TP3: Capa de Transporte
Manejo de Conexiones usando Raw Sockets

Teoría de las Comunicaciones
Departamento de Computación
FCEN - UBA

28.05.2014

1. Introducción

En este trabajo práctico ejercitaremos las nociones del nivel de transporte estudiadas en la materia a través de la implementación y análisis de un protocolo sencillo. En primera instancia, mencionaremos la normativa del trabajo práctico, seguido por las tareas a desarrollar. A continuación, en la sección 4, presentaremos el protocolo dando su especificación completa y describiendo cómo está implementado.

2. Normativa

- Fecha de entrega: hasta el martes 17 de junio inclusive.
- El código junto con el informe deberá haber sido enviado por correo hasta esa fecha con el siguiente formato:
  to: tdc-doc at dc uba ar
  subject: debe tener el prefijo [tdc-conexiones]
  body: nombres de los integrantes y las respectivas direcciones de correo electrónico
  attachment: el código fuente desarrollado
- Se deberá entregar el informe impreso y abrochado con la lista de integrantes y sus correos electrónicos (los mismos que fueran enviados por mail).

3. Consignas

El trabajo consiste, esencialmente, en simular efectos típicos de una red (tales como pérdida de paquetes o latencia) y analizar el impacto de esto en la implementación de un protocolo de transporte, evaluando para ello la eficiencia y el desempeño del protocolo en el contexto de una red local. Siguiendo la línea de los trabajos prácticos anteriores, la primera etapa implementativa sienta las bases para la segunda etapa analítica, en donde deberemos hacer experimentos, graficar resultados y sacar conclusiones de lo observado.

3.1. Primera parte: implementación

Tomando como punto de partida el código suministrado por la cátedra (explicado en la sección 4.8), implementar las siguientes modificaciones al protocolo:

1. Introducir delay al momento de enviar los ACKs. Este valor puede ser constante a lo largo del ciclo de vida de una instancia del protocolo, pero no obstante debe poder ser fácilmente editable para poder realizar el análisis de la segunda consigna.

2. Definir una probabilidad p de pérdida de paquetes. Una forma posible de simularla es, al momento de enviar un ACK, decidir si éste será efectivamente enviado "tirando una moneda" con dicha probabilidad.

3.2. Segunda parte: experimentación y análisis

Para el protocolo original y para las versiones con ACKs demorados/perdidos, en el contexto de una red local (LAN), enviar archivos de distinto tamaño (e.g., 1 KB, 5 KB, 10 KB, 50 KB, 100 KB, etc.) y medir el tiempo total de transmisión de los mismos. Tomar la medida representativa como el promedio de un número N de experimentos. A partir de esto, analizar cómo impactan los efectos de red simulados en la performance del protocolo. Por ejemplo, graficar lo siguiente y sacar conclusiones:

- Throughput percibido (i.e., cantidad de bits por segundo) en función del tamaño de archivo.
- Throughput en función del delay en los ACKs (para un tamaño de archivo constante).
- Cantidad de retransmisiones en función del delay en los ACKs (para un tamaño de archivo constante).

Notar que esta lista no es exhaustiva. Se valorará especialmente la creatividad a la hora de presentar y analizar los resultados obtenidos.

4. PTC: especificación e implementación

PTC es un protocolo de transporte basado en TCP, aunque sumamente simplificado. Fue (y continúa siendo) desarrollado en el contexto de la materia con el objetivo de proveer un marco de trabajo sencillo y acotado para llevar a la práctica diversos conceptos entre los que se destacan el control de flujo y, en particular, la manipulación de la ventana deslizante, el establecimiento y liberación de conexiones y las transiciones entre estados.

4.1. Características básicas

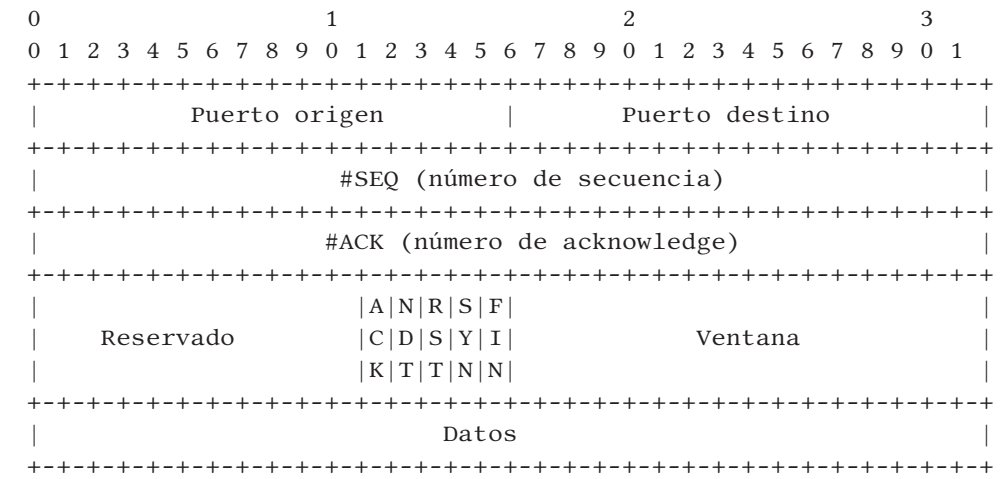
Desde un punto de vista general, **PTC** presenta las siguientes características:

