**EE414 Embedded System**

**20180745 – Tung-Duong Mai**

**Lab 5. Network**

1. Purpose:

Understand how to use Ethernet and program a typical network program using sockets, and implement a remote user command input for the metronome.

Design and implement a metronome which can respond to the remote user command input from remote PC or notebook via network, specifically using Ethernet stream socket using TCP..

1. Experiment sequence:
2. Setup cross development environment:
3. Connection

Connect Beaglebone with Ethernet cable and Power adapter.

Connect Beaglebone with Linux PC by USB cable.

By this, the PC can access the Beaglebone via the network or USB.

1. Start PC NFS server:

After this step, PC can act as NFS server and it waits for Beaglebone to mount as an NFS client.

1. Connect to Beagle bone:

Using minicom or ssh

I prefer ssh connection method since it is more reliable (I was worried that the stale lock file might happen again) and more intuitive (I can execute directly from the PC terminal without switching to minicom interface). The result I showed is from ssh connection.

However, for complete experiment and item 1 in the discussion, I also retest all experiments with minicom.

1. Start Beaglebone nfs client

I switch to root user to have enough permission to run sysfs command (I tried give gpio permission to john (my user) but there is some incompatibility, so I think it is the best to just run in root)

Now, the Beaglebone can access the directory on PC that we setup in the previous lab. We are ready for cross development.

NFS or scp both work correctly. However, I think NFS is more intuitive since it gives me the sentiment that the data are shared and I can work as if the data is locally stored (for scp, I have to explicitly transfer the data).

1. Test stream socket: client.c and server.c

Purpose: We check the code and operation of stream socket.

Code skimming and document reading:

* The server opens a stream socket, setup the socket options. Then, a socket binds to the socket, and start listening to any client attempt to connect (in a loop). After detecting connect request, the server will accept the connect and create a new process (parent process remains in loop for possibly more connect requests).
* The client gets the host information, opens a stream socket and connect to the server. Upon acceptance from server, the client can communicate with the server.
* Client and server then can use send and recv to send and receive packet

Code comprehension: Then, I dissect the program to understand the code:

* Open a socket: int socket(int domain, int type, int protocol);

socket() creates an unbound socket in a communications domain, and return a file descriptor associated to the socket.

Arguments:

domain: the communications domain in which a socket will be created. AF\_INET for IPv4, AF\_INET6 for IPv6.

type: the type of socket. SOCK\_STREAM (TCP), SOCK\_DATAGRAM (UDP), SOCK\_

protocol: a particular protocol to be used with the socket. Protocol 0 means an unspecified default protocol.

These information is available by calling getaddrinfo(), which translates the name of a service location (for example, a host name) and/or a service name and shall return a set of socket addresses and associated information. By using a member of this set, we can open the socket.

getaddrinfo(const char \*restrict node,

const char \*restrict service,

const struct addrinfo \*restrict hints,

struct addrinfo \*\*restrict res)

We give information by hints to restrict the returned set, with family AF\_UNSPEC for unspecified domains and SOCK\_STREAM

We can use AF\_UNSPEC since TCP will retry if transmission fails.

* Setup socket options

int setsockopt(int sockfd, int level, int optname, const void \*optval, socklen\_t optlen);

It setups the socket associated with sockfd with level (SOL\_SOCKET for socket level), optval and optlen is used to access option values.

* Bind the socket: int bind(int sockfd, const struct sockaddr \*my\_addr, socklen\_t addrlen);

Bind() assigns the address specified by addr to the socket associated with sockfd

* Listen listen for socket connections:

int listen(int socket, int backlog);

listen() listen for socket connections requests to the socket associated with sockfd

* Connect to a socket:

int connect(int sockfd, const struct sockaddr \*saddr, socklen\_t addrlen);connect() connects the address specified by addr to the socket associated with sockfd.

* Accept the connection:

int accept(int sockfd, struct sockaddr \*restrict addr, socklen\_t \*restrict addrlen);

accept() accepts connection on socket associated with sockfd. It extracts the first connection request on the queue of pending connections for sockfd, creates a new connected socket, and returns a new file descriptor referring to that socket.

* Send: ssize\_t send(int sockfd, const void \*buf, size\_t len, int flags);

Send data in buffer buf from the socket associated with sockfd.

