15-440 Spring 2025
Project 3: Implementation and Performance Tuning of a Scalable Web Service

Important Dates:

Project Handout:	Tuesday March 11, 2025
Checkpoint 1 due:	Tuesday March 18, 2025, 11:59 PM EST
Checkpoint 2 due:	Tuesday March 25, 2025, 11:59 PM EST
Final Due:	Tuesday April 1, 2025, 11:59 PM EST
Submission Limits:	**10 Autolab submissions** per checkpoint without penalty (5 additional with increasing penalty)

You will learn to:

1. Identify potential bottlenecks in a distributed system.
2. Devise experiments to confirm the nature of the current bottleneck.
3. Devise techniques to alleviate the current bottleneck.
4. Understand resource versus performance tradeoffs.
5. Identify scaling signals.
6. Experience multidimensional optimization with multiple parameters.
7. Cope with the nondeterminism that affects even very simple distributed systems.

> *Unlike the other three projects, unpredictability and non-determinism are at the heart of this project. This is how real distributed systems behave. Even without any code changes, your implementation may exhibit wildly different results from run to run. This is not a bug in the project specification or the supplied code base. Rather, it is inherent in this kind of work. Learning to optimize a distributed system in spite of this annoyance is a skill that we want you to learn.*

Introduction

A critical advantage of cloud-hosted services is *elasticity*, the ability to rapidly scale out 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 out a simulated, cloud-hosted, multi-tier web service (a web storefront). You will then evaluate

the system to understand what bottlenecks manifest at different loads, and use this information to decide when to scale out which components in order to improve performance. This is a critical skill to develop, as even a very simple distributed system (as implemented in this project) can have very complex performance behavior.

This project requires you to consider two types of scaling to meet client demand. First, you will look at scaling out a service by running multiple servers. Next, you will split the service into multiple tiers to improve performance. These tiers can themselves be scaled out independently. A critical question is: when should one scale out a tier by adding servers? This depends very much on the characteristics of the application itself. For example, at a given load, if you determine that the bottleneck limiting performance is the middle tier, then adding more servers for the middle tier will help. However, by doing this, you will likely shift the bottleneck elsewhere in the system, so adding even more middle tier servers will not help. Perhaps the bottleneck is now in the front-end, so you should consider adding more front-end servers. Or perhaps the bottleneck is now the database; in this case adding more servers to either front or middle tier will not help.

You will need to run experiments to benchmark the service at varying conditions to find the bottlenecks and determine the optimal number of servers (in each tier) for a given load (client arrival rate). This is sufficient for optimally scaling the service when the load is predictable or does not change (checkpoint 1). However, for dynamic or unexpected workloads as seen in the real world (and checkpoints 2 and 3), your code will need to monitor the system at run time, and add or remove servers as needed. Minimizing the number of servers used to handle a particular workload is important, as these resources are not free. Finally, this project also requires you to deal with the real-world issue of nondeterminism: each run may be slightly different even for the same input conditions.

We will provide a simulated environment for your service to run in. It is implemented in Java and your service must also be implemented in Java (see below for details). You are free to use any language you like to automate and analyze your performance experiments. You should turn in all of the code you write. This project only requires a little actual implementation work; expect to spend the majority of your time on testing and developing tools to find bottlenecks and sources of inefficiency.

Your service is, notionally, an online store. Your code receives two types of requests from clients: browse requests, asking for information on available items and categories of items, and purchase requests, asking to buy an item. We provide a `ServerLib` class that handles the details of request processing; the code *you* must write is the service's main loop. A serial (one client at a time) version of this loop would cycle among three operations:

1. Waiting for a client to connect (`ServerLib.acceptConnection` method).
2. Reading a request from the client (`ServerLib.parseRequest`).
3. Processing the request and sending the response (`ServerLib.processRequest`).

