Actividad: Sockets orientados a conexión con Stop & Wait

Marcar esta página

En esta actividad **implementaremos sockets orientados a conexión** usando solo ***sockets UDP* y *Stop & Wait***. Para ello usted deberá implementar un emisor, un receptor y la clase `socketTCP`.

A continuación veremos algunos de los conceptos que necesitaremos saber para realizar esta actividad. Para esta actividad además necesitará del material provisto en el **video: TCP, UDP y Stop & Wait.**

Puntaje: 4.5 pts código + 1.5 pts informe

El desglose de puntaje por funcionalidad del código es el siguiente (Si su código pasa las pruebas debería cumplir con los puntos mencionados aquí):

- (+1.0) El handshake funciona correctamente
- (+1.0) El cierre funciona correctamente
- (+2.5) Las funciones `send` y `recv` funcionan correctamente (esto incluye manejo de pérdidas)

Antes de comenzar

- **Stop & Wait:** El protocolo de control de errores *Stop & Wait* es utilizado para asegurarse de que los paquetes enviados por el emisor lleguen al receptor, realizando reenvíos de paquetes en caso de que alguno se pierda. En *Stop & Wait*, cuando el emisor envía un paquete, este se queda esperando a que el receptor le confirme que dicho paquete llegó mediante un mensaje ACK. Una vez el emisor recibe el mensaje ACK del receptor, este procede a enviar el siguiente paquete de información. El emisor esperará el mensaje ACK por un periodo de tiempo '*timeout*', en caso de no recibir el mensaje ACK que esperaba dentro del periodo de timeout el emisor reenviará el paquete al receptor.
- **Simular pérdida y delay (netem):** Para testear el comportamiento de una red bajo pérdida de mensajes y/o retardo (delay) en el tiempo de envío de dichos mensajes podemos usar netem. Podemos ejecutar netem en *localhost* usando el siguiente comando

  ```
  % sudo tc qdisc add dev lo root netem loss [% de pérdida] delay [tiempo de delay]
  ```

  Para detener la pérdida de mensajes pueden usar:

  ```
  % sudo tc qdisc del dev lo root netem
  ```

- **Simulando pérdidas a mano:** En el módulo de sockets (semana 1) se provee código para simular pérdidas a mano en caso de **no** poder utilizar *netem*. Si no cuenta con un sistema operativo que le permita usar *netem*, deberá implementar pérdidas a mano.

Actividad

Para esa actividad deberá programar una clase `socketTCP` la cual implementará sockets orientados a conexión usando **Stop & Wait**. Esta clase deberá funcionar de forma similar a los sockets orientados a conexión de python. Además usted deberá implementar un cliente y un servidor que usen objetos tipo `socketTCP` para comunicarse. Su clase `socketTCP` deberá utilizar sockets UDP para enviar la información, por lo que su tarea será implementar `TCP` simplificado con Stop & Wait incluyendo **timeouts**, **mensajes** `ACK`, etc. Su cliente debe ser capaz de enviar un archivo a la dirección `(IP, Puerto)` donde se encuentre escuchando el servidor. Por ejemplo, si queremos enviar el archivo `'archivo.txt'` a la dirección '`(localhost, 8000)`' veríamos algo como:

```
% python3 cliente.py localhost 8000 < archivo.txt
```

Dado que los programas implementan comunicación confiable se espera que el archivo sea recibido por el programa receptor de manera íntegra.

Para probar su implementación defina que el máximo tamaño de paquete (sin incluir headers) es de 16 bytes, por lo tanto cualquier comunicación que supere esa cantidad de bytes debe ser dividida en múltiples paquetes. Se sugiere cargar todo el archivo a memoria y usarlo como un arreglo de bytes para dividirlo más fácilmente en paquetes. Recuerde que usted debe implementar comunicación confiable junto a todo lo que necesite, como timeouts y mensajes ACK.

Los pasos a seguir son:

1. Cree un cliente y un servidor **provisorios** que se comuniquen con sockets UDP. El cliente debe leer un archivo cuya ubicación o ruta se recibe desde *entrada estándar* (usando la función `input` de python). Luego, el cliente debe enviar los contenidos de este archivo al receptor en trozos de a lo más 16 bytes. El servidor debe recibir los mensajes desde el cliente e imprimir el contenido del archivo en salida estándar. Compruebe que el archivo se envió completamente, **si bien en este punto no se ha implementado comunicación confiable** el envío de mensajes pequeños a localhost no debiese tener pérdida.

