Model-Based Analysis of Microservice Resiliency Patterns

Nabor C. Mendonça, Carlos M. Aderaldo
Post Graduate Program in Applied Informatics
University of Fortaleza
Fortaleza, CE, Brazil
Email: {nabor,carlosmendes}@unifor.br

Javier Cámara
Department of Computer Science
University of York
York, UK
Email: javier.camaramoreno@york.ac.uk

David Garlan
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
Email: garlan@cs.cmu.edu

***Abstract*—Microservice application developers try to mitigate the impact of partial outages typically by implementing service-to-service interactions that use well-known resiliency patterns, such as *Retry*, *Fail Fast*, and *Circuit Breaker*. However, those resiliency patterns—as well as their available open-source implementations—are often documented informally, leaving it up to application developers to figure out when and how to use those patterns in the context of a particular microservice application. In this paper, we take a first step towards improving on this situation by introducing a model checking-based approach in which we use the PRISM probabilistic model checker to analyze the behavior of the *Retry* and *Circuit Breaker* resiliency patterns as continuous-time Markov chains (CTMC). This approach has enabled us to quantify the impact of applying each resiliency pattern on multiple quality attributes, as well as to determine how to best tune their parameters to deal with varying service availability conditions, in the context of a simple client-service interaction scenario.**

***Index Terms*—microservices, resiliency patterns, probabilistic model checking**

I. Introduction

Designing reliable distributed systems requires accounting for several types of operational uncertainties, such as service failures and network delays [1]. Anticipating and dealing with different types of failures is part of a fundamental design paradigm commonly referred to as *design for failure* [2], which is one of the tenets of the microservices architectural style [3].

Microservices, like any type of distributed system, are typically fragile. Multiple reasons, such as network, hardware, or application-level issues, might render them unavailable, inaccessible, or simply make them fail. Owing to service dependencies, any service can become temporarily inaccessible to its consumers. At the application level, communication failures will likely occur often because of the sheer number of services involved and of messages being exchanged [4].

To mitigate the impact of partial outages, microservice application developers must build resilient services that can gracefully respond to failure. One common way to do this is by implementing service-to-service interactions using *resiliency patterns*, such as *Retry*, *Fail Fast*, and *Circuit Breaker* [5]. There are several open-source service invocation libraries currently available that implement such resiliency patterns, most of them derived from production-grade solutions originally developed by industry, like Hystrix [6], Finagle [7], and Resilience4J [8]. Those libraries can be used to deal with different types of failures (e.g., transient and non-transient service failures), requiring careful configuration of their invocation parameters (e.g., timeout duration and failure thresholds). However, the existing documentation on those service invocation libraries—as well as on their underlying resiliency patterns—is largely informal, usually provided in the form of general rules of thumb and usage guidelines, e.g.:

> "*Use [the Retry] pattern when an application could experience transient faults as it interacts with a remote service or accesses a remote resource. These faults are expected to be short lived, and repeating a request that has previously failed could succeed on a subsequent attempt*" [9].

In the documentation quoted above, the exact meaning of terms such as *transient* and *short-lived*, used to qualify systems faults, is left open to interpretation. As a consequence, it's mostly up to application developers to figure out the environmental conditions upon which a given resiliency pattern should be used, and how to configure its invocation parameters to deal with the expected failure types of a particular microservice production environment.

Despite recent interest in microservices as a research topic [4], there has been surprisingly little interest in the study of existing resiliency patterns by the research community. Works by Montesi and Weber [10] and Preuveneers and Joosen [11] on circuit breakers are two notable exceptions, but even those works do not consider how a particular circuit breaker implementation should be configured in terms of its timeout and failure threshold parameters.

In this paper, we take a first step towards improving on this situation by shedding some light on the proper use and configuration of two popular microservice resiliency patterns. Specifically, we describe a model checking-based approach in which we use the PRISM probabilistic model checker [12] to analyze the behavior of the *Retry* and *Circuit Breaker* patterns captured as continuous-time Markov chains (CTMC) [13]. This approach enables us to quantify the impact of choosing a

This material is based upon work supported by CNPq under grants 313553/2017-3 and 424160/2018-8, and the NSA under Award No. H9823018D0008.

particular resiliency pattern and its configuration on multiple quality attributes, under varying service availability conditions, in the context of a simple client-service interaction scenario. To the best of our knowledge, this is the first approach that applies probabilistic model checking to analyze the behavior of popular microservice patterns.

The rest of the paper is organized as follows: Section II gives an overview of the *Retry* and *Circuit Breaker* patterns. Section III describes how we modeled those patterns as CTMC. Section IV reports on the method and results of a quantitative analysis of the behavior of the patterns using the PRISM model checker. Section V discusses related work. Finally, Section VI presents conclusions and future research directions.

II. Microservice Resiliency Patterns

Some of the most well-known microservice resiliency patterns (e.g., *Retry*, *Fail Fast*, *Bulkhead*, and *Circuit Breaker*) were introduced over a decade ago in the book “Release It!” [14] by Michael Nygard. Since then, those patterns have grown in popularity, especially in industry, where they have been implemented and used in production as part of a number of fault-tolerant service invocation libraries (e.g., Hystrix [6]) and service mesh tools (e.g., Istio [15]). Here, we briefly describe the *Retry* and *Circuit Breaker* patterns, which are the focus of our work, using a simple client-service interaction scenario to illustrate their use and configuration. Details on these, as well other resiliency patterns, can be found elsewhere [5], [14].

A. Example Scenario

We consider an example scenario in which a single client service sequentially invokes an operation provided by a target remote service (see Fig. 1). The target service serves requests from the client service with a given *response rate*, and may fail and recover from failure with some given *failure rate* and *recovery rate*. Each request may either succeed, in which case the client service receives an “OK” response from the target service, or fail due to the target service being either unavailable or too slow, in which case the client service receives an error message from its underlying service invocation mechanism. In the latter case, the invocation mechanism typically has to wait until either a connection timeout (e.g., when the target service is unavailable) or a response timeout (e.g., when the target service is too slow) occurs before returning the error message to the client service.

An important challenge related to the implementation of the client service in this scenario is how to deal with a non-responsive target service. There are two undesirable cases that should be avoided under such circumstances:

1) If the client service continually retries each failed request, it increases its resource contention, thereby contributing to further overloading the network and/or the target service;

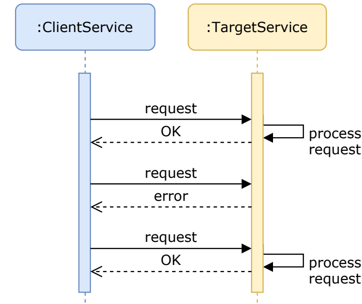


Fig. 1: ClientService-TargetService interaction scenario.

