Service mesh circuit breaker: from panic button to performance management tool

Mohammad Reza Saleh Sedghpour
Cristian Klein
Johan Tordsson
{msaleh,cklein,tordsson}@cs.umu.se
Department of Computing Science, Umeå University
Umeå, Sweden

Abstract

Site Reliability Engineers are at the center of two tensions: On one hand, they need to respond to alerts within a short time, to restore a non-functional system. On the other hand, short response times is disruptive to everyday life and lead to alert fatigue. To alleviate this tension, many resource management mechanisms are proposed to help a system overcome faults and handle overload. One recent such mechanism is circuit breaking in service meshes. Circuit breaking rejects incoming requests to protect latency at the expense of availability, but in many scenarios achieve neither due to the difficulty of knowing when to trigger circuit breaking in highly dynamic microservice environments.

We propose an adaptive circuit breaking mechanism, implemented through an adaptive controller, that not only avoids overload, but keeps the tail response time below a given threshold while maximizing service throughput. Our proposed controller is experimentally compared with a static circuit breaker across a wide set of overload scenarios in a testbed based on Istio and Kubernetes. The results show that our controller maintains tail response time below the given threshold 98.49% of the time (including cold starts) on average with an availability of 70.79% with 29.18% of requests circuit broken. This compares favorably to a static circuit breaker configuration, which features a 63.97% availability, 30.85% circuit broken requests, and more than 5% of requests timing out.

Keywords: micro-services, circuit breaker, performance management, control theory

1 Introduction

Site reliability engineers (SRE) are responsible for ensuring the service behaves reasonably even in the face of high demand [19]. To achieve this goal, SRE teams should respond to alerts according to service level objective, meanwhile the essence of their duties can be daunting and highly stressful. For instance, if the SRE team response time is defined 5 minutes, then the SRE needs to be connected to network all the the time, while if the SRE team response time is defined 30 minutes, the SRE can leave their home for a short commute [6]. To create automatic capacity management tools that alleviate SRE teams, various resource management techniques has been proposed, such as 1. Scheduling of resources, 2. Resource utilization estimation, 3. Application scaling and provisioning and 4. Workload management [11]. Some techniques are based on changing available resources such as autoscaling techniques, others on making more efficient use of resources such as scheduling techniques, yet others on restricting access to resources to cap performance such as circuit breaking in service mesh.

A service mesh is an infrastructure layer comprising a set of configurable proxies to which services connect. In this way, the network is completely abstracted, providing a single point of network interaction for each service. Service mesh mechanisms are responsible for load balancing, processing network requests, service discovery, authentication, authorization, circuit breaking, and etc. [18]. Circuit breakers provide conditional logic to disallow traffic on a per request level, enabling reactions to overloads that are much faster than those achievable through capacity auto-scaling. This helps prevent poor performance during load spikes [13]. In a chained micro-service architecture, a fault may cascade through multiple layers [24], which can be prevented by using circuit breakers.

Dropping requests for the sake of performance is not a new idea; for example, RED (random early detection) has been used in network routers for many years [7]. The challenge is figuring out when to drop a request in a large dynamic distributed system. This is further complicated by the fact that circuit breaker technologies such as in Istio [1], and Linkerd [3] only allow thresholds to be defined in terms of the number of queued requests or similar metrics that are difficult to map to intuitive measures of service performance, such as response times. Because many factors can affect service performance, even a carefully tuned static circuit breaker threshold can quickly become obsolete.

To illustrate the complexity of configuring static thresholds for circuit breaking, we present an experiment (for details, see Section 4) in which we run a simple application with a circuit breaker carefully tuned for a service using 2 CPUs in a Kubernetes cluster. On this cluster, despite overload, we obtain a 95th percentile response time below 500 ms (left

