Programming is understanding.
- Kristen Nygaard (1926-2002)

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

Assigned: March 24
Due: April 7 by class meeting time (i.e., 12:30pm)
The project must be done in a group of up to three students.

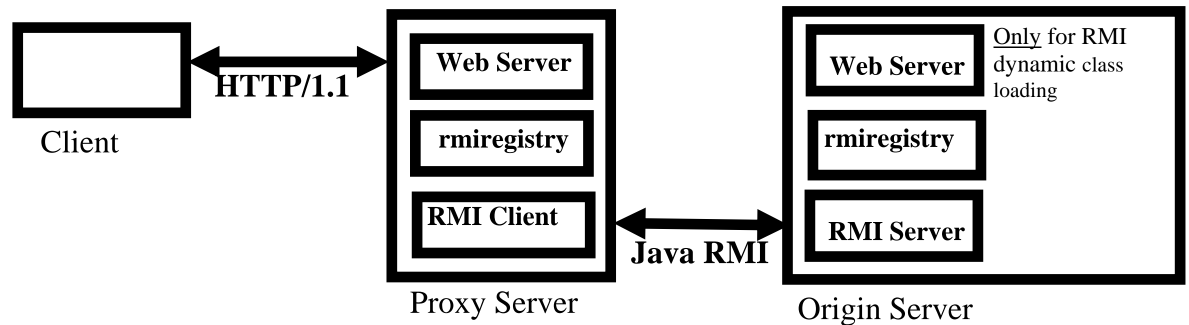
Overview

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 your team composition has changed since the first project, you can reuse the code written by any member of your team.

The design that you'll be implementing is somewhat unusual: caching HTTP proxies communicate with remote Web servers through HTTP, which uses regular TCP sockets underneath. 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 to communicate with application servers (e.g., JBoss, WebSphere, etc.) within the same administrative domain. Specifically, J2EE uses RMI-based communication widely.

General Description

- You will be implementing a 3-tier client server architecture, running on three different hosts: client, proxy server, and origin 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 the origin server parts.
- The client will communicate with the proxy server, and the proxy server will communicate with the origin server. The client and the origin server will never communicate with each other directly.
    - The client and the proxy server will communicate through HTTP/1.1 using a persistent, pipelined connection.
    - Whenever started, the proxy server should have *no* resources stored locally (i.e., its cache is empty); when the client requests a resource, be it dynamic or static, the proxy must first fetch the resource from the origin server and cache it locally to be used for subsequent client requests.



- While the client communicates to the proxy server through regular HTTP/1.1, the proxy must communicate with the origin server through Java RMI.
    - An additional Web server, running at the origin host, will be used for dynamic class loading (stubs and servlet subclasses) in the RMI communication.
    - The Proxy Server and the Origin Server hosts will run a copy of rmiregistry for bootstrapping the RMI communication.
- Please disregard cookies in this project.
- Java servlets do not come ready for mobility. For this project, you *only* have to be able to move servlets that are derivatives of the *HelloWorld* Servlet. Also, you should place a duplicate copy of the *LocalStrings.properties* file at the proxy server, as your mobility mechanism will not have the capabilities required to move this resource file at runtime.
- For all static resources, the caching scheme is very simple: when first requested by the client, they should be fetched from the origin server and then cached 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 origin server, transfer 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 (i.e., by generating 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 to that particular URL.

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 (i.e., 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.

One the most important decisions that one must make when building a distributed system is deciding how to coordinate the execution of multiple programs running in different address spaces on different machines. One of the coordination issues is bootstrapping: deciding how different components of a distributed system can start communicating with each other. In sockets programming, it is enough for the client socket to know the IP address and the port of the server socket. In RMI programming, one has to start the *rmiregistry* name service program on each node of the distributed execution. It is also possible to start *rmiregistry* programmatically—see the documentation for *java.rmi.registry.LocateRegistry* class and particularly its *createRegistry* for details.

Another important design decision that you have to make is how to cache various resources at the proxy. One issue is deciding on which data structure to use for storing cached data. These data structures also have to be able 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, however, that these mapping data structures are *not* synchronized by default.

In this project, we will treat Servlets as mobile resources. This will require slight changes to the Servlets Specification. Whereas according to the spec, the *Servlet* methods *init()* and *destroy()* are called only once in a servlet's lifetime, your implementation should also call *init()* once a servlet object has moved to a new node (i.e., from the origin server to the proxy server), and *destroy()* when the servlet object is about to leave its current node (i.e., running the origin server).

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 should be spent on evaluation.

Evaluation setup

Your experimental setup will consist of running your client and the proxy server on one machine, while the origin 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 *all* 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. Consider accessing a web site (*http://courses.cs.vt.edu/~cs4244/spring.08/*). This site has many static resources, and a proxy server would have to request each one of them from the main CS@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 *http://courses.cs.vt.edu/~cs4244/spring.08/index.htm* document contains references to other resources (e.g., images, style files) and have the proxy request them all at once in one batch from the origin 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.htm*, it could fetch this document, cache it, return it to the client, and while waiting for other requests from the same client, determine which other resources will be requested in the future (by parsing it) 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. Instead you should 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_htm_referenced_resources.txt* that would contain all the resources referenced by *index.htm.* By consulting 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 one at a time? 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--5**n*, 10**n*, and 20**n*, 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 already has benchmarking capabilities, these experiments 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

- The submission instructions are the same as the ones used in the first programming project.
- Create a jar file that contains your report, all the source files, and the client scenario files. Please also include instructions on how to build your source files.
- Your submission must be originated before the assignment submission deadline: no extensions will be granted.

Grading

- 25% your code; 75% your report (6 pages max) that describes your design, experiments, and findings.
- Total: 100 points.