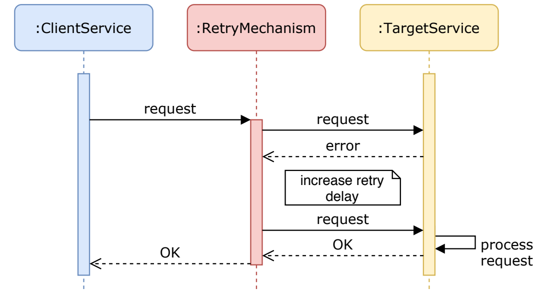


Fig. 2: *Retry* pattern example.

2) If the client service waits too long for the target service to respond, it increases its own execution time, potentially delaying the execution of all other services that depend on it.

The *Retry* and *Circuit Breaker* patterns are well-known design solutions to cope with (the consequences of) failures during service interactions, and that can be used to establish a trade-off between resource contention and execution time, as discussed above. In the following, we describe those two patterns in terms of their purpose, context, and implementation considerations. To this end, we largely draw from the documentation provided in [5].

B. Retry Pattern

Purpose: Enable an application to handle transient failures when it tries to invoke a remote service, by transparently retrying a failed operation.

Context: A distributed application is required to be resilient to the transient faults that can occur in a distributed environment. These faults (e.g., momentary loss of network connectivity, temporary unavailability of a service, and timeouts that occur when a service is busy) are typically self-correcting, and if the action that triggered a fault is repeated after a suitable delay, it is likely to be successful.

Solution: If an application detects a transient failure when it tries to send a request to a remote service, it should wait a suitable amount of time (“backoff”) before retrying the request. This process is repeated until the request succeeds or a failure threshold is reached, in which case the operation is considered to have failed definitely. Fig. 2 shows an example of the *Retry* pattern being used in the context of the client-

service interaction scenario shown in Fig. 1. In this example, the retry mechanism attempts to invoke the target service twice before receiving a successful response. Note that the retry mechanism increases the retry delay after the first attempt, thus allowing a longer period of time for the target service to recover.

Implementation considerations: The period between retries should be chosen to spread requests from multiple instances of the application as evenly as possible. This reduces the chance of a busy service continuing to be overloaded. If necessary, the retry mechanism can increase the delays between retry attempts, until some maximum number of retries have been attempted. The delay can be increased incrementally or exponentially, depending on the type of failure and the probability that it will be corrected during this time. If a request still fails after a significant number of retries, it is better for the retry mechanism to prevent further requests going to the same resource and simply report a failure.

An alternative strategy to deal with less-transient failures is using a circuit breaker.

C. Circuit Breaker Pattern

Purpose: Enable an application to handle faults that might take a variable amount of time to recover from, when invoking a remote service or resource.

Context: In a distributed environment, calls to remote resources and services can fail due to unanticipated events (e.g., loss of connectivity, service failure), and might take much longer to fix than typically self-correcting transient faults. In these situations, it might be pointless for an application to continue to retry an operation that is unlikely to succeed: instead, the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if the calling operation implements a timeout, all concurrent requests to that operation will be blocked until the timeout period expires, possibly leading to cascading failures in other parts of the system. In these situations, it would be preferable for the calling operation to fail immediately, and only attempt to invoke the service if it is likely to succeed.

Solution: A circuit breaker acts as a local proxy for operations that might fail. The proxy should monitor the number of recent failures that have occurred, and use this information to decide whether to allow the operation to proceed, or simply return an error immediately.

The proxy can be implemented as a state machine with the following states that mimic the functionality of an electrical circuit breaker: Closed, Open, and Half-Open (see Fig. 3) [16]. Initially, the circuit is Closed, which means that the proxy will forward every incoming request to the target service. The circuit will remain Closed until a certain number of consecutively failed requests is reached, in which case the circuit moves to the Open state. In this state, the circuit will immediately return an error message to the client service for every new incoming request without invoking the target service, until a timeout expires, in which case the circuit moves to the Half-Open state. Once in this state, the circuit resumes

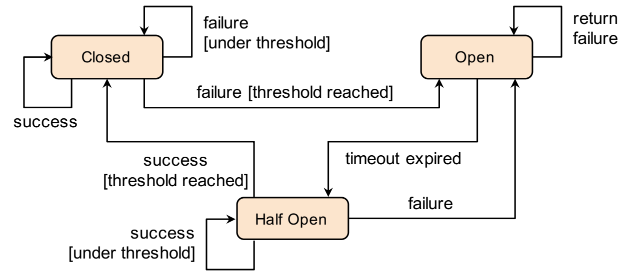


Fig. 3: Circuit breaker state machine (adapted from [16]).

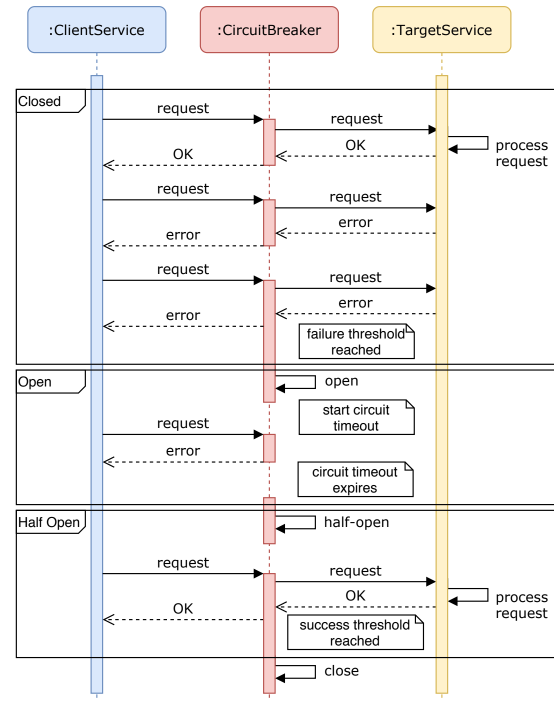


Fig. 4: *Circuit Breaker* pattern example.

forwarding incoming requests to the target service and then either moves to the Closed state, if a number of consecutive successful requests is reached, or goes back to the Open state if a specified number of requests fail.

Fig. 4 shows an example of the *Circuit Breaker* pattern used in the context of the client-service interaction scenario shown in Fig. 1. In this example, the failure threshold (used to determine when the circuit should switch from Closed to Open) is set to two consecutive requests, while the success threshold (used to determine when the circuit should switch from Half-Open back to Closed) is set to one request.

Implementation considerations: The *Circuit Breaker* pattern is customizable and can be adapted according to the type and expected duration of the possible failure. For example, one could place the circuit breaker in the Open state for a few seconds initially, and then if the failure has not been resolved to increase the timeout to a few minutes, and so on. In some cases, rather than the Open state returning failure and raising an exception, it could be useful to return a default value that

is meaningful to the application.

An application invoking an operation through a circuit breaker must be prepared to handle the exceptions raised if the operation is unavailable. The way exceptions are handled will be application-specific. For example, an application could temporarily degrade its functionality, invoke an alternative operation to try to perform the same task or obtain the same data, or report the exception to the user and ask them to try again later.

III. Retry and Circuit Breaker Pattern Models

In this section, we give a brief overview of probabilistic model checking and the CTMC formalism, and then present our CTMC models for the *Retry* and *Circuit Breaker* patterns.

