CMU 15-640 Project 3
Design Document

Xiaoyun Ye
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
xye2@andrew.cmu.edu

System description

This is a simple 4-tier system providing cloud-hosted web service. The system implements dynamic scaling according to different loads. The system splits its service into multiple tiers: the front tier receives, gathers requests from clients and pass the requests to the next tier, the middle tier processes the requests and get required data from the database.
Both front and middle tier consist of several servers, and each tier can scale in/out independently. There is one master server in control of all the rest servers and keep tracks and records of the system's status. The scaling scheme of the system is based on benchmark values from experiments.
In order to alleviate the load of the database, the system has a cache tier which keeps a write-through cache of the database to cope with the read requests.

Server coordination

This is a master-slave server system. We first initializes a master server (VM 1) in the front tier, and all the other servers in the front or middle tier are slave servers which all communicate through the master server. The master server keeps a list of all servers' information: server ids and the tier they belong to.
All front servers receive requests from clients via Java RMI. The master server collects all incoming requests in a request queue and the middle servers poll requests from the queue.

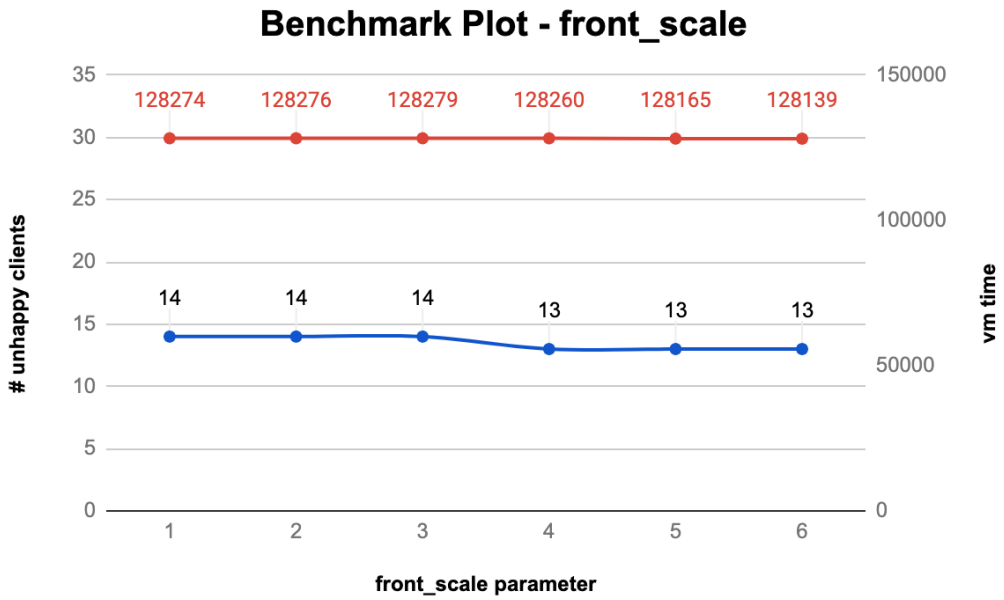
Initial scaling setting

The system initializes with only **2** front servers(one is the master server) and **1** middle server. This is to minimize the booting time and the initial request drops.

Scaling-out strategy

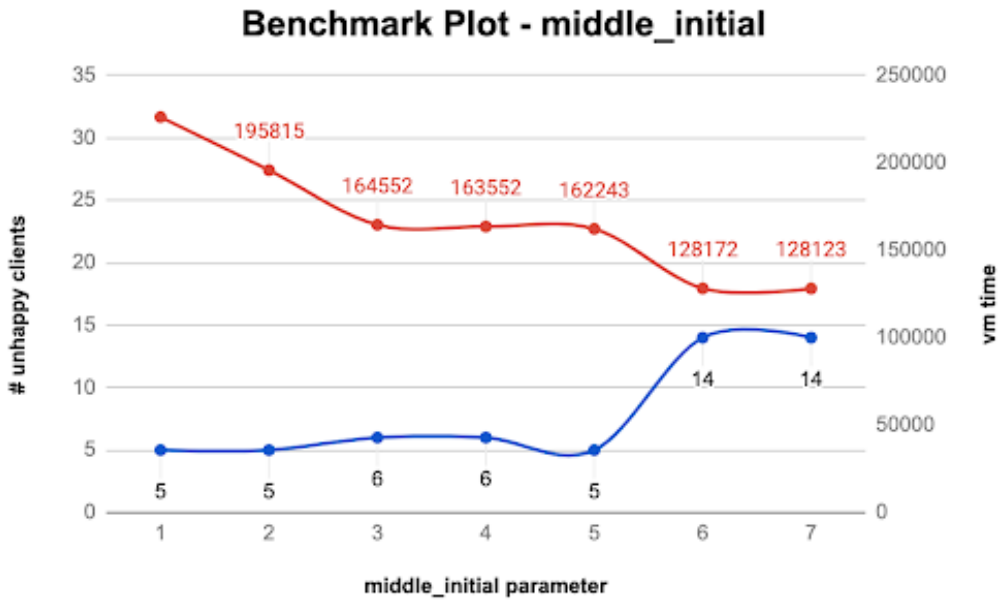
Both the front and the middle tier scale out based on the number of dropped requests.

Front tier
During booting time, the front tier does not scale out, instead, it just drops the coming requests.
During running time, The front tier scales out a new front server when the request queue size is **5** times the number of front servers. The `front_scale` parameter in the code represents this threshold value, which is chosen based on the experiments as the following plot shows.

