

CS4150 Computer Networks Laboratory – Assignment 8

Subhash S – 111801042
Computer Science and Engineering, IIT Palakkad

Common Definitions

1. `get_addr`
Depending on the input IP and port, it populates the `sockaddr_in` which is used to identify the server or the client.
2. `tokenize`
Give a string and a delimiter, it returns a list of strings by splitting them on the delimiter. This is used to separate various components of messages in `h2` and `h3`.
3. `try_socket_creation`
Tries to create a socket, exits if it fails
4. `try_accept`
Tries to accept an incoming TCP connection, exits if it fails
5. `try_bind`
Tries to bind a socket descriptor to the given address, exits if it fails
6. `set_socket_options`
Sets various socket options to reuse ports and ignore the timeouts to reuse them. This allows us to disconnect and connect dynamically.

Apart from the above-mentioned ones, there are several macros each of which is self-explanatory by their names.

Files

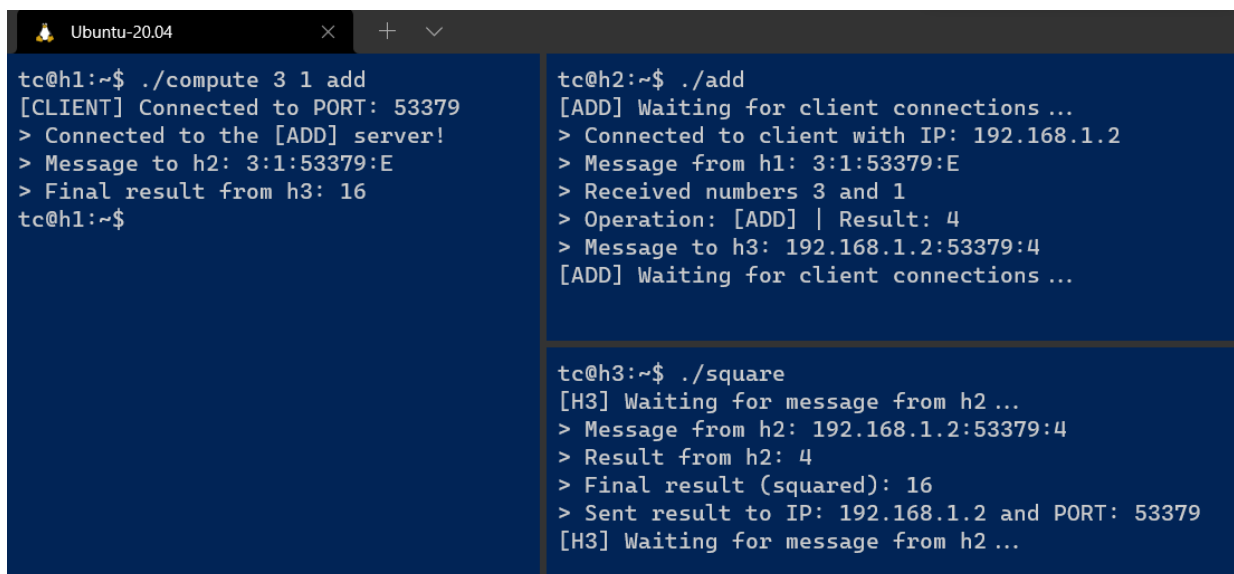
H1: `compute.cpp common.h Makefile`

H2: `compute.cpp inetd.cpp add.cpp sub.cpp mul.cpp idiv.cpp inetd_add.cpp
inetd_sub.cpp inetd_mul.cpp inetd_idiv.cpp common.h Makefile`

H3: `compute.cpp square.cpp common.h Makefile`

Complete Flow

Figure 1: Complete flow of a request (client -> h2 -> h3 -> client).



```
tc@h1:~$ ./compute 3 1 add
[CLIENT] Connected to PORT: 53379
> Connected to the [ADD] server!
> Message to h2: 3:1:53379:E
> Final result from h3: 16
tc@h1:~$

tc@h2:~$ ./add
[ADD] Waiting for client connections ...
> Connected to client with IP: 192.168.1.2
> Message from h1: 3:1:53379:E
> Received numbers 3 and 1
> Operation: [ADD] | Result: 4
> Message to h3: 192.168.1.2:53379:4
[ADD] Waiting for client connections ...

tc@h3:~$ ./square
[H3] Waiting for message from h2 ...
> Message from h2: 192.168.1.2:53379:4
> Result from h2: 4
> Final result (squared): 16
> Sent result to IP: 192.168.1.2 and PORT: 53379
[H3] Waiting for message from h2 ...
```

The figure shows the logs on the client, `h2` (add server) and `h3` (clockwise from top-left).

