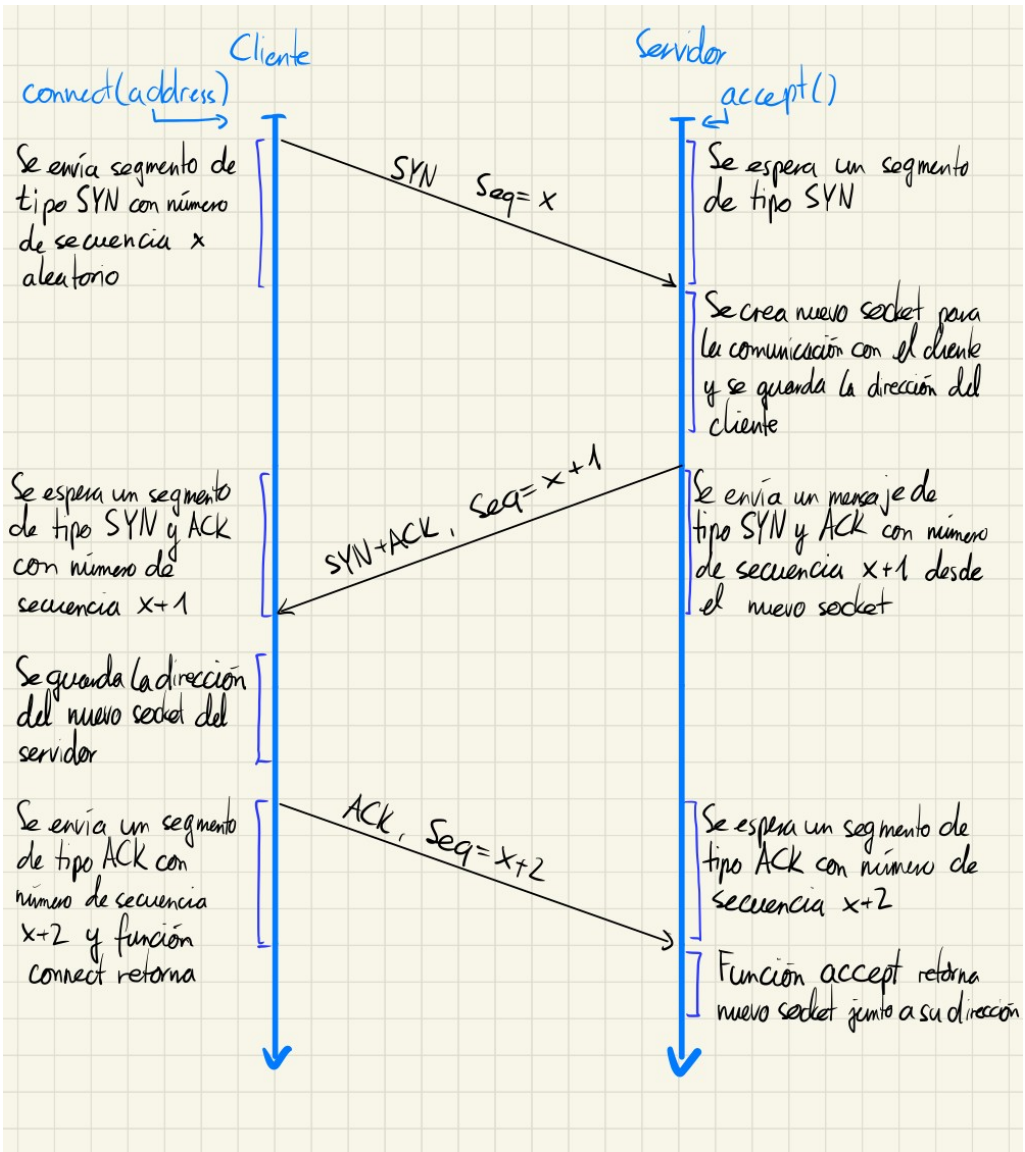
Informe Actividad 3:

Sockets orientados a conexión con Stop & Wait

Nombre: Vicente Olivares Gómez

Funcionamiento de 3-way handshake

El 3-way handshake es el procedimiento que permite establecer un canal para la conexión entre el socket de un cliente y el socket de un servidor, hablando en ambos casos de sockets TCP. Este tiene 3 partes, de ahí proviene el nombre, y cada parte consiste en el envío y la recepción de un segmento con headers y un número de secuencia. La siguiente imagen muestra un diagrama del funcionamiento del 3-way handshake.

