Write-Up for Project 3: Auto-Scaling Cloud Server Cluster

Adam Li *zli3@andrew.cmu.edu*

1 Project Description

This is a 4-tier distributed system providing cloud-hosted web service. The system employs dynamic scaling in response to various loads. The system divides its function into numerous tiers: the front tier receives, accumulates, and passes requests from clients to the next tier; the middle tier processes the requests and retrieves relevant information from the database; and the back-end tier (database) receives, gathers, and passes response to the front tier. Both the front and middle tiers are made up of several servers, and each layer may scale in and out on its own. One master server is in charge of all of the other servers and keeps track of the system's condition. The system's scaling method is based on experimental benchmark values.
The system contains a cache tier between the mid-tier servers and the database as a proxy to retain a write-through cache of the database to augment performance and alleviate congestion.

2 Master Server Coordination

In the front tier, we first set up a master server (VM 1), and then all of the other servers in the front or middle tier are servers that communicate with the master server. The master server keeps track of all servers' information, including their server ids and tiers.
Client queries are first processed by a load balancer to distribute to different servers registered as frontend servers. All incoming requests polled by different frontend servers are collected and then send to master's request queue via RMI, and requests from the queue are polled and processed by the mid-tier servers.

2.1 Booting handling

Master's procedure when the first mid-tier sever is still booting. Process at most 15 requests and when process interval is smaller than 800 ms, drop the first request in the serverLib queue. When rate is too high during booting, start 3 middle-tier and 1 front-tier in a row.

3 Scale Out Strategy

3.1 Front-tier

The scaling out of the front tier is base on the message queue length that the master server is maintaining. If the length of the master's message queue is longer than the product of the number of mid-tier servers in the system and a mid factor, i.e. $L_{master} > N_{front} * C_{front-factor}$, then we scale out one mid-tier server in the system. The $C_{front-factor}$ correspond to the `frontFactor` parameter in the code. We set it to 3.0.

Also there is a cool down period for adding front servers. This is for preventing adding too many front-end servers at the same time. The add front interval is set to 4000 ms.

3.2 Mid-tier

The scaling out of the middle tier is also base on the message queue length that the master server is maintaining. If the length of the master's message queue is longer than the product of the number of mid-tier servers in the system and a mid factor, i.e. $L_{master} > N_{mid} * C_{mid-factor}$, then we scale out one mid-tier server in the system. The $C_{mid-factor}$ correspond to the `midFactor` parameter in the code. Based on the following bench marking, we set it to 3.3.
For stepping up test case (self-simulated)
`c-900-123,10,c-555-123,10,c-444-123,10,c-333-123,10,c-133-123,20`.

midFactor	Happy	Total	VM time
3.8	136	227	469
3.4	160	227	493
3.3	174	227	435
3	95	227	402

Table 1: Step up performance, when allowedIdleCycle = 3000

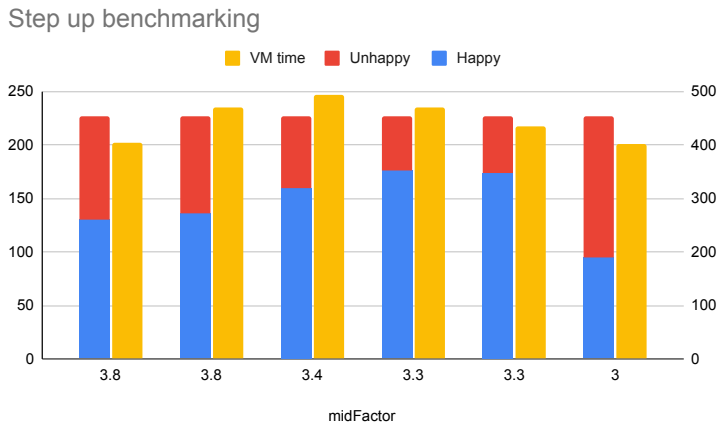

Figure 1: Step up performance, when allowedIdleCycle = 3000

midFactor	Happy	Total	VM time
3.4	117	227	399
3.3	148	227	402
3.2	142	227	412
3.0	98	227	392

Table 2: Step up performance, when allowedIdleCycle = 2000

4 Scale In Strategy

4.1 Front-tier

If master request queue stays under 2 in length for 55 cycles, shutdown current front server. Here cycle denotes the while loop routine of each front-end server. I decide not to add cool down period

for scaling in of front-tier because this can provide a better performance when request rate is quickly stepping down.

4.2 Mid-tier

If the mid-tier server keep getting null request for `allowedIdleCycle` ms, this means that the quantity of mid-tier servers in the system is an overkill and need to be shutdown. Getting null request means that the master request queue is empty and there is more consumers (request processors) than providers (client requests). We set `allowedIdleCycle` to 2000 ms. When `allowedIdleCycle` is 2000 ms, it can provide most happy clients for stepping up, as well as satisfying scale in performance for stepping down test cases. Most importantly, it offers the least VM time.

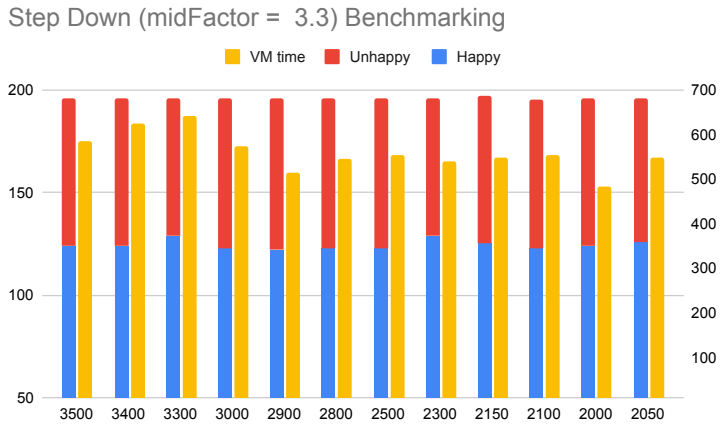

Figure 2: Step Down (midFactor = 3.3) Benchmarking

5 Cache Implementation

The cache is implemented as a proxy for the database, caching the response of the request made by each client, thus augmenting the performance of the entire system. It is a check-on-use and write-through cache, which handles most of the read operations for the "browse" requests

6 Things I Learned about Scaling

There is no one best scaling architecture that can be applied to all cases, hence scaling should be adaptable. Experiments and trials should be used to test the scaling design if at all practicable. It's more about trade-offs when it comes to scaling. Several criteria should be considered when determining when and how many to scale in and out. The load on each server and the in-coming request rate are the essential elements in this project.