Trabajo Práctico 3: Capa de Transporte

Programación de protocolos end-to-end

Teoría de las Comunicaciones

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

22.10.2014

Agenda

1. Introducción
   - Objetivos del TP
   - Background: sockets
2. PTC: especificación e implementación
   - Especificación
   - Implementación
3. Consignas

Agenda

Objetivos de este trabajo

- Poner en práctica las nociones estudiadas del nivel de transporte: conexiones, control de flujo, estados, etc.
- Tener contacto directo con el funcionamiento e implementación de protocolos de networking.
- Continuar profundizando el enfoque analítico de las instancias anteriores.

¿Qué es un socket?

- Como parte de la información contextual del trabajo, veremos con un poco más de detalle de qué tratan los **sockets**.
- Un socket es en esencia un canal de comunicación entre procesos.
- Los sockets entre procesos remotos se llaman **Internet sockets**.
- Hay otras variantes, como los sockets Unix: establecen un canal de comunicación entre procesos corriendo en el mismo SO.

Tipos de Internet sockets

- Existen varios tipos de sockets, siendo tal vez los más relevantes los siguientes:
  - **Stream sockets**,
  - **Datagram sockets**, y
  - **Raw sockets**.
- Los stream sockets se apoyan en TCP, mientras que los datagram sockets en UDP.
- De los raw sockets hablaremos en unos minutos!

Socket API

- Los sistemas operativos suelen ofrecer una interfaz para manipular sockets: conjunto de llamadas al sistema que abstraen al usuario de las problemáticas de networking que el SO resuelve.
- La API *standard* de hoy en día es básicamente la de los Berkeley sockets.
- Esta implementación de sockets apareció en los ’80 en 4.2BSD.
- Luego evolucionó (aunque no demasiado) para dar origen a los sockets POSIX.
- POSIX: conjunto de standards de la IEEE para mantener compatibilidad entre distintos SOs, por lo general basados en Unix.

Llamadas al sistema de la API de sockets

Algunas de las llamadas al sistema más importantes:

- `socket`: crea un nuevo socket de un tipo dado, identificado por un file descriptor.
- `bind`: típicamente usada por el servidor, liga el socket a una dirección local (i.e., IP más puerto).
- `listen`: usada por el servidor, lleva el socket (TCP) al estado de LISTEN. Recibe un argumento que indica la cantidad máxima de solicitudes a encolar.

Llamadas al sistema de la API de sockets (cont.)

- `connect`: usada por el cliente, inicia el proceso de conexión a una dirección dada. Asigna un puerto local libre al socket.
- `accept`: acepta un pedido remoto de conexión enviado por un cliente y devuelve un nuevo socket con la conexión establecida.
- `send` y `recv`: envían y reciben datos de un socket.
- `close`: finaliza la conexión (si el conteo de referencias del socket llegó a cero) y libera los recursos reservados en el SO para el socket.
- `shutdown`: permite hacer un cierre asimétrico de la conexión (i.e., cerrar el stream de lectura, el stream de escritura o ambos).

Ejemplo (rápido) en Python: conexión entre dos procesos

```
>>> import socket
>>> sock1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock1.bind(('127.0.0.1', 12345))
>>> sock1.listen(1)
>>> sock, dst_address = sock1.accept()
```

```
>>> import socket
>>> sock2 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
>>> sock2.connect(('127.0.0.1', 12345))
```

Ejemplo (rápido) en Python: intercambio de datos

```
...
>>> sock2.send('hola' * 10)
40
```

```
...
>>> sock.recv(10)
'holaholaho'
>>> sock.recv(10)
'laholahola'
>>> sock.recv(100)
'holaholaholaholahola'
```

Sockets raw

- Los **sockets raw** conforman otra variante de sockets que posee capacidades poderosas no presentes en los otros tipos mencionados.
- Con ellos, un proceso puede leer y escribir datagramas IP con un número de protocolo no procesado por el kernel del SO.
  - Este campo de 8 bits indica qué protocolo viaja dentro de IP (e.g., 6 representa a TCP).
  - Como corolario, permite diseñar e implementar protocolos de transporte ad-hoc a nivel de usuario.
- También, con un socket raw, un proceso tiene la posibilidad de construir él mismo el datagrama IP (seteando la opción `IP_HDRINCL`).
- En este trabajo práctico utilizaremos sockets raw para enviar y recibir los segmentos de nuestro protocolo.

