Aalto University
School of Electrical Engineering

S-38.3610 – Network Programming
Spring 2013

HDserver
Documentation

Riku Lääkkölä 69896S
riku.laakkola@aalto.fi

Contents

1 Usage Instructions and Disclaimer

DISCLAIMER 1: For this assingment I had two attempts. First, I tried to implement a single-threaded server that uses select() and nonblocking I/O for multiplexing. I managed to get serving GET request to be somewhat functional, but otherwise there seem to be problems. I had to fall back to using code that I wrote for last year's course that forks a child process for each client, which is somewhat more inefficient, but at least the code works...

And now for the instructions: First, the program must be compiled by running `make`. There are no special requirements (at least make runs flawlessly on the course servers) After this, just run the server! Usage as follows:

To start child forking server:

`./fork_srv <working_directory> <port>`

To start selecting server:

`./select_srv [-d] [-p <working_directory>] <port>` (*d* for daemonize)

The forking program logs information to LOG_LOCAL7 and the selecting one prints very verbosely to stdout (unless run as daemon).

2 Overview

The program serves HTTP GET and PUT requests from clients. The program can handle both IPv4 and IPv6 addresses, and when using DNS names, the user does not need to worry about the IP version.

2.1 Forking version

Internally, the software is divided into four components which are:

1. *myhttp*, which generates and parses HTTP messages and reads and writes them from sockets.
2. *npbserver*, which handels DNS queries, establishing connection to the server and separating the child logic from the main loop.
3. *main*, which parses the user provided command line arguments, daemonizes the process and handles the main accept loop and forking

More specifically, the program handles starting itself and processing a GET request from a client in the following manner:

1. The listening socket is opened using tcp_listen in npbserver.
2. A for-loop is started. The loop blocks at accept until a client connection attempt arrives.
3. If connection is established successfully, a child is forked, and functionality moves to the web_child function in npbserver.c.
4. Data is read from the socket to a buffer until CRLF occurs, and after that, the header is parsed.
5. If the header is valid, the appropriate function for the request is called.
6. The process_get function first writes the appropriate header to the socket and then starts reading the file into a buffer and writing to the socket from the buffer.

DISCLAIMER 2: The *readn* and *writen* functions which try to write and read exactly n bytes from or into a socket are copied almost directly from the Stevens book examples (also available in lecture materials in

Optima). Also the *tcp_connect*, *daemon_init* and *tcp_listen* function is copied almost in full from the lecture examples.

2.2 Selecting version

The selecting version tries to handle parallel activity using select and nonblocking I/O. The listening socket is set up in the same way as above, but instead of a blocking accept in the main loop, we have select that returns whenever there are events in sockets we are interested in.

In the beginning the only socket that is set in the fd_sets provided to select is the listening socket. When select returns, it means that there is a connection attempt. If the connection is successfull, a new client structure is initialized and appended to a linked list of all clients. This list is checked always before select, and relevant clients are added to relevant fd_sets.

When select returns, the list of clients is checked again in order to see if their respective sockets have become readable or writable during select. If so, depending on the socket status and client state, the appropriate functions are called for processing requests. The processing is fairly similar to the forking version, with the exception that each I/O function is called only once, and if they fail with EWOULDBLOCK, the function returns to the select loop. Also the client struct pointer is passed along, so that the client state may be altered by the functions.

3 Testing and limitations

The servers were tested on the nwprog test servers and my home computer, using the client from phase 1 and normal web browsers. The selecting version does not handle unexpected conditions nicely... The forking version is a bit better.

4 Specifics

1. The forking version uses separate processes for parallel activity, and the selecting version uses select and nonblocking I/O. If a

connected client stops transfer, the I/O functions will block until the default timeout and after that the child exits. The selecting one will maybe have the bad client in the list forever, but the sockets will not appear active in the fd_sets.

2. The implementation is address family independent. This is accomplished by providing *getaddrinfo* with a *hints* structure that has *ai_family* of value *AF_UNSPEC*. This was tested by running the software from my home (IPv6 connectivity) and Otakaari 5 (no IPv6) and some informative statistics printing.
3. For some reason, apache benchmark timeouts when stress-testing, so no acceptable high load test results are available. The correct number of bytes seem to be transferred with tens of simultaneous connections with ab. With keepalive enabled, 10 concurrent connections go through, but 100 kills the server.
4. With abrupt stopping of transmission from client side, both servers seem to function correctly (serving to functional clients still works afterwards), and in both cases resources of malfunctioning clients should be freed after timeouts.
5. I tested jariliezen's and hhamalai's servers by uploading a 16M file and then downloading the same one and diffing. The first one worked perfectly, but the second one seems to miss about 10 kilobytes from the end of the file when downloading!

5 Diary

When participating this course last year, I figured I'd take the easy way out and use the simplest method: fork child for each client after connect. This year I aimed for a more sophisticated or elegant or efficient solution, but failed a bit, as the forking version seems to function a bit more reliably.

In the select ad nonblocking I/O way of multiplexing, keeping track of the state of each client and assigning appropriate tasks at the right moments seems to be quite tricky. Also, I have yet to find a way to make absolutely certain, that no malfunctioning or dead clients linger on in lists...

Moreover, I might have a problem with my way of implementing my designs: I tend to desing carefully, then start implementing in a way, that compiling before the entire thing is finished is not possible. This leads to my current situation, where I write code fiercely for 20 hours, then spend a couple of hours to get it to compile and then 10–∞ hours of debugging and adding bubble gum.

At first I was considering using libevent, but then ended up using plain old select for multiplexing, as I was afraid that libevent would make things too abstract for this course. According to creators of libevent however, each platform seems to have its own most efficient way of event based multiplexing, so using libevent would probably be advised for portability of efficiency!

All in all, I still seem to have a couple of things to learn about using select and nonblocking I/O.