University of Engineering and Technology Lahore

Computer Networks Lab

Assignment 4 (Due Date: March 21,2022)

Three things are due (total marks of this assignment are 15):

1) Sequence and State Diagrams (5 marks):

Draw sequence and state diagrams of peer (peer.cpp). Each diagram need not exceed two pages; you are required to show what a peer has to do when it is created and how a peer communicates with another peer. While one state diagram is sufficient for entire operation of peer, multiple sequence diagrams have to be made to show various cases, e.g. one sequence diagram for successful join, one sequence diagram for redirect, etc.

These diagrams must be drawn BY HAND (print will not be accepted).

2) Running code (2 marks):

Code should compile and run on your laptop.

Using make in the lab2 folder should be sufficient to compile the code.

Run the code as described in the PDF in lab2.

If segmentation fault occurs, delete all object files and all executables, then compile and run again. If segmentatoin fault persists, it is probably due to your mistakes

Attach screenshots of output with the PDF file submission.

3) Viva (8 marks):

My questions will mainly be WHAT and WHY of the parts of lab2 that you have to develop (these parts are the blank spaces along with the comments YOUR CODE HERE in the files netimg.cpp and imgdb.cpp). The learning in this lab is acquired in doing the lab; what my viva will test is whether or not you've done the lab yourself. For this purpose, I may ask "What is this variable's type? Where it's declared and where it's used?" which is something you'd know if you've done the lab yourself. To be able to answer these, you should do the following.

Write pseudocode for the parts of lab2 that you have to develop.

Then write code from pseudocode.

If you are having difficulty in doing it completely by yourself, you may refer to solution of lab2 for help, but after studying the solution, close it then do it yourself. Simple copy-pasting will get caught in the viva.

EECS 489 Lab 2: A Peer Node

This assignment is due on Monday 21, 2022

Introduction

The majority of socket programs, including Netimg of Lab1, follows the client-server paradigm, where a server waits on a well-known port for clients' connections. In this lab, we'll explore peer-to-peer programming. A peer is basically both a server and a client. It accepts connections from other peers and also connects to one or more peers.

You're provided with a skeleton code, a single file called `peer.cpp` as part of this lab. You can download the support code from the Course Folder. The provided `Makefile` builds a program called `peer`. The program takes a single, optional argument on the command line:

```
% peer [-p <hostname>:<port>]
```

The `-p` option tells the `peer` program which peer to connect to initially. If this option is not provided, the `peer` starts as a server listening on a random, ephemeral port.

To boot strap the peer-to-peer (p2p) network, we first start a peer by itself. Everytime a peer runs, it prints to screen/console its fully qualified domain name (FQDN) and the port number it is listening on. When a peer is run with the `hostname:port` of another peer as its command line argument, the new peer tries to join the provided peer in the p2p network by creating a socket and connecting to the peer.

A peer that receives a join request will accept the peer if and only if its peer table is not full. Whether a join request is accepted or not, the peer sends back to the requesting peer the `hostname:port` of a peer in its peer table, if the table is not empty, to help the newly joined peer find more peers to join.

In completing this lab, you may consult the sample code `server.c` and `client.c` from Lab 0 and your code for Lab 1. In terms of code you have to write, the majority of it is very similar to what you did in Lab 1. Try to take a step back and look at the big picture, i.e., how the two pieces of code that were implemented in two different processes now reside in the same process and how this process is serving the role of both a client and a server. Pay particular attention to how this is accomplished by monitoring multiple sockets on a single thread. Another goal of this lab is for you to gain an early experience with protocol design. In this case, we're designing a simple peer-to-peer join protocol, with redirection.

Task 1: Server Side

Your first task is to implement the server side of a peer. You can search for the string "Task 1" in the code to find places where "Task 1" related code must be filled in. You can search for the string "YOUR CODE HERE" in the code to find places where your code must go.

If `peer` is run without any option on the command line, it calls `peer_setup(port)` with the argument `port` = 0. Fill in the function `peer_setup(port)` by first creating a TCP socket. Since we will be re-using the same port number with both a server listening socket and a client connect socket, set the socket option to allow for address reuse. Then bind the socket to the `port` passed in as argument to `peer_setup()` and listen for connection, with listen queue length set to the macro `PR_QLEN`. If `port` = 0, the OS will assign a random, ephemeral port to the socket. Finally, return the socket descriptor to the caller. The `peer_setup()` has been commented such that it should be clear where you need to make which socket API call. Depending on the error checking you do, it takes only 5 to 9 lines of code to complete this function.

Back in `main()`, if `peer` was run without any command line option, `peer_setup()` would have obtained a random, ephemeral port number on the returned socket. Find out the port number assigned to the socket and store it in the `self` variable. Next find out the name of the host the peer is currently running on. Store the name in the memory space pointed to by `pname[1]`, which we're using as scratch space. The current host name is used for printing user-friendly status messages to the console. This part of the task takes 3 to 5 lines of code.