Sockets raw: cómo funcionan

- La creación de un socket raw requiere permisos elevados: esto prohibe a usuarios normales la inyección de datagramas IP arbitrarios en la red.
- Al crearse, un socket raw define con qué número de protocolo va a trabajar:

```
sock = socket.socket(socket.AF_INET, socket.SOCK_RAW, protocol)
```

- Cuando se define la opción `IP_HDRINCL`, el output de datos en el socket se espera que contenga el header IP seguido del payload arbitrario.
  - No obstante, el kernel calcula el checksum de IP.
- Observar que no es necesario invocar a `connect` con un socket raw.

Sockets raw: qué hace el kernel al recibir datagramas

- Al recibir datagramas IP, el kernel **no** reenvía segmentos UDP ni TCP a los sockets raw.
- En cambio, reenvía los paquetes IP con número de protocolo desconocido.
  - Sólo realiza checkeos básicos sobre estos paquetes (checksum, dirección destino, versión de IP, etc.)
- En este proceso, el kernel examinará **todos** los sockets raw de **todos** los procesos.
- Una copia del datagrama será entregada a cada socket que satisfaga una serie de tests que involucran el checkeo del protocolo y de las direcciones especificadas.

Agenda

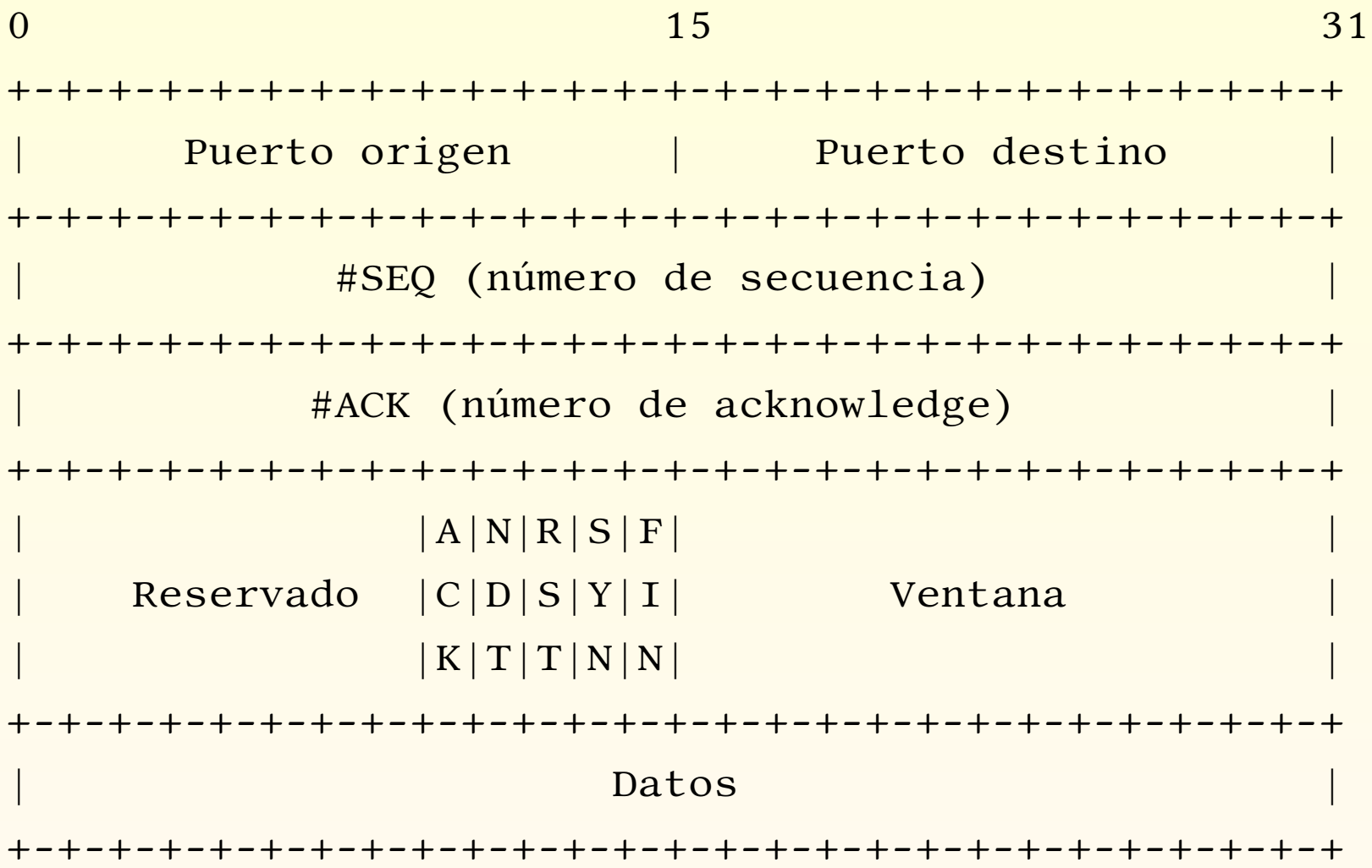
¿Qué es **PTC**?

