Canary-Coalminer Bot: Measuring Messaging Latency

A Technical Report submitted to the Department of Computer Science

Presented to the Faculty of the School of Engineering and Applied Science
University of Virginia • Charlottesville, Virginia

In Partial Fulfillment of the Requirements for the Degree
Bachelor of Science, School of Engineering

Daniel Zhao
Fall, 2022

On my honor as a University Student, I have neither given nor received unauthorized aid on this assignment as defined by the Honor Guidelines for Thesis-Related Assignments

Signature _____ Date _____
Daniel Zhao

Approved _____ Date _____
Briana Morrison, Department of Computer Science

ABSTRACT

Wickr is a business-oriented, secure end-to-end encrypted messaging platform that lacked a tool to measure and report on message latency. Engineers were reactively, rather than proactively, addressing issues that could cause a downtime in service. As an intern last summer, I designed a pair of bots to continuously send ping messages and measure the delay in send and receive times. I built the bot pair on the Wickr Bot API, a library of commands and functions that allowed the creation of bot users, and deployed onto the Wickr client so it could emulate real-time user messaging. The project demonstrated successful measurements and visualization of latency data to help engineers diagnose system health. Further improvements to the bot pair would focus on being able to diagnose the latencies of each individual step in the messaging process, as well as adapting the bot pair to more metrics for insight into other aspects of the system.

1. INTRODUCTION

A common issue with all communication tools is the potential for messaging failure and lag. The culprits can vary from a crashing server, high network traffic slowing messages, or back-end issues slowing down message reception. In all of these cases, the symptoms manifest as trouble for users. Personally, I have had countless issues with Facebook Messenger failing to send messages, but this issue is prevalent through all kinds of messaging systems, such as IMessage, SMS, email, WhatsApp, etc. This problem was the motivation for my summer project, since Wickr also faces this messaging failure problem, especially as it is relatively novel and rapidly developing. At the time, there was no tool that engineers could use to measure message latency or failure.

2. RELATED WORKS

Various methods of measuring latency has already been used and compared. Friston, et. al (2014) attempted to measure latency in virtual environments with frame counting and Steed's algorithm.[2] However, to fully apply techniques, I had to adapt to network latency measurements as Underwood, et. al (2018) did, which found that measuring network performance is equivalent to measuring the change in variation in the latencies. [4]

Many services now are being containerized due to the low overhead needed to scalably support micro-services. One tool that solves the virtualization problem effectively is Docker. Potdar, et. al (2020) found that Docker outperforms a standard virtual machine in RAM speeds, Disk operations, CPU usage, compression speed, etc.[3] This positions Docker as the best possible tool for deploying microservices such as the Canary-Coalminer bot pair.

3. PROJECT DESIGN

The project is built on the two foundations: the Wickr Pro messaging system, and the WickrIO API platform. The messaging system hosts and gives a user-interface for the bots, while the API platform provides the code-base and deployment necessary to create Wickr bots – vital for ping latencies since message communication exists on these foundations. The bot would also be incorporated with AWS services to allow for data visualization and real-time alerts.

3.1 Wickr Architecture

Wickr is built on AWS infrastructure, key components are deployed through EC2 (Elastic Compute Cloud) instances, such as the central message processor and servers, as well as SQS (Simple Queue Service) for message ordering, RDS (Relation Database Service) for message storage, and many more components. A key subgroup of Wickr is Wickr Integrations, otherwise known as WickrIO, which is an API that handles development and deployment of Wickr bots, and the team that I worked closely with.

WickrIO integrations are built on NodeJS and hosted on custom Docker images. All integrations consist of creating bot users through an admin dashboard (Fig 1), loading the necessary code onto the integration, and then interacting with the bot using commands in any chatroom the bot resides in. Although the WickrIO system is proprietary to Wickr, it still allows integrations with third party software, such as AWS compatibility, and thus offers powerful capabilities.

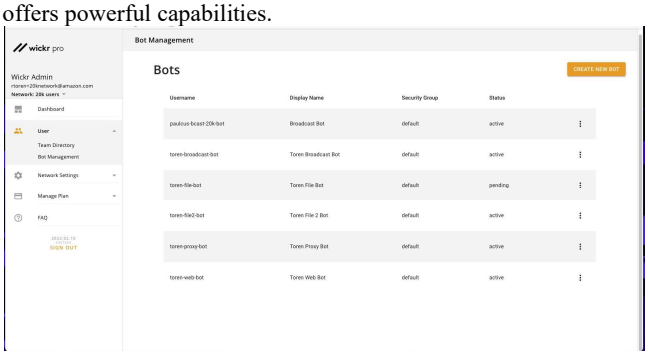

Figure 1: WickrIO Bot Client Creation Page.

