CMSC 212
Project #6 – Socket Communication and Event Handling
Due 12/13/2005 11:59 PM

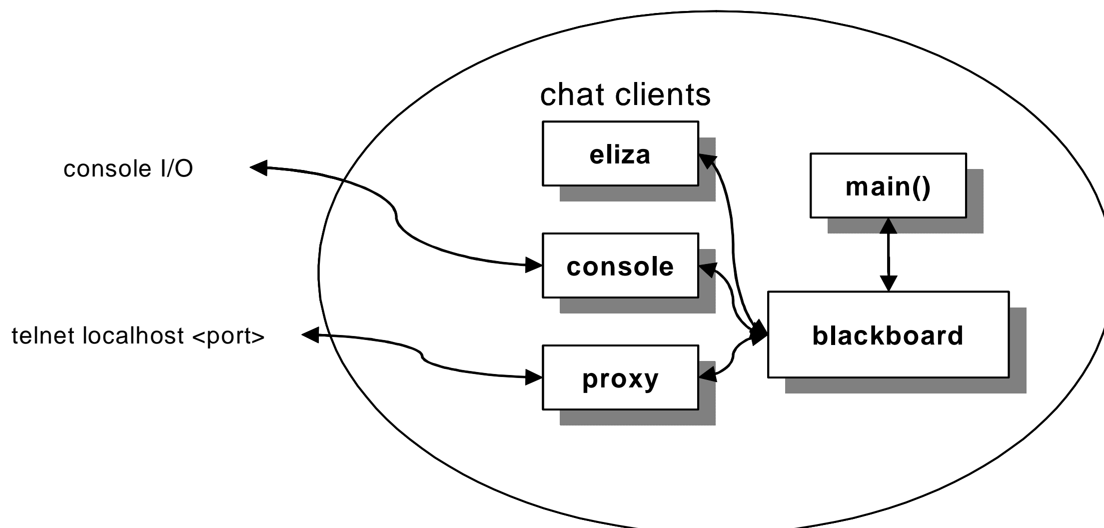
Background

This project will implement a black-board engine that will allow exchanging information among bits of software. Examples of applications that can use this type of system include chat (instant message) servers, multi-player game engines, and distributed computing applications such as SETI@home.

You will implement a blackboard API that will relay messages to interested parties. You will use the API to implement a simple chat server, allowing local, remote, and local robotic users to chat.

This assignment will use TCP ports to communicate. To prevent interference, go to http://titan.cs.umd.edu/ports to get a range of ports. Although you will need only one port for the assignment, we provided 10 in case of problems with any one of them. You may go back to get a new range if you need to (if you forget the previous range, for example).

The structure of the system is shown below:



Main lines of communication.

You will build a single binary called 'blackboard'. You main routine will initialize three chat clients: "eliza", a robotic chatter, "console", which allows you to type in messages on the command line, and "proxy", which allows other programs to connect via network sockets (and the "telnet" command). Each chatter will register a set of handlers for various types of events. Finally, your main routine will call `blackboardMainLoop()`, which will implement an event loop, repeatedly reading messages from a chatter and relaying to others.

Blackboard API

The following is the API of the core blackboard routines. With the lone exception of `blackboardMainLoop()`, all of these routines are called by chat client code.

boardHandle connectBlackboard(char *id)

> Register a user, whose name is specified by `id`, with the blackboard. Called by each chat *user*. The returned `boardHandle` points to a structure that keeps per-user information.

int registerMessageHandler(boardHandle, char *eventType, eventHandlerFunc)

> Register a message handler. Whenever a message of type eventType is sent, function passed in eventHanderlFunc is called. If a NULL value is passed for eventHandlerFunc, the existing handler for that message type should be removed. Note that the contents pointed to by a `boardHandle` are not specified, you decide this yourselves.

int registerNewUserHandler(boardHandle, newUserHandlerFunc)

> Register a function to be called whenever a new user connects or leaves the system. Existing newUserHandlerFuncs are called whenever a new user joins or leaves. A new newUserHandlerFunc is also called for each user already in the system, so registering a newUserHandlerFunc allows a client to be informed of the all clients in the system. If a NULL value is passed for newUserHandlerFunc, the existing new user handler for that boardHandle should be removed.

int sendMessage(boardHandle, char *eventType, int lineCount, char *data[])

> Send a message to all users of the blackboard (except the sender of the message) who currently have a message handler for a message type eventType. If there are no such users, the message is simply discarded. The message consists of lineCount lines sent in an array of character pointers in data.

int leaveBlackboard(boardHandle)

> User with "id" is leaving the system. All registered event handlers for that user should be deleted, and then any `newUserHandlerFuncs` for other users are called.