- **PTC** es un protocolo de transporte basado en TCP, aunque sumamente simplificado.
- Desarrollado en el contexto de la materia con el objetivo de llevar a la práctica los conceptos inherentes al nivel de transporte.
  - Control de flujo: ventana deslizante.
  - Establecimiento y cierre de conexiones.
  - Transiciones entre estados.
  - Estimación dinámica de RTO.
  - Próximamente más (e.g., delayed ACKs, opciones, control de congestión, etc.).

Características básicas

- **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.
- **Confiabilidad**: a través de un algoritmo de ventana deslizante, garantiza que los datos enviados por cada interlocutor lleguen correctamente a destino.

Formato del paquete



Algunos comentarios sobre el paquete

- #SEQ indica el primer byte de datos contenido en el paquete. Los flags SYN y FIN **deben** secuenciarse.
- #ACK contiene el valor del próximo byte del stream que está esperando recibir el emisor. Una vez establecida la conexión, este valor **debe** enviarse siempre.
- Ventana indica el número de bytes comenzando con el denotado por #ACK que el emisor puede aceptar actualmente.

Al igual que TCP, los paquetes de **PTC** también viajarán dentro de IP: el campo proto del header IP debe definirise con valor 202, tradicionalmente sin uso.

Establecimiento y cierre de conexión

- Para conectar, se usa esencialmente el *three-way handshake* de TCP, en su escenario básico: SYN $\rightarrow$ SYN/ACK $\rightarrow$ ACK.
- El cierre, al igual que TCP, puede ser asimétrico.
  - Un FIN indica la intención de cerrar el stream de escritura del emisor.
  - Luego de éste, puede seguir recibiendo datos.
  - El cierre del stream de escritura del interlocutor es independiente.

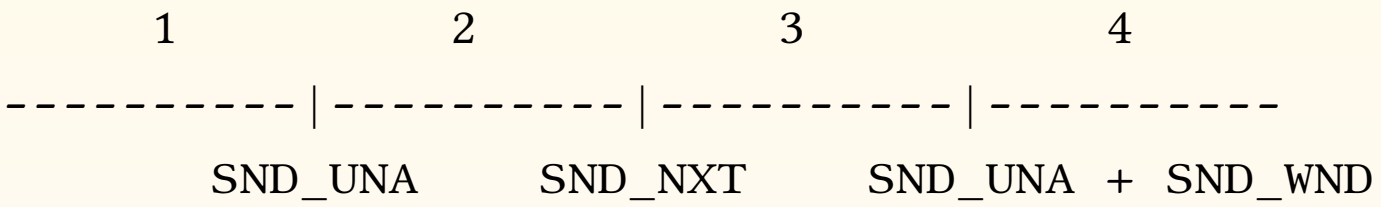
Ventana deslizante

- El control de flujo en **PTC** es en su mayor parte análogo al de TCP (sin opciones).
- ACKs acumulativos; ventana de recepción de tamaño variable (usualmente ligada a la memoria disponible para el buffer de entrada).
- Implementada en el *bloque de control*: estructura que mantiene el estado de las ventanas de emisión y recepción y controla los buffers de entrada y salida.

