Reporte práctica 1

Autor: Miguel Pérez García

Descripción del código

Todo el código puede ser encontrado en el este repositorio.

Estas son las funciones que se implementaron y se exponen como parte de la librería, a continuación explicaré la implementación que e hizo para cada una.

```
void set_encryption_key(uint8_t key[16], uint8_t iv[16]);

void config_crc(CRC_Type *base, uint32_t polynomial, uint32_t seed);

struct netconn* start_server(const ip_addr_t *addr, u16_t port);
struct netconn* accept_connection(struct netconn *server);
MessageRequest wait_request(struct netconn *client, uint8_t *result);
MessageResponse get_response(MessageRequest *request);
uint8_t write_response(struct netconn *sender, MessageResponse response);
void close_client(struct netconn *conn);
```

`set_encryption_key`: Esta función tiene como única responsabilidad almacenar las llaves que se utilizarán para el cifrado. Existe una función `intit_encryption_context` que es quien interactúa con la librería para configurar el contexto de AES.

```
void set_encryption_key(uint8_t key[KEY_SIZE], uint8_t iv[KEY_SIZE]){
    memcpy(int_key, key, KEY_SIZE);
    memcpy(int_iv, iv, KEY_SIZE);
}
```

`config_crc`: Con esta función se configura el CRC en la tarjeta. La función `int_config_crc` es quien configura el periférico como tal, de esta forma las otras funciones pueden utilizar la interna sin tener que estar pasando como parámetro los valores de la semilla y el polinomio.

```
void config_crc(CRC_Type *base, uint32_t polynomial, uint32_t seed){
    int_polynomial = polynomial;
    int_seed = seed;
    int_config_crc(base);
}
```

`start_server`: Esta función permite crear el servidor, además de asignar la dirección IP y el puerto, deja al socket escuchando por conexiones.

```
struct netconn* start_server(const ip_addr_t *addr, u16_t port) {
    struct netconn *server;

#if LWIP_IPV6
    // Aquí se crea el socket que se utilizará para recibir las conexiones TCP
para IP V6
    server = netconn_new(NETCONN_TCP_IPV6);
    netconn_bind(server, addr, port);
#else /* LWIP_IPV6 */
    // Aquí se crea el socket que se utilizará para recibir las conexiones TCP
para IP V4
    server = netconn_new(NETCONN_TCP);
    netconn_bind(server, addr, port);
#endif /* LWIP_IPV6 */
    LWIP_ERROR("tcpecho: invalid conn", (server != NULL), return NULL;);

    // Se le indica al socket que debe estar escuchando en el puerto,
    // ahora se puede esperar por conexiones
    netconn_listen(server);

    return server;
}
```

`accept_connection`: Con este método se abstrae el esperar conexiones, de haber un problema con la conexión se regresa un NULL para indicar que no hubo una conexión efectiva.

```
struct netconn * accept_connection(struct netconn *server){
    struct netconn* newconn;

    // Se le indica al socket que debe esperar por conexiones
    err_t err = netconn_accept(server, &newconn);
    if(err != ERR_OK) return NULL;

    // Regresamos ahora el socket del cliente que se conectó
    return newconn;
}
```

`wait_request`: En este método se indica que debe esperar por datos en el socket del cliente, además al recibirlos los procesa para generar la petición que está llegando.

```
MessageRequest wait_request(struct netconn *client, uint8_t *result){
    struct netbuf *buf;
    void *data;
    u16_t len;
    // Espera por datos del cliente
    if(netconn_recv(client, &buf) == ERR_OK) {
        do {
            // Obtiene los bytes del buffer de red
```

```
            netbuf_data(buf, &data, &len);
            // Obtiene la petición de bytes recibidos
            MessageRequest rec_request = from_packet((uint8_t*)data, len, result);
            if(0 == *result) {
                // De haber sido un paquete válido, se regresa la petición que se
recibió
                netbuf_delete(buf);
                *result = 0;
                return rec_request;
            }
        } while (netbuf_next(buf) >= 0);
        netbuf_delete(buf);
    } else {
        // Si ya no se están recibiendo datos porque el cliente se desconectó,
        // se culmina la conexión y el resultado se indica como un valor para
indicar
        // que terminó
        close_client(client);
        *result = 1;
    }

    MessageRequest request = {};
    *result = 1;
    // Se regresa una respuesta vacía
    return request;
}
```

