Cloud System Reliability: Expanding the Infrastructure to Support a Commercial Login Service

CS4991 Capstone Report, 2024

Sidhardh Burre
Computer Science
The University of Virginia
School of Engineering and Applied Science
Charlottesville, Virginia USA
ssb3vk@virginia.edu

ABSTRACT

A major American bank holding company, with over 100 million customers, launched a two-factor authentication system that was prone to untraceable crashes and overloaded components. To improve the service, I utilized Splunk and New Relic to construct new tooling and Spring WebFlux to prototype a higher throughput component. Splunk was used to trace individual customers and New Relic monitored system health with service-encompassing dashboards. Java's Spring WebFlux library was used to prototype an async component of the service, enabling greater throughput. The developed tools and dashboards improved crash detection and diagnosis time by 2-3x, while the prototype resulted in a >7x increase in supported concurrent users. Moving forward, the dashboards could be streamlined to focus on high-priority components and automatically generate actionable alerts. The prototype could undergo further testing and a production deployment to ensure more efficient SSE communications.

1. INTRODUCTION

In the summer of 2023, I began an internship with a team that launched two-factor authentication (2FA) method for customers logging into the browser application called the Browser Notification Service (BNS). The goal for the service was to become the de facto 2FA method for all customer-facing endpoints. Therefore, it needed to be highly reliable and have the capacity to support (potentially) the entire user-base simultaneously. But, as the service was in its infancy, it lacked monitoring capabilities and performant implementations, preventing it from meeting reliability and throughput standards.

Cloud system monitoring is a key tool in an incident manager's toolbox. According to a Microsoft study on Cloud production incidents, "quick detection of an incident is crucial to limit its impact" (Ghosh, et. al., 2022). Such quick detection is "achieved with automated monitors." At the time, the team was using rudimentary process monitors. Splunk was used to query across numerous JSON records emitted by every component of the BNS to isolate failed components. But Splunk would take a significant amount of time to run the queries and generate the corresponding visuals. Combined with the fact that there were four versions of this service (one on each coast, as well as a blue/green deployment scheme), the total time taken to run the Splunk query, generate the graphic, and find the source of failure was unacceptably high. Further, the lack of low-level visibility such as hardware utilization, AWS auto-scaling functionality, and more requited a new approach to monitoring the BNS service.

A key part of BNS is a Server-Sent-Events (SSE) stream to a customer's desktop application. This ensures that the moment a customer approves a log-in on their phone, the browser application reflects the authentication. The existing implementation used a Thread Pool

to support synchronous, blocking SSE requiring many resources and providing relatively little throughput in return. Therefore, re-implementing this service more efficiently could reduce hardware utilization and increase scalability, ensuring that the service could scale with user needs.

2. RELATED WORKS

While it is difficult to find related works that properly encapsulate the process at hand, the seminal work by Leners, et. al. (2011) provides an understanding of the problem landscape. In the past, cloud systems were applications executed on bare metal hardware, modern cloud systems are an application running within an OS, which is running on some virtualization software (VMM), which is running on bare metal. Each of these layers of abstraction presents a new opportunity for failure and a new subject of observation. Further, when numerous sub-systems are tied together, as seen in Huang (2018), it becomes difficult to localize the cause of a failure rendering fault assignment and issue resolution difficult.

In a traditional threading model, a new thread is used to support each new request. This causes performance overhead and issues with scalability. Filipchenko (2023) a Google Engineer, explains how Spring WebFlux is a framework that decomposes requests into asynchronous tasks. These tasks are executed in a non-blocking manner enabling the “thread to move on to handle other requests while the tasks are executed in the background.” This framework makes efficient use of CPU resources, enabling high throughput I/O dependent tasks such as SSE.

3. PROJECT DESIGN

This section is structured to offer an understanding of the system in focus. It describes the system's architecture and operational framework, highlighting its deployment strategy and components. Following this foundational overview, the discussion shifts to the implemented monitoring mechanisms, detailing both the challenges solutions applied. Then it transitions into a deep dive into the API re-design efforts, elaborating the strategies employed to enhance system efficiency. It concludes with the results of the changes.

3.1 System Description

The BNS is a 2FA service for a web application. It has four deployments and is hosted via AWS’ Elastic Container Service (ECS) with auto-scaling capabilities. Each microservice is composed of an Auto-Scaling Group (ASG) and an Elastic Load Balancer (ELB) enabling complete automation of the allocation and deallocation of compute resources.