Next call `select()` to wait for connection on the listened on socket (1 to 2 lines of code). When a connection is made, `main()` checks if its peer table is full (for this lab, we restrict the peer table size to 2). If the table is not full, `main()` calls `peer_accept()` to accept the connection and then calls `peer_ack()` to send back a welcome (`PM_WELCOME`) message with a list of all the other peers it knows of, if any (well, in this lab, "all" means the only other peer in its peer table). The new peer is then stored in the peer table. On the other hand, if the peer table is full, `main()` calls `peer_accept()` and `peer_ack()` as before, but in the call to `peer_ack()`, it sends back a redirect (`PM_RDIRECT`) message, along with the first peer it knows of, if any, and closes the connection.

The function `peer_accept(sd, pte)` accepts the connection on the socket `sd`. Since we will be sending back acknowledgement message when the peer table is full and we must close the connection, set the socket option so that the socket will linger for `PR_LINGER` amount of time upon closing to give time for the acknowledgement message to arrive at the redirected peer. This part takes 5 to 7 lines of code.

The function `peer_ack(td, type, peer)` marshalls together a message of type `pmsg_t` defined at the top of `peer.cpp`. It fills in the fields of the message: `pm_vers` must be set to `PM_VERS`, `pm_type` set to the `type` argument passed in to `peer_ack()`, if the `peer` pointer passed in is a NULL pointer, set `pm_npeers` to 0, otherwise set it to 1 and copy the address and port of the peer pointed to by `peer` to `pm_peer`. Then `peer_ack()` sends the marshalled message through the provided socket `td`. If there's any error in sending, for example, if the other side of the connection has been closed by the peer, close the connection. This part takes about 13 lines of code.

That's all for Task 1. It should take about 27 to 36 lines of code in total. After completing Task 1, you should test your code before continuing to Task 2. See the Testing section below for some guidelines on testing your code using the reference implementation of `peer`.

Task 2: Client Side

You can search for the string "Task 2" in the code to find places where "Task 2" related code must be filled in.

If peer is run with the `-p` option, the user must provide a peer hostname and a port number of the peer to connect to, with the port number separated from the peer hostname by a colon. The provided function `peer_args()` handles parsing of the command line. Upon return from the call to `peer_args()`, the peer's hostname will be stored in `pname[0]` and the port number will be stored, in network byte order, in `pte[0]`. Given the peer's hostname stored in `pname[0]`, determine the peer's IPv4 address and store it in peer table entry 0 (`pte[0]`). Then call the `peer_connect()` function, passing it a pointer to the first peer table entry. When `peer_connect()` connects to the provided peer, it will be assigned a random, ephemeral port number by the OS.

The function `peer_connect(pte)` connects to the provided peer. First create a new TCP socket, store the new socket descriptor in `pte->pte_sd`. Since we will be re-using the same port number with both a server listening socket and a client connect socket, set the socket option to allow for address reuse. Next initialize the socket with the destination peer's IPv4 address and port number with the peer table entry pointed to by the `pte` argument. Finally connect to the destination peer and return to caller. When you connect to the destination peer, the OS will assign a random, ephemeral port to the socket. If there were any error during the connect process, terminate process.

Back in `main()`, find out the assigned ephemeral port number and store it in the `self` variable, along with the IPv4 address of the current host. The function `peer_connect()` should take 5 to 10 lines of code. The code in `main()` prior to and upon return from the call to `peer_connect()` together should take about 4 to 6 lines.

At this point in `main()`, we'll be calling the `peer_setup()` function you wrote earlier in Task 1. However, instead of calling the function with `port` = 0, we'll be calling it with the random, ephemeral port number assigned by the OS when you connected with the user-provided peer. In the call to `select()`, we will be waiting for activities on the listened to socket and all connected socket(s), if any. When a message arrives from a connected peer, we call `peer_recv(td, msg)`.

The function `peer_recv(td, msg)` receives a message from the provided socket `td` into the buffer space pointed to by the provided `msg` pointer, and returns the size of the received message, which in this lab should always be `sizeof(pmsg_t)`. If there is any error in receiving the message, close the socket `td` and return to caller the error code returned by the socket receive API. This function should take about 11 lines of code. Back in `main()`, if the received packet contains another peer, print out the third peer's address and port number. If the received packet is a `PM_RDIRECT` packet, inform the user that the join has been declined (redirected). The user can manually try to connect to the third peer returned in the redirect packet.

That's all for Task 2. The total number of lines for Task 2 should be about 20 to 27 lines of code. And the total number of lines for both tasks together is about 47 to 62 lines.