* Receive: ssize\_t recv(int sockfd, void \*buf, size\_t len, int flags);

Receive data and write to buffer on the socket associated with sockfd

1. Test datagram socket:

Code skimming and document reading:

* The server opens a stream socket, setup the socket options. Then, a socket binds to the socket
* The client gets the host information, opens a stream socket.
* Client and server then can use sendto and recvfrom to send and receive packet

(Connectionless -> much simpler)

Code comprehension: Then, I dissect the program to understand the code

Functions are basically similar with stream socket, with some difference:

Socket setup: We give information by hints to restrict the returned set, with family AF\_INET for IPv4 (or AF\_INET6 for IPv6) and SOCK\_DGRAM

We cannot use AF\_UNSPEC since UDP will not retry if transmission fails. If the server and client use different domain, data will not be received.

Due to connectionless nature, we need to specify the destination/source to send/receive data.

* Use sendto() to additionally specify the destination:

int sendto(int sockfd, const void \*msg, int len, unsigned int flags,

const struct sockaddr \*to, socklen\_t tolen);

* Use recvfrom() to additionally specify the source

int recvfrom(int sockfd, void \*buf, int len, unsigned int flags,

struct sockaddr \*from, int \*fromlen);

1. Test bidirectional stream socket:
2. Necessary functionality:
3. Synchronous:

Ensure synchronization.

I test send and receive communication 5 times each

1. Asynchonous:

Main and thread works at different rate. I test main communication 5 times with 1 Hz, and thread communication 10 times with 2 Hz.

Tunable main communication rate according to user input.

1. Design choice:
2. Synchronous

* Simply use a loop in both client and server (since the work in the same frequency, we just need to synchronize the starting point)
* Use sleep to do timing

1. Asynchronous

* Use thread to separate send and receive

Main: Client sends a message and server receives the message.

Thread: Server sends messages and client receives messages in higher rate.

* Use argv to set the frequency of main
* Use sleep for timing

1. Consideration:
2. Synchronous:

Allow time to send reply after the last recv by sleeping for a bit. The synchronization is already achieved with the same frequency.

1. Asynchronous:

I set a global variable (initially 0) activated to 1 only after server recv the first cmd packet.

In the reply in thread only, I wait until activated is not 0.

Allow time to send reply after the last recv by sleeping for a bit.

1. Test metronome
2. Necessary functionality:

* Mode manipulation by keyboard: Using non-blocking mode as in lab 3.
* LED manipulation by mmap: Reuse the code Metronome\_tui\_thread.c in lab 3.
* Main function: Setting necessary parameters, setting the interrupt and setting
* Interrupt handler function: Handle the interrupt (timer interrupt) by appropriate function and update interrupt parameters. This is the link between main program and LED manipulation.

Note that Ctrl+C interrupt is already handled in lab 2-3.

* Server routines: Basic server function: open a socket and bind it, accept a connection, main routine (receiving the command packet)
* Client routines: Basic client function: open a socket and connect it to server, main routine (sending command packet), thread routine (receiving the strength signal)

1. Design choice:

I design 7 (8 with PC simulation) code files:

Server side:

* Metro\_server.c for main server loop and interrupt handler
* userLEDmmap.h for interfacing between gpio and metro\_server
* server\_routine\_gpio.h for server routines
* Gpio\_led\_fu.c / gpio\_led\_fu\_sim for LED manipulation (real and simulation)

Client side:

* Metro\_client.c for main client loop and threading management
* Key\_input\_fu for keyboard mode manipulation
* client\_routine.h for client routines

1. Build the algorithm:
2. Server side:

Three main-parts

1. Initialization: We need to initialize these parameters

Init GPIO LED

Init HR timer

Make signal handler for HR timer

Init stream socket

Set default values to parameters (TimeSig 3 (3/4) , Tempo 90, Stop

2. Loop:

Wait accept and connection.

