CS433/533 Homework Assignment Three: Network Servers and Server Performance Evaluation

FAQ (last updated 9:02 PM, Oct. 13).

Rubric

This assignment gives you a chance to become familiar with concurrent network clients, servers, covering topics including threads, synchronization, wait/notify (monitor), asynchronous I/O, and benchmarking.

Due:

- Check point [5 points] (Part 1 and Part 2 code): 10 PM on Oct. 11
- All [95 points]: 8:00 PM, Friday, Oct. 18, 2013 by uploading to classes

Protocol

The server that we will design is a simplified version of HTTP 1.0. A server is started with a document directory <DocRoot>. The most basic application message, encoded in ASCII, from the client to the server is:

```
GET <URL> HTTP/1.0
CRLF
```

where CRLF is carriage return and line feed, representing an empty line. This request asks for the file stored at location <DocRoot>/<URL>.

For example, if <DocRoot>=/tmp/mydoc, and <URL> is /file1.html, the server will return the file /tmp/mydoc/file1.html, if it exists.

The basic reply message from the server to the client, encoded in ASCII, is:

```
HTTP/1.0 <StatusCode> <message>
Date: <date>
Server: <your server name>
Content-Type: text/html
Content-Length: <LengthOfFile>
CRLF
<file content>
```

CRLF again represents an empty line. If the file is found and readable, the returned <status code> is 200 and you can give a message such as OK. Otherwise, please give an error code of 400. If you are curious about HTTP error codes, you can see http://www.ietf.org/rfc/rfc1945.txt. You can use Java File class to obtain file size.

Part 1: Simple Client

Your test client should be multithreaded. The client can generate test requests to the server with the following command line:

```
%java SHTTPTestClient -server <server> -port <server port> -parallel <# of threads> -files <file name> -T <time of test in seconds>
```

In particular, the <file name> is the name of a file that contains a list of files to be requested. For example, a file may look like the following:

```
file1.html
file2.html
file3.html
file1.html
```

Then each thread of the client will request file1.html, then file2.html, then file3.html, and then file1.html. The thread then repeats the sequence. The client simply discards the received reply. The client stops after <time of test in seconds>. The client should print out the total transaction throughput (# files finished downloading by all threads, averaged over per second), data rate throughput (number bytes received, averaged over per second), and the average of wait time (i.e., time from issuing request to getting first data). Think about how to collect statistics from multiple threads.

Part 2: Sequential and Multithreaded Servers

In class we have covered multiple approaches to implement network servers:

- sequential;
- per request thread;
- thread pool with service threads competing on welcome socket;
- thread pool with a shared queue and busy wait;
- thread pool with a shared queue and suspension;
- asynchronous i/o.

In this part, you will implement the first 5 approaches and compare their performance using your test client or a browser. You can feel free to reuse the example code provided in class.

Following Apache configuration style (http://httpd.apache.org/docs/2.0/vhosts/examples.html; note that we implement a single server name, not multiple, as the example configuration shows), we program each server by reading a configuration file:

```
%java <servername> -config <config_file_name>
```

The basic configuration parameters are listening port and document root:

```
Listen <port such as 6789>
DocumentRoot <root dir>
```

For a thread pool based server, the configuration file allows specification of the number of threads:

```
ThreadPoolSize <number of threads>
```

Each server uses a cache to speedup handling of requests for static files. The cache is a simple Java Map, with key being the file and content the whole file in an array. Before reading a file from disk, the server checks whether it is already cached. Think: how to handle multiple threads reading and adding to the Map.

The cache size can be specified in the configuration file:

```
CacheSize <cache size in KBytes>
```

To simplify your server, there is no cache replacement; i.e., when the cache is full, no addition to the cache.

You can always specify some configuration parameters in the command line, e.g., java <servername> -ThreadPoolSize <size>. A commandline specification will overwrite the configuration file specification. We recommend that you consider a hash map in your program to implement configurations.

