka Circuit Creaker Pattern. http://doc.akka.io/docs. (2017). Section 7.4.
[10] Fabrizio Montesi. 2013. *Choreographic Programming*. Ph.D. Thesis. IT University of Copenhagen. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.
[11] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. 2014. Service-Oriented Programming with Jolie. In *Web Services Foundations*. Springer, 81–107.
[12] Fabrizio Montesi and Dan Sebastian Thrane. 2017. Packaging Microservices. In *DAIS (Lecture Notes in Computer Science)*, Vol. 10320. Springer, 131–137.
[13] Netflix. 2017. Hystrix. https://github.com/Netflix/hystrix. (2017).
[14] Michael T. Nygard. 2007. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf.
[15] Object Management Group. 2011. Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/. (2011).
[16] Nicolas Tabareau, Mario Südholt, and Éric Tanter. 2014. Aspectual session types. In *MODULARITY*. 193–204. https://doi.org/10.1145/2577080.2577085
[17] W3C. 2004. Web Services Choreography Description Language. https://www.w3.org/TR/ws-cdl-10/. (2004).
[18] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. 2013. The Scribble Protocol Language. In *TGC*. 22–41. https://doi.org/10.1007/978-3-319-05119-2_3