Purpose: To use some of the advanced server techniques from Chapter 30.

You will build an upgraded browser server/client pair. You will call your browser server `browserd2.c` and your client `browser2.c`.

Your browser client/server will provide the same **c**, **g**, **l** functionality as before. Do not change this part of your program.

Client Upgrades:

1) Your browser client will use broadcasts (instead of the `arvg[1]`/`localhost` stuff) to find a browser server that is on the same cable. When you browser client starts it will send a (UDP) broadcast and wait for a (one) response to that broadcast. It will then close the UDP socket and do a `connectTCP` to the server that sent the response. If more than one server responds, all except the first one are ignored. The string your broadcast time client printed (`inet_ntoa`), is exactly the string you need to send to `connectTCP`.)

In summary, remove the switch statement and the default server (`localhost`) and replace that portion of the code with the broadcast code. Do one broadcast, and assume a server responds.

Notice: the only changes in the client are to in the startup code.

Server Upgrades:

1) Open an additional socket that is a UDP socket with broadcasts enabled. Since UDP and TCP sockets are distinct, you can use the same port number (your `get_port` number) for both. (We've did something similar to this with the time server.) If you get input on the UDP socket, respond to the sender (somewhat like in your broadcast time server) with a packet that has a single character. It doesn't really matter what you send in the packet since all you are need to send back is your "address" and that is contained in the packet header.

2) Add the "auto-background" code (see Chapter 30). Testing caution: since this will cause your server to run in the background, control-C will no longer be an option to stop your server; you will need to use the `kill` command with the process ID of your server.

3) Add code that logs (`syslog`) a message each time a brower is started or stopped. Use facility `LOG_LOCAL0` and level `LOG_WARNING`. As your "program name" use your assigned port number from `get_port`, that will allow you to distinguish your log entries from other peoples.

4) Add code to "reload" a configuration file whenever you receive a `SIGHUP`. To make this easier, you won't do a real reload (especially since there is no configuration file), you will simply log (i.e., `syslog`) a message that says "reload requested".

Notice: the only addition to the `TCPbrowser` function in your server is the addition of the `syslog` calls at the start and end of that function. You could actually change the way you call that procedure (so the call preceeds the exit call instead of being inside it) and all your changes would be outside the browser function.

Testing: Log files are often not readable for security reasons. To allow you to test, I have configured `panther` so that it will put the `local0` log entries in the file `/var/log/local0` which is readable. So, to test the log file portion of your program, run your server on `panther` and run your client on `cougar` (which is on the same cable so the broadcast will work).

To test the broadcast portion, use a couple machines in the same row and move your server around to see if the client can find if after the move. Remember, when running the client, you do not enter the location of the server, it should auto-locate the server.

Addendum:

Other changes to the client we could have made but didn't: The client could send a broadcast, and use a select with a timeout. If it times-out it could send another broadcast and keep trying until if found a server. The client could list all servers (somewhat like in the broadcast homework) and allow you to select one of them.