Circuit Breakers, Discovery, and API Gateways in Microservices

Fabrizio Montesi
Department of Mathematics and Computer Science
University of Southern Denmark
5230 Odense M, Denmark
fmontesi@imada.sdu.dk

Janine Weber
Department of Mathematics and Computer Science
University of Southern Denmark
5230 Odense M, Denmark
jaweb10@student.sdu.dk

ABSTRACT

We review some of the most widely used patterns for the programming of microservices: circuit breaker, service discovery, and API gateway. By systematically analysing different deployment strategies for these patterns, we reach new insight especially for the application of circuit breakers. We also evaluate the applicability of Jolie, a language for the programming of microservices, for these patterns and report on other standard frameworks offering similar solutions. Finally, considerations for future developments are presented.

Keywords

Design Patterns; Microservices; SOA

1. INTRODUCTION

In the *microservices* architectural style [15], the components of an application are autonomous services that execute independently and communicate via message passing [14]. This style is inspired by Service-Oriented Architecture (SOA). The key difference between the two approaches lies in granularity. Even if services in SOA applications also communicate via message passing, differently from microservices the internal components of each application are all part of a single executable artifact, called a monolith. Consider, for example, a service in an SOA that includes an Auth(entication) and an Email component. In microservices, the two components would also be external services, each with its own database. Such services are sometimes called microservices, to point out that they are designed using the microservices style.

Some key advantages that come from the granularity of a microservice architecture (MSA for short) are (see [14] for a more thorough analysis):

- Components can be deployed separate, allowing for the independent management of their respective lifecycles.
- New versions of components can be gradually introduced in a system, by deploying them side to side with previous versions. This advantage can be incorporated in Continuous Integration.
- Components can be more specialised, since they can be written in different technologies – as long as these technologies support interaction with the other technologies used in the same MSA, via message passing.
- Scaling a microservice architecture does not imply a duplication of all its components and developers can conveniently deploy/dispose instances of services with respect to their load [16].

Microservices has become increasingly popular over the last few years. Some companies, such as Netflix and Amazon, have successfully become early adopters of microservices in the setting of large-scale software systems.

Alas, the adoption of microservices also comes with its own set of issues. While many of these issues are inherited directly from distributed systems, they are also exacerbated by the high degree of distribution of an MSA and the fact that we must take them into account even for the composition of internal components. Some key issues include:

- Interactions among microservices happen via message passing, which introduces the possibilities of communication failures and timeouts among components.
- Services may become overloaded, because of too many concurrent client requests or resources being kept busy while waiting for replies from other services. This may easily trigger disastrous cascading failures.
- Microservices are optimised for cloud computing, so some services may be relocated at runtime.
- Microservices can use different technologies, enabling specialisation to specific clients and tasks. Also, MSAs are flexible and their APIs may change over time. Therefore, MSAs should be supported by means for the rapid publishing of new APIs of different natures.

The solutions to these problems come from different sources. The first two issues can be solved with patterns from highly-available systems, the third by using discovery and deployment mechanisms studied for SOA. Furthermore, many of such solutions are aimed at specific applications and are provided by different vendors (e.g., the API gateways by Amazon [1] and Netflix [32]). Differently, our aim in this work is to discuss the principles behind the respective solutions for these problems. Our main contribution is a homogeneous overview of a set of common solutions that microservices developers should be aware of, equipped with novel insight related to their specific applications to microservices. We also demonstrate how the solutions that carry novelty in the setting of microservices can be prototyped using constructs developed for service composition in Jolie [26], a native microservice programming language [20]. This is useful both as a reference and as an evaluation of Jolie itself.

Structure of the paper. We start our investigation from the pattern of circuit breaker, which deals with the first two issues given above (§ 2). While circuit breakers are typically employed client-side, we observe that they can be useful also at other locations, and develop a Jolie prototype that can be transparently reused regardless of where it is located. We proceed by reporting on service discovery, which deals with the third issue, and discuss briefly two main strategies: client- and server-based discovery (§ 3). Our survey ends with a discussion of API Gateway, a pattern for the rapid deployment of new APIs in MSAs (§ 4). Related work is in § 5. We report on conclusions and future work in § 6.