A. Probabilistic Model Checking

Probabilistic model checking (PMC) [13] is a set of formal verification techniques that enable (a) modeling of systems that exhibit probabilistic behavior, and (b) the analysis of quantitative properties concerning costs/rewards (e.g., resource usage, time) and probabilities of certain events happening in the system (e.g., of reaching a state that violates a safety invariant).

In PMC, systems are modeled as state-transition systems augmented with probabilities such as discrete-time Markov chains (DTMC), Markov decision processes (MDP), and continuous-time Markov chains (CTMC). CTMC are particularly useful in performance analyses, as they allow specifying the rates of transition from one state to another. Probabilistic choice, in this type of model, arises through race conditions when two or more transitions in a state are enabled. These properties make CTMC a perfect fit to capture the behavior of microservice architectures with high time fidelity, in a natural way:

Definition 1: A labeled Continuous-Time Markov Chain (CTMC) extended with rewards is a tuple $\mathcal{C} = (S, s_i, R, L, \rho, \iota)$, where S is a finite set of states, $s_i \in S$ is the initial state, $R : S \times S \to \mathbb{R}^+$ is the transition rate matrix, $L : S \to 2^{AP}$ is a labeling function which assigns to every state $s \in S$ a set $L(s)$ of atomic propositions valid in that state, $\rho : S \to \mathbb{R}^+$ is a function that defines the rate at which reward is acquired in a state ($t \cdot \rho(s)$, with time $t \in \mathbb{R}^+, s \in S$), and $\iota : S \times S \to \mathbb{R}^+$ is a transition reward function that assigns a reward every time a transition occurs in the CTMC.

In the definition above, the transition rate matrix R determines how transitions between states (e.g., message exchange between services) are triggered in the CTMC. Concretely, the probability of a transition being triggered within t time units is equal to $1 - e^{-R(s,s')\cdot t}$ (a transition of rate $1/t$ will take on average t time units to be triggered). Moreover, the reward assignment function can be used to encode rewards and costs, e.g., to quantify the number of messages exchanged, or the time elapsed during system execution.

System properties are expressed using some form of probabilistic temporal logic, such as Continuous Stochastic Logic (CSL) or Probabilistic Computation Tree Logic (PCTL), which enable quantifying some probability or reward, or stating that they meet some threshold. In particular, CSL reward quantification properties can be employed to analyze times in a microservice architecture described as a CTMC. For instance, the class of property $\mathsf{R}^{\mathsf{r}}_{=?}[\mathsf{F}\ \phi]$ allows quantifying the reward r accrued along system paths leading to states that eventually satisfy the state formula ϕ. An example of a property employing this operator for quantifying contention time in the system might be $\mathsf{R}^{\mathsf{contention_time}}_{=?}[\mathsf{F\ end}]$, meaning "accrued contention time along paths that lead to the end of the system's execution." This property assumes a state reward function contention_time in the model, defined as $(\mathsf{under_contention}, 1)$, where under_contention is a predicate over system states capturing those in which network resources are being held. A more in-depth discussion of CTMC and their theoretical underpinning can be found in [13].

In the remainder of this paper, we illustrate our approach using the high-level syntax of the PRISM language [12] to describe CTMC. A PRISM CTMC model is built as a set of processes or *modules* (delimited by keywords module/endmodule) that are encoded as a set of commands:

$$[action]\ guard \to r_1 : u_1 + \ldots + r_n : u_n,$$

where $guard$ is a predicate over the model variables (which can be either booleans or bounded-range integers). Each update u_i describes a transition that the process can make (by executing $action$) if the guard is true. An update is specified by giving the new values of the variables and has an assigned transition rate r_i. Multiple commands with overlapping guards introduce transition *race conditions* in the model [13].

The probabilistic model corresponding to a CTMC specification is constructed as the parallel composition of its modules. In every state of the model, there is a set of actions (belonging to any of the modules) which are enabled, i.e., whose guards are satisfied in that state. The choice of which action is triggered by the model is captured as a "race" between transitions in different modules.

CTMC reward (or cost) structures containing one or more reward items can be defined using the syntax:

```
rewards reward_label
    [action] guard : reward;
endrewards
```

In transition-based reward definitions, $reward$ is accrued under $reward_label$ for every transition that executes $action$ from a state that satisfies $guard$. State rewards are specified using a similar syntax, but removing the $[action]$ component of the definition.

Fig. 5 shows an example of a CTMC model specified in the PRISM language, represented both textually (left) and graphically (right), for a simplified version of the client-service interaction scenario shown in Fig. 1, in which the target service does not fail.[1] This model is specified in two modules:

[1] In the remainder of the paper, we will show only CTMC models in their graphical representations, as these are more compact and easier to explain.

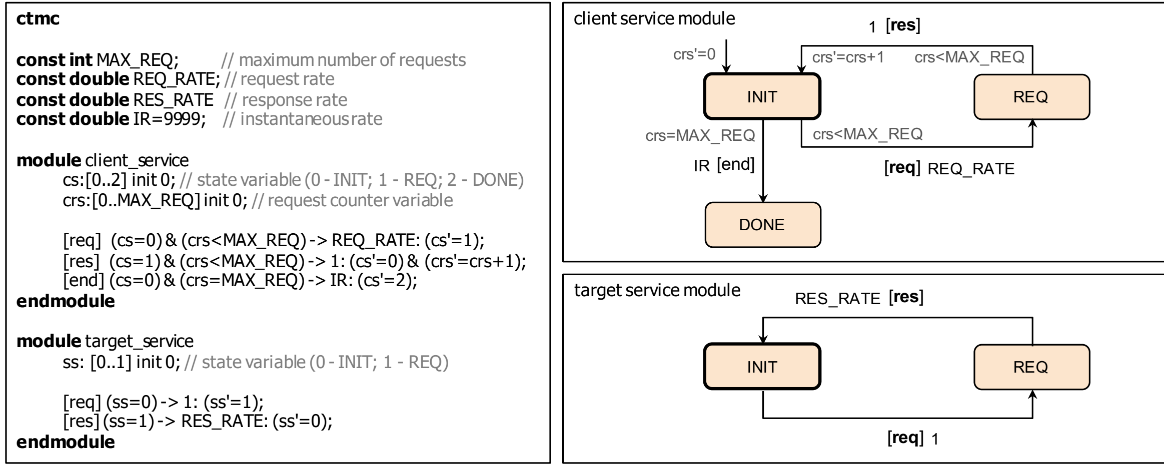


Fig. 5: A simple CTMC model in the PRISM language: textual representation (left) and graphical representation (right).

one captures the behavior of the client service and the other captures the behavior of the target service.

Both modules start in their respective INIT states. The client service module uses a local variable (crs) to count the total number of requests served by the target service, which is initially set to zero. Whenever the client service is in the INIT state and the request count is less than the maximum number of requests allowed (MAX_REQ), the shared action req is enabled and both modules move to their respective REQ states with rate REQ_RATE (note that action req is specified with rate 1 in the target service module, which means that the rate of that transition is always dominated by the corresponding rate specified in the client service module).