Testing Your Code

We will use the same four hosts CAEN has set up for this course. Again, don't use CAEN's login server (`login.engin.umich.edu`) which will redirect you to one of `caen-vnc*` hosts as these hosts do not allow for connection to random ports. You can also run multiple peers on a single host and form p2p connections between them. When multiple peers are running on the same host, you can use `localhost` in place of the peer's hostname in the command line to `peer`.

In addition to the skeletal code and `Makefile`, we've also provided an executable binary of `peer`, called `refpeer`, that runs on CAEN eecs489 hosts. It is available on `/afs/umich.edu/class/eecs489/w15/FILES/`. As in Lab 1, this is a GNU/Linux executable, not to be downloaded nor run on your Mac OS X, Ubuntu, nor Windows machines. Remember that you can connect to the CAEN eecs489 hosts only through UMVPN and MWireless or from CAEN Lab desktops. You should test your code as soon as you completed Task 1. Use `refpeer` to connect to your `peer`. Similarly, after completing Task 2, connect your `peer` to `refpeer`. To see the expected behavior of the code, run multiple `refpeer`s and have them connect to each other.

Here is an example test scenario, assuming that you have built the program `peer` and it is residing in your working directory/folder for this lab. Create four windows on your local host.

1. On the first window, `ssh` to `eecs489p1.engin.umich.edu`, change to your working directory for this lab, run `peer` without any command line argument:

   ```
   p1% ./peer
   ```

   It should print to screen (with a different port number, depicted in bold here):

   ```
   This peer address is caen-eecs489p01.engin.umich.edu:43945.
   ```

   Note that `eecs489p1.engin.umich.edu` is an alias/CNAME for `caen-eecs489p01.engin.umich.edu`. On the four eecs489 machines, but not from your laptop, you can also refer to each of them as `p1` to `p4`.

2. On the second window, `ssh` to `eecs489p2.umich.edu`, change to your working directory for this lab, run `peer` with the following command line argument (replacing the port number with the one that got printed for you on the first item above):

   ```
   p2% ./peer -p p1:43945
   ```

   It should print to screen (with different port numbers):

   ```
   Connected to peer p1:43945
   This peer address is caen-eecs489p02.engin.umich.edu:56535
   Received ack from p1:43945
   ```

   Meanwhile, on the first window, you should see the following additional line printed to screen:

   ```
   Connected from peer p2:56535
   ```

3. On the third window, `ssh` to `eecs489p3.engin.umich.edu`, change to your working directory for this lab, run `peer` with the following command line argument (replacing the port number with the one from the first item above):

   ```
   p3% ./peer -p p1:43945
   ```

   It should print to screen (with different port numbers):

   ```
   Connected to peer p1:43945
   This peer address is caen-eecs489p02.engin.umich.edu:48141
   Received ack from p1:43435
     which is peered with: p2:56535
   ```

   Meanwhile, on the first window, you should see the following additional line printed to screen:

   ```
   Connected from peer p3:48141
   ```

4. On the fourth window, `ssh` to `eecs489p4.engin.umich.edu`, change to your working directory for this lab, run `peer` with the following command line argument (replacing the port number with the one from the first item above):

   ```
   p4% ./peer -p p1:43945
   ```

   It should print to screen (with different port numbers):

   ```
   Connected to peer p1:43945
   ```

```
This peer address is caen-eecs489p04.engin.umich.edu:40231
Received ack from p1:43945
  which is peered with: p2:56535
Join redirected, try to connect to the peer above.
```

Meanwhile, on the first window, you should see the following additional line printed to screen:

```
Peer table full: p4:40231 redirected
```

5. Staying on the fourth window, if the peer hasn't automatically exited, terminate it by entering 'q' (and hit "enter"), and run `peer` again with the following command line argument (replacing the port number with the one from the **fourth** item above):

   ```
   p4% ./peer -p p2:56535
   ```

   It should print to screen (with different port numbers):

   ```
   Connected to peer p2:56535
   This peer address is caen-eecs489p04.engin.umich.edu:50095
   Received ack from p2:56535
     which is peered with: p1:43945
   ```

   Meanwhile, on the `second` window, on `p2`, you should see the following additional line printed to screen:

   ```
   Connected from peer p4:50095
   ```

That ends our sample test scenario and you can quit all four peers.

Remember not to use any libraries or compiler options not already used in the Makefile to ensure that we will be able to build your code for grading. If we can't compile your code, you will get 0 point.

Submission Instructions

Test your compilation! Your submission must compile **without** errors.

Your "*Lab2 files*" comprises your `peer.cpp` file only.