In Child process (fork so that parents remain listening to new connections)

a. Get the command packet

b. Enable HRTimer: Play Metronome via sig\_handler

c. Print single line message: Input & Status (without linefeed)

3. Cleanup:

- Print quit message

- Close socket descriptor

- Un-mmap GPIO

1. Client side:

Three main-parts

1. Initialization: We need to initialize these parameters

Init client

Init thread

Init key processing - Set termios - Print title and menu – metronome processing

Set default values to parameters (TimeSig 3 (3/4) , Tempo 90, Stop) and print default values

2. Loop:

- Wait for key press and update the mode depending on input key. Arm or disarm the timer when appropriate.

- Print single line message: Input & Status (Without linefeed)

3. Cleanup:

- Print quit message

- Reset termios

- Close socket descriptor, join (exit) thread

1. Code design:

Divide main function to server and client: detail in (c). We essentially handle input in client and GPIO in server.

Server routines:

I divide a big routine in Beejs guide (server.c) into smaller subroutines so that main can insert accordingly: init (open a socket and bind it), connect (accept a connection), main routine (receiving the command packet)

Client routines:

I divide a big routine in Beejs guide (client.c) into smaller subroutines so that main can insert accordingly: init (open a socket and connect it to server), main routine (sending command packet), thread routine (receiving the strength signal)

However, due to main routine is too simple with 1 function call, I just use the code directly in main (instead of function call).

Shared variables:

We need to share multiple information between gpio and metro\_server.c. Therefore, I bring many shared variables outside to be global variables. By this, we could share information between threads. These are defined in the userLEDmmap.h, hence available in both gpio and main file.

We need to share the socket information between main and interrupt handler of metro\_server.c. Therefore, I bring the new\_fd outside to be global variables in metro\_server.c,

We need to share the socket information between main and thread of client\_server.c. However, I could pass the sock\_fd by parameter to thread, hence we don’t need global variable for it.

GPIO refinement: Essentially the previous code, but we additionally return the printed character in gpio\_play so that we can make reply to client.

Signal Handler: The handler is based on the lab4, but we add the interpretation of the output from gpio (1/3/7) to (#/./!) and send them to receiver

Setting the HR timer is done similarly with test\_hrt\_timer.c. and key processing is the same with previous lab’s code. The timer is stop (disarmed) if we stop and armed if we start.

Disarm and arm is done by timer\_settime. Disarm is when we pass 0 interval as new timespec. Else, it will update and start the timer.

Thread is init before the loop and destroyed after completing. The main thread will not terminated before the thread exit.

Command packet: I send TimeSig index and Start/stop, to simplify the decipher process

1. Difficulty & Debug:

* Interrupted system call error: Since the recv and accept are blocking call, and might be called while in interrupt handler, it could generate this error. We add codes in the error handler to catch this error and handle (ignore) it instead of printing the error (EINTR).
* We want the metronome to restart if we change mode while running (Piazza TA’s answer). Therefore, instead of just update the timer when we press ‘m’, we will update it for ‘c’, ‘b’, ‘z’ as well.
* We also need to make sure that we start at the first beat of the new state after the state (time signature/tempo/on-off) is changed. I do that by using a (wrap-around) counter and reset it to 0 (first beat) when state is changed.
* Start/End packet needed special attention:

Instead of sending the initialized state as a command packet, we simply set the initial state in server ourselves.

When we press ‘Q’, we need to send the Quit message before quitting, so the server quit it child process as well. Also, since it will change the ‘keepgoing’ to 0 to stop the loop, this parameter is not shared to thread. Hence we need to force the termination for thread by thread\_cancel.

* Forking: forking will make some the global variable different from parent process, affecting the sharing. Specifically, the timer is not the same with the handler set in parent. Hence, I reset the timer in the child process.

1. Experiment:

Except GPIO, most of the code can be test with PC. Hence, we use the simulation to debug on PC.

i. Debug on PC

This is the early debug for our code, since we would like to make sure that everything in the input handling is correct before moving on the Beaglebone control

In gpio\_led\_fu\_sim.c, I just make gpio\_map and gpio\_unmap do nothing, and gpio\_play will only print the message (no gpio manipulation)

The program simply add the output of character corresponding to the LED in server side ( ‘7’ means 3 LEDs, ‘3’ means 2 LEDs, and ‘1’ means 1 LED. The corresponding signal will be printed on client side (#!.).

After this step, we confirm that the algorithm could pass the desired correct LEDs status to the Beaglebones and the Beaglebones now can realize the LED status.