Once the modules are in their respective REQ states, the shared action res is enabled and both modules return to their INIT states with rate RES_RATE, which is specified in the target service module. Finally, when the client service module is in the INIT state and the maximum number of requests is reached, the action end is enabled and that module moves to the DONE state with an instantaneous rate IR.[2]

A CTMC reward for computing the expected total execution time of the client service module can be specified as follows:

$$\begin{array}{l}\textsf{rewards}\ total_time \\ \qquad true : 1; \\ \textsf{endrewards}\end{array}$$

The above specification instructs PRISM to accrue reward at rate 1 in every state of the model, enabling quantification of total execution time.

B. Basic ClientService-Proxy-TargetService Model

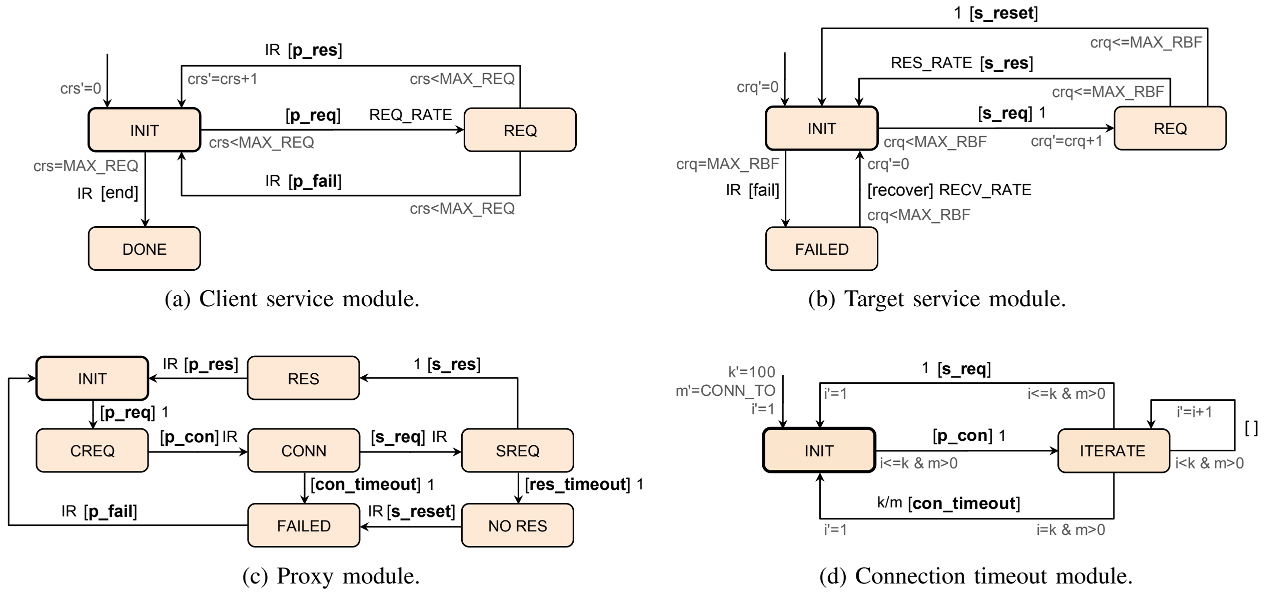
To make the *Retry* and *Circuit Breaker* CTMC models easier to describe, we have decoupled the specification of the client service from that of the target service using a proxy as a mediator. The proxy encapsulates the invocation mechanism used by the client service to send requests to the target service and to receive its corresponding responses. This mechanism is also responsible for handling possible failures of the target service, which are detected by means of either a connection timeout or a response timeout. The proxy abstracts the behavior of the invocation mechanism from both the client service and the target service, facilitating their composition with different implementations of the invocation mechanism (e.g., with or without the *Retry* and *Circuit Breaker* patterns).

Fig. 6 shows the CTMC modules for the client service, the target service, and a basic version (i.e., with no resiliency strategy) of the proxy. The client service module (Fig. 6a) is similar to the module of the same name shown in Fig. 5: the only difference is the renaming of the request (p_req) and response (p_res) actions, which are now shared with the proxy module, and the inclusion of a new action (p_fail), which is triggered when the proxy fails to either, connect to, or to get a response from the target service.

The target service module (Fig. 6b) has been extended with the inclusion of a new state, FAILED, which captures the failure of the target service, and of three new actions: fail, which is triggered whenever the number of requests received by the target service reaches a threshold (MAX_RBF); recover, which is triggered whenever the target service recovers from a failure with a recovery rate (RECV_RATE); and s_reset, which is shared with the proxy and is triggered to force the target service to return to its initial state whenever the proxy detects a response timeout.

The proxy module (Fig. 6c) captures the behavior of the proxy, receiving a request from the client service (CREQ state), connecting to and getting a response from the target service (CONN and SREQ states, respectively), and, finally, forwarding the response to the client service (RES state). This module also captures the two cases in which the proxy must handle possible failures of the target service. The first case is captured by the con_timeout action, which is enabled in the CONN state and triggered after a fixed amount of time (CONN_TO) whenever the target service is in the FAILED state. The second case is captured by the res_timeout action,

[2]In PRISM, an instantaneous transition can be encoded in a CTMC by using a large value for its rate.



(a) Client service module.

(b) Target service module.

(c) Proxy module.

(d) Connection timeout module.

Fig. 6: ClientService-Proxy-TargetService CTMC model.

which is enabled in the SREQ state and triggered whenever the target service takes longer than a fixed amount of time (RES_TO) to serve a request it has received from the proxy. In the latter case, the proxy has to force the target service to return to its initial state by triggering the s_reset action.

To model fixed time delays in a CTMC, which are not currently supported by PRISM, but are required for capturing the connection and response timeout behavior of the proxy, we use an approximation strategy suggested in the PRISM FAQ [17]. This strategy is based on the generation of iterative transitions following an Erlang distribution with shape k (corresponding to the number of transitions to be generated) and mean m (corresponding to the expected fixed delay to be modeled). The greater the value of k, the better the fixed delay approximation, but the larger the size of the resulting model. In our work, we have set $k = 100$, which establishes a reasonable trade-off between fixed delay accuracy and model growth [17]. The CTMC module capturing the connection timeout delay (CONN_TO) and action (con_timeout) used by the proxy module is shown in Fig. 6d. In that module, shared actions p_con (triggered when the proxy attempts to connect to the target service) and s_req (triggered when the proxy successfully connects to the target service) are used to start and interrupt the timeout iterations, respectively. The response timeout behavior of the proxy (not shown here) is captured in an analogous way.

