2022 IEEE 19th International Conference on Software Architecture (ICSA) | 978-1-6654-1728-0/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSA53651.2022.00010

# Using Gossip Enabled Distributed Circuit Breaking for Improving Resiliency of Distributed Systems

Aashay Palliwar
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar
Bhubaneswar, India
avp10@iitbbs.ac.in

Srinivas Pinisetty
School of Electrical Sciences
Indian Institute of Technology Bhubaneswar
Bhubaneswar, India
spinisetty@iitbbs.ac.in

***Abstract*—Distributed systems are rife with failures. Several resiliency patterns are used to improve the ability of these systems to tolerate faults and maintain functionality. The circuit breaker pattern is a popular resiliency pattern that is especially suitable for the case when the faults causing dependency failures take a variable amount of time to resolve. In this paper, we propose a modification to the traditional circuit breaker pattern. We also propose a gossip-based information dissemination protocol that enables the (modified) circuit breakers deployed on multiple client-service instances to take a concerted and more informed decision when a common dependency is facing persistent failures. We formally model the client-server systems that use traditional and the proposed distributed circuit breaker patterns in UPPAAL to analyze and compare their performance. The statistical model checking queries performed on the models show that, as compared to the traditional circuit breaker pattern, the distributed version results in fewer unsuccessful requests, that consume system/network resources, with practically the same total execution time under various availability conditions - only at the famously low cost of a robust gossip-based information dissemination protocol.**

***Index Terms*—Circuit Breaker, Formal Modeling and Analysis, Gossip Protocol, Microservices, Resiliency Patterns, Statistical Model Checking, Timed Automata, UPPAAL**

## I. Introduction

Systems using architectural paradigms like service-oriented architecture [1] and microservice architecture [2] have one idea in common. These distributed systems are composed of loosely coupled reusable software components that communicate and coordinate over a network to serve a request or accomplish a task. These components are often separate processes and are deployed on different machines.

Distributed systems are susceptible to a number of failures. The individual nodes in a distributed environment may fail due to software or hardware faults. Communication among nodes may fail due to network partitions or congestion. A node may become unavailable to other nodes due to a high load and lack of resources to handle new requests. Requests or actions that require coordination of several nodes might fail if some nodes are experiencing such failures. Some failures are self-correcting - a failed action may succeed on subsequent retries. Some failures might require a variable amount of time to resolve (with or without human intervention).

Another problematic scenario is that of a cascading failure down the dependency chain. Whenever a service instance/node receives a request, it allocates certain system and/or network resources for fulfilling that request. If the rate of arrival of requests ($R_1$) is less than the rate of handling requests ($R_2$) (and hence less than the rate of releasing the held up resources), the node stays stable. However, if $R_1$ becomes greater than $R_2$, the resources would be used up quickly and the node may become unresponsive or unavailable. This is a scenario where the node becomes unstable. Engineers deploying a system take into account relevant estimation of these factors during the capacity planning of the system and allocate resources accordingly. In case when a service on which the node is dependent becomes unavailable or faces persistent failures, the rate $R_2$ may decrease suddenly and render the node unstable and eventually unavailable. This might start a cascade of failures down the dependency chain if relevant measures are not taken to detect a dependency unavailability and remedy the situation.

A system's ability to handle failures and recover from them constitutes its resiliency. A resiliency pattern [3] is an architectural pattern for service-service interaction that helps prevent cascading failures and maintain functionality in case of dependency failures. There are several resiliency patterns, e.g. bulkhead pattern [4], retry pattern [5], circuit breaker pattern (CB) [6] etc. A caller process using the bulkhead pattern isolates resources for each service dependency so that failure of one dependency does not cause requests to pile up and use up critical resources. If the resources are used up, it might prevent calls to other service dependencies from succeeding. Software components employing the retry pattern transparently retry failed requests a number of times as per the retry policy. If the request still fails after multiple retries, the request is declared to be failed. The retry pattern works well only for failures caused due to transient faults. These faults are often self-correcting, and if the action is repeated after a suitable delay, the action is likely to succeed. For failures caused due to faults that take a variably large amount of time to get resolved, the circuit breaker pattern is more suitable.

M. Nygard popularized the circuit breaker pattern through [7]. Electrical circuit breakers detect a surge of current and save an electrical system from a current it cannot handle. In a similar way, for a client-server system, a circuit breaker module uses the present success-failure statistics to infer a

978-1-6654-1728-0/22/$31.00 ©2022 IEEE
DOI 10.1109/ICSA53651.2022.00010

probable server failure and restricts the flow of requests to the server for a predefined time interval as the server might not be able handle the requests at the moment. This saves an already suffering server from additional load and also prevents client resources from being held up unnecessarily for making requests that are very likely to fail.

The circuit breaker module may be implemented on the client-side, on a proxy or on the server-side. The disadvantage of using a server-side or a proxy circuit breaker is that network partitioning or proxy/server failure may render the circuit breaker module unavailable to the clients. The clients would not have any advantage of circuit breaking whatsoever during such failures. Several production-grade solutions like [8], [9], [10] provide libraries with circuit breaker implementations that may directly be used on the client-side.

A simple circuit breaker infers a server unavailability when a failure threshold is crossed. If there are multiple client instances that use client-side circuit breaker solutions for interacting with the same server, each client has to face failures beyond the set threshold for their circuit breakers to infer a server unavailability. Resiliency may be improved if the nodes can take a more informed and early decision about restricting the request flow based on each other's success/failure statistics instead of trying to infer dependency unavailability individually.

As an extension to the Polly Project [9], D. Reisenberger provided a proof of concept [11] for a circuit breaker that utilizes the knowledge of multiple clients to improve the resiliency. It provided a single implementation which uses the durability and reliability guarantees of Azure Entity Functions [12]. This implementation accumulates and stores success/failure statistics across calls made by all instances consuming the common service. In essence, this work provides a scheme where multiple clients may use the same circuit breaker. However, the idea is susceptible to network partitioning. It also adds a latency overhead as a remote call is necessary for checking the circuit breaker state every time a dependency call is to be made. The clients also have to know in advance the identity of the circuit breaker group to take advantage of the group knowledge.

