

## CG1112 Engineering Principles and Practices II for CEG

### Week 8 Studio 2

#### Under the Hood: TCP/IP and TLS Programming

---

#### INTRODUCTION

---

In the Week 8 Studio 2 Activities we looked at how to generate CA keys and certificates, and how to use these to sign certificates for clients and servers. We also used a set of libraries that greatly simplified writing TLS clients and servers

In this document we will look into much greater detail into how to write TCP/IP and TLS programmes. Materials in this document are produced for your own edification and are not examinable.

---

#### ACTIVITY 1. BUILDING A TCP/IP CLIENT AND SERVER

---

In this activity we will be building a simple TCP/IP echo server (a server that echoes back whatever you type) and a client to test it.

TCP/IP programming centres around a data structure called a “Socket” shown below:



Well, not really. But TCP/IP programming does use something called a “socket”, which is a special data structure (more accurately, a file descriptor) that lets you open a connection to a host, and read/write data from/to the host.

#### Step 1.

In this step we will set up a TCP/IP server in C. To do this we need to do the following (complete code given at the end, but read through this first):

- a. Create two integer variables called `listenfd` and `connfd`. These two variables are called “file descriptors” and are identifiers that allow us to read and write “sockets”. A “socket” is a structure that lets us read and write data on TCP/IP networks.

```
int listenfd, connfd;
```

- b. We want to specify which port to listen to, and to do this we create a variable of type `struct sockaddr_in`. We will call this variable “`serv_addr`”.

```
struct sockaddr_in serv_addr;
```

```
// Sets every byte in serv_addr to 0.  
memset(&serv_addr, 0, sizeof(struct sockaddr_in));
```

- c. Now we will create a “listener socket”, which is an integer identifier that lets us tell the Operating System which connection we want to talk to. Remember that there may be multiple connections (e.g. one for Chrome, one for SSH, one for VNC, and another for this server we are writing), so being able to identify actually which connection we are interested in is very important.

```
listenfd = socket(AF_INET, SOCK_STREAM, 0);
```

`AF_INET` means that we will use IPv4. Other options include `AF_INET6` which means we will use IPv6 instead (with 128 bit addresses instead of 32 bit), `AF_IPX` to use the Novell IPX protocol (largely obsolete), `AF_X25` to use the X25 packet switching protocol (even more obsolete), etc.

`SOCK_STREAM` means that we will use the reliable TCP connection-based transport layer protocol. Other options include `SOCK_DGRAM` that lets us use connectionless best-effort UDP (User Datagram Protocol) protocol etc. `SOCK_DGRAM` is best-effort, but is very useful for broadcasts and for streaming video and music, because video and music can tolerate packet losses without much degradation.

The 3<sup>rd</sup> argument is an index that selects which protocol to use. Since `AF_INET` consists only of one protocol (TCP/IP), we will use an index of 0.

- d. We now need to bind our socket to a port number. Remember that port numbers identify particular services on our host. You can use any port number of 1024 to 65535. If you use a port number of 1023 and below you must run your server as a superuser.

To bind our server to its port, we first configure the `serv_addr` variable we created earlier. See the comments below for detailed explanation.

```
// Use the IPv4 family of addresses.
serv_addr.sin_family = AF_INET;

// If our host has multiple LAN / WiFi connetions,
// it will have multiple IP addresses.
// INADDR_ANY binds our port to all our IP addresses.
// htonl converts INADDR_ANY from the host's long format
// (32-bit little endian) to the
// standardized network long format (32-bit big endian)
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);

// Bind to port 5000. htons works like htonl but for short 16-bit numbers
serv_addr.sin_port = htons(5000);
```

- e. We are now ready to bind the socket to port 5000. This means that connections to port 5000 will automatically be associated with the socket in `listenfd` (and thus to our server).

```
int c = sizeof(sockaddr_in);
bind(listenfd, (struct sockaddr *) &serv_addr,
      (socklen_t *) &c);
```

- f. Now we listen out for connections. We will allow a FIFO queue of up to 10 “unhandled connections”. A “unhandled” connection is one that has not been “accepted”. See part g.

```
listen(listenfd, 10);
```

- g. We use the `accept(.)` call to remove the next unhandled connection from the listen queue. The `accept` call creates a new socket and returns the file handler to that socket.

```
struct sockaddr_in client;
connfd = accept(listenfd, &client, sizeof(client));
```

The first argument `listenfd` tells us which listener socket to accept the connection from. The second argument “client” points to a variable of type `struct sockaddr_in`. This client variable will contain address information about clients connecting to the server.

- h. Now we can use `read` and `write` to read from the client or write to the client, using the `connfd` file descriptor returned by `accept`.

```
char *message="Hello world";
char buffer[128];
int len;

len = write(connfd, message, strlen(message));

printf("Wrote %d bytes\n", len);

len = read(connfd, buffer, 128);

if(len == 0)
    printf("Client closed connection\n");
```

As shown above, both read and write return the actual number of bytes written or read. The write function returns 0 if there's a writing error. The read function returns 0 when the client closes the connection (e.g. hit CTRL-C to exit).

