

CC51C Comunicación de Datos

Capa Datos

1 Supuestos

- cable bidireccional (circuito virtual, cable serial, conexión telefónica, ethernet, FDDI, etc)
- `write()` a un lado y `read()` al otro, llamadas al nivel físico.
- se puede ver como un pipe. Primitivas que entrega el nivel físico:
 - `char Pgetc();`
 - `void Pputc(char c);`
 - `char Pflush();`
- el "pipe" puede tener errores:
 - pérdida de bytes
 - alteración de bytes

2 Objetivos

- proveer un canal de comunicación punto a punto libre de errores.
- en la práctica no es buena idea hacerlo en esta capa, pero sí lo es para fines docentes.

3 Encapsulamiento (Framing)

- técnica usual para detectar errores: encapsular en frames. Permite sincronizar y descartar frames erróneos.
- se agregan bytes a los datos que permiten detectar errores del frame (checksum) este checksum debe incorporar también los datos del frame.
- también los bytes que definen los frames pueden perderse.
- Ejemplo: si pongo primero el largo y luego los datos (character count), ¿qué pasa si se altera el largo? ¿Es posible volver a sincronizar?
- técnica usada: delimitadores de comienzo y final: character stuffing y bit stuffing.

4 Control de Errores

4.1 Naturaleza de los errores

- los errores tienen cierta probabilidad p de ocurrir en un bit.
- los errores tienden a ocurrir en ráfagas de largo l , donde el primer bit de la ráfaga tiene un error, el último también
- distancia de *Hamming*: número de bits en 1 al hacer el X-OR.
- distancia de un código: la distancia mínima de todos los posibles códigos dentro de los 2^m válidos entre los 2^n posibles.
- la secuencia de n bits que contiene la redundancia se llama *codeword*.

4.2 Corrección de errores

- número de bits enviados $n = m + r$, donde m es la información y r la redundancia.
- para detectar d errores, es necesario tener un código de distancia $d + 1$. Ejemplo: bit de paridad (distancia = 2, puede detectar errores simples)
- para corregir d errores, es necesario tener un código de distancia $2d + 1$ porque así codewords con d errores aún tienen menor distancia hacia el original que hacia cualquier otro codeword válido.
- Ejemplo: considerar un código con sólo cuatro codewords: 000000000, 0000011111, 1111100000 y 1111111111. Este código tiene distancia 5, y puede corregir errores de distancia 2. ($m = 2$, $r = 8$, $n = 10$)
- la corrección de errores se utiliza en medios muy propensos a error, o en medios simplex (donde no se puede solicitar una retransmisión).
- paridad 2D: se dividen los bits para formar una matriz de i filas y j columnas. Se le agrega una columna y una fila adicional, las cuales se llenan con la paridad por cada columna y por cada fila, respectivamente. Así se agregan $i + j + 1$ bits de redundancia, que permiten identificar y corregir cualquier error de 1 bit, o bien detectar cualquier ráfaga de no más de $j + 1$ bits de largo. También es capaz de detectar cualquier error en 2 bits que se pueda presentar.

4.3 Detección de errores

- en general, la detección y retransmisión en caso de error es más eficiente
- considerar un canal con errores con probabilidad 10^{-6} por bit. Si el tamaño del bloque es 1000 bits, se necesitan 10 bits por bloque para proveer

corrección de errores. En un megabit, se necesitan 10000 bits de redundancia. Para detectar bloques con un solo error se necesita 1 bit extra en cada bloque, y retransmitir un bloque de cada 1000. Así, para 1 megabit se transmiten 2001 bits extra.

- errores contiguos (en ráfagas): usar matriz y calcular bit de paridad por columna. Esto reduce la probabilidad de aceptar un código erróneo de 0.5 a 2^{-n} , donde n es el número de columnas de la matriz.
- *Checksumming*: los bits de entrada se agrupan de k bits y la suma de los bits (o de sus complementos, como en IP) forma el checksum.
- *CRC: Cyclic redundancy code*: Permite detectar todos los errores de 1 y 2 bits, todos los errores de un número impar de bits, todas las ráfagas de errores de largo 16 o menos y más del 99.99 % de las ráfagas de más bits contiguos. Se basa en buscar una secuencia dada para que los bits transmitidos sean divisibles por una constante G dada, o sea $T = n * G$, usando aritmética polinomial en base 2 (Ver figura 1).

$$T = D * 2^r \oplus R = n * G$$

o bien

$$D * 2^r = n * G \oplus R$$

donde D son los datos (d bits) a enviar, R es el código CRC de r bits que se agregan y G es un *generador* de $r + 1$ bits que conocen ambas partes. Finalmente, R se calcula como el resto de dividir D por G usando aritmética polinomial mod 2.