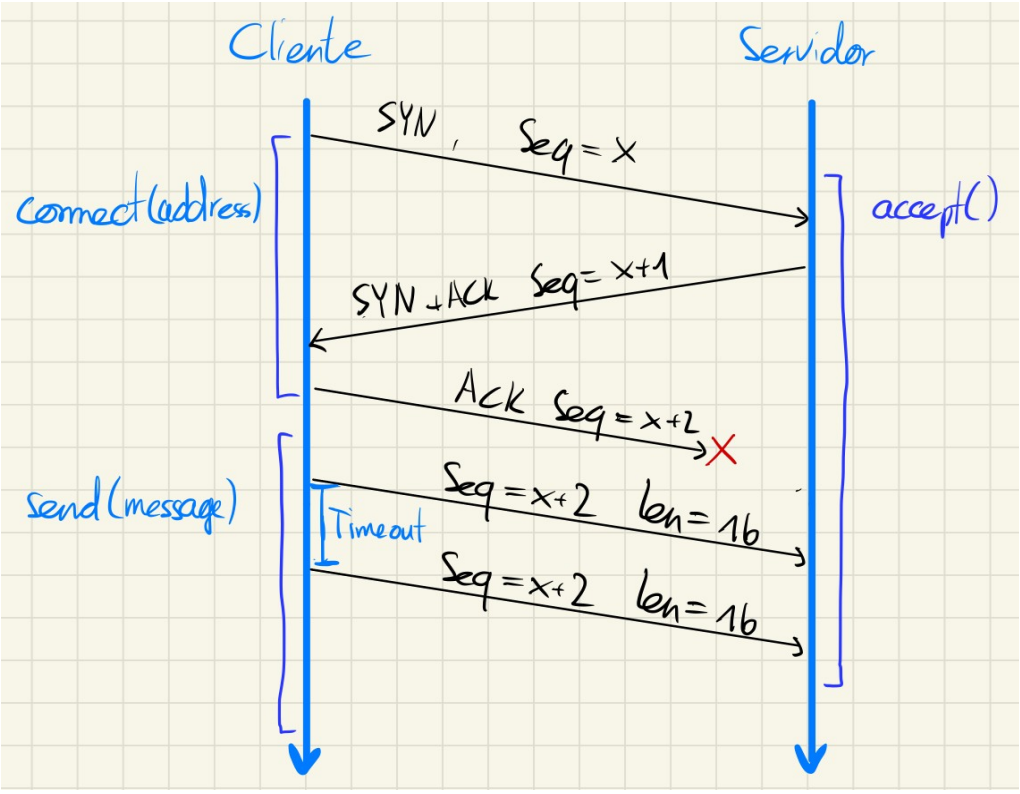


Desde la perspectiva del cliente, el 3-way handshake comienza cuando se llama al método connect(address), siendo *address* la dirección (IP, puerto) del servidor con el que se desea comunicar. Al llamar a este método, el cliente envía un segmento de sincronización, es decir con header SYN, con un número de secuencia *x* generado aleatoriamente, solicitando una conexión de cliente a servidor, al servidor de dirección address. Luego espera a recibir un segmento de confirmación y sincronización, es decir con headers ACK y SYN, con número de secuencia *x*+1. Después de recibirlo, guarda la dirección del socket que le envió el segmento y envía a esa dirección un mensaje con header ACK con número de secuencia x+2, confirmando la conexión de servidor a cliente. El método connect retorna una vez el último mensaje de confirmación haya sido enviado, lo que marca el final del handshake para el cliente.