   Test 1: Envíe un mensaje de largo mayor a 16 y vea que todo llega sin pérdidas. Note que en este punto no tiene cómo saber el largo del mensaje, por lo que está ok si su receptor se queda "pegado" esperando más mensajes una vez ya recibió todo.
   Test 2: Repita la prueba, esta vez induzca pérdida usando `netem` y vea que ocurren pérdidas. Luego de probar su código **recuerde detener las pérdidas** tal como se indica en la sección ***Antes de comenzar***. Si no puede utilizar `netem`, simule pérdidas como se indica en la sección ***Material e Indicaciones*** en la **actividad de Sockets**.

   ```
   % sudo tc qdisc add dev lo root netem loss 20.0% delay 0.5s
   ```

2. Ahora vamos a crear la clase `SocketTCP` (guarde la clase en un tercer archivo distinto al cliente y el servidor). **El constructor** de esta clase deberá ser capaz de almacenar todos los recursos que va a necesitar para la comunicación (socket UDP, dirección de destino, dirección de origen, número de secuencia, **todo lo que usted considere necesario**). Su constructor no debe recibir parámetros, es decir, se invoca como:

   ```
   mi_socket = SocketTCP()
   ```

   `>>>` En la sección ***Material e indicaciones para la actividad*** puede encontrar un ejemplo de cómo crear clases en python.

3. Considerando headers `TCP` que contengan la información necesaria para usar *Stop & Wait* como se mostró en el video*, añada a su clase dos funciones estáticas: `parse_segment` que pueda pasar segmentos TCP a alguna estructura de datos más cómoda y `create_segment` que pueda crear segmentos a partir de dicha estructura de datos. Para la estructura de sus headers TCP puede usar como guía el ejemplo provisto más abajo en la sección ***Material e indicaciones para la actividad*** o puede codificar su header usando bytes (use lo que le sea más cómodo considerando ventajas y desventajas).
   * Es decir: ACK, SYN, FIN y seq.

   Test: Modifique el código creado en el ***paso 1*** para que el emisor envuelva cada segmento en headers TCP antes de enviarlos y haga que su receptor extraiga el contenido de los segmentos recibidos. Es decir, ahora cada segmento enviado por el emisor comienza con un área de headers y luego continúa con el área de contenido el cual contiene a lo más 16 bytes. De esta forma, el receptor deberá ser capaz de extraer el contenido del segmento para rearmar el mensaje original.

4. Implemente *3-way handshake* para que el emisor le avise al receptor que va a comenzar el envío de mensajes. Por ahora **no** se preocupe de manejar pérdidas. **Para ello añada a la clase `socketTCP` las siguientes funciones:**
   `- bind(address)` : Función que se encarga de que el objeto `socketTCP` escuche en la dirección `address`.

   `- connect(address)`: Función que inicia la conexión desde un objeto `socketTCP` con otro que se encuentra escuchando en la dirección `address`. Dentro de esta función deberá implementar el lado del cliente del *3-way handshake*. Por simplicidad, haga que su número de secuencia inicial sea elegido aleatoriamente entre 0 y 100.

   `- accept()`: Función que se encuentra esperando una petición de tipo SYN. Dentro de esta función deberá implementar el lado del servidor del *3-way handshake*. Si el handshake termina de forma exitosa, **esta función deberá retornar un nuevo** objeto del tipo `socketTCP` **junto a la dirección** donde se encuentra escuchando (bind) dicho objeto. La **dirección** del **nuevo socket** debe ser **distinta** a la del socket que llamó a `accept()`. Cuide que su nuevo socket esté correctamente asociado a la nueva dirección y que recuerde los números de secuencia, pues es este nuevo socket el que será utilizado posteriormente para enviar y recibir mensajes.

   >> INDICACIÓN: Dibuje el diagrama del *3-way Handshake* y asocie los pasos correspondientes a las funciones `connect` y `accept`. **Añada el diagrama a su informe final. <<**
   Test: Pruebe que su handshake funciona para ello ejecute los siguientes códigos en su cliente y servidor. Recuerde que debe hacer `import` de su clase SocketTCP. Por ahora no es necesario que maneje pérdidas en el handshake.