La gracia de usar aritmética polinomial en base 2 es que no hay “carry” en la suma, es decir que en vez de sumar se usa \oplus (X-OR). Los fundamentos matemáticos dicen que los bits de T , G y R son los coeficientes de polinomios, y por eso a estos códigos se les llama también códigos polinomiales.

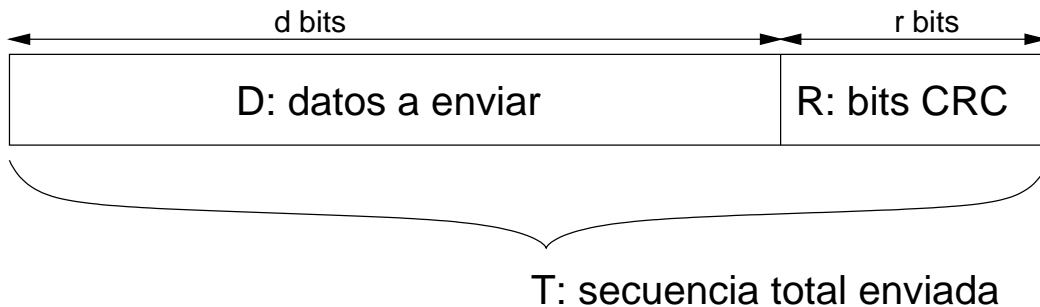


Figura 1: Código CRC

Existen definiciones para CRC de 8, 12, 16 y 32 bits (valor de r). Por ejemplo, para CRC-32, $G_{CRC-32} = 100000100110000010001110110110111$.

Cuando el cálculo se hace en software, se trata que no consuma mucha CPU. Un ejemplo de checksum tomado de IP checksum con $k = 16$:

```
short checksum(b, n)
unsigned char *b;
int n;
{
    int i, cc, sum;

    for( i=0, sum=0 ; i<n ; i++ )
        sum += b[i];
    cc = (sum >> 16) + (sum & 0x0000ffff);
    cc += (cc >> 16);

    return( (short) ~ (cc & 0x0000ffff) );
}
```

- cuando se detecta un error, se elimina el frame completo y se debe retransmitir.
- para que el emisor sepa cuándo retransmitir, se pueden enviar ACKs/NACKs
- si se pierde un frame completo, no llega ACK ni NACK. La única salida es usar un timeout en alguno de los lados (típicamente el emisor).
- pueden llegar frames repetidos y en desorden

5 Control de Flujo

- un receptor lento puede perder datos si el emisor no le da el tiempo necesario.

6 Protocolos de retransmisión

Se tiene un layer de framing que permite enviar secuencias de un lado a otro usando la siguiente interfaz:

```
int Fread( char *buf, int size );

void Fwrite( char *buf, int size );

void Freset();
```

En base a esto, trataremos de proveer dos funciones que permitan enviar un buffer de datos libre de errores:

```
void Dread( char *buf, int size );
void Dwrite( char *buf, int size );
```

Vamos a suponer que el paquete de datos sigue el esquema de la figura 2.

Type	Size	Data	CC
1	2	1 -- MAX_DATA	2

Figura 2: Paquete de Datos

6.1 Directo

- Dwrite invoca a Fwrite
- Dread lee el paquete usando Fread y lo revisa. Si el checksum está OK, se envía un ACK. Si no, se envía un NACK.
- Dwrite invoca a Fread esperando el [N]ACK.

