

15-440 Spring 2015

Project 3: Scalable Service

Important Dates:

Project Handout:	Tuesday March 24, 2015
Checkpoint 1 due:	Friday March 27, 2015, 5:00 PM EST
Checkpoint 2 due:	Friday April 3, 2015, 5:00 PM EST
Final Due:	Friday April 10, 2015, 5:00 PM EST

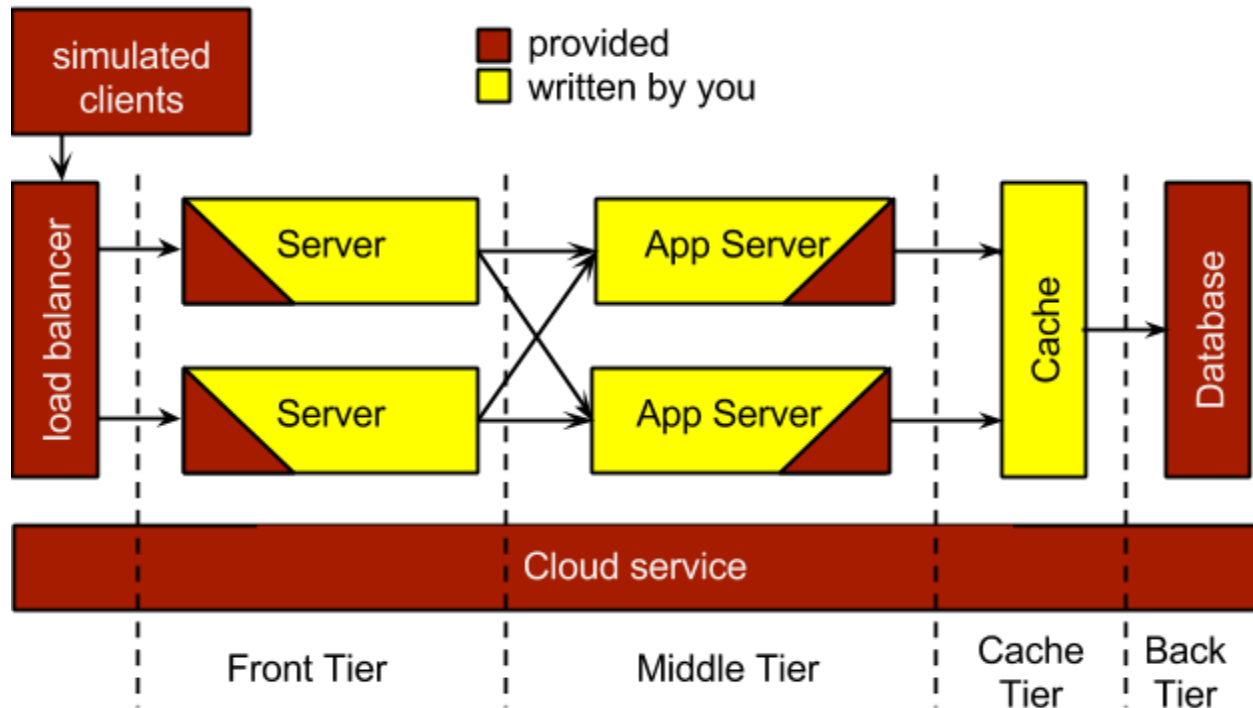
Introduction

A critical advantage of Cloud-hosted services is elasticity, the ability to rapidly scale up a service without needing to purchase and install physical hardware. Cloud providers allow tenant services to add additional virtual servers on-demand, enabling them to meet changes in load (rate of arriving client requests). In this project, you will implement various techniques to scale up a simulated, Cloud-hosted, multi-tier web service (a web store front).

We will provide a simulated cloud service. This provides methods to start, stop, and get status of “VMs”. In our simulation these VMs are actually processes that run the Java server class you will write. The cloud service also provides a simulated load balancer – once your servers register with this, it will deliver client requests to all of the servers in a round-robin fashion (i.e., a request is given to the next server in line, after which the server moves to the end of the line).