To turn in your Lab2:

1. Email your IA/GSI the SHA1's of your *Lab2 file*. Use "EECS489: Lab2 Submission" as your email's "Subject:" line. Once you've sent in your SHA1's, don't make any more changes to the files, or your SHA1's will become invalid.
2. Upload your *Lab2 files* by pointing your web browser to [Course folder] and navigate to your `lab2` folder under your uniqname. Or you can `scp` the files to your `lab1` folder on IFS: `/afs/umich.edu/class/eecs487/w15/FOLDERS/<uniqname>/lab2/`.
   This path is accessible from any machine you've logged into using your ITCS (`umich.edu`) password. Please report any problems to ITCS.
3. Keep your own backup copy! Don't make any more changes to the files once you've submitted your final SHA1's.

The timestamp on your SHA1 email will be your time of submission. If this is past the deadline, your submission will be considered late. You are allowed multiple "submissions" without late-policy implications as long as you respect the deadline. Try not to email your SHA1 to your IA/GSI until you've finalized your code. You don't want to annoy them.

Do NOT turn in an archival (`.zip` or `.tgz`) file, instead please turn in your solution files individually. Turn in ONLY the files you have modified. Do not turn in support code we provided that you haven't modified. Do not turn in any binary files (object files, executables, or images) with your assignment.

Do remove all `printf()`'s or `cout`'s and `cerr`'s and any other logging statements you've added for debugging purposes. You should debug using a debugger, not with `printf()`'s. If we can't understand the output of your code, you will get zero point. You can keep error reporting messaages that you print out prior to terminating your code.

Computer Networks

PEER LAB (LAB 2 OF UMICH)

CODE SNAPS (Sections)

```
/*
 * peer_setup: sets up a TCP socket listening for connection.
 * The argument "port" may be 0, in which case, the
 * call to bind() obtains an ephemeral port.  If "port"
 * is not 0, it is assumed to be in network byte order.
 * In either case, listens on the port bound to.
 *
 * Terminates process on error.
 * Returns the bound socket id.
*/
int
peer_setup(u_short port)
{
  /* Task 1: YOUR CODE HERE
   * Fill out the rest of this function.
   */
  /* create a TCP socket, store the socket descriptor in "sd" */
  /* YOUR CODE HERE */
  int sd;
  struct sockaddr_in self;

  sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  /* initialize socket address */
  memset((char *) &self, 0, sizeof(struct sockaddr_in));
  self.sin_family = AF_INET;
  self.sin_addr.s_addr = INADDR_ANY;
  self.sin_port = port; // in network byte order
```

```
  /* reuse local address so that bind doesn't complain
     of address already in use. */
  /* YOUR CODE HERE */
  int on = 1;
  setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  /* bind address to socket */
  /* YOUR CODE HERE */
  bind(sd, (struct sockaddr *)&self, sizeof(struct sockaddr_in));


  /* listen on socket */
  /* YOUR CODE HERE */
  listen(sd, PR_QLEN);

  /* return socket id. */
  return (sd);
}
```

```c
/*
 * peer_accept: accepts connection on the given socket, sd.
 *
 * On connection, stores the descriptor of the connected socket and
 * the address+port# of the new peer in the space pointed to by the
 * "pte" argument, which must already be allocated by caller. Set
 * the linger option for PR_LINGER to allow data to be delivered to client.
 * Terminates process on error.
*/
int
peer_accept(int sd, pte_t *pte)
{
  struct sockaddr_in peer;

  /* Task 1: YOUR CODE HERE
     Fill out the rest of this function.
     Accept the new connection, storing the address of the connecting
     peer in the "peer" variable. Also store the socket descriptor
     returned by accept() in the pte */
  /* YOUR CODE HERE */
  int len = sizeof(struct sockaddr_in);
  int td = accept(sd, (struct sockaddr *)&peer, (socklen_t *)&len);
  pte->pte_sd = td;
```

```c
  /* make the socket wait for PR_LINGER time unit to make sure
     that all data sent has been delivered when closing the socket */
  /* YOUR CODE HERE */
  struct linger linger_time;
  linger_time.l_onoff = 1;
  linger_time.l_linger = PR_LINGER;
  setsockopt(td, SOL_SOCKET, SO_LINGER, &linger_time, sizeof(linger_time));

  /* store peer's address+port# in pte */
  memcpy((char *) &pte->pte_peer.peer_addr, (char *) &peer.sin_addr,
         sizeof(struct in_addr));
  pte->pte_peer.peer_port = peer.sin_port; /* stored in network byte order */

  return (pte->pte_sd);
}
```

```
/*
 * peer_ack: marshalls together a pmsg_t message of "type" as provided.
 * If the provided "peer" pointer is NULL, sets the number of peers to
 * 0, otherwise sets number of peers to 1 and copy the peer's address
 * and port number onto the message.  Then sends the message to the
 * peer connected at socket td.  (See the definition of pmsg_t at the
 * top of this file.)
 *
 * If there's any error in sending, closes the socket td.
 * In all cases, returns the error message returned by send().
*/
int
peer_ack(int td, char type, peer_t *peer)
{
  int err;

  /* Task 1: YOUR CODE HERE
   * Fill out the rest of this function.
   *
   * Marshall together a message of type pmsg_t.
  */
  /* YOUR CODE HERE */
  pmsg_t msg;
  msg.pm_vers = PM_VERS;
  msg.pm_type = type;
  if(peer)
  {
    msg.pm_npeers = htons(1);
```

```
  if(peer)
  {
    msg.pm_npeers = htons(1);
    memcpy((char*)&msg.pm_peer.peer_addr, (char*) &peer->peer_addr, sizeof(struct
in_addr)); // copy address
    msg.pm_peer.peer_port = peer->peer_port; // copy port
  }
  else
  {
    msg.pm_npeers = 0;
  }

  /* send msg to peer connected at socket td,
     close the socket td upon error in sending */
  /* YOUR CODE HERE */
  err = send(td, &msg, sizeof(pmsg_t), 0);
  if(err<= 0)
  {
    close(td);
  }
  return(err);
}
```

```
/*
 * peer_connect: creates a new socket to connect to the provided peer.
 * The provided peer's address and port number is stored in the argument
 * of type pte_t (peer table entry type).  See the definition of pte_t
 * at the top of this file.  The newly created socket must be stored in
 * the same pte passed in.
 *
 * On success, returns 0.
 * On error, terminates process.
 */
int
peer_connect(pte_t *pte)
{
```