- i. When the client has closed the connection, close the connfd socket:

```
close(connfd);
```

Now here is the full server code, enclosed as server.cpp. Please go through this code to understand it:

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>

int main()
{
    // Declare two integer variables that will
    // point to sockets.
    int listenfd, connfd;

    // serv_addr will be used to configure the port number
    // of our server.
    struct sockaddr_in serv_addr;

    // Set every element in serv_addr to 0.
    memset(&serv_addr, 0, sizeof(serv_addr));

    // Open up a TCP/IP (AF_INET) socket, using the reliable TCP
    // protocol (SOCK_STREAM). You can also create a best effort UDP socket
    // by specifying SOCK_DGRAM instead of SOCK_STREAM. The "0" means
    // use the first protocol in the AF_INET family. The AF_INET family
    // has only one protocol so this is always 0.
    listenfd = socket(AF_INET, SOCK_STREAM, 0);

    // We use perror to print out error messages
    if(listenfd < 0)
    {
        perror("Unable to create socket: ");
        exit(-1);
    }

    // Configure our server to bind to all interfaces (INADDR_ANY)
    // including all network cards and WiFi ports. We also
    // set our port number to 5000.
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    // Now actually bind our socket to port 5000
    if(bind(listenfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
    {
```

```

    perror("Unable to bind: ");
    exit(-1);
}

// And start listening for connections. We maintain a FIFO
// queue of 10 entries of "unaccepted" connections. I.e.
// connections pending acceptance using the accept() function below.

printf("Now listening..\n");
if(listen(listenfd, 10) < 0)
{
    perror("Unable to listen to port: ");
    exit(-1);
}

int c = sizeof(struct sockaddr_in);
while(1)
{
    // Accept a new connection from the queue in listen. We will
    // build an echo server
    struct sockaddr_in client;
    connfd = accept(listenfd, (struct sockaddr *) &client, (socklen_t *) &c);

    char clientAddress[32];

    // Use inet_ntop to extract client's IP address.
    inet_ntop(AF_INET, &client.sin_addr, clientAddress, 32);

    printf("Received connection from %s\n", clientAddress);

    char buffer[1024];

    int len=0;
    do
    {
        // Read in data
        len = read(connfd, buffer, sizeof(buffer));

        // Echo it back. Note if len=0 means other side closed the connection.
        if(len > 0)
            write(connfd, buffer, strlen(buffer)+1);

    } while(len > 0);

    printf("Client closed connection.\n");
    close(connfd);
}

```

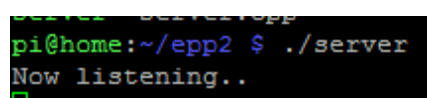
j. Compile the code and run the code by typing:

```

gcc server.cpp -o server
./server

```

If all goes well you should see:



```

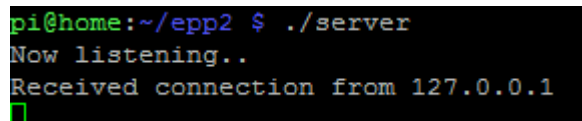
pi@home:~/epp2 $ ./server
Now listening..

```

- k. We will now test our server using netcat (see Week 8 Studio 2 Optional Activity). Open up a new shell and connect to your server by typing:

```
nc localhost 5000
```

Your server will now show:

A terminal window with a black background and green text. The text shows a user at a pi@home prompt in the directory ~/epp2, running the command ./server. The output of the program is "Now listening.." followed by "Received connection from 127.0.0.1" and a green cursor on the next line.

```
pi@home:~/epp2 $ ./server
Now listening..
Received connection from 127.0.0.1
█
```

- l. At the terminal where you typed “nc localhost 5000” start typing. You will see whatever you typed being echoed back to you when you hit enter. This is known as an “echo server” and is often used for debugging networks.
- m. If you are running server on your friend’s computer, ensure that you are both on the same network, find out the IP address and connect using:

```
nc <friend’s IP address> 5000
```

## Step 2.

Now that you have built a server and tested it using netcat (nc), it’s time for us to build our own client! to do this:

- a. We use a #define to create a constant holding our server’s IP address, or get it some other way. The enclosed client.cpp gets a hostname from the command line then translates it to an IP address using gethostbyname.

```
#define IP_ADDRESS      "127.0.0.1"
```

- b. As before we need a “file descriptor” to to read and send data to the network. Create an integer variable sockfd, which will be our socket.

```
int sockfd;
```

- c. We need to specify the IP address and port number of the server we are accessing, and for this we must create a variable called serv\_addr of type struct sockaddr\_in. We zero this variable to ensure that all fields are 0.

```
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));
```

- d. We fill serv\_addr with the details of our server, telling the Operating System that we are using TCP/IP (AF\_INET). See the comments for more details.

```
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(5000);

// inet_pton is a function that converts from our
// usual dot addresses into a 32-bit IP address
// in network format (i.e. big endian).
inet_pton(AF_INET, IP_ADDRESS, &serv_addr.sin_addr);
```

- e. As before we create a socket using the socket call.

```
sockfd = socket(AF_INET, SOCK_STREAM, 0);
```

- f. Then we connect to our server, using connect, giving it the socket variable sockfd, and the address in serv\_addr. Note that connect requires serv\_addr (currently of type struct sockaddr\_in) to be cast to sockaddr.

```
connect(sockfd,
        (struct sockaddr *) &serv_addr, sizeof(serv_addr));
```

- g. And use sockfd to read/write the server

```
char *message = "Hello world!";
write(sockfd, message, strlen(message)+1);
char buffer[1024];
int len;

len = read(sockfd, buffer, sizeof(buffer));

printf("Server sent %d bytes: %s\n", len, buffer);
```

- h. Finally close the socket if you don't need it anymore.