Each of this is explained clearly in the following sections.

Client

File: compute.cpp

Aim

1. Read input numbers and operation from the command line.
2. Send the number to the appropriate server on h2 depending on the operation.
3. Spawn a server and receive the final result from h3.

Program

1. Depending on the operation specified by the client, I decide the port on h2 to connect to.
2. I create a socket for the client and server and bind the server socket to a specific port. The binding is done here to have the port ready for h3 to connect.
3. The *connect* function is used to connect to h2 on the port that was obtained from 1.
4. Now I send the two numbers, the port and end of message symbol delimited by a colon to the connection obtained in 3.
5. Now the server socket tries to accept connections from h3 in a blocking manner.
6. Once a connection is accepted from h3, the number is read and output to the console.

Protocol Flow

1. Client -> H2 uses TCP.
2. H2 -> H3 uses UDP.
3. H3 -> Client uses TCP.

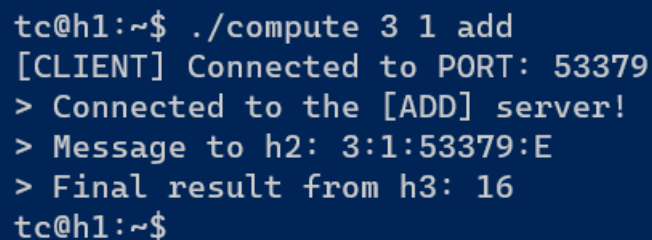
Example

If the user executes `./compute 3 1 add`, the following happens:

In 4 above, the client sends a message "3:1:53379:E" to h2 on port 12500 (add server).

In 6 above, the client receives a message "16" from h3.

Figure 2: The flow as seen from the client(h1)



```
tc@h1:~$ ./compute 3 1 add
[CLIENT] Connected to PORT: 53379
> Connected to the [ADD] server!
> Message to h2: 3:1:53379:E
> Final result from h3: 16
tc@h1:~$
```

The above image shows the entire flow with clear logs.

1. The *client* connects to the *add* server (since the client sent "add" as the last argument).
2. The *client* sends the two numbers as "3:1:53379:E" to the *add* server.
3. The *client* binds to port 12500 (not hard-coded in h2 and h3) and accepts TCP connections.
4. The client receives the message from *h3* and prints it.

Server on H2 (Add)

File: *add.cpp*

Note: Although this section focuses on *add.cpp*, the other files are the same except for their operations.

Aim

1. Read an incoming message from the client and get the two numbers.
2. Perform the appropriate operation (addition in this case).
3. Determine the client's IP address and port.
4. Send clients' IP address, port and the result of the operation to h3.

Program Flow

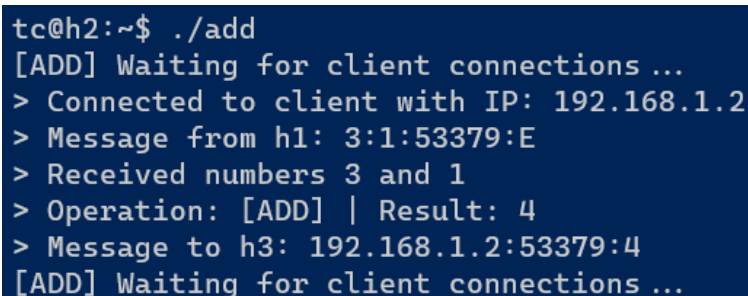
1. Create a TCP socket to connect with the client.
2. Create a UDP socket to connect with h3.
3. Populate the address structures for h2 and h3.
4. Bind h2 to the correct port (12500 in this case) to accept connections from the client.
5. Start listening on the port for incoming TCP connections.
6. Forever,
 - a. Accept an incoming client connection in a blocking manner.
 - b. Find the client's IP address and port and store it in local variables.
 - c. Get the two numbers and port from the client. They will be of the form "num1:num2:port:E".
 - d. Perform the appropriate operation on them (addition in this case).
 - e. Send the message to h3 in the form "client_ip:client_port:result" where the result is the sum of *num1* and *num2* in this case.
 - f. Go back to a and start listening for more connections.