Bloque de control: variables de emisión

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

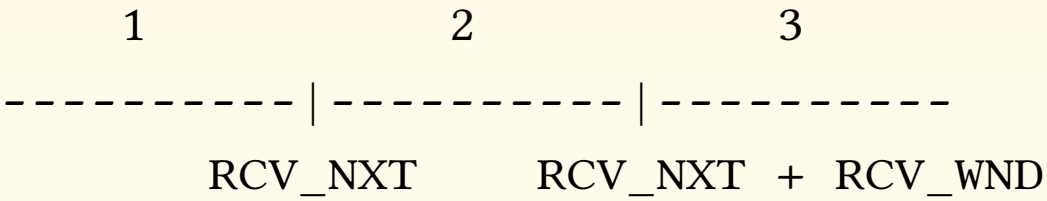
A partir de esto, el espacio de secuenciamiento puede visualizarse así:



Bloque de control: variables de recepción

- RCV_NXT: el próximo número de secuencia del interlocutor que **PTC** espera recibir.
- RCV_WND: cantidad de bytes que **PTC** está dispuesto a recibir actualmente.

En este caso, el espacio de secuenciamiento puede esquematizarse de la siguiente manera:



Retransmisiones y RTO

- Al enviar un segmento con datos, **PTC** también encolará este segmento en la cola de retransmisión.
- Éste permanecerá allí hasta ser eventualmente reconocido.
- El tiempo de retransmisión se calcula dinámicamente según el RFC 6298.
- Además, se define 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.

Retransmisiones y RTO (cont.)

Para cada RTT muestreado (usando el algoritmo de Karn):

$$\begin{aligned} \text{RTTVAR} &= (1-\beta)\,\text{RTTVAR} + \beta\,|\text{SRTT} - \text{RTT}| \\ \text{SRTT} &= (1-\alpha)\,\text{SRTT} + \alpha\,\text{RTT} \\ \text{RTO} &= \text{SRTT} + \text{máx}(1, K \cdot \text{RTTVAR}) \end{aligned}$$

- Por defecto, siguiendo el RFC, $\alpha = 1/8$, $\beta = 1/4$ y $K = 4$.
- Al tomar la primera muestra, SRTT = RTT y RTTVAR = RTT/2.
- Cada vez que expira el RTO,
  - Se hace back-off de este tiempo.
  - Se retransmite sólo el paquete que esté al principio de la cola.
- Cuando un ACK reconoce datos,
  - Si todos los datos en vuelo fueron reconocidos, se apaga el timer.
  - Caso contrario, se reinicia con el RTO actual.

Estados

- Los estados de **PTC** son los mismos de TCP, a excepción de `TIME_WAIT`.
  - `LISTEN`, `SYN_SENT`, `SYN_RCVD`, `ESTABLISHED`, `FIN_WAIT1`, `FIN_WAIT2`, `CLOSE_WAIT`, `CLOSING`, `LAST_ACK`, `CLOSED`.
- Las transiciones entre ellos se disparan por medio de tres tipos de eventos: acciones del usuario (`close`/`shutdown`, `listen` y `connect`), arribo de segmentos y exceso de retransmisiones.