Your server must support the following:

- Methods: The server must support HTTP 1.0 (http://www.w3.org/Protocols/HTTP/1.0/spec.html) GET method.
- Headers: The server must send the Last-Modified header and understand the If-Modified-Since header from client. This means that you will need to parse date format. For this assignment, we use the rfc1123-date format. Your server also needs to understand the User-Agent header. For other headers, your server can skip. Remember that POST has a message body.
- URL Mapping: If the URL ends with / without specifying a file name, your server should return index.html if it exists; otherwise it will return Not Found. If the request is for DocumentRoot without specifying a file name and the User-Agent header indicates that the request is from a mobile handset (e.g., it should at least detect iphone by detecting iPhone in the User-Agent string), it should return index_m.html, if it exists; index.html next, and then Not Found.
- Your server needs to check if a mapped file is executable. If so, it should execute the file and relay the results back to clients. Our assignment only handles the case that the input to the external program is from GET. Please see Java ProcessBuilder on how to start set environment variables and start a dynamic process. The example of the doc can be helpful. You will need to read RFC 3875 to set the right environment variables. You will need to write a dynamic CGI program to test your invocation.
- Your server also needs to implement a heartbeat monitoring URL service to integrate with a load balancer (e.g., Amazon Load Balancer we covered in class). In particular, a load balancer may query a virtual URL (i.e., no mapped file) named `load` (i.e., with request GET /load HTTP/1.0). If the server is willing to accept new connections, it should return status code 200; otherwise, it returns code 503 to indicate overloading. Your software design should allow "plugin", at run time, of different algorithms to compute overloading conditions. Please describe a particular design and implement it.

Part 3: Asynchronous Server

Part 3.1: Your ASync Server

In this part, you implement an asynchronous server with functions as specified in Part 2. We have the following requirements:

- The software structure of your asynchronous server should be based on v3 of the `EchoServer` that we discussed in class. You need to write a handler for the particular protocol. You can feel free to modify the structure if you see any way to improve it (fix error handling, etc). You need to document your changes.
- The server should have an additional timeout thread. Upon accepting a new connection, the accept handler should register a timeout event with the timeout thread with a callback function. The timeout value is specified by `IncompleteTimeout <timeout in seconds>`. The default timeout value is 3 seconds. If the connection does not give a complete request to the server *approximately* within timeout from the time of being accepted, the server should disconnect the connection. Note that the timeout monitoring thread should not directly close a channel that the dispatcher thread is still monitoring (why?). You need to think very carefully about the exact details of the interaction between these two threads, propose a software design, and implement it.

Part 3.2: Comparison of Designs

Please read the source code and document of x-Socket, a high performance software library for reusable, asynchronous I/O servers. Please discuss in your report the following questions (please refer to the line numbers of the source code when discussing the design):

- How many dispatchers does x-Socket allow? If multiple, how do the dispatchers share workload?
- What is the basic flow of a dispatcher thread?
- What is the calling sequence until the `onData` method of EchoHandler (see `EchoHandler, EchoServer`, and `EchoServerTest`) is invoked? Please check this link for testing code: http://sourceforge.net/p/xsocket/code/HEAD/tree/xsocket/core/trunk/src/test/java/org/xsocket/connection/

- How does x-Socket implement Idle timeout of a connection?
- Please give an example of how the library does testing (see http://sourceforge.net/p/xsocket/code/HEAD/tree/xsocket/core/trunk/src/test/java/org/xsocket/connection/EchoServerTest.java for an example). Please describe how you may test your server with idle timeout?

Part 4: Performance Benchmarking

One important computer systems skill is to evaluate the performance of design alternatives. In this assignment, we conduct performance evaluation of the alternatives:

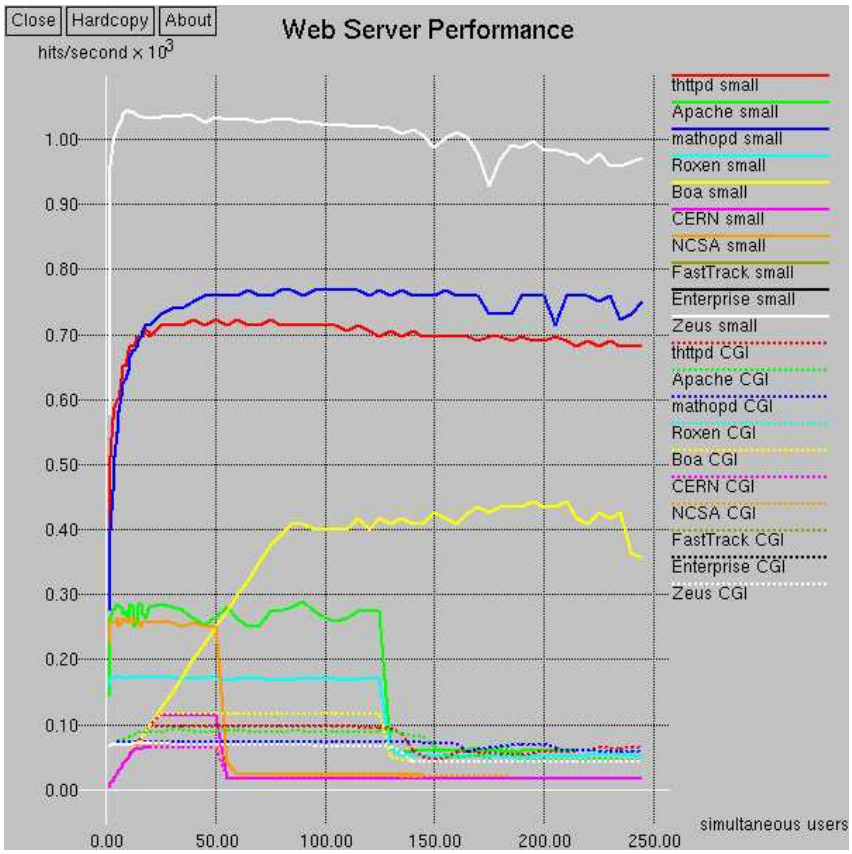
- To conduct the testing, you will need to setup the DocumentRoot at the server. It is highly recommended that you generate a number of files of different sizes under DocumentRoot named such as file1.html, file2.html, ..., file1000.html. If you download gen.tar, and untar it (tar -xvf gen.tar), you will see a directory named doc-root and a directory named request-patterns.