C. Retry Pattern Model

The *Retry* pattern is modeled as a direct extension of the proxy module shown in Fig. 6c. The key idea is to specify its behavior in a way that can play the role of the client service from the perspective of the proxy, and the role of the proxy from the perspective of the client service. To allow the parallel composition of the new retry module (shown in

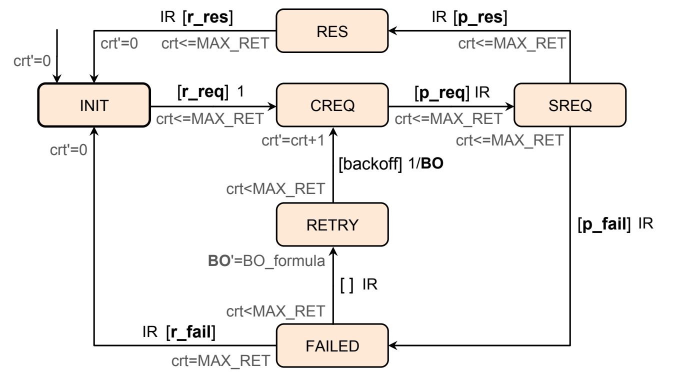


Fig. 7: *Retry* pattern CTMC model.

Fig. 7) with the client service module and the proxy module presented in the previous subsection, the three actions that the client service module shares with the proxy were renamed to r_req, r_res and r_fail, so that those actions are now shared with the retry module. The retry module shares with the proxy module actions p_req, p_res and p_fail (which were previously shared by the proxy and the client service module).

In essence, the retry module captures the loop behavior of the *Retry* pattern, as explained in Section II-B, in which a failed request (FAILED state) is retried (RETRY state) until the request succeeds (RES state), or a retry threshold (MAX_RET) is reached. In the latter case, a failure action (r_fail) is immediately triggered to notify the client service of the failure and to return the retry module to its initial state. Before each new retry, the retry module updates the *backoff delay* (BO) according to a *backoff formula* (BO_formula), and then triggers a backoff action with a rate computed as the inverse of the backoff delay. This causes the module to wait

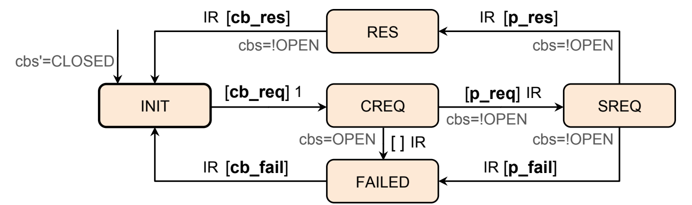
for a variable amount of time determined by an exponential distribution with the backoff delay as its mean [12]. Therefore, by providing the CTMC model with different formulas for computing the backoff delay, one can configure the retry module to behave differently at each retry iteration, e.g., using linear or exponential backoff increments.

D. Circuit Breaker Pattern Model

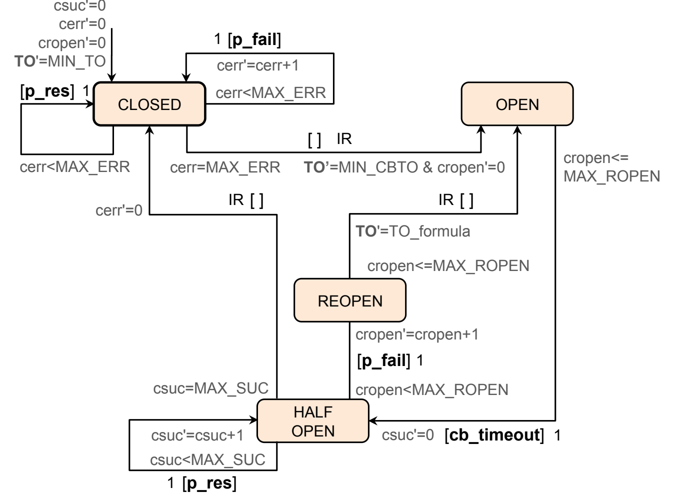
The *Circuit Breaker* pattern model (shown in Fig. 8) is also specified as a direct extension of the proxy module shown in Fig. 6c. This is done by renaming the set of actions that the client service module originally shared with the proxy module to cb_req, cb_res and cb_fail, and by having the circuit Breaker module share the same set of actions with the proxy module, as previously explained for the *Retry* pattern model.

The circuit breaker module (Fig. 8a) only forwards requests from the client service to the proxy when it is not in the OPEN state; otherwise, the circuit immediately moves to the FAILED state, which triggers a cb_fail action to notify the client service of the failure and to make the circuit return to its initial state. The circuit breaker state machine module (Fig. 8b) captures a dynamic version of the pattern described in Section II-C, where the timeout used to half-open the circuit is reset to a given minimum value every time the circuit opens (i.e., moves from CLOSED to OPEN), and increases exponentially until a given maximum value when the circuit is continuously reopened (i.e., moves from REOPEN to OPEN).

The circuit breaker timeout module (not shown here due to space constraints) is specified similarly to the connection timeout module shown in Fig. 6d. However, the former module has no "interrupt" action, as the circuit has to remain open for the duration of the timeout. The circuit breaker timeout value is given in the global variable TO. This variable is exponentially increased by the circuit breaker state machine module using a timeout formula (TO_formula), which is a function of the value of the reopen counter (cropen).



(a) Circuit breaker module.



(b) Circuit breaker state machine module.

Fig. 8: *Circuit Breaker* pattern CTMC model.

IV. Model Analysis

In this section, we report on the objectives, method, and results of an analysis of the CTMC models presented in Section III, using the PRISM model checker.

A. Objectives

We analyzed the behavior of four different CTMC models, each capturing a different version of the client-service scenario depicted in Fig. 1, in which a client service has to complete a required number of successful invocations of a target service. The *Simple Proxy* (SP) model captures the scenario where the client service invokes the target service using only the proxy, i.e., with no resiliency pattern implemented. The *Retry Pattern* (RP) model captures the scenario where the client service invokes the target service using the *Retry* pattern. Finally, the *Static Circuit Breaker* (SCB) and *Dynamic Circuit Breaker* (DCB) models capture two variations of the scenario where the client service invokes the target service using the *Circuit Breaker* pattern: in the static variation, the circuit breaker timeout is statically defined as one of the parameters of the model, whereas in the the dynamic variation the circuit break timeout is exponentially increased when the circuit is reopened, as explained in Section II-C.

Our analysis was meant to shed some light on the following research questions:

- RQ1 What is the impact of using different SP model configurations on the client service's quality attributes?
- RQ2 Compared to the best SP model configuration, what is the impact of using different RP model configurations on the client service's quality attributes?
- RQ3 Compared to the best SP and RP model configurations, what is the impact of using different SCB and DCB model configurations on the client service's quality attributes?