2. CIRCUIT BREAKERS

Given enough incoming requests, even the most reliable of services will eventually exhaust its capabilities and fail. Failure in an MSA is inevitable, and should be embraced with precaution rather than ignored. What makes matters complex is that, in an MSA, a failing service probably has other services that depend on it. What happens if our failing service becomes unresponsive? If we do not properly plan for this event, we risk all the other services that rely on it to become unresponsive, too. This is called a cascading failure.

The circuit breaker pattern is aimed at preventing the failure of a single component to cascade beyond its boundaries, and thereby bring the entire system down with it. The motto here is to fail fast: when a service becomes unresponsive, its invokers should stop waiting for it, assume the worst, and start dealing with the fact that the failing service may be 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.

Concretely, a 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. meeting More specifically, the pattern can be implemented as a finite-state machine, depicted in Figure 1. We describe these states in the following.

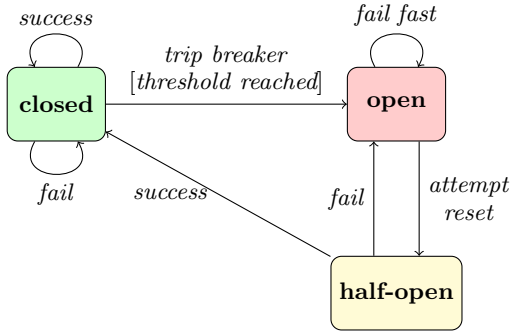

Figure 1: Circuit Breaker State Diagram.

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. Should these counters exceed a specified threshold, or should another predefined criteria be met (e.g., a particular fault was raised), 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 from the open 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. Provided that the target service sends back successful replies, the circuit breaker is reset back into the closed state, and its failure and timeout counters are reset. Should, however, 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.

Deployment.

One of the most famous implementations of circuit breakers is provided by the Hystrix library [30], which allows to wrap Java code in a procedure that will be controlled by a circuit breaker. The idea is that the circuit breaker is used directly inside of the client. Here, we make the (novel) observation that it makes sense to deploy a circuit breaker also in other places than just inside of clients. Specifically, circuit breakers may also be introduced on the side of services, or in proxies that operate between clients and services. Each strategy has its own advantages and disadvantages. Therefore, we envision that practical applications should combine them. We start from the standard strategy of client-side circuit breakers, for reference, and then move to the other ones.

Parameter	Example Value	Explanation
callTO	20s	timeout the client request after 20 seconds without a response from the server
rollingWindow	60s	monitor errors over a rolling window of 60 seconds
tripThreshold	5%	open the circuit if the error rate gets $\geq 5\%$
resetTO	30s	attempt to reset the circuit after 30 seconds of opening the circuit

Table 1: Example of Circuit Breaker Parameters

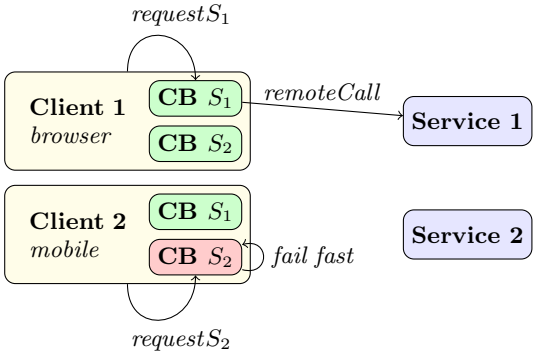

Figure 2: Client-side Circuit Breakers.

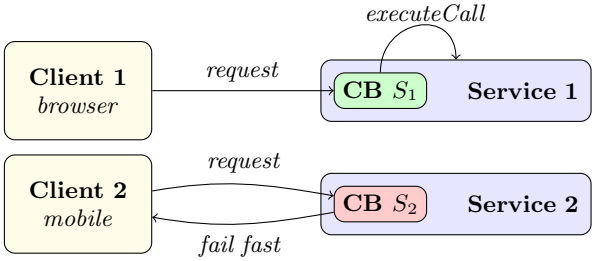

Figure 3: Service-side Circuit Breakers.