Diagrama de estados: establecimiento de conexión

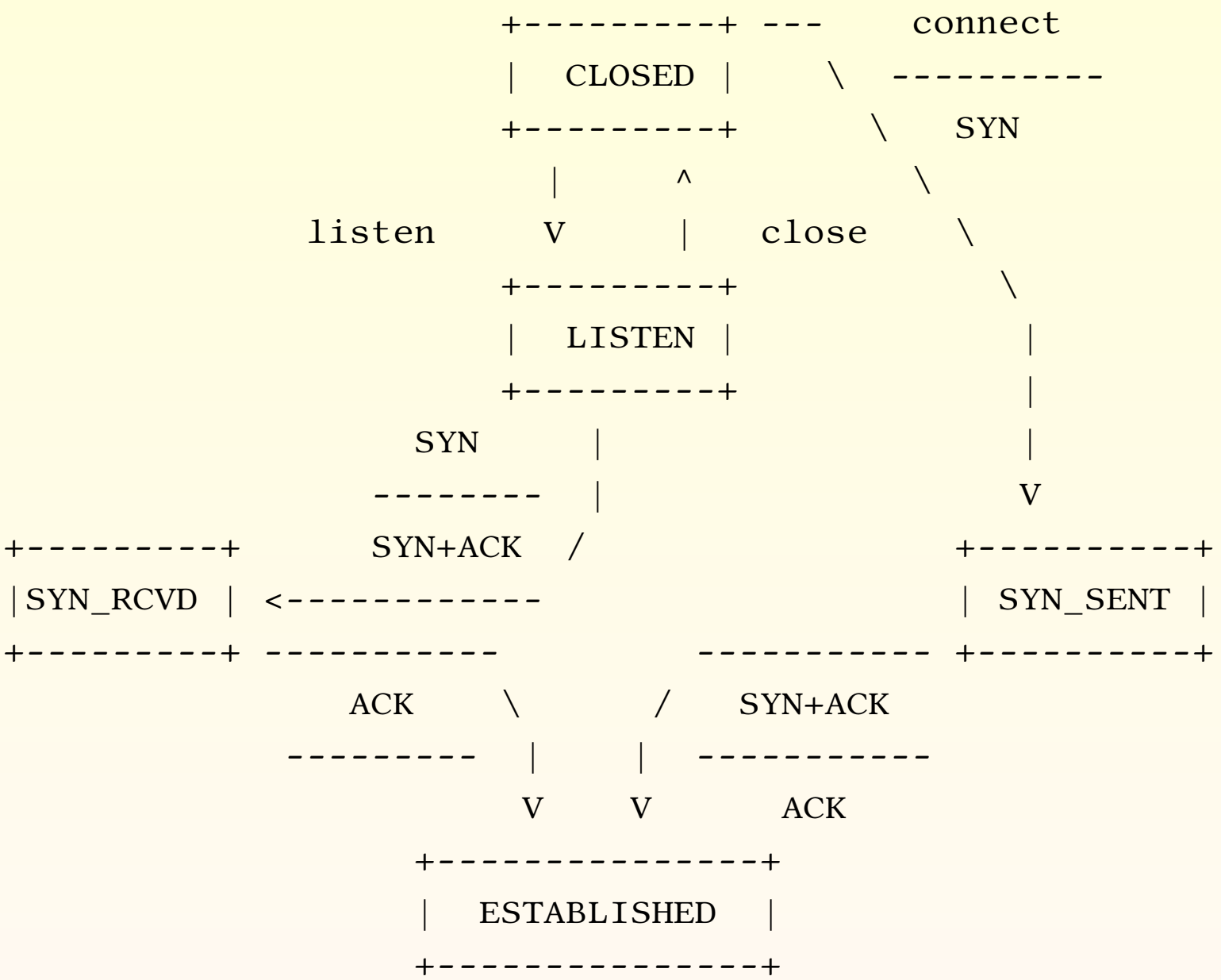
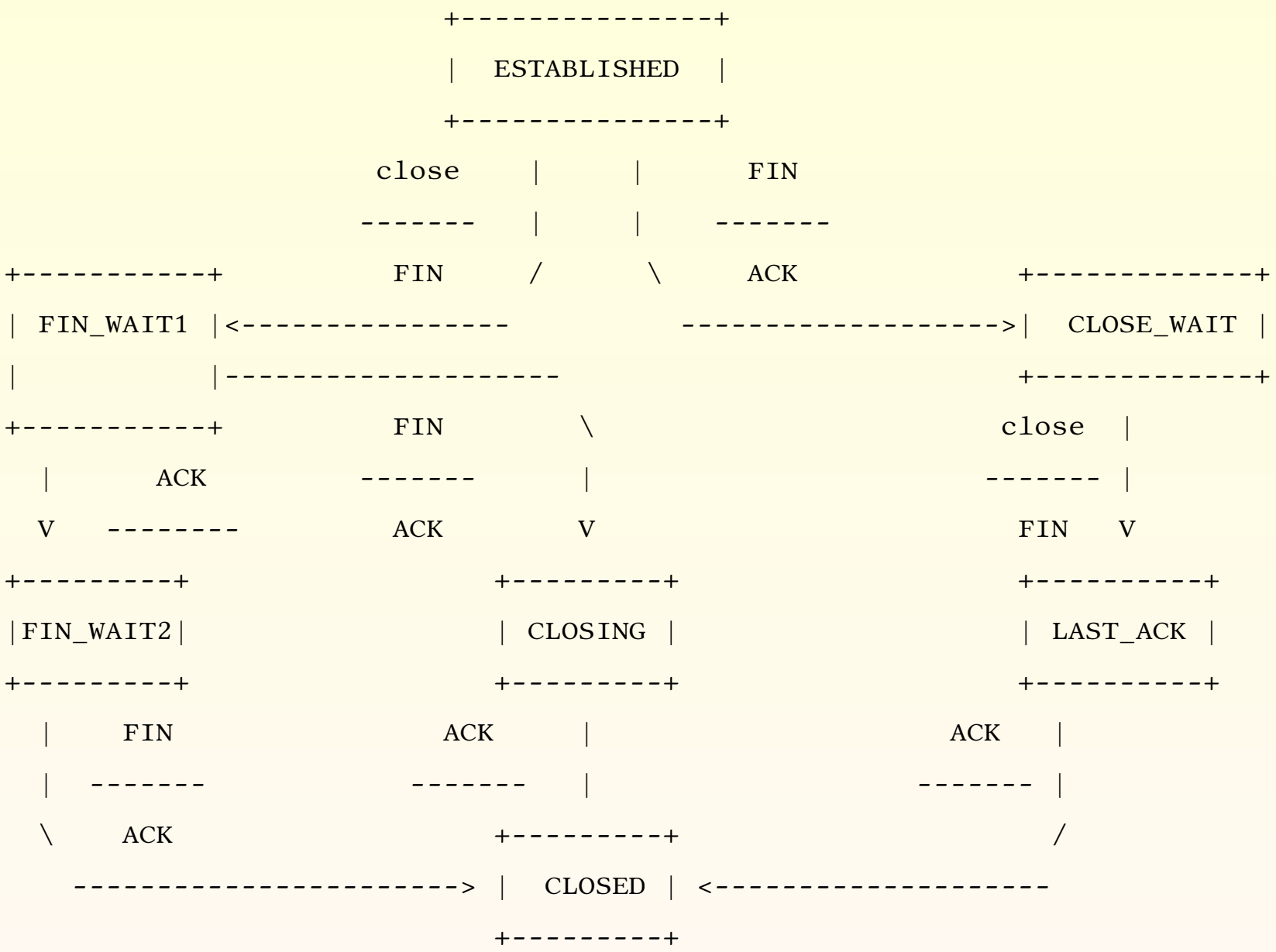