- **Bidireccionalidad**: se trata de un protocolo full-duplex en el que las dos partes involucradas pueden enviar sus datos independientemente y en simultáneo.
- **Orientación a conexión**: contempla procesos formales de establecimiento y liberación de conexión. Esta conexión permite que los interlocutores generen estado para lograr luego una comunicación efectiva.
- **Confiabilidad**: a través de un algoritmo de ventana deslizante, garantiza que los datos enviados por cada interlocutor lleguen correctamente a destino.

En lo que sigue daremos un panorama más detallado de estas características, aunque en primer lugar mostraremos cómo es el formato de los paquetes.

4.2. Formato del paquete

El encabezado es de tamaño fijo: 16 bytes. Exhibe campos para representar los puertos de origen y destino, para secuenciar y reconocer los paquetes, para indicar el propósito de ellos (i.e., flags o bits de control) y para informar el tamaño actual de la ventana de recepción. La imagen que sigue muestra cómo se disponen estos campos:



A continuación damos una breve descripción de cada uno de ellos:

- `Puerto origen` (16 bits): el número de puerto utilizado por el emisor del paquete.
- `Puerto destino` (16 bits): el número de puerto utilizado por el receptor del paquete.
- #SEQ (32 bits): indica el primer byte de datos contenido en el paquete. En relación a esto, notar que los flags SYN y `FIN` (explicados más abajo) **deben** secuenciarse.
- #ACK (32 bits): contiene el valor del próximo byte del stream que está esperando recibir el emisor del paquete. Una vez establecida la conexión, este valor **debe** enviarse siempre.
- `Reservado` (11 bits): reservado para uso futuro; **debe** ser cero.
- Bits de control (5 bits):
  - ACK **debe** estar prendido siempre, e indica que el campo #ACK es significativo.
  - SYN sólo debería prenderse al iniciar una conexión o al responder a un paquete SYN recibido anteriormente. Se utiliza para sincronizar números de secuencia.
  - De igual manera, `FIN` debería prenderse cuando el emisor desea cerrar su stream de escritura. Observar que el receptor podrá seguir enviando información, con lo cual el emisor **debe** continuar procesando correctamente los datos de los paquetes entrantes.
  - Por el momento, los flags restantes (i.e., RST y NDT) no tienen uso.
- `Ventana` (16 bits): el número de bytes comenzando con el denotado por #ACK que el emisor puede aceptar actualmente.
- `Datos` (longitud variable): la información a enviar provista por los protocolos de nivel superior. El tamaño depende únicamente del valor actual de la ventana de recepción registrado por el emisor.

Así como un segmento TCP se encapsula dentro de un datagrama IP, los paquetes de **PTC** también viajarán dentro de IP. Para que los hosts puedan reconocer este hecho, el campo `proto` del header IP debe definirise con valor 202, tradicionalmente sin uso (el protocolo TCP se identifica con el valor 6).

4.3. Establecimiento y cierre de conexión

Antes de poder enviar sus datos, las instancias de **PTC** deben establecer una conexión. Para ello, se utiliza un algoritmo que sigue esencialmente los pasos del *three-way handshake* de TCP:

1. En primer lugar, la instancia activa (*cliente*) debe enviar un segmento con el flag SYN prendido y con un valor arbitrario en el campo #SEQ que indique el número de secuencia inicial que utilizará el cliente para identificar los bytes de su stream. Además, el campo `Ventana` debe contener el tamaño en bytes del buffer de recepción reservado por el cliente. El resto de los campos pueden tener un valor arbitrario; no serán tenidos en cuenta por su interlocutor.

2. El interlocutor (*servidor*) responderá con un paquete con flags SYN y ACK cuyo propósito es no sólo reconocer la intención del cliente sino además informar su número de secuencia inicial. El campo #ACK deberá ser el incremento del #SEQ recibido. Las observaciones sobre los demás campos hechas en el ítem anterior también son válidas en esta etapa.

3. Finalmente, el cliente reconocerá el SYN del servidor enviando un paquete ACK en el que el campo #ACK deberá ser, también, el incremento del #SEQ recibido.

Una vez hecho esto, ambas partes considerarán que la conexión está establecida.

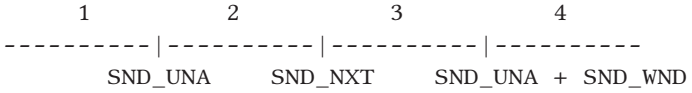
Para cerrar la conexión, la secuencia a seguir es también similar al comportamiento adoptado por TCP. Esto significa, además, que el cierre es asimétrico: cuando un **PTC** manifiesta a su interlocutor la intención de cerrar la conexión, no podrá enviar más datos útiles posteriormente (i.e., su stream de salida queda cerrado) pero sí podrá continuar recibiendo datos (i.e., su stream de entrada sigue abierto). Esto se materializa mediante el bit de control FIN. En la sección 4.6 explicaremos con mayor detalle los distintos escenarios de cierre posibles.

