6.005 Elements of Software Construction | Fall 2011
Problem Set 5: Finding Prime Factors with Networking

The purpose of this problem set is to introduce you to aspects of Java's I/O and networking API, to help you get started building network applications.

You have substantial design freedom on this problem set. However, in order for your solution to be graded, your solution must not change the name, method signature, class name, package name, or specification of the following:

- factors.client.PrimeFactorsClient.main()
- factors.server.PrimeFactorsServer.main()
- echo.client.EchoClient.main()
- echo.server.EchoServer.main()

Revision: You can modify *throws* clauses as you see fit for the above methods. Do not modify anything else in any of these main methods.

Background

Passing sensitive information over a network is a tricky operation, especially if listeners are ready to intercept your messages. Encryption has become central in protecting your important information. Modern cryptography systems encrypt your data in such a manner so that listeners would have to solve intractable problems, in particular the prime factorization of the product of two very large primes. Finding an efficient solution to factor this number makes it possible to crack the underlying message, exposing the encrypted sensitive data. In this problem set, we will create a solution that will perform brute-force prime factor searching, but distributed over multiple machines.

The **client-server** architecture models a distributed method of computing characterized by two parts, *client systems* and *server systems* . Both systems communicate with one another either over a network or perhaps even on the same system. Accordingly, this architecture is an example of a *distributed system* whereby we use multiple computers to solve a problem.

Client systems often initiate and make the requests to the servers. Generally, clients are seeking some service or have to rely on the resources of a server to handle their needs. These may include access to files, peripherals, or processing power. In this architecture, clients do not share resources with one another (this characteristic is more in tune with the *peer-to-peer* architecture, which lacks a centralized service provider).

Server systems, on the other hand, hosts server programs whose resources are shared with all connected clients. Servers will generally wait for client requests prior to serving out resources.

Common systems that follow the client/server model include email exchanges, web access, and database access.

You may find the Wikipedia article of the Client/Server Model helpful in understanding more of the details.

Before You Begin ...

Before starting on this problem set, please ensure that you have *Telnet* installed. *nix operating systems should have telnet installed by default.

Windows users should first check if Telnet is installed by running the command "`Telnet`" in command line. If you do not have it, you can install it via Control Panel ---> Programs and Features ---> Turn windows features on/off ---> Telnet client.

You can have Telnet connect to a host/port (for example, "localhost:4444") from the command line with "`telnet localhost 4444`".

Alternatively you can open the Telnet program and connect from there with the command "`open localhost 4444`".

Overview

We will be creating two different Client/Server systems, namely an Echo system and a Prime Factors search system.

With the Echo system, our goal is to learn how users, clients, and servers interact with one another under the Java networking framework. The system will be very simple, taking in User input from Standard input, navigating through the entire network system, and finally returning the original User input back to the User via Standard output.

With the Prime Factors system, one of the focuses is to find the prime factors of a given input (which can be potentially very large). We will attempt to do this through a distributed system in the following fashion:

1. Start several servers dedicated to searching for prime factors in a given range of numbers.
2. Start a client that will send requests to these servers.
3. Accumulate the server responses in some meaningful manner.

You should take note that performance is not a criterion on this problem set. Brute-force search for factors is fine. The client is given multiple servers to communicate with and you should use as many servers as given to your program.

Do NOT use multithreading for this problem set. Each Server instance will handle at most one client at a single time. Though we will have multiple Servers running asynchronously, none of them will be run on the same thread.

We will be using the Java networking library for communication between our client and our servers. Please read about Streams (specifically about Buffered Streams), and Sockets before proceeding.

Echo System Specifications

The *Echo system messages* are defined below. Underlined terms are considered terminals in our grammar.

- User-to-Client messages define what the User can input on the standard input stream, consequently what the client reads in.

```
User-to-Client Echo Message Protocol
  Valid-Input := String NewLine
  String      := [^NewLine]+
  NewLine     := \n
```

- Client-to-User messages define what the client outputs on the standard output stream, consequently what the user can read in console.

```
Client-to-User Echo Message Protocol
  Valid-Input := Prefix Space String NewLine
  Prefix      := >>>
  String      := [^NewLine]+
  Space       := " "   // " " is a single space character
  NewLine     := \n
```

- Client-to-Server messages define all messages that the client will send to the server for processing.

