

Tarea 3 – CC51C Comunicación de Datos

23 de Junio 2001

Plazo: 2 semanas

Entrega: 13 de Agosto 2001

1 Objetivos

Se tiene una implementación parcial de todas las capas para un protocolo cliente-servidor, como se ha visto en clase.

El alumno deberá proveer una capa de Datos, como la vista en clases. Para validar su funcionamiento, se provee un servidor, con su propia implementación de la capa Datos, que se usará como contraparte en una conexión simulada, agregando errores controlados en la transmisión.

2 Interfaces

2.1 Física

La interfaz física y su comunicación con la capa de encapsulamiento es provista por la biblioteca.

2.2 Encapsulamiento

La capa de encapsulamiento (framing) está provista en la biblioteca, y tiene la siguiente interfaz para ser usado desde la capa Datos:

- `Fwrite(char *buf, int size)` Envía los `size` bytes de `buf`, encapsulándolos. Si es necesario fragmentarlos los divide en sub-paquetes en forma transparente.
- `int Fread(char *buf, int size)` Lee `size` bytes (y se bloquea esperándolos si faltan), desencapsulando los datos que llegan y juntándolos si es necesario. Retorna el número de bytes efectivamente puestos en `buf` (= `size`). Si hubo un timeout, `Fread` retorna -1.
- `int Freset()` Descarta todo lo que queda pendiente del paquete actual, pasa al estado correspondiente a leer el próximo paquete. Sirve para descartar un paquete incomprensible, como cuando el header no es algo válido.

2.3 Datos

Esta capa deben proveerla en forma de una función que implementará toda la capa datos usando funciones provistas por la biblioteca del curso. Las versiones de `Dread` (`Dwrite`) para recibir (enviar) datos del (al) Data layer vienen provistas en la biblioteca. La función que ustedes implementen obtendrá y enviará datos con otras funciones provistas, y se define como sigue:

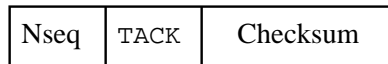
```
void Dlink_layer();
```

El protocolo es un Selective Repeat, con una ventana de tamaño definido por ustedes pero $< \text{MAX_SEQ}$. El receptor tiene una ventana de tamaño 1. Los datos manejados por este layer son buffers de tamaño máximo MAX_DATA sin el header ni checksum.

Hay dos tipos de paquetes: TDATA y TACK. El formato de los paquetes de datos es un poco diferente de los acknowledgements (ver figura 1).



Paquete de Datos



Paquete de ACK

Largos: Nseq 1 bytes
TACK/TDATA 1 byte
Largo 2 bytes (short)
Checksum 2 bytes (short)

Figura 1: Paquetes del nivel Datos

Los paquetes van numerados con un número de secuencia de 0 a $\text{MAX_SEQ}-1$. Toda la aritmética debe hacerse en módulo MAX_SEQ .

El servidor implementa un Data Link con su propia ventana, y no es necesario que sea del mismo tamaño de la de ustedes. La tarea se juzga en su capacidad de sobrevivir a los errores, sin generar inconsistencias. Al recibir un paquete fuera de rango (distinto al esperado), generen siempre un ACK para el último paquete recibido correcto. Al recibir un ACK superior al esperado, consideren que han recibido todos los ACK's anteriores a ese.

Las funciones provistas para implementar este layer son:

- `void Dinit()`
Debe ser llamada antes de todas las otras funciones de este layer.
- `int Dtime()`
Retorna el tiempo en milisegundos transcurridos desde la creación del mundo. Sirve para anotar la hora de envío de un paquete y así descubrir cuando retransmitirlo. El timeout total de un paquete puede ser de unos 200 milisegundos.

- `int DGet_next_event(int timeout)`

La rutina del nivel Datos será un ciclo infinito que obtendrá el próximo evento con esta función y lo tratará. Esta rutina recibe como parámetro el tiempo en milisegundos máximo de espera antes de retornar. Si es negativo, la función espera indefinidamente el próximo evento. Si es cero, revisa si hay eventos pendientes y retorna de inmediato.

Los valores retornados son:

- `D_READY`
Señala que hay datos disponibles del layer superior (ver `FromUpperLayer`).
- `F_READY`
Señala que hay datos disponibles en el layer Físico, que deben ser obtenidos con `Fread`.
- `F_TIMEOUT`
Señala que el tiempo máximo especificado ha transcurrido sin recibir otro evento.

- `int FromUpperLayer(char *buf)`

Recibe los datos de la aplicación (de tamaño máximo `MAX_DATA`) y retorna el largo del buffer. Estos datos son enviados por la aplicación con `Dwrite`.

- `void ToUpperLayer(char *buf, int size)`

Envía los datos a la aplicación. Estos datos son leídos con `Dread`.

- `DisableUpperLayer()`

Deshabilita los eventos de tipo `D_READY`, y la aplicación se bloqueará si intenta enviar más datos.

- `EnableUpperLayer()`

Habilita los eventos de tipo `D_READY`.

- `short checksum(char *buf, int size)`

Esta rutina calcula los dos bytes de checksum del buffer, se debe incluir el header.