4.4. Ventana deslizante y bloque de control

El control de flujo en **PTC** es en su mayor parte análogo al de TCP (sin opciones). Utiliza un esquema de reconocimiento que usa ACKs acumulativos y provee una ventana de recepción de tamaño variable, usualmente ligada a la memoria disponible para el buffer de entrada. Todo lo referente a la ventana deslizante queda encapsulado en una estructura denominada *bloque de control*, que es instanciada por un **PTC** una vez hecho el contacto con su contraparte. El bloque mantiene el estado de las ventanas de emisión y recepción y controla los buffers de entrada y salida. En cuanto a la emisión, las variables que se utilizan son las siguientes:

- SND_UNA, que indica el número de secuencia más chico que aún no fue reconocido.
- SND_NXT, que representa el siguiente número de secuencia a ser utilizado en un paquete saliente.
- SND_WND, que contiene la máxima cantidad de bytes que pueden enviarse actualmente, a partir de lo informado por el interlocutor.
- SND_WL1, que indica el número de secuencia del interlocutor que fue utilizado para actualizar la ventana de emisión.
- SND_WL2, que, de manera similar, indica el número de ACK del segmento entrante que se usó para definir el tamaño de ventana actual.

El siguiente diagrama permite visualizar el espacio de secuenciamiento surgido a partir de lo anterior:



En él, las secciones numeradas pueden entenderse como sigue:

1. Números de secuencia que ya fueron reconocidos.

2. Números de secuencia utilizados en paquetes salientes que aún no fueron reconocidos (i.e., datos en vuelo).

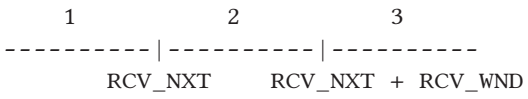
3. Números de secuencia disponibles para enviar nuevos datos.

4. Números de secuencia que podrán utilizarse en el futuro, conforme vayan llegando los reconocimientos.

La ventana de recepción se define a través de las siguientes variables:

- RCV_NXT, cuyo valor se corresponde con el próximo número de secuencia del interlocutor que **PTC** espera recibir.
- RCV_WND, que indica la cantidad de bytes que **PTC** está dispuesto a recibir actualmente.

En forma análoga, el espacio de secuenciamiento puede esquematizarse así:



En este caso, la porción 1 contiene los números de secuencia del interlocutor que ya fueron recibidos y reconocidos, mientras que en 2 se encuentran los números de secuencia que **PTC** está dispuesto a aceptar. Finalmente, en 3 están los números de secuencia aceptables en el futuro. Éstos irán pasando a la porción 2 a medida que vayan llegando los datos y RCV_NXT se mueva hacia la derecha.

4.5. Retransmisiones

Al enviar un segmento con datos, **PTC** también encolará este segmento en la cola de retransmisión. Éste permanecerá allí hasta ser eventualmente reconocido. Por otro lado, el cliente también define un tiempo máximo de espera RETRANSMISSION_TIMEOUT para esperar por estos reconocimientos. De superarse este tiempo, se asumirá que el paquete se extravió en los rincones de la red y por ende será retransmitido.

Se define asimismo un número máximo admisible de retranmisiones, MAX_RETRANSMISSION_ATTEMPTS. Si algún segmento debiera ser retransmitido más veces que esta cantidad, se debe asumir que la conexión se perdió y se pasará a cerrarla sin enviar FIN, liberando directamente todos los recursos reservados por la conexión.