The BNS is composed of twelve microservices and five API endpoints. These services are responsible for the management of users, management of user devices, and the authentication for user login. Yielding seven unique workflows.

While each service produces individual logs, it is difficult to stitch together these logs for a holistic view of the health of the system. Further, the platform on which these logs are hosted, Splunk, is unsuitable for the large scale, repeated log digestion required to analyze the system from a log-based approach. The solution for this problem will be explored in the System Monitoring subsection.

A component of the BNS service is the SSE API. This API is responsible for initializing and maintaining the SSE streams to deliver authorization to the user’s browser. The original implementation of the SSE API was recognized as a bottleneck of the overall service. Despite being the primary microservice, it was highly resource intensive and unable to serve the requisite volume of users efficiently, driving AWS costs in provisioning and maintaining additional compute. The solution for this problem will be explored in the API Re-Design subsection.

3.2 System Monitoring

3.2.1 Monitoring Requirements

One of the earliest steps of any debugging process is fault localization. Without this, it is difficult to determine where and how to allocate resources to effectively address the problem at hand. As the BNS was in its infancy, it lacked the mature monitoring capabilities required for a high-reliability system.

To properly understand the state of the system, the following service metrics must be tracked: the number and health of the allocated Docker containers, the throughput and latency of each microservice, and the quantity and distribution of error codes. Further, the state of the supporting infrastructure must also be known. This includes the hardware resources and the health of the allocated Docker containers.

Prior to the completion of this project, the team would use AWS CloudWatch to monitor the above metrics. While AWS CloudWatch is capable, the UI and latency left much to be desired. Further, the lack of consolidation of CloudWatch information meant that a new tab for each micro-service was required. This left the team seeking a consolidated view of the system's health.

The BNS has seven unique workflows. Monitoring and tracing users through each workflow is essential to understanding the health of the system as well as isolating points of failure from a results-oriented perspective. As such, the final solution must include the capabilities to monitor workflows and track the service metrics listed above.

3.2.2 Monitoring Solutions

As the existing service monitoring software, Splunk was the obvious first candidate. Despite the extensive hours sunk into the Splunk monitoring solution, it was relegated. Splunk queries frequently took minutes to complete, if ever. While this response time may be acceptable in a data analysis setting, it is unacceptable for real-time system health monitoring. Further, the lack of hardware health metrics left Splunk lacking. Due to Splunk's weaknesses, the need for an alternate monitoring tool became apparent.

Splunk was restructured to create a workflow tracing dashboard. Within this dashboard, an engineer could query for a unique transaction ID, readily available in debug logs, to obtain a workflow-length trace of the transaction. This trace provided significant metadata, enabling engineers to debug the workflow and gain a results-oriented view of the system's health. This approach played greatly into Splunk's strengths of high-specificity log parsing.

Yet, we still required monitoring capabilities for the system's hardware, Docker containers, and high level metrics. New Relic was selected as tool of choice for this problem due to its existing integrations with AWS ECS. By injecting the New Relic agent into each and every service, it was possible to forward AWS CloudWatch metrics directly into New Relic dashboards. From that point, it was a process of isolating the specific AWS service and integrating the service into the dashboard's display.

As a result, a New Relic dashboard was created with seven tabs, each targeting a specific component of the service. Three primary tabs will be discussed: the ELB tab, the ASG tab, and the SSE-API tab.

The ELB tab was used to assess the flow of traffic between the various instances within each ASG. It reported on the number and health of the down-stream docker containers as well as the quantity and distribution of error codes. This provided a view of the health of the ELB and redundant monitoring of the underlying Docker containers. With this method, even if the ASGs failed to report their health, it would still be possible to assess the health of the ASGs via the ELB.

The ASG tab provided a high-level overview of each service. It reports the CPU, Memory, Disk, and Networking usages for each of the Docker containers within the ASG. In the

case of aggressive demand, it is possible for the ASG to fall behind in allocating further instances. In this case, load could be shed (requests are automatically denied) if they time out. Therefore, analyzing the ASG dashboard became crucial to assessing the effectiveness of ASG auto-scaling policies.

Finally, each of the four copies of the BNS service featured its own tab of the SSE-API tab. Within each tab were throughput and latency graphs for each microservice component. These tabs are essential in garnering an at-a-glance assessment of each system's health.

3.3 API Re-Design

3.3.1 API Requirements

The BNS-API made use of SSE streams between the server and client. Yet, due to the CPU load requisite for supporting such streams, the existing implementation of the service demanded high resources for relatively poor throughput. The source of this issue was isolated to the programming paradigm used in implementing this service.