`get_response`: Dada una petición, se obtiene la respuesta adecuada basado en el tipo del enumerable

```
MessageResponse get_response(MessageRequest *request) {
    MessageResponse response = {
            .type = request->type
    };
    switch(request->type){
        case A: { response.response = "This is response to A"; break; }
        case B: { response.response = "This is response to B"; break; }
        case C: { response.response = "This is response to C"; break; }
        case D: { response.response = "This is response to D"; break; }
        case E: { response.response = "This is response to E"; break; }
        case F: { response.response = "This is response to F"; break; }
        case G: { response.response = "This is response to G"; break; }
        case H: { response.response = "This is response to H"; break; }
    }

    return response;
}
```

`write_response`: Se escribe la respuesta a ser enviada en el socket del cliente

```
uint8_t write_response(struct netconn *sender, MessageResponse response){
    uint8_t response_buffer[256] = {0};
    // A partir de la respuesta, se genera el paquete en bytes, durante este
proceso
    // también se encriptan y se le añade el CRC.
    uint32_t written_bytes = to_packet(&response, response_buffer);

    // Ahora se envían por el socket del cliente conectado
    err_t err = netconn_write(sender, response_buffer, written_bytes,
NETCONN_COPY);
    if (err != ERR_OK) {
        // En caso de haber un error, se marca regresando un 1.
        printf("tcpecho: netconn_write: error \"%s\"\n", lwip_strerr(err));
        return 1;
    }
    // Si todo salió bien, se regresa un 0
    return 0;
}
```

`close_client`: En esta función solo se agrupa el proceso para cerrar el socket del cliente.

```
void close_client(struct netconn *conn){
    netconn_close(conn);
    netconn_delete(conn);
}
```

`from_packet`: Convierte de un buffer a una estructura que representa a un una petición, además tiene un parámetro para identificar posibles errores en el proceso.

```
MessageRequest from_packet(uint8_t *buffer, uint32_t length, uint8_t *result){
    CRC_Type *crc = CRC0;
    // Configuramos el CRC0
    int_config_crc(crc);
    // Copiamos la información del buffer sin el CRC al periférico
    CRC_WriteData(crc, buffer, length - 4);
    // Obtenemos el CRC
    uint32_t checksum = CRC_Get32bitResult(CRC0);

    // El CRC está en 4 bytes porque es un entero de 32 bits, por ello necesitamos
reconstruirlo
    uint32_t received_checksum = 0;
    int8_t i = 0;
    for(i = 0; i < 4; i++){
        received_checksum |= (uint32_t)(buffer[length - 1 - i] << ((3 - i) * 8));
    }

    // Si no coinciden, regresar con un error
    if(checksum != received_checksum){
```

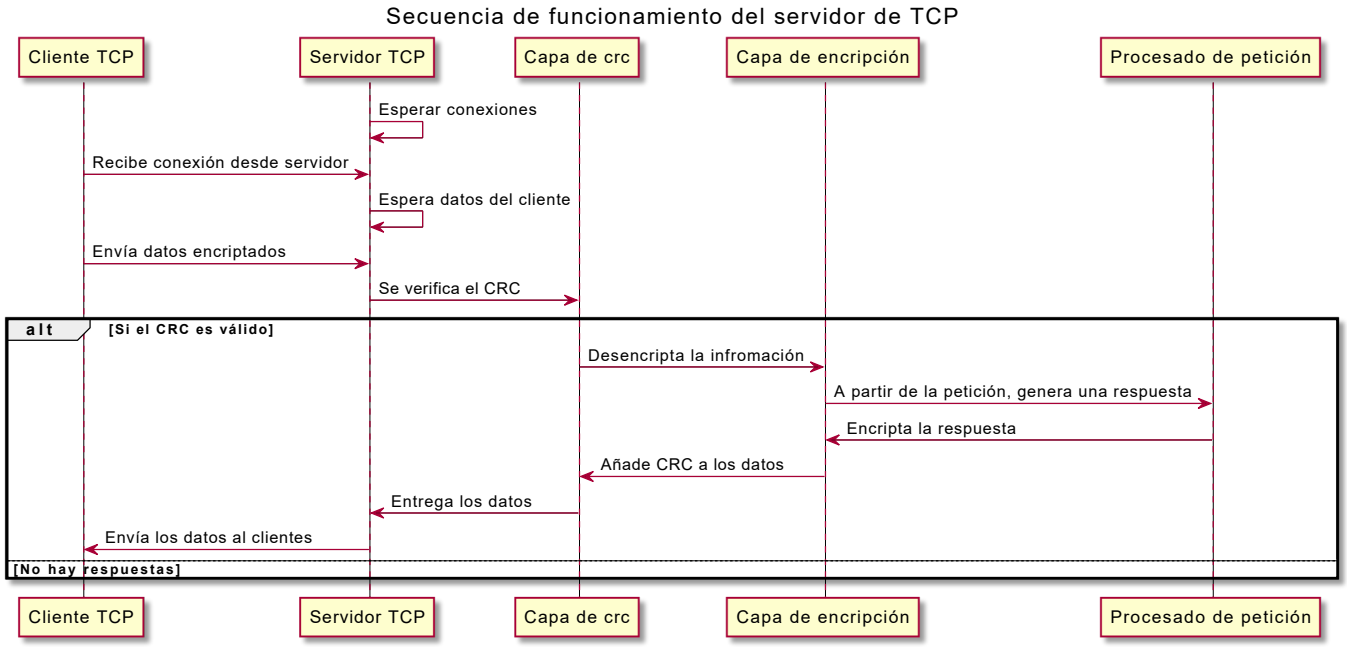
```
        *result = 1;
        MessageRequest temp = { };
        return temp;
    }

    // Ahora desciframos el "cuerpo" del paquete
    size_t len_no_crc = length - 4;
    uint8_t *temp = malloc(len_no_crc * sizeof(uint8_t));
    memcpy(temp, buffer, len_no_crc);
    init_encryption_context();
    AES_CBC_decrypt_buffer(&aes_ctx, temp, len_no_crc);

    // Y convertimos del buffer a la estructura que representa el request
    MessageRequest message = from_decrypted_packet(temp, len_no_crc - 2);
    free(temp);

    *result = 0;
    return message;
}
```