The web service will be a simulated online store. Your server needs to handle requests from clients. These can be either a browse request (to provide information on items and categories of items), or a purchase request to buy an item. We will provide a `ServerLib` class that handles most of this for you -- a simple “`processRequest`” method will handle generating and sending back replies to the clients. Your `Server` class needs to instantiate the `ServerLib` class and then register as a frontend with the cloud service (in order to receive requests). The main loop of your server needs to pull the next request from the input queue, and call the `processRequest` method of `ServerLib`. `ServerLib` itself will get the information needed to process requests from a “database” (also hosted in the cloud) that contains the lists of items, categories, prices, etc. This database serves as the back-end tier of the service; your `Server` class will form the front-end and middle tiers.

The system will simulate clients arriving at random intervals. Each client will perform one or more “browse” requests, and may follow up with a purchase, but only if the replies are received within a short time. Your goal is to maximize the total revenue by ensuring short response times, while minimizing costs (number of VMs running).



Requirements/Deliverables

We will provide:

- We provide several components for this project. These require that the project be implemented in Java, in a Unix / Linux environment, e.g., Andrew Unix servers
- We will provide the simulated cloud service that provides interfaces for starting and stopping “VMs” (these are actually implemented as processes).
- We will provide a code to simulate requests from clients
- We will provide a simple database populated with categories, items, prices, etc. for the web store.
- We will provide you a Java class (ServerLib) that provides methods to access the Cloud VM services, to get the next client request, and to process a client request. This will handle interpreting the requests and generating the replies.

You will create:

- You will create a Server class that implements a main() method. This server should instantiate ServerLib, and use it to get and process requests from clients.

Your code should do the following:

- Your Server class will implement a main() method. This will be run by the Cloud class as a separate process once at the start, and again for each call to the startVM method.
- Your Server needs to be able to operate in up to 3 different roles: as the front end (pulling requests off of the input queue and sending them to the middle tier), as an app

server / middle tier (actually handling the requests using the processRequest method of ServerLib), and as a cache (cache read-only operations on the database). Note that the cache aspect is only needed for the final submission. For the first checkpoint, each server instance will do the front and middle tier operations.

- Your Server class needs to use Java RMI to communicate between instances acting as front and middle tiers, to implement the caching proxy to the database, and to coordinate with each other the roles each instance will play. For the database cache, you will need a class that extends RemoteUnicastObject, implements the Cloud.DatabaseOps interface, and passes cache misses or modify operations to the actual database object, which you can get using getDB().
- Your Server needs to call startVM in ServerLib to launch an additional Server instance. A single Server instance will be launched at the beginning of a run, but your code needs to explicitly request any additional “VMs” of your server to be started.
- Your Server should exit cleanly if it is no longer needed, or another instance should use stopVM in ServerLib to forcefully terminate such instances. Servers that don’t terminate will continue to use resources and accrue “costs” in the cloud service.
- Your Server needs to ensure most clients don’t time out. Clients will expect browse requests to be serviced within 1 second, and purchases within 2 seconds. Different operations will incur various time costs, including getting a request from the input queue, dropping a request, processing a browse request, and processing a purchase request. There will also be a fixed time delay for new Server instances to start (booting the “VM”). Each database operation also incurs some time overhead.
- Your server will be provided two command line arguments: <cloud ip> and <port>. These specify the IP address and port of the Java RMI registry provided by the Cloud class. Your code will pass these to the constructor of the ServerLib class, which will mediate access to the Cloud service.

