Assignment 1 of CS-3715

Due on January 22, 2007

Note The assignment must be submitted before 9am, Monday, January 22th in the CS3715 class. The assignment must contain a cover page with the course name, and your student number. Late assignments will not be accepted. For the programming part, source code must be submitted on paper, and with submit-assignment. Source code should be commented in a reasonably informative manner, but not required to be in the javadoc format. For the written part, only hard copy needs to be submitted.

Written Part (5 marks)

- Briefly compare the WWW and Internet. How are they related and how do they differ?

Programming Part (15 marks)

- "Paired chatters"

  Write Java programs to design a protocol where a server is responsible to match up two chatter clients. The server listens on a TCP port for upcoming connections.

  1. If no client is already connected to the server to be paired, the server accepts the connecting client, C_1, and makes it wait for another client. To do that, it sends a `"WAIT"` command to C_1. When receiving `"WAIT"`, C_1 constructs a `ServerSocket` instance to listen on port `p`. Then it sends a message `"CLIENT_PORT $p"` to the server for future clients, where `$p` is the port number of on which C_1 listens for connections.
  2. On the other hand, when a client C_2 seeks a connection with the server, it may already have another client, say C_1, waiting to be paired. In this case, the server informs C_2 the existance of C_1 by sending a message `"PEER_LOC $h:$p"` to C_2, where `$h` is the hostname of C_1 and `$p` is the port number that C_1 is accepting connections. After C_2 receives this message, it seeks a connection to C_1 using the obtained information.

The two clients exchange messages using the dedicated direct socket until either party, say C_1, sends an "end of stream" (Ctrl-D in Linux). In this case, C_1 terminates and C_2 constructs a `ServerSocket` instance to listen on some port `p`. C_2 then sends `"CLIENT_PORT $p"` to the server for future clients.

When you design the client, avoid deadlocks where two clients wait for each other's messages. Remember that each client should respond to the inputs from both the local user and the remote user. A `while` loop invoking `readLine()` of both `BufferedReader`s will create such deadlocks. Instead, use `java.io.BufferedReader.ready()` to test if a buffer is filled and ready to be read.

The programs are simple in the sense that a server can manage at most two clients. However, it is a miniature of the *Napster* (semi-)peer-to-peer file swapping. In Napster, when a client A queries for a file, it resorts to the server for directory lookup. When the server determines that some connected client, say B, does have a copy of that file, it directs A to seek a direct connection with B. Napster is a hybrid of the client/server and the P2P architectures due to the "bootstrapping" functionality of the server. Since the file transfer between Napster clients are carried by HTTP, each Napster client is also an HTTP server. Your programs could certainly be extended so that a server can manage multiple pairs of chatters and each connecting chatter may choose from a list of waiting clients to pair with. This assignment does not require you of such extension, but do think about what you need to implement the extension.

You need to submit the source code for both the server and clients. They must be commented informatively though not necessarily compliant with javadoc. You may want to include a README file if your programs are not intuitive to use.