  To compare the performance with Apache, we will use the department zoo Apache server. We will use /home/httpd/html/zoo/classes/cs433/web/www-root to store testing files. Suppose we want to fetch /home/httpd/html/zoo/classes/cs433/web/www-root/html-small/doc1.html. To use the department Apache server, since the department server has set DocumentRoot as /home/httpd/html/zoo, the URL should be:
  http://zoo.cs.yale.edu/classes/cs433/web/www-root/html-small/doc1.html

  To use your server, suppose you set the DocumentRoot as /home/httpd/html/zoo/classes/cs433/web/www-root, and you run your server on cicada.cs.yale.edu at port 9876. Then the URL is:
  http://cicada.cs.yale.edu:9876/html-small/doc1.html
- For the test, you will need to generate a request file for the client. The request pattern can have a major impact on your server performance (how requests repeat). The TA will use a Pareto distribution to generate request patterns to test your server. You can write a simple Java program or script to generate the request.
- You should vary the client parallel (see Client command line above) with a reasonable increment schedule (e.g., 1, 2, 3, 4, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 60, 70, ...). A reasonable test time is 60 to 120 seconds. You can write a simple script to automate this task.
- For multithreaded server, please try two thread pool sizes: one small and one large.

Part 5: Report

- You should submit a report on your server design.
  - Please answer any question we specified above.
  - Please report the measured performance of both Apache and your best server for these performance metrics: throughput and (mean) delay. You can use open office or gnuplot to generate figures. Below is an example figure showing the performance of multiple servers.
  - The TA will benchmark all servers and pick the one with the highest throughput. This server will receive a bonus of 25%.



Submission

- **Please submit using class server. Please include README to tell the TA the directory structure, e.g., which file is the report. Please generate a single jar file containing all of your files.**

Suggestions

During your async i/o design, think how you implement a finite state machine to handle each request (e.g., initial state after accepting a connection, what other states). Java async i/o does not allow you to select events on a file channel. There are can be multiple design options to handle file i/o:

- Use standard file i/o by assuming that file system is fast and will not become bottleneck;
- Try out mapped file i/o:

```
FileInputStream fin = new FileInputStream(args[0]);
FileChannel in = fin.getChannel();
ByteBuffer input = in.map(FileChannel.MapMode.READ_ONLY, 0, in.size());
```

- Try out direct transfer: See FileChannel.transferTo;
- Use standard file i/o and use a thread pool to help with reading files.

References

- Book
  - Java Network Programming, 3rd, 2005, by Elliotte Harold is a good reference book on Java network programming. In particular, you should read chapter 5 on Thread. Yale library has electronic version of this book (you need Yale IP address to gain access). If this link does not work, please try to search Orbis and follow the link. The examples codes can be found here. For details about ByteBuffer, please read chapter 12 of Java Network Programming.
- General java information:
  - Development environments:
    - zoo has Java installed:
    - eclipse provides a nice java IDE.
  - Java book
    - An overall very good book on Java is **Thinking in Java (3rd Edition)** by Bruce Eckel, Dec. 2002, by Prentice Hall. This book is also available online at web sites such as: http://www.bruceeckel.com/.

Last Update: October 1, 2013 09:59:24 PM -0500