In this paper we provide a modified version of the traditionally used circuit breaker pattern which, coupled with the proposed gossip-based protocol, provides a distributed circuit breaking scheme that is resilient to network partitioning and to the changes in the set of clients interacting with a common dependency. The gossip-based protocol facilitates self-organization of clients into groups that may share each others' knowledge for a faster circuit breaking. The proposed scheme does not add any overhead to the remote calls. The results obtained from the model-based statistical analysis show that as compared to the traditional client-side circuit breakers, the proposed distributed version results in fewer unsuccessful requests, that consume system/network resources, with practically the same total execution time under various availability conditions. To the best of our knowledge, this is the first work proposing a distributed circuit breaking scheme enabled by a gossip-based information dissemination protocol.

The rest of the paper is organized as follows: Section II describes the functioning and the philosophy behind working of the circuit breaker pattern. In section III we introduce the notion of distributed circuit breaking and discuss how it may be implemented. Section IV specifies the gossip-based protocol that is at the core of the proposed implementation for distributed circuit breaking. In Sections V and VI, we describe how we formally modeled the systems and measured the efficacy of our proposals. We discuss the trade-offs to be considered while selecting the parameter values for the implementation of the proposed pattern in section VII and then conclude in section VIII.

## II. Circuit Breaker Pattern

In this section, we first describe the functioning of a simple circuit breaker (CB) module and then explain the philosophy behind the working of the resiliency pattern.

### A. Functioning of a Circuit Breaker Module

The client instance that uses the circuit breaker pattern includes a module that tracks the success and failure status of requests which it makes to a server instance (dependency). Several policies may be defined for the working of CB which might differ in the way the failure metric is calculated. Some policies may involve varying the parameters in response to certain events. Figure 1 shows the state machine for a simple circuit breaker with parameters mentioned in Table I.

A client starts with its circuit breaker in the closed state. The client-server (or caller-callee) interaction is unhindered when the circuit breaker is in the closed state. We may constitute request timeouts as failures for the sake of explanation. If F timeouts are encountered in the last WS requests, and F is greater than the threshold FT, the circuit breaker moves to the open state. No requests are forwarded to the server (dependency) when the circuit breaker is in the open state. Instead, responses with appropriate messages are returned immediately. After the time duration OD, the circuit breaker moves to the half-open state. In this state, a few requests from the set of pending requests are made to the server. If the number of successful requests S goes beyond a threshold HOST, the circuit breaker moves to the closed state, and the client-server interaction becomes normal again. If the failure threshold HOFT is crossed while in the half-open state, the circuit breaker transitions to the open state.

### B. Philosophy Behind the Working of the CB Pattern

When the number of failures in the closed state crosses the empirically decided failure threshold FT, it indicates some problem with the server instance that might have rendered it unavailable. A failure threshold that is greater than 1 may be used so as to gain sufficient confidence that the fault causing the dependency unavailability is not transient in nature.

Not forwarding any requests when the circuit breaker is in the open state will help reduce the stress on an already suffering server. On the other hand, returning an immediate

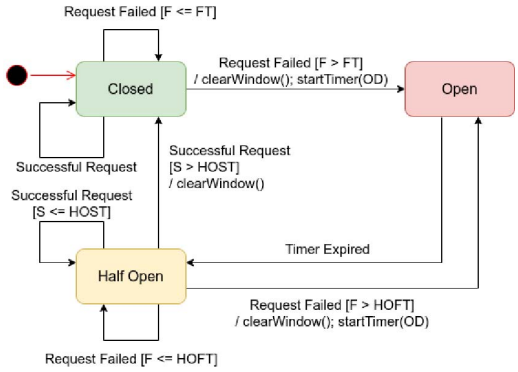



Fig. 1: State machine for a simple circuit breaker module

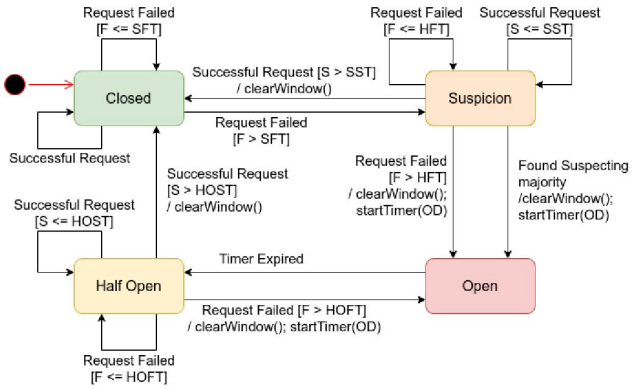



Fig. 2: State machine for a Distributed Circuit Breaking Node

failure response would not tie up the client resources for making requests that will probably fail.

After the predefined delay OD, the circuit breaker moves to the half-open state and the client tries sending some requests. Encountering failures would indicate that the fault causing dependency unavailability is yet to heal, and hence the circuit breaker moves back to the open state. On the other hand, successful requests beyond a threshold would indicate that the dependency is up and running correctly and hence the circuit breaker moves to the closed state.

## III. Distributed Circuit Breaking

Let us consider a system with multiple client instances using the simple circuit breaker pattern that we described in Section II. Let us also assume that all these clients are dependent on a single server instance and all these clients send requests to this instance. If the server becomes unavailable, every client continues sending it requests until the number of failures experienced by it crosses the failure threshold and its circuit breaker transitions to the open state. Every client infers a possible dependency failure on its own.

### A. Observations and Proposal

*Observation 1:* If the server suffers from a failure caused due to faults which are local to the server and which require a variably large amount of time to resolve, the server becomes unavailable to all the clients simultaneously.

*Observation 2:* If a client C finds the server S to be unavailable as a result of the two being in different network partitions, all other clients in the same network partition as that of C also find the server S to be unavailable.

We claim that all correctly functioning clients in a given network partition start to face server unavailability simultaneously if the faults causing the unavailability are not transient in nature. Transient faults may well be due to network congestion or issues local to the client. All clients may not experience failures caused due to transient faults simultaneously. Thus, if the faults causing the server unavailability requires a variably large amount of time to resolve, the clients present in the same network partition would start facing an increasing failure trend roughly around the same time. If this group of clients is able to detect (via some mechanism) that a suspicious increase in failures is being experienced by a majority of the clients in the group, the circuit breakers that are distributed over several client nodes can make a concerted decision to transition into the open state.