Client-side Circuit Breaker. The first deployment strategy is to place circuit breakers directly within clients, as depicted in Figure 2. In this strategy, each client includes a separate circuit breaker for intercepting calls to each external service that the client may call. The strongest advantage of this strategy is that, when the circuit breaker is open, the target service will not receive any messages from the client. This means that the service does not need to implement any similar protection mechanisms of its own, relieving it from using resources for such mechanisms. However, this requires two strong assumptions: we are able of forcing clients to use our circuit breakers (e.g., access to the client source code); and, we are guaranteed that all clients are not malicious (they will actually execute our code). What if our scenario does not satisfy these requirements? Another disadvantage is that the knowledge about the availability of a service is local to the client, depending on how often requests are made to that service. For instance, in Figure 2 *Client 1* is unaware of the unavailability status of *Service 2*, whereas the circuit breaker of *Client 2* for the same service has been recently blown (denoted by the red colour). To counteract this issue, regular pings could be sent out to every service in order to inquire about their health status (but this functionality should then be supported by the services).

Service-side Circuit Breaker. Circuit breakers can also be implemented on the side of services, as presented in Figure 3. The idea is that 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 here is that we do not make any of the assumptions necessary for client-side circuit breakers (in particular, clients can be malicious). However, we have to change the behaviour of the service (e.g., by changing its source code); also, the service uses resources to run the circuit breaker and receive messages even when the circuit breaker is open. An interesting aspect here is that the service can see aggregate information about its responsiveness that encompasses requests from all clients. A possible application is to develop a circuit breaker that throttles requests to temporarily lighten the load on the service.

Proxy Circuit Breaker. The last option that we present for the deployment of circuit breakers is a composition of the previous two strategies. In this strategy, circuit breakers are deployed in a proxy service that sits between clients and services, which handles all incoming and outgoing messages as displayed in Figure 4. The proxy contains a circuit breaker for every client and every service within the system.[1] 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. For instance, in the case illustrated in Figure 4, *Client 1* is allowed to send a request to *Service 1*, but receives an exception when trying to call *Service 2*. Furthermore, *Client 2* immediately gets denied any requests. 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. However, we will see that a proxy aggregating multiple services makes sense in another pattern presented later in § 4, the API Gateway.

Using proxies for deploying circuit breakers has two main benefits. First, this architectures simply requires to configure clients to point to the proxy instead of the services directly. In many cases, this does not require access to the client source code, but simply either a network reconfiguration or passing some location parameter to clients to bind them correctly. It is also unnecessary to modify the code or configuration of the target services, which can be seen as black boxes. Second, clients and services are equally

[1] In systems where new clients and/or services may join at runtime, circuit breakers may have to be created dynamically, but this is orthogonal to our discussion.

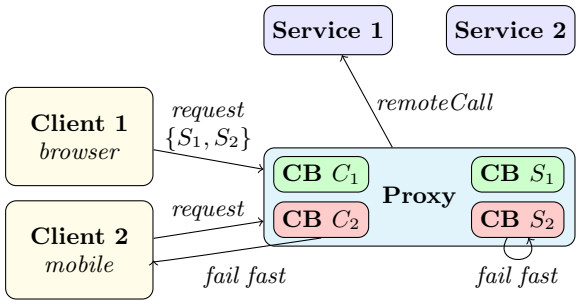

Figure 4: Proxy with Circuit Breakers.

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 opens up the possibility of using shared knowledge among the circuit breakers, for more refined strategies.

Implementation.

We sketch an implementation of a circuit breaker using Jolie, a native programming language for microservices [26]. The motivation for using Jolie is that all components in Jolie are (micro)services, and their definition is independent of their deployment. As such, our prototype can be adopted in all deployment strategies that we reported in the previous discussion, simply by loading it appropriately where desired. Specifically, this is because services in Jolie can be deployed both as internal components that communicate using local memory or as distributed over a network. Our definition is also independent of the interface of the target service, and the transport used for communicating messages (we support all transports offered by Jolie, e.g., HTTP/JSON, HTTP/XML, and the binary protocol SODEP).

Our prototype, given in Figure 5, is simple enough that it can be discussed without assuming prior knowledge of the Jolie language. We describe its functioning after its code, given in the following, and describe the necessary Jolie concepts as we encounter them.

