2543289, sbn338
Bas van der Borden

Internet Programming

Assignment 2: Distributed Programming with Sockets

All programs have been tested on Ubuntu 16.04 LTS. Besides the standard libraries, the GNU extension for the hsearch function has been used. After running make all shells can be invoked by their respective ./executables. For writing the writen function from Stevens book has been used. For reading we simply check whether or not the read function returns an error or not. **Servers 2 and 3 do not work as expected.** Reason for this is that I was unable to get the shared memory between the processes working. The best way to see this is to run server 3 with 2 processes, after a put the first get will then return nothing whilst the second get will return the correct value, because this is executed by the process which executed the put.

The client has been implemented according to the assignments requirements. It takes multiple arguments, from which the first two are the hostname and port number. The client uses getaddrinfo to resolve the host and sets up the socket and tries to connect to the host. If any of these actions fails the client exits and prints the error which caused the exit. After the connection with the host has been established, the client loops through the remaining arguments. If in this loop the first argument is put then the next two arguments are taken as key and value, when the argument is get the next argument is taken as the key. If the argument is neither put or get, the client closes the connection. Otherwise the connection is only closed when the client runs out of arguments.

The first server is implemented as a iterative server. The server keeps the bytes specifying the actions to take as global ints. Starting in main the server sets up the socket, binds and sets the socket options. After this it calls the function create from keyvalue.c, this function creates the hashtable with size 64. The hashtable is set up with the hcreate_r function such that it creates a global hashtable which can easily be referenced. After this the server starts listening on the socket for incoming connections. The server then enters a while loop to keep it running. The server accepts incoming connections and starts reading the message send by the client. The put and get bytes are set through memset and with memcmp the server decides whether the request was a put or get.

In the case of put the server calls the put function from keyvalue.c. For this server implementation the put function only has to perform the storing of the key and value. This is done through assigning the key and value to ENTRY item and then letting the function hsearch_r enter the item in the global hashtable. To update an existing key the value is once again updated after the hsearch_r call.

With a get the server calls the get function from keyvalue.c. This function takes the key as argument, then duplicates this key into a ENTRY item. This item is then used with hsearch_r to find the associated value in the hashtable. If a value is found we return this value as a char, otherwise we return "NULL". This returned char is then compared within the server. If the returned value is not "NULL" then the correct value is set for the return message to the client and the value from the hashtable gets concatenated. The server then writes this back to the client. In the case that a value has not been found the server writes back the byte 110. After the write to the client the return string is set to zero.

When all requests have been handled the server closes the connection to the client.

The second server implementation is based upon the one-process-per-client architecture. It differs from the first server implementation in that before the while loop starts it calls signal, this is done to keep the server's main process waiting on all of its child processes. Within the while loop the server accepts connections from clients and forks a process when a connection has been made. Within this fork the message from the client is read and processed in a similar way to that of the iterative server.

Other differences with the first server is that we need to take synchronisation into account. For this we implement a semaphore within the while loop before a call to put is made, thus blocking other put calls. This does mean that a get request may not return the correct value as there is no lock on the key used.

Finally because we are dealing with child processes we need to keep the hashtable synchronised between processes. This can be done through the use of shared memory, but unfortunately I was not able to implement this.

The third server is a preforked server. This server mainly follows the one-process-per-client server but after listening to the socket it preforks processes. These child processes are given the file descriptor of the socket and then all perform like the iterative server. They differ from the one-process-per-client server in that before they can accept a request they need to hold the semaphore locking access to the accept call, this is done such that there is no race condition on the accept call. Furthermore this server suffers the same shortcomings as the second server. *sidenote all forked processes need to be terminated through task/system monitor.*

The fourth server is a multi-threaded server. This server follows the implementation of the preforked server but instead of forking the server creates threads which are then run as though they are iterative servers. Instead of the semaphore lock we use a mutex lock on the accept call. Furthermore we lock on put, thus no two puts can be done concurrently. This decision has been made because I was unable to build an implementation which locked on keys. Finally this server does return the right value from the hashtable regardless of which thread handles the get request. This is because threads do use the same global variable, thus using the same hashtable.

For the second part of the assignment the talk program has been made. This program starts in either server or client mode based upon the amount of arguments supplied when the executable is called. Server mode when called with 1 argument and client mode when called with two arguments. From there either server or client is called with their respective arguments. The server program sets up the socket, binds, sets socket options and listens for connections. When a connection is made it calls conversation with the file descriptor of the accepted connection.

The client program starts with getting the address info of the host specified. After this it sets up it’s socket and tries to connect with the host/server. When the client connects to a host it calls conversation with it’s file descriptor.

The conversation program is the same for both the client and the server. Conversation starts with the signal call to intercept the ctrl+c command. When this command is performed the program which catches this command sends a kill message to the other participant and exits. This message gets recognised and makes the other program exit. Furthermore the conversation program uses a select to monitor for both input or a received message. Before reading a message the buffer is first zero'ed. After this the message is read and compared to the kill message. If this message is not the kill message, the message is printed and the file descriptor is cleared with FD_CLR. When a user wants to send a message the program detects this with the select and uses fgets to get input from the user. This input is then send and the file descriptor for user input is cleared.