Submission and grading:

- You will be graded on the correctness and performance of your system. The number of clients that timeout or are explicitly dropped will be assessed, as will the total revenue of the web store, and costs (total VM time).
- This project will use an autograder to test your code. See below on how to submit.
- You need to submit a short (1-2 pages) document detailing your design. See below.
- The late policy will be as specified on the course website, and will apply to the checkpoints as well as the final submission
- Coding style should follow the guidelines specified on the course website

Checkpoint 1 (15%)

Due: 5:00 PM, Friday March 27, 2015

Checkpoint 1 requires you to implement a simple scale-out of the “web server”. Here, the system is two-tiered: your server is the frontend, the database (accessed by ServerLib) is the backend. There is no cache tier, and the middle tier functions are performed by the frontend. Each server is monolithic – i.e., like the sample code, the main loop takes the next request off

of the queue and processes it immediately. Your code will need to launch several of these server “VMs” to meet the client demand. For this checkpoint, you can statically decide how many servers to launch based on “time of day” reported by the simulation. You do not need to react dynamically to load changes. However, your code does need to be able to coordinate which server is responsible for starting the other VMs, etc. You will be graded on how well the demand is met -- i.e., the number of clients that succeed, timeout, or are dropped, as well as total revenue and costs of VMs will be accounted.

Checkpoint 2 (35%)

Due: 5:00 PM, Friday April 3, 2015

Checkpoint 2 requires you to implement a 3-tier system. In this simulation, performing both the front end operations (managing the input queue) and application / middle tier operations (processing the requests) in the same “VM” is less efficient than performing these in separate ones. So, to take advantage of this, your Server class should split these operations into separate “VMs.” You will need to start multiple instances (using the startVM mechanism) and use Java RMI to pass requests between the front and middle tiers and to coordinate the roles of your Servers. You should scale out the front and middle tiers separately to minimize the number of VMs needed to meet the demand. You will be graded on how well the demand is met -- i.e., the number of clients that succeed, timeout, or are dropped, as well as total revenue and costs of VMs will be accounted.

Final (50%)

Due: 5:00 PM, Friday April 10, 2015

The final submission needs to implement a 4-tier system, adding a caching tier to your Checkpoint 2 solution. Your checkpoint 2 solution should be able to scale nicely until the database itself becomes a bottleneck. To work around this, you will add a write-through cache that handles most of the database read operations for the “browse” requests, saving database processing capacity for the purchases. Your cache should implement the same interface as the database, and pass through purchase transactions and any misses. Your server can tell ServerLib to use your cache by invoking a second variant of the processRequest method that takes a database interface object reference as a parameter. Your server code will need to coordinate all of these roles and pass requests and database queries between “VMs” using RMI. Grading will be similar to Checkpoint 2, but with higher and more variable client loads. Your code should dynamically scale up and down the number of server “VMs” based on the observed load. (30%)

You will also need to write and submit a 1-2 page document, describing the major design aspects of your project, including how you coordinate the roles of the different server instances, how you decide how many in each tier to run, when you decide to add or remove servers, and how you implemented the database cache. Highlight any other design decisions you would like us to be aware of. Please include this as a PDF in your final tarball. (10%)

Your final source code will also be graded on clarity and style. (10%)

Submission Process and Autograding

We will be using the Autolab system to evaluate your code. Please adhere to the following guidelines to make sure your code is compatible with the autograding system.

First, untar the provided project 3 handout into a private directory not readable by anyone else (e.g., ~/private in your AFS space):

```
cd ~/private; tar xvzf ~/15440-p3.tgz
```

This will create a 15440-p3 folder with needed libraries, classes, and test tools. You should create your working directory in the 15440-p3 directory. It is important that from your working directory, the provided java classes should be available at ../lib.

Write your code and Makefile in your working directory. You must use a makefile to build your project. See the include sample code for an example. You will need to add the absolute path of your working directory and the absolute path of the lib directory to the CLASSPATH environment variable, e.g., from your working directory:

```
export CLASSPATH=$PWD:$PWD/../lib
```

Ensure that by simply running "make" in your working directory, your server and support classes are built. Please name the class implementing your server "Server". Make sure the java and generated .class files are in your working directory (i.e., not in a subdirectory). Your server class should implement main. Do not place your classes in a java package! Leave them in the default package. This naming convention and relative file locations are critical for the grading system to build and run your programs.