We describe the prototype. The underlying idea is simple: our program defines a service that intercepts all calls from a client to a target service, applying the logic of a circuit breaker. The `outputPort` `TargetSrv` (Line 1) contains the binding information towards the target service of the circuit breaker. We omit its definition, since this changes depending on the deployment of the circuit breaker. To deploy the circuit breaker client-side or in a proxy, then `TargetSrv` would point to a remote location. For the service-side case, instead, `TargetSrv` would point to a local memory location. In Lines 8–11, the `inputPort` `CB` deploys the input endpoint that will receive client messages. We abstract from the location where it is concretely deployed and the transport protocol that it uses; these are just configuration parameters, denoted *Loc* and *Proto*. The `Aggregates` part is key: it instructs Jolie 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. The part `with` `CBIfaceExt` declares that the types of all operations are augmented according to the definition of `CBIfaceExt`, given in Lines 3–6. Specifically, `CBIfaceExt` states that all operations can now also throw the fault `CBFault`, which we will use to notify clients of failures generated by the circuit breaker. Jolie allows us to write arbitrary code to process the messages intercepted from clients to the target service, called a `courier` behaviour. We define this code in Lines 32–47. In Line 33, we state that we want to 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, using the (global) variable `state` to store our state. We assume that there exists an internal `Stats` component that stores and computes the statistics used in the logic of a circuit breaker, configured accordingly to the tripThreshold and rollingWindow parameters (from Table 1). We report the different cases depending on the current state.

Closed (Line 35) We call procedure `forwardMsg`, defined in Lines 24–31. The procedure starts by calling another procedure `callTO` (omitted here), which starts a timer with duration set by the callTO parameter. We then install a fault handler, which will be executed in case invoking the target services raises an error. In case of error, the handler would cancel the call timer (Line 27), register the failure in `Stats`, and check whether we should change state by invoking procedure `checkErrorRate`. The latter is a simple procedure (Lines 15–22), which if we are currently in a closed state asks `Stats` whether we should trip the circuit breaker, based on the data accumulated so far about successes, timeouts, and failures. In Line 29, 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 30).

What if the call timer expires before it is cancelled (either by a success or an error raised by the `forward` statement)? In this case, a message for operation `callTO` will be sent to our circuit breaker. This is handled in the `main` procedure of the service, Lines 49–59, where if we receive a message for `callTO` we update the internal statistics by registering that a timeout occurred and then check the error rate of the service (which may cause the circuit breaker to trip).

Open (Line 37) While in this state, the circuit breaker does not forward client requests and instead replies directly with a message containing fault `CBFault`.

Half-Open (Lines 39-44) In this state, we ask `Stats` whether the message can pass (Line 39). If so, we proceed with `forwardMsg` as in the open state. Otherwise, we send back to the client a fault `CBFault` as in the open state.

The procedure used to trip the circuit breaker is `trip` (Line 13), which also starts a reset timer. When this timer expires, operation `resetTO` will be invoked and make the transition to the half-open state (Line 56).

```
outputPort TargetSrv { ... }

interface extender CBIfaceExt {
RequestResponse:
  *(void)(void) throws CBFault
}

inputPort CB {
Location: Loc    Protocol: Proto
Aggregates: TargetSrv with CBIfaceExt
}

define trip { state = Open; resetTO }

define checkErrorRate {
  if ( state == Closed ) {
    shouldTrip@Stats()( shouldTrip );
    if ( shouldTrip ) { trip }
  } else if ( state == HalfOpen ) {
    trip
  }
}

define forwardMsg {
  callTO;
  install( default =>
    cancelCallTO; failure@Stats();
    checkErrorRate );
  forward( request )( response );
  success@Stats(); cancelCallTO
}

courier CB {
  [ 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 {
  [ callTO() ] {
    timeout@Stats(); checkErrorRate
  }

  [ resetTO() ] {
    if ( state == Open ) {
      reset@Stats(); state = HalfOpen
    }
  }
}
```

Figure 5: Circuit Breaker Implementation in Jolie.

3. SERVICE DISCOVERY

In practice, the location of a microservice may not be statically known at design time. This is because microservices may be deployed in a cloud-based system, which could replicate and relocate services at runtime. Therefore, a participant in an MSA may need to employ a service discovery mechanism. This is typically achieved with the same idea adopted for Service-Oriented Architecture (SOA), i.e., by using a service registry. A service registry is a service that can be used by other components to retrieve binding information about other components. Microservices talk to the registry in order to publish their locations (or that of other services), whereas clients address the registry to discover registered services.