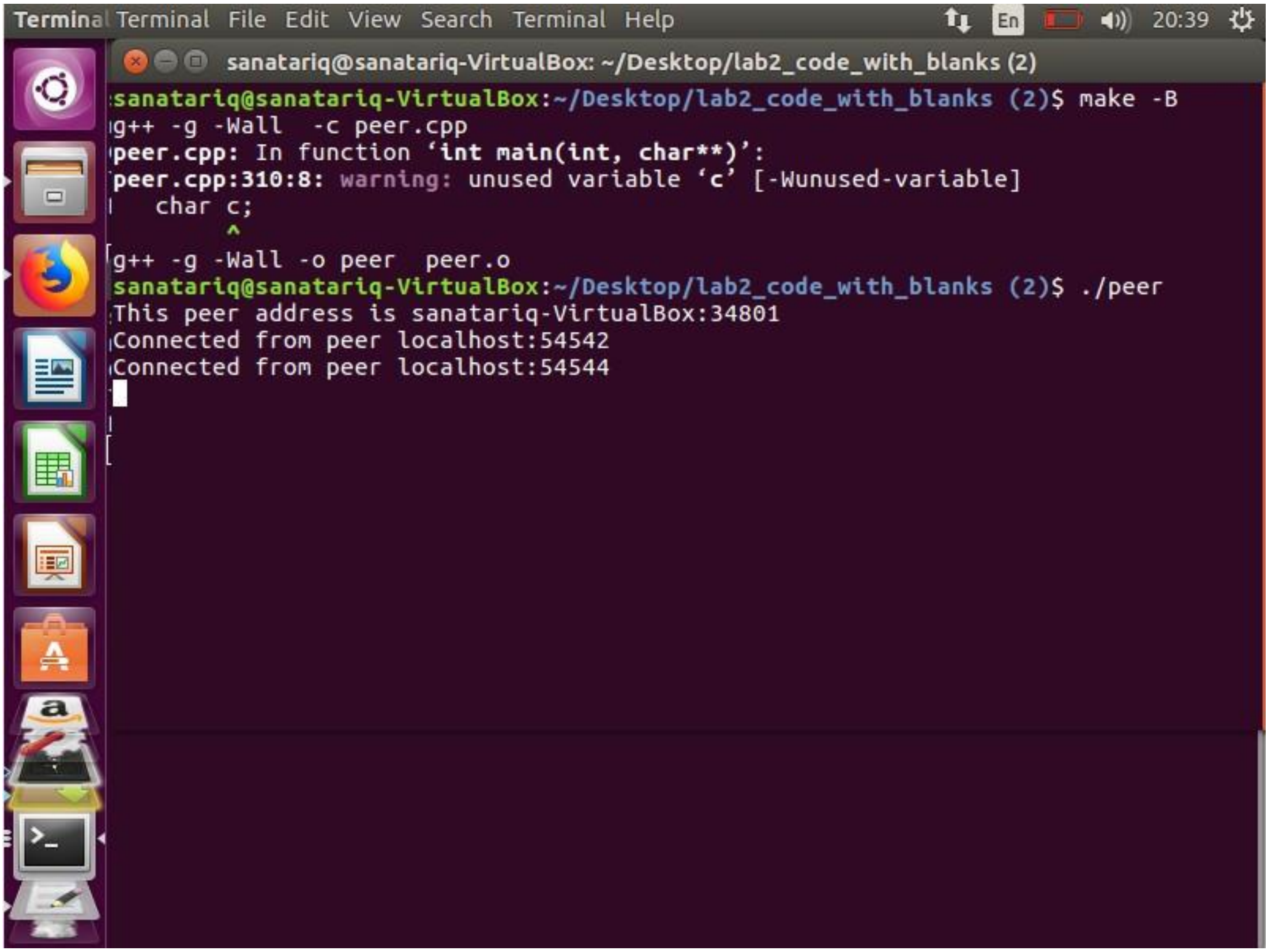
```
  /* Task 2: YOUR CODE HERE
   * Fill out the rest of this function.
  */
  /* create a new TCP socket, store the socket in the pte */
  /* YOUR CODE HERE */
  pte->pte_sd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

  /* reuse local address so that the call to bind in peer_setup(), to
     bind to the same ephemeral address, doesn't complain of address
     already in use. */
  /* YOUR CODE HERE */
  int on =1;
  setsockopt(pte->pte_sd, SOL_SOCKET, SO_REUSEADDR, &on, sizeof(on));

  /* initialize socket address with destination peer's IPv4 address and port
number . */
  /* YOUR CODE HERE */
  struct sockaddr_in server;
  memset((char *)&server, 0, sizeof(struct sockaddr_in));
  server.sin_family = AF_INET;
  server.sin_port = pte->pte_peer.peer_port;
  memcpy(&server.sin_addr, &pte->pte_peer.peer_addr, sizeof(struct in_addr));

  /* connect to destination peer. */
  /* YOUR CODE HERE */
  if(connect(pte->pte_sd, (struct sockaddr *)&server, sizeof(struct
sockaddr_in))<0)
  {
    perror("error in connecting");
```