We debug step-by-step:

A. metro\_client\_pc1 metro\_server\_sim1 TUI, Print command packet

B. metro\_client\_pc2 > metro\_server\_sim2 TxRx cmd, Action

C. metro\_client\_pc2 > metro\_server\_sim3 Gen reply & Print reply packet.

D. metro\_client\_pc4 >< metro\_server\_sim4 TxRx reply, Display reply

ii. Test on Beaglebones

Finally, we write the code to manipulate GPIO and realize the LED on/off status.

Before and after experiment, I reuse the shell file from last lab to turn off all LEDs and restore LEDs.

In gpio\_led\_fu\_sim.c, I just implement the gpio mapping, unmapping and gpio manipulation (as we did in lab 2-3) and make it gpio\_led\_fu.c

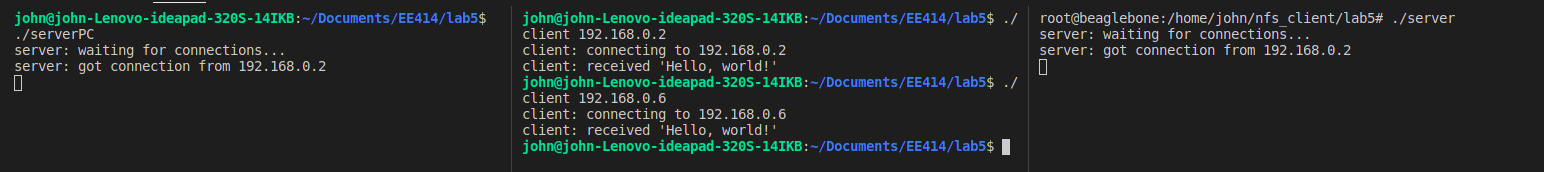
1. Experimental results:

Note: To produce all the executable files, simply use make command is enough.

* 1. Test stream socket: make stream

(1) Server on PC <-> Client on PC

(2) Server on Bone <-> Client on PC

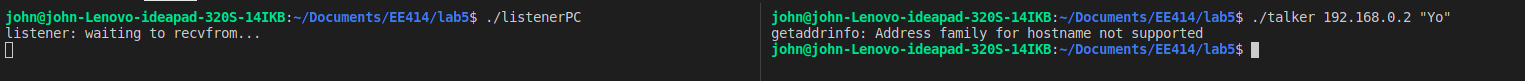


* 1. Test datagram socket: make datagram

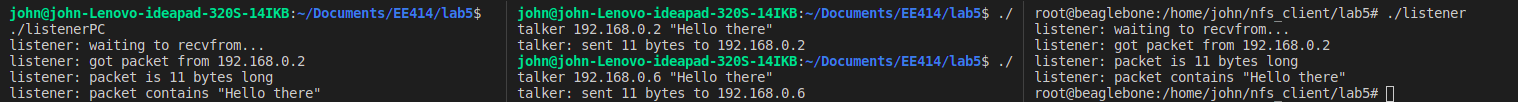
(1) Listener on PC <-> Talker on PC

(2) Listener on Bone <-> Talker on PC

If different family (IPv4 and IPv6)



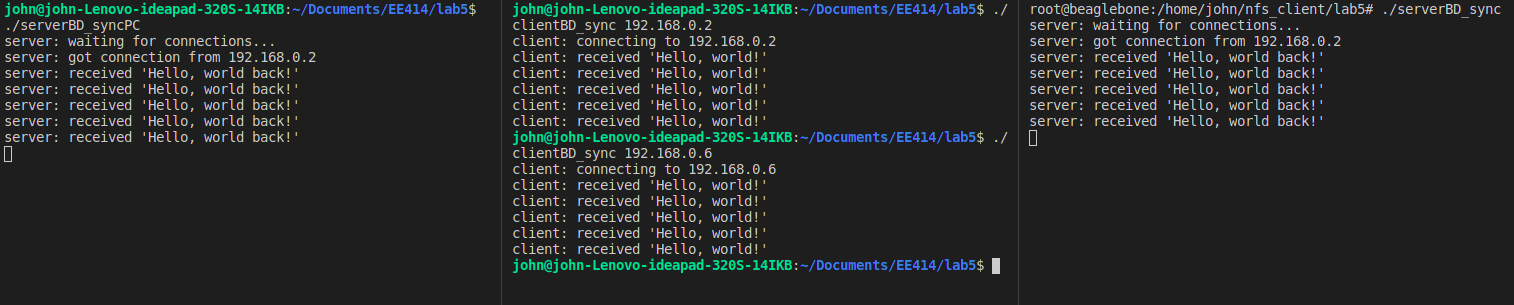
Same family (IPv4)