In order to define a suspicious failure trend, we propose a soft failure threshold (SFT) - a threshold that is lower than the original failure threshold of the traditional circuit breaker pattern (now referred to as the hard failure threshold - HFT). Failures beyond SFT would indicate a suspicion that the dependency might be facing issues. The SFT value may be decided to be high enough to rule out transient failures. As the SFT value may be considerably lower than HFT, clients can infer a possible server failure faster (with the help of a mechanism to know other clients' failure trends) than they would have otherwise done. Due to a faster inference of a dependency failure and, thus, a quicker transition of the circuit

| Parameter | Meaning |
|---|---|
| WS | Window Size |
| S | Number of Successes |
| F | Number of Failures |
| FT | Failure Threshold |
| HOFT | Half-Open Failure Threshold |
| HOST | Half-Open Success Threshold |
| OD | Open Duration |

TABLE I: Parameters of a simple circuit breaker

| Parameter | Meaning |
|---|---|
| SST | Suspicion Success Threshold |
| SFT | Soft Failure Threshold |
| HFT | Hard Failure Threshold |

TABLE II: Additional parameters for DCBN

breaker to the open state, critical client and network resources would be held up for a lesser amount of time for making requests that are very likely to fail. When the server instance is unavailable, client service instances can fall back faster to a cache or a third party in order to serve the arriving requests, thus improving the externally observed user experience.

We call this scheme, where a group of clients use the knowledge of each other's failure trends in order to make a cautious and more informed choice of moving their circuit breakers to the open state, as "Distributed Circuit Breaking" or DCB. Figure 2 shows the state machine for the individual circuit breaker that is proposed to participate in the distributed circuit breaking scheme. From here on, we refer to this circuit breaker module as "Distributed Circuit Breaking Node" or DCBN. Table II provides the additional parameters required for the functioning of DCBN.

We introduce a suspicion state to the DCBN state machine in addition to the states present in the state machine of the traditional circuit breaker pattern. When the number of failures crosses the threshold SFT, the DCBN transitions to the suspicion state. For the circuit breaker pattern, the closed state is the only state that corresponds to a situation where the client-server interaction is unhindered and as expected. All other states indicate a recent issue in communicating with the server. When a client's DCBN is in the suspicion state, it uses a mechanism (that we will propose in section IV) to monitor if a majority of clients in the same network partition also do *not* have their DCBNs in the closed state. If so, the client's DCBN immediately transitions into the open state as a majority of clients are facing issues in communicating with the server and the suspicion stands validated. If such a majority is not observed and in the meanwhile, the number of failures crosses the hard failure threshold, the circuit breaker transitions to the open state as it would have done in the case of a traditional circuit breaker. If a client is not observing any persistent server unavailability, its DCBN will never transition to the open state and prevent the client from making remote calls. This is because a client's DCBN may transition to the open state only from the suspicion state. A client's DCBN enters the suspicion state only if it is experiencing failures itself.

### *B. Strategy for Implementing DCB*

In a distributed system, service instances may come up or go down over time due to failures or scaling the services up or down. Reliably and efficiently discovering the identity and DCBN states of all other client nodes which are interacting with the same server node in the same network partition is a challenging task. One solution can be to have a new service responsible for keeping the record of active clients, their identities (say IP address), and their DCBN states. However, the consumption of such a service would be susceptible to server-side failures as well as network partitioning. Such faults might cause the pattern to not work at its full potential.

The seminal work by Demers et al. [13] was the first to adopt the concepts from epidemiology to solve the problem of information dissemination in distributed systems. The proposed methods in [13] took inspiration from how rumors spread quickly by certain social interactions or gossiping. Instead of flooding the network with information or one-one interaction of all nodes, the gossip protocol depends upon periodic interactions between each node and a small randomly chosen subset of all nodes.

Following the original work of Demers et al., several others proposed and analyzed variants of gossip-based protocols for a variety of use cases such as to detect failures [14], compute aggregates [15], maintain membership information in P2P networks [16] etc. In [17], K. Birman mentioned a few benefits of gossip-based protocol. The work highlighted how gossip-based protocols are simple to implement, robust to transient network disruptions, can be designed to converge quickly and provide bounded load on the participating nodes. Owing to these properties, the gossip-based approach is used in several complex systems like [18], [19], [20]. Keeping in mind these favorable properties, we propose a gossip-based protocol to implement an efficient information sharing mechanism within a group of clients for the DCB scheme. We also utilize the behaviour of gossip-based protocols towards persistent network failures to ascertain if a given client belongs to the same network partition or not.

We adopt the following scheme of gossiping as the core of the protocol described in Section IV for DCB:

- In the first stage, a node with a piece of information shares it with a fixed number of randomly selected nodes in the cluster.
- At the next stage, every node with the piece of information shares it with a fixed number of nodes chosen randomly and independently of the past and present choices.
- This process is repeated at regular intervals.

This scheme of gossiping is similar to that of spreading a rumor. In [21], B. Pittel estimated that a system using the above scheme, that consists of N nodes (where N is large), requires O(log N) stages of gossiping (expected number of stages) before the information reaches all the nodes in the system.

We call this scheme where we couple DCB nodes with the proposed gossip-based protocol for facilitating distributed circuit breaking as "Gossip Enabled Distributed Circuit Breaking" or GEDCB.

## IV. Proposed Protocol for Implementing GEDCB

The proposed protocol consists of two phases - A and B. Both the phases of the protocol run in parallel. Phase A of the protocol is responsible for efficient and reliable sharing of information about the DCBN states of the clients, that are interacting with a given server, amongst each other. Phase B of the protocol keeps the set of clients participating in the gossip-based information dissemination up to date (and thus facilitates the discovery of client nodes which are interacting with the server in context).

### A. Phase A: Client-Client Gossiping

For illustration, let us assume that there are five clients consuming the dependency service S and that they already know each other's identities.

| #2 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Opinion | 0 | 0 | 0 | 1 | 1 |
| Age | 2 | 0 | 1 | 3 | 2 |

Fig. 3: Example GS state of client #2

Let us call the set of clients that participates in the gossiping process as the gossip-set. In the context of GEDCB, the gossip-set should ideally include all the clients that are interacting with the same server instance. As shown in Figure 3, every client maintains a state (hereafter referred to as gossip-set state or GS state) that consists of opinions and ages (of corresponding opinions) about all the clients in the gossip-set. An opinion of value 0 indicates that the DCBN is in the closed state and a value of 1 indicates that the DCBN is not in the closed state. The age of an opinion is the number of gossip cycles completed since the opinion originated from the source of truth. If the client #2 has a GS state as shown in Figure 3, it infers that client #4's DCBN was not in the closed state three gossip cycles ago. It infers that client #1's DCBN was in the closed state before two gossip cycles and so on. The self-opinion is always zero gossip cycles old as it is updated by the client itself in realtime.