```
close(sockfd);
```

- i. The full client code is shown below, enclosed as client.cpp. Note that in this code we use gethostbyname to get the IP address of a host using DNS. This client also contains an example of how to catch the CTRL-C keypress, formally called a SIGINT signal. 😊 Go through this code to understand it fully.

```
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <arpa/inet.h>

// We create our file descriptor for the socket as a global
```

```

// so that our CTRL-C handler can close the socket.
int sockfd;

void intHandler(int dummy)
{
    printf("\n\nWHOA MAMA!!!! Why did you press CTRL-C? XD\n\n");
    printf("Closing the socket.\n");
    close(sockfd);
    exit(0);
}

int main(int ac, char **av)
{
    if(ac != 3)
    {
        fprintf(stderr, "\nUSAGE: %s <host name> <port number>\n\n", av[0]);
        exit(-1);
    }

    // Create a structure to store the server address
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if(sockfd < 0)
    {
        perror("Cannot create socket: ");
        exit(-1);
    }

    // Get host IP address
    char hostIP[32];
    struct hostent *he;

    he = gethostbyname(av[1]);

    if(he == NULL)
    {
        perror("Unable to get host IP address: ");
        exit(-1);
    }

    struct in_addr **addr_list = (struct in_addr **) he->h_addr_list;

    strncpy(hostIP, inet_ntoa(*addr_list[0]), sizeof(hostIP));
    printf("Host %s IP address is %s\n", av[1], hostIP);

    // Now form the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(av[2]));
    inet_pton(AF_INET, hostIP, &serv_addr.sin_addr);

    // Now let's connect!
    if(connect(sockfd, (struct sockaddr *)&serv_addr,
        sizeof(serv_addr)) < 0)
    {
        perror("Error connecting: ");
        exit(-1);
    }
}

```



```

    printf("Now we write to the server and get its response. To exit press
CTRL-C\n");

    // Install handler to catch CTRL-C (SIGINT)
    signal(SIGINT, intHandler);

    int len;

    // Now we read from the keyboard, send to the server,
    // and print the server's response.
    do
    {
        char buffer[1024], c;
        printf("Say something: ");
        fgets(buffer, sizeof(buffer), stdin);
        len = write(sockfd, buffer, strlen(buffer)+1);

        if(len > 0)
        {
            printf("Wrote %d bytes to the server\n", len);

            // Now read from the server
            len = read(sockfd, buffer, sizeof(buffer));

            if(len > 0)
                printf("Received %d bytes from server: %s\n", len, buffer);
        }

    } while(len > 0);

    printf("Server closed connection\n");
    close(sockfd);
}

```

- j. Compile your client using:

```
gcc client.cpp -o client.
```

- k. To run the client you will need to give it two arguments: A host name and a port number. Open up two shells.

In one shell we will run our server earlier on:

```
./server
```

In the other shell, type:

```
./client localhost 5000
```

Now play around. When you are done hit CTRL-C to exit the client. 😊

If you are running the server on your friend's computer, ensure that you're all on the same network, then find out his or her IP address, and connect using:

```
./client <friend's IP address> 5000
```

You can also annoy our SoC webserver by typing:

```
./client www.comp.nus.edu.sg 80
```

This will open a connection to the SoC website. Type "GET" and hit enter, which will return an error page.

You can try playing around with GET requests and see if you can get SoC's website to give you its homepage. This is left as an exercise for you. 😊

- I. Now launch 3 or more shells, run the server in one shell, and either using nc or client, connect to the server simultaneously.

Try typing on each of the clients. You will notice that the server can only process one client at the time.

That's horrible. But we will fix that.

---

## ACTIVITY 2. USING MULTITHREADING

---

It should be obvious from the end of Activity 1 that our server can only take one connection at a time. This is not very useful. In this activity we will look at multithreading, and how to increase the capacity of both our client and server. Multithreading allows our programs to do more than one thing at a time. For example, our server can accept and handle multiple connections, while our client can read the network data and our keyboard at the same time.

In a word, multithreading is magic.

### Step 1.

POSIX threads (henceforth called "pthreads") is an API specification for threads on Unix based systems. Pthreads packages are available for non-Unix systems as well. The basic thread creation, joining and destruction calls are:

Call	Description
<code>int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void * (*start_routine)(void *), void *arg)</code>	Creates a new thread. Returns 0 if successful. Arguments:  thread - A data structure which will contain information about the created thread. attr - Thread attributes. Can be NULL. start_routine - Pointer to the thread's starting function. Must be declared as void *fun(void *) arg - Argument passed to the starting routine.
<code>void pthread_exit(void *value_ptr)</code>	Exits from a thread. The value in value_ptr is passed to another thread that joins with this exiting thread.

<pre>void pthread_join(pthread_t thread, void **value_ptr)</pre>	<p>Suspends execution until the thread specified by "thread" completes execution. If value_ptr is not NULL, it will point to a location containing the value passed by "thread" when it exits using pthread_exit.</p> <p>pthread_join returns 0 if successful.</p>
--	--

You need to `#include <pthread.h>` to use these. We will now look at an example of how to use threads.

On your Pi, Ubuntu machine, Windows 10 Linux subsystem, etc, create a file called `mythread.cpp` and type in the following code:

```
#include <stdio.h>
#include <pthread.h>

// Global variable.
int ctr=0;
pthread_t thread[10];

void *child(void *t)
{
    // Print out the parameter passed in, and the current value of ctr.
    printf("I am child %d. Ctr=%d\n", t, ctr);
    // Then increment ctr
    ctr++;
    pthread_exit(NULL);
}

int main()
{
    int i;

    // Initialize ctr
    ctr=0;

    // Create the threads
    for(i=0; i<10; i++)
        pthread_create(&thread[i], NULL, child, (void *) i);

    // And print out ctr
    printf("Value of ctr=%d\n", ctr);
    return 0;
}
```

Now compile this code using:

```
gcc mythread.cpp -pthread -o mythread
```

Note the use of `"-pthread"`: This tells the compiler that you are using POSIX threads and that it should bring in the pthread library.

There is one annoying thing though; main has a tendency of exiting before all the 10 child threads are done. To prevent this, we will use `pthread_join`.

Amend main as shown (new lines are in **bold.**):

```
int main()
{
    ...

    for(i=0; i<10; i++)
        pthread_join(thread[i], NULL);
    // And print out ctr
    printf("Value of ctr=%d\n", ctr);
    return 0;
}
```

Now recompile and enjoy the difference. 😊

### Step 2.

Now we will change our server.cpp code to become multithreading. Open server.cpp:

- a. Right at the very top of server.cpp, type in:

```
#include <pthread.h>
```

- b. Just above main() type in:

```
void *workerThread(void *p)
{
    int connfd = (int) p;
}
```

- c. Now locate this following code inside main. Cut it completely from main and paste it inside workerThread after the `int connfd = (int) p;` statement.

```
char buffer[1024];

int len=0;
do
{
    // Read in data
    len = read(connfd, buffer, sizeof(buffer));

    // Echo it back. Note if len=0 means
    //other side closed the connection.
    if(len > 0)
        write(connfd, buffer, strlen(buffer)+1);

} while(len > 0);

printf("Client closed connection\n");
close(connfd);
```

So now your workerThread will look like this:

```
void *workerThread(void *p)
{
    int connfd = (int) p;

    char buffer[1024];

    int len=0;
    do
    {
        // Read in data
        len = read(connfd, buffer, sizeof(buffer));

        // Echo it back. Note if len=0 means other side closed the connection.
        if(len > 0)
            write(connfd, buffer, strlen(buffer)+1);

    } while(len > 0);

    printf("Client closed connection.\n");
    close(connfd);
}
```

What you have done now is to create something known as a “worker thread” (hence the name of the function) which will receive new connections and process those new connections.

- d. Now go back to main: under the “`printf(“Received connection from %s\n”, clientAddress);`” line, type in the following code to launch a new worker thread each time we get a new connection:

```
pthread_t worker;
pthread_create(&worker, NULL, workerThread, (void *) connfd);
pthread_detach(worker);
```

This statement launches a new workerThread for EACH new connection, passing to it the socket. The `pthread_detach(.)` call tells the operating system that resources used by the workerThread should be freed as soon as the thread exits, preventing “zombie threads”.

The `while(1)` part of your main will now look like this:

```
while(1)
{
    // Accept a new connection from the queue in listen. We will
    // build an echo server
    struct sockaddr_in client;
    connfd = accept(listenfd, (struct sockaddr *) &client,
                    (socklen_t *) &c);

    char clientAddress[32];

    // Use inet_ntop to extract client's IP address.
    inet_ntop(AF_INET, &client.sin_addr, clientAddress, 32);

    printf("Received connection from %s\n", clientAddress);

    pthread_t worker;
    pthread_create(&worker, NULL, workerThread, (void *) connfd);
    pthread_detach(worker);
}
```

Note that here we use a `pthread_detach` (which tells LINUX to free any resources used by the thread once it is done) rather than `pthread_join`. If we use `pthread_join` it will block until the workerThread completes, which will prevent the server from taking new connections.

Now compile your server and run it:

```
gcc server.cpp -pthread -o server
./server
```

By opening multiple shells you can now launch multiple instances of `nc` or `client`, and see that each one gets a connection. 😊

**NOTE: If you mess this up, the code is provided in `servermt.cpp`. But it's better if you did it yourself. To compile: `gcc servermt -pthread -o servermt`**

### Step 3.

Now that we have a multithreaded server, let's do something even more fun. Let's build a multithreaded client!

Why you ask? Because we can. 😊

Seriously though, if you looked at the `client.cpp` code you will notice one deficiency:

- If client is waiting for keyboard input, it cannot receive data from the network because it is stuck at the fgets until you hit enter.
- If client is waiting for network input, it is stuck at read and cannot read the keyboard.

The trick then is to put the keyboard read and network read in separate threads and run them together!

Rather than making you type the somewhat large number of changes to be made, the multithreaded version of client.cpp is instead included as clientmt.cpp. The code is shown here in full:

```
#include <pthread.h>
#include <sys/socket.h>
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <signal.h>
#include <arpa/inet.h>

volatile int exitFlag=0;

void intHandler(int dummy)
{
    printf("\n\nWHOA MAMA!!!! Why did you press CTRL-C? XD\n\n");
    exitFlag=1;
}

void *kbThread(void *p)
{
    int sockfd = (int) p;
    while(!exitFlag)
    {
        int len;
        char buffer[1024];
        printf("Say something: ");
        fgets(buffer, sizeof(buffer), stdin);

        len = write(sockfd, buffer, strlen(buffer)+1);
        printf("Wrote %d bytes to the server\n", len);
    }

    printf("Keyboard thread exiting\n");
    pthread_exit(NULL);
}

void *readThread(void *p)
{
    int sockfd = (int) p;
    while(!exitFlag)
    {
        int len;
        char buffer[1024];
        len = read(sockfd, buffer, sizeof(buffer));

        if(len > 0)
            printf("Received %d bytes from server: %s\n", len, buffer);
    }
}
```