Because of the way HTTP works,[1] each client connection can transmit only one request. To make a sequence of requests (e.g. browse followed by purchase) the client must connect multiple times. `processRequest` takes care of closing the client's socket after it sends the response to that request.

The simulator generates requests from (simulated) clients following a semi-randomized pattern (see the discussion of *<rand_spec>* under "How to Use the Supplied Classes"). Each client makes a series of connections, and sends one or more "browse" requests, possibly followed by a "purchase" request. If any client request is dropped, or takes too long, or results in an error, that client will give up on further requests and we say it has "left the store unhappy." Your goal is to minimize the number of "unhappy" clients, while simultaneously minimizing the amount of cloud resources consumed. To achieve this, you cannot use a simple serial main loop; you must scale to meet the load.

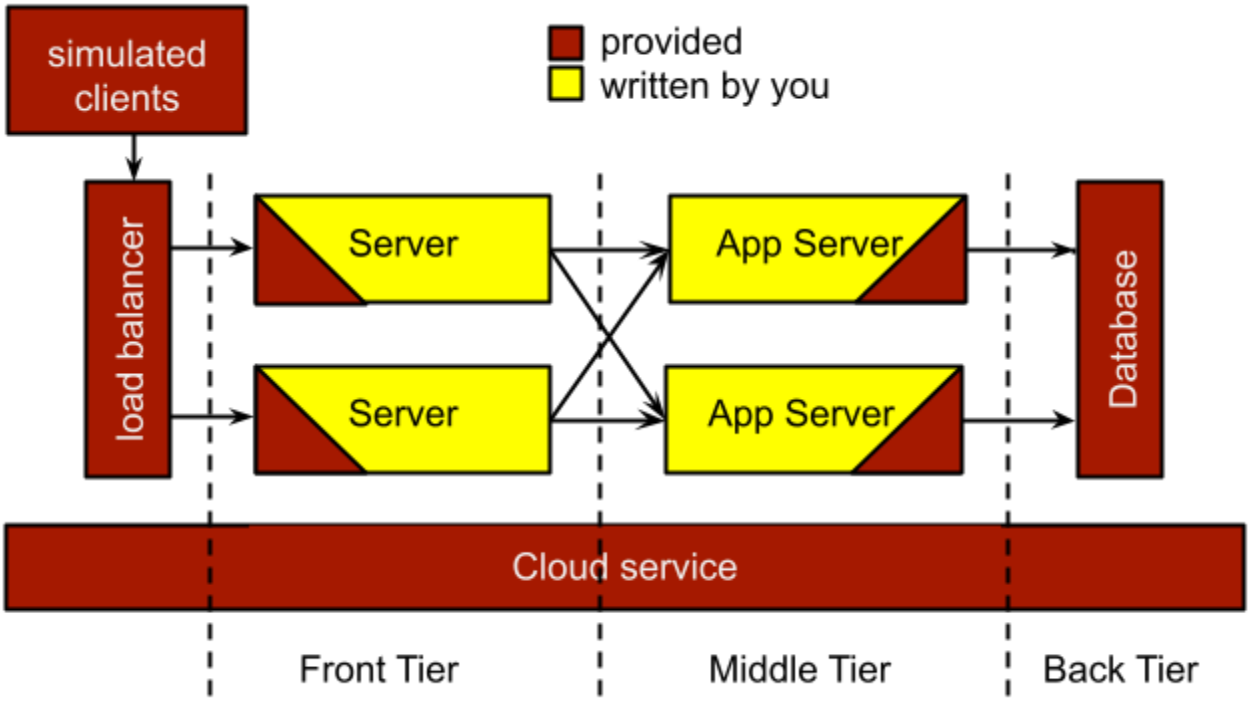
The simulated environment provides a simple "cloud services" API, with methods to start, stop, and get the status of simulated virtual machines (VMs). At the beginning of the simulation, there will be two VMs running: the simulator itself (VM id #0) and the initial instance of your server (VM id #1). Your server can start more VMs, each of which will also run your server's code. The environment also provides a "back-end" database, which you will not have to interact with directly, and a load balancer that receives requests from the simulated clients and spreads them across all of your front-end VMs.