After every $T_1$ unit of time (time period of gossiping), every client increments the age of all opinions (except the self-opinion) and gossips (sends its own state) to a randomly selected fixed-size subset of all the clients in its gossip-set. On receipt of a gossip message, a client updates its own GS state with opinions of a smaller age.

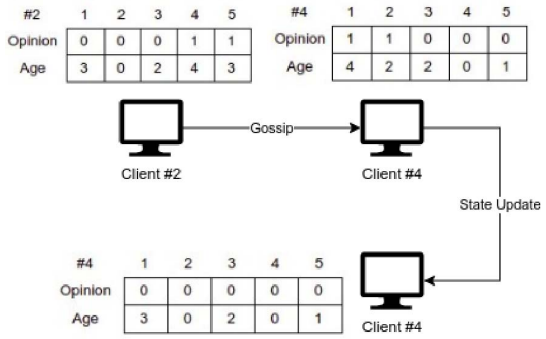



Fig. 4: Example GS state update for client #4

Figure 4 shows how the GS state of client #4 is updated when it receives a gossip message from client #2. Client #4 finds opinions of smaller age about clients #1 and #2 in the gossip message received and hence it assimilates these opinions into its own GS state.

Every client keeps gossiping with a self-opinion of zero age and acts as the source of truth for the state of its own DCBN. Since the opinions that reflect the latest information about the DCBN states are always of a smaller age than the opinions that were shared via gossip messages previously, the latest information about the DCBN states is always assimilated by the receiving client. Owing to the property of the gossiping process, the latest DCBN state information eventually reaches all the clients.

If a node in the gossiping set of nodes goes down or becomes unreachable due to network partitioning, there will be no source of truth for the state of this node's DCBN. Subsequent gossip cycles would cause the age of opinion about this node to increase incessantly. To address this, an empirical threshold is fixed beyond which the age of an opinion is not allowed to increase. For example, if the threshold is fixed to be of ten gossip cycles, opinion about a node of age ten would indicate that the node has either crashed or is in a different network partition. During subsequent gossip cycles, the age of such an opinion is not incremented further. This fulfills our requirement for a client to keep track of and identify all valid clients interacting with the same server instance and present in the same network partition.

| #2 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Opinion | 0 | 1 | 0 | 1 | 1 |
| Age | 10 | 0 | 2 | 4 | 3 |

Fig. 5: GS state of client #2 when client #1 is unreachable

For example, if the client #2 has the GS state shown in Figure 5, it infers that client #1 is either not alive or is in a different network partition. Therefore, opinion about the DCBN state of client #1 should not be used for DCB. Whenever a client finds its DCBN in the suspicion state, it can monitor its GS state to understand the group failure trends. For example, client #2 observes that there are four nodes in the gossip-set (#2, #3, #4 and #5) whose opinions should be considered. As per the information available with the client #2, a majority of these clients do not have their DCBNs in the closed state. Thus, the client #2 infers a dependency failure and its DCBN immediately transitions into the open state. The threshold on the age of opinions is only used for deciding which opinions should be considered for DCB. Even if the client #1 is unreachable, the entire set of five clients is considered for random selection of clients to send the gossip messages. This is done with the optimistic view that client #1 will become active again.

### B. Phase B: Gossip-Set Revision

As the set of client nodes interacting with a given server node S changes as a result of node failures or activities related to scaling the services up or down, the valid nodes still interacting with the server should be able to discover the latest set of all valid client nodes (the current correct gossip-set). The phase B of the protocol takes care of this discovery problem as well as the problem of bootstrapping the newly joined client nodes in the system.

The server S maintains a state as shown in Figure 6. The state includes a version number and the identities of client

| version = 4 | #1 | #2 | #5 | #6 | #7 |
|---|---|---|---|---|---|

Fig. 6: Example state maintained by the server S

nodes that consumed its service till the given point in time. After every $T_2$ time duration, the server increments the version number, sends the list to a fixed-size subset of client nodes on the list and then clears the list.

The clients' GS states are also annotated with the version number of the server state from which they were derived. When a client receives a set-revision message from the server, it updates its version number based on the message. It discards the opinions about clients which are no longer in the new gossip-set suggested by the server. It retains the opinions about the clients which are still present in the new gossip-set. It adds optimistic opinions about the new clients found in the gossip-set. These opinions about the new clients are assigned the highest age permissible for an opinion in the protocol. Due to this arrangement, on receiving subsequent gossip-messages, the client correctly accepts DCBN state information about these new clients which is encoded in the gossiped opinions (which are bound to have an age less than or equal to the highest permissible age).

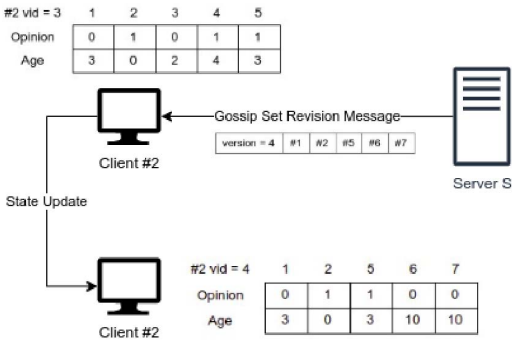



Fig. 7: Example state update based on a set revision message

Figure 7 shows an example scenario where client #2 receives a set-revision message from server S. Once the state is revised, the client starts gossiping with the updated gossip-set. When a client receives a gossip message from another client and it contains a GS state annotated with higher version number, it uses same revision rules to assimilate the new set - discarding opinions about clients not in the new gossip-set, accepting opinions about the existing common and new nodes in the new gossip-set. A gossip message having a GS state annotated with a lower version number is ignored.