Example

If the user executes `./compute 3 1 add`, the following happens:

The add server on h2 receives the message "3:1:53379:E" and detects that the client IP is 192.168.1.2 and it is running on port 53379 so it sends h3 the message "192.168.1.2:53379:4".

Figure 3: The flow as seen from h2.



```
tc@h2:~$ ./add
[ADD] Waiting for client connections ...
> Connected to client with IP: 192.168.1.2
> Message from h1: 3:1:53379:E
> Received numbers 3 and 1
> Operation: [ADD] | Result: 4
> Message to h3: 192.168.1.2:53379:4
[ADD] Waiting for client connections ...
```

The above image shows the entire flow with clear logs.

1. The add server waits for client connection and accepts one from h1.
2. It determines the client IP and port from which it received the request, stores and prints it.
3. It receives the two numbers from h1 (delimited by a colon) and performs the operation.
4. Sends the message to h3 which contains the IP address and port of h1 and the result i.e. 4.

Server on H3

File: compute.cpp

Aim

1. Read an incoming message from h2, find the client IP, port and the result of the operation.
2. Connect to the client using the IP and port and send the square of the result from h2.

Program Flow

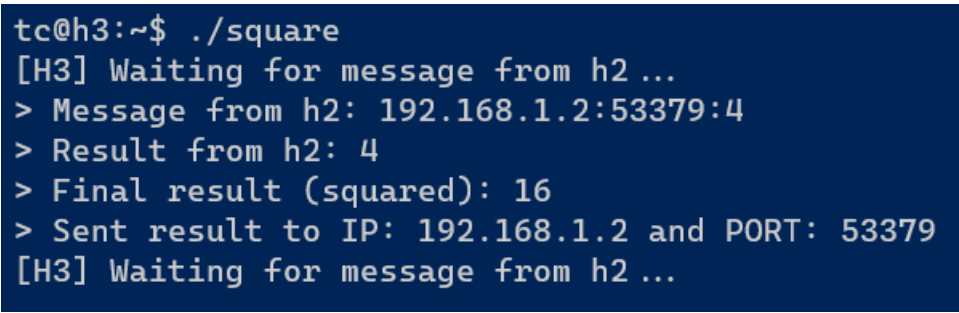
1. Create a UDP socket to get messages from h2. I will be creating a socket to connect to clients on demand if h3 receives a message.
2. Bind h3 to a port to accept UDP messages. I chose to go with port 12504.
3. Forever,
 - a. Receive a message from h2 in a blocking manner.
 - b. Get the client IP address, port and the result from h2, all these are delimited by a colon. This can be seen in Figure 4.
 - c. Square the number obtained in the above step and store it.
 - d. Connect to the client (h1 in this case) using the IP and port obtained in b.
 - e. If the connection was successful, send the result obtained in c to the client.
 - f. Go back to a

Example

If the user executes `./compute 3 1 add`, the following happens:

The UDP server on h3 gets the message "192.168.1.2: 53379:4" from h2. It is of the form "IP:PORT:RESULT" where the result is the sum of the two numbers in this case. The server on h3 then squares the number, connects to the client with the given port and IP and sends the result i.e., 16 to the client.

Figure 4: The flow as seen from h3.



```
tc@h3:~$ ./square
[H3] Waiting for message from h2 ...
> Message from h2: 192.168.1.2:53379:4
> Result from h2: 4
> Final result (squared): 16
> Sent result to IP: 192.168.1.2 and PORT: 53379
[H3] Waiting for message from h2 ...
```

The above image shows the entire flow with clear logs.

1. The server on h3 receives the message from h1.
2. It squares the result and prints it.
3. It connects to the client depending on the IP address and port it received from h2 and sends the final result to the client.
4. After sending the result, it goes back to waiting for further messages.

INETD Daemon

File: *inetd.cpp*

Aim

1. Listen on all four ports (12500, 12501, 12502 and 12503) on h2 for client connections.
2. If a connection is received on a port, then redirect the socket to stdin and start the process depending on the port.