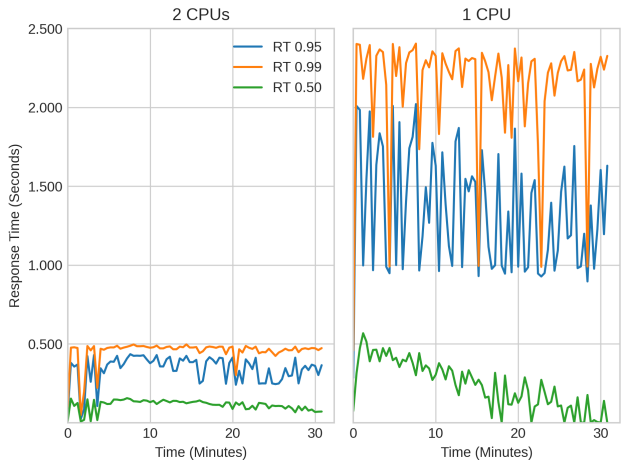

Figure 1. Average (RT50) and tail (RT95 and RT99) response times of a simple service on two servers of different capacities, but equivalent load, with a static circuit breaker tuned for the left-hand scenario.

part of Fig. 1) along with good availability (not shown). However, substantially higher 95th percentile response times and poor availability were observed when deploying the same application on another system with 1 CPU allocation (right part of Fig. 1) using an identically tuned circuit breaker, even though the workload was scaled down proportionally to the reduced CPU allocation. Similarly, changes to the application itself (which may be frequent due to the wide use of continuous integration and continuous deployment pipelines) would also invalidate the tuning of the circuit breaker.

To solve the problem of finding the optimal circuit breaker configuration, we define a controller that protects the tail response time by dynamically adjusting circuit breaker queue-length thresholds. The proposed dynamic controller is compared to a static circuit breaker in multiple experiments on an isolated environment with a 3-node Kubernetes cluster. Over 12 hours' worth of experiments are presented, with more than 13M generated requests. Our approach is shown to maintain response times below the threshold 98.49% of the time (including the cold start time required for the controller to adapt to the operating conditions) in diverse scenarios, while the controller availability is 70.79% and static configuration's availability is 63.97%. Regarding failed requests, 29.18% of requests are circuit broken using our controller, while 30.85% of requests are circuit broken in static configuration using Istio and Kubernetes.

2 Related Works

Dynamic performance management for clouds has attracted considerable attention and there have been several notable reviews of the literature in this area, focusing on autoscaling [16], scheduling [5], and the field in general [11]. Some particularly notable individual studies are highlighted below.

Zhou et al. [25] proposed the DAGOR overload control system for microservice architectures. DAGOR uses the average waiting time of requests in the pending queue to profile the load status of a server. Server overload is detected based on an empirically determined average request queuing time threshold.

Qui et al. [20] proposed a framework that monitors microservices and leverages machine learning methods to detect the root causes of Service Level Objective (SLO) violations before taking action to mitigate those causes via dynamic reprovisioning. Unfortunately, this framework has some limitations in terms of scalability and usability that may not be alleviated without making changes inside the code.

Aquino et al. [4] demonstrate the advantages of the circuit breaker pattern using a traffic light system prototype, in which the circuit breaker significantly improved performance, availability, and accuracy. Montesi et al. [17] distinguished three variations of the circuit breaker pattern. The first is the *client-side circuit breaker*, in which each client includes a separate circuit breaker for intercepting calls to each external service that the client may call. The second is the *service-side circuit breaker*, in which all client invocations received by a service are first processed by an internal circuit breaker that decides whether the invocation should be processed. Finally, there is the *proxy circuit breaker* strategy, where circuit breakers are deployed in a proxy service that sits between clients and services and handle all incoming and outgoing messages. Service mesh-based technologies fall into the latter category because they add a sidecar proxy.

Sun [23] proposed a routing algorithm to improve the latency of service meshes. Moreover, Envoy has a configuration named adaptive concurrency control filter [2], which dynamically adjusts the number of requests that can be outstanding (concurrency) for all hosts in a given cluster at any time. Concurrency values are calculated by latency sampling of completed requests and comparing the samples measured in a given time window to the expected latency for hosts in the cluster [10]. To incorporate Envoy into an architecture, it must be manually added and configured for each service. Istio currently uses Envoy as a sidecar proxy but does not support an adaptive concurrency control filter.

Some aspects of service mesh technologies in need of further research have been highlighted in [15]. As noted by Sheikh et al. [22], an important advantage of service meshes is that they can separate the functionality of run-time operations and micro-services without requiring changes inside the code.