When a new client node joins the system and starts interacting with the server S, its GS state is yet to be initialized. At this point, this client's DCBN module essentially acts as a traditional circuit-breaker. Following gossip-set revisions, the client would eventually receive a gossip/set-revision message which it can use to initialize its GS state.

The notion of version number of GS/server state ensures that the correct and the most recent information, about which client nodes constitute the gossip set, is always accepted by the nodes receiving the information from the set revision messages or client-client gossip messages. As gossiped information eventually reaches all nodes in the gossiping cluster, the information about the latest gossip-set reaches all the clients present in that set.

## V. Formal Modeling

To support our claims of superiority of distributed circuit breaking over the traditional circuit breaker pattern, we attempt to model three systems - a system that uses GEDCB, one that uses the traditional circuit breakers (CB) and a baseline system that uses no resiliency pattern. We can get a good idea about the efficacy of our proposals based on the relative performance of the three models.

In this work, we model the systems using the UPPAAL [22] model checker. UPPAAL is suitable for modeling real-time systems as networks of timed automata (TA) [23] that may be extended with data structures. Timed automata are finite state machines extended with clock variables. UPPAAL provides clock variables that may be reset. In version 4.1, UPPAAL allows the usage of a clock variable as a stopwatch [24] by setting the rate of progress of the clock variable to zero in certain states, thus, effectively pausing the clock variable in those states. UPPAAL also allows users to specify functions that may be invoked on certain state transitions. The systems in our context have a great importance to the notion of time with factors like timeout duration, response time, periodicity of sending gossip messages, revising gossip-set, circuit breaker transitions etc. affecting the system behaviour. Modeling the systems in UPPAAL, thus, is well suited for our use case.

UPPAAL also comes with a statistical model checking (SMC) module [25]. SMC module can monitor several runs of the system, and then use results from statistics to get an overall estimate of the value of certain model variables. For our study, total execution time and the number of timeouts are the variables of interest that are compared for understanding the relative efficacy of the three systems under consideration.

We now describe how we modeled the system components in brief. Table III provides the key model parameters and the values which were used. It must be noted that, in practice, deciding values of parameters pertaining to a given resiliency pattern is very subjective. These parameters should be tuned taking into account information about average latency, usual failure rates, availability guarantees etc. In our efforts to model the systems, we fixed values for some parameters and selected the others in a way that appears suitable for the case.

### A. Modeling the server

Figure 8 shows the timed automaton[1] used to model the server in UPPAAL. The server's model includes two clock variables - *clock* and *exec_clock*. The former is a utility clock

[1]In the TA diagrams provided in this work, *I* denotes a location invariant, *G* denotes a guard, *U* denotes an update and *S* denotes a channel synchronization

| Parameter | Meaning | Value Used |
|---|---|---|
| RT | Response Time | 4 |
| TP | Timeout Period | 25 |
| UST | Unavailability Streak Time | 250 |
| WS | Window Size | 10 |
| SFT | Soft Failure Threshold | 2 |
| (H)FT | (Hard) Failure Threshold | 6 |
| HOFT | Half-Open Failure Threshold | 1 |
| HOST | Half-Open Success Threshold | 2 |
| SST | Suspicion Success Threshold | 2 |
| OD | Open Duration | 100 |
| CSP | Client Shuffling Period | 500 |
| $T_1$ | Client Gossiper Period | 4 |
| $T_2$ | Gossip Set Revision Period | 40 |
| MRC | Maximum Request Count | 500 |
| GC | Number of clients gossiped to | 2 |
| RMC | Number of clients receiving set revision message | 2 |

TABLE III: Key model parameters

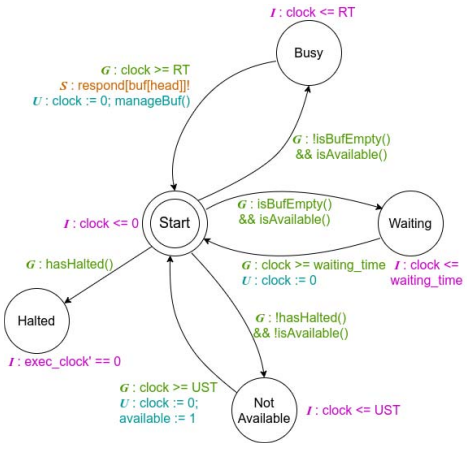



Fig. 8: Timed automaton for the server

while the latter is paused when the server halts (by setting its rate of progress to zero in the *halted* state). The *exec_clock* finally holds the total execution time. A global array *buf* coupled with FIFO access functions acts as the server's request buffer. A global variable *available* is set to 1 when the server is available to serve the clients' requests and is set to 0 otherwise. When the server is available and the request buffer is empty, it waits for a *waiting_time* duration. If the buffer is not empty, it waits for *RT* time before responding to the client whose request is at the head of the request queue. The server invokes the *respond* channel corresponding to the client in order to trigger sending a response. The function *manageBuf()* pops the responded request from the request buffer and updates several counter variables used for modeling unavailability scenarios (to be discussed in Section V-B). The server halts when it successfully responds to a total of *MRC* requests and it sets the global variable *halted* to one.

### B. Modeling Failures

The circuit breaker pattern is specially suited for the case where faults causing dependency failures take a variably large amount of time to resolve. So, we compare the traditional and distributed circuit breaking patterns in identical environments which are characterized by streaks of unavailability. After responding to *ASRC* (Availability Streak Request Count) number of requests, the server becomes unavailable for a *UST* time duration. The server, thus, alternates between streaks of availability and unavailability characterized by the parameters *ASRC* and *UST*.