`to_packet`: Convierte de una respuesta a un buffer de salida

```
uint32_t to_packet(MessageResponse *message, uint8_t *buffer){
    // Obtenemos el tamaño en bytes de la respuesta que enviaremos
    uint32_t size = get_message_size(message);
    uint8_t *pre_enc = malloc(size * sizeof(uint8_t));
    memset(pre_enc, 0, size);
    // Convertimos la respuesta a un buffer de bytes
    to_decrypted_packet(message, pre_enc);

    // Le damos padding al buffer porque se requiere que el tamaño sea múltiplo de
16
    uint8_t padded_msg[512] = {0};
    size_t real_size = strlen(pre_enc);
    size_t padded_len = real_size + (16 - (real_size % 16));
    // Y copiamos los datos del buffer encodeado al buffer con padding
    memcpy(padded_msg, pre_enc, size);
    free(pre_enc);

    // Iniciamos el contexto de cifrado
    init_encryption_context();
    AES_CBC_encrypt_buffer(&aes_ctx, padded_msg, padded_len);

    // Iniciamos el CRC
    int_config_crc(CRC0);
    CRC_WriteData(CRC0, padded_msg, padded_len);
    // Acomodamos el CRC en 4 bytes porque es de 32 bits
    uint32_t checksum = CRC_Get32bitResult(CRC0);
    for(size_t i = 0; i < 4; i++)
        padded_msg[padded_len + i] = (uint8_t)(checksum >> (i * 8));

    // Añadimos los 4 bytes que corresponden al CRC
```

```
    padded_len += 4;
    // Copiamos al buffer de salida
    memcpy(buffer, padded_msg, padded_len);

    // Regresamos el tamaño total escrito
    return padded_len;
}
```

Diagramas de funcionalidad



Problemas encontrados

Uno de los primeros problemas a los que me enfrenté fue a que pensé que al configurar la unidad de CRC una única vez y solo estar cambiando los datos para obtener el CRC del nuevo stream de datos. Sin embargo, no tardé mucho en darme cuenta que se tenía que configurar cada vez que se va a utilizar.

El principal problema al que me enfrenté fue el poder regresar mas de una cosa en una llamada de un método, primero tenía el siguiente código:

```
MessageRequest wait_request(struct netconn *client, uint8_t *result, struct
netconn *client){
    err_t err = netconn_accept(server, &client);
    // Se omité el resto del código
}
```

Yo esperaba que al regresar del método, el socket del cliente fuera válido para utilizarse en las llamadas subsecuentes ya que se utilizaba directamente en la llamada para aceptar clientes nuevos, sin embargo no lo era. Aunque después me percaté que al implementarlo de esta forma, cada vez que se esperara recibir un mensaje estaría esperando una conexión nueva lo cual también es incorrecto y mejor lo cambié a la

implementación final, en la que se espera la conexión del cliente y posteriormente se trabaja solo con ese socket.

Conclusiones

Si bien yo en otras cuestiones me ha tocado trabajar escribiendo servidores TCP, el modo de implementarse es un poco distinto en un lenguaje de alto nivel, sin embargo los conceptos son bastante similares. La diferencia mas grande fue en la parte de encripción de datos, usualmente en lenguajes de alto nivel existe algún estilo de *stream* en el que se escriben los datos y se obtienen los encriptados pero tampoco es algo muy distinto a lo que he utilizado antes.