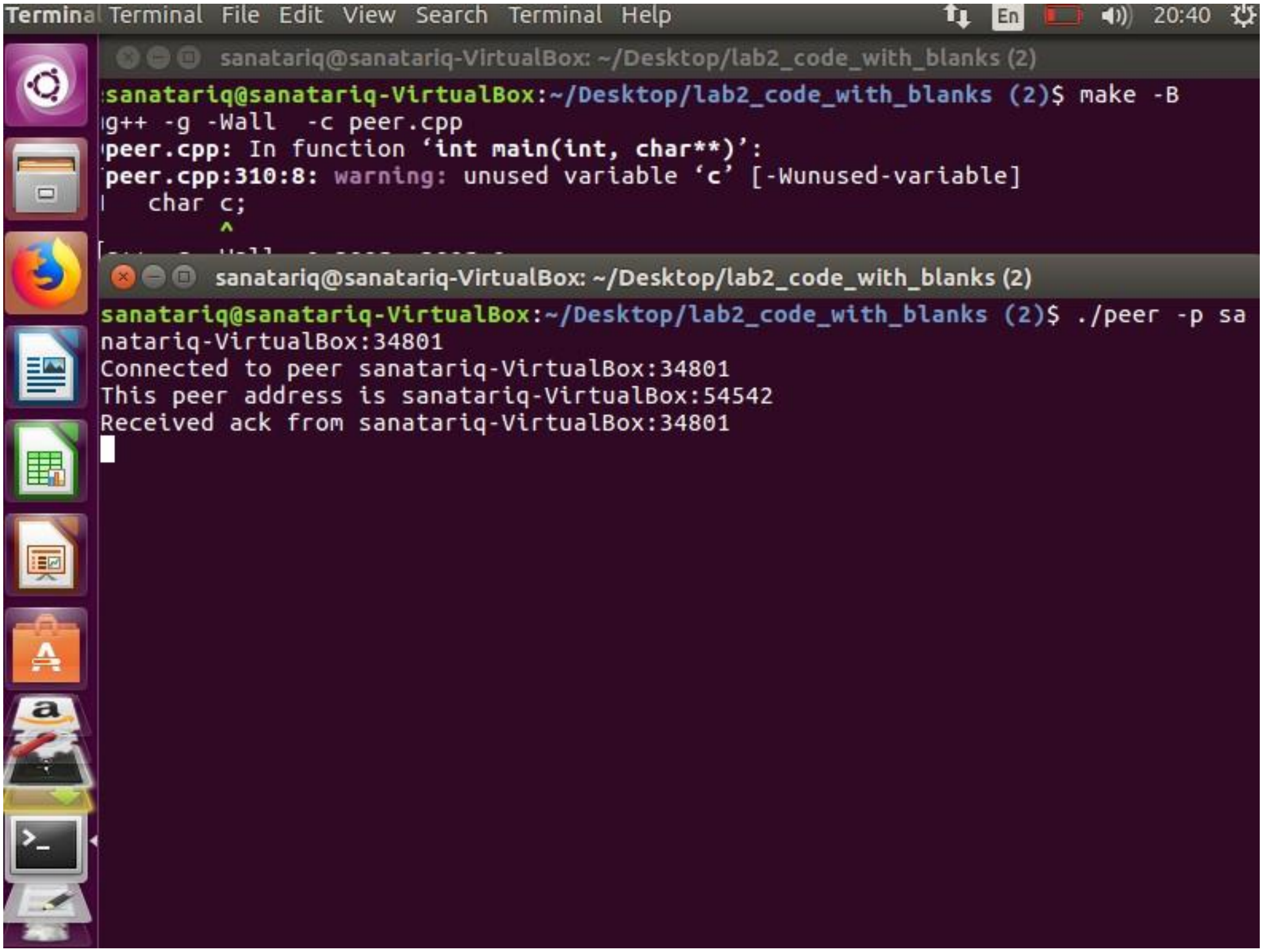
```
    exit(1);
  }

  return 0;
}
```

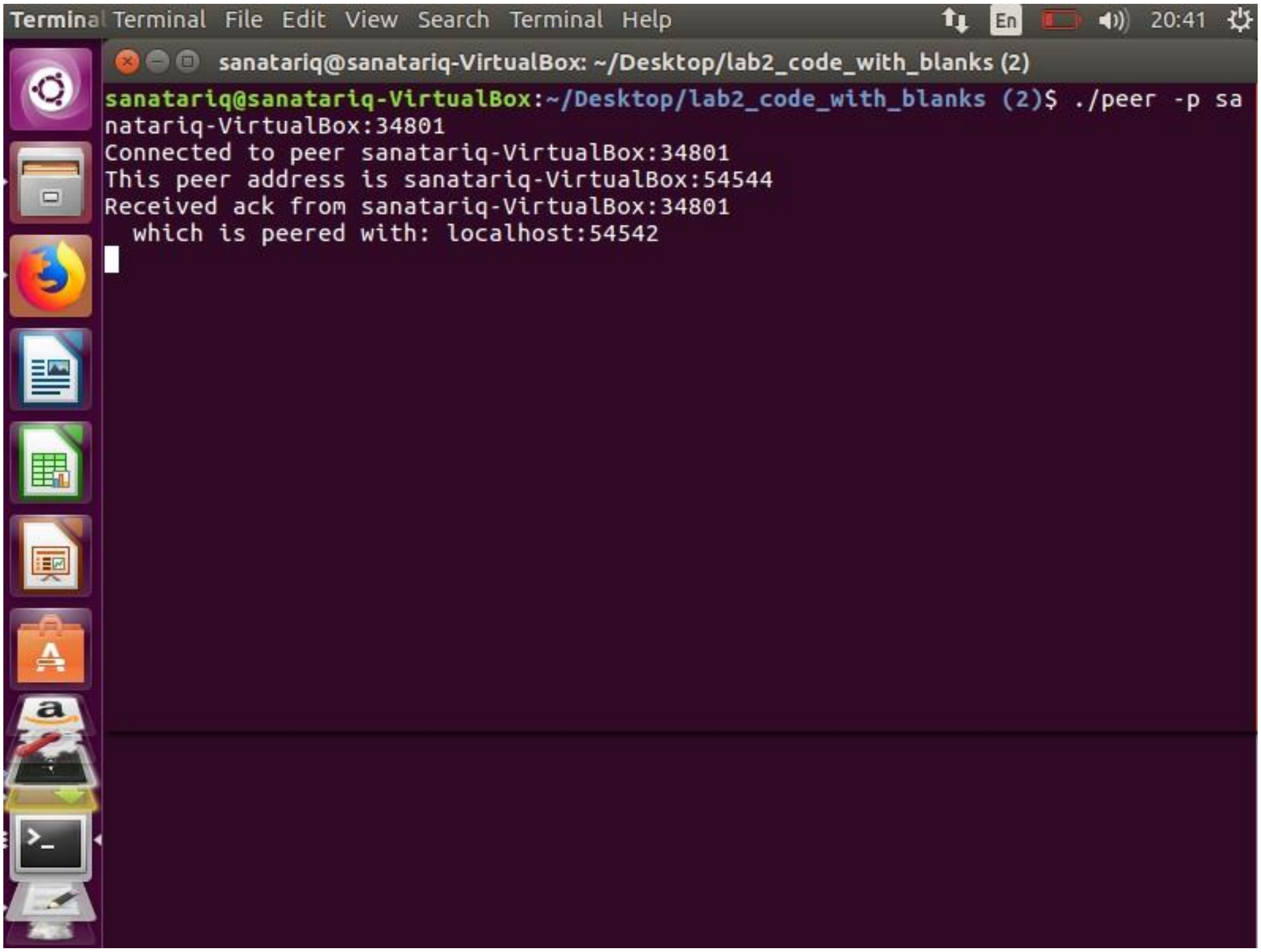
```
 * peer_recv: receives off socket td a pmsg_t message and stores it in
 * the memory pointed to by the "msg" argument, this memory must be
 * pre-allocated by the caller.  Be sure to receive all sizeof(pmgs_t)
 * worth of data.  If there's error in receiving or if the connection
 * has been terminated by the peer, closes td and returns the error.
 * Otherwise, returns the amount of data received, which should be
 * sizeof(pmgs_t) in all cases.
*/
int
peer_recv(int td, pmsg_t *msg)
{

  /* Task 2: YOUR CODE HERE
   *
   * Receive a pmsg_t message into the pre-allocated
   * memory pointed to by "msg"
  */
  /* YOUR CODE HERE */
  int status = recv(td, msg, sizeof(pmsg_t), 0);
  if(status<=0)
  {
    close(td);
    return status;
  }

  msg->pm_npeers = ntohs(msg->pm_npeers);
  return (sizeof(pmsg_t));
}
```

```
    /* Task 2: YOUR CODE HERE
     *
     * Upon return from peer_connect(), the socket descriptor in
     * pte[0] should have been initialized and connected to the peer.
     * Obtain the ephemeral port assigned by the OS kernel to this
     * socket and store it in the variable "self" (along with the
     * peer's address).
    */
    /* YOUR CODE HERE */
    int len = sizeof(struct sockaddr_in);
    getsockname(pte[0].pte_sd, (struct sockaddr *)&self, (socklen_t *)&len);