To hand in your code, from your working directory, create a gzipped tar file that contains your make file and sources. E.g.,

```
tar cvzf ../mysolution.tgz Makefile Server.java
```

Of course, replace these with your actual files, and add everything you need to compile your code. If you use subdirectories and/or multiple sources, add these. Do not add any files generated during compilation (e.g. the .class files) -- just the clean sources. Also, do not add the class files that we have provided -- these will be installed automatically when grading.

You can then log in to <https://autolab.cs.cmu.edu> using your Andrew credentials (see link at bottom of the log in page). Submit your tarball (mysolution.tgz in the example above) to the autolab site. Note, each of the checkpoints show up as a separate assessment on the Autolab course page. For your final submission, include your write up as a PDF document in your tarball.

How to Use the Supplied Classes

Cloud class

We will provide a class called Cloud. This is the main class for the simulated service, and will start the Java RMI registry and all of the other classes and processes. To run the program, ensure your CLASSPATH is set correctly (and includes both the lib directory and the directory with your Server class), then execute:

```
java Cloud <port> <db_file> <rand_spec> <hour>
```

Here, <port> specifies the port that the Java RMI registry should use. The <db_file> parameter specifies a text file that will be loaded as the contents of the backend database. A sample file (db1.txt) is provided in the lib directory. The <hour> parameter is the time of day to be reported by the simulation (0-24).

Finally, rand_spec indicates the arrival pattern for the simulated clients. There are a few options:

- **c-xxx-sss** - Constant arrival rate with interarrival time xxx ms, and random seed value of sss.
- **u-aaa-bbb-sss** - Uniform random interarrival times, between aaa ms and bbb ms; random seed value of sss.
- **e-aaa-sss** - Exponential random interarrival times, mean of aaa ms (Poisson arrivals); random seed value of sss.

Once started, the Cloud class will start a Java RMI registry and register itself. It will then start a “VM” (actually a process) for the database, which will be initialized with the contents of the atomifile supplied on the command line. Next, a single process is started that executes your Server class. Finally, the simulated clients are started. The simulation will run for 30 seconds, and the client results, total revenues, and total VM time used are reported. Client results include: failed to connect, timeout, explicitly dropped, made purchase, ok (everything went well, but no purchase made), and bad purchase (client timed out, but purchase went through anyway). You want clients to result in either purchase or ok status.

ServerLib class

The ServerLib class needs to be initialized by your server, and provides methods to access to all of the Cloud services and to handle requests. Its constructor needs the IP address and port of the Java RMI registry set up by the Cloud, and provided to your Server class as command line arguments.

Cloud / VM operations:

- startVM() - launch another “VM” that executes your Server class; returns VM id
- endVM(id) - force stop VM indicated by id; your Server process can also just exit cleanly to stop the associated VM
- getStatusVM(id) - returns status of VM indicated by id; this can be NonExistent, Booting, Running, or Ended

- `getTime()` - returns the simulation time of day (hours, 0-24)
- `getDB()` - returns the remote object implementing the database

Front End operations:

- `register_frontend()` - register with load balancer and start receiving client requests
- `unregister_frontend()` - stop receiving requests; this is needed before your Server changes roles or exits so requests are not lost
- `getNextRequest()` - gets the next client request from the input queue of this Server
- `getQueueLength()` - returns number of requests in the queue
- `dropHead()` - drops the next request in the queue
- `dropTail()` - drops the last request in the queue

Middle tier operations:

- `processRequest(r)` - handle request `r`, using the default database, and send back reply
- `processRequest(r,db)` - handle request `r`, but using the provided `db` (e.g., your cache), and send back reply
- `drop(r)` - drop the request `r`