In principle, there is no difference between using a service registry in an SOA or in an MSA. However, in practice, service discovery for SOA is often implemented as part of an Enterprise Service Bus (ESB), which acts as a central coordination point for service communications. Instead, in microservices, there are still no standards but rather various custom implementations (e.g., Eureka [29] and AWS Elastic Load Balancing [2]). Nevertheless, we can distinguish between two main implementation strategies: client-side and server-side discovery.

In the client-side discovery pattern, depicted in Figure 6, the client is aware that services do not have fixed locations. It thus queries the service registry for the location of all the services that it needs. Thereafter, the client contacts the target services directly. This architecture is simple, but it requires that clients are designed to follow this methodology. The implementation of clients thus becomes more complex, since it has to implement the discovery logic. This logic needs to be replicated for each programming language and/or framework used for the implementation of clients.

In the alternative server-side discovery pattern, displayed in Figure 7, we delegate the discovery logic to a dedicated router service. The client exclusively talks to the router responsible for the services, which is set at a fixed location. Upon receiving a request, the router talks to the service registry to discover the requested service, and then forwards the client request to the latter. Contrary to client-side discovery, this pattern does not require clients to be aware of the fluid deployment of microservices. However, the programmer needs to deploy an additional service (the router) that will consume resources.

Often, both patterns are present within large-scale applications. A server-side discovery structure exposes the public services to the outside world, whereas the client-side discovery pattern handles server- or cluster-internal interactions.

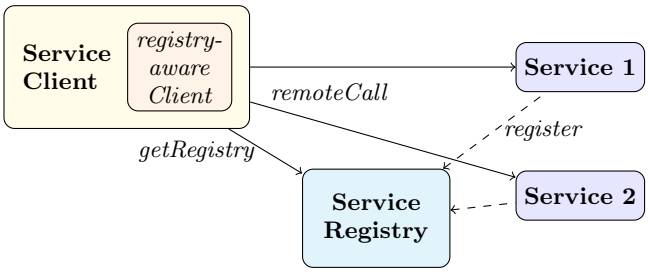

Figure 6: Client-side Service Discovery.

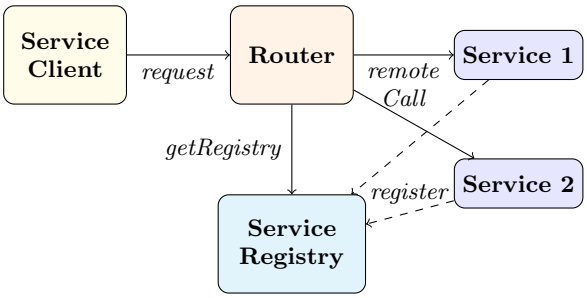

Figure 7: Server-side Service Discovery.

4. API GATEWAYS

An MSA may need to serve different kinds of clients and user interfaces, such as those found in web browsers and various smart devices (e.g., smartphones). Every client may have different needs, depending on its target usage, form factor, and processing power. The needs of a client may even change over time. For example, depending on the quality of its current network connection, a device may want to use an API that is more or less network intensive – for example, the description of a product may include more and higher-quality resources, like pictures or embedded instructions.

The API Gateway is a service that addresses the issue of having clients of different natures. It is a single entry point that provides access to many APIs (Figure 8). An API Gateway provides functionalities for: publishing multiple APIs, each one dedicated to a different set of clients; and, updating the set of published APIs at runtime (since developers may deploy new services during the lifecycle of the MSA).

