Machine Problem 5: Multi-Threading Over a Network

CSCE-313-501

By: Andrew Schlotzhauer and Jaiden Gerig

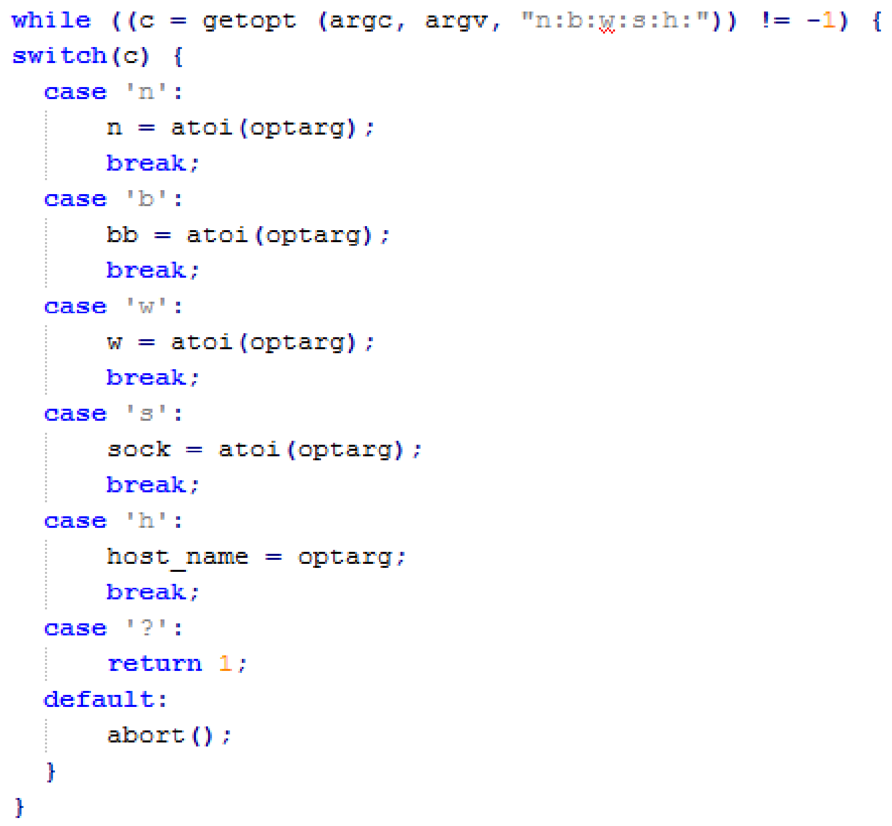
May 3, 2015

Introduction

In computer science today, nearly everything is done over a network, computers speaking to one another across short or long distances. Therefore it is very important to get a firm grasp on how computers speak to one another both through the applications and hardware. In this machine problem we are to extend upon our last machine problem which was multi-threading across a data server on a single machine. We are to make it where the same thing can happen, but can interact with another program on a different machine through a single TCP connection. Also in order for this to work we needed to add a few arguments such as the port number of the server host and name of the server host.

