Implementation and Performance Tuning of a Scalable Web Service

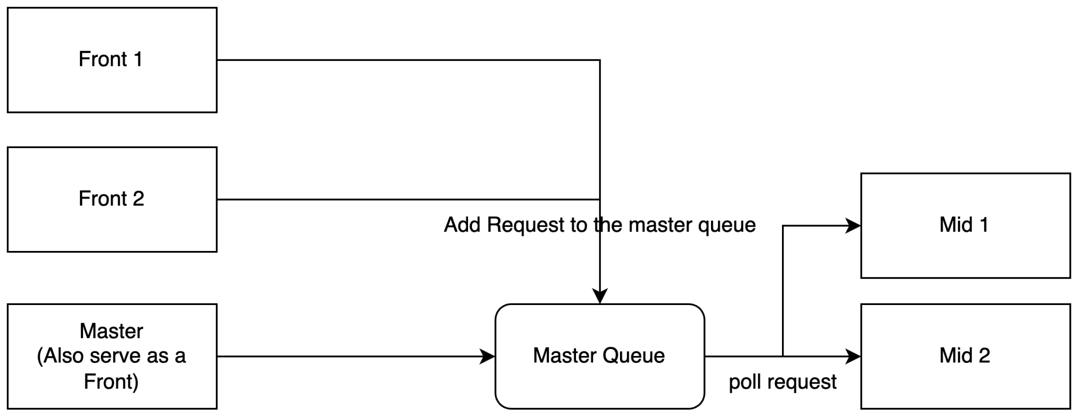
Ruobing Wang AndrewID: ruobing2

1 Project Goals

The primary goal of this project is to develop an efficient, scalable, and cost-effective cloud-hosted multi-tier web service (a web storefront) that can automatically adapt to varying client demands by leveraging the elasticity of cloud-hosted services. This will be achieved by implementing and evaluating techniques for both horizontal scaling through the addition of servers, and multi-tier scaling to improve performance. Furthermore the project contains conducting systematic experiments and benchmarking to identify bottlenecks at different load levels, determining the optimal number of servers for each tier, and implementing dynamic monitoring and scaling to adapt to changing workloads.

2 Design in High-Level

To achieve the aforementioned goal, we propose a design that employs a Master (Coordinator) server, using RMI to coordinate the slave servers. The system will have a single Master server and two types of slave servers: front-tier and middle-tier. The Master server will be responsible for coordinating the scaling in or out of servers through a central master queue. The front-tier slave servers will parse requests from the load balancer and place them into the central master queue. Meanwhile, the middle-tier slave servers will process the requests by polling them from the central master queue. This design aims to facilitate efficient communication and coordination between the tiers, allowing for effective scaling in response to varying client demands. And here is an outlook to the design:


Front 1
Front 2
Master (Also serve as a Front)
Add Request to the master queue
Master Queue
poll request
Mid 1
Mid 2

3 Roles Coordination

In this design, inspired by the concepts of MapReduce and TA, we establish a master server for coordination. To prevent resource loss, the master server will also handle processing requests from the load balancer and it will do some jobs while the first extra VM is booting. With the ability to monitor the number of front-tier and middle-tier servers, the master server utilizes a central queue to manage to scale in and out of servers through Java RMI. Requests are retrieved and added to the queue, while middle-tier servers poll and process the requests from this central queue. This design aims to ensure efficient resource allocation and dynamic scalability in response to fluctuating client demands.

4 Scaling Strategies

4.1 When Booting

Initially, as per checkpoints 1 and 2, we will begin with a predefined number of servers. In checkpoint 3, we will promptly set up a middle-tier server, and during server booting, we will monitor the incoming requests within a specific time interval. If the requests arrive too quickly, we can dynamically scale out additional servers based on the arrival rate to handle scenarios like the Real-Black-Friday test, ensuring optimal performance under varying load conditions.

4.2 Scale Out

We have conducted extensive testing in Ckpt1 and Ckpt2. In Ckpt3, we combine tests with random and varying arrival rates to evaluate our system. An example test configuration is as follows: c-500-111,10,c-250-111,10,c-200-111,10,c-400-111,10,c-250-123,10 5 50. Through benchmarking, we discovered that a 4 + 6 configuration yields excellent results, which we can apply to various tests. In our design, each front-tier server can work with three middle-tier servers. Each front-tier server can handle about 3 to 4 requests, and since there are fewer front-tier servers than middle-tier servers, we need to maintain them for a longer duration, thus giving them a longer survival time. For the middle-tier servers, each can process about 1 - 2 requests to maintain speed, requiring rapid scaling out. Therefore, the cool-down period for middle-tier servers will be shorter.

Front Servers	Mid Servers	Happy Clients	Unhappy Clients
7	5	103	73
6	5	97	79
5	6	144	32
4	6	134	42
5	7	138	38

4.3 Scale In

For scale-in, each slave server will have a time count, which varies depending on whether it is a front or middle-tier server. If there are no remaining jobs in the slave servers after the time count is exceeded, the server will be shut down. Front-tier servers, being fewer in number and handling lighter jobs for processing requests from the load balancer, will have a longer time count, allowing them to stay active for a longer duration. On the other hand, middle-tier servers need to quickly scale out and in to handle step-up and down tests. To address this, we have developed a time count that allows for rapid scaling. Given the relationship that each middle-tier server can handle approximately 2 requests and each front-tier server can handle around 4 requests, the system can efficiently manage server allocation and performance.

5 Conclusions and thoughts

There is no one-size-fits-all solution for any system. The system must be fine-tuned according to real-world scenarios. Through practical experience, we discover that the bottleneck in the system can shift; as a result, we must make trade-offs and identify the knee point to achieve the optimal solution within the given design constraints. Furthermore, horizontal scaling can sometimes introduce added complexities. For instance, in this project, implementing a cache layer could reduce processing time and improve overall efficiency.