  Server-to-Client messages define the processed output for the respective Client-to-Server message.

  There are no special protocols for Client-to-Server or Server-to-Client messages for the Echo system.

```
Client-to-Server and Server-to-Client Echo Message Protocol
  Valid-Input := String NewLine
  String      := [^NewLine]+
  NewLine     := \n
```

The *Echo Server* component is specified below.

- The server takes in at most one integer Program Argument, the port incoming clients will connect through.

    - If there is no Program Argument, default server to listen to port 4444
    - If server fails to listen on the assigned port, terminate the server.
    - In Eclipse, you can specify a Program Argument by creating a new Run Configuration --> Arguments Tab --> Write some port number in Program Arguments (example: 4444).
- Server should only handle ONE client at any given time. **Do not use any multithreading in your solution.**
- When client disconnects, server will listen for new incoming client connections on the same assigned port.
- When receiving one Client-to-Server message, the exact contents of this message are sent back to client in exactly one Server-to-Client message.

The *Echo Client* component is specified below.

- The client takes in exactly one Program Argument, the address of the server.
    - If there is no Program Argument, output a helpful error statement and terminate client.
- Client will read messages passed in by User from the standard input stream (`System.in`).
    - If user's standard input stream is closed, terminate the client.
- Client will push messages back out to user through the standard output stream (`System.out`).
- The message passing and processing are specified as follows:
    1. User-to-Client messages are read in from the standard input stream.
    2. Client passes these messages directly to the Server as Client-to-Server messages.
    3. Client listens for exactly one Server-to-Client messages.
    4. When this message is received, the contents of the message are sent as a Client-to-User message onto the standard output stream.

Problem 1: Setting up an Echo Server

We will first set up a server instance that accepts an incoming client connection and echoes all incoming statements from the user.

In your EchoServer class, there is a main() method where all Server setup logic should go. As with the example provided in the Java tutorial, you should set up a ServerSocket to listen for incoming client connections. Your EchoServer must follow the specifications indicated above.

When a client connects, your server should listen to input messages from the connected client and send it right back to the client over the connection.

Remember, we only require your server to handle one client at a time (NO Multithreading!). However, if the connected client disconnects, your server should go back to listening for any clients that want to connect.

You can manually check your server is working correctly as follows:

1. Run your EchoServer with a single program argument for some port number.
2. Open up telnet and connect to localhost:[port number] (example: "telnet localhost 4444", or "open

localhost 4444")
3. If this connection fails, telnet will notify you.
4. If it succeeds, try typing some input. You should see the exact same input spit back at you.
5. Close telnet, and then use it again to reconnect to the server. The server should work for the second connection as well.

a. [15 points] Implement and test the EchoServer, which reads incoming messages and outputs the same message on the output stream. It should follow all the specifications defined in the earlier section.

Problem 2: Setting up an Echo Client

We will now write a client that can connect to our EchoServer.

In your EchoClient class, there is a main() method where all Client setup logic should go. As with the example provided in the Java tutorial, you should set up a Socket to connect and communicate to a Server.

You will be able to manually check your client is working by having a Server instance running in a process and attempting to connect to it with a new client instance.

The following is an example conversation between your User and EchoServer through the EchoClient, displayed through console:

```
Hello, I am EchoClient!
>>> Hello, I am EchoClient!
No, I am EchoClient!
>>> No, I am EchoClient!
You are exhausting to talk to.
>>> You are exhausting to talk to.
```

Note that all the statements preceded with ">>>" are a result of the message passing through the EchoServer back through the Client-to-User protocol.

a. [15 points] Implement and test the EchoClient. It should follow all the specifications defined in the earlier section.

Problem 3: Finding Prime Factors

We will now implement a Java function to find prime factors.