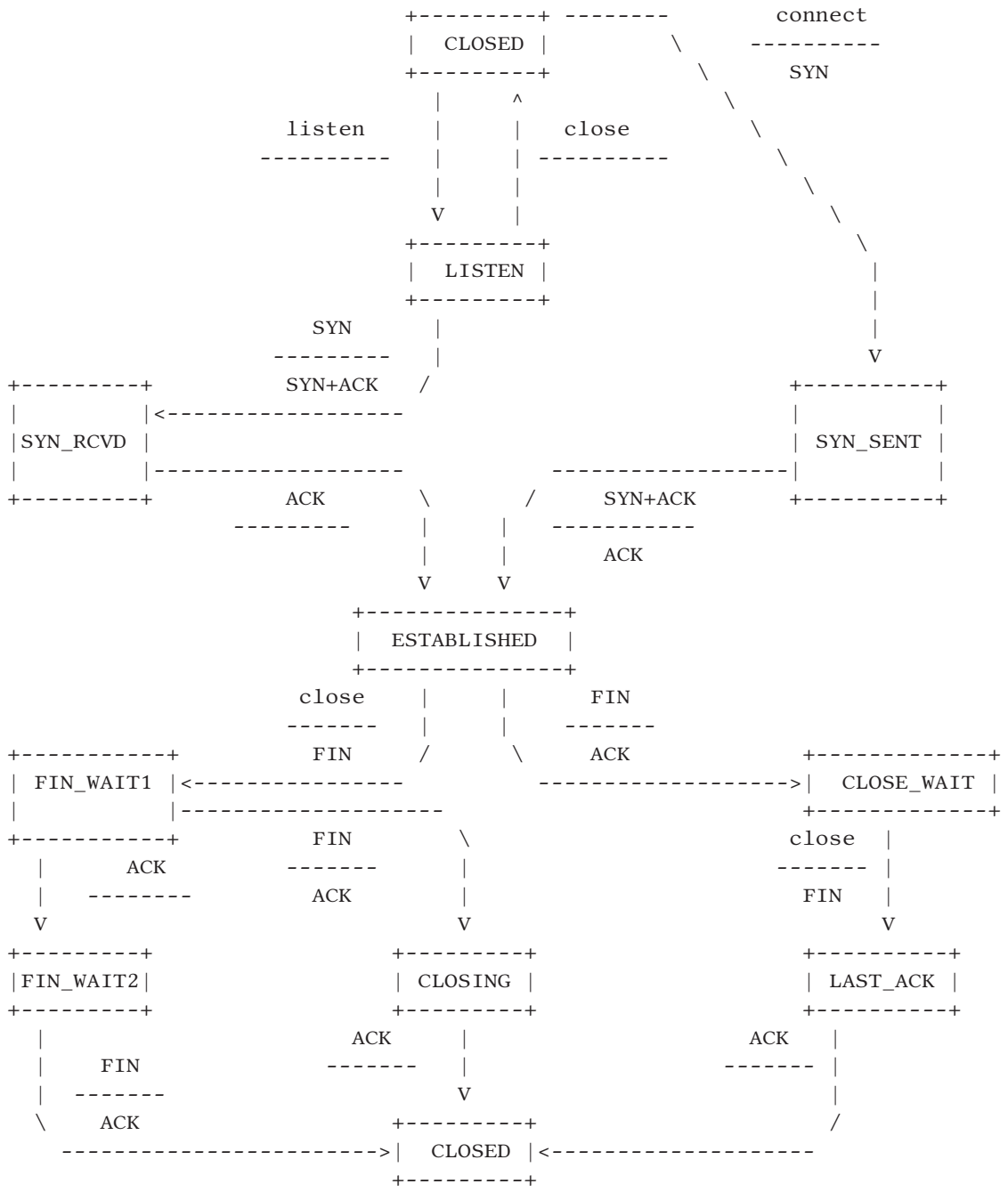
4.6. Estados

Cada **PTC** puede atravesar una serie de estados durante el ciclo de vida de una conexión:

- LISTEN, que representa la espera de una conexión entrante por parte de un **PTC** remoto.
- SYN_SENT, en el que el **PTC** está a la espera de una respuesta a la solicitud de conexión previamente enviada.
- SYN_RCVD, donde el **PTC** ya recibió una solicitud de conexión y respondió afirmativamente a ella, y está aguardado a que este último mensaje sea confirmado.
- ESTABLISHED, que indica una conexión abierta y activa. Es el estado normal de intercambio de datos.
- FIN_WAIT1, en donde el **PTC** inició el cierre de su stream de salida y está aguardado la confirmación de su interlocutor.
- FIN_WAIT2, en donde el **PTC** ya cerró exitosamente su stream de salida y queda a la espera de que su interlocutor cierre el suyo.
- CLOSE_WAIT, que indica que el interlocutor ya cerró su stream de salida y el **PTC** está esperando que el usuario decida cerrar la conexión.
- CLOSING, que representa un cierre simultáneo de ambas partes.
- LAST_ACK, en el que el **PTC** sólo espera a recibir la confirmación del cierre de su stream de salida (su interlocutor ya cerró previamente el suyo).

- CLOSED, que es la ausencia de conexión.

Las transiciones entre dichos estados se disparan por medio de tres tipos de eventos: acciones del usuario (`close`/`shutdown`, `listen` y `connect`), arribo de segmentos y exceso de retransmisiones. El diagrama que se muestra abajo resume los cambios de estado posibles junto con los respectivos eventos y las acciones tomadas por el protocolo en respuesta a ellos.



A los efectos de mantener la simplicidad del diagrama, el paso a CLOSED luego de exceder el máximo de retransmisiones fue omitido. Observar que esta transición podría originarse no sólo en ESTABLISHED sino también en cualquier otro estado sincronizado que involucre el envío de un paquete de datos o un paquete FIN.

4.7. Procesamiento de paquetes

Al recibir un paquete, éste se procesará de una u otra manera en función del estado transitado por **PTC** . En lo que sigue, notaremos con SEG_SEQ, SEG_ACK y SEG_LEN el número de secuencia del paquete entrante, su número de ACK y el tamaño de sus datos respectivamente. Notar además que todo paquete cuyo flag de ACK esté apagado **debe** ser automáticamente descartado sin importar en qué estado esté el protocolo (a excepción de LISTEN).