Unlike the previous two projects, it is possible to run the simulator on any computer with a Java runtime. We still recommend that you do your work on the Andrew compute cluster (`unix.andrew.cmu.edu`). Since the load on your service is simulated, it is fine to run experiments on the Andrew cluster even when many other students are using it. In the past, students have successfully completed this project by working on a *native* installation of GNU/Linux on one of their own computers, instead of the Andrew cluster. However, working within some other operating system (e.g. macOS, Windows, or Linux in a hosted virtual machine) tends to make your performance measurements disagree with Autolab's measurements.

Remember that Autolab has the final say regarding your code's performance, and that course staff can only help you with problems that can be reproduced on the Andrew cluster or within Autolab.

[1] Newer versions of HTTP have relaxed this restriction, but our simulated environment does not implement them.

Requirements and Deliverables



We will provide:

- The simulated cloud service, which provides a mechanism for starting and stopping "VMs" (these are actually implemented as processes), a load balancer, a database, and a Java RMI registry.
- The simulated workload (generation of requests from clients).
- The simulated responses to client requests.
- A Java class, called `ServerLib`, which provides methods for accessing the simulated cloud services, registering with the simulated load balancer, and receiving and processing client requests.

You will create:

- The server main loop. The simulator expects this to take the form of a Java class called `Server`, which provides the standard Java program entry point (`public static void main(String[] args)`). In this `main` function, you should instantiate `ServerLib` and use it to get and process requests from clients. Each checkpoint requires you to implement the main loop somewhat differently; see below for details.
- Benchmarking experiments, to determine the most efficient way to scale your server to the simulated load in each phase of the project.

- Reports on your design and the results of your benchmarks.

Your server should do the following:

- As described above, you must provide a `Server` class with a program entry point method (main). The simulator will invoke this entry point (in a separate process from the simulator itself) once at the beginning of the simulation, and again (in another separate process) each time you call `ServerLib.startVM`.

- Each time your main is called, it will receive three command line arguments. The first two specify the IP address and port of a Java RMI registry provided by the simulator. You must pass these down to the `ServerLib` constructor. You may also use the registry directly, to create your own inter-VM RPC services. The third argument is a positive integer, the "VM id" of this instance of your server. Each invocation of main is guaranteed to supply a VM id that is unique among all currently running simulated VMs.

  Every invocation of main will receive the same set of arguments. The VM id will be different for each instance, but you cannot, for instance, supply additional arguments to main when you call `startVM`.

- Your server should automatically start additional VMs as needed to handle the load. Your goal, as described above, is to ensure that most clients don't time out. Clients will expect browse requests to be serviced within 1 second, and purchases within 2 seconds.

  To simulate the work that needs to be done in the front and middle tiers of a real web application, both `parseRequest` and `processRequest` are very slow. It is not possible to handle the simulated load using only a single VM. Starting new `Server` instances with `startVM` is also slow, to simulate the costs of booting an entire virtual machine.

- Your server should also automatically *stop* VMs that are not currently needed, to conserve resources. You can either have the `Server` main loop shut itself down, or you can have another instance use `ServerLib.endVM` to terminate extra instances.

  The ServerLib API has been designed to make it easy to tell whether a VM is idle. VMs that are waiting for acceptConnection to return are idle. VMs that are waiting for another VM to call a RMI remote object method are also idle. VMs that are doing simulated work inside parseRequest or processRequest, on the other hand, are busy.