B. Method

All four CTMC models were analyzed with respect to the same quality attributes: namely *Expected Total Time*, computed as the total time the client service needs to complete the required number of successful invocations of the target service, and *Expected Contention Time*, computed as the total time the client service spends holding network resources, either by attempting to connect to, or by waiting to get a response from the target service. For the analysis with PRISM, these quality attributes were calculated using two CTMC reward structures,

TABLE I: Model Analysis Parameters

Model	Parameter	Description
SP	MAX_REQ=100	Required number of successful requests
	IR=9999	Instantaneous rate
	MIRT=1	Mean inter-request time
	REQ_RATE=1/MIRT	Request rate
	MRT=5	Mean response time
	RES_RATE=1/MRT	Response rate
	MAX_RBF=10	Maximum number of requests between failures
	MTBF=MAX_RBF*MRT	Mean time between failures
	AV=[0.5,0.6,0.7,0.8,0.9,1.0]	Target service availability
	MTTR=((1/AV)-1)*MTBF	Mean time to repair
	RECV_RATE=MTTR=0?IR:1/MTTR	Mean recovery rate (MRR)
	CONN_TO=[2,10,20,40]	Connection timeout (CTO)
	RES_TO=2*MRT	Response timeout
RP	MAX_RET=5	Maximum number of retries per request
	MIN_BO=5	Minimum mean backoff delay
	MAX_BO=[10,20,40]	Maximum mean backoff delay (MBO)
	BO_formula=min(MAX_BO, MIN_BO*2^crt)	Exponential backoff formula
SCB & DCB	MAX_ERR=2	Maximum number of failed requests before opening the CB
	MAX_SUC=2	Maximum number of successful requests before closing the CB
	MIN_TO=[5,10,20,40]	Minimum CB timeout
	MAX_TO=[10,20,40]	Maximum CB timeout (CBTO)
DCB (only)	TO_formula=min(MAX_TO, MIN_TO*2^cropen)	Exponential CB timeout formula

one for each attribute. The expected total time reward was specified as shown in Section III-A. The expected contention time reward was specified as follows:

$$\begin{array}{l}\textsf{rewards}\ contention_time\\ \quad (ps = CONN) : 1;\\ \quad (ps = SREQ) : 1;\\ \textsf{endrewards}\end{array}$$

The above reward accrues a reward at rate 1 in states where the proxy module holds network resources waiting for the target service: i.e, whenever that module is in either the CONN state or the SREQ state. We use PRISM to quantify these two rewards in our four CTMC models by checking the following two properties, respectively:

$$\mathsf{R}^{\mathsf{total_time}}_{=?}[\mathsf{F}\ \mathsf{crs} = \mathsf{MAX_REQ}]$$

and

$$\mathsf{R}^{\mathsf{contention_time}}_{=?}[\mathsf{F}\ \mathsf{crs} = \mathsf{MAX_REQ}]$$

Both properties use the same predicate defined over the client service module's successful request counter as their reachability state predicate.

In an ideal scenario, the client service would complete the required number of successful invocations as quickly as possible without holding network resources for too long. However, the client service's quality attributes might be affected by the possibility of the target service temporarily being unavailable or taking too long to respond, as well as by the different invocation strategies the client service may use to handle those types of failure. For instance, if the client service simply retries all failed requests without any backoff strategy, it may reduce its total execution time while increasing total contention time. If, in contrast, the client service backs off for too long waiting for the target service to recover every time a request fails, it may reduce its total contention time while increasing its total execution time. In that respect, the use of (different configurations of) the *Retry* and *Circuit Breaker* patterns may result in different quality trade-offs for the client service when compared to a naive invocation strategy (i.e., one in which all failed requests are immediately retried with no backoff).

Table I shows the set of CTMC parameters used in our analysis. Parameters that are assigned multiple values, e.g., the SP model's connection timeout (CTO) and the RP model's maximum mean backoff delay (MBO), were used as varying model configuration parameters during the analysis, while the remaining single-value parameters were treated as constants. All time-dependent model parameters and analysis results are represented in CTMC time units.

For each CTMC model we considered six availability values for the target service, as prescribed for the availability (AV) parameter in Table I.

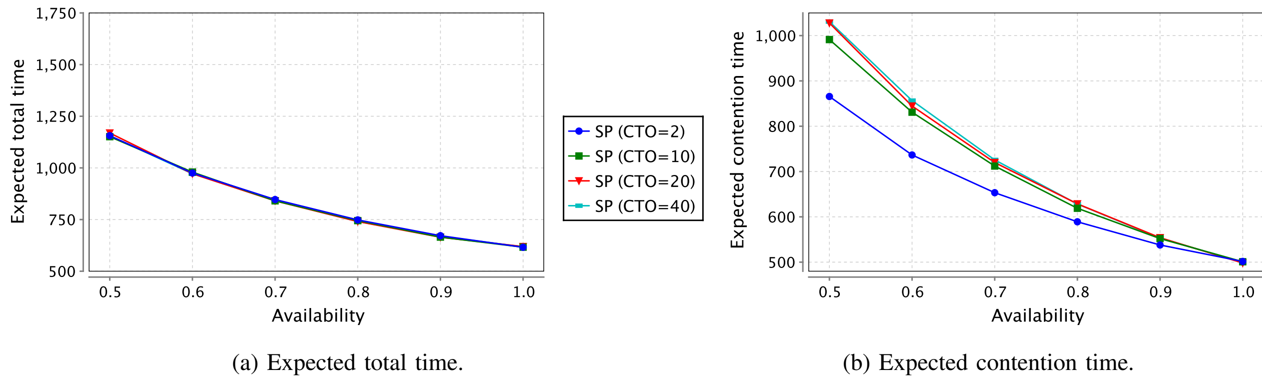
Given an availability value AV ($0 \leq AV \leq 1$), the mean recovery rate MRR of the target service is given by:

$$MRR = \begin{cases} \dfrac{AV}{MTBF * (1 - AV)} & \text{for } 0 \leq AV < 1; \\ IR & \text{for } AV = 1; \end{cases} \quad (1)$$

where $MTBF$ is the target service's mean time between failures [18] and IR is the instantaneous rate.

C. Results

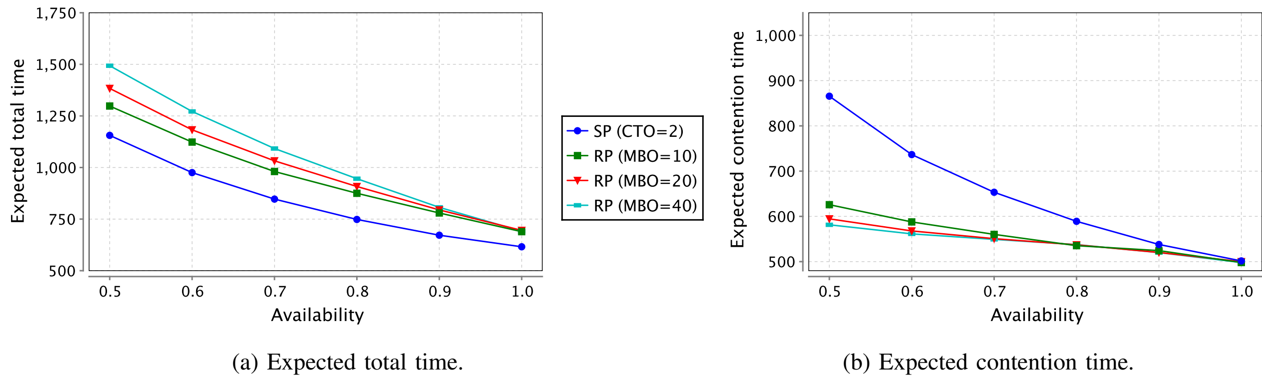
Fig. 9 shows the results of all configurations analyzed for the SP model. Those results, which answer RQ1, reveal that increasing the connection timeout (CTO) has no negative effect on the expected total execution time of the client service, independently of the target service's availability. The reason is that in the SP model the client service is always waiting for the target service to respond, which means that it will get a successful response from the target service as soon



(a) Expected total time.