* 1. Test synchronous bidirectional stream socket: make stream

(1) Server on PC <-> Client on PC

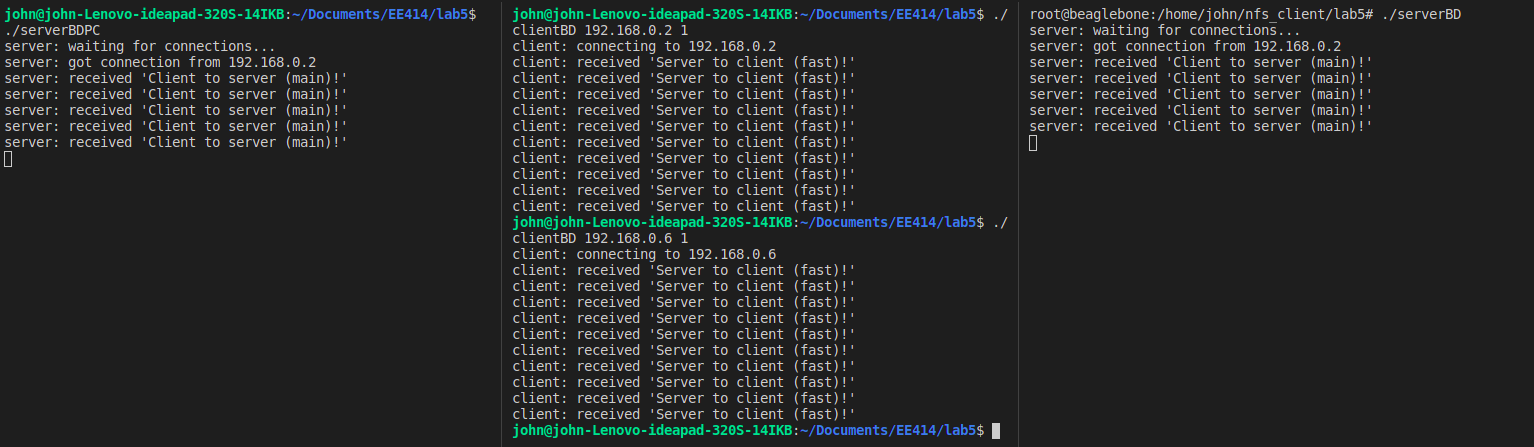
(2) Server on Bone <-> Client on PC



* 1. Test asynchronous bidirectional stream socket: make stream

(1) Server on PC <-> Client on PC

(2) Server on Bone <-> Client on PC

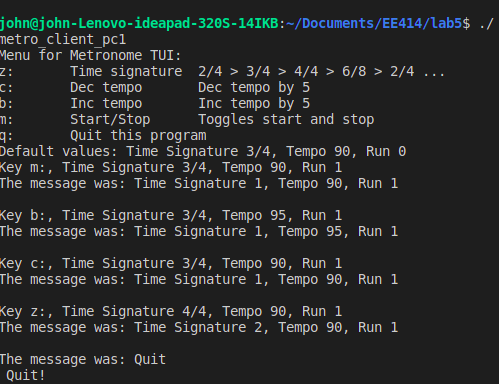
 For every 2 threads communication, there is 1 main communication

If I set frequency to 2, both communication appears at the same rate (hence, thread communication finish later, since it has 10 message)