3.2 Bot Design

Prior to my project, the Wickr DevOps team began creating a pair of bots that could expose the message latency. The goal was to create an internal tool that existed on the Wickr client to measure and send data to Prometheus, a data visualization tool. The project was able to produce a working bot-pair; however the team quickly discovered a host of developmental issues and dependency problems. More precisely, due to miscommunication between the DevOps and Bots teams, this early version of the Canary Bot project utilized a legacy API for the bot's development. The bot would periodically crash and eventually stop sending data, so the first attempt at message diagnostics was shelved. Though it had many problems, this first bot was well-designed and well-architected, and I ended up designing my solution based on this iteration. I expanded upon the design of the previous iteration of the Canary-Coalminer bot through a few key design choices.

First, because Wickr is hosted entirely on AWS Cloud infrastructure, I architected all components to use native AWS components as it was more sensible and secure. Secondly, I could improve the utility and scalability of the bot by updating to the most recent API version and designing with a factory design pattern. And lastly, I integrated the bots with native dashboard visualization and metric alarming to decrease response time of engineers in critical events. The final design consisted of two bots communicating with each other, one to send messages – the canary, and another to process the messages and create data points – the coalminer.

3.3 Key Functionality

The architecture for the Canary-Coalminer bot includes:

- Canary Instance
- Coalminer Instance
- CloudWatch
- SNS
- IAM Role
- IAM Policy

Figure 2 shows the design of the bot architecture. There are two bots, each hosted on their own instance, in any region. The Coalminer interacts with AWS resources in order to provide engineers insights, such as with metric analysis and alarms. In order to connect, however, the EC2 instance must contain an IAM Role and Policy that gives the bot special privileges to interact with CloudWatch and SNS. The CloudWatch and SNS resources reside in the same region as the Coalminer. CloudWatch is the primary data visualization and alert tool. SNS provides the functionality to notify engineers of bot downtime, latency spikes, or other alarming scenarios. An SNS Topic is an endpoint provided for the alarm to send messages. Often, it is an email endpoint, but can also be texts, SQS queues, HTTP/HTTPS endpoints, etc.

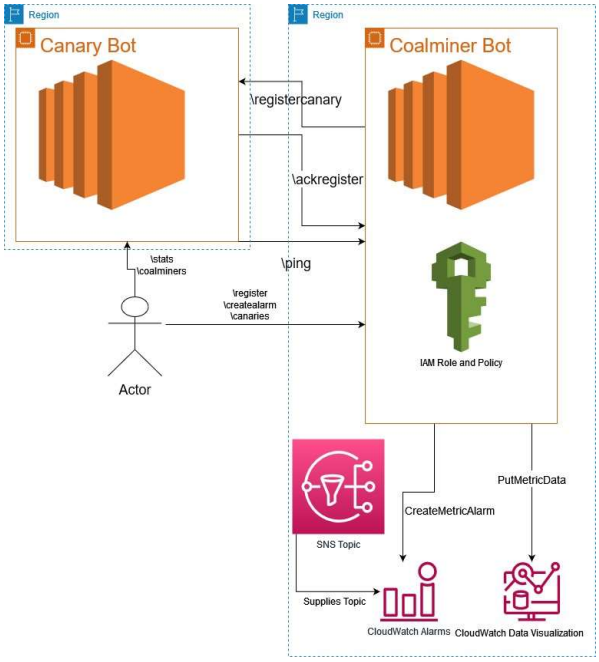

Figure 2: Canary-Coalminer Architecture.

The Canary and Coalminer instances interact with both the user and CloudWatch. After creating and starting the two bots, the user can initiate a connection by calling \register <canary-name to the Coalminer bot, at which point a SYN-ACK handshake is created between the two bots; the Coalminer attempts to register with the Canary, and the Canary will confirm with the Coalminer after it has been registered.