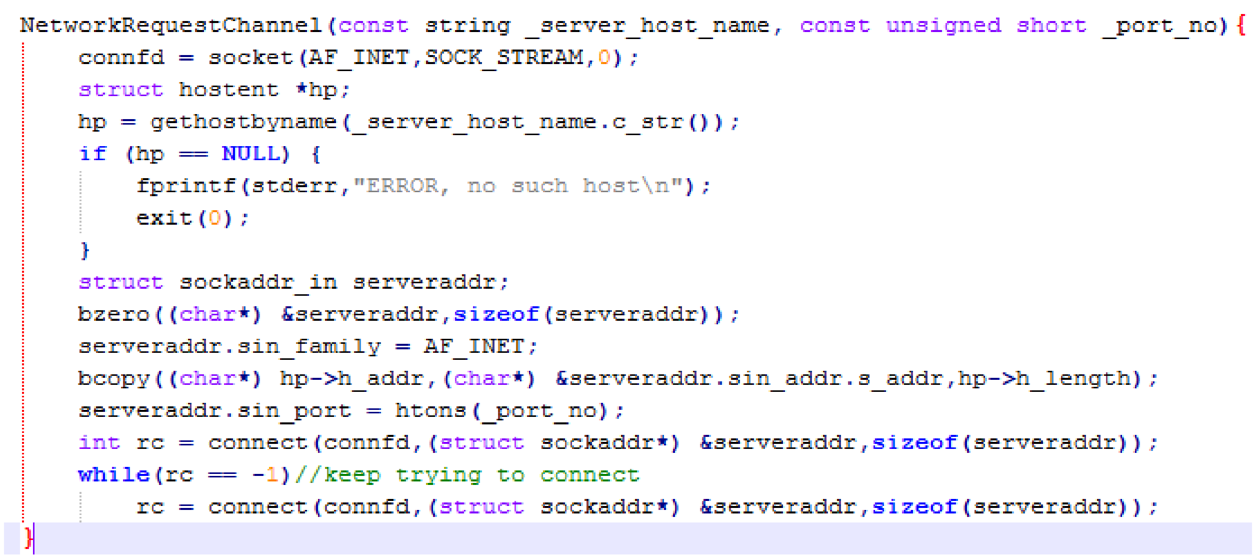
Procedures

The first thing we did was add the command line argument statements to the old getopt command line argument block because it was easy and didn't take very long.



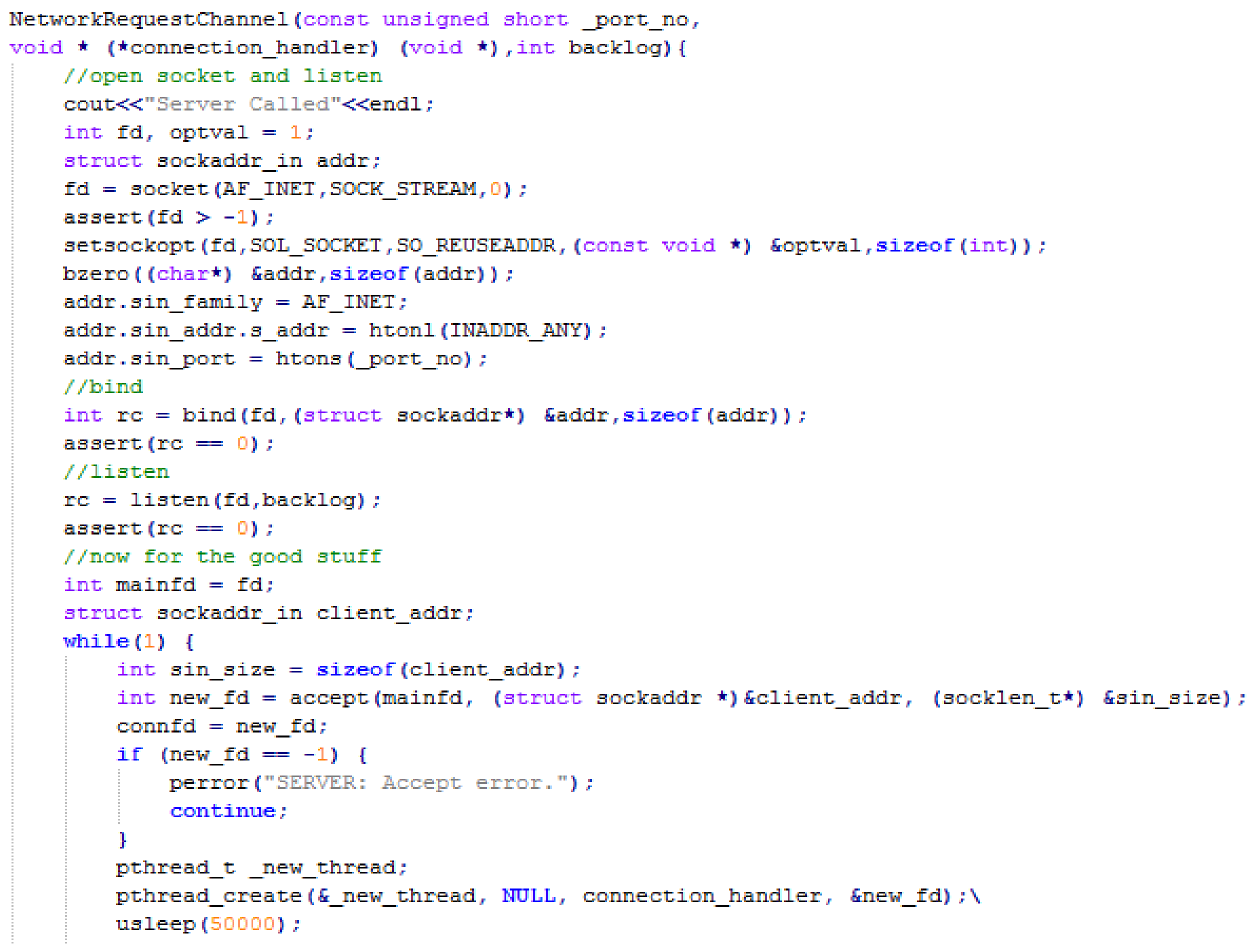
```
while ((c = getopt (argc, argv, "n:b:w:s:h:")) != -1) {
switch(c) {
  case 'n':
      n = atoi(optarg);
      break;
  case 'b':
      bb = atoi(optarg);
      break;
  case 'w':
      w = atoi(optarg);
      break;
  case 's':
      sock = atoi(optarg);
      break;
  case 'h':
      host_name = optarg;
      break;
  case '?':
      return 1;
  default:
      abort();
  }
}
```