In contrast to the works mentioned above, this paper proposes a controller solution for adaptive circuit breaking that can be applied to arbitrary black-box services in order to maintain acceptable response times and throughput.

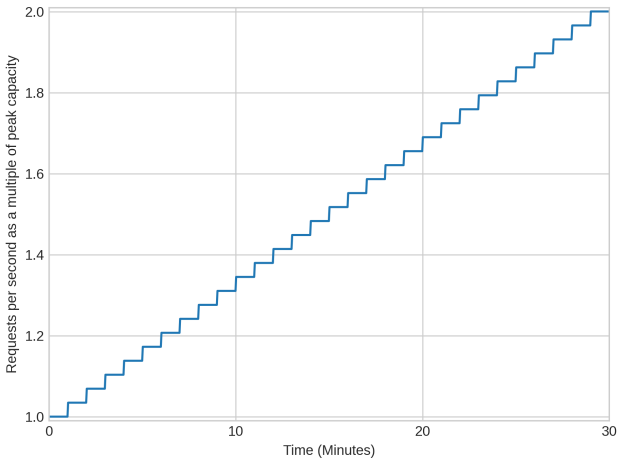

Figure 2. The generated traffic scenario

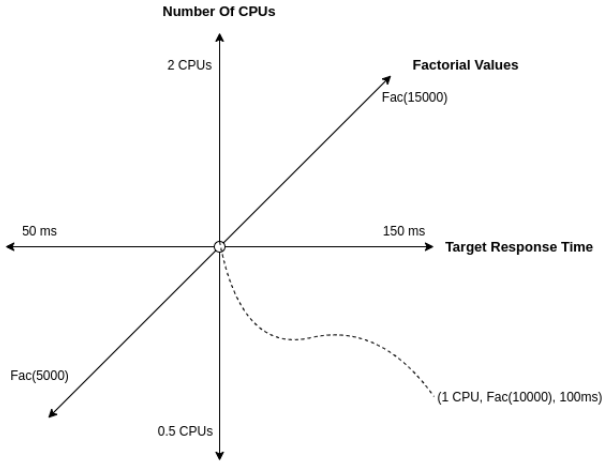

Figure 3. The three-dimensional experimental design space.

3 Solution Framework

We design a controller that uses a circuit breaker to maintain response times below a predefined threshold. The controller's inputs are (tail) response times and queue length, which can be easily monitored from within the service mesh. The actuator is the queue length threshold at which the circuit breaker activates and starts dropping requests. To model the non-linear relationship between tail response time and queue length, we introduce a simple parameter *alpha*, which is the ratio of these two quantities. The controller tunes this parameter online using an exponential smoothing function in order to adapt to changes in workload, server capacity, or service complexity.

Algorithm 1: Controller design

```
Result: Circuit breaker configuration
Parameters: TRT, p;
while Service is running do
    wait 5 seconds;
    retrieve currentResponseTime,
      currentQueueLength;
    notSmoothAlpha = (currentResponseTime /
      currentQueueLength);
    smoothAlpha = (p * smoothAlpha) + (1-p) *
      notSmoothAlpha;
    circuitBreakerThreshold = TRT / smoothAlpha;
    set circuitBreakerConfiguration;
end
```

The design of the proposed controller is shown in Algorithm 1. The controller is configured with two parameters: the Target Response Time (TRT) and a smoothing factor (p). The smoothing factor controls the trade-off between controller responsiveness (maximized when p approaches 0) and controller reactivity (maximized when p approaches 1). The default queue length is 1024, and the initial values for both smooth and non-smooth alpha are both calculated such that they are equal to this default value. Every 5 seconds, we monitor the current response time and queue length, and recalculate both alpha and smooth alpha. The smooth alpha is used to prevent noisy controller behavior. If the smooth alpha increases, the queue length threshold in the circuit breaker is increased, and vice versa. Finally, we use the controller output to configure the number of queued requests in the circuit breaker.

4 Performance Evaluation

In this section, we evaluate the proposed controller's performance with respect to response time and throughput. We first present the experimental setup. Then we explain how we tune the controller, i.e., how we choose the parameter p. Finally, we compare the controller to a static circuit breaking approach in terms of tail response time and throughput. The experiments collectively take over 12 hours to run and generate over 13M requests.