Program Flow

1. The first few lines store all the programs, ports and ids in arrays. Since each port has a very similar startup code, this helps in not repeating the code for each operation.
2. Line 26-32 is responsible for creating sockets and listening for TCP connections on behalf of all the four servers on h2.
3. Then I initialize *fd_set* which stores a set of file descriptors that are *ready* for some class of I/O operation (TCP connection in this case).
4. Forever,
 1. Set all the file descriptors using *FD_SET*, this is done because select listens for *read* operations on the descriptors that are set in the *fd_set*.
 2. Then I call select, instructing it to listen for *read* operations (connection from a client) by setting all other parameters to NULL.
 3. If select returns with a positive value, I go through each of the socket descriptors to check which one is set in *fd_set*, this tells the socket that received the connection request.
 4. Once the socket has been identified, I try to accept the connection from the client by calling the *try_accept*.
 5. After accepting a connection, I find the IP address and port of the client and store it.
 6. Now I fork a process and do the following:
 - i. If this is the child process, I duplicate the socket descriptor obtained in 4 to the *stdin* of the child process. This lets the child process read data from the client from stdin. I close the socket descriptor and run the appropriate program depending on the port that received the connection. For ex. If the connection was requested on port 12501, it means the client wants a subtract operation so the *inetd* daemon calls *execl* with the *inetd_sub* binary and passes the client IP as input argument.
 - ii. If this is the parent process (*inetd* daemon) then I close the socket descriptor and wait for the child to exit before moving to the next step.
 7. Go back to 1 and start over again.

Server on h2 that uses inetd

File: *inetd_add.cpp*

Note: Although this section focuses on *inetd_add.cpp*, the other files are the same except for their operations.

Aim

1. Read the message from the client from *stdin*. The *inetd* daemon takes care of redirecting the input to *stdin* after the fork.
2. Perform the appropriate operation (addition in this case).
3. Send clients' IP address, port and the result of the operation to h3.

Program Flow

1. On a connection for add operation, the *inetd* daemon forks a child, duplicates the socket descriptor onto this server's *stdin* and executes *inetd_add* by passing client IP address and port as command-line arguments.
2. Read the message from the client from the standard input. It will be of the form "num1:num2:port:E", then perform the appropriate operation (addition in this case) and store the result.
3. Create a UDP socket to send the result and client details to h3.
4. Populate the address structure to point to the address of h3.
5. Send the final result along with client details to h3. The message will be in the form "client_ip:client_port:result" where the result is the sum of *num1* and *num2* in this case.

Example

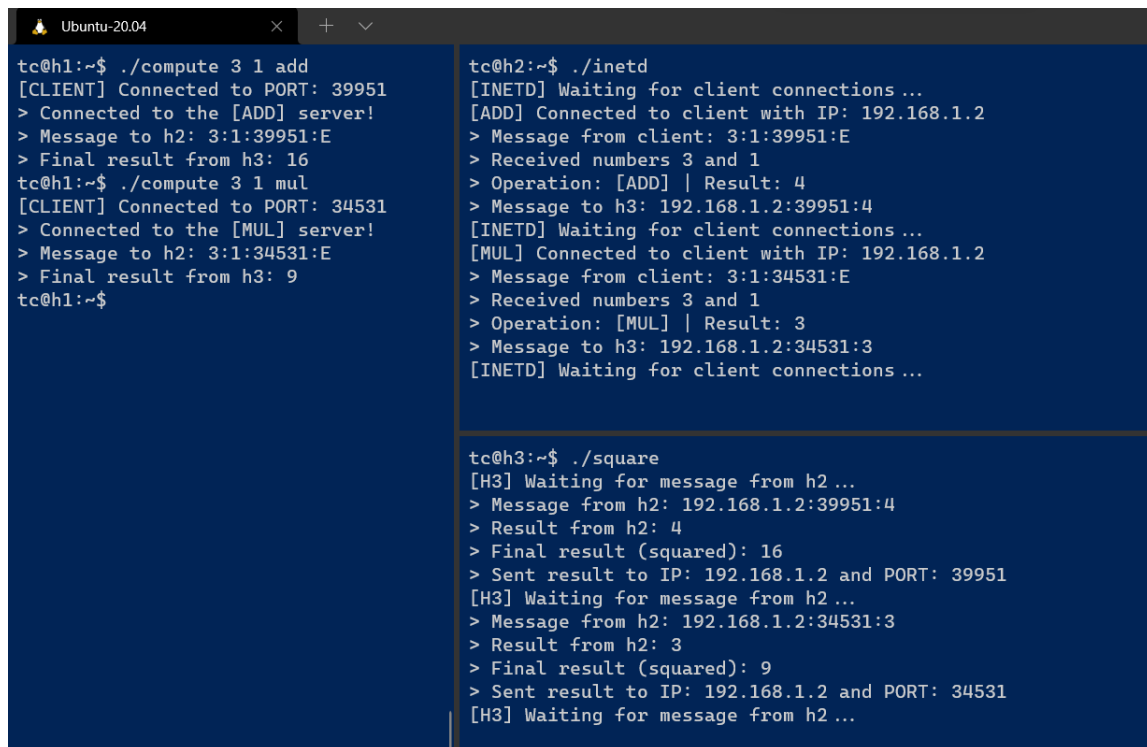
If the user executes `./compute 3 1 add`, the following happens:

The *inetd* daemon receives the connection requests and it creates the *inetd_add* server as explained above. The server gets the client IP and port from *inetd* and it reads the numbers from standard input, performs the addition and sends h3 the message "192.168.1.2:39951:4".

Figure 5: The *inetd* daemon spawning servers on-demand.

```
tc@h2:~$ ./inetd
[INETD] Waiting for client connections ...
[ADD] Connected to client with IP: 192.168.1.2
> Message from client: 3:1:39951:E
> Received numbers 3 and 1
> Operation: [ADD] | Result: 4
> Message to h3: 192.168.1.2:39951:4
[INETD] Waiting for client connections ...
[MUL] Connected to client with IP: 192.168.1.2
> Message from client: 3:1:34531:E
> Received numbers 3 and 1
> Operation: [MUL] | Result: 3
> Message to h3: 192.168.1.2:34531:3
[INETD] Waiting for client connections ...
```

Figure 6: INETD daemon spawning correct server on demand.



```
tc@h1:~$ ./compute 3 1 add
[CLIENT] Connected to PORT: 39951
> Connected to the [ADD] server!
> Message to h2: 3:1:39951:E
> Final result from h3: 16
tc@h1:~$ ./compute 3 1 mul
[CLIENT] Connected to PORT: 34531
> Connected to the [MUL] server!
> Message to h2: 3:1:34531:E
> Final result from h3: 9
tc@h1:~$

tc@h2:~$ ./inetd
[INETD] Waiting for client connections ...
[ADD] Connected to client with IP: 192.168.1.2
> Message from client: 3:1:39951:E
> Received numbers 3 and 1
> Operation: [ADD] | Result: 4
> Message to h3: 192.168.1.2:39951:4
[INETD] Waiting for client connections ...
[MUL] Connected to client with IP: 192.168.1.2
> Message from client: 3:1:34531:E
> Received numbers 3 and 1
> Operation: [MUL] | Result: 3
> Message to h3: 192.168.1.2:34531:3
[INETD] Waiting for client connections ...

tc@h3:~$ ./square
[H3] Waiting for message from h2 ...
> Message from h2: 192.168.1.2:39951:4
> Result from h2: 4
> Final result (squared): 16
> Sent result to IP: 192.168.1.2 and PORT: 39951
[H3] Waiting for message from h2 ...
> Message from h2: 192.168.1.2:34531:3
> Result from h2: 3
> Final result (squared): 9
> Sent result to IP: 192.168.1.2 and PORT: 34531
[H3] Waiting for message from h2 ...
```

The above figure shows the client, h2 and h3 (clockwise from left) receiving and sending messages.

We can see that the first operation is *add* and from the top-right image, we can see that the *inetd* service handed over the control to the *add* server. The second operation is *mul* and from the top-right image, we can see that the *inetd* service handed over the control to the *mul* server.

In both the operations, the h3 keeps listening for the message from h2 and it sends the squared result from h2 to the client depending on the IP address and port sent by h2.

We can also see that each time the client connects it uses a different port, this shows that h3 can handle requests from clients with any IP address and port and that h3 is not aware of the client until h2 tells h3 about it.

Instructions to Run

1. Run the **make** command in the host machine. This will copy the required files from the host to each of the VMs and compile all the files inside the VMs using *expect* command.
2. After the above step, you can connect to each of the machines using the **connect.sh** script and start running the required binaries.

Other optional targets:

make copy: Copy required files to the VM.

make compile: Enter each VM using *expect* and run **make local**

make local: Compile the source files locally (From the machine in which the command is run).

make clean: Enter each VM using *expect* and run **make clean_local**.

make clean_local: Remove compiled files locally.