Database

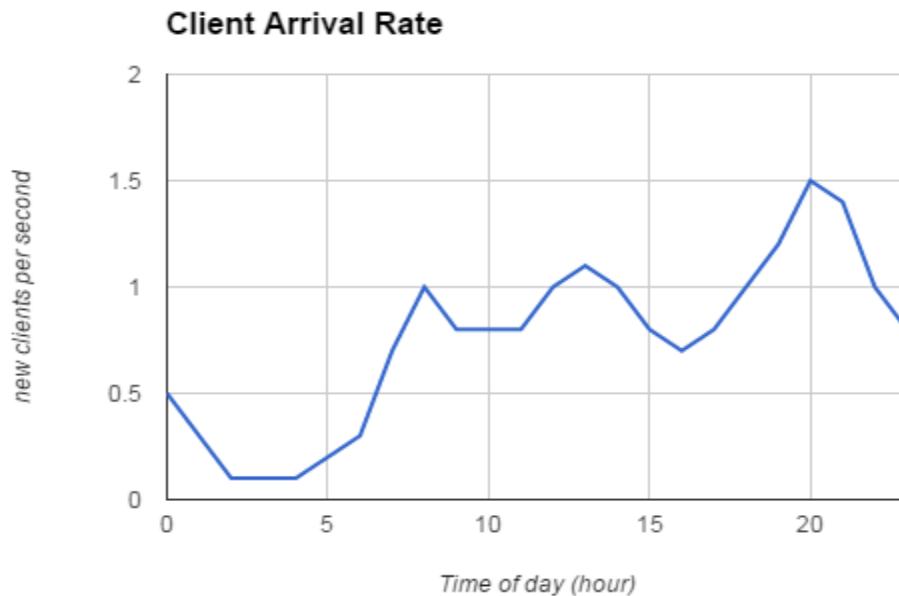
The database used in this simulation is actually a key-value store. The interface is defined in `Cloud.DatabaseOps`. This provides 3 methods:

- `get(key)` - returns the value (string) associated with the supplied key (also a string)
- `set(key, value, auth)` - inserts the key value pair; this is a restricted operation that requires a password (auth string)
- `transact(item, price, qty)` - purchase specified item at the specified price and quantity; returns true on success, or false otherwise

Your Server generally does not directly use the database. Instead, the `processRequest` method of `ServerLib` will access the database. However, for the final submission, you will need to implement a cache for the database that implements this interface. You should cache read operations (`get`), and pass through any misses or write operations (`transact`) to the database.

Diurnal load curve

The number of clients accessing web sites typically varies by the time of day. Often, this is a regular pattern based on sleep, work, and meal times of customers. For this project, assume the diurnal load cycle below. You can tune the number of Servers you use based on the “time” reported by the simulation (`getTime()`). However, be aware that the actual load can vary from the average in this chart, so you should use this as a starting point, but adjust the number of Servers based on the observed load, queue lengths, etc.



Notes / Hints

- You should try running your system at various loads (adjust the rand-spec value to change the arrival rate of clients) to determine how many clients you can handle with different number of servers.
- You should use Java RMI to communicate between different instances of your Server class. You do not need to create your own registry, though -- you can use the one set up by the Cloud class.
- If you have instantiated an object that has an RMI interface (ie., it extends `UnicastRemoteObject`), your process will not terminate cleanly unless you stop the object's remote interface. You can create a shutdown method for your object that calls:

```
UnicastRemoteObject.unexportObject(this, true);
```

This will turn off its RMI server, and let the process terminate.

- Hint: Can you use RMI registry `bind()` to help figure out if your particular server instance is the first one running?
- Hint: You should use drop if there is no way to handle the client in time. This can be used to prevent bad purchases (client times out but the purchase goes through anyway), or to drop some requests when the queue is too long (and would cause every client to timeout).