(b) Expected contention time.

Fig. 9: Results of all configurations of the SP model.



(a) Expected total time.

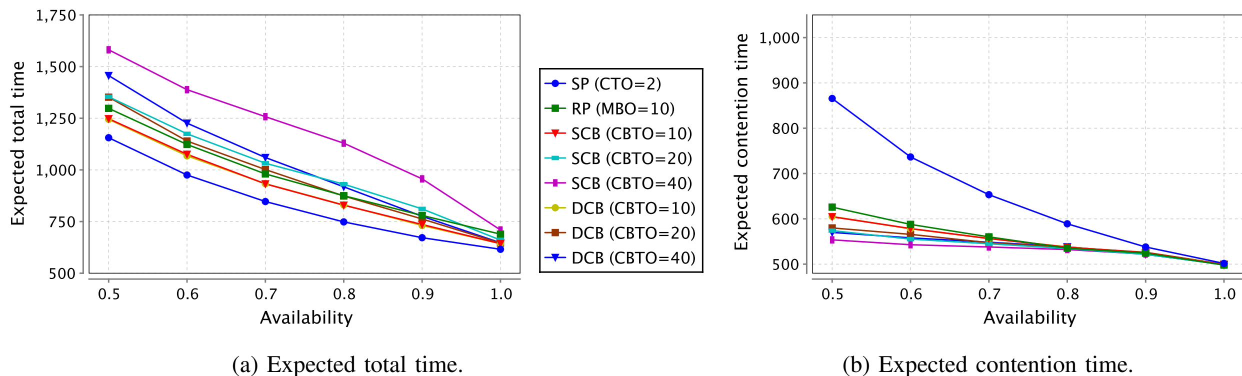
(b) Expected contention time.

Fig. 10: Results of all configurations of the RP model and the best configuration of the SP model.

as a response is available. In contrast, the client service's expected contention time is clearly affected by larger CTO values, especially when the target service has low availability. This result is justified because a larger CTO forces the client service to hold network resources for a longer period when the target service is not responsive. Since all SP model configurations produced similar results with respect to the expected total execution time, and configurations with a larger CTO negatively affect the expected contention time, we select the configuration with the lowest CTO (i.e., CTO=2) as the best SP model configuration overall.

Fig. 10 shows the results of all configurations analyzed for the RP model, along with the results of the best SP model configuration. The results, which answer RQ2, show that using an exponentially increasing backoff delay, as prescribed by the *Retry* pattern, directly affects the client service's expected total execution time when the target service has low availability. This happens because the retry mechanism forces the client service to wait increasingly longer for the target service to recover after a sequence of failed requests. However, the retry mechanism causes a noticeable reduction (over 30%) in the client service's expected contention time compared to the best SP model configuration, as the client service no longer retries every failed request. While the different values analyzed for the retry mechanism's maximum backoff delay (MBO) parameter had a very similar impact on the expected contention time, higher MBO values had a higher impact on the expected total time. This means that increasing the retry mechanism's maximum backoff delay may not be a good invocation strategy because this is likely to have a large impact on the client service's total execution time without a significant gain in its expected contention time. For this reason, we select the configuration with the lowest MBO (i.e., MBO=10) as the best RP model configuration overall.

Finally, Fig. 11 shows the results of all configurations analyzed for the SCB and DCB models, along with the results of the best configurations of the SP and RP models. Those results, which answer RQ3, convey a number of interesting findings. First, the SCB model configuration with the largest circuit breaker timeout (CBTO) value (i.e., CBTO=40) has by far the worst impact on the client service's expected total time, while producing only a minor gain in terms of its expected contention time, as compared to the other SCB and DCB model configurations. That configuration is followed by the DCB and SCB model configurations with CBTO=40 and CBTO=20, respectively, as the worst configurations for the expected total time. The RP model configuration with MBO=10 and the DCB model configuration with CBTO=20 are next with similar expected total time results. The best SCB and DCB model configurations in that respect are those with the lowest CBTO values (i.e., CBTO=10), which produced only a minor increase (less than 5%) in the client service's



(a) Expected total time.

(b) Expected contention time.

Fig. 11: Results of all configurations of the SCB and DCB models and the best configurations of the SP and RP models.

expected total time compared to the SP model configuration with CTO=2. In terms of the expected contention time, the SCB and DCB models configurations, as well as the best RP model configuration, had comparable results, all being significantly better than the best SP model configuration under low to moderate availability values.

Based on the above results, we single out the SCB and DCB model configurations with the lowest CBTO, followed by the RP model configuration with the lowest MBO, as the service invocation strategies that offer the best trade-offs in terms of the expected total time and the expected contention time quality attributes, under varying availability conditions, over all models analyzed.

V. Related Work

There is an extensive body of research and development in the areas of system reliability engineering [1], [18], software reliability modeling and prediction [19]–[23], and software architecture analysis [24]. Although early work in these areas has not been specifically developed targeting modern microservice architectures, many of the proposed ideas are still relevant. For example, the automated translation of failure-retry blocks in the Palladio Component Model [21] offers a similar solution to the *Retry* pattern.

Despite the recent popularity of microservices, only a few research works have thus far studied microservice resiliency, in general, and resiliency patterns, in particular. Montesi and Weber [10] discuss the use and implementation of several *microservice design patterns* [25] in the context of the Jolie microservice language [26], including three variants of the Circuit Breaker pattern. Heorhiadi *et al.* [27] present a systematic resiliency-testing framework that can be used to capture high-level failure scenarios in microservice-based applications. Preuveneers and Joosen [11] describe a Circuit Breaker framework enhanced with the notion of Quality of Context to improve the resiliency of context-aware distributed applications. Finally, Yin *et al.* [28] present a Microservice Resilience Measurement Model (MRMM) and framework, which can be used to elicit resiliency requirements and to quantify resilience metrics in a microservice-based system.

Our work differs from these works in that we apply probabilistic model checking to analyze the behavior of different configurations of the *Retry* and *Circuit Breaker* patterns, in terms of different quality attributes, under varying availability conditions. Even though our probabilistic models are only simplified representations of the microservice interaction scenarios typically found in real-world production environments [29], [30], we believe that, by capturing the behavior of critical resiliency design concepts at an appropriate level of abstraction such a model-based approach can be a promising companion to performing microservice resiliency tests in production [31].

