$Id: asg5-chatter-threads.mm,v 1.33 2010-03-11 14:01:23-08 - - $
/afs/cats.ucsc.edu/courses/cmps109-wm/Assignments/asg5-chatter-threads

1. Overview

In this assignment you will write a program in Java to implement a client/server chatter application. You will be making use of internet connections using sockets. Each process will consist of several threads. Finally, the client will be wrapped in a GUI.

2. Program Specification

The program is specified in the foramt of a Unix **man**(1) page.

NAME

chatter — internet chatter application

SYNOPSIS

chatter [**-g**] [**-@** *flags*] [*hostname*:]*port username*

DESCRIPTION

A server application is run as a process which accepts connections from client applications. Whenever the server receives a message from any client, it forwards that message to all other clients as chatter. There is no enforced limit to the number of clients who may connect.

OPTIONS

All options must precede all operands and may not be chained. There may be no space between an option and its operand.

-g Bring up the client application as a GUI instead of as a command line application. The command line application is the default. It is an error if this option is specified for the server.

-@ *flags*
Debug flags are set. All debug is printed to **stderr**, regardless of whether the application is a GUI or not.

OPERANDS

Operands must be specified exactly. It is an error if they are invalid, too few, or too many.

[*hostname*:]*port*
If only a port number is given, a server process is started. If both a hostname and a port number are specified, separated by a colon, then a client process is started.

username
The username of the client for the purposes of printing in front of the echoed messages. It must be specified for a client, and may not be specified for a server.

EXIT STATUS

0 The program ran successfully to completion and no errors were detected. A broken connection on either the client or server side is noted, but that does not cause an error.

1 One or more errors were detected.

3. Implementation

Begin by studying the Java programs in the **Examples/week8*** directories: They contain examples of the use of threads, client/ server applications using sockets, and some GUI code.

3.1 Makefile

The code in the **code/** subdirectory is not intended to be useful, other that as a hint as to how to construct your **Makefile**. This is shown in Figure 1.

```
 1  # $Id: Makefile,v 1.2 2010-02-23 19:47:57-08 - - $
 2
 3  JAVASRC      = chatter.java client.java gui.java server.java
 4  SOURCES      = ${JAVASRC} Makefile README
 5  MAINCLASS    = chatter
 6  CLASSES      = ${JAVASRC:.java=.class}
 7  INNERCLASSES = chatter\$$options.class \
 8                 client\$$receiver.class client\$$sender.class \
 9                 gui\$$receiver.class gui\$$sender.class \
10                 server\$$receiver.class server\$$sender.class
11  JARCLASSES   = ${CLASSES} ${INNERCLASSES}
12  JARFILE      = chatter
13  LISTING      = ../asg5-chatter.code.ps
14
15  all : ${JARFILE}
16
17  ${JARFILE} : ${CLASSES}
18          echo Main-class: ${MAINCLASS} >Manifest
19          jar cvfm ${JARFILE} Manifest ${JARCLASSES}
20          - rm Manifest
21          chmod +x ${JARFILE}
22
23  %.class : %.java
24          javac $<
25
26  lis : ${SOURCES} all
27          mkpspdf ${LISTING} ${SOURCES}
28
29  clean :
30          - rm ${JARCLASSES}
31
32  spotless : clean
33          - rm ${JARFILE}
34
35  ci : ${SOURCES}
36          checksource ${SOURCES}
37          cid + ${SOURCES}
38
39  submit : ${SOURCES}
40          submit cmps109-wm.w10 asg5 ${SOURCES}
41
```

Figure 1. Makefile

(1) Note the creation of the jar file. Java class files are packed into a jar file in order that they may be used just as any other executable images.

(2) Carefully note the names of any inner classes that are created. They are known by the dollar sign in the filename. Only outer classes are listed as targets of the build.

(3) However one the **`jar`** command, the inner classes must also be mentioned, but in a **`Makefile`**, the dollar character is replaced by **`\$$`**.

3.2 Main Program Implementation

The main program's function is the analyze options and start one of three processes, namely a server, a text client, or a GUI client.

(1) Begin by writing the `main` function of class `chatter`, which will analyze options.

(2) You should have an inner class `chatter.options` which sets up the following fields:

(i) `String traceflags` has some trace flags similar to those in your C++ projects. These are not well specified, and are at your discretion.

(ii) `boolean is_server` will indicate whether a server or a client is to be started.

(iii) `boolean client_is_gui` will indicate whether to start a text or GUI client. This is incompatible with `is_server` being true.

(iv) `String server_hostname` will tell a client what the server's hostname is. This is ignored for the server.

(v) `int server_portnumber` is used both in the client and server to specify the connection port.

(vi) `String username` is an arbitrary username used by the server to identify clients when broadcasting messages back to the client.

(3) When you are testing your program, you should not use the same port numbers as others in the class, unless you specifically want to test your program in conjunction with theirs.