Desde la perspectiva del servidor, el saludo comienza cuando se llama al método accept(). Al invocar accept, el servidor espera un segmento de sincronización. Cuando lo recibe, crea un nuevo socket que servirá para establecer el canal con el socket del cliente, guarda la dirección del cliente y el número de secuencia *x* del segmento recibido. Luego, desde el nuevo socket se envía un segmento de confirmación y sincronización con número de secuencia *x*+1, y se vuelve a esperar por un segmento, esta vez de confirmación y con número de secuencia *x*+2. Al recibirlo, la función accept retorna una tupla con el socket utilizado para la comunicación con el cliente y la dirección de este socket, marcando el final del 3-way handshake.

Pérdida del último segmento ACK en 3-way handshake

El funcionamiento del 3-way handshake visto anteriormente es incompleto, ya que no se están considerando pérdidas de segmentos, lo cual lleva a diversos casos bordes. Uno de ellos es que el segmento de confirmación enviado por el cliente al servidor, correspondiente al tercer y último paso del handshake, no llegue al cliente. En la siguiente imagen se puede ver un diagrama de tal caso.



Desde la perspectiva del cliente, el handshake terminó y el método connect retornó, puesto que el segmento de confirmación fue enviado. Por lo tanto es muy probable que el cliente llame al método send para el envío de datos hacia el servidor, y envíe un segmento con número de secuencia x+2 sin el header ACK, suponiendo que se sigue la implementación de la actividad. Sin embargo, desde la perspectiva del servidor, el handshake no ha terminado, porque no se recibió un segmento de confirmación con el número de secuencia x+2. En cambio, el servidor recibirá un segmento de número de secuencia x+2 pero sin header ACK, por lo que lo ignorará y seguirá esperando. Como el segmento enviado por el cliente ya es parte del envío de datos, al ser ignorado por el servidor, el cliente no recibirá respuesta causando que se cumpla el timeout asociado a Stop & Wait, haciendo que se vuelva a enviar el segmento. Si no hay intervención externa, el cliente seguirá enviando el segmento y el servidor seguirá rechazándolo, por lo que se puede caer en un loop infinito, mostrando la importancia de manejar este caso borde.

Código de la Actividad

El código de Python de esta actividad está dividido en cuatro archivos: *constants.py*, *SocketTCP.py, client.py* y *server.py.* Estos dos últimos son los archivos que se deben ejecutar para hacer funcionar el envío de datos con Stop & Wait.

Archivo *constants.py*

Dentro de *constants.py* se encuentran definidas una constante utilizada en los archivos *server.py* y *client.py*. Esta es SERVER_ADDRESS, que es la tupla con la IP y el puerto utilizados correspondientes al servidor. Es utilizada como parámetro para llamar al método *bind* en *server.py* y al método *connect* en client.py. Se creó este archivo para tener fácil y rápido acceso a SERVER_ADDRESS y modificarlo, en caso de ser necesario.

Archivo *SocketTCP.py*

En este archivo se encuentra definida la clase SocketTCP utilizada para implementar a un socket que sigue el protocolo TCP. Al comienzo del archivo se definen las constantes *SEGMENT_SEPARATOR, SYN*, *ACK*, *SEQ*, *FIN* y *DATA.* Estas seis constantes se definieron con el objetivo de tener rápido acceso a su valor y evitar errores al momento de escribirlas, ya que SEGMENT_SEPARATOR corresponde a la cadena de bytes que marca la separación entre headers en un segmento y el resto corresponde a strings que son las llaves de los diccionarios obtenidos al parsear segmentos, por lo tanto son valores que se repiten bastante en el archivo y si se llegan a escribir mal, pueden causar errores difíciles de detectar.

La clase SocketTCP tiene los siguientes atributos:

- ***socket_udp*:** Socket no orientado a conexión utilizado para implementar un Socket orientado a conexión.
- ***my_address*:** Guarda la dirección a la cual debe escuchar *socket_udp* luego de llamar al método *bind*. Inicialmente tiene el valor None.

