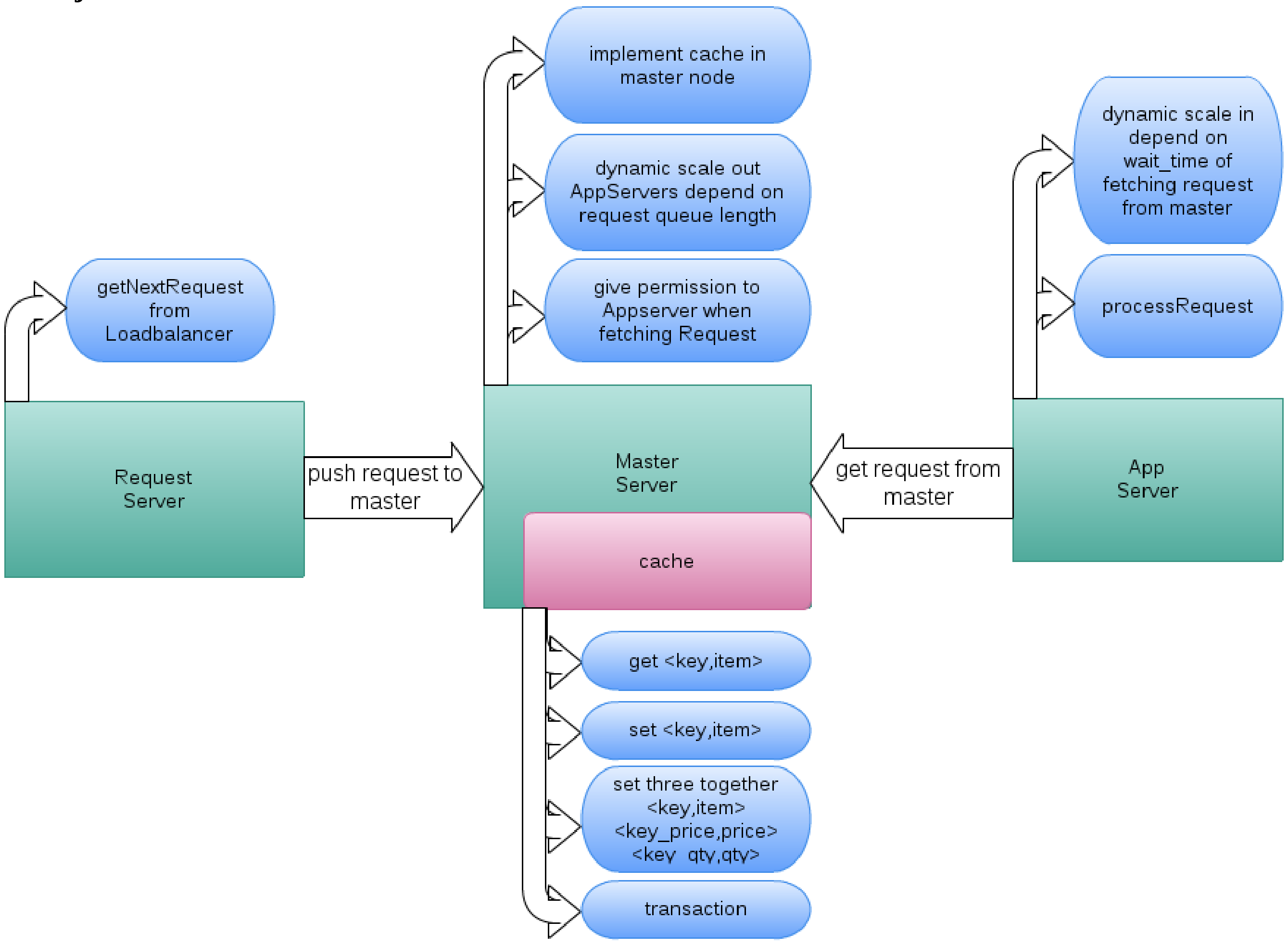
Project3: Scalable Service WriteUp

jinxi@andrew.cmu.edu

1 Project Architecture


implement cache in master node
dynamic scale out AppServers depend on request queue length
give permission to Appserver when fetching Request
dynamic scale in depend on wait_time of fetching request from master
processRequest
getNextRequest from Loadbalancer
Request Server
push request to master
Master Server
cache
get request from master
App Server
get <key,item>
set <key,item>
set three together <key,item> <key_price,price> <key_qty,qty>
transaction