`int registerFileEventHandler(int fd, fdEventHandlerFunc)`

> Register a function, eventHandlerFunc, to be called when ever there is data to be read from the file descriptor fd. If a NULL value is passed for eventHandlerFunc, the existing handler for that file descriptor should be removed.
>
> This routine will need to be called to register handlers for the console and network proxy chat clients.

void blackboardMainLoop()

Main event loop of the system. Wait in a loop for new data to arrive on one of the file descriptors that has an event handler defined. This loop should **not** return until there are no file descriptor handles registered. Your main function will call this as its last act.

Unless otherwise indicated, all API routines with an integer return value should return 0 for success, and –1 for any sort of failure or error case. Registering a new handler replaces any existing handler for that user.

handler functions prototypes

The file blackboard.h (which you may not change) contains the definitions for the following function pointer types for the various callback functions in the API.

typedef void (newUserHandler*)(char *id, int arriving);

`arriving` non-zero if the user is arriving, zero if leaving.

typedef void (eventHandlerFunc*)(boardHandle board, char *id, char *event, int lineCount, char *data[]);

typedef void (*fdEventHandlerFunc)(int fd);

Chat Application

As part of this assignment you will build a chat system that uses your blackboard API. The way the chat system works is that each user will send messages to the blackboard system. The blackboard system will distribute these messages to other users. You will write two chat components (besides the main routine): console and proxy. You can use any message type, but we use message type "chat" (and eliza only listens to "chat" messages).

Console Client

The console chat user will read messages from standard input and print any messages it receives to standard output. Since this is an event driven program, you will need to register a read event handler for the file descriptor associated with standard input (0). This read event handler will then call fgets to get a line of input and send it to the blackboard server. Your client should also have an event handler for users arrival/departure events and print a message of the form "User <user name> [joined|left] the chat". If the console's stdin handler gets called and fgets returns no bytes (end of file from standard input because of control-D), the chat system should immediately exit. Incoming messages should be printed out, preceded by a line of the form "Message from <id>", where id is the proper blackboard id.

Network Proxy Client

The proxy client allows remote users to connect to your blackboard system. To do this, your proxy will wait for new TCP connections on a socket at a port number which is passed in as a command line argument to the blackboard system. This port number is passed to your proxy routines through a function you need to write, declared as follows:

```
int initProxy(int port);
```

`initProxy()` should return 0 on success, non-zero otherwise.

The proxy will define a read event handler that is bound to the supplied port. When a new connection arrives, that handler function will get called. This handler function will use the accept system call to get a network connection to the new user. It will create a blackboard handle for the remote user. It will then create a new file descriptor event handler that calls a function remoteUserHandler when new data arrives for that user. It should also register a message handler for this connection. When the new the message handler is called, it should send the message to the remote connection (via the write system call). When the remoteUserHandler is called (indicating data to be read from the remote user), the handler should use the read system call to get the data from the network and pass it to the blackboard system via a sendMessage call. You can use any convention you like in assigning blackboard IDs to network clients.

To test your proxy, you can use the Unix command telnet to connect to your chat server. Use the command "telnet localhost <port>" where port is the port number you supplied when starting your blackboard server.

As above, incoming messages should be printed out, preceded by a line of the form "Message from <id>", where id is the proper blackboard id.

Robot Client

We have also supplied a user for the blackboard system. This user is a robot chatter that will read other peoples messages and attempt to reply in a way that appears that it is another person. This client is called Eliza[1] and is supplied in the file eliza.o. As far as you are concerned, this user is simply a plug-in and all you need to do to use it is to link the file eliza.o into your blackboard program and call the function initEliza() before blackboardMainLoop().

What To Do

Your project should produce one UNIX executable file (blackboard) which contains the blackboard routines (blackboard.c), the console client (main.c), the proxy client (proxy.c). We will supply a third client (eliza.o) which uses the blackboard API, eliza.o should also be linked into the blackboard executable.

---

[1] Eliza was the name of an early program that attempted to respond to commands in a way that mimicked human intelligence.

Your blackboard program should take one command line argument, port, which is the port the proxy should listen on for network connections.

Files

Files to be handed out are in ~212files. We include the source of the public test "message".

212files also includes a sample binary version of the blackboard program, called "blackboardSample". You can run it and see how the system is supposed to work.

Hints

Socket Communication

The network proxy creates a socket endpoint as follows:

```
socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

After you check the return value, you should mark the socket as one that can be re-used:

```
int optVal = 1;
ret = setsockopt(sock, 0, SO_REUSEADDR, (char *)&optVal,
                 sizeof(int));
```

Sockets are checked for data ready to be read (you do not ever need to register *write* or *exception* fd_sets) via `select()`.

A partial list of socket functions you will need to use is:

- socket()
- setsockopt()
- bind()
- listen()
- accept()

The Zen of the Console Client

The console needs to register an event handler for incoming data on file descriptor 0.

The Zen of the Proxy Client

The proxy creates a socket, tells the OS to listen on it, and registers a `fileEventHandler` for it. `blackboardMainLoop`()'s `select`() will return a read event on this socket when a remote user attempts to connect. The handler then

accepts the new connection, resulting in a new socket, which must also be registered with the blackboard so that data may be read from it.