The existing implementation used a synchronous threading model. Each SSE stream was processed synchronously forcing each thread to occupy CPU resources for the duration of the SSE connection, bottlenecking the maximum number of SSE. To rectify the issue, an asynchronous programming paradigm was explored. In asynchronous programming, the CPU is free to dedicate its resources elsewhere, only handling the thread when action is required. With this approach, it was possible to increase the concurrent throughput of the service while utilizing the same quantity of resources.

3.3.2 API Solutions

To redesign the SSE API, the Spring WebFlux library was used. This library implemented asynchronous approaches to SSE, message queues, and more. On the backend, this enabled the async monitoring of a Kafka message queue for user approval. On the front end, this implementation enabled the immediate return of a SSE stream promise to the user. When a Kafka message for approval, disapproval, or otherwise was detected, the message was passed to the user via the SSE stream in a asynchronous fashion. This way, waiting for the Kafka message nor maintaining the SSE stream occupied CPU resources (except for keep-alive actions for the SSE stream). To test the efficiency of the new SSE API, an Apache JMeter test bench was configured. Due to time constraints, all testing was performed locally as opposed to deploying and testing the API when hosted within AWS.

4. RESULTS

The monitoring solutions proved highly effective in improving incident resolution latency for the BNS service. This was due to the improved workflow for detecting errors within the service. Prior to the end of the internship, there was an issue with the BNS service. Before Capital One's internal teams could detect the issue, a New Relic alert pinged the entire team about excessive response latency for the "GET Device" API endpoint. By the time the Capital One incident response team contacted the BNS team, the BNS team had already begun to triage the situation. A fix for the service was delivered by the end of the day and according to the manager, the incident was resolved 2-3 times quicker solely due to the newly developed Splunk and New Relic dashboards.

On the SSE API side, it was found that the new implementation of the SSE API could support over eight times as many concurrent users as the prior implementation. Upon presentation of these results to upper management, a request for an abstraction of this service to be used on other teams was communicated. Approximately seven months after the end of the internship, the new SSE API service was deployed into production.

5. CONCLUSION

This project demonstrates the transformative power of utilizing advanced monitoring tools and asynchronous

programming paradigms in improving the reliability and scalability of a critical two-factor authentication service. By integrating Splunk and New Relic for enhanced system monitoring and utilizing Java's Spring WebFlux for redesigning a bottleneck component, we achieved improvements in crash detection and diagnosis time as well as in the system's capacity to handle concurrent users. The monitoring tools facilitated a much quicker response to incidents, thereby enhancing the service's reliability, while the prototype using Spring WebFlux enabled a more than sevenfold increase in supported concurrent users. These outcomes not only improved the immediate performance and reliability of the Browser Notification Service but also laid the groundwork for future enhancements. Looking ahead, focusing on the automation of alerts based on monitoring data and the potential wide-scale implementation of the asynchronous service component could offer further improvements in service reliability and efficiency. This project exemplifies how technological innovation, when strategically applied, can significantly enhance system performance and user experience in a high-stakes banking environment.

6. FUTURE WORK

Future work includes monitoring the New Relic log management feature for further maturity to port over Splunk functionality into New Relic. Further, the existing automated New Relic monitors need to be fine-tuned to reduce the number of false positives, potentially with AnomalyBERT or a similar ML model.

7. ACKNOWLEDGMENTS

I would like to thank my team, Tarundeep Sodhi, Daniel Kreuger, and Matt Daytner. As well as my co-interns Connor Walters and Patrick Lin.

REFERENCES

Ghosh, S., Shetty, M., Bansal, C., & Nath, S. (2022, November 1). *How to Fight Production Incidents? An Empirical Study on a Large-scale Cloud Service*. SoCC 2022. https://www.microsoft.com/en-us/research/publication/how-to-fight-production-incidents-an-empirical-study-on-a-large-scale-cloud-service/

Huang, P., Guo, C., Lorch, J. R., & Zhou, L. (n.d.). *Capturing and Enhancing In Situ System Observability for Failure Detection*.

Leners, J. B., Wu, H., Hung, W.-L., Aguilera, M. K., & Walfish, M. (2011). Detecting failures in distributed systems with the Falcon spy network. *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, 279–294. https://doi.org/10.1145/2043556.2043583

Filipchenko, F. (2023). *An Intro to Spring WebFlux Threading Model.* Retrieved February 17, 2024, from https://hackernoon.com/an-intro-to-spring-webflux-threading-model