VI. Conclusion

There is a growing interest in applying resiliency patterns to improve the reliability of microservice-based distributed systems. This paper presented a model-based analysis of two popular resiliency patterns—namely *Retry* and *Circuit Breaker*—in the context of a simple client-service interaction scenario, using the PRISM model checker. Overall, the analysis results show that *properly configured*, both patterns can significantly reduce resource contention at the client-side compared to a naive approach of continuously retrying failed requests, with only a moderate increase in execution time. We made our models publicly available[3] to facilitate the replication of our study and to promote further research in this area.

This paper is admittedly an early step towards a more systematic evaluation of the resiliency of microservice-based applications. The main limitation of our work is that our models capture only the behavior of a single client service interacting with a single target service, and they ignore possible network failures and delays. This was a deliberate decision to avoid cluttering the exposition, and to tame state explosion during model checking. Moreover, our analysis results have not been validated empirically. Finally, we left out of the analysis other popular resiliency patterns that rely on asynchronous communication, e.g., *Event Sourcing* [32] and *Queue-Based Load Leveling* [33]. We plan to address these limitations in future work.

[3]https://github.com/ppgia-unifor/resiliency-patterns

References

[1] B. Beyer *et al.*, *Site Reliability Engineering: How Google Runs Production Systems*. O'Reilly, 2016.

[2] B. Lindsay, "Designing for failure may be the key to success—interview by Steve Bourne," *ACM Queue*, vol. 2, no. 8, 2004.

[3] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term," https://martinfowler.com/articles/microservices.html, 2014, [Online; last access on February 25, 2020].

[4] P. Jamshidi *et al.*, "Microservices: The journey so far and challenges ahead," *IEEE Software*, vol. 35, no. 3, pp. 24–35, 2018.

[5] Microsoft Azure, "Resiliency patterns," https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency, 2017, [Online; last accessed on February 25, 2020].

[6] Netflix, "Hystrix: Latency and Fault Tolerance for Distributed Systems," https://github.com/Netflix/Hystrix, [Online; last accessed on February 25, 2020].

[7] Twitter, "Finagle: A fault tolerant, protocol-agnostic RPC system," https://github.com/twitter/finagle, [Online; last accessed on February 25, 2020].

[8] Resilience4j, "Resilience4j: A Fault tolerance library designed for functional programming," https://github.com/resilience4j/resilience4j, [Online; last accessed on February 25, 2020].

[9] Microsoft Azure, "Retry Pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/retry, 2017, [Online; last accessed on February 25, 2020].

[10] F. Montesi and J. Weber, "Circuit breakers, discovery, and api gateways in microservices," *arXiv preprint arXiv:1609.05830*, 2016.

[11] D. Preuveneers and W. Joosen, "QoC^2 Breaker: intelligent software circuit breakers for fault-tolerant distributed context-aware applications," *Journal of Reliable Intelligent Environments*, vol. 3, no. 1, pp. 5–20, 2017.

[12] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, ser. LNCS, G. Gopalakrishnan and S. Qadeer, Eds., vol. 6806. Springer, 2011, pp. 585–591.

[13] ——, "Stochastic Model Checking," in *Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM'07)*, ser. LNCS (Tutorial Volume), M. Bernardo and J. Hillston, Eds., vol. 4486. Springer, 2007, pp. 220–270.

[14] M. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2007.

[15] Istio.io, "Istio: Connect, secure, control, and observe services," https://istio.io/, [Online; last accessed on February 25, 2020].

[16] M. Fowler, "CircuitBreaker," https://martinfowler.com/bliki/CircuitBreaker.html, 2014, [Online; last access on February 25, 2020].

[17] Prism FAQ, "How can I add deterministic time delays to a CTMC model?" http://www.prismmodelchecker.org/manual/FrequentlyAskedQuestions/PRISMModelling#det_delay, 2010, [Online; last accessed on February 25, 2020].

[18] A. Birolini, *Reliability Engineering: Theory and Practice*. Springer Science & Business Media, 2013.

[19] V. S. Sharma and K. S. Trivedi, "Reliability and performance of component based software systems with restarts, retries, reboots and repairs," in *2006 17th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2006, pp. 299–310.

[20] ——, "Quantifying software performance, reliability and security: An architecture-based approach," *Journal of Systems and Software*, vol. 80, no. 4, pp. 493–509, 2007.

[21] F. Brosch, B. Buhnova, H. Koziolek, and R. Reussner, "Reliability prediction for fault-tolerant software architectures," in *Proceedings of the joint ACM SIGSOFT Conference and ACM SIGSOFT Symposium on Quality of Software Architectures (QoSA) and Architecting Critical Systems (ISARCS)*, 2011, pp. 75–84.

[22] F. Brosch, H. Koziolek, B. Buhnova, and R. Reussner, "Architecture-based reliability prediction with the palladio component model," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1319–1339, 2011.

[23] R. Mirandola, P. Potena, E. Riccobene, and P. Scandurra, "A reliability model for service component architectures," *Journal of Systems and Software*, vol. 89, pp. 109–127, 2014.

[24] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on software Engineering*, vol. 28, no. 7, pp. 638–653, 2002.

[25] D. Taibi, V. Lenarduzzi, and C. Pahl, "Architectural patterns for microservices: a systematic mapping study," in *8th Int. Conf. Cloud Computing and Services Science (CLOSER)*, 2018.

[26] Jolie Language, "Jolie: The first language for Microservices," https://www.jolie-lang.org/, [Online; last accessed on February 25, 2020].

[27] V. Heorhiadi, S. Rajagopalan, H. Jamjoom, M. K. Reiter, and V. Sekar, "Gremlin: Systematic resilience testing of microservices," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, 2016, pp. 57–66.

[28] K. Yin, Q. Du, W. Wang, J. Qiu, and J. Xu, "On representing and eliciting resilience requirements of microservice architecture systems," *arXiv preprint arXiv:1909.13096*, 2019.

[29] The Netflix Tech Blog, "Fault Tolerance in a High Volume, Distributed System," https://medium.com/netflix-techblog/fault-tolerance-in-a-high-volume-distributed-system-91ab4faae74a, 2012, [Online; last accessed on February 25, 2020].

[30] B. Ibryam, "It takes more than a Circuit Breaker to create a resilient application," https://developers.redhat.com/blog/2017/05/16/it-takes-more-than-a-circuit-breaker-to-create-a-resilient-application/, 2017, [Online; last accessed on February 25, 2020].

[31] A. Schaffer, "Testing of Microservices," https://labs.spotify.com/2018/01/11/testing-of-microservices/, 2018, [Online; last accessed on February 25, 2020].

[32] M. Fowler, "Event Sourcing," https://martinfowler.com/eaaDev/EventSourcing.html, 2005, [Online; last access on February 25, 2020].

[33] Microsoft Azure, "Queue-Based Load Leveling Pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/queue-based-load-leveling, 2019, [Online; last accessed on February 25, 2020].