LISTEN Sólo se debe aceptar un paquete SYN y se debe inicializar el bloque de control a partir de la información provista por dicho paquete y por un número de secuencia inicial (ISS) computado aleatoriamente. Luego de cambiar el estado, se deberá enviar un paquete SYN/ACK en respuesta y por último incrementar SND_NXT.

SYN_SENT Sólo se debe aceptar un paquete con los flags de SYN y ACK prendidos. Además, SEG_ACK debe ser el valor del número de secuencia previamente enviado más uno. En tal caso, se deberá inicializar el bloque de control, pasar a ESTABLISHED y enviar el reconocimiento respectivo.

SYN_RCVD Si SEG_ACK es aceptable (i.e., su valor es uno más que el número de secuencia enviado), se debe pasar a ESTABLISHED e incrementar SND_UNA (recordar que el flag SYN también se secuencia).

ESTABLISHED Si el flag de FIN está prendido, deberá validarse que SEG_SEQ coincida con RCV_NXT, en cuyo caso se deberá pasar al estado CLOSE_WAIT, incrementar RCV_NXT (dado que el FIN se secuencia) y enviar un reconocimiento adecuado. Si SEG_SEQ tuviese un valor inesperado, simplemente se deberá enviar un ACK informando el valor actual de RCV_NXT.

En otro caso (i.e., el paquete no es FIN), se debe validar en el bloque de control la aceptación del paquete. SEG_ACK es aceptable si su valor está dentro de los valores de secuenciamiento esperados:

$$\text{SND\_UNA} < \text{SEG\_ACK} \leq \text{SND\_NXT}$$

En tal caso, el nuevo valor de SND_UNA será precisamente SEG_ACK. Por otro lado, los datos serán aceptables sólo si los números de secuencia contenidos en el paquete tienen intersección no nula con la ventana de recepción:

$$\text{RCV\_NXT} \leq \text{SEG\_SEQ} < \text{RCV\_NXT} + \text{RCV\_WND} \quad \text{o bien}$$
$$\text{RCV\_NXT} \leq \text{SEG\_SEQ} + \text{SEG\_LEN} - 1 < \text{RCV\_NXT} + \text{RCV\_WND}$$

De ser así, se deberá guardar en el buffer de entrada la porción de datos que esté contenida en la ventana de recepción. Si esta porción comienza en RCV_NXT, entonces se debe actualizar este valor sumándole la longitud de los datos aceptados. Además de esto, se debe actualizar el valor de RCV_WND decrementándola para reflejar el hecho de que el buffer contiene más información.

El bloque de control también debe analizar si corresponde actualizar el valor de SND_WND con este paquete. Esto se hará en caso de que el paquete esté reconociendo números de secuencia esperados:

$$\text{SND\_UNA} \leq \text{SEG\_ACK} \leq \text{SND\_NXT}$$

Notar que en esta ocasión es necesario el $\leq$ de la izquierda para procesar correctamente potenciales actualizaciones de ventana con valores de ACK repetidos.

Además de SND_WND, también se requiere actualizar SND_WL1 y SND_WL2 sólo si se trata de un segmento "más nuevo":

$$\text{SND\_WL1} < \text{SEG\_SEQ} \quad \text{o bien}$$
$$\text{SND\_WL1} = \text{SEG\_SEQ} \text{ y } \text{SND\_WL2} \leq \text{SEG\_ACK}$$

En este escenario, se hará efectiva la actualización de las variables: SND_WND tomará el valor de la ventana informada en el paquete, SND_WL1 tomará el valor de SEG_SEQ y SND_WL2, el de SEG_ACK.

Finalmente, si el paquete contiene datos, se debe enviar un ACK independientemente de si fue o no aceptado. Para ello, el protocolo podría hacer piggybacking si es que hubiera datos en el buffer de salida aguardando a ser enviados. Caso contrario, un ACK ad-hoc deberá enviarse.

FIN_WAIT1 Si el bloque de control acepta SEG_ACK, significa que este paquete reconoce el FIN enviado. Luego, se debe pasar a FIN_WAIT2. Si además viniese un FIN, éste se deberá procesar tal como se describió para el caso de ESTABLISHED, pero pasando al estado CLOSED.

Si SEG_ACK no fuese aceptado y el paquete fuese un FIN, se deberá hacer lo mismo sólo que el próximo estado debe ser CLOSING (dado que se trata de un cierre simultáneo).

En cualquier caso, se deberá validar el paquete en el bloque de control y eventualmente enviar un ACK tal como se explicó más arriba. Esto es porque el paquete podría contener datos válidos. Observar que en este caso no podrá hacerse piggybacking dado que ya se envió el FIN.

FIN_WAIT2 Si el paquete es FIN, se deberá proceder tal como en ESTABLISHED pero, a diferencia, el siguiente estado deberá ser CLOSED. De lo contrario, se deberá procesar el paquete en el bloque de control y enviar un ACK si el paquete tuviera datos (notar que esto se corresponde con el último párrafo de lo explicado para FIN_WAIT1).