4.1 Experiment Setup

To evaluate our proposed method, we test a factorial service in an isolated environment. While a factorial service is certainly not a common workload, we chose it, since it allows us to experiment with the resource demand per request, as required to compare our approach in a variety of scenarios.

The incoming traffic is generated using the HTTPMon traffic generator [12], which selects a think-time and a number of users, and maintains a number of client threads equal to the number of users. Each client thread runs an infinite loop that waits for a random time and then issues a request for an item or a story. The random waiting time is chosen from an exponential distribution whose rate parameter is

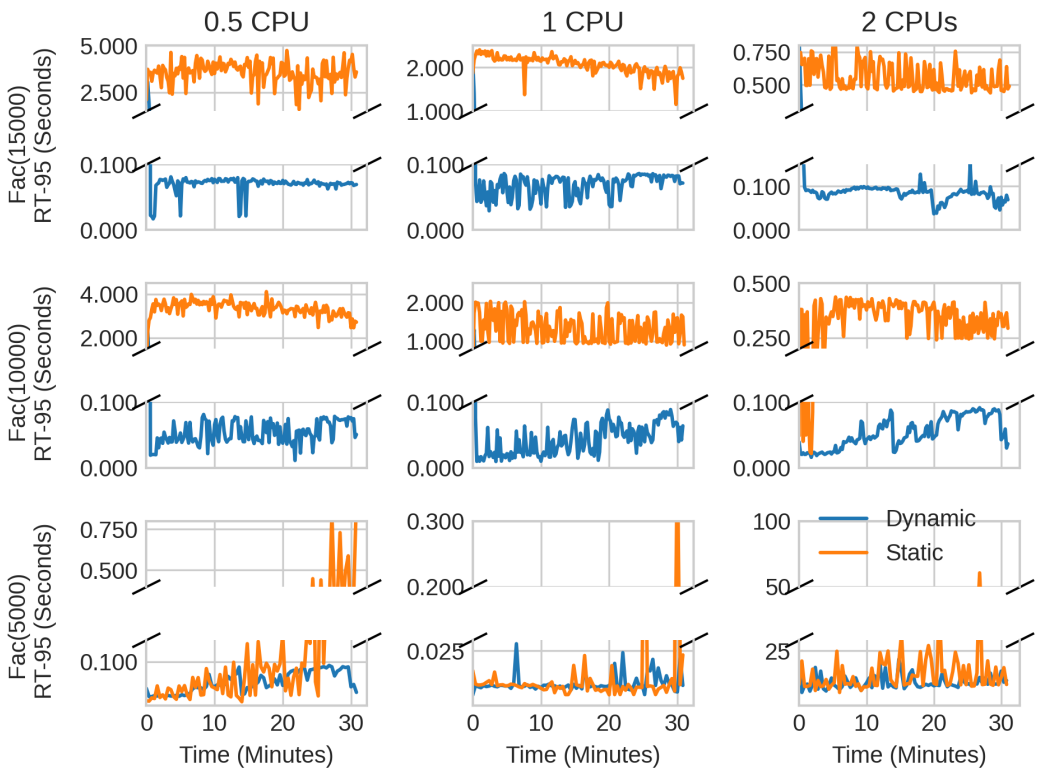

Figure 4. Tail response times for the proposed controller (blue lines) and the static circuit breaking configuration (orange lines) with applications of differing complexity (rows) and different system sizes (columns).

the reciprocal of the think-time. The generated traffic scenario is shown in Fig. 2. As discussed before, SRE teams may not be available all the time, thus the SRE response time of 30 minutes seems reasonable. Therefore, the duration of each scenario is 30 minutes. Experiments were conducted with differing numbers of CPUs, different factorial values, and different target response times, giving rise to the three-dimensional experimental design space shown in Fig. 3.

All experiments are performed on a bare-metal machine with 32GB of RAM, an Intel Core i7-7700 3.6 GHz CPU with four cores and hyper-threading, and a 256 GB NVMe hard drive running Ubuntu 18.04 LTS. We set up three Virtual Machines using KVM and libvirt. In the experiments, each Virtual Machine is allocated two CPU cores, four GB of RAM, and 40 GB of storage; the virtualized operating system is Ubuntu Linux 20.04. To isolate the processes of each virtual machine, we configure CPU affinity for them. The cluster is set up with Kubernetes 1.19.6, Docker 19.03, and Istio 1.8.1.