Diagrama de estados: cierre de conexión



Procesamiento de paquetes

- Al recibir un paquete, éste se procesará de una u otra manera en función del estado transitado por **PTC** .
- Todo paquete cuyo flag de ACK esté apagado **debe** ser automáticamente descartado sin importar en qué estado esté el protocolo (a excepción de LISTEN).
- Notación para las próximas slides:
  - SEG_SEQ: número de secuencia del paquete entrante.
  - SEG_ACK: número de ACK del paquete entrante.
  - SEG_LEN: longitud de los datos del paquete entrante.

Procesamiento en LISTEN

- Sólo aceptar un paquete SYN.
- Se debe inicializar el bloque de control a partir de la información de dicho paquete y un número de secuencia inicial (ISS) computado aleatoriamente.
- Cambiar el estado a SYN_RCVD, enviar SYN/ACK en respuesta y por último incrementar SND_NXT.

Procesamiento en SYN_SENT

- Sólo aceptar un paquete SYN/ACK.
- SEG_ACK debe ser el valor del número de secuencia previamente enviado más uno.
- Inicializar el bloque de control, pasar a ESTABLISHED y enviar el reconocimiento respectivo.

Procesamiento en 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).

Procesamiento en ESTABLISHED

- Si el paquete es un FIN, deberá validarse que SEG_SEQ == RCV_NXT.
  - En ese caso, pasar a CLOSE_WAIT, incrementar RCV_NXT (dado que el FIN se secuencia) y enviar un reconocimiento adecuado.
  - En otro caso, enviar un ACK informando el valor actual de RCV_NXT.
- Si no es FIN, validar en el bloque de control la aceptación del paquete.
  - SEG_ACK es aceptable sii:

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

  - En este caso, poner SND_UNA $\leftarrow$ SEG_ACK.