Since an API Gateway is an entry point for the MSA, it is natural to equip it with, e.g., service discovery, load balancing, monitoring, and security. Its position in the system is also ideal for adopting the proxy circuit breaker pattern, by equipping the API Gateway with circuit breakers for clients and/or services. The way in which these additional features are implemented depends on the implementation technology. In Jolie, this would simply require to compose the specific services developed for the respective functionalities (e.g., our circuit breaker prototype, or a service registry). Other technologies may need to add these features directly inside of the codebase that implements the API Gateway. Observe that for service discovery, both the client-side and server-side discovery patterns make sense here so we should refer to their own advantages and disadvantages to choose between them. For client-side discovery, the API Gateway

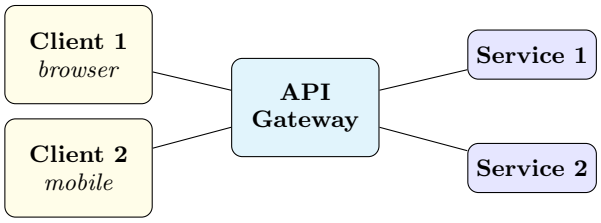

Figure 8: API Gateway.

```
1  inputPort APIGateway {
2  Location: "socket://gateway.com:80/"
3  Protocol: http
4  Redirects:
5    MobileAPI => MobileService,
6    DesktopAPI => DesktopService
7  }
8
9  main {
10   [ deploy( request )() {
11     loadEmbeddedService@Runtime
12                              ( ... )( ... );
13     setRedirection@Runtime( ... )( ... )
14   } ]
15 }
```

Figure 9: A sketch of an API Gateway in Jolie.

should provide clients with access to the service registry – while this access would still be brokered by the gateway, it would still be the clients that select the services that they want (recall Figure 6). For server-side discovery, the API Gateway would simply act as the router (recall Figure 7).

In Jolie, we can implement an API Gateway by using redirections [26]. A redirection makes an API available under a specific name at a service input port. We sketch a prototype in Figure 9. In Lines 1–7, we declare the input port of the gateway (the location and protocol are just examples). The input port provides two APIs, which can be reached respectively at the URLs http://gateway.com:80/MobileAPI and http://gateway.com:80/DesktopAPI. The service offers a `deploy` operation for deploying new APIs at runtime (Lines 10–14). We omit the concrete data used in the operation. The idea is that, upon request, the gateway will use the Jolie standard library to embed (run an internal service) all the necessary services for guarding the new API (e.g., circuit breakers) and set a new redirection to publish the API. Redirection in Jolie takes care of doing the necessary transformations between different communication transports, but sometimes developers need to use special adapters to translate calls using ad-hoc procedures. An advantage in this task is the possibility to use procedures written in different languages. Jolie supports Jolie itself, JavaScript, and Java. For instance, if the mobile API in our redirection example required an adapter written in JavaScript, we could use the following embedding instruction.

```
1 embedded {
2 JavaScript:
```

```
3    "mobile_adapter.js" in MobileService
4 }
```

5. RELATED WORK

Being essentially distributed, microservices is founded on the well-known mechanism of message passing. However, MSAs are much more involved than other distributed applications where services are implemented as monoliths, because all internal components are subjects to potential communication failures and overloads. So far, most proposals for dealing with these problems have been produced by practitioners, and therefore plenty of valuable information has to be found in books and web resources. We give here an overview of such solutions and compare to our work where appropriate. A summary of the technologies that we mention is given in Table 2. We make an abuse of notation by reporting Jolie along with frameworks, even though Jolie is actually a programming language and thus patterns must be implemented. The idea is to point out that all such patterns are naturally supported by Jolie via its language constructs.

Circuit Breakers. Circuit breakers have first been popularised in [35], where their role is discussed in the context of availability (resilience) for enterprise systems.

Akka [22] provides a circuit breaker implementation that supports basic configuration parameters, such as call timeout, failure threshold and reset threshold. Hystrix [30] is much more flexible and is currently one of the reference solutions: it supports rolling statistics, fallback mechanisms, resource control, and control over the states and transitions of circuit breakers. Our circuit breaker prototype in Jolie (Figure 5) is of course not as mature as these implementations, but it is interesting because it can be freely deployed in any of scenarios described in § 2. Furthermore, our circuit breaker is parametric on the interface of the target service. This means that if such interface changes over time (as can often happen in microservicese, e.g., by adding an operation), then the circuit breaker can be re-adopted immediately without any changes. Hystrix does not support this capability: supporting a new operation requires writing an additional implementation of a `HystrixCommand`. Importing this feature could be an interesting future development.