The next thing we did was started working on the Network Request channel class by doing the constructors by going by the guidelines in the given instructions.



```
NetworkRequestChannel(const string _server_host_name, const unsigned short _port_no){
    connfd = socket(AF_INET,SOCK_STREAM,0);
    struct hostent *hp;
    hp = gethostbyname(_server_host_name.c_str());
    if (hp == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }
    struct sockaddr_in serveraddr;
    bzero((char*) &serveraddr,sizeof(serveraddr));
    serveraddr.sin_family = AF_INET;
    bcopy((char*) hp->h_addr,(char*) &serveraddr.sin_addr.s_addr,hp->h_length);
    serveraddr.sin_port = htons(_port_no);
    int rc = connect(connfd,(struct sockaddr*) &serveraddr,sizeof(serveraddr));
    while(rc == -1)//keep trying to connect
        rc = connect(connfd,(struct sockaddr*) &serveraddr,sizeof(serveraddr));
}
```

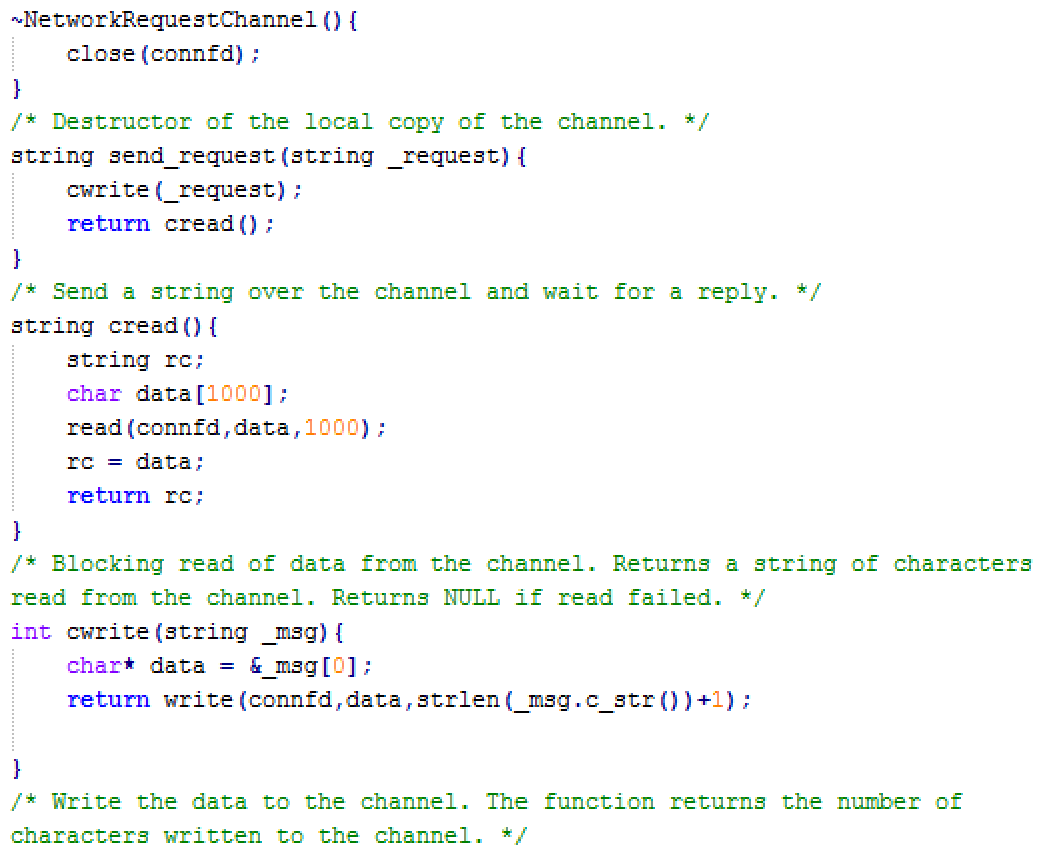
This just simply sets up the network request channel by getting the host name and server host port number, and connects to the server. The constructor above is called by the client and basically creates a CLIENT-SIDE local copy of the channel which is connected to the given port number at the given server host.



```
NetworkRequestChannel(const unsigned short _port_no,
void * (*connection_handler) (void *),int backlog){
    //open socket and listen