- `int between(int first, int last, int elt)`

Esta rutina permite ver si un subíndice (`elt`) está en el rango entre `first` (incluido) y `last` excluido en un arreglo circular. Retorna `TRUE` si está en el rango y `FALSE` si no.

2.4 Aplicación

La aplicación servidor y cliente son provistos en la biblioteca, y se preocupa de usar la versión de `Dlink_layer()` que ustedes implementarán. El resultado de compilar y linkear la biblioteca y el `Dlink_layer` es un cliente que se comunica con el servidor entregado. Para que esto funcione, el programa servidor deberá encontrarse en el mismo directorio donde se está ejecutando el cliente.

2.4.1 Opciones de ejecución

Las opciones del cliente son las siguientes: `cliente [-d n] [-D n] [noise]`

- `-d n`: setea el nivel de debug de la capa física a n . Por defecto este valor es 0, y cuando se setea en 3 se obtiene el mayor debugging.
- `-D n`: setea el nivel de debug de la capa datos del servidor, además de las funciones provistas para la capa datos, como `Dexit()`, `EnableUpperLayer()` y `DisableUpperLayer()`. Puede usar este valor también para agregar debugging de distintos niveles en su implementación de `Dlink_layer()`.
- `noise`: setea el nivel de ruido que se simula en el enlace físico entre el cliente y el servidor. Se espera que la tarea pueda soportar al menos un nivel de ruido de 3.5, ojalá hasta un nivel de 5 (puede demorarse bastante en terminar, pero la idea es que no se bloquee, eso se puede determinar a partir del debugging) Si no se especifica, se asume un ruido de 0, y esta variable tiene los siguientes efectos:

– `noise == 0`

Nada se pierde, el layer físico entrega todos los datos intactos.

– `0 < noise <= 1`

El layer físico pierde paquetes con probabilidad $0.1 * \text{noise}$, pero no altera ni pierde bytes. Un paquete es al nivel físico, es decir en cada llamada a `Pflush` (ver apuntes).

– `1 < noise <= 2`

El layer físico pierde paquetes con probabilidad $0.1 * \text{noise}$ y pierde bytes dentro del paquete con probabilidad $0.1 * \text{noise}$ (por paquete).

– `2 < noise`

Los paquetes se pierden con probabilidad $0.1 * \text{noise}$, se pierden bytes con probabilidad $0.1 * \text{noise}$ y se alteran bytes con probabilidad $0.05 * \text{noise}$.

2.4.2 Comandos disponibles

El cliente y servidor proveen una serie de comandos que pueden utilizar para revisar el funcionamiento de la capa Datos. A continuación se incluye una descripción detallada de cada uno:

- `quit` Termina la comunicación. El servidor muere luego de contestar este pedido:

```

cliente          servidor
QUIT            QUIT

```

- `load` Pregunta al servidor la carga actual del sistema, que es un número en punto flotante (representación binaria):

```

cliente          servidor
LOAD            <float>

```

Para verificar esto, usar el comando `uptime` del sistema.

- `ping n` Pide al servidor `n` enteros desde el 0 hasta $n - 1$ en orden creciente.

```

cliente          servidor
PING <integer>   0 1 2 3 4 ...

```

- `cat <filename>` Pide al servidor que le mande un archivo. El cliente debe mostrarlo en la salida standard.

```

cliente          servidor
CAT <largo> <filename> <largo_archivo>
                                     <archivo completo>

```

Los largos son `int`. El primer largo (enviado por el cliente) indica el largo del filename, puesto que no termina en cero.

- `wc <filename>` Le envía el contenido del archivo `<filename>` al servidor, el servidor retorna el número de palabras (como el comando `wc -w`) que había en él.

```

cliente          servidor
WC <largo_archivo> <largo_archivo>
<archivo completo> <integer>

```

Los largos son `int`.

3 Archivos provistos

Para compilar la tarea, deben usar algunos archivos del directorio `~cc51c/T3`. Existe un Makefile que prepara todo lo necesario, y el único archivo que se debe modificar es el `Dlayer.c`, el cual ya viene con una buena parte de su estructura predefinida. En particular, en `link.h` se definen las constantes requeridas (`MAX_PACK`, `SOH`, `MAX_SEQ`, etc), y en `liblink.a` están las funciones provistas (nivel físico, etc.).

Para demostración, pueden probar el comando en `~cc51c/T3/client`, que contiene implementación de la tarea. Si se usa linux, es recomendable un kernel 2.2.x o 2.4.x y `glibc ≥ 2.1`.

Los archivos dependientes de la arquitectura son provistos para linux y solaris, con la extensión correspondiente. Se incluyen versiones de `Dlayer.o`, que es la versión con la que se construyó el cliente de ejemplo (solo para que se convenzan que realmente es posible hacerlo y que compila sin problemas).

Cualquier duda o pregunta o reporte de bugs, dirigirse a `cc51c@dcc.uchile.cl`, o ponerlo en `uch.ing.cursos.cc51c`.

4 Observaciones

Para saber qué paquetes se deben transmitir, es necesario guardar la información de cuándo fue (re-)enviado, y el paquete transmitido, para no tener que volver a armarlo si se necesita retransmitir.