A factorial service is deployed on a worker node with no other non-required services. The factorial service takes a number as a parameter and responds with its factorial; it performs no caching. The whole stack is monitored using Prometheus. We deploy our proposed proportional controller on the Master node. The controller queries Prometheus to obtain the queue length and response time, and then tunes the maximum number of queued requests in the circuit breaker's configuration (HTTPMaxRequest).

4.2 Metrics

Our controller needs the current queue length and tail response time as input to work. We measure tail response times based on the 95th percentile of the response time distribution (RT95), because it correlates well with user experience [8] while providing a less noisy signal than higher percentiles. Specifically, we extract this metric from Istio's `istio_request_duration_milliseconds_bucket`. To measure the queue length, we use `envoy_http_inbound_downstream_rq_active` from Istio. To evaluate our controller, we use latency and availability metrics with help of aforementioned metrics and `istio_requests_total` from Istio.

4.3 Tuning the controller

Various methods for controller tuning have been proposed [14]. However, the use of such methods is outside the scope of this work because we focus on comparing the proposed controller to static circuit breaking. To find suitable controller

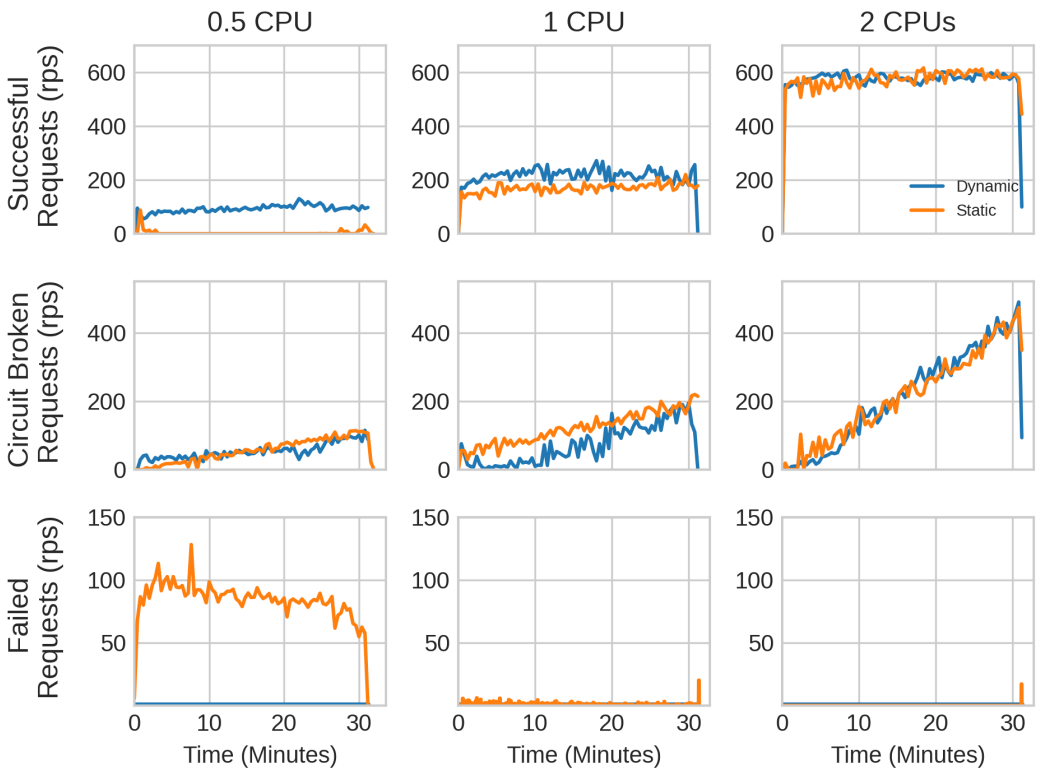

Figure 5. Throughput achieved with the proposed controller (blue lines) and the static circuit breaking configuration (orange lines) for different request statuses (rows) and system sizes (columns).

settings based on well-defined metrics, we deploy the Factorial service and our controller (designed as specified in Algorithm 1) on a server with 1 CPU. We use RT95 as the input of our controller and generate traffic as in Fig. 2, setting a target RT95 value of 100ms. Finally, we run a series of experiments varying the value of the parameter p. From these experiments, we conclude that setting p = 0.9 gives a better trade-off between stability and responsiveness than the other two tested values (0.1 and 0.5). This is consistent with our intuition that the controller should preferentially avoid drastic output changes and be rather stable. Accordingly, p is set to 0.9 in all other experiments discussed herein.