Procesamiento en ESTABLISHED: datos aceptables

- Los datos son aceptables sii

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

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

- En tal caso, 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 actualizar este valor sumándole la longitud de los datos aceptados.
- Además de esto, decrementar RCV_WND (pues el buffer contiene más información).

Procesamiento en ESTABLISHED: actualización de ventana

- Se da cuando el paquete esté reconociendo números de secuencia esperados:

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

- El $\leq$ de la izquierda permite procesar correctamente potenciales actualizaciones de ventana con valores de ACK repetidos.
- También 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 caso, poner SND_WND $\leftarrow$ SEG_WND, SND_WL1 $\leftarrow$ SEG_SEQ y SND_WL2 $\leftarrow$ SEG_ACK.

Procesamiento en ESTABLISHED: envío de ACKs

- Finalmente, si el paquete contiene datos, se debe enviar un ACK independientemente de si fue o no aceptado.
- Hacer piggybacking de haber datos en el buffer de salida aguardando a ser enviados.
- Caso contrario, enviar un ACK ad-hoc.

Procesamiento en FIN_WAIT1

- Si el bloque de control acepta SEG_ACK, este paquete reconoce el FIN enviado.
  - 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, validar el paquete en el bloque de control (dado que puede contener datos) y eventualmente enviar un ACK como antes.
  - Observar que en este caso no podrá hacerse piggybacking dado que ya se envió el FIN.

Procesamiento en 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 (ídem caso FIN_WAIT1).

Procesamiento en CLOSE_WAIT

- 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.

Procesamiento en CLOSING y LAST_ACK

- Lo único relevante se da cuando 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.

Código fuente de **PTC**

- La implementación actual de **PTC** está hecha en Python 2.7.
- Los módulos principales están bajo el directorio `ptc`.
- Mencionaremos brevemente cada uno de ellos en lo que sigue.

Módulo `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.
- 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:

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

Módulo `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.

Módulo constants

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

Módulo `exceptions`

- Definición de una excepción genérica (`PTCError`).
- Representa 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.

Módulo `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.

Módulo `packet`

- Implementación del paquete **PTC** , `PTCPacket`.
- Brinda una interfaz que permite definir el valor de cada campo del segmento y también de las direcciones IP involucradas.
- Define el operador `in` para verificar fácilmente si un flag está prendido en el paquete.

Módulo `packet_utils`

- Herramientas para facilitar la manipulación de paquetes:
  - `PacketDecoder`: mapea los datos recibidos de la red a un `PTCPacket`.
  - `PacketBuilder`: 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 (`build_packet`) que se apoya en el `PacketBuilder` para armar paquetes con la información actual del bloque de control.

Módulo `thread`

- Implementación de los threads del protocolo:
  - `Clock`: simula el clock del sistema. Cada `CLOCK_TICK` segundos (definido por defecto en 0.01) invocará al método `tick` del protocolo.
  - `PacketReceiver`: monitorea el socket subyacente y recibe los paquetes. 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).
  - `PacketSender`: envía los paquetes de datos y eventualmente el `FIN` mediante el método `handle_outgoing` del protocolo. Éste 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 ACKs o invocaciones a `send` por parte del usuario).

Módulo `rqueue`

- Implementación de la cola de retransmisión (`RetransmissionQueue`).
- Los paquetes se van introduciendo a medida que son enviados por primera vez.
- Sólo se retransmite de a uno por vez cuando expire el RTO.
- 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.

Módulo `rto`

- Implementación directa del RFC 6298.
- La clase `RTOEstimator` mantiene variables que van reflejando el cómputo de SRTT, RTTVAR y RTO.
- El protocolo tiene una instancia de esta clase e interactúa con ella a medida que envía/recibe paquetes.