Service Discovery and Load Balancing. Eureka [29] and Ribbon [31] combined together provide client-side load balancing and service discovery. Amazon Web Services Elastic Load Balancing (AWS ELB), instead, implements balancing and discovery using a server-side solution. Therefore, ELB is generally used to expose edge services to the public, and Eureka to handle internal service communications. However, this also means that ELB can become a bottleneck. Eureka is more resilient, as all information is cached by clients. Discovery and load balancing can be achieved in Jolie using the techniques described in [11, 26].

API Gateways. Both Zuul [32] and Amazon Web Services (AWS) [1] provide roughly the same functionalities in their implementation (e.g., security, authentication, monitoring, and load balancing). Zuul consists of different libraries (such as Eureka[29], Hystrix[30], and Ribbon[31]), whereas AWS provides a single framework that may be quicker to use in the beginning. However, since Zuul and all its dependencies are open source, it has the advantage in customisation potential. Redirections in Jolie as we used in § 4 do not take care of extra features such as security and monitoring by themselves. The idea is that an API Gateway in Jolie should be composed with other patterns, e.g., circuit breakers, to achieve this extra features, keeping concerns separate. This is easy to do since all components in Jolie are relocatable services that must define interfaces for enabling composition.

6. CONCLUSIONS AND FUTURE WORK

We reviewed three mainstream mechanisms found in Microservice Architectures (MSAs): Circuit Breaker, Service Discovery, and API Gateway. We discussed different strategies for their implementation, and elicited the interplay between deployment topologies and circuit breakers.

These patterns are 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 these to appear in the future. It is interesting that these patterns are structural, in the sense that they do not change the operations that services provided by developers offer, which are more custom to the specific MSA at hand. Being of this nature, their implementations benefit from parametricity to achieve reusability, as we have shown for circuit breakers by using interface parametricity in Jolie. 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.

Development methodologies for service communications typically employ choreographies for the description of service protocols. Choreographies do not require central control, a critical feature for the scalability of MSAs. Formal methods and languages based on choreographies have been developed for various purposes, including: documenting systems using choreographies [37, 5]; synthesising service implementations starting from choreographies [25, 9]; and, verifying safety properties of choreographies (e.g., deadlock-freedom)[38, 19]. The patterns that we considered cannot be readily implemented in these models. They require extensions to deal with some necessary features, specifically: circuit breakers require timeouts, faults, and interface parametricity; service discovery requires dynamic binding (the capability of connecting to a remote service whose location is discovered at runtime); and, API gateways require the capability of loading new services at runtime. There are promising works that deal with timeouts [4], faults [7, 6], dynamic binding [28], and parametric behaviour [36, 8]. However, all these works are not integrated with one another and a coherent choreography language that can capture, for example, circuit breakers still has to appear. Therefore, designing a model capable of capturing the patterns that we described represents interesting future work. Useful inspiration may be gained also from related work in the area of process calculi and components, given their vicinity in the adopted techniques, including [21, 17, 27, 12].

Pattern	Frameworks, Libraries & Languages
API Gateway	AWS[1], Netflix[32], Nginx[33], Jolie[26, 20]
Circuit Breaker	Hystrix[30], Akka[22], Jolie[26, 20]
Load Balancing & Service Discovery	Nginx[34], Ribbon[31], ELB[2], Eureka[29], etcd[10], Zookeeper[3], Marathon[24], Consul[18], Jolie[20]
Monitoring & Metrics	Docker[13], Hystrix[30], Lightbend[23], Marathon[24], Jolie[26, 20]

Table 2: State of the Art Microservice Frameworks, Libraries & Languages.

7. REFERENCES

[1] Amazon. Amazon Web Services API Gateway. https://aws.amazon.com/api-gateway/.

[2] Amazon. Amazon Web Services Elastic Load Balancing. https://aws.amazon.com/elasticloadbalancing/.

[3] Apache. Zookeeper. https://zookeeper.apache.org.

[4] Laura Bocchi, Julien Lange, and Nobuko Yoshida. Meeting deadlines together. In *CONCUR*, pages 283–296, 2015.

[5] Business Process Model and Notation. http://www.omg.org/spec/BPMN/2.0/.

[6] Sara Capecchi, Elena Giachino, and Nobuko Yoshida. Global escape in multiparty sessions. *Mathematical Structures in Computer Science*, 26(2):156–205, 2016.

[7] Marco Carbone. Session-based choreography with exceptions. *Electr. Notes Theor. Comput. Sci.*, 241:35–55, 2009.