4.4 Comparing response time

To compare the response times achieved with our proposed controller to those for the static configuration, we consider RT95 as the measures of performance. In the response time experiments, we deploy the same service while varying the application complexity and system capacity. Application complexity is varied by setting the factorial parameter to 5000, 10000, or 15000, while the system capacity is varied by setting the number of allocated CPU cores on our Kubernetes cluster to 0.5, 1, or 2. For each experiment, we emulate the traffic, as shown in Fig. 2. The results for the dynamic and static configurations on different testbeds are shown in Fig. 4. The response times for the static configuration vary widely with the system size and service complexity, whereas the proposed controller maintains the response time below the target value. When the system is underloaded (i.e. when the factorial parameter is set to 5000), identical results are obtained with the dynamic and static circuit breakers because the system is not overloaded sufficiently to activate the circuit breaker, but there are some instability in static configuration in case of half CPU allocation. These results show that our controller successfully maintains acceptable response times in setups with widely varying capacities and applications.

To summarize the results of all experiments, the average 50th percentile of the response time distribution for our controller is 4.77 ms while for static configuration is 112.78 ms and the average 95th percentile of the response time distribution for our controller is 49.98 ms while for static configuration is 1263.38 ms.

4.5 Comparing throughput

We also compare the numbers of successful requests (200 HTTP status codes), failed requests (5xx HTTP Status codes), and circuit broken requests (503 HTTP Status codes with UO

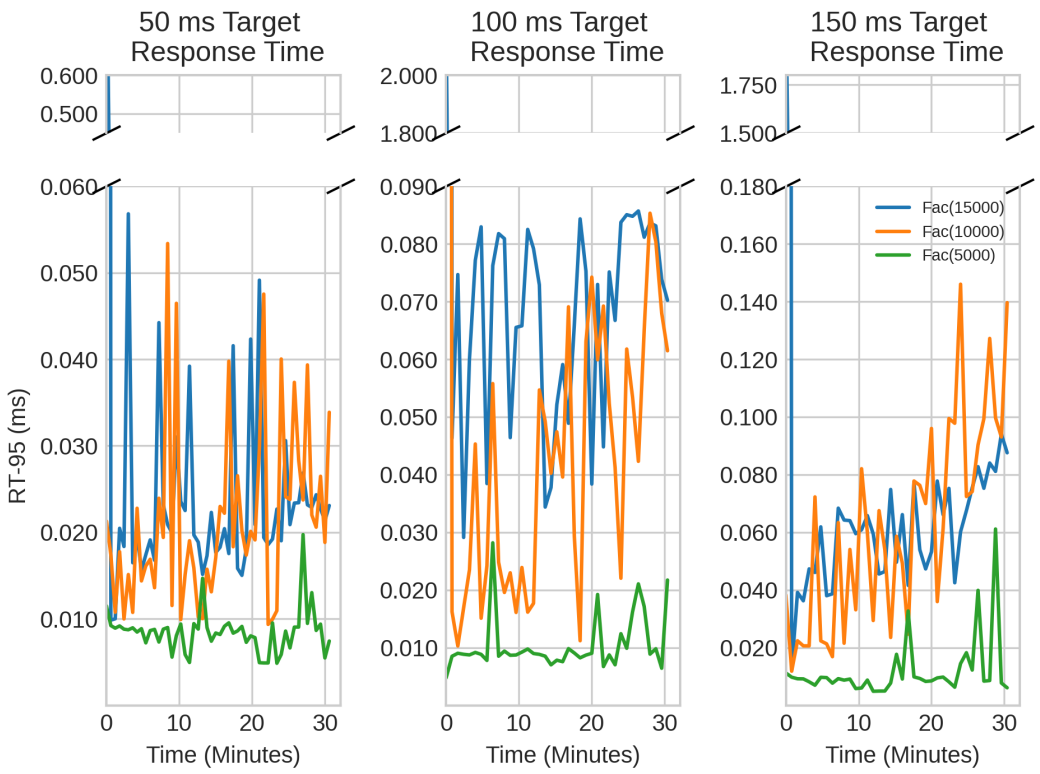

Figure 6. Tail response times achieved using the proposed controller for Fac(15000) (blue lines), Fac(10000) (orange lines), and Fac(5000) (green lines) with different target response times (columns).