(4) The Well Known Ports are numbered from 0 to 1023, and may not be used by any of your programs. They are reserved for privileged applications, such as `ssh`(1). The Registered Ports are those from 1024 through 49151, and should probably be avoided. See `http://www.iana.org/assignments/port-numbers` for a list.

(5) The Dynamic and/or Private Ports are those from 49152 through 65535, and may be arbitrarily chosen by you, as long as you don't choose the same port number on the same server as someone else.

(6) When starting either a client or server, pass the `options` structure into the constructor. Outside the `chatter` class, it will be known as `chatter.options`. The dot works much like the double colon does in C++.

(7) Create a `trace` class similar to the ones in the C++ projects. Note that this is actually easier in Java because of the introspection capabilities. See `stacktrace.java`. You will probably want to make it significantly similar to that one. This is optional, since it will only be used by you and only for debug purposes.

3.3 Server Implementation

Next, try the server implementation. A client is useless without it.

(1) You might want to use `roboclient.java` initially, while you develop your server. Post any bugs you find in `roboclient` to the newsgroup.

(2) The server's main thread will just wait for connections, and when received will pass off the work to a pair of worker threads: One thread listens to the client and copies the client's messages into the queue. The other thread scans the queue and sends any unsent messages back to the client.

(3) The server will have a queue data structure to keep track of communications between clients. `LinkedList<String>` would be suitable, but encapsulated in a buffer class. Since this is not a bounded buffer, you may simply create a buffer class with `synchronized` get and put operations. A better way is to use a `java.util.concurrent.BlockingQueue <message>` which can be instatiated as either an `ArrayBlockingQueue` or a `LinkedBlockingQueue`, both of which take a generic contents parameter.

(4) The client listener thread will go into a loop reading lines from the client, prepending the username to the front, and then appending the new message to the buffer:

```
username + ": " + message
```

(5) When the client listener notices that the client has closed the socket, it sends a message to the user manager and the stops.

(6) The client sender will loop, repeatedly accepting messages from the queue manager, and sending these off to the client. It should also check with the user manager and stop if a flag is set.

(7) The user manager is notified whenever a client connects. This can be implemented by a simple thread containing a `HashMap <String, Boolean>` The string is the username, which is true when a client connects. It is set false by the client listener just before it quits. When the client writer notices that it is false, it tells the manager to delete the entry and then quits.

(8) Another possibility is a `HashSet <Thread>`, where the user manager iterates sending messages to each thread, and when a client is done, it just deletes itself from the hash.

(9) Another thread is the queue manager, which maintains messages. It accepts a new message from any client listener. Periodically, and without any waiting, it repeatedly removes a message from the queue, and sends it to all of the client writers.

(10) A client writer must, therefore register with the user manager, which maintains another `HashMap`, containing a mapping from a username onto a thread.

(11) There is no way internal to the system to shut down a server. It runs forever in an infinite loop. There are several ways to stop it :

(i) If it is running in the foreground at an xterm, `Ctrl/C` will kill it.

(ii) If it is running in the background, it can be brought into the foreground.

(iii) Alternately, it can be killed by one of the commands

```
kill -9 1234
kill -9 %2
```

where `%2` (or whatever) is the terminal job number which can be discovered with the `jobs` command. Or use `1234` or whatever is the process id (pid) as shown by

```
ps -ef | grep server
```

3.4 Text Client Implementation

The text client is rather simple, and can be just a suitable modification of the roboclient.

(1) The text client starts up two threads, the listener and the reporter.

(2) The server listener thread repeatedly keeps taking messages from the server and printing them to `System.out`. When it notices that the server has no more lines, it waits for more from the server.

(3) The sender thread repeatedly reads lines from `System.in` and sends them to the server. It stops at end of file, which a user can signal via a `Ctrl/D` as the first character on an input line. When the sender thread sees EOF, the client shuts down.

(4) The code to interact with the server should be completely separate from the code that monitors the terminal. This is done so that the GUI client can just call the text client's code for interacting with the server.

3.5 GUI Client Implementation

The GUI client does not make use of either standard input nor output.

(1) Begin with the sample code for `textarea.java`, which has the basic GUI infrastructure already built.

(2) The GUI will quit when you press the `Quit` button.

(3) Modify the title bar code so that it reflects the actual information about the client and server. It also needs two threads, one to listen and one to send.

(4) The large text area is for a log of the conversation. It is owned by the thread that listens to the server and just appends a new line to the text area. When the text area fills up, scroll bars will appear, so don't attempt to delete anything from the text area.

(5) At the bottom of the window is a text field. When a line is entered, followed by return, send the line to the server. Delete the code that displays it in the text area. The local information is only seen when returned by the server.

4. What to Submit

Submit all necessary Java source files, but do not submit any class files or the jar. Submit `Makefile` with the usual targets, and `README`. If you are doing pair programming, follow the instructions in `/afs/cats.ucsc.edu/courses/cmps109-wm/Syllabus/pair-programming/`.