6.2 Stop & Wait

Es el más simple de estos esquemas (que funcione). El envió se queda bloqueado hasta que logra enviar los datos. Sólo después trata de enviar más datos.

```
#define TYPE 0 /* char */
#define SIZE 1 /* short */

/* Tamano del header en bytes */
#define HEADER_SIZE 3

/* Tamano de un Ack (sin checksum) */
#define ACK_SIZE 1

#define TDATA 'D'
#define TACK 'A'

Dwrite(char *buf, int size) {
    short cc;
    int outsize;
    char InBuf[ACK_SIZE + sizeof(short)];

    /* Armos el paquete de salida en OutBuf */
```

```

    OutBuf[TYPE] = TDATA;
    bcopy(buf, OutBuf + HEADER_SIZE, size);
    bcopy(&size, OutBuf + SIZE, sizeof(short));
    cc = checksum(OutBuf, size + HEADER_SIZE);
    bcopy(&cc, OutBuf + size + HEADER_SIZE, sizeof(short));
    outsize = size + HEADER_SIZE + sizeof(short);

    for (;;) {
Fwrite(OutBuf, outsize);
/* Espero el ACK */
if (Fread(InBuf, ACK_SIZE + sizeof(short)) !=
    ACK_SIZE + sizeof(short)) {
    Freset();
    continue;
}
if (InBuf[TYPE] != TACK) {
    Freset();
    continue;
}
bcopy(InBuf + ACK_SIZE, &cc, sizeof(short));
if (checksum(InBuf, ACK_SIZE) != cc) {
    Freset();
    continue;
}
return;
    }
}

char Dpbuf[MAX_DPACK];
int Dpsize = 0;
char *Dpcp = Dpbuf;

char Ack[] = { TACK, ' ', ' ' };

Dread(char *buf, int size) {
    for (;;) {
if (Dpsize >= size) {
    bcopy(Dpcp, buf, size);
    Dpcp += size;
    Dpsize -= size;
    return;
}
if (Dpsize > 0) {
    bcopy(Dpcp, buf, Dpsize);
    size -= Dpsize;
    buf += Dpsize;

```

```

        Dpcp = Dpbuf;
        Dpsize = 0;
    }

    /* El resto debo leerlo */
    Dpsize = Dget_packet(Dpbuf);
    Dpcp = Dpbuf + HEADER_SIZE;
    }
}

int Dget_packet(char *buf) {
    short size;
    short cc;

    /* Trato de obtener un paquete correcto */
    for (;;) {
    if (Fread(buf, HEADER_SIZE) != HEADER_SIZE) {
        Freset();
        continue;
    }
    if (buf[TYPE] != TDATA) {
        Freset();
        continue;
    }
    }
    bcopy(buf + SIZE, &size, sizeof(short));
    if (size > MAX_DATA || size <= 0) {
        Freset();
        continue;
    }
    if (Fread(buf + HEADER_SIZE, size + sizeof(short)) !=
        size + sizeof(short)) {
        Freset();
        continue;
    }
    }
    bcopy(buf + size + HEADER_SIZE, &cc, sizeof(short));
    if (checksum(buf, size + HEADER_SIZE) != cc) {
        Freset();
        continue;
    }
}

/* Esta OK, envio Ack */

cc = checksum(Ack, ACK_SIZE);
bcopy(&cc, Ack + ACK_SIZE, sizeof(short));
Fwrite(Ack, ACK_SIZE + sizeof(short));

```

```

/* Retorno el paquete de datos */
return (size);
    }
}

```

6.3 Go Back N

Este mecanismo usa una ventana corredera, y básicamente envía varios paquetes al mismo tiempo, aprovechando mejor el canal. Luego espera un ACK para cada paquete de ese grupo. Si se pierde algo, se considera perdida toda la ventana y se vuelven a enviar todos los paquetes pendientes. Nótese que Stop & Wait es como tener un esquema *Go Back N* con tamaño de ventana 1.

En *Go Back N* el receptor no tiene ventana (o tiene ventana de tamaño 1), por lo cual no puede admitir paquetes “adelantados”.

Es importante notar que, dado que hay que retransmitir y recibir [N]ACKS, la forma de programar estos layers no puede ser mediante sub-rutinas simples. La capa datos debe funcionar en forma paralela con la capa aplicación. Ver figura 3

Definimos entonces un proceso Dlayer que funciona en base a eventos:

1. D_READY Hay datos de la aplicación para ser leídos (la aplicación usó `Dwrite(...)`). Para leer estos datos se debe usar `FromUpperLayer(char *buf)`. El máximo de datos a leer es `MAX_DATA`.
2. F_READY Hay datos del framing layer para ser leídos usando `Fread(...)`. Para enviarlos a la aplicación (si corresponde), se usa `ToUpperLayer(char *buf, int size)`.
3. F_TIMEOUT Pasó más tiempo del esperado.

6.3.1 Optimizaciones:

- si llega un paquete adelantado, se puede de inmediato avisar al emisor que no llegó el anterior.
- si al emisor le llega un ACK del paquete n , puede ser interpretado como un ACK para cada paquete $i \leq n$ en la ventana
- si llega un NACK para el paquete n , puede ser interpretado como un NACK para cada paquete $i \geq n$. Al mismo tiempo, se puede interpretar como un ACK para todo $i < n$.