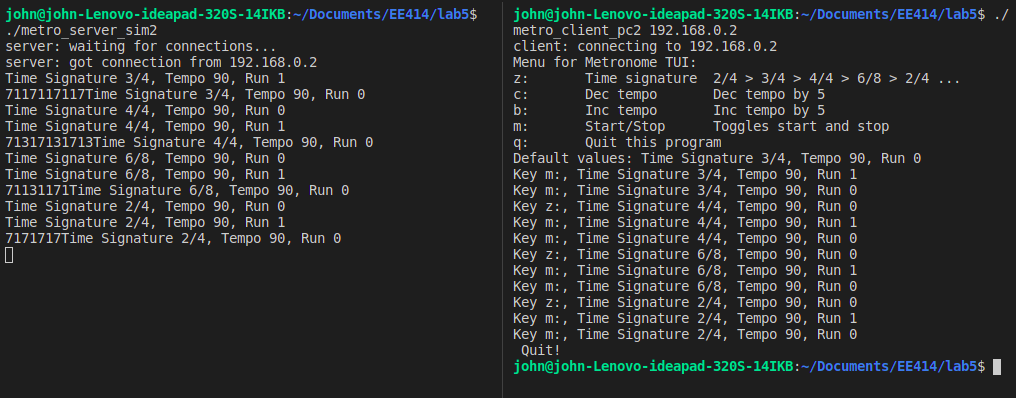
* 1. Test metronome:

(1) Server on PC <-> Client on PC: make sim

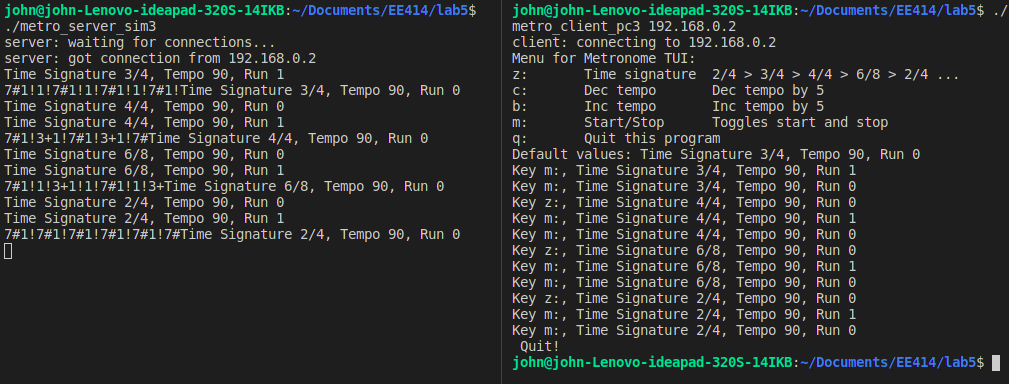
A. metro\_client\_pc1 *metro\_server\_sim1* (no needed) TUI, Print command packet



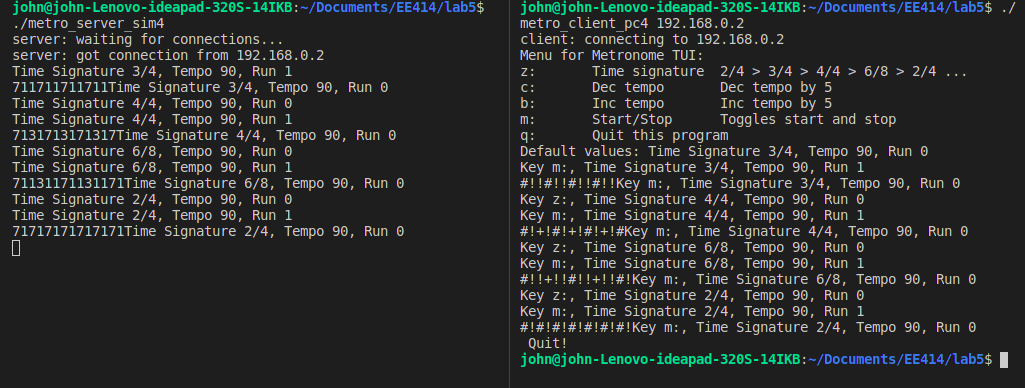
B. metro\_client\_pc2 > metro\_server\_sim2 TxRx cmd, Action