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, along with the total cloud 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 plot (pdf) of benchmarking results with Checkpoint 1. See below.
- You need to submit a short (1 or 2 pages) document detailing your design with Checkpoint 3. 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 (17%) Due: 11:59 PM, Tuesday March 18, 2025

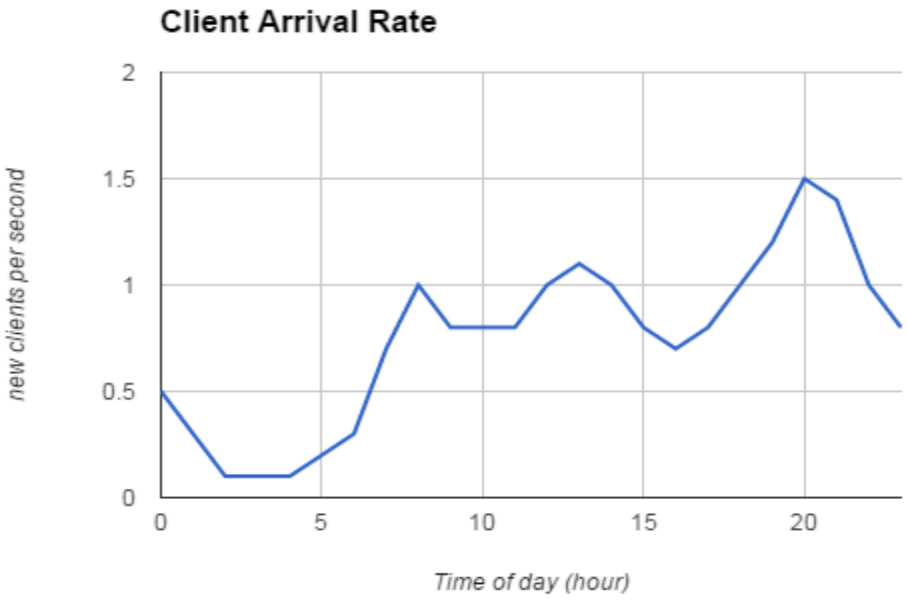
In checkpoint 1, you will implement basic horizontal scaling-out of the “web server.” The simulated load will be light enough that you won’t need the three-tiered architecture shown in the figure at the top of “Requirements and Deliverables.” Instead, your Server class can implement all three of the steps of the service main loop within a single “VM.” However, you *will* need to be able to launch additional VMs as required to meet client demand; a serial implementation, as described in the introduction, will not be enough.

You will be graded on the number of unhappy clients (those whose requests are dropped, take too long, or incur other errors), as well as the total resources (VM time) consumed by your system. (12%)

As performance testing is an important aspect of designing a scaling system, you will also be graded on a simple benchmark. Run your system on a trace described by a `rand_spec` of “`c-1000-`*`sss`*” (see “How to Use the Supplied Classes”, below) where *sss* is any random seed you wish. First run with just 1 server and measure the number of unhappy clients. Then repeat for 2, 3, 4, … servers. Create a plot, where the number of unhappy clients is on the Y-axis and the number of servers on the X-axis (make sure to label the axis of your plot and ensure that it is readable). Submit a PDF that contains this plot and a one-paragraph description of results to autolab. Be sure to note the shape of the curve, any “knee” or bend in the curve, and explain what the plot suggests is a good number of servers for this load. (5%)

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 checkpoint 1, test loads will be constant over each run of the simulator (but will vary from run to run) and can be predicted based on the diurnal load plotted below. Therefore, for this checkpoint, you do not *need* to implement dynamic load scaling. (You can if you want, though!) Instead, you can check the “time of day” reported by the simulation (`ServerLib.getTime`) and use that to determine what the average load will be. You will need to do experiments to discover the optimal number of VMs to use for any particular arrival rate. Keep in mind that the chart shows only the *average* arrival rate of new clients, and that each client may perform several requests.