    cout<<"Server Called"<<endl;
    int fd, optval = 1;
    struct sockaddr_in addr;
    fd = socket(AF_INET,SOCK_STREAM,0);
    assert(fd > -1);
    setsockopt(fd,SOL_SOCKET,SO_REUSEADDR,(const void *) &optval,sizeof(int));
    bzero((char*) &addr,sizeof(addr));
    addr.sin_family = AF_INET;
    addr.sin_addr.s_addr = htonl(INADDR_ANY);
    addr.sin_port = htons(_port_no);
    //bind
    int rc = bind(fd,(struct sockaddr*) &addr,sizeof(addr));
    assert(rc == 0);
    //listen
    rc = listen(fd,backlog);
    assert(rc == 0);
    //now for the good stuff
    int mainfd = fd;
    struct sockaddr_in client_addr;
    while(1) {
        int sin_size = sizeof(client_addr);
        int new_fd = accept(mainfd, (struct sockaddr *)&client_addr, (socklen_t*) &sin_size);
        connfd = new_fd;
        if (new_fd == -1) {
            perror("SERVER: Accept error.");
            continue;
        }
        pthread_t _new_thread;
        pthread_create(&_new_thread, NULL, connection_handler, &new_fd);\
        usleep(50000);
```

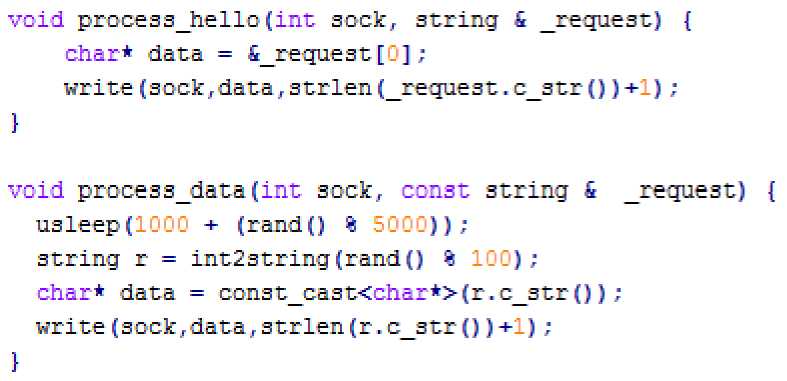
The constructor above Creates a SERVER-SIDE local copy of the channel that is accepting connection at the given port number. It opens the socket and listens for a connection.

Then we added a few functions to read and write and send requests as shown below and a destructor which closes the file descriptor.