A rough pseudocode for solving this problem is provided. You may not modify the specifications defined below. However, you can improve on this algorithm however you choose.

```
@requires BigInteger N. such that 2 <= N
@requires BigInteger low, hi. such that 1 <= low <= hi
```

```
@effects finds all prime BigIntegers x
       such that low <= x <= hi AND x divides N evenly.
       Repeated prime factors will be found multiple times.

0. Given BigInteger N, BigInteger low, BigInteger hi:
1.     for x from lo to hi:
2.         if x is prime then
3.             while x divides evenly into N:
4.                 add x to the collection of prime factors.
5.                 N = N/x
```

You may use the BigInteger function `isProbablePrime` to approximate if a number is prime and `remainder` to check if one BigInteger fully divides another. See BigInteger for further details.

The following are a few example input/outputs for your function:

- `(N= 85=5*17, lo= 2, hi=17)        -> [5,17]`
- `(N= 85=5*17, lo= 2, hi=16)        -> [5]`
- `(N= 85=5*17, lo= 2, hi=4)         -> []`
- `(N= 264=2*2*2*3*11, lo= 2, hi=17) -> [2,2,2,3,11]`

Note that your outputs do not have to be in any specific order.

a. [10 points] Implement and test the function to find all prime factors of a number, given the range of values to search through.

Prime Factors System Specifications

The *Prime Factors system messages* are defined below. As before, Underlined terms are considered terminals in our grammar.

-
  ```
  User-to-Client Echo Message Protocol
    Valid-Input := Space N Space NewLine
    N           := [0-9]+
    Space       := (" ")*   // Inner " " is a single space character
    NewLine     := \n
  ```

  You can make the assumption that User input is valid only for N >= 2.

-
  ```
  Client-to-User Echo Message Protocol
    Valid-Input := Prefix Space
                   ( (N Equals Factor (Mult Factor)* NewLine) | Invalid )
  ```

```
    Prefix      := >>>
    N           := Number
    Factor      := Number
    Invalid     := invalid
    Equals      := =
    Mult        := *
    Number      := [0-9]+
    Space       := " "   // " " is a single space character
    NewLine     := \n
```

The following are a few example Client-to-User messages:

- ❍ `>>> 85=5*17`
- ❍ `>>> 1332425524=2*2*17*19594493`

•

```
  Client-to-Server Message Protocol
    Message   := Factor Space N Space LowBound Space HighBound NewLine
    Factor    := factor
    N         := Number
    LowBound  := Number
    HighBound := Number
    Number    := [0-9]+
    Space     := " "   // " " is a single space character
    NewLine     := \n
```

You can make the assumption that the message is valid only for N >= 2.

The following are a few example Client-to-Server messages:

- ❍ `factor 85 2 17`

  means find all prime factors of 85 between 2 and 17.
- ❍ `factor 1332425524 2 16`

  means find all prime factors of 1332425524 between 2 and 16.
- ❍ `factor 1 1 100`

  is NOT valid under this context because N=1.

•

```
  Server-to-Client Message Protocol
    Protocol  := Message*
    Message   := Found Space N Space Factor NewLine
               | Done Space N Space LowBound Space HighBound NewLine
               | Invalid NewLine
    Found     := found
```

```
Done      := done
Invalid   := invalid
N         := Number
Factor    := Number
LowBound  := Number
HighBound := Number
Number    := [0-9]+
Space     := " "   // " " is a single space character
NewLine     := \n
```

The following are a few example Server-to-Client messages:

- `found 85 5`

  means the server found 5 as a prime factor of 85.
- `found 85 17`

  means the server found 17 as a prime factor of 85.
- `done 85 2 17`

  means the server in charge of finding factors of 85 between 2 and 17 is complete.

The *Prime Factors Server* component is defined below.

- Server takes in at most one integer Program Argument, the port incoming clients will connect through.
  - If there is no Program Argument, default server to listen to port 4444
  - If server fails to listen on the assigned port, terminate the server.
- Server should only handle ONE client at any given time. **Do not use any multithreading in your solution.**
- When client disconnects, server listens for new incoming client connections on the same assigned port.
- Its functionality will be defined as:
  - Server will listen for Client-to-Server messages. If it does not fit the protocol defined above, send a Server-to-Client "invalid" message.
  - Server will send a Server-to-Client "found" message for each prime factors x of N such that LowBound <= x <= HighBound.
  - When server has found all such x, it sends the Server-to-Client "done" message.