flag) for the dynamic and static circuit breaker configurations using the three system sizes (0.5, 1, and 2 allocated CPUs) considered in the previous experiments. In this case, the application complexity is held constant by setting the factorial parameter to 10000 and we emulate the same workload as in the response time experiments. As shown in Fig. 5, our controller yields more successful requests for most system sizes, while the static configuration drops more requests. When the system size is 0.5 CPU, there are some failed requests in the static configuration but not in the dynamic configuration. Our controller thus has negligible effects on throughput.

To summarize the results of all experiments, 70.79% of requests get the served successfully, 29.18% requests are circuit broken and 0.01% of requests fail to get a response due to timeout when using our controller. In contrast, the static configuration obtains an availability of 63.97%, with 30.85% of requests being circuit broken and 5.17% of requests timing out.

4.6 Comparing different target response times

To determine the effect of varying the target response time on the performance of our controller, we deploy the same service as in the response time experiments with the factorial parameter set to 5000, 10000, and 15000 on a system with 1 CPU. Three target response times are tested: 50 ms, 100 ms, and 150 ms. As shown in Fig. 6, the dynamic configuration generally maintains the RT95 response time below 50, 100, and 150 ms independently of the factorial parameter.

4.7 Comparing the different workloads

To study the impact of different workloads on the behavior of the static configuration and the proposed controller, we use a testbed similar to that described above. The results show that workload variation has no impact on the performance of either the static circuit breaker or our dynamic approach.

5 Conclusion

This paper applies control theory to circuit breaker design in order to improve application performance and response times. The goal of this work is to design a dynamic circuit breaking mechanism that avoids the problems encountered using current approaches based on static threshold configurations. We propose an adaptive controller and evaluate it with over 12 hours of experiments and more than 13 million requests. Evaluations are performed under various conditions, using different system sizes and static configurations. We show that our proposed method can be applied to a black box service and is able to efficiently tune the circuit breaker

configuration. As expected, there is a trade-off between response time and throughput, and our controller can easily be configured to optimize this trade-off.

For future work, we intend to see the behavior of our controller in a more complex environment, including multiple services and multiple instances of services. In such scenarios, excessive overload due to so called retry storms [21] are expected to further complicate overload control and circuit breaking, potentially causing limplock (slowlock) problems [9].

6 Acknowledgments

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation. Some parts of the experiments were supported by the Google Cloud Platform research credits program.

References

[1] [n.d.]. https://istio.io/

[2] [n.d.]. Enovy Proxy adaptive Concurrency filter. https://www.envoyproxy.io/docs/envoy/latest/configuration/http/http_filters/adaptive_concurrency_filter

[3] [n.d.]. The world's lightest, fastest service mesh. https://linkerd.io/

[4] Gibeon Aquino, Rafael Queiroz, Geoff Merrett, and Bashir Al-Hashimi. 2019. The Circuit Breaker Pattern Targeted to Future IoT Applications. In *Service-Oriented Computing*, Sami Yangui, Ismael Bouassida Rodriguez, Khalil Drira, and Zahir Tari (Eds.). Springer International Publishing, Cham, 390–396.