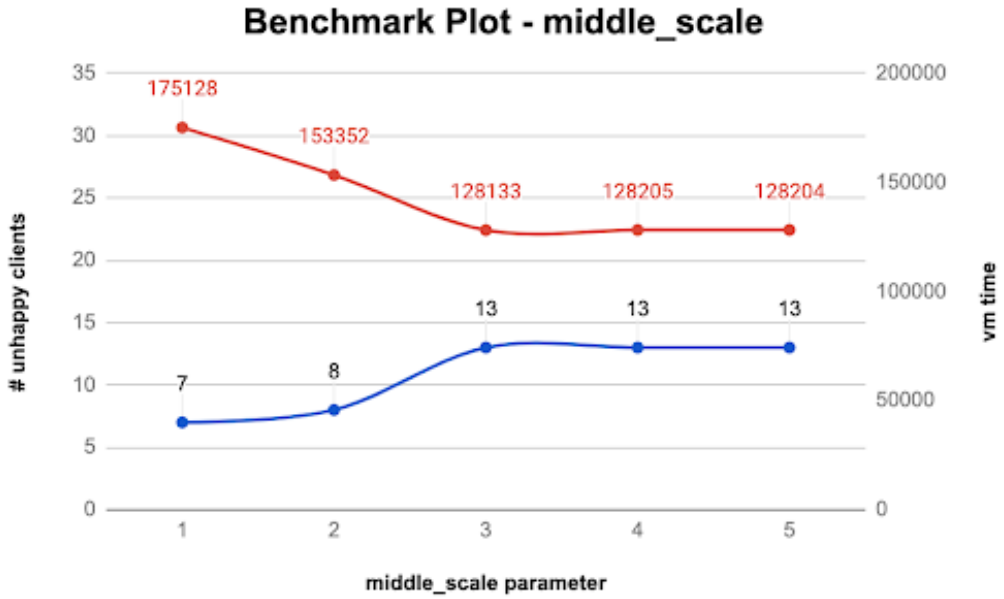


Middle tier

During booting time, the middle tier scales out a new middle server when the the number of dropped requests is a multiple of **5**. The `middle_initial` parameter in the code represents this threshold value, which is chosen based on the experiments as the following plot shows.

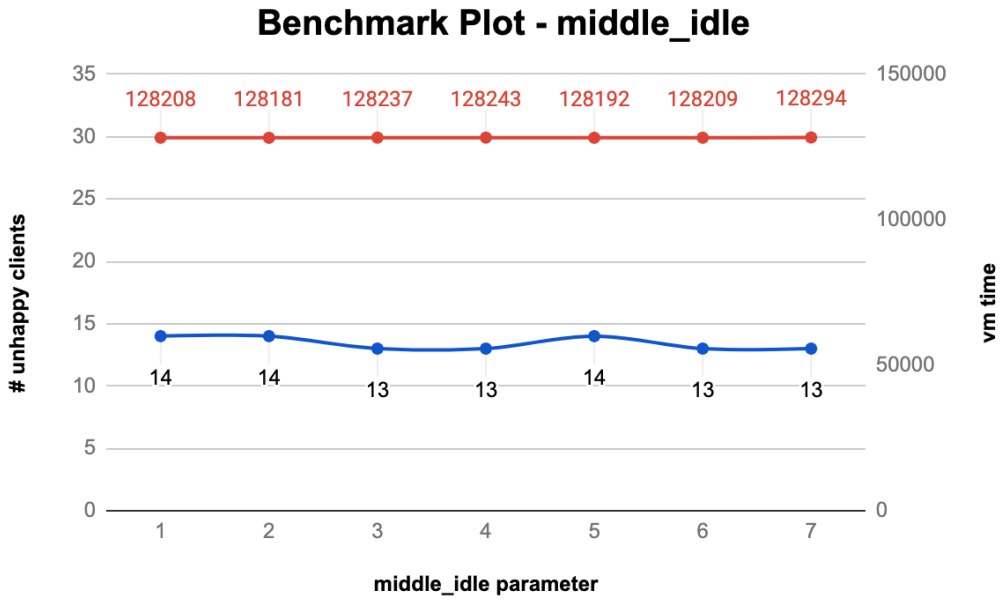


During running time, the middle tier scales out a new middle server when consecutively **3** requests have been dropped. The `middle_scale` parameter in the code represents this threshold value, which is chosen based on the experiments as the following plot shows.



Scaling-in strategy

Middle tier
The middle tier scales in and removes an idle middle server when that server gets **3** consecutive time-outs/null value when polling requests from the master server's request queue. The `middle_idle` parameter in the code represents this threshold value, which is chosen based on the experiments as the following plot shows.



Front tier
The front servers only receive requests and do not process them so the workload in the front tier is not time-consuming. And since incoming requests can change rapidly in different scenarios and

frequently scaling in/out the front tier will cause a waste of VM time. So the system does not implement explicit scaling-in strategy for the front tier.

Database cache implementation

The cache implements cache-on-use policy, fetches the data from the database when `get` requests are called and stores it locally. And the cache only serves for `get` requests and return the information about the required items from local copy (when cache hits) or from the cloud database (when cache misses).
For `set` and `transaction` requests, the cache will not process but directly pass those requests to the database.

What I learned about scaling

Scaling should be flexible, there is no one optimal scaling design that can be applicable to all scenarios. If possible, we should use experiments and trials to test the scaling design.
Scaling is all about trade-offs. When considering when and how many to scale in and out, we should take several factors into consideration. In this project, two important factors are the running time and the drops and timeouts.