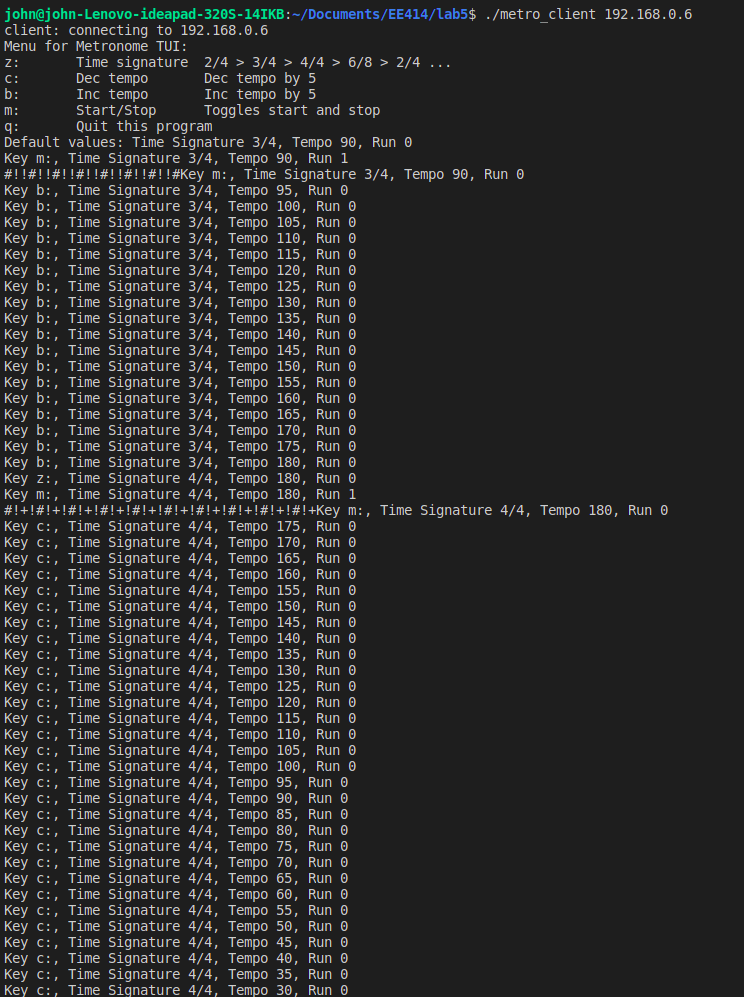
C. metro\_client\_pc2 > metro\_server\_sim3 Gen reply & Print reply packet.