[8] Marco Carbone, Sam Lindley, Fabrizio Montesi, Carsten Schürmann, and Philip Wadler. Coherence generalises duality: A logical explanation of multiparty session types. In *CONCUR*, 2016. To appear.

[9] Marco Carbone and Fabrizio Montesi. Deadlock-freedom-by-design: multiparty asynchronous global programming. In *POPL*, pages 263–274, 2013.

[10] CoreOS. etcd. https://github.com/coreos/etcd.

[11] Mila Dalla Preda, Maurizio Gabbrielli, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Interface-based service composition with aggregation. In *ESOCC*, pages 48–63, 2012.

[12] Ornela Dardha, Elena Giachino, and Michael Lienhardt. A type system for components. In *SEFM*, pages 167–181, 2013.

[13] Docker. Docker. https://www.docker.com/.

[14] Nicola Dragoni, Saverio Giallorenzo, Alberto Lluch-Lafuente, Manuel Mazzara, Fabrizio Montesi, Ruslan Mustafin, and Larisa Safina. Microservices: yesterday, today, and tomorrow. *CoRR*, abs/1606.04036, 2016.

[15] M Fowler and J Lewis. Microservices. *ThoughtWorks*, 2014.

[16] Maurizio Gabbrielli, Saverio Giallorenzo, Claudio Guidi, Jacopo Mauro, and Fabrizio Montesi. Self-reconfiguring microservices. In *Theory and Practice of Formal Methods*, pages 194–210. Springer, 2016.

[17] Claudio Guidi, Ivan Lanese, Fabrizio Montesi, and Gianluigi Zavattaro. Dynamic error handling in service oriented applications. *Fundam. Inform.*, 95(1):73–102, 2009.

[18] HashiCorp. Consul. https://www.consul.io.

[19] Kohei Honda, Nobuko Yoshida, and Marco Carbone. Multiparty asynchronous session types. *Journal of the ACM*, 63(1):9, 2016. Also: POPL, 2008, pages 273–284.

[20] Jolie. Official Website. http://www.jolie-lang.org/.

[21] Cosimo Laneve and Gianluigi Zavattaro. Foundations of web transactions. In *FOSSACS*, pages 282–298, 2005.

[22] Lightbend. Akka's Circuit Creaker Pattern. http://doc.akka.io/docs. Section 7.4.

[23] Lightbend. Monitoring. https://www.lightbend.com/products/monitoring.

[24] Mesophere. Marathon. https://mesosphere.github.io/marathon.

[25] Fabrizio Montesi. *Choreographic Programming*. Ph.D. thesis, IT University of Copenhagen, 2013. http://www.fabriziomontesi.com/files/choreographic_programming.pdf.

[26] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-Oriented Programming with Jolie. In *Web Services Foundations*, pages 81–107. Springer, 2014.

[27] Fabrizio Montesi and Davide Sangiorgi. A model of evolvable components. In *TGC*, pages 153–171, 2010.

[28] Fabrizio Montesi and Nobuko Yoshida. Compositional choreographies. In *CONCUR*, pages 425–439, 2013.

[29] Netflix. Eureka. https://github.com/Netflix/eureka.

[30] Netflix. Hystrix. https://github.com/Netflix/hystrix.

[31] Netflix. Ribbon. https://github.com/Netflix/ribbon.

[32] Netflix. Zuul. https://github.com/Netflix/zuul.

[33] Nginx. Nginx API Gateway Solution. https://www.nginx.com/solutions/api-gateway.

[34] Nginx. Nginx Load Balancing Solution. https://www.nginx.com/solutions/load-balancing/.

[35] Michael T. Nygard. *Release It!: Design and Deploy Production-Ready Software (Pragmatic Programmers)*. Pragmatic Bookshelf, 2007.

[36] Nicolas Tabareau, Mario Südholt, and Éric Tanter. Aspectual session types. In *MODULARITY*, pages 193–204, 2014.

[37] W3C. Web Services Choreography Description Language. https://www.w3.org/TR/ws-cdl-10/.

[38] Nobuko Yoshida, Raymond Hu, Rumyana Neykova, and Nicholas Ng. The scribble protocol language. In *TGC*, pages 22–41, 2013.