6.3.2 Consideraciones:

- tamaño óptimo de la ventana: mientras más errores hay en la línea, más chica debe ser la ventana.

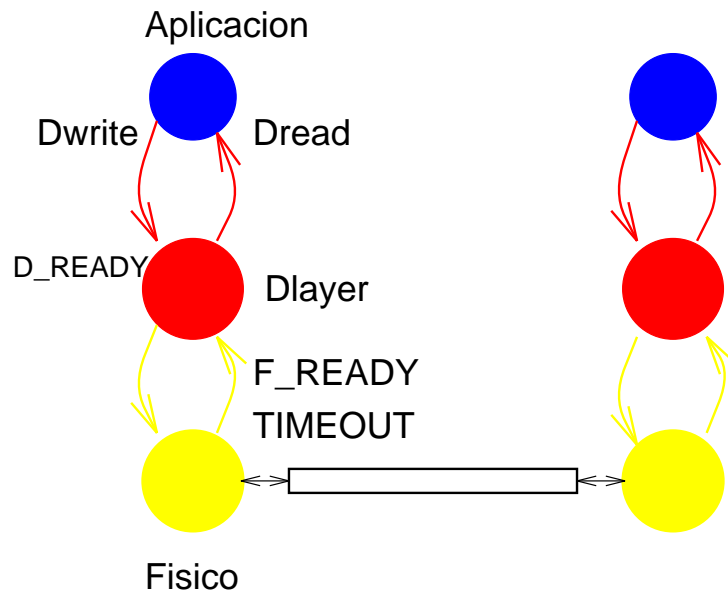


Figura 3: Eventos del proceso Dlayer

- tamaño máximo de la ventana (para que el reuso de números de secuencia no sea ambiguo).
- una implementación real es bidireccional, acá estamos simplificando. Si es bidireccional, se pueden aprovechar paquetes de datos para agregar un ACK dentro de ellos (si la definición del paquete lo permite): *piggybacking*.

6.3.3 Receptor

```

for(;;)
  switch(Get_next_event(-1))
  {
  case F_READY:
    <<Leer proximos datos con Fread>>
    <<Ver que esten correctos>>
    if( buf[SEQN] == ExpectedSeqN )
    {
      AckBuf[SEQN] = ExpectedSeqN;
      Fwrite(AckBuf, ACK_SIZE );
      ExpectedSeqN++;
      ToUpperLayer(buf+DATA, buf[size]);
    }
  }

```

6.3.4 Emisor

Además de enviar y esperar los ACKs, debemos fijarnos en que si la aplicación sigue enviando datos y no podemos seguir escribiendo los datos si la ventana está en su tamaño máximo. Para esto se definen dos primitivas: `DisableUpperLayer()` y `EnableUpperLayer()`.

```
timeout = -1;
win_size = 0;
ExpectedAck = 0;
SeqN = 0;
for(;;)
{
    switch(Get_next_event(timeout))
    {
        case D_READY:
            size = FromUpperLayer(buf);
            <<Armo el paquete Data Link, con checksum y Header (incluye SeqN)>>
            <<Pongo el paquete en window[Last], con la hora actual>>
            Fwrite(window[Last].buf, window[Last].size);
            SeqN++;
            Last++;
            win_size++;
            if( win_size == WINDOW_SIZE )
                DisableUpperLayer();
            break;
        case F_TIMEOUT:
            for(i = First; i < Last; i++)
            {
                <<Pongo hora actual en el paquete>>
                Fwrite(window[i].buf, window[i].size);
            }
            break;
        case F_READY:
            Fread(Ackbuf, ACK_SIZE);
            if( Ackbuf[SEQN] == ExpectedAck )
            {
                win_size--;
                First++;
                ExpectedAck++;
                EnableUpperLayer();
            }
            break;
    }

    if( win_size > 0 ) timeout = window[First].time + TIMEOUT;
    else                 timeout = -1;
}
```

}

¿Qué pasa si se recibe un ACK fuera de rango?

6.4 Selective Repeat

Es el esquema más general de ventana corredera. Sólo que en el receptor también tengo una ventana de tamaño > 1 . Eso significa que puede recibir paquetes adelantados sin necesidad de retransmitir todos.

6.4.1 Consideraciones

- ya no es posible optimizar los ACKs del paquete n considerando que vale para todos los paquetes $< n$