Let $\mathcal{A}$ ($0 < \mathcal{A} \leq 1$) be the availability of the server. Let $T_{total}$ denote the time elapsed before the server halts. Let $T_{useful}$ denote the time that is useful for the system to make progress.

$$T_{total} = T_{useful} + T_{unavailable}$$

$$T_{useful} = \mathcal{A} \times T_{total}$$

If clients keep the server involved for the entirety of the useful time then the following holds for our models:

$$T_{useful} \approx MRC \times RT$$

$$T_{unavailable} \approx ((1 - \mathcal{A}) \times MRC \times RT)/\mathcal{A}$$

$$USC = T_{unavailable}/UST$$

We use these results to calculate *USC* (number of unavailability streaks) and decide the value of *ASRC* for various availability conditions. Table IV provides the parameter values used to model different availability scenarios.

| Availability | ASRC | USC |
|---|---|---|
| 1 | 500 | 0 |
| 0.8 | 167 | 2 |
| 0.6 | 84 | 5 |
| 0.4 | 39 | 12 |
| 0.2 | 16 | 32 |

TABLE IV: Parameters for modeling availability scenarios

### C. Modeling a Client That Uses No Resiliency Pattern

Figure 9 provides the timed automaton used for modeling the client for a baseline system. We include eight clients in the baseline system. At any given point, five of these eight clients are marked to be alive. Every client has a utility clock variable *clock* that dictates the state transitions. A global counter variable *CPRC* (Clients' Pending Request Count) holds the size of the pool of pending requests. If a client is marked alive

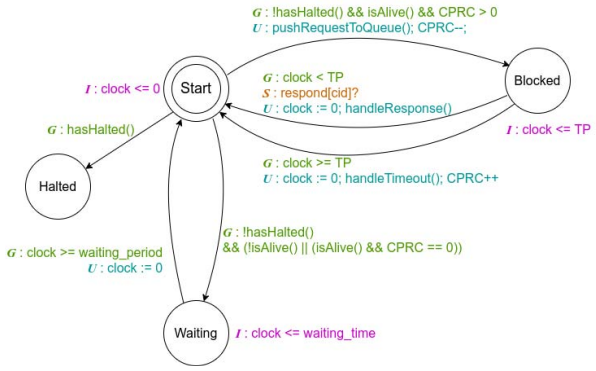



Fig. 9: Timed automaton for client with no resiliency pattern

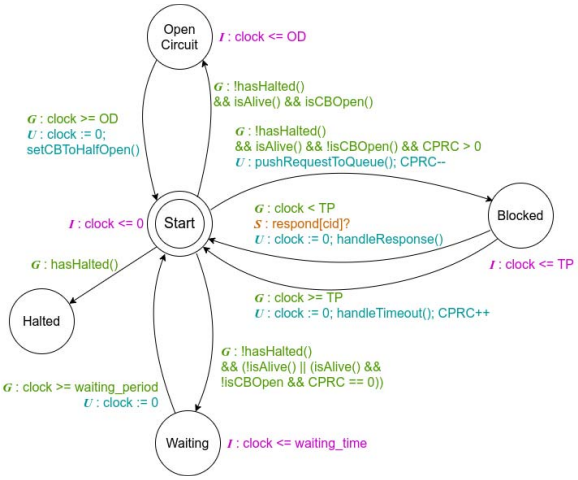



Fig. 10: Timed automaton for client using CB or DCBN

and *CPRC* is zero, it waits for a *waiting_time* duration before checking again. If *CPRC* is not zero, it decrements *CPRC* by one and makes a request to the server by pushing its identifier into the server's request buffer. On making a request, the client waits for a stipulated time *TP*. If the channel *respond[cid]* (where *cid* is the client identifier) is not invoked by the server for synchronization within the *TP* period, the client's request times out. On this timeout event, the function *handleTimeout()* is invoked which increments counter variable *TC* used for keeping track of the number of timeouts. The invocation also removes the entry of its request from the server's buffer. *CPRC* is incremented when a timeout occurs. If the server uses the channel variable *respond[cid]* for synchronization before the time duration *TP*, it indicates a successful response. The client halts after the server halts.

### D. Modeling a Client That Uses CB or DCB

Figure 10 shows the timed automaton for a client that uses a circuit breaker (CB or DCBN). It is an extension of the TA we used for modeling a client which does not use any resiliency pattern. We include eight clients in each system out of which five are marked as alive at any given point. Along with the constructs defined in Section V-C, every client instance of this type has a local variable *cb* that encodes the current state of its circuit breaker. The client model keeps track of the status of a window of last *WS* requests it made to the server. The *handleTimeout()* and *handleResponse()* functions trigger the updating of various counters and the window. Based on the policies described for the circuit breakers in Sections II and III, the state of the circuit breaker *cb* is also updated. This logic of updating the state is different for traditional CB and DCBN and is specified in the code that accompanies the TA formalization in UPPAAL. A client using circuit breaking waits for the duration *OD* when it is in the open state. An "Open Circuit" state and transitions are added to model the same. The client halts after the server halts.

### E. Modeling the Client Shuffler

In a distributed system, service instances may come up or go down over time due to failures or scaling of the services up or down. In order to model this real world scenario, we periodically shuffle the set of active client model instances that are interacting with our server model. The client shuffler randomly marks five of the eight client model instances as alive and the remaining three as dead periodically after every *CSP* time period. Figure 11 shows a timed automaton for the client shuffler module. The client shuffler halts after the server halts.

### F. Modeling the Client Gossiper

For the system using DCB, every client is associated with a gossiper component. Every client's gossiper component should periodically trigger the sending of gossip messages to a random *GC* (Gossip Count) number of the clients it knows. We use the timed automaton shown in Figure 12 for modeling the gossiper. The GS states maintained for DCB by all the

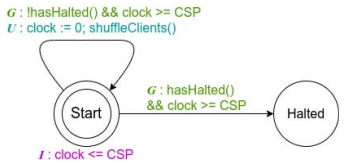



Fig. 11: Timed automaton for the client shuffler

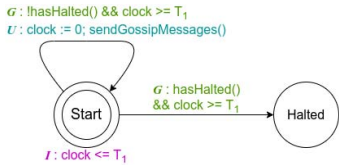



Fig. 12: Timed automaton for the client gossiper

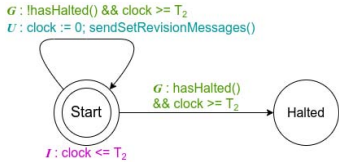



Fig. 13: Timed automaton for the gossip-set reviser

clients are present as globally accessible data structures. The logic implemented using the code that accompanies the TA formalization of the client that uses DCB refers to this data structure when the DCBN is in the suspicion state. After every $T_1$ time period, the gossiper invokes a function that modifies relevant states to mimic the sending and receiving of the gossip messages and updating a client's GS state as per the protocol defined in Section IV. The gossiper halts once the server halts.

