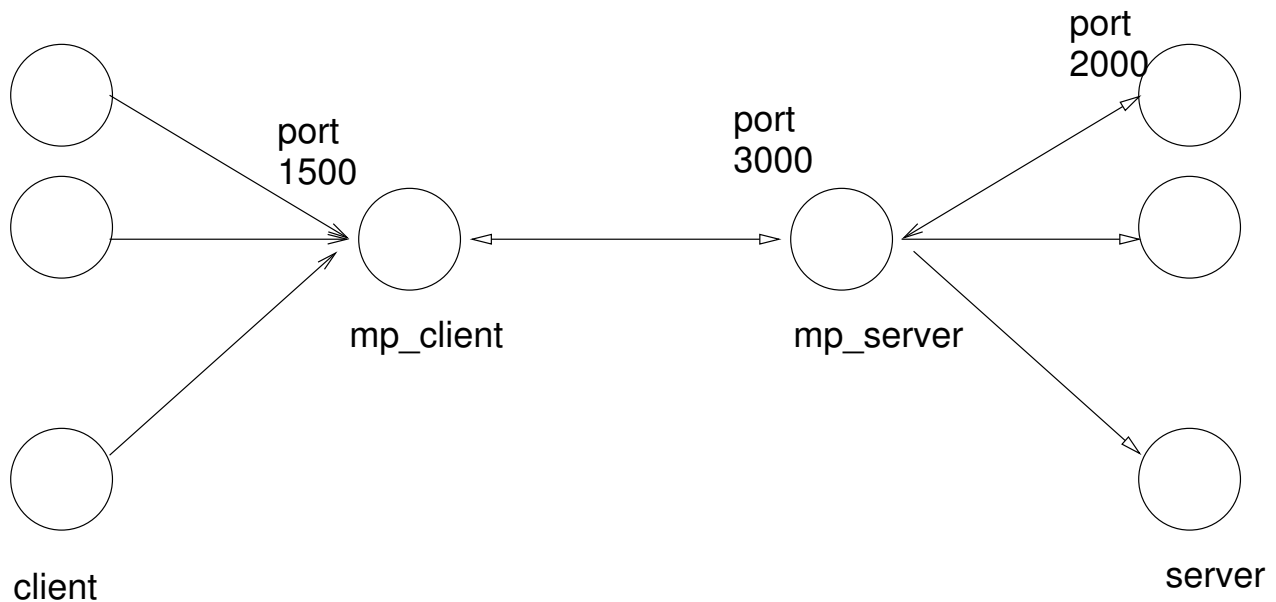


1. Objetivos

Esta tarea persigue construir un Multiplexor tolerante a fallas en los enlaces. Es una modificación del multiplexor cliente de la tarea3, de modo de soportar ruido entre él y el multiplexor servidor. No hay ruido ni errores entre los clientes y servidores de la tarea2 y los multiplexores. El cliente de la tarea2 debe modificarse un poco para inicializar la comunicación solamente (enviar el nivel de ruido deseado) y el servidor queda sin modificaciones.

El multiplexor también se divide en un cliente y un servidor, el servidor está provisto y está funcionando en dichato:



Cuando el cliente de la tarea2 se conecta al **mp_client**, le envía el nivel de ruido que quiere tener en su conexión particular en un byte. Ese mismo parámetro debe enviarlo el **mp_client** al **mp_server** al iniciar esta conexión. El **mp_server** implementa todo el ruido para esa conexión solamente, ruido 0 es sin ningún error y 100 es infinito.

Ustedes deben programar y entregar el **mp_client** completo, implementado un protocolo de Stop-and-Wait por conexión, con el **mp_server**. No hay modificaciones de

protocolo entre los clientes tarea2 aparte del byte de ruido inicial y no hay pérdidas en esas conexiones tampoco.

2. Framing

Esta capa queda igual a la tarea3.

3. Multiplexión (`mp_client`)

El multiplexor se conecta con el `mp_server` y ese socket será el de multiplexión. Luego espera conexiones de los clientes. El multiplexor actúa como un servidor hacia el lado de los clientes, y todo lo que recibe debe encapsularlo en el socket multiplexor, en un frame (comenzando con SOH, escapando los caracteres y terminando con EOP y sin fragmentarlo) de la siguiente forma:

- Conexión de cliente: Al recibir una nueva conexión, el multiplexor debe leer un byte con el nivel de ruido deseado para esa conexión. Un valor 0 implica sin pérdida, un valor 100 implica 100 % de pérdida. El multiplexor debe informar en el socket para que se cree la conexión y el servidor respectivo al otro lado. Para eso envía un paquete de control con 5 bytes:

```
C id seq cc noise
```

Donde `id` es un byte (entre 0 y 255) que identifica en forma única esta conexión (`mp_client` debe asignarlo y demorarse lo más posible en reutilizarlos para evitar confusiones), `seq` es el número de secuencia inicial de esta conexión (se usa el mismo para el envío y la recepción) (`mp_client` debe asignarlo en forma aleatoria) y `noise` es el ruido (entre 0 y 100). Se utiliza un checksum de un byte (`cc`) para verificación de integridad.

El ruido afecta a toda la conexión, incluido el paquete de conexión, por lo que este paquete también requiere un ACK y ser retransmitido.

- Término de conexión: Al detectar el término de una conexión (EOF) debemos informar al otro extremo que esa conexión se cerró, con un paquete de 4 bytes:

```
S id seq cc
```

Este paquete también requiere un ACK.

- Paquete de Datos: Cada paquete de datos entre cliente y servidor va en el multiplexor como:

```
D id seq cc data ... ..  
+----- size bytes ----+
```

Donde va el número de secuencia de este paquete (cuenta paquetes, no bytes) y el checksum.

- ACK: Al recibir bien un paquete, se emite un ACK:

A id seq cc

El número de secuencia es el del último paquete recibido correcto.

El multiplexor puede implementarse con threads o con select, pero no con