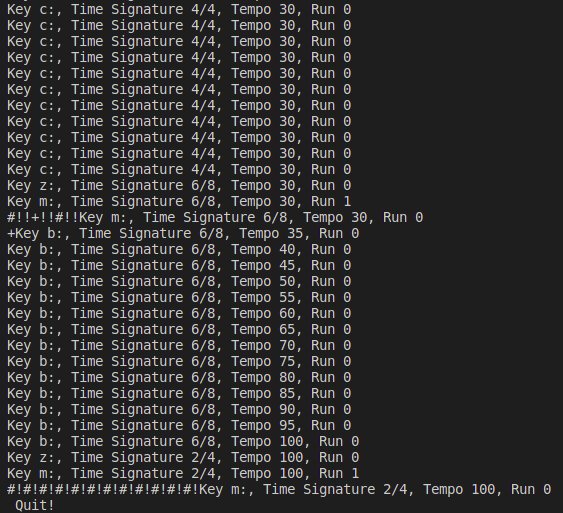


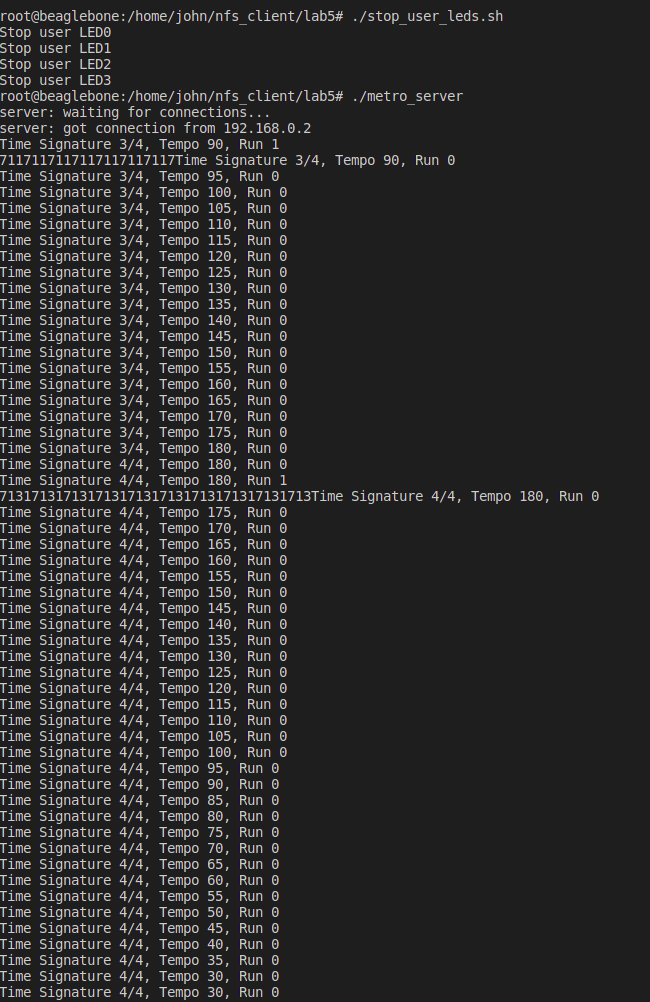
D. metro\_client\_pc4 >< metro\_server\_sim4 TxRx reply, Display reply

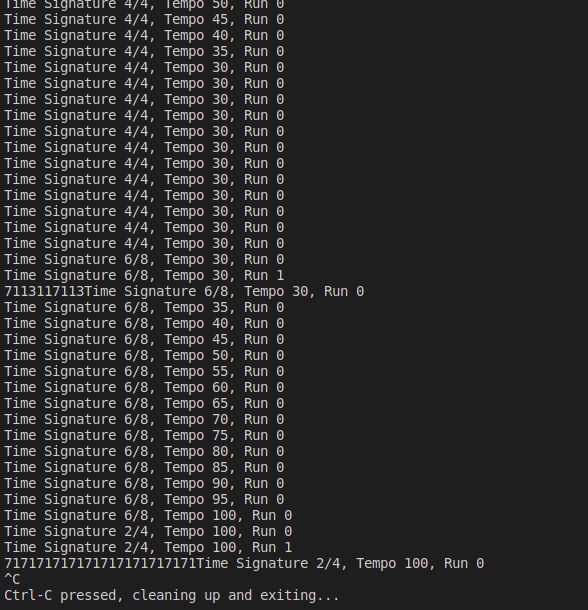


(2) Server on Bone <-> Client on PC: make met



s





1. Discussion:
2. Suppose multiple MetroClients wants to connect to the Metronome server. We have only one resource of Metronome service. How can you handle this situation?

The fork() function allow the server to serve many client’s connection at the same time. Hence, we can theoretically send and receive the packet to many clients.

However, the LEDs on the Metronome cannot served multiple clients (unless we find some way to multiplex between users, which make it not really a metronome anymore).

Hence, I suggest we can send and receive packets from multiple clients and serve them separately, but only the first users will control the actual LEDs. Other users still have the illusion of controlling the LEDs via the received !\*+ signals. We can also make a generative picture to illustrate the ‘virtual’ LEDs on user’s screen.

1. Question: Suppose you are going to replace the notebook computer to an Android smartphone. Is this possible? Discuss the required functionality of Remote MetroClient App

I think as long as Android could run the executable file, we can run it on the phone.

Network programming is environment-independent, since the communication uses the TCP/IP protocol. The socket already abstract and isolate device complexities.

The connection can be via WiFi. The keyboard will instead be key tap.

The phone just needs a compatible socket, with TCP options.

V- References:

[1] Beej’s Guide to Network Programming, Internet.

[2] Beaglebone Rev. A5. System Reference Manual,

http://circuitco.com/support/index.php?title=Beaglebone#Rev\_A5. NOTE.

[3] “Embedded Linux Primer”, C. Hallinan, Prentice Hall.

[4] Lab and lecture material, EE414 Teaching Staffs, KLMS

[5] EBC Exercise series, elinux.org

[6] GPIO Programming series, ics.com