The *Prime Factors Client* component is defined below.

- Client will take in at least one Program Argument, the addresses of the PrimeFactorsServers your client will query.
  - If there is no provided Program Argument, output a helpful error statement and terminate the client.
- Your client will read messages passed in by your User from the standard input stream (`System.in`).
  - If the User's standard input stream is closed, you should terminate the Client.

- Your client will push messages back out to your User through the standard output stream (`System.out`).
- The entire procedure of message processing is defined as:
  1. User-to-Client messages are read in from the standard input stream, indicating the number to factor.
  2. Client passes Client-to-Server messages, one for each Server. Each message to the Server specifies a range of numbers, the union of which covers all Integers from 2 through `sqrt(N)`.
  3. Client listens for Server-to-Client messages, each indicating a prime factor x.
  4. Client aggregates prime factors and sends the appropriate Client-to-User message. If the client receives malformed messages or data, it will send the invalid message.

Problem 4: Making a Server to do Prime Factor Searching

The intention of this problem set is to solve this problem in a distributed fashion. We can do so by breaking the problem down into smaller subproblems, as alluded to in the previous problem.

We will create a server that takes in messages indicating three numbers N, LowBound, HighBound.

It will respond with a message for every x where x is a prime factor of N and LowBound <= x <= HighBound.

The Client-to-Server and Server-to-Client Message protocols are defined in the Specifications section above.

You can first check that this works by communicating with your server through telnet. A sample conversation may be as follows:

```
factor 264 2 5
found 264 2
found 264 2
found 264 2
found 264 3
done 264 2 5
factor 1 2 5
invalid
what? why??
invalid
```

(The italics may not be rendered in your Telnet. It is simply an indication of output from the Server.)

Your Server does NOT need to respond with factors in any specific order.

a. [30 points] Implement and test the PrimeFactorsServer to listen for incoming Client connections. You should follow all of PrimeFactorsServer specifications defined in the previous section. You should use the function you defined in Problem 3.

Problem 5: Integrating your Client with Multiple Servers

We will now create a client that will make requests to servers it is connected to, to get back all prime factors.

We have to first define our search space. In fact, we only have to look for prime factors between 2 and `sqrt(N)`, as there is guaranteed to be *at most one* prime factor greater than `sqrt(N)`. Accordingly, to extend the algorithm presented in Problem 3, you can do the following:

```
1.     for x from 2 to sqrt(N):
2.         if x is prime then
3.             while x divides evenly into N:
4.                 x is a prime factor!
5.                 N = N/x
6.     if N != 1 then
7.         N is a prime factor!
```

You can find a BigInteger `sqrt` function in the provided `util.BigMath` class.

You should partition your search space in some manner across all servers that your client is connected to.

For example, if your client is connected to three servers, you can simply divide the range [2, `sqrt(N)`] in three even parts, [2, `sqrt(N)/3`], [`sqrt(N)/3 + 1`, `2*sqrt(N)/3`], [`2*sqrt(N)/3 + 1`, `sqrt(N)`]

After deciding how to divide your search space, you must now utilize the connected servers in a concurrent fashion.

You must have the client's connected servers perform its operations concurrently, in different processes, in this problem set. The goal is to have your servers perform its search algorithm over different ranges in *parallel* with one another. It is not sufficient for this problem set to perform the searching algorithm over just one of the provided servers. You must use *all* servers provided to the client to perform the search algorithm.

We can trigger the server to start its operations by sending a correct Server-to-Client message across the connection. This should be done for each connected server. Each server will perform its respective operations upon reading in the respective Server-to-Client message, thus operating in parallel with one another.

When your client has accumulated all of the prime factors, it should display them according to the Client-to-User protocol.

A sample conversation is shown:

```
264
>>> 264=2*2*11*3*2
```

```
        1332425524
>>> 1332425524=2*2*17*19594493
cool!
>>> invalid
```

a. [30 points] Implement and test the PrimeFactorsClient. It should obey the above User-to-Client, Client-to-User protocols when interacting with Standard input/output. It should also obey the Server/Client protocols to send and obtain necessary information.