

CS 450 – Introduction to Networking – Spring 2012

Homework Assignment 2

Converting an Application to a Distributed App

Due: Thursday 23 February. Electronic copy due at 1:30 P.M., Optional paper copy to be delivered to the TA. (Note: the paper copy must match the electronic copy exactly.)

Overall Assignment

For this assignment you are to take the application that you developed in HW1 and convert it to a distributed application supporting at least two players. The server will be responsible for maintaining the status of the playing field, and transmitting a new board to each of the clients whenever a change occurs. Clients will re-display the board whenever a new one is received from the server, and will process user input and send that information back to the server.

Server Operation and Details

- The syntax for running the server shall be:

```
gameServer [ port ] ...
```

- port is the port to listen to for new connections. Default value is 60607.
 - You may support additional arguments to appear after the ones shown here, provided they are well documented in your README file.
- The server shall (ultimately) perform the following tasks:
 1. Establish an initial playing field, and data structures for keeping track of clients.
 2. Create a socket to listen for new connection requests.
 3. Use **select()** to determine which socket(s) have data available.
 - If new connection requests are present, establish new connections and add the new sockets to the list of sockets to listen to. This will also involve increasing the number of current players.
 - If new messages have arrived from clients, read the messages and respond accordingly. Generally this will involve updating the status of the playing field and transmitting a new copy to all connected clients. It may also involve decreasing the number of current players and disconnecting the appropriate connections.
 4. Periodically (once per second or faster?) update the playing field by moving pieces, update scores if appropriate, and transmit the new board and other info to all connected clients.
 5. Repeat steps 3 and 4 forever, or until something stops the program.
 6. Close all sockets when exiting.
 - Data transmitted from the server to the clients will include the full playing board, (minimum 20 rows by 50 columns), plus player scores, paddle locations, and any other information necessary to display the board and play the game. It is recommended all this information be packaged into a single data structure, (i.e. a struct or class object), most likely of fixed length, (i.e. supporting a pre-determined maximum number of players / clients.)

Client Operation and Details

- The syntax for running the client shall be:

```
gamePlayer remoteHostName [ port ] . . .
```

- The remote host name is the name of the computer running the server program.
 - port is the port on the server that is listening for TCP connection requests, default 60607.
 - You may support additional arguments to appear after the ones shown here, provided they are well documented in your README file.
- The client shall (ultimately) perform the following tasks:
 1. Contact the server to establish a new TCP connection.
 2. Display the initial board, (scores, etc.), received from the server before checking for any user input.
 3. Use **select()** to check if there is any data to process from either the server or the keyboard.
 4. Display the board, (scores, etc.), whenever a new playing board is received from the server.
 5. Process user input whenever keyboard input is received. Normally this will involve transmitting user movements to the server, but it may also involve processing the Q key if the user wants to quit or the H key if the user is asking for help, etc.
 6. Repeat steps 3 to 5 until the user decides to quit the game.
 7. Report all results to the screen, and close down all connections.

Socket Details

- The server needs to perform the following tasks regarding sockets:
 1. Call **socket()** to create a socket for accepting TCP connection requests. The type should be **AF_INET** (or **PF_INET**) and **SOCK_STREAM**.
 2. Call **bind()** to assign a name (port number) to this socket.
 3. Call **listen()** to mark this socket as ready to accept connection requests.
 4. Call **accept()** to create new sockets based on connection requests received on the socket created in step 2. All further communications with this client will be through the new socket created by **accept()**.
 5. Use **read()** and **write()** to communicate with this client via the newly created socket.
 6. **Note:** You can use **select()** to determine which socket(s) have data available to be read. Note that the data structure sent to select gets modified, so you need to set it up again before each call to select.
 7. Call **close()** when necessary to close down the socket.
- The client needs to perform the following tasks regarding sockets:
 1. Call **gethostbyname()** to lookup the IP address of the server.
 2. Call **socket()** to create a socket for communications with the server.
 3. Call **connect()** to request a connection from the server.
 - Connect will need to know the name used by the server in the **bind()** step listed above.
 - Connect will fail if the client tries to connect before the server is ready. Check the return value, and use a busy loop (with a **sleep(1)** call in it) if connect fails with **errno** equal to **ENOENT**.
 4. Use **read()** and **write()** to communicate with the server.