- ***other_side_address*:** Guarda la dirección del socket con el que se establece el canal. Tiene el valor None al inicio y cuando se cierra el canal de comunicación.
- ***seq_num*:** Guarda el número de secuencia del último segmento enviado por el socket.
- ***udp_buff_size*:** Almacena el tamaño del buffer de recepción de *socket_udp.*
- ***bytes_to_receive:*** Guarda el número de bytes que faltan por recibir de un mensaje, ayudando a definir si una llamada al método *recv* corresponde a la recepción de un nuevo mensaje o de la continuación de un mensaje que no cupo en el buffer de recepción.
- ***last_asigned_port:*** Guarda el último número de puerto asignado a un socket creado por el método *accept*, para evitar que se asigne el mismo puerto a más de un socket.
- ***last_overflow:*** Guarda los bytes que fueron recibidos en una llamada al método recv y no cupieron en el buffer del socket TCP.
- ***timeout:*** Corresponde al tiempo que espera el cliente antes de volver a enviar un segmento si no recibe respuesta de parte del servidor, en el contexto de Stop & Wait.

La clase tiene también definidos los siguientes métodos:

- (Método estático) ***parse_segment(segment : bytes):*** Retorna un diccionario con los headers y el área de datos del segmento recibido. El segmento recibido debe seguir el formato indicado en la descripción del método *create_segment.* Las llaves del diccionario generado son “ACK”, “SYN” y “FIN” correspondientes a los headers del segmento y están asociados al número 1 si el segmento posee dicho header y el número 0 en caso contrario. También se tiene la llave “DATA” que está asociada a los bytes del área de datos del segmento entregado.
- (Método estático) ***create_segment(segment_dict : dict):*** Retorna un segmento en bytes con los headers y el área de datos indicada en el diccionario recibido. Los headers y el área de datos deben estar asociados a las llaves mencionadas en la descripción del método *parse_segment.* El segmento retornado sigue el formato en bytes: SYN|||ACK|||FIN|||SEQ|||DATA. SYN, ACK, FIN y SEQ son los headers del segmento y DATA representa al área de datos. SYN, ACK y FIN son de largo 1 y pueden tomar los valores uno y cero, indicando si tienen dicho header o no respectivamente. SEQ es una cadena de 4 bytes y representa el número de secuencia. Se decidió hacer más largo que el resto de headers, ya que el valor que contiene aumenta con cada segmento enviado y un byte no sería suficiente en caso de querer enviar mensajes largos. Finalmente DATA es una cadena de a lo más 16 bytes.
- ***bind(address : tuple):*** Indica a *socket_udp* la dirección a la cual debe escuchar y guarda la dirección recibida en *my_address.*
- ***connect(address : tuple):*** Establece un canal de conexión con el socket que escucha la dirección recibida. El método se encarga de llevar a cabo el 3-way handshake desde el lado del cliente. El método no maneja pérdidas de segmentos, porque no alcancé a implementarlo.
- ***accept():*** Establece un canal de conexión con algún socket que lo esté solicitando. El método se encarga de llevar a cabo el 3-way handshake desde el lado del servidor. Retorna una tupla con

el socketTCP que se encargará de la comunicación con el cliente y la dirección del socket. El método no maneja pérdidas de segmentos, porque no alcancé a implementarlo.