   ```
   # server
   server_socketTCP = SocketTCP.SocketTCP()
   server_socketTCP.bind(address)
   connection_socketTCP, new_address = server_socketTCP.accept()
   ```

   ```
   # client
   client_socketTCP = SocketTCP.SocketTCP()
   client_socketTCP.connect(address)
   ```

5. Implemente *Stop & Wait* usando un timeout fijo definido por ustedes. Para ello aproveche la función `settimeout` de los sockets no orientados a conexión. **Note que estos timeouts levantan errores al cumplirse el tiempo límite**. Su código debe **manejar** estos errores para saber cuándo debe retransmitir un segmento (para ver cómo implementar un timeout vea ***Materiales e Indicaciones para la actividad***).
   Para implementar *Stop & Wait* añada a la clase `socketTCP` las siguientes funciones:
   `- send(message)`: Esta función será la encargada de manejar *Stop & Wait* desde el lado del emisor tal como vimos en la versión simplificada mostrada en el video. Su función `send` deberá encargarse de dividir el mensaje `message` en trozos de tamaño máximo 16 bytes. Para ello, use como guía el código provisorio que implementó en el ***paso 1***.

   Para evitar que el receptor espere para siempre, **haga que el primer segmento enviado por la función `send`** le informe al receptor **el largo en bytes del mensaje** `message` que le va a enviar (`message_length = len(message)`) y luego, a partir del segundo segmento, comience a enviar el mensaje. **Note** que `send` usa como número de secuencia inicial el último número de secuencia almacenado.

   `- recv(buff_size)`: Esta función será la encargada de manejar *Stop & Wait* desde el lado del receptor tal como vimos en la versión simplificada mostrada en el video. El **primer segmento** que reciba esta función contendrá el largo en bytes del mensaje que va a recibir desde el emisor `(message_length)`. Su función `recv` debe retornar una vez el largo de los **datos** recibidos (sin headers) sea igual a `min(message_length, buff_size)`.
   >>>Importante: Note que buff_size afecta solo el funcionamiento del socketTCP y no debería ser el mismo que el buffer del socket UDP utilizado internamente. El socket UDP debería tener un tamaño de buffer fijo.

   Si `message_length > buff_size` debe ser posible llamar a la función `recv(buff_size)` las veces que sea necesario para recibir el resto del mensaje, **una forma lograr esto** es definir dentro de su constructor una variable que le permita saber la cantidad de datos que aún no se han recibido. Si al llamar a `recv` dicha variable es 0 se asume que `recv` está recién comenzando a recibir datos y por lo tanto espera recibir el `message_length,` en caso contrario se continúa recibiendo el mensaje *sin esperar un message_length*. **Hay muchas otras formas de lograr esto, elija la que más le acomode**.

   Note que es posible que al ocurrir el evento `len(message_received) > buff_size` se encuentre con que `len(message_received) - buff_size > 0`. En dicho caso usted deberá elegir cómo manejar esto, lo importante es que su función retorne a lo más `buff_size` bytes y que no se pierdan datos.

   IMPORTANTE: No olvide que cada parte del mensaje `message` deberá ser encapsulado usando headers TCP (use las funciones implementadas en el punto 4). **El segmento completo (headers + datos) deberá ser enviado dentro del área de datos del datagrama UDP,** por lo que deberá considerar un tamaño de buffer para su socket UDP tal que quepa su sección de headers y su sección de datos.

   Test: Pruebe los códigos mostrados en el apartado ***Test send y recv*** de la sección ***Material e indicaciones para la actividad***. Por ahora haga la prueba sin pérdida de datos.

6. Implemente el *fin de conexión* para liberar recursos en emisor y receptor. Para ello añada las funciones `close()` y `recv_close()` a la clase `socketTCP`. La función `close()` debe implementar el cierre de conexión desde el lado del "Host A" según lo visto en el video. La función `recv_close()` **será la encargada de manejar el fin de conexión**, es decir, esta se deberá encargar de continuar el cierre de conexión desde el lado del "Host B" según lo visto en el video.

   Test: Añada al final del código de pruebas la parte anterior las líneas necesarias para cerrar la conexión y vea que la conexión se cierra con éxito. Pruebe sin pérdidas

7. Modifique sus funciones de handshake (`connect` y `accept`) para que puedan **manejar pérdidas** utilizando *Stop & Wait*. Note que existe un caso borde en que se pierde el último ACK del handshake, **dibuje un diagrama de este caso y adjuntelo a su informe**. A partir de este diagrama determine una solución. Es posible que para solucionar este caso borde deba modificar otras funciones como `send` o `recv`.