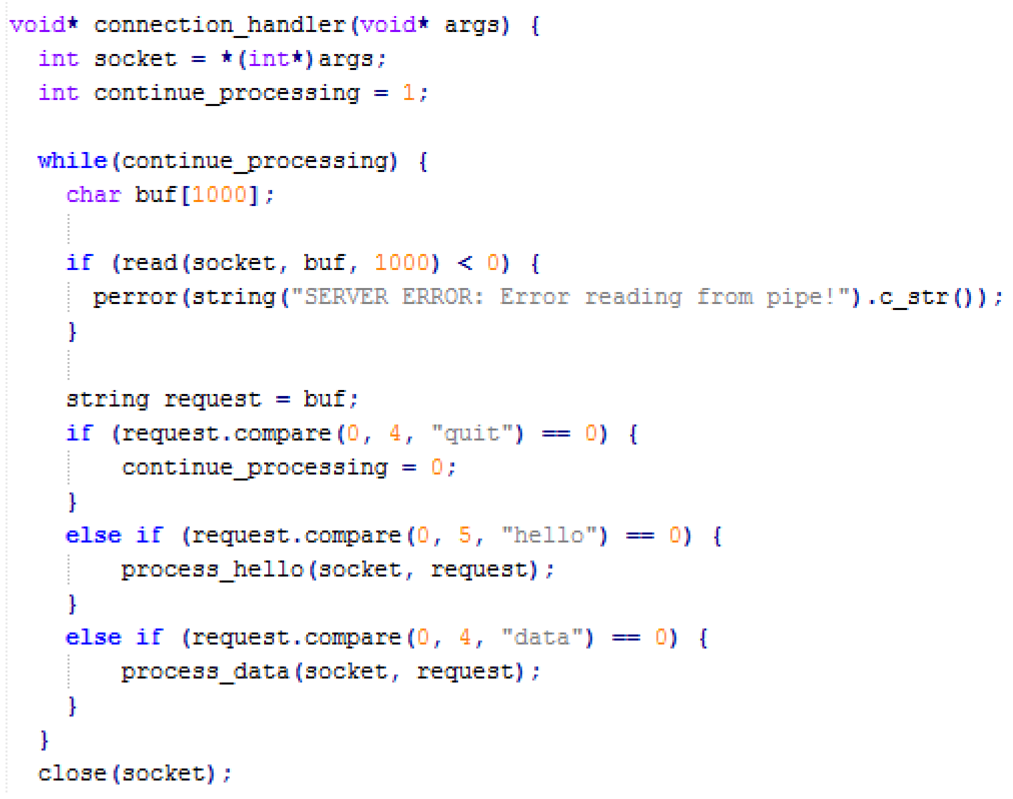
```
~NetworkRequestChannel(){
    close(connfd);
}
/* Destructor of the local copy of the channel. */
string send_request(string _request){
    cwrite(_request);
    return cread();
}
/* Send a string over the channel and wait for a reply. */
string cread(){
    string rc;
    char data[1000];
    read(connfd,data,1000);
    rc = data;
    return rc;
}
/* Blocking read of data from the channel. Returns a string of characters
read from the channel. Returns NULL if read failed. */
int cwrite(string _msg){
    char* data = &_msg[0];
    return write(connfd,data,strlen(_msg.c_str())+1);

}
/* Write the data to the channel. The function returns the number of
characters written to the channel. */
```

Then we added some stuff to the dataserver.C file so it would work with the socket implementation. The functions below basically do precisely what they are named. They both use the write function that was created above in the network request channel.



```
void process_hello(int sock, string & _request) {
    char* data = &_request[0];
    write(sock,data,strlen(_request.c_str())+1);
}

void process_data(int sock, const string &  _request) {
  usleep(1000 + (rand() % 5000));
  string r = int2string(rand() % 100);
  char* data = const_cast<char*>(r.c_str());
  write(sock,data,strlen(r.c_str())+1);
}
```

Then we did the connection handler which uses the read function made in the network request channel class and keeps reading from the pipe until quit is received then closes the socket.