```

        // Exit if we read zero bytes. This means the server closed the
        // connection
        exitFlag=(len == 0);
    }

    printf("Read thread exiting\n");
    pthread_exit(NULL);
}

int main(int ac, char **av)
{
    if(ac != 3)
    {
        fprintf(stderr, "\nUSAGE: %s <host name> <port number>\n\n", av[0]);
        exit(-1);
    }

    int sockfd;
    // Create a structure to store the server address
    struct sockaddr_in serv_addr;
    memset(&serv_addr, 0, sizeof(serv_addr));

    // Create a socket
    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    if(sockfd < 0)
    {
        perror("Cannot create socket: ");
        exit(-1);
    }

    // Get host IP address
    char hostIP[32];
    struct hostent *he;

    he = gethostbyname(av[1]);

    if(he == NULL)
    {
        perror("Unable to get host IP address: ");
        exit(-1);
    }

    struct in_addr **addr_list = (struct in_addr **) he->h_addr_list;

    strncpy(hostIP, inet_ntoa(*addr_list[0]), sizeof(hostIP));
    printf("Host %s IP address is %s\n", av[1], hostIP);

    // Now form the server address
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(atoi(av[2]));
    inet_pton(AF_INET, hostIP, &serv_addr.sin_addr);

    // Now let's connect!
    if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
    {
        perror("Error connecting: ");
        exit(-1);
    }

    printf("Now we write to the server and get its response. To exit press CTRL-C\n");

```



```

// Install handler to catch CTRL-C (SIGINT)
signal(SIGINT, intHandler);

int len;

pthread_t _kb, _rd;

pthread_create(&_kb, NULL, kbThread, (void *) sockfd);
pthread_detach(_kb);
pthread_create(&_rd, NULL, readThread, (void *) sockfd);
pthread_detach(_rd);

// Keep looping until we exit
while(!exitFlag);

// Now join the threads
printf("Closing socket and exiting.\n");
close(sockfd);
}

```

Some things to note:

- The keyboard read was moved to its own thread in kbThread, and the network read was moved to readThread.
- All the threads loop until a shared exitFlag is set to 1. This allows all the threads to exit at the same time.
- Main creates the kbThread and readThread threads in these calls:

```

pthread_t _kb, _rd;

pthread_create(&_kb, NULL, kbThread, (void *) sockfd);
pthread_detach(_kb);
pthread_create(&_rd, NULL, readThread, (void *) sockfd);
pthread_detach(_rd);

```

Then goes into a loop until the threads exit, afterwhich it closes the socket.

```

// Keep looping until we exit
while(!exitFlag);

// Now join the threads
printf("Closing socket and exiting.\n");
close(sockfd);

```

- The SIGINT handler was similarly modified to set exitFlag to 1, allowing all the threads to exit.
- Since the SIGINT handler no longer closes the socket, sockfd has been moved into main. The sockfd variable is passed to the threads in the pthread\_create calls.
- The kbThread thread doesn't actually exit gracefully because it is stuck at the fgets. However the OS will kill this thread once the clientmt process exits.

To compile and run (assuming that server is running locally):