   Test: Repita las pruebas del paso 6, esta vez induciendo pérdidas. Se recomienda crear pruebas nuevas enviando mensajes más largos. Para simular pérdidas utilice `netem`. En caso de no poder utilizar `netem`, puede inducir pérdidas "a mano". Note que es posible que las pérdidas causen problemas con el cierre de conexión, pues este aún no maneja pérdidas. Esto es esperable, **lo importante** es que hasta antes del cierre su código logre transmitir los mensajes de manera íntegra *a pesar* de las pérdidas.

8. Modifique su función de cierre para que tolere pérdidas. Para ello haga las siguientes modificaciones:
   `- close()` : Si su función `close()` no recibe los mensajes FIN y ACK esperados luego de un timeout, renueve el timeout y vuelva a enviar el mensaje FIN. Haga que `close()` pueda esperar hasta 3 timeouts. Si al cumplirse el tercer timeout no ha recibido los mensajes, simplemente asuma que la contraparte se cerró y cierre la conexión.

   En caso de recibir los mensajes FIN y ACK con éxito, reenvíe el último ACK tres veces. Espere un tiempo timeout entre cada envío.

   `- recv_close()`: Modifique su código para que, en caso de no recibir el último ACK de la secuencia de cierre, espere hasta 3 timeouts. Luego del tercer timeout simplemente asuma que la contraparte se cerró y cierre la conexión.

   Test: Repita las pruebas de la parte anterior y verifique que ahora el cierre de conexión siempre se logra desde el lado que llama a la función `close()`. En el caso de la contraparte, verifique que una vez `recv` recibe un mensaje FIN, este siempre logra cerrar la conexión.

Pruebas

Antes de comenzar con las pruebas no olvide **añadir a su informe los diagramas solicitados en los pasos 4 y 7**.

Dado que en esta actividad su código implementa manejo de pérdidas con *Stop & Wait* se espera que su clase `SocketTCP` en efecto sea capaz de transmitir información de origen a destino de manera íntegra, incluso si se induce pérdida.

Para probar que su código maneja bien las pérdidas de datos puede crear un cliente y un servidor simples usando objetos `SocketTCP`. Puede reutilizar los códigos de pruebas de los tests de la actividad, pero recuerde que su código cliente de entrega debe poder ejecutarse como se indicó al principio de la actividad:

```
% python3 cliente.py localhost 8000 < archivo.txt
```

Para las pruebas basta que su cliente le mande un archivo (de más de 16 bytes) al servidor y luego cierre la conexión. Recuerde que puede simular pérdidas usando `netem loss` o a mano como se indica en el ***paso 1***. Para sus pruebas añada un **modo debug** que indique si su código está manejando pérdida y cómo la va manejando. Usted puede determinar los detalles de su modo debug. **En caso de usar pérdidas a mano, indíquelo en su informe** y añada una forma simple de manejar las pérdidas (ej: puede poner variables globales que permitan manejar la tasa de pérdidas, no olvide que las perdidas pueden ocurrir en ambas direcciones).

Además pruebe que `recv(buff_size)` se comporta como se pide en el ***paso 5*** cuando el tamaño del buffer es tal que `buff_size < message_length`. Para probar esto le puede enviar un archivo de tamaño `2n` bytes y llamar 2 veces a `recv` con un `buff_size=n`. Verifique que esto funciona cuando `n` **no es un múltiplo de 16**.

IMPORTANTE: Si encuentra cualquier situación de borde no especificado en la actividad **indíquelo en su informe**. Recuerde indicar en su **informe** las **decisiones de diseño** tomadas, particularmente aquellas relacionadas al ***paso 5***.