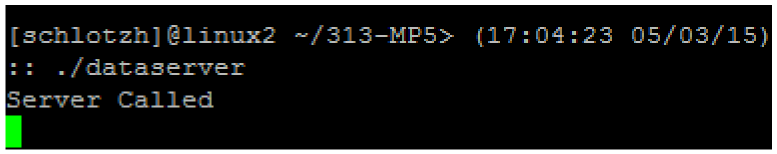


```
void* connection_handler(void* args) {
  int socket = *(int*)args;
  int continue_processing = 1;

  while(continue_processing) {
    char buf[1000];

    if (read(socket, buf, 1000) < 0) {
      perror(string("SERVER ERROR: Error reading from pipe!").c_str());
    }

    string request = buf;
    if (request.compare(0, 4, "quit") == 0) {
        continue_processing = 0;
    }
    else if (request.compare(0, 5, "hello") == 0) {
        process_hello(socket, request);
    }
    else if (request.compare(0, 4, "data") == 0) {
        process_data(socket, request);
    }
  }
  close(socket);
```

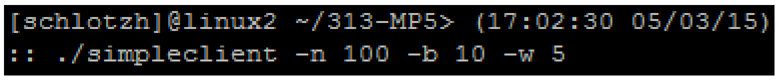
Then it was all put together in our simple client main.

Results

Our results were what were expected. In order to run the program you must have 2 terminals open and call the dataserver on one by typing ./dataserver then in the other terminal type in ./simpleclient with any parameters you want such as the following:

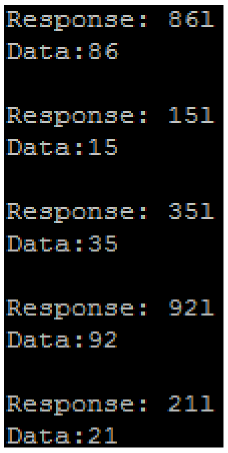


```
[schlotzh]@linux2 ~/313-MP5> (17:04:23 05/03/15)
:: ./dataserver
Server Called
```



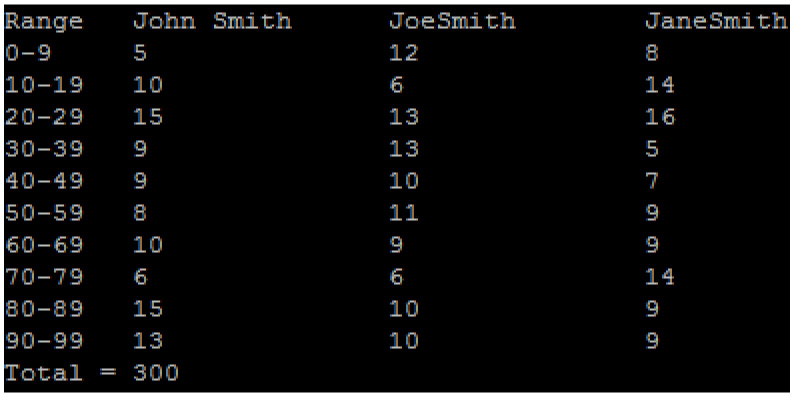
```
[schlotzh]@linux2 ~/313-MP5> (17:02:30 05/03/15)
:: ./simpleclient -n 100 -b 10 -w 5
```

Then you should get the following for how ever large n was times 3 since there are 3 clients.



```
Response: 861
Data:86

Response: 151
Data:15

Response: 351
Data:35

Response: 921
Data:92

Response: 211
Data:21
```

Then we print our histogram like this:



```
Range    John Smith       JoeSmith         JaneSmith
0-9      5                12               8
10-19    10               6                14
20-29    15               13               16
30-39    9                13               5
40-49    9                10               7
50-59    8                11               9
60-69    10               9                9
70-79    6                6                14
80-89    15               10               9
90-99    13               10               9
Total = 300
```

As you can see the total is 300 and was passed across a server which is exactly what we wanted.

Conclusion

This is a very important concept to understand and be able to work with because everything is over networks these days. I believe we handled the Machine problem the way it was asked of us and I believe we understood the problem and what needed to be done to solve it. All in all our program worked as expected and gave us good results and we understand sockets much better having done this problem and how important these types of mechanisms are in networking when trying to allow separate machines long distances apart to talk to each other and concurrently as well.