    // getsockname() returns the current address to which the socket is bound,
copy the addr to second arg
    npeers++;

    /* inform user of connection to peer */
    fprintf(stderr, "Connected to peer %s:%d\n", pname[0],
            ntohs(pte[0].pte_peer.peer_port));
  }

  /* setup and listen on connection */
  sd = peer_setup(self.sin_port);  // Task 1: fill in the peer_setup() function
above
  if (!self.sin_port)
  {
    /* Task 1: YOUR CODE HERE
       Call select() to wait for any activity on any of the above
       descriptors. */
    /* YOUR CODE HERE */
    struct timeval t_value;
    t_value.tv_sec = 2;
    t_value.tv_usec = 500000;
    select(maxsd+1, &rset, NULL, NULL, &t_value);

    if (FD_ISSET(sd, &rset))
    {
      // a connection is made to this host at the listened to socket
      if (npeers < PR_MAXPEERS)
      {
        /* Peer table is not full.  Accept the peer, send a welcome
         * message.  if we are connected to another peer, also sends
         * back the peer's address+port#
         */
        // Task 1: fill in the functions peer_accept() and peer_ack() above
        peer_accept(sd, &pte[npeers]);
        err = peer_ack(pte[npeers].pte_sd, PM_WELCOME,
                       (npeers > 0 ? &pte[0].pte_peer : 0));
        err = (err != sizeof(pmsg_t));
        net_assert(err, "peer: peer_ack welcome");
        pte[npeers].pte_pname = pname[npeers];
```