- ***send(message : bytes):*** Envía el mensaje recibido al servidor con el que se estableció un canal. El método se encarga de enviar el mensaje por segmentos utilizando Stop & Wait desde el lado del cliente.
- ***recv(buff_size : int):*** Retorna una cadena de bytes enviada desde el cliente con el que se estableció un canal. Esta cadena tiene largo máximo *buff_size*. El método se encarga de recibir el mensaje por segmentos mediante Stop & Wait desde el lado del cliente. Si se desea recibir un mensaje más largo que el tamaño del buffer, se puede volver a invocar el método las veces que sea necesario hasta que se reciba todo el mensaje. No se perderán partes del mensaje en este proceso, ya que los bytes que fueron recibidos por *socket_udp* pero no cupieron en el tamaño de buffer indicado por *buff_size* son guardados en el atributo *last_overflow* y son incorporados en el siguiente llamado del método.
- ***close():*** Solicita el cierre del canal de comunicación establecido. El método puede ser invocado tanto por cliente como por servidor. El método no maneja pérdidas de segmentos, porque no alcancé a implementarlo. Termina dejando al atributo *other_side_address* en None. Lo cual sirve para revisar que el fin de conexión fue exitoso.
- ***close_recv():*** Cierra el canal de comunicación establecido ante una solicitud de fin de conexión. El método puede ser invocado tanto por cliente como por servidor. El método no maneja pérdidas de segmentos, porque no alcancé a implementarlo. Termina dejando al atributo *other_side_address* en None. Lo cual sirve para revisar que el fin de conexión fue exitoso.

Archivo *client.py*

Este archivo representa al cliente. Es necesario entregarle el archivo *file_to_send.txt* por entrada estándar al momento de ejecutarlo para su correcto funcionamiento. Al ejecutarlo parte leyendo el archivo de texto de la entrada estándar y lo codifica en bytes para usarlo luego. Luego se crea el socket *client_socketTCP* y se conecta con el servidor. Después, se realizan cuatro tests, los que corresponden al envío de 4 mensajes con el método *send* del socket creado. Los tres primeros corresponden a los tests de la actividad y el cuarto es el contenido del archivo de texto leído. Finaliza cerrando la conexión con el método *close* y se testea que se haya cerrado correctamente, consultando el atributo *other_side_address* del socketTCP, pues este debería ser None en caso de que se haya cerrado correctamente. En caso contrario, este atributo debería seguir siendo la dirección del socket del servidor.

Archivo *server.py*

Este archivo representa al servidor. Al ejecutarlo, comienza leyendo el archivo de texto *file_to_send.txt,* por lo tanto es necesario tener al archivo de texto en el mismo directorio que server.py para el correcto funcionamiento de este. El contenido del archivo de texto se guarda para su uso futuro. Luego, se crea el socket *server_socketTCP,* se llama a *bind* para que escuche a la dirección SERVER_ADDRESS del

archivo *constants.py* y se llama al método *accept* para establecer un canal con el cliente. Después se realizan cuatro tests. Cada test corresponde a la recepción de un mensaje enviado por el cliente en *client.py.* Los tres primeros corresponden a los tests de la actividad y el cuarto corresponde a la recepción del contenido del archivo de texto leído al comienzo de la ejecución. En cada test se verifica que el mensaje recibido sea exactamente el mismo que se esperaba, y se realiza un print indicando si se tuvo éxito o se falló. Finalmente se llama al método *close_recv* para cerrar la comunicación con el socket del cliente y se testea que se haya cerrado correctamente, consultando el atributo *other_side_address* del socketTCP, pues este debería ser None en caso de que se haya cerrado correctamente. En caso contrario, este atributo debería seguir siendo la dirección del socket del cliente.

Pruebas de la Actividad

Para las pruebas del código se reutilizaron los códigos de prueba de los tests de la actividad, agregando un cuarto test, que corresponde al envío del contenido del archivo de texto ingresado por entrada estándar. Respecto al modo debug, los métodos *send* y *recv* son los únicos que manejan pérdidas, por lo que son los únicos que tienen mensajes de este modo, pero como connect ni accept manejan pérdidas, solo es posible visualizarlo si se utiliza un bajo porcentaje de pérdidas, en mi caso con pérdidas de hasta 15%.

Al realizar las pruebas sin pérdidas, se verificó con el test 1 que al enviar un mensaje del mismo tamaño del buffer, este llega correctamente y con los siguientes tres tests se verificó que un mensaje llega correctamente cuando este es de largo mayor al tamaño del buffer.