### G. Modeling the Gossip Set Reviser

The Gossip Set Reviser module is modeled as the timed automaton shown in Figure 13. The list of clients who interacted with the server in the last $T_2$ time period is recorded in a globally accessible array. After every $T_2$ duration, the reviser component should send this list to the *RMC* (Revision Message Count) number of clients on the list. This message passing is mimicked by invoking a function that updates the GS state of clients who receive the set revision message as per the protocol proposed in Section IV. The function call also triggers the clearing of the list and increment in the version number. The reviser halts once the server halts.

## VI. Results and Comparison

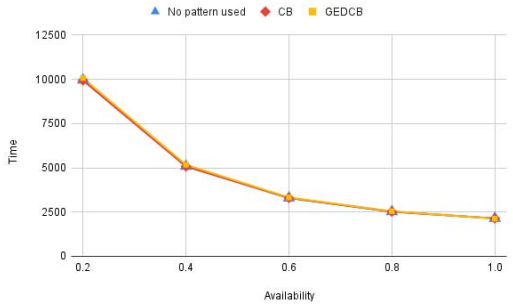
We compose the timed automata described in Section V into three systems of interest - one which uses traditional circuit breakers, one which uses GEDCB and one which does not use any resiliency pattern. We attempted to model a real-world distributed system that is characterized by persistent failures along with frequent changes in the set of clients interacting with a server. We did not model network partitioning within the set of clients. We performed the following two example SMC queries on the three models:

$$E[<= 10500; 500](max : TC)$$

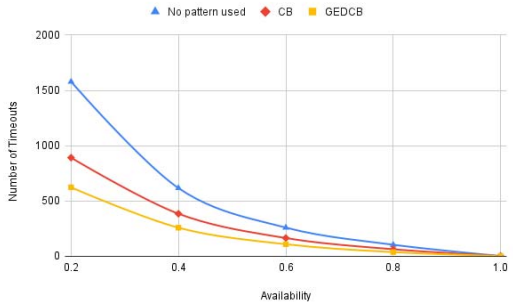
$$E[<= 10500; 500](max : exec\_clock)$$

UPPAAL SMC [25] provides the expected maximum values of certain model variables by using the fact that measurements follow Student's t-distribution [26]. The queries shown above provide the expected maximum value of the variable TC (timeout count) and the clock variable *exec_clock* respectively within the time 10500 units evaluated over 500 simulations. For a given availability value, we first run a few simulations to find the time bound by which the system halts. Once the system halts, the two variables of interest do not increase in value with time. Hence the queries to find the expected maximum value of the variables within this experimentally found time-bound provide the correct estimations.

The expected number of timeouts under various availability conditions is a variable of interest as it indicates the number of failed attempts made by the group of clients for the system to fulfill a given number of requests. The number of timeouts is also a proxy for the cost-overhead needed in terms of client and network resources that are wasted when the system fulfills a given number of requests. Thus, lesser the expected number of timeouts for a given availability condition, better is




(a) Expected execution time for 500 requests




(b) Expected number of failed requests

Fig. 14: Results of SMC queries

the resiliency of the system. We also measure the total time required for the system to fulfill *MRC* number of requests to probe if the proposed pattern slows down the system progress as compared to the traditional pattern.

From the data points obtained by running SMC queries on the three system models, we observe that the time required to successfully respond to *MRC* (500) requests by the server in all three models stays practically the same under any given availability scenario (Figure 14a). This indicates that the proposed resiliency pattern does not introduce any significant overheads that may slow down the system progress. As the DCBN module is local to the client, no latency overhead is added for a client to decide if it should send a request to the server or not.

The data on the number of timeouts (Figure 14b) ratifies the well-known efficacy of the circuit breaker pattern as compared to the case where no resiliency pattern is used. It is observed that GEDCB further reduces the number of timeouts as compared to the traditional circuit breaker pattern. The faster inference of a probable dependency failure helps GEDCB reduce the number of requests that fail. A faster inference would also help to reduce the possibilities of cascading failures as client resources are held up without progress for even shorter time duration when the server becomes unavailable.

## VII. Discussion on GEDCB Parameters and Cost

For any distributed system, deciding the values of system parameters is a crucial and challenging task. In the case of

systems discussed in this paper, the values of parameters used in the resiliency patterns and the timeout duration can affect the behaviour of the system. Selection of appropriate values is necessary for an acceptable performance of a system. An empirical or a model-driven approach may be employed for the same. Mendonca et al. attempted to address this issue in [27] and presented performance results for a different set of parameters for a few resiliency patterns using a model based approach. In GEDCB, the gossip-protocol's parameters dictate the efficacy and the additional cost that accompanies the enhancement over the traditional circuit breaking scheme.

The protocol proposed in section IV does not put a limit on the size of the gossip-set. Based on the use case, an upper bound on the size of the gossip-set may be set in order to put an acceptable bound on the memory requirements, bandwidth usage for individual gossip messages, cost of processing the messages and the number of cycles in which the gossiped information reaches all relevant clients. If the number of active clients is well beyond the upper bound, the server can partition the set of all active clients into multiple gossip-sets that operate in parallel.

The parameters $T_1$ and $T_2$ would also affect the bandwidth usage for the gossip-protocol. In GEDCB, when a client crosses the soft failure threshold, it starts suspecting a dependency failure. The client expresses its suspicion in the subsequent gossip messages. In the meanwhile, the client keeps sending requests to the server. GEDCB would be efficacious only if it is possible to conclude, well before HFT is crossed, whether a majority of client instances are also either facing or suspecting a dependency unavailability. So, the efficacy of GEDCB depends upon how fast this is propagated to all the relevant clients. By this argument, it would be desirable for the clients to gossip very frequently. However, frequent gossiping would increase the network cost. At this juncture, it is observed that this parameter must be set keeping in mind the trade-off between the cost and the efficacy of the resiliency pattern. If gossip messages are sent very less frequently, the GEDCB will regress to the performance of a traditional circuit breaker. This is because, in this case, the circuit breaker might cross the hard failure threshold by the time any inference may be made on the basis of gossip mechanism. Figure 15 shows the effect of varying the delay between two client-client gossip iterations on the expected number of failures in 0.4 and 0.6 availability conditions.

The delay between two consecutive gossip set revisions should also be set according to the system at hand. If the set of client nodes that interact with a given server does not change frequently, the revision may be done less frequently. Based on the system characteristics, different policies might be framed for when to revise the gossip-set. In this work, we modeled a simple policy to update the gossip set at regular intervals.