Terminal Results




Terminal Terminal File Edit View Search Terminal Help
20:40
sanatariq@sanatariq-VirtualBox: ~/Desktop/lab2_code_with_blanks (2)
sanatariq@sanatariq-VirtualBox:~/Desktop/lab2_code_with_blanks (2)$ make -B
g++ -g -Wall -c peer.cpp
peer.cpp: In function 'int main(int, char**)':
peer.cpp:310:8: warning: unused variable 'c' [-Wunused-variable]
char c;
sanatariq@sanatariq-VirtualBox: ~/Desktop/lab2_code_with_blanks (2)
sanatariq@sanatariq-VirtualBox:~/Desktop/lab2_code_with_blanks (2)$ ./peer -p sanatariq-VirtualBox:34801
Connected to peer sanatariq-VirtualBox:34801
This peer address is sanatariq-VirtualBox:54542
Received ack from sanatariq-VirtualBox:34801


Terminal Terminal File Edit View Search Terminal Help
En
20:41
sanatariq@sanatariq-VirtualBox: ~/Desktop/lab2_code_with_blanks (2)
sanatariq@sanatariq-VirtualBox:~/Desktop/lab2_code_with_blanks (2)$ ./peer -p sanatariq-VirtualBox:34801
Connected to peer sanatariq-VirtualBox:34801
This peer address is sanatariq-VirtualBox:54544
Received ack from sanatariq-VirtualBox:34801
which is peered with: localhost:54542