Programming Project #2: Experimenting with Proxying, Caching, and Mobility

Assigned: March 15
Due: March 31 by class meeting time (i.e., 10:10am)
The project can be done either individually or in a group of up to three students.

Overview
The purpose of this project is to help you understand some of the issues involved in the design of caching proxies, get hands on experience with Java RMI, and explore how object mobility can be implemented in Java. In this project, you will be designing, building, and evaluating a caching proxy server that communicates with a remote server through Java RMI. You are encouraged to reuse as much as possible the code that you wrote for your first project. If you worked on the first project alone and decided to team up for the second one, you can reuse the code written by any member of your group.

The design that you’ll be implementing is somewhat unrealistic: indeed caching HTTP proxies communicate with remote Web servers through regular HTTP, and they accomplish that by using regular sockets. Nevertheless, you have already gotten plenty of sockets programming experience while working on the first programming project. In addition, today’s Web servers do use RMI and other remote method invocation mechanisms to communicate with other servers such as application servers. For example, J2EE uses RMI-based communication quite widely.

General Description
• Your implementation will have three parts: client, proxy server, and remote server.
• Start with the code you wrote for the first project: you can completely reuse the client part, and reuse a large portion of the server code in the proxy and remote server parts.
• In your implementation, the client will communicate with the proxy server, and the proxy server will communicate with the remote server. Your client and the main server will never talk to each other directly.
  o The client and the proxy server will talk to each other through HTTP/1.1 using a persistent, pipelined connection.
  o Whenever started, the proxy server should have no resources stored locally (i.e., its caches are empty); when the client requests a resource, be it dynamic or static, the proxy must first fetch the resource from the remote server and cache it locally.
  o While the client talks to the proxy server through regular HTTP/1.1, the proxy must communicate with the remote server through Java RMI.
You can disregard cookies for this assignment.

For all static resources, the caching scheme is very simple: when first requested by the client, they should be fetched from the remote server and then stored locally at the proxy server for all the subsequent requests (from the client).

For dynamic resources (e.g., HTML text generated by executing a servlet), you have to implement the following three strategies:

1. Generate the resource (e.g., a dynamic page) at the remote server, bring it to the proxy server, and return it to the client. (No caching is possible in this case: since pages are generated dynamically, the generating servlet might create different content every time it services a request).

2. Move an instantiated servlet object to the proxy server and use it to service all the subsequent client requests (e.g., generate a dynamic page).

3. Move the servlet class file to the proxy, create a new servlet object instance (i.e., call the servlet constructor), and use this servlet instance to service all the subsequent client requests.

**Implementation Details**

The bulk of the implementation would be constructing an RMI-based service for the proxy server to request various resources from the server. This service will be very simple and might consist of a single remote class (e.g., extending `java.rmi.server.UnicastServerObject`). A good place to start might be to design a remote interface that defines all the remote methods for requesting different types of resources from the server (e.g., a static resource, a servlet, a Java class file, several resources at once, etc.). The remote server object that provides all the functionality for your service will “implement” this remote interface. If this stage of the project takes you longer than a couple of hours, you are doing something wrong.

As is always the case in distributed computing, the biggest challenge is coordinating programs running in different address spaces, possibly on different machines. One of the issues, you have to decide is how your distributed system communicates through RMI. In sockets programming, it was enough for the client socket to know the IP address and the port of the server socket. In an RMI-based program, one has to start the `rmiregistry` program on each node of the distributed execution. It is also possible to start `rmiregistry` programmatically--you might find the `java.rmi.registry.LocateRegistry` class and particularly its `createRegistry` method useful.

You also have to decide how various resources will be cached at the proxy. Among the issues that you have to decide are what kind of data structures to use to implement these caches efficiently and how to ensure that these data structures are equipped to deal
correctly with concurrent access by multiple threads. The Java API does provide several mapping data structures that you might find helpful. Among them are `java.util.Map` and `java.util.HashMap`. Notice that these mapping data structures by default are not synchronized.