CLOSING En este caso, sólo resta esperar que SEG_ACK sea aceptado, lo cual sólo puede significar que el FIN fue reconocido y que por ende el protocolo debe pasar al estado CLOSED.

CLOSE_WAIT Dado que el interlocutor cerró su stream de datos, sólo deben esperarse reconocimientos en este estado. Por ello, se deberá procesar el paquete en el bloque de control para ajustar las variables de la ventana de emisión.

LAST_ACK El procesamiento en este caso es análogo al de CLOSING.

4.8. Código fuente

La implementación de **PTC** está hecha en Python 2.7 y viene acompañada de un plugin para Wireshark que permite visualizar los paquetes capturados. En las secciones que siguen veremos la estructura de módulos a alto nivel, mostraremos algunos ejemplos de uso de sockets **PTC** y mencionaremos cómo correr los casos de prueba.

4.8.1. Módulos

Los módulos que implementan el protocolo están en el directorio `ptc`. A continuación listamos cada uno de ellos.

buffer Implementación de un buffer de bytes (`DataBuffer`) que es usado por el bloque de control para definir los buffers de entrada y de salida. Este buffer ofrece funcionalidad para reflejar el hecho de que los datos pueden llegar potencialmente fuera de orden. Por ejemplo, el método `add_chunk` recibe un offset dentro del buffer y los bytes a agregar a partir de dicho offset. El siguiente fragmento ilustra un posible uso del mismo:

```
>>> buffer = DataBuffer()
>>> buffer.add_chunk(8, 'baz')
>>> buffer.add_chunk(4, 'bar ')
>>> buffer.put('foo ')
>>> buffer.get(15)
'foo bar baz'
>>> buffer.empty()
True
```

Notar que `get` extrae los datos del buffer. El argumento que recibe indica la máxima cantidad de bytes a extraer, aunque potencialmente podrían ser menos.

cblock Implementación del bloque de control, `PTCControlBlock`. Todo lo referente a los buffers de datos y la manipulación de las ventanas de emisión y recepción está en esta clase.

constants Definición de diversas constantes utilizadas por el protocolo, entre las que se encuentran, por ejemplo, los estados.

exceptions Definición de una excepción genérica (`PTCError`) para representar errores del protocolo o de uso inválido del mismo. El constructor recibe un string como argumento que permite indicar con mayor detalle qué fue lo que realmente ocurrió.

handler Implementación del handler de paquetes entrantes, `IncomingPacketHandler`. El método principal, `handle`, recibe un paquete que acaba de ser recibido y, en función del estado del protocolo, termina derivando en otro método específico para tal estado.

packet Implementación del paquete **PTC** , `PTCPacket`. Esta clase brinda una interfaz que permite definir el valor de cada campo del segmento y también de las direcciones IP involucradas:

```
>>> packet = PTCPacket()
>>> packet.set_source_ip('192.168.0.1')
>>> packet.set_destination_ip('192.168.0.100')
>>> packet.set_source_port(12345)
>>> packet.set_destination_port(80)
>>> packet.set_seq_number(8989)
>>> packet.add_flag(SYNFlag)
>>> packet.set_payload('hola!')
>>> packet.set_window_size(1024)
>>> packet
From: (192.168.0.1, 12345)
To: (192.168.0.100, 80)
Seq: 8989
Ack: 0
Flags: SYN
Window: 1024
Payload: hola!
>>> packet.get_ack_number()
0
>>> packet.get_seq_number()
8989
>>> # Con el siguiente método podemos conocer el intervalo de números
>>> # de secuencia consumidos por el paquete (#SEQ + |payload|).
>>> packet.get_seq_interval()
(8989, 8994)
```

Para averiguar si un flag determinado está presente en un paquete, puede utilizarse el operador `in`:

```
>>> ACKFlag in packet
False
>>> SYNFlag in packet
True
>>> FINFlag in packet
False
```

packet_utils Herramientas para facilitar la manipulación de paquetes: un decodificador de bytes (`PacketDecoder`), que es utilizado para mapear los datos recibidos de la red a un `PTCPacket`, y un constructor de paquetes (`PacketBuilder`), que simplemente recibe argumentos (flags, número de secuencia, número de reconocimiento, ventana, etc.) y arma un paquete con tales características.

