15-640 Distributed Systems

Project 3 Implementation and Tuning of a Scalable Web Service

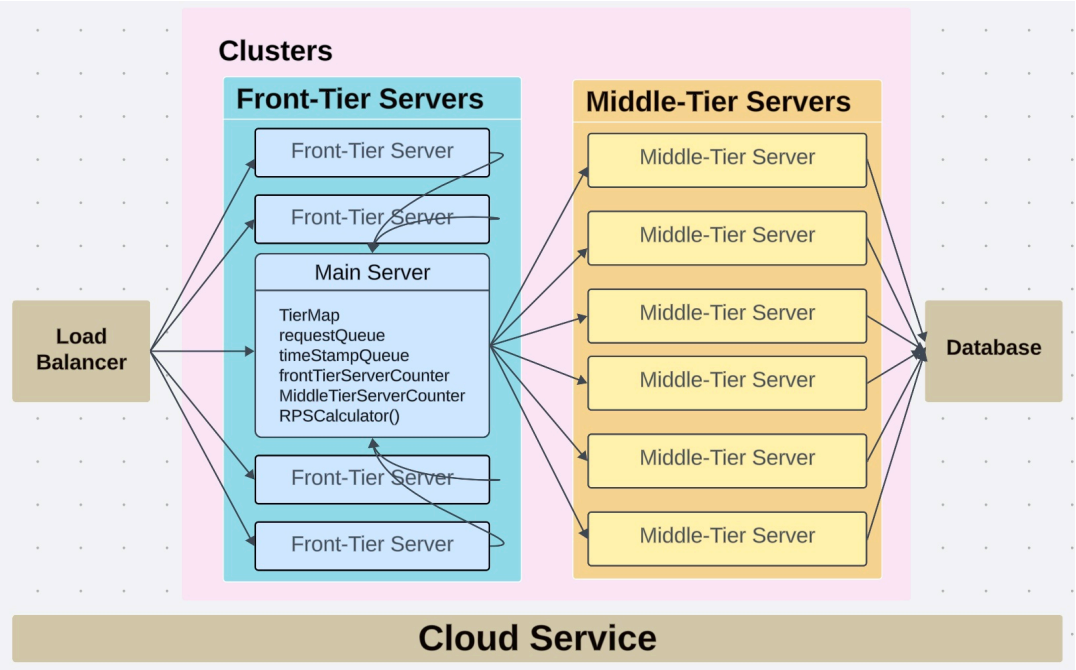
Siqi Guo(siqiguo) April 1, 2024

1 Abstract

In this project, given the context that cloud-hosted services are able to rapidly scale out by adding additional virtual servers on-demand to meet load changes, I implement a three-tier Scalable Web Service and fine-tune it to satisfy dynamic or unexpected workloads by auto-scaling.
During the fine-tuning, I also evaluate the performance of the system by benchmarking and figuring out the bottleneck. My final distributed system is simple but have complex and decent behaviors.

2 The Roles Coordination & 3-Tier Architecture

2.1 3-Tier Architecture



The simulated cloud service, simulated workload and responses, a load balancer, a database, and a Java RMI registry have been provided. I implement the middle tier and the front tier shown in the figure, which I call them the cluster.

2.2 Roles Coordination

Based on this three tier architecture, I implement the following roles in the system: I implement one main server and many other worker servers, inspired by Map-Reduce's concept. The main server is responsible for detecting the workload and deciding whether to scale out or scale in as a Coordinator in this cluster. Itself also can work as a front-tier server, given we want to guarantee the efficient use of VM resources. The worker servers are responsible for handling the requests from the clients. To be more specific, the worker server in the front-tier will mainly parse the requests from the load balancer, and the worker servers in the

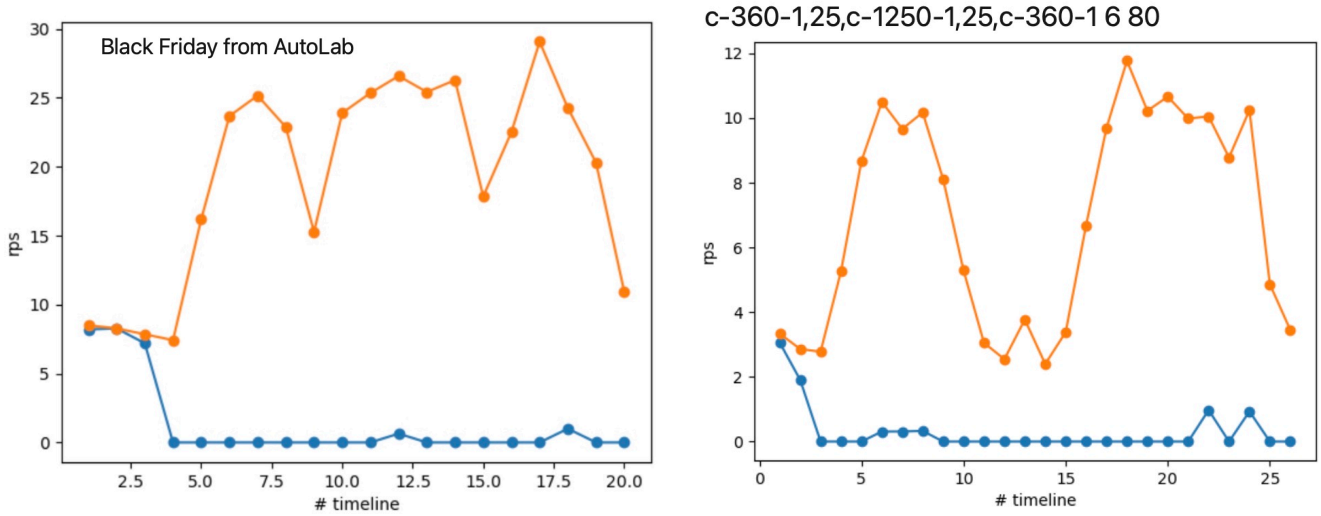
middle-tier (the middle-tier server) will only process the requests from the main server's parsed request queue and reply.
The main server should scale out the servers when parsing loads or processing loads are above some thresholads and provide the information other worker servers need to scale in.
The main server communicate with other worker servers by RMI.

In sum, the main server as the coordinator, could automatically start additional VMs as needed to handle the unexpected load, and VMs that are not currently needed could be shut down, to conserve resources. This design could make the system dynamically scalable and efficient.

3 Auto-Scaling & Fine-tuning the System

3.1 How to decide when to scale out or scale in

I choose requests per second and accumulated requests per second as the metrics to decide when to scale out or scale in. The requests for calculating ARPS are the requests accumulated in the queue, shown as the blue curve. I find it could be metrics to compare with the scaling-out thresholds. However, after warming up, a better metric would be the rps shown as the orange curve in the following figure. By fine-tuning, I find the optimal thresholds for scaling out based on these two metrics.



The servers should be scaled in when they are not working for a while. The length of this period is also a hyperparameter to be tuned. Besides, the window size to calculate the smoothed rps and arps is a hyperparameter to be tuned as well.
What should be noted is that scaling-out detecting should have an interval to let the servers boot, otherwise, the detection results are outdated.

I set several offsets to the number of the servers (both front-tier and middle-tier) to be scaled out at one time, making sure the scaling is rapid and efficient. These offsets are also hyperparameters to be tuned, which represent how much VM resources this system need to satisfy a specific amount of the workload.

Regarding the whole project, and scaling a service by adding tiers and by scaling out the tiers, I realize the importance of the system designing and fine-tuning for a real oriented scenario. Apart from that, figuring out the bottleneck is also crucial for the system to be tuned.