Checkpoint 2 (35%) Due: 11:59 PM, Tuesday March 25, 2025

Checkpoint 2 increases the challenge in two ways. First, the load will no longer be predictable based on the time of day, although it will remain at the same rate over any one test run. Therefore, you will need to implement dynamic scaling to adapt to the load you observe in each test run. Second, the load will be high enough that you *will* need the three-tier architecture shown in the figure at the top of "Requirements and Deliverables." Specifically, because both `parseRequest` and `processRequest` are very slow, it will be more efficient to have a "tier" of VMs that call `acceptConnection` and `parseRequest` but not `processRequest`, and a second tier of servers that only call `processRequest`. You will need to use RMI to pass requests from the first tier to the second tier, and you will need to scale out each tier to meet the load experienced by that tier.

Remember that you should not use Autolab as an oracle. You should test your code with your own performance experiments, and only submit to Autolab when you are confident that your code can handle anything the autograder might throw at it, within the limits described above.

As with checkpoint 1, you will be graded on the number of unhappy clients (those whose requests are dropped, take too long, or incur other errors), as well as the total resources (VM time) consumed by your system.

Final (48%) Due: 11:59 PM, Tuesday April 1, 2025

Checkpoint 3 increases the challenge again: not only will the autograder use average loads even higher than in checkpoint 2, the load will no longer remain at the same level over any one test run. Your scaling logic will need to react “on the fly” to changing conditions. The average client arrival rate might go either up or down (or even oscillate!), so you will need to be able to start *and* stop VMs to match the observed load.

Again, you will be graded on the number of unhappy clients (those whose requests are dropped, take too long, or incur other errors), as well as the total resources (VM time) consumed by your system. (28%)

In your final submission for this checkpoint, 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, and when you decide to add or remove servers. You are encouraged to include any plots from your benchmarking that indicate how many servers are needed for different arrival rates. Discuss what you have learned about scaling a service by adding tiers and by scaling out the tiers. Highlight any other design decisions you would like us to be aware of. Please include this as a PDF in your final tarball. (10%)

The code in your final submission will be graded for 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` directory with needed libraries, classes, and test tools. In the `doc` subdirectory, you will find detailed documentation in HTML format for the provided Java classes.

Create a subdirectory of the `15440-p3` directory; this will be your working directory. You can name it anything you like, but you must do your work in a subdirectory; the autograder will not be able to build your code if you try to work in the top level of `15440-p3`. Write your code and a Makefile in your working directory. You must use a makefile to build your project. See the included 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
```

The autograder will expect to be able to build your server and all support classes by simply running make in your working directory. Make sure the `.java` and generated `.class` files are in your working directory (not a deeper subdirectory). Furthermore, you must not place your classes in a java package! Leave them in the default package. Finally, as discussed above, your server class must be named `Server` and it must implement `main`. This naming convention and these relative file locations are critical for the grading system to build and run your code.

To hand in your code, **from your working directory**, create a gzipped tar file that contains your make file and sources. It should unpack into the current directory, not a subdirectory. For example:

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

Make sure that you include all of the .java files you have written, and any other files that are relevant for this checkpoint. However, do not include .class files or any other files that are generated by running make. Also, do not include any files from the handout; the autograder already has those files. Remember to include your .pdf writeup for checkpoints 1 and 3.

You might find it useful to add a goal to your Makefile that creates this tar file.

You can then log in to **https://autolab.andrew.cmu.edu** using your Andrew credentials. Submit your tarball (mysolution.tar.gz in the example above) to the autolab site. Alternatively, if you are working on the Andrew cluster, there is a command-line tool named `autolab` that you can use to submit your tarball directly. Each checkpoint shows up as a separate assessment on Autolab.

You are encouraged to use version control to record your progress. However, if you back up your VCS history to a public “forge” (e.g. GitHub), make sure to use a private repository to avoid any possibility of AIVs.

How to Use the Supplied Classes

Cloud class

The entry point for the simulator is the Java class called `Cloud`, provided by us. To run the simulator, ensure your `CLASSPATH` is set correctly (including both the lib directory and the directory with your Server class), then execute:

```
java Cloud <port> <db_file> <rand_spec> <hour> [<duration>]
```

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 (0–23). The optional *<duration>* parameter indicates the number of seconds to run the experiment; the default is 30 seconds.

Finally, *<rand_spec>* indicates the arrival pattern for the simulated clients. This is a code string, which can take several different forms:

c-*xxx*-*sss*
: Constant arrival rate with interarrival time *xxx* ms and random seed value of *sss*.

u-*aaa*-*bbb*-*sss*
: Uniformly random interarrival times, between *aaa* ms and *bbb* ms; random seed value of *sss*.

e-*aaa*-*sss*
: Exponentially distributed random inter-arrival times, mean *aaa* ms, random seed value of *sss*

spec1*,*duration1*,*spec2*,*duration2*,…,*specN
: Use multiple specifications, one after another. *Spec1*, *spec2*, etc. must each be one of the first three forms. *Spec1* will be used for *duration1* seconds, followed by *spec2* for *duration2* seconds, etc. and finally *specN* for whatever time is left in the simulation (notice that this final spec does not have a duration specified). The sum of all the *duration*s should be less than the overall *<duration>* argument.

The simulator will create a Java RMI registry, a database, and a load balancer, and then it will invoke your Server class in a separate “VM” (actually just a separate process). Then it will run the simulation for the specified *<duration>*. Finally, it will print out a report including each client’s status, the total revenue of the store, and the total VM time used.

Possible client results include: failure to connect, timeout, explicitly dropped, “ok” (successful browsing, no purchase attempted), successful purchase, and bad purchase (client timed out but the purchase went through anyway). Your goal is to maximize the number of clients that report either “ok” or a successful purchase, while minimizing the total VM time used.

ServerLib class

The ServerLib class is your interface to all of the simulated cloud services, and also provides methods for receiving and handling requests. You must construct an instance of this class in each of your server VMs. Its constructor requires the IP address and port of the Java RMI registry created by the simulator; these are provided to your Server class as command line arguments.

The APIs provided by ServerLib are outlined below. For more detail, see the JavaDoc files in the doc subdirectory of the handout.

ServerLib methods for simulated cloud services

`int startVM()`

Launch another "VM." In that VM, your Server class's main method will be called, with the same RMI registry address and port, but a unique VM id. Returns the ID of the new VM. Keep in mind that the new VM will not be available immediately; booting takes time.

`void endVM(int id)`

Forcibly stop the VM indicated by *id*. You should only use this method to stop a *different* VM than the one making the call. To stop the VM that would make the call, unexport all of its remote objects and return from main.

`Cloud.CloudOps.VMStatus getStatusVM(int id)`

Get the status of the VM indicated by id. Returns a code from the `Cloud.CloudOps.VMStatus` enumeration; possible values are `NonExistent`, `Booting`, `Running`, or `Ended`.

`float getTime()`

Returns the simulation time of day, in hours and fractional hours. (Range: $[0, 24)$).

There is intentionally no method to get the VM id of the currently running VM. This value is passed as the third argument to your main method when it is started.

ServerLib methods for front-end server operations

`boolean registerFrontend()`

Register the calling VM with the load balancer. Once you have done this, you may call `acceptConnection` to receive the next client connection in the queue. Make sure to call only once per VM. The return value can be ignored.

`boolean unregisterFrontend()`

Disconnect the calling VM from the load balancer. A front-end VM that is about to shut down should call this method, then continue to call acceptConnection until it returns null, and only then exit. If you do not do this, requests may be lost. Make sure to call only once per VM. The return value can be ignored.

`ServerLib.Handle acceptConnection()`

Get the next incoming connection from the load balancer. Returns a handle to the connection; this is an opaque object, which you can think of as a socket descriptor.

`Cloud.FrontEndOps.Request parseRequest(ServerLib.Handle h)`

Read and parse a request from a client connection. Takes ownership of the Handle passed in. *This method takes a long time to complete.*

`int getQueueLength()`

Returns the number of pending connections that the load balancer has assigned to this server.