[5] AR. Arunarani, D. Manjula, and Vijayan Sugumaran. 2019. Task scheduling techniques in cloud computing: A literature survey. *Future Generation Computer Systems* 91 (2019), 407 – 415. https://doi.org/10.1016/j.future.2018.09.014

[6] B. Beyer, N.R. Murphy, D.K. Rensin, K. Kawahara, and S. Thorne. 2018. *The Site Reliability Workbook: Practical Ways to Implement SRE.* O'Reilly Media. https://books.google.se/books?id=fElmDwAAQBAJ

[7] Bob Birscoe and Jukka Manner. 2014. *Byte and Packet Congestion Notification.* RFC 7141. RFC Editor. 1–40 pages. https://www.rfc-editor.org/rfc/rfc7141.txt

[8] Jeffrey Dean and Luiz André Barroso. 2013. The Tail at Scale. *Commun. ACM* 56, 2 (Feb. 2013), 74–80. https://doi.org/10.1145/2408776.2408794

[9] Thanh Do, Mingzhe Hao, Tanakorn Leesatapornwongsa, Tiratat Patana-anake, and Haryadi S Gunawi. 2013. Limplock: Understanding the impact of limpware on scale-out cloud systems. In *Proceedings of the 4th annual Symposium on Cloud Computing*. 1–14.

[10] Envoyproxy. 2020. Adaptive Concurrency. *Adaptive Concurrency* (2020). https://www.envoyproxy.io/docs/envoy/v1.15.0/configuration/http/http_filters/adaptive_concurrency_filter.html

[11] Brendan Jennings and Rolf Stadler. 2015. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management* 23, 3 (2015), 567–619.

[12] Cristian Klein, Martina Maggio, Karl-Erik Årzén, and Francisco Hernández-Rodriguez. 2014. Brownout: Building More Robust Cloud Applications. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 700–711. https://doi.org/10.1145/2568225.2568227

[13] Lars Larsson, William Tärneberg, Cristian Klein, Maria Kihl, and Erik Elmroth. [n.d.]. Towards Soft Circuit Breaking in Service Meshes via Application-agnostic Caching. ([n. d.]).

[14] Alberto Leva and Martina Maggio. 2009. The PI+p controller structure and its tuning. *Journal of Process Control* 19, 9 (2009), 1451 – 1457. https://doi.org/10.1016/j.jprocont.2009.05.007

[15] W. Li, Y. Lemieux, J. Gao, Z. Zhao, and Y. Han. 2019. Service Mesh: Challenges, State of the Art, and Future Research Opportunities. In *2019 IEEE International Conference on Service-Oriented System Engineering (SOSE)*. 122–1225.

[16] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A. Lozano. 2014. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal of Grid Computing* 12, 4 (Oct. 2014), 559–592. https://doi.org/10.1007/s10723-014-9314-7

[17] Fabrizio Montesi and Janine Weber. 2016. Circuit Breakers, Discovery, and API Gateways in Microservices. arXiv:1609.05830 [cs.SE]

[18] K. Y. Ponomarev. 2019. Attribute-Based Access Control in Service Mesh. In *2019 Dynamics of Systems, Mechanisms and Machines (Dynamics)*. 1–4.

[19] Luis Quesada Torres and Doug Colish. 2020. SRE Best Practices for Capacity Management. *login Usenix Mag.* 45, 4 (2020). https://www.usenix.org/publications/login/winter2020/quesada-torres

[20] Haoran Qui, Subho Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. 2020. FIRM: An Intelligent Fine-grained Resource Management Framework for SLO-oriented Microservices. *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov 2020).

[21] Casey Rosenthal and Nora Jones. 2020. *Chaos engineering.* O'Reilly Media, Incorporated.

[22] Ozair Sheikh, Serjik Dikaleh, Dharmesh Mistry, Darren Pape, and Chris Felix. 2018. Modernize Digital Applications with Microservices Management Using the Istio Service Mesh. In *Proceedings of the 28th Annual International Conference on Computer Science and Software Engineering* (Markham, Ontario, Canada) *(CASCON '18)*. IBM Corp., USA, 359–360.

[23] Zhen Sun. 2019. *Latency-aware Optimization of the Existing Service Mesh in Edge Computing Environment.* Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[24] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, Edward Suh, and Christina Delimitrou. [n.d.]. Sinan: ML-Based & QoS-Aware Resource Management for Cloud Microservices. ([n. d.]).

[25] Hao Zhou, Ming Chen, Qian Lin, Yong Wang, Xiaobin She, Sifan Liu, Rui Gu, Beng Chin Ooi, and Junfeng Yang. 2018. Overload Control for Scaling WeChat Microservices. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 149–161. https://doi.org/10.1145/3267809.3267823