La clase que implementa el protocolo, `PTCProtocol` (mencionada más abajo) ofrece un método de conveniencia que se apoya en el `PacketBuilder` para armar paquetes con la información actual del bloque de control:

```
def build_packet(self, seq=None, ack=None, payload=None, flags=None,
                 window=None):
    if seq is None:
        seq = self.control_block.get_snd_nxt()
    if flags is None:
        flags = [ACKFlag]
    if ack is None and ACKFlag in flags:
        ack = self.control_block.get_rcv_nxt()
    if window is None:
        window = self.control_block.get_rcv_wnd()
    packet = self.packet_builder.build(payload=payload, flags=flags,
                                       seq=seq, ack=ack, window=window)
    return packet
```

De esta forma, al momento de enviar un ACK, sólo basta con invocar a este método sin argumentos para generar el paquete adecuado.

thread Implementación de los threads en los que se apoya el protocolo para su funcionamiento:

- Uno de ellos se encarga de simular el clock del sistema (`Clock`). Cada `CLOCK_TICK` segundos (definido por defecto en 0.1) invocará al método `tick` del protocolo.
- Otro tiene como objetivo monitorear el socket y recibir los paquetes (`PacketReceiver`). Al detectar la llegada de uno, se invocará el método `handle_incoming` del protocolo (que a su vez se apoyará en el handler mencionado más arriba).
- El último de ellos es el que envía los paquetes de datos y eventualmente el `FIN` (`PacketSender`). Este comportamiento queda definido en el método `handle_outgoing` del protocolo, que es ejecutado en el contexto de este thread cada vez que ocurre algún evento que podría motivar el envío de nuevos datos (e.g., llegada de reconocimientos o invocaciones a `send` por parte del usuario).

rqueue Implementación de la cola de retransmisión (`RetransmissionQueue`). Al encolarse, los paquetes se asocian con un timestamp que irá revisándose en cada tick del reloj (mediante el método `tick`, que de hecho es invocado por el método homónimo del protocolo). Cada vez que expira un timeout, el paquete respectivo se mueve a una lista interna de paquetes a retransmitir que luego es consumida por el protocolo. Por otra parte, al procesar un ACK, el método `remove_acknowledged_by` permite extraer de la cola todo paquete cuyo payload quede completamente cubierto por el #ACK contenido en el paquete.

seqnum Implementación de números de secuencia (`SequenceNumber`). Son utilizados dentro de los paquetes y dentro del bloque de control para representar las variables de la ventana deslizante ligadas a números de secuencia (como SND_UNA o SND_NXT). Internamente trabaja con aritmética modular y sobrecarga los operadores aritméticos tradicionales de manera de poder utilizarlos en contextos donde se esperen enteros standard:

```
>>> n = SequenceNumber(5, modulus=10)
>>> m = SequenceNumber(9, modulus=10)
>>> n + m
```

```
4
>>> n + 16
1
>>> n > 1
True
>>> n > 6
False
```

La clase `SequenceNumber` también provee una serie de métodos de clase que permiten hacer comparaciones en rango teniendo en cuenta que, eventualmente, puede haber *wrap-around*. Por ejemplo, si el límite superior de la ventana de emisión supera el máximo valor permitido, éste arrancará desde 0. Por ende, las comparaciones efectuadas en el protocolo deben tolerar tales escenarios. A raíz de esto, el método `a_leq_b_leq_c` recibe tres valores `a`, `b` y `c` y computa el valor de verdad de $a \leq b \leq c$:

```
>>> eight = SequenceNumber(8, modulus=10)
>>> five = SequenceNumber(5, modulus=10)
>>> three = SequenceNumber(3, modulus=10)
>>> # Caso normal
>>> SequenceNumber.a_leq_b_leq_c(three, five, eight)
True
>>> # Caso en el que el c superó el máximo
>>> SequenceNumber.a_leq_b_leq_c(five, eight, three)
True
>>> # Caso en el que b y c superaron el máximo, b < c
>>> SequenceNumber.a_leq_b_leq_c(eight, three, five)
True
>>> # b y c superaron el máximo pero b > c
>>> SequenceNumber.a_leq_b_leq_c(eight, five, three)
False
>>> # b > max(a, c)
>>> SequenceNumber.a_leq_b_leq_c(three, eight, five)
False
>>> # b < min(a, c)
>>> SequenceNumber.a_leq_b_leq_c(five, three, eight)
False
```

`ptc_socket` Provee un wrapper sobre el protocolo (`Socket`) para definir una interfaz de uso similar a la de los sockets de Python tradicionales. El usuario final del protocolo interactuará directamente con instancias de `Socket`, tal como se ve en la sección 4.8.2.

`soquete` Provee una abstracción del socket raw subyacente al protocolo (`Soquete`). Esto permite desligarse de la declaración y uso del mismo, pudiendo así evitar la manipulación explícita de los bytes dentro del código del protocolo.

`protocol` Implementación del núcleo del protocolo (`PTCProtocol`). Además de los handlers invocados por los threads, también manipula la cola de retransmisión y ofrece métodos que implementan el comportamiento del `Socket` mencionado más arriba. El protocolo mantiene una instancia de `Soquete` en la variable `socket`. A través de ella es posible inyectar paquetes en la red invocando al método `send` y pasando como argumento el paquete que deseamos enviar a destino. Por otro lado, el protocolo provee el método `send_and_queue` en caso de querer no sólo enviar el paquete sino además encolarlo en la cola de retransmisión.