In this project, we will assume that Servlets are mobile resources. So while according to the Servlets spec, the methods `init()` and `destroy()` must be called only once, in your implementation the method `init()` will be also called when a servlet is moved to a new node (i.e., from the remote server to the proxy server), and the `destroy()` method will also be called right before the servlet leaves its current node (i.e., the remote server’s node).

**Evaluation details**

As you can see, this project does not require a significant implementation effort. In fact, most of the time working on this project will probably be spent on evaluation.

**Evaluation setup**

Your experimental setup will consist of running your client and the proxy server on one machine, while the remote server on another machine (most likely in the same lab). You might have some issues with making your proxy server talk to the remote server through RMI because of firewall issues. Java RMI does not use port 80 for communication: by default `rmiregistry` runs on port 1099 and then assigns some random ports to RMI sockets). A representative of the Techstaff in the CS Department has assured me that all the Linux machines in undergraduate labs are not firewalled for intra-domain access. Windows machines are more heavily firewalled, but this still should not be a problem for intra-domain access. You’ll have to find a setup that’ll make it possible for you to run your experiments. Since this is a Java-based project, you should have no problem running your programs on any operating system.

**Experiments**

1.) Pre-fetching. This experiment involves only static resources. Imagine accessing a web site such as `www.vt.edu`. This site has many static resources, and a proxy server would have to request each one of them from the main VT server before serving the resource to the client. One approach is to fetch each resource to the proxy, cache it, and return it to the client only when this resource is actually requested. Another approach is to recognize that `www.vt.edu/index.html` document contains references to other resources (e.g., images, scripts) and have the proxy request them all at once in one batch from the remote server, even before the client has actually requested these resources from the proxy server. In other words, when the proxy is asked to fetch `index.html`, it could fetch this document, cache it, return it to the client, and while waiting for other requests from the same client, foresee which other resources will be requested in the future and pre-fetch them. To make pre-fetching possible, you have to provide a method in your RMI interface that requests multiple resources (e.g., taking an array of names and returning an array of resources). You do not have to parse HTML for this experiment. It’s OK to create a text file that would contain a list of all the resources referenced by a particular document.
and use it in the experiment. For example, at your proxy server, you can have a file called index_html_referenced_resources.txt that would contain all the resources referenced by index.html. By parsing this document, your proxy server can create a pre-fetching request to the remote server.

Design an experiment that investigates the value of pre-fetching. Does pre-fetching make sense or is it better to request each resource sequentially? If pre-fetching is beneficial for performance, then for how many resources? In other words, if you have many static resources, does it make sense to pre-fetch them all at once or to split them into several pre-fetching batches of smaller sizes?

2.) You have three mobility strategies for servlets (no mobility—just getting the generated resource, moving a servlet, and moving the Java class of a servlet—see the General Description section for details). Design experiments to evaluate each of these mobility strategies for servlets. One possibility is to create three different servlets, generating dynamic content of three different sizes—$5n$, $10n$, and $20n$, where $n$ is some unit (for example a line of an HTML document). You can use the code of HelloWorldExample servlet as a basis for your experiment, and, for example, create HelloWorldExample_5, HelloWorldExample_10, and HelloWorldExample_20 servlets. Alternatively, you can pass a parameter to a servlet specifying the document size you want it to generate, but parsing POST requests could be a pain, as you must have experienced).

Since your client has benchmarking capabilities, this experiment should be fairly easy to construct. You can benchmark a scenario that sends a series of, say, 10 GET requests to each servlet. The mobility strategy for your proxy server can be specified as an input parameter to its main method. Which one of the three mobility strategies results in the best overall execution time for each servlet?

**How to turn in**

- Create a jar file named as a concatenation of the first and second names of all the team members and the string “Project1”, separated by underscores. For example, for a team of Eli Tilevich and Zaki Malik, the submission jar would be called EliTilevich_ZakiMalik.jar
- This jar file should contain your report, all the source files, and the client scenario files. Please also include instructions on how to build your source files.
- Because we have encountered many problems with e-mail submissions, we’ll use the Curator system this time. We’ll update you on the specifics.
- Your submission must be originated before the assignment submission deadline: no extensions will be granted.

**Grading**

- 25% your code; 75% your report that describes your design, experiments, and findings.
- Total: 100 points.