`void dropHead()`

Drop (intentionally discard) the client connection that the load balancer has assigned to this server and that is *next* in line to be accepted.

`void dropTail()`

Drop (intentionally discard) the client connection that the load balancer has assigned to this server and that is *last* in line to be accepted.

`void dropConnection(ServerLib.Handle h)`

Drop (intentionally discard) a client connection that has been returned by acceptConnection but not yet passed to parseRequest.

ServerLib methods for middle-tier (aka application server) operations

`void processRequest(Cloud.FrontEndOps.Request r)`

Process request *`r`*, send back the reply, and close the connection. *This method takes a long time to complete.*

`void dropRequest(Cloud.FrontEndOps.Request r)`

Drop (intentionally discard) the request *`r`*.

Notes and Hints

1. Remember that the focus of this project is performance tuning and analysis. You should not have to write a lot of code for your `Server` implementation. Instead, expect to spend the majority of your time on testing and developing tools to find bottlenecks and sources of wasted VM time.
2. Remember that a client will go away unhappy if its request was dropped, took too long to get a reply, or if some error occurred. Happy clients are those that end up with status “purchased” or “ok”; all others are unhappy clients. You need to try to minimize unhappy clients.
3. Repeat your experiments (especially for the randomized-arrival workloads). Randomness and high variability can mislead your intuition. Longer runs may also help in this regard. Although by default the simulation runs for only 30 seconds, you can choose to run for much longer by specifying the optional *`<duration>`* parameter to the Cloud class.
4. 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 numbers of servers.
5. 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.
6. VMs that have instantiated objects with RMI interfaces (extending `UnicastRemoteObject`) will not stop upon returning from main. They will continue running until forcibly stopped (by endVM) or until all of the RMI interfaces have been unexported. One way to deal with this is to include “shutdown” methods in each RMI interface, like this:
   ```
   public void shutdown() {
       UnicastRemoteObject.unexportObject(this, true);
   }
   ```
7. The load balancer assigns client connections to servers eagerly—as soon as they come in, not when servers call `acceptConnection`. (Another way to put it is that each server has its own queue of incoming connections.) `unregisterFrontend` does not reassign

any connections that are already assigned to the calling server, it only tells the load balancer to stop sending any more connections to that server. Thus, after you call `unregisterFrontend`, you still need to accept and handle any connections that are already assigned to you.

8. The `drop` methods exist because you might get into a situation where there is no way to handle all the pending clients before they time out. Explicitly shedding load allows you to avoid doing work for clients that you know you cannot serve fast enough.
9. The arrival rate in the diurnal load curve, and as specified in the *<rand_spec>* parameter to the `Cloud` class, refer to arrivals of new clients. This is not the same as the request rate. Since each client may make multiple browse or purchase requests, the request rate will generally be higher than the client arrival rate.
10. The *<hour>* parameter to the `Cloud` class specifies the simulated time of day (just the hour, from 0 to 23). This is the value that getTime will return. It is not used in any other way. It is useful only to let your code know which part of the diurnal load curve to use to select the number of servers. When you test locally, you must provide a *<rand_spec>* value to set the client arrival rate; it will not be set automatically based on the *<hour>* parameter.
11. At the end of a simulation run, the simulator will kill all of your `Server` processes in some arbitrary order. Because of this, your code may get RMI exceptions due to blocking RMI calls that fail due to the server going away. These exceptions are expected and ok if they occur only at the end of a run.
12. The simulated database counts as a VM. It will always be assigned VM id 0, and your initial Server instance will be VM id 1. (We're just telling you this in case you wonder why there is an extra VM, and why your initial VM doesn't get id 0.)
13. Remember that you are scaling only the first and middle tiers. At some point the database may become the bottleneck resource.