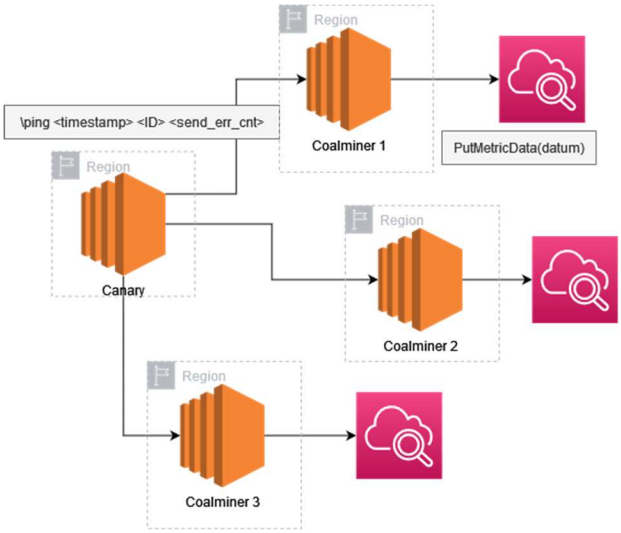

Figure 3: Canary Ping Design.
Canary-Coalminer Bot: Measuring Messaging Latency

After registering, the Canary will start pinging all the registered Coalminers at regular intervals with pertinent data, including the timestamp and ID of the ping. The Coalminer takes this pinged data and processes it to create latency data, missing and out-of order message counts, and message send-error counts. The data is then sent to CloudWatch metrics through the PutMetricData API call. The metric data resides in CloudWatch as pure data points, but also automatically sent to customized data dashboards for quick visualization. As seen in Figure 3, a canary will repeatedly ping the Coalminers, and each will process the data to send to CloudWatch.

The advantages of modular bots are that each Coalminer can be hosted in a different region from each other and from the Canary. This allows engineers to gather latency data from multiple regions using a 1-many design, and can help diagnose how latency is affected by the region the Coalminer is hosted in. The user can also create alarm notifications through the bot by using commands to provision SNS topics or in-house ticketing. These alarms will send warnings to the subscribed endpoints when a metric is acting unusual. For example, if the latency peaks for the past 2 out 3 data points, a message can send to engineers to notify them.

3.4 Challenges

Many challenges arose due to the coupling between the WickrIO infrastructure and the server-API endpoints. Many times throughout the project, the bot would crash due to back-end server failures, and because WickrIO bot clients are not able to automatically restart themselves, valuable data and alarming functionality is lost in the downtime. The bot pair is not designed to notify when the server has issues, which also limits its functionality. Only small patches can be done, such as an alarm to notify when data is no longer being processed. Occasionally, when the bot performed a maintenance reset, it would max out on CPU usage and thus be unable to perform any more functionalities.

4. RESULTS

The bot-pair was deployed in the alpha environment with connections to two separate regions, as well as in the beta environment in one region. At time of departure, the bots were

actively running in these two environments, with automatic provisioning support for CloudWatch dashboards and alarms. The bots also supported notification alerts through SNS topics and Simple Issue Management (SIM) ticketing. Now, with the bots implemented, high latency or downtime that engineers would have been reactively mitigated can now be preemptively fixed before any customers are exposed to the consequences. Thus, the bots achieved the goal of decreasing the response time for engineers.

5. CONCLUSION

I architected and implemented a tool to measure and analyze message latency metrics on using WickrIO and NodeJS. The bot-pair is capable of sending pings between different AWS regions and measuring the holistic delay involved in receiving these pings. These measurements were then sent to a dashboard featuring number gauges and percentile summaries to make the data more available and usable for engineers. By decreasing engineer response time and preemptively alerting of system downtime, the bot-pair demonstrated its functionality as a vital component in the diagnostic toolset that Wickr needed.

Although the tool is theoretically simple and straightforward, it nonetheless is important in measuring system latency and throughput. All network related systems desire low latency and high bandwidth/throughput, and so mature network systems will invest into proper diagnostic tools to maintain these standards. The system I created is important due to its purpose of filling this niche, while also streamlining data visualization and analysis.

6. FUTURE WORK

The major area of improvement for the bot-pair would be to decompose the latency metrics. Essentially, the ping metrics are holistically generated simply as the difference between the time received and the time sent. This unfortunately ignores the details of what sub-process is causing delays, such as a specific server endpoint, or the host experiencing slowdowns on its own system. The ability to subdivide latency into network, processing, and computational delay has already been proven [1]. Thus, being able to pinpoint which server-API endpoint or back-end process is causing the delay will help engineers troubleshoot and fix problems much more quickly.

REFERENCES

[1] K. Chen, P. Huang, and C. Lei. 2009. Effect of Network Quality on Player Departure Behavior in Online Games. IEEE Transactions on Parallel and Distributed Systems 20, 5 (2009), 593–606. https://doi.org/10.1109/TPDS.2008.148

[2] S. Friston and A. Steed. 2014. Measuring Latency in Virtual Environments. IEEE Transactions on Visualization and Computer Graphics 20, 4 (2014), 616–625. https: //doi.org/10.1109/TVCG.2014.30

[3] A. Potdar, N. D G, S. Kengond, and M. Moin Mulla. 2020. Performance Evaluation of Docker Container and Virtual Machine. Procedia Computer Science 171 (2020), 1419–1428. https://doi.org/10.1016/j.procs.2020.04.152 Third International Conference on Computing and Network Communications (CoCoNet'19).

[4] R. Underwood, J. Anderson, and A. Apon. 2018. Measuring Network Latency Variation Impacts to High Performance Computing Application Performance. In Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering (Berlin, Germany) (ICPE '18). Association for Computing Machinery, New York, NY, USA, 68–79. https://doi.org/10.1145/3184407.3184427