## VIII. Conclusion

Resiliency patterns are increasingly being used to improve the resiliency of distributed systems. Modern systems that use

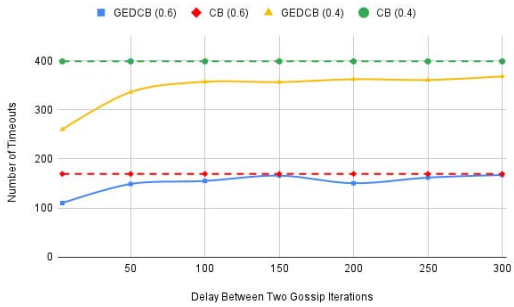



Fig. 15: Effect of gossip period on expected failures

microservice architectures [2] having large and complicated dependency graphs often have to employ resiliency measures to maintain an acceptable QoS. In this paper, we proposed a gossip-enabled distributed circuit breaking scheme (GEDCB). We formally modeled systems that use traditional and the distributed circuit breaker patterns using UPPAAL. We used UPPAAL SMC to do statistical analysis on the modeled systems. The results obtained clearly demonstrated the superior efficacy of GEDCB over the traditional circuit breaker pattern. Finally, we discussed the additional cost of implementing GEDCB and the relevant trade offs that must be considered while deciding the pattern parameters. We have made our models publicly available[2] to facilitate the replication of our study.

Elaborating on the limitations of this work, the very core of the proposal is a gossip-based protocol that expects clients to communicate with each other directly. Needless to say, this might constrain the network environment in which the clients using GEDCB may exist. Clients in different subnets utilizing NAT may not be able to directly communicate with each other. Practitioners should consider their use case and decide if any workarounds are possible and how they affect the efficacy of the pattern.

In this attempt at formal modeling, we considered a simplified case with a small number of client nodes that always keep the server busy. In future, we aim to create a framework for prototyping and implementing various circuit breaking policies and analyzing the resiliency under different availability conditions, network partitioning scenarios, varying parameter values and traffic patterns. There is a need to check how the GEDCB pattern scales with the number of clients and how different workarounds such as multiple smaller gossip-sets affect the efficacy of the pattern. In future, we also aim to provide a proof of concept implementation for GEDCB.

## Acknowledgement

This work has been partially supported by IIT Bhubaneswar Seed Grant (SP093).

[2]The model files may be found at https://github.com/aashaypalliwar/gedcb-uppaal-models

## REFERENCES


[1] T. Erl, *Service-Oriented Architecture: Analysis and Design for Services and Microservices*, 2nd ed., ser. Prentice Hall Service Technology Series. Prentice Hall, 2017.

[2] J. Lewis and M. Fowler, "Microservices: a definition of this new architectural term (2014)," http://martinfowler.com/articles/microservices.html, 2014, [Online; accessed 29-October-2021].

[3] A. Andrews, D. Lee, A. Buck, D. Coulter, and N. Peterson, "Resiliency," https://docs.microsoft.com/en-us/azure/architecture/framework/resiliency/reliability-patterns#resiliency, 2021, [Online; accessed 29-October-2021].

[4] A. Andrews et al., "Bulkhead Pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/bulkhead, 2021, [Online; accessed 29-October-2021].

[5] A. Buck et al., "Retry Pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/retry, 2021, [Online; accessed 29-October-2021].

[6] M. Narumoto et al., "Circuit Breaker Pattern," https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker, 2021, [Online; accessed 29-October-2021].

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

[8] Netflix, "Hystrix: Latency and Fault Tolerance for Distributed Systems," https://github.com/Netflix/Hystrix, 2012, [Online; accessed 29-October-2021].

[9] App vNext, "Polly: A .NET resilience and transient-fault-handling library," https://github.com/App-vNext/Polly, 2017, [Online; accessed 29-October-2021].

[10] Resilience4j, "Resilience4j: A fault tolerance library designed for java8 and functional programming," https://github.com/resilience4j/resilience4j, 2017, [Online; accessed 29-October-2021].

[11] D. Reisenberger, "Distributed circuit breaker consumable from within azure functions or distributed, over http," https://github.com/Polly-Contrib/Polly.Contrib.AzureFunctions.CircuitBreaker, 2019, [Online; accessed 29-October-2021].

[12] Microsoft, "Entity Functions," https://docs.microsoft.com/en-us/azure/azure-functions/durable/durable-functions-entities?tabs=csharp, 2021, [Online; accessed 29-October-2021].

[13] A. Demers, D. Greene, C. Houser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic algorithms for replicated database maintenance," *SIGOPS Oper. Syst. Rev.*, vol. 22, no. 1, p. 8–32, Jan. 1988.

[14] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, ser. Middleware '98. Springer-Verlag, 2009, p. 55–70.

[15] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, no. 3, p. 219–252, Aug. 2005.

[16] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen, "Gossip-based peer sampling," *ACM Trans. Comput. Syst.*, vol. 25, no. 3, p. 8–es, Aug. 2007.

[17] K. Birman, "The promise, and limitations, of gossip protocols," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 5, p. 8–13, Oct. 2007.

[18] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, p. 35–40, Apr. 2010.

[19] Basho Technologies, "Riak," https://riak.com/, 2009, [Online; accessed 29-October-2021].

[20] Hashi Corp, "Consul," https://www.consul.io/, 2014, [Online; accessed 29-October-2021].

[21] B. Pittel, "On spreading a rumor," *SIAM J. Appl. Math.*, vol. 47, no. 1, p. 213–223, Mar. 1987.

[22] Uppsala Universitet and Aalborg University, "Uppaal 4.1.25," https://uppaal.org/, 2019, [Online; accessed 29-October-2021].

[23] "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183–235, 1994.

[24] H. H. Løvengreen, "Stopwatches in uppaal," http://www2.imm.dtu.dk/courses/02224/uppaalsw.html, 2021, [Online; accessed 29-October-2021].

[25] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC Tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, Aug 2015.

[26] Student, "The probable error of a mean," *Biometrika*, pp. 1–25, 1908.

[27] N. C. Mendonça, C. Aderaldo, J. Cámara, and D. Garlan, "Model-based analysis of microservice resiliency patterns," in *Proceedings of the 2020 IEEE International Conference on Software Architecture*, 16-20 March 2020.