Evolutionary Development

It is recommended that the program be developed in the following order, making sure each step is working before beginning development on the next step:

1. Develop initially with the client and server running on the same computer, or on two adjacent computers in the computer lab.
2. Develop simple server and client programs that create sockets and establish basic communications. Verify that they can exchange data without any concern for the game at this point.
3. Implement the game, and let the server assume there will only be a single client. **Select()** is not needed in this case. (By the server. The client may still need **select()** or some other mechanism to check for keyboard input separately from the arrival of screen updates.)
4. Modify the program to handle multiple players. At this point the server will need to implement **select()** to determine which connection(s), if any, have incoming data to process.
5. Work out any remaining bugs so that the client-server system efficiently allows either a single player or multiple players to enjoy the game.
6. Optional Enhancements – See below.

Data Packet Format and Contents

- At a minimum, packets going from server to client will need the complete playing board, and whatever scores and paddle location information is necessary to play your game. Packets going from client to server will normally need player movement information, and/or control messages such as quit.
- You are welcome to add any additional data fields you wish, provided they are well documented in your readme file. This may include diagnostic information and/or data needed to play your particular game.

Required Output

- All programs should print your name and CS account ID as a minimum when they first start.
- Beyond that, this program should play the game as described above.
- It may be appropriate to report some summary statistics when the program closes, such as the amount of data transferred, average response rate, etc. More details on this may be provided in class and/or on the class web site.

Other Details:

- A makefile is required, that will allow the TA to easily build your program. As always, you are free to develop wherever you wish, but the TA will be grading the programs based on their performance on the CS department Linux machines.
- (The makefile may be omitted if compilation is straightforward, and documented in README.)
- MOSS will be used to check submissions not only against other submissions from this semester, but also against past submissions from previous semesters and any similar programs found on the web, so be sure to do your own work.

What to Hand In:

1. Your code, **including a makefile if needed, and a readme file**, should be handed in electronically using Blackboard.
2. The intended audience for the readme file is a general end user, who might want to use this program to perform some work. They do not get to see the inner workings of the code, and have not read the

homework assignment. You can assume, however, that they are familiar with the problem domain (e.g. computer networking.)

3. A secondary purpose of the readme file is to make it as easy as possible for the grader to understand your program. If there is anything special the grader should know about your program, be sure to document it in the readme file. In particular, if you add any additional command-line parameters or special fields to the data packets or do any of the optional enhancements, then you need to document what they are and anything special the TA needs to do to run your program and understand the results.
4. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
5. Make sure that your name appears at the beginning of each of your files. Your program should also print this information when it runs.
6. The readme file must specifically provide direction on how to run the program, i.e. what command line arguments are required and whether or not input needs to be redirected. It must also specifically document any optional information or command line arguments your program takes, as well as any differences between the printed and electronic version of your program.
7. If there are problems that you know your program cannot handle, it is best to document them in the readme file, rather than have the TA wonder what is wrong with your program.
8. A printed copy of your program, along with any supporting documents you wish to provide, (such as hand-drawn sketches or diagrams) may be provided **to the TA** on the day the assignment is due or the following school day. The hard copies must match the electronic copy exactly.
9. Make sure that your **name and your ACCC account name** appear at the beginning of each of your files. Your program should also print this information when it runs.

Optional Enhancements:

It is course policy that students may go above and beyond what is called for in the base assignment if they wish. These optional enhancements will not raise any student's score above 100 for any given assignment, but they may make up for points lost due to other reasons. Note that all optional enhancements need to be clearly documented in your readme files. The following are some ideas, or you may come up with some of your own:

- Analyze the data traffic and response times of your application, either by collecting statistics within the game itself or with analysis tools such as Wireshark. Document any modifications you make to improve performance, with some measure of the results (e.g. speedup) of the modifications.
- Anything else you can think of – Check with the TA for acceptability.