Material e indicaciones para la actividad

- **Ejemplo de 'clases' en python:**

```
class Mascota:
    def __init__(self):
        # inicializamos las variables que definen una mascota
        # los datos que aun no sabemos se ponen como None
        self.especie = None
        self.peso = None
        self.tamanno = None
        self.buena_mascota = True

    @staticmethod
    def parse_mascota(pet_str):
        nueva_mascota = Mascota()
        pet_split = pet_str.split(" ")

        if len(pet_split) > 1:
            nueva_mascota.especie = pet_split[0]
            nueva_mascota.tamanno = pet_split[1]

        return nueva_mascota

    def set_from_str(self, pet_str):
        nueva_mascota = self.parse_mascota(pet_str)
        self.especie = nueva_mascota.especie
        self.peso = nueva_mascota.peso
        self.tamanno = nueva_mascota.tamanno

    def set_peso(self, peso):
        self.peso = peso

    def set_mala_mascota(self):
        print("No, no hay mascotas malas, me niego")

    def is_buena_mascota(self):
        return self.buena_mascota

    def is_chonky(self):
        if self.especie == "gato":
            if self.tamanno == "smol":
                if self.peso > 5:
                    return "está chonky"
                else:
                    return "no está chonky"
            else:
                if self.peso > 7:
                    return "está chonky"
                else:
                    return "no está chonky"
        else:
            return "la verdad es que ni idea, esto es un ejemplo chiquito"

# usamos la clase que recién creamos
mi_gata = Mascota()
mi_gata.set_from_str("gato smol")
mi_gata.set_peso(6)
print(mi_gata.is_chonky())
```

- **Ejemplo headers TCP:** Para esta actividad debe definir una estructura para sus headers. Una forma de definir sus headers (sin usar directamente bytes) es la siguiente.

  ```
  [SYN]|||[ACK]|||[FIN]|||[SEQ]|||[DATOS]
  ```

  Aquí los datos necesarios del *header* se encuentran definidos con strings divididos por una secuencia "|||". Las variables [SYN], [ACK] y [FIN] corresponden a flags cuyo valor es 0 en caso de no estar siendo utilizadas, y en 1 en caso de ser utilizadas. La variable [SEQ] corresponde al número de secuencia. De esta forma un mensaje SYN-ACK se podría ver como:

  ```
  SYN-ACK -> 1|||1|||0|||8|||
  ```

  Mientras que una secuencia de datos con su respectivo ACK podría verse como:

  ```
  Datos   -> 0|||0|||0|||98|||Mensaje de prueba
  ACK     -> 0|||1|||0|||115|||
  ```

- **Timeouts en sockets de python:** El módulo de sockets en python provee una función para establecer timeouts en las funciones *bloqueantes* de un socket (puede ver la documentación en este enlace). Por ejemplo, en el siguiente código, `server_socket` va a esperar por 5 segundos antes de lanzar un error por timeout. Note que una vez se cumple el timeout *se lanza un error* el cual, de no ser correctamente manejado, va a resultar en que su código se caiga.

  ```
  server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) # creamos un socket
  server_socket.settimeout(5) # le indicamos que espere a lo más 5 segundos en cualquier comando bloqueante (recv, recvfrom, accept, etc)
  message, address = s_socket.recvfrom(128) # esperamos a que nos envíen algo. Si en 5 segundo no recibimos algo se levanta un error
  ```

- **Test send y recv:** Código para probar funciones `send` y `recv` usando el cliente y el servidor. Recuerde que debe hacer import de su clase SocketTCP.

```
# SERVER
server_socketTCP = SocketTCP.SocketTCP()
server_socketTCP.bind(address)
connection_socketTCP, new_address = server_socketTCP.accept()

# test 1
buff_size = 16
full_message = connection_socketTCP.recv(buff_size)
print("Test 1 received:", full_message)
if full_message == "Mensje de len=16".encode(): print("Test 1: Passed")
else: print("Test 1: Failed")

# test 2
buff_size = 19
full_message = connection_socketTCP.recv(buff_size)
print("Test 2 received:", full_message)
if full_message == "Mensaje de largo 19".encode(): print("Test 2: Passed")
else: print("Test 2: Failed")

# test 3
buff_size = 14
message_part_1 = connection_socketTCP.recv(buff_size)
message_part_2 = connection_socketTCP.recv(buff_size)
print("Test 3 received:", message_part_1 + message_part_2)
if (message_part_1 + message_part_2) == "Mensaje de largo 19".encode(): print("Test 3: Passed")
else: print("Test 3: Failed")
```

```
# CLIENT
client_socketTCP = SocketTCP.SocketTCP()
client_socketTCP.connect(address)
# test 1
message = "Mensje de len=16".encode()
client_socketTCP.send(message)
# test 2
message = "Mensaje de largo 19".encode()
client_socketTCP.send(message)
# test 3
message = "Mensaje de largo 19".encode()
client_socketTCP.send(message)
```

Anterior   Siguiente