```

gcc clientmt.cpp -pthread -o clientmt
./clientmt localhost 5000

```

---

### ACTIVITY 3. TLS PROGRAMMING

---

In this week's Studio 1 you are given a library with `make_tls_server.cpp` and `make_tls_client.cpp`, which greatly simplify the creation of TLS clients and servers. Here we will create the TLS clients and servers using a slightly harder, but more powerful method.

For this activity to work, you need to copy over the `alex.key`, `alex.crt` and `signing.pem` files for the TLS example server, and `laptop.key`, `laptop.crt` and `signing.pem` for the TLS client.

To help you along, three libraries have been provided to you:

- i. `tls_server_lib.h` / `tls_server_lib.cpp`: TLS routines for servers.
- ii. `tls_client_lib.h` / `tls_client_lib.cpp`: TLS routines for clients.
- iii. `tls_pthread.h`/`tls_pthread.cpp`: TLS callbacks for use in multithreaded environments; sadly OpenSSL is not “thread-safe” – i.e. OpenSSL corrupts and crashes when used in a multithreaded environment. Fortunately they provide us some mechanisms to prevent this, and `tls_pthread` implements these mechanisms.

To help you understand the libraries, we will look at how they work.

#### SERVER SIDE TLS

To start a server-side TLS, you need to perform the following steps. See the `tls_server_lib.cpp` file for more details.

- i. Initialize the SSL library. You can see how this is done in the `init_openssl` function in `tls_server_lib.cpp`:
  - a. Call `ssl_library_init` to initialize the library.
  - b. Call `SSL_load_error_strings` to load all error messages.
  - c. Call `OpenSSL_add_all_algorithms` to load all cipher algorithms.
- ii. Create an SSL context. You can see how this is done in “`create_context`” in `tls_server_lib.cpp`:
  - a. Create a variable called `method` of type `SSL_METHOD`.
  - b. Create a context variable called `ctx` of type `SSL_CTX`.
  - c. Choose an SSL method and assign it to “`method`”. We will use `SSLv23_server_method` which gives you the option of using `SSLv2`, `SSLv3`, `TLS_1.0`, `TLS_1.1` or `TLS_1.2` handshake methods. These handshake methods allow the client and server to trade encryption keys.
  - d. Since `SSLv2` and `SSLv3` have security flaws and are unsafe, we will force TLS to use `TLS_1.0`, `1.1` or `1.2`. We can do this by calling `SSL_CTX_set_options` to disable `SSLv2` and `SSLv3`.
- iii. Load the server's certificate and private key. See “`configure_context`” in `tls_server_lib.cpp` for how to do this:

- a. Call `SSL_CTX_set_ecdh_auto` to tell TLS to automatically select the right elliptic curve for use in the Elliptic Curve Diffie-Hellman key-exchange protocol. This protocol allows the client and server to exchange private keys for the session encryption without actually sending the key, through the use of partial shared information.
  - b. Call `SSL_CTX_use_certificate_file` to load the server's certificate. This must be in PEM format (if you can read the certificate as a series of hexadecimal numbers, it's already in PEM format. If it's a bunch of unreadable binary characters, it's in the wrong format)
  - c. Use `SSL_CTX_use_PrivateKey_file` to load the server's private key. The private key must be in PEM format.
- iv. Create a server socket just like in Activities 1 and 2.
- v. Accept a new connection with a new client socket `connfd`.
- vi. Create a new SSL session and associate it with `connfd`. See `connectSSL` in `tls_server_lib.cpp` for details:
  - a. Call `SSL_new` with the context created in step ii. above.
  - b. Associate the socket with the SSL session using `SSL_set_fd`.
  - c. Accept a new SSL connection using `SSL_accept`. At this step TLS performs the handshake that was described earlier, negotiating which symmetric cipher to use, performing key exchange using Diffie-Hellman or other algorithms, and where the server presents its certificate. If host-verification is enabled, TLS will check that the certificates Common Name (e.g. [www.facebook.com](http://www.facebook.com), or the server's IP address) matches the Common Name you specify.
- vii. Your SSL session is now ready to use. Use `SSL_read` and `SSL_write` in place of `read` and `write` to securely transmit data.
- viii. When done:
  - a. Call `SSL_free` to free the ssl session.
  - b. Close the client connection.
- ix. When exiting the program:
  - a. Close the listener socket.
  - b. Call `SSL_CTX_free` to free the context.
  - c. Call `cleanup_openssl` to clean up the SSL session. This calls `EVP_cleanup` to free all the resources used by OpenSSL.

## CLIENT SIDE TLS

Client side TLS is more complicated because of the need to verify the server's certificate and to check that the server's IP address or "common name" (e.g. [www.facebook.com](http://www.facebook.com)) is correct. See the `tls_client_lib.cpp` file for more details.

- i. As before we begin by initializing the OpenSSL library. See `init_openssl` in `tls_client_lib.cpp` for more details:
  - a. Call `SSL_library_init` as before to initialize the OpenSSL library.
  - b. Call `ERR_load_crypto_strings` to load all cryptographic error messages.
  - c. Call `SSL_load_error_strings` to load all TLS error messages.
  - d. Call `OpenSSL_add_all_algorithms` to load all encryption algorithms.
- ii. Create a context. See the `create_context` function in `tls_client_lib.cpp` for details:
  - a. Create two variables of type `SSL_METHOD` and `SSL_CTX` as before.
  - b. Choose `SSLv23_client_method` and assign it to your `SSL_METHOD` variable. This tells the client to initiate either an `SSLv2`, `SSLv3`, `TLS_1.0`, `TLS_1.1` or `TLS_1.2` session with the server. Since we disabled `SSLv2` and `SSLv3` on the server, the client will never use these. In any case we will disable `SSLv2` and `SSLv3` in Step g below.
  - c. Call `SSL_CTX_new` with the chosen method to create the context and assign it to the `SSL_CTX` variable.
  - d. Call `SSL_CTX_set_verify` to tell TLS to verify the server's certificate.
  - e. Call `SSL_CTX_set_verify_depth` to set the verification depth. In Week 8 Studio 2 Activity 1 we signed Alex's certificate using the CA certificate and key. In actual fact Alex can also sign certificates, and those can in turn be used to sign other certificates. This function tells us how far back to verify certificates. TLS will stop once either when it has reached the depth specified or when it finds a CA certificate, whichever is sooner.
  - f. Call `SSL_CTX_load_verify_locations` to load the CA certificate (as stated, you MUST copy this certificate onto all clients).
  - g. Call `SSL_CTX_set_options` to disable `SSLv2` and `SSLv3`.
- iii. Create a TCP/IP session socket and connect to the server as shown in Activity 1.
- iv. Create a new SSL session and associate it with the socket in Step iii. See `connectSSL` in `tls_client_lib.cpp` for more details.
  - a. Call `SSL_new` with the context created in Step ii. above.
  - b. Call `SSL_get0_param`, `X509_VERIFY_PARAM_set_hostflags` and `X509_VERIFY_PARAM_set1_host` to enable host checking, if needed (See `setHostVerification` in `tls_client.cpp` for details).
  - c. Call `SSL_set_fd` to associate the socket with the SSL session.
  - d. Call `SSL_connect` to initiate the SSL handshaking with the server. At this step host name verification is done if set in part b.
- v. Do certificate verification (see `verifyCertificate` for details):
  - a. Call `SSL_get_verify_result` to check the certificate. It should return `X509_V_OK` if the certificate is valid.

- vi. Now use `SSL_read` and `SSL_write` in place of `read` and `write` to securely communicate with the server.
- vii. When DONE:
  - a. Call `SSL_free` to free the SSL session.
- viii. When exiting the program:
  - a. Call `SSL_CTX_free` to free the context.
  - b. Call `cleanup_openssl`, which calls `EVP_cleanup` to free all data associated with OpenSSL.

## HANDLING MULTITHREADING

As mentioned before OpenSSL is not “thread-safe”, meaning that running it in a multithreading environment is likely to result in crashes. To prevent the crashes, there is a group of functions you need to create. See `tls_pthread.cpp` for details.

## MAKING A TLS SERVER

Whew, that was a lot to handle. Now let’s look at how to actually make an SSL server. See `tls_servermt.cpp` for the full code:

### Step 1.

Follow the steps in Activity 1 to create a TCP/IP server:

```
// Declare two integer variables that will
// point to sockets.
int listenfd, connfd;

// serv_addr will be used to configure the port number
// of our server.
struct sockaddr_in serv_addr;

// Set every element in serv_addr to 0.
memset(&serv_addr, 0, sizeof(serv_addr));

// Open up a TCP/IP (AF_INET) socket, using the reliable TCP
// protocol (SOCK_STREAM). You can also create a best effort UDP socket
// by specifying SOCK_DGRAM instead of SOCK_STREAM. The "0" means
// use the first protocol in the AF_INET family. The AF_INET family
// has only one protocol so this is always 0.
listenfd = socket(AF_INET, SOCK_STREAM, 0);

// We use perror to print out error messages
if(listenfd < 0)
{
    perror("Unable to create socket: ");
    exit(-1);
}

// Configure our server to bind to all interfaces (INADDR_ANY)
// including all network cards and WiFi ports. We also
// set our port number to 5000.
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(5000);

// Now actually bind our socket to port 5000
if(bind(listenfd, (struct sockaddr *) &serv_addr, sizeof(serv_addr)) < 0)
{
    perror("Unable to bind: ");
    exit(-1);
}

// And start listening for connections. We maintain a FIFO
// queue of 10 entries of "unaccepted" connections. I.e.,
// connections pending acceptance using the accept() function below.

printf("Now listening...\n");
if(listen(listenfd, 10) < 0)
{
    perror("Unable to listen to port: ");
    exit(-1);
}
```

## Step 2.

Initialize OpenSSL:

- Call `init_openssl()`.
- Call `create_context()`
- Call `configure_context`, providing the filenames for the certificate and private key files.
- Call `CRYPTO_thread_setup()` to set allow OpenSSL to operate safely in a multithreaded environment.

```
int c = sizeof(struct sockaddr_in);

// NEW: Initialize the SSL library
init_openssl();

// NEW: Create SSL Context
SSL_CTX *ctx = create_context();

// NEW: Configure context to load certificate and private keys
configure_context(ctx, "vincent.crt", "vincent.key");

// NEW: Configure multithreading in OpenSSL
CRYPTO_thread_setup();
```

## Step 3.

Call `accept` to accept any new connections to the server. Once accepted, call `connectSSL` to actually start the SSL session. In this example we call `pthread_create` to create a worker thread to talk to the client:

```
while(1)
{
    // Accept a new connection from the queue in listen. We will
    // build an echo server
    struct sockaddr_in client;
    connfd = accept(listenfd, (struct sockaddr *) &client, (socklen_t *) &c);

    char clientAddress[32];

    // Use inet_ntop to extract client's IP address.
    inet_ntop(AF_INET, &client.sin_addr, clientAddress, 32);

    printf("Received connection from %s\n", clientAddress);

    SSL *ssl = connectSSL(ctx, connfd);

    if(ssl != NULL)
    {
        pthread_t worker;

        // NEW: We pass in ssl instead of connfd
        pthread_create(&worker, NULL, workerThread, (void *) ssl);
        pthread_detach(worker);
    }
}
```

#### Step 4.

Within the worker thread, call SSL\_read and SSL\_write to communicate:

```
void *workerThread(void *p)
{
    // NEW: We pass in ssl instead of connfd
    SSL *ssl = (SSL *) p;

    char buffer[1024];

    int len=0;
    do
    {
        // NEW: We use SSL_read and SSL_write instead of read/write. Also
        // we use ssl rather than the file descriptor for the socket.
        // Read in data
        len = SSL_read(ssl, buffer, sizeof(buffer));

        // Echo it back. Note if len=0 means other side closed the connection.
        if(len > 0)
            SSL_write(ssl, buffer, strlen(buffer)+1);

    } while(len > 0);

    printf("Client closed connection.\n");
    SSL_free(ssl);
    pthread_exit(NULL);
}
```

See the tls\_servermt.cpp program for more details. To compile on Alex / Ubuntu / Windows 10 LINUX Subsystem:

```
g++ tls_servermt.cpp tls_server_lib.cpp tls_pthread.cpp -pthread -lssl -lcrypto -o
tls_servermt
```

To compile on OSX:

```
g++ tls_servermt.cpp tls_server_lib.cpp tls_pthread.cpp -L/usr/local/opt/openssl/lib
-l/usr/local/opt/openssl/include -pthread -lssl -lcrypto -o tls_servermt
```

**Note:** Your compilation may fail with a “ssl/ssl.h not found” or other similar error if the OpenSSL development libraries are not installed. To install:

```
sudo apt-get install libssl-dev
```

To run the server

```
./tls_servermt
```

## MAKING A TLS CLIENT

Let's look now at how to build a TLS client. See `tls_clientmt.cpp` for more details:

### Step 0.

Create a "verification callback function" at the top of your client program. This function is called when OpenSSL has completed a certificate verification. IT IS REQUIRED but it's not very important to us, so we will leave the body largely empty except for a return statement:

```
// Cert verification callback. Implement this in the client
int verify_callback(int preverify, X509_STORE_CTX *x509_ctx)
{
    return preverify;
}
```

It must be declared EXACTLY as shown or your code won't compile.

### Step 1.

Create a TCP/IP client program as per Activity 1:

```
// Create a structure to store the server address
int sockfd;
struct sockaddr_in serv_addr;
memset(&serv_addr, 0, sizeof(serv_addr));

// Create a socket
sockfd = socket(AF_INET, SOCK_STREAM, 0);

if(sockfd < 0)
{
    perror("Cannot create socket: ");
    exit(-1);
}

// Get host IP address
char hostIP[32];
struct hostent *he;

he = gethostbyname(av[1]);

if(he == NULL)
{
    perror("Unable to get host IP address: ");
    exit(-1);
}

struct in_addr **addr_list = (struct in_addr **) he->h_addr_list;

strncpy(hostIP, inet_ntoa(*addr_list[0]), sizeof(hostIP));
printf("Host %s IP address is %s\n", av[1], hostIP);

// Now form the server address
serv_addr.sin_family = AF_INET;
serv_addr.sin_port = htons(atoi(av[2]));
inet_pton(AF_INET, hostIP, &serv_addr.sin_addr);
```



### Step 2.

Initialize the SSL library, create a context, and call CRYPTO\_thread\_setup to allow OpenSSL to work safely in a multithreaded environment. Note: Your CA certificate (signing.pem) MUST be present in the current directory. This will be used in certificate verification later on:

```
// NEW: Initialize SSL
init_openssl();

// NEW: Create context
ctx = create_context("signing.pem");

// NEW: Enable multithreading in openssl
CRYPTO_thread_setup();
```

### Step 3.

Connect to the client using the TCP/IP connect call, then call connectSSL to initiate an SSL session, giving it the context you created in Step 2, the socket you created in Step 1, and Alex's IP address (or NULL to disable host name verification).

NOTE: In the example given below, I used the IP address 192.168.1.16. THIS WILL FAIL ON YOUR SYSTEM UNLESS YOUR ALEX'S IP ADDRESS IS ALSO 192.168.1.16! Change this to the IP address you used when you created Alex's certificate. You can also use a Common Name like alex.epp.com, as you did in the Studio.

```
// Now let's connect!
if(connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr)) < 0)
{
    perror("Error connecting: ");
    exit(-1);
}

ssl = connectSSL(ctx, sockfd, "192.168.1.16");
```

### Step 4. (OPTIONAL)

You can call printCertificate to print out details of the server's certificate. The printCertificate function is provided to you in tls\_server\_lib.cpp. You can look at how it works:

```
printf("CERTIFICATE DATA:\n");
printCertificate(ssl);
```

### Step 5.

Call `verifyCertificate` to verify the server's certificate. This function returns `TRUE` if the certificate is valid and was signed by the CA corresponding to the certificate file `signing.pem`. If you use a certificate file from another CA this step will fail.

```
if(!verifyCertificate(ssl))
{
    printf("SSL Certificate validation failed.\n");
    closeAndExit(-1);
}
else
    printf("SSL CERTIFICATE IS VALID\n");
```

Note: `closeAndExit` is a function created in `tls_clientmt.cpp` to just neatly deallocate resources used by OpenSSL, and to exit the program. This function is shown below:

```
void closeAndExit(int exitCode)
{
    close(sockfd);
    SSL_free(ssl);
    SSL_CTX_free(ctx);
    thread_cleanup();
    cleanup_openssl();
}
```

### Step 6.

Use `SSL_read` and `SSL_write` to read and write:

```
void *readThread(void *p)
{
    SSL *ssl = (SSL *) p;
    while(!exitFlag)
    {
        int len;
        char buffer[1024];
        len = SSL_read(ssl, buffer, sizeof(buffer));

        if(len > 0)
            printf("Received %d bytes from server: %s\n", len, buffer);

        // Exit if we read zero bytes. This means the server closed the
        // connection
        exitFlag=(len == 0);
    }

    printf("Read thread exiting\n");
    pthread_exit(NULL);
}
```

To compile on Alex / Ubuntu / Windows 10 Linux Subsystem:

```
g++ tls_clientmt.cpp tls_client_lib.cpp tls_pthread.cpp -lssl -lcrypto -pthread -fpermissive  
-o tls_clientmt
```

To compile on OSX:

```
g++ tls_clientmt.cpp tls_client_lib.cpp tls_pthread.cpp -lssl -lcrypto -pthread  
-L/usr/local/opt/openssl/lib -I/usr/local/opt/openssl/include -o tls_clientmt
```

To run:

```
./tls_clientmt <host name> 5000
```

Here <host name> is the name or IP address of the machine running tls\_servermt. if you are running tls\_servermt on the same machine:

```
./tls_clientmt localhost 5000
```