2 coordinate the roles of different server instances

To make sure every VM know its own role, only the master node will call the startVM() method, and it will push the “role_of_new_VM + VMid” into a static stack named <ids>. For example, following script shows how to launch a new AppServer in master node:

```
launch_id = SL.startVM();
ids.add("AppServer" + " " + launch_id);
App_ids.add(launch_id);
```

As a slave server, when it start running, it will call the RMI method server.getId() and pop the <ids> stack to get it role. It will decide itself to be a appServer or requestServer depend on the return role from master. The dynamic scale in-out policy is only implemented in Appserver.

3 scale out policy

There is a request queue which stores the unprocessed requests in master node, every time requestServer call getNextRequest from loadbalancer and call RMI pushNextRequestToMaster method to trigger enqueue. AppServer will call RMI getRequestFromMaster method to trigger dequeue. So the basic scale out policy is checking the current size of request queue and decide whether to scale out. What is more, current system time also has influence on the number of new AppServer it will launch, it time is small, it is more possible to launch more AppServers. Every time scale out successfully, a cool time (make it 500ms) is implemented to avoid sensitivity. A upper limit (11) of AppServers is enforced.

Last but not least, to implement this policy, a filter is used when checking the queue size. Every

20ms the master node will sample the current queue size for 100 times. Then get the average queue size in the past 2second.

4 Scale in policy

Every time the AppServer fetch a request from master, it will be blocked until the master make sure the current queue is bigger than 1 and this operation is synchronized. AppServer will record this block time. The basic idea of scale in is according to this block time, if the block time is more than a certain value, it will shutdown itself.

When accessing the block time, a filter also used here. A sliding window of whose size is 6 is implemented. That is to say, we will get the sum of most recent 6 sample of block time.

What is more, we cannot make decision of scale in only depend on block time. Every time the AppServer is going to shutdown itself, it will ask for permission from master node. Master will count the number of current AppServers and a cool time also promised here. Only the the number of current AppServers is bigger than 1 and it is not in “cool state”, a permission is granted to the AppServer. Otherwise the AppServer cannot shutdown itself.

5 difference of scale in and scale out

Scale in: AppServer monitor itself block time and make decision to kill itself or not. But to submit scale in operation, it should ask for permission from master node. Every time only one AppServer can be killed.

Scale out: Master node monitor the request queue size and make decision to scale out or not. No permission from slaves node needed. The number of AppServer to be scaled out is changeable.

6 cache implementation

Wrote a single class named Cache which implements synchronized method named get(), set(), transaction(), and setThreeTogether() (this is called when the item.equals(“ITEM”), want to put item, price, qty in one RMI call). At the master node, it will create a separate thread to create a instance of Cache class and register as a RMI object.

How to access cache: The AppServer will decide whether the request is “purchased” or not, if it is “purchased”, then go to database directly; if it is “not purchased”, then it will verify whether this item is in cache. If cache hit, then fetch this item from cache directly; If cache miss, it will get <key,value> pair from database and put them into cache, then handle the request through cache.

6 use wait() and notify() to make sure everytime AppServer can get valid request when call RMI getNextRequestfromMaster() method. Avoid completing condition and meaningless RMI call.

Since every AppServer has it own id, so in the getNextRequestfromMaster(), it will lock itself and push this unique id to the queue named <readyAppServers>.

```
Integer integer = key;
readyAppServers.add(integer);
synchronized (integer) {
      integer.wait();}
```

In the master thread, every 10ms it will check the both queue named <readyAppServers> and <requestQueue>, if there exist request in the <requestQueue> and exist AppServer waiting for fetching request, it will notify the specific AppServer and finish the RMI call successfully.

```
if (requestQueue.size() > 0) {
      if (readyAppServers.size() > 0) {
            Integer integer = readyAppServers.get(0);
            readyAppServers.remove(0);
            synchronized (integer) {
                  integer.notify();}}}
```