4.8.2. Modo de uso

La forma de usar el protocolo desde otras aplicaciones es a través del wrapper `Socket` nombrado en la sección anterior. Al implementar los métodos más comunes de los sockets tradicionales, la experiencia de uso será esencialmente análoga. No obstante, todo programa que use sockets **PTC debe** ejecutarse con permisos elevados al requerir instanciaciones de sockets raw. Además, a los efectos de asegurar un correcto cierre del programa, sugerimos utilizar los sockets dentro de bloques `with`. De no hacer esto, será necesario llamar manualmente a `close` o a `free` para que el protocolo pueda finalizar la conexión y apagar los threads. En el caso donde esta llamada no se haga, el programa podría quedar "colgado" a causa de tales threads.

A continuación mostramos un breve ejemplo en el que un script liga un socket **PTC** al puerto 6677 de localhost y otro script diferente declara y conecta un nuevo socket a dicho puerto. Ambos se envían datos mutuamente y por último muestran por la salida standard la información recibida:

```
# Script que liga un socket al puerto 6677 de localhost.
from ptc import Socket
to_send = 'Lorem ipsum dolor sit amet'
received = str()
with Socket() as sock1:
    sock1.bind(('127.0.0.1', 6677))
    sock1.listen()
    sock1.accept()
    received += sock1.recv(15)
    sock1.send(to_send)
    sock1.close()
print 'sock1 received: %s' % received
```

```
# Script que abre un socket y lo conecta al puerto 6677 de localhost.
from ptc import Socket, SHUT_WR
to_send = 'foo bar baz'
received = str()
with Socket() as sock2:
    sock2.connect(('127.0.0.1', 6677))
    sock2.send(to_send)
    received += sock2.recv(10)
    # Cerramos el stream de escritura pero podemos seguir recibiendo datos.
    sock2.shutdown(SHUT_WR)
    received += sock2.recv(20)
print 'sock2 received: %s' % received
```

Al finalizar, se debería visualizar `sock1 received: foo bar baz` en la salida del primer script y `sock2 received: Lorem ipsum dolor sit amet` en la salida del segundo.

4.8.3. Tests

La implementación de **PTC** provee un conjunto de casos de prueba con la particularidad de que éstos corren íntegramente en memoria. Además de optimizar el tiempo de ejecución y evitar depender de la red, esto también permite desarrollar sin necesidad de instanciar un socket raw, de lo que se desprende que deja de ser requisito tener permisos elevados para ejecutar el código.

La forma de correr los tests es ejecutando el script `run_tests.py` con el intérprete de Python 2.7. Al no recibir argumentos adicionales, el script extraerá todos los tests que encuentre en el directorio `test` y correrá uno tras uno hasta finalizar, mostrando los resultados por la salida standard. Es posible, no obstante, ejecutar clases de test individuales. Esto se logra pasando un argumento de la forma `--<identificador>`, en donde los valores de `<identificador>` se corresponden con los nombres de los archivos de test. La siguiente lista muestra los identificadores disponibles:

- `syn`: tests de inicio de conexión.
- `fin`: tests de finalización de conexión.
- `ack`: tests que involucran distintos escenarios de reconocimiento.
- `data`: tests de intercambio de datos.
- `control-block`: tests del bloque de control.
- `retransmission`: tests del mecanismo de retransmisión en el protocolo.
- `rqueue`: tests de la estructura de datos que implementa la cola de retransmisión.
- `buffer`: tests de la estructura de datos que implementa el buffer.
- `seqnum`: tests de la clase que implementa los números de secuencia.
- `packet`: tests de paquetes (mapeo de bytes a paquete y viceversa).

A modo de ejemplo, la siguiente línea puede usarse para correr los tests de inicio y finalización de conexión:

```
$ python run_tests.py --syn --fin
```

4.9. Disector de Wireshark

El directorio `dissector` contiene un plugin de Wireshark para interpretar los paquetes del protocolo. Por medio de éste, podremos observar qué paquetes envían y reciben nuestros sockets **PTC** y qué contiene cada uno de sus campos. El modo de uso es simple: basta con usar `ptc` como filtro para que Wireshark nos muestre sólo los paquetes capturados que pertenezcan a nuestro protocolo.

El disector está compilado para arquitecturas Intel de 32 ó 64 bits; se deberá escoger la versión correspondiente según la máquina donde se vaya a utilizar el protocolo. Para instalarlo, es necesario copiar el archivo al directorio de plugins globales de Wireshark, que puede conocerse accediendo al menú `Help` → `About Wireshark` → `Folders`. Tener en cuenta que el plugin es compatible con versiones de Wireshark posteriores a la 1.6, por lo que es muy posible que no funcione en versiones previas. Por este motivo, sugerimos revisar la versión a utilizar y hacer la actualización si fuese necesario.

Referencias

[1] RFC 793: Transmission Control Protocol

[2] RFC 1122: Requirements for Internet Hosts -- Communication Layers

[3] W. Richard Stevens (1993). *TCP/IP Illustrated, Volume 1: The Protocols.* Addison-Wesley. `http://www.pcvr.nl/tcpip`

[4] Tanenbaum, A. *Computer Networks*, 3ra Ed. Capítulo 3: págs. 202-213.