Módulo `timer`

- Implementación de timers del protocolo (sólo `RetransmissionTimer` por ahora).
- Unidad de tiempo: ticks (relativo a la constante `CLOCK_TICK`).
- El protocolo, en cada interrupción del clock, hace que se avance un tick por cada timer.
- Internamente, mantienen la cuenta de cuántos ticks faltan para dispararse.
- Cada timer debe implementar un método `on_expired` para decidir qué hacer si el tiempo expira.

Módulo seqnum

- Implementación de números de secuencia (`SequenceNumber`).
- 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).
- Trabaja con aritmética modular y sobrecarga los operadores aritméticos tradicionales de manera de poder utilizarlos en contextos donde se esperen enteros standard.
- También provee una serie de métodos de clase que permiten hacer comparaciones en rango teniendo en cuenta que puede haber *wrap-around*.

Módulo `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`.

Módulo soquete

- Abstracción del socket raw subyacente al protocolo (Soquete).
- Permite desligarse de la declaración y uso del mismo, pudiendo así evitar la manipulación explícita de los bytes dentro del código del protocolo.

Módulo `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.
- 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.
  - Con el método `send_and_queue` podemos no sólo enviar el paquete sino además encolarlo en la cola de retransmisión.

Modo de uso

- A través del wrapper `Socket` mencionado anteriormente.
- Experiencia esencialmente análoga a la de los sockets de Python tradicionales.
- Pero todo programa que use sockets **PTC debe** ejecutarse con permisos elevados al requerir instanciaciones de sockets raw.
- Sugerimos utilizar los sockets dentro de bloques `with` para asegurar un cleanup adecuado y evitar "cuelgues".
- Demo en vivo...

Tests

- La implementación de **PTC** provee un conjunto de casos de prueba que corren íntegramente en memoria.
  - No sólo optimiza la ejecución sino que además permite evitar la instanciación de un raw socket.
  - i.e., pueden correrse sin tener permisos elevados.
- Para correrlos, ejecutar `run_tests.py` con el intérprete de Python 2.7.
- Sin argumentos, extraerá y correrá todos los tests que encuentre en el directorio `test`.
- Pasando argumentos de la forma `--<identificador>` pueden correrse tests específicos:

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

Disector de Wireshark

- Tenemos también un plugin de Wireshark para interpretar los paquetes del protocolo.
- Compilado para arquitecturas `x86` o `x86-64`.
- Compatible con versiones de Wireshark posteriores a la 1.6. Verificar la versión!
- Para instalarlo, copiar el archivo al directorio de plugins globales de Wireshark.
  - Ver cuál es en `Help` → `About Wireshark` → `Folders`.
- Modo de uso simple: usar `ptc` como filtro para que Wireshark nos muestre sólo los paquetes **PTC**.

Agenda

Primera parte: escenario de análisis

Tomando como punto de partida el código suministrado por la cátedra, implementar las siguientes modificaciones al protocolo. La idea es generar un escenario de análisis sobre el que estudiaremos qué parámetros α, β optimizan el cálculo del RTO.

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.

Segunda parte: experimentación y análisis

En el contexto de una red local (LAN), definir un esquema cliente-servidor en el que el cliente envía datos y el servidor sólo los reconoce. Para distintas combinaciones de α y β, analizar cómo evoluciona la estimación del RTO en el cliente y buscar valores que acerquen tanto como sea posible esta estimación al RTT "real". A partir de esto, analizar cómo impactan los efectos de red simulados en los parámetros encontrados. Por ejemplo, graficar lo siguiente y sacar conclusiones:

- RTO en función de α, β para un delay fijo.
- Cantidad de retransmisiones en función de α, β.
- Comparación entre el RTO y el RTT "real".

Referencias

- W. Richard Stevens (2003)
  Unix Network Programming, Volume 1: The Sockets Networking API
  Addison-Wesley
- *Beej's Guide to Network Programming*
  `http://beej.us/guide/bgnet/`
- RFC 793: Transmission Control Protocol
- RFC 1122: Requirements for Internet Hosts -- Communication Layers
- W. Richard Stevens (1993)
  TCP/IP Illustrated, Volume 1: The Protocols
  Addison-Wesley
  `http://www.pcvr.nl/tcpip`
- A. Tanenbaum (1996)
  Computer Networks, 3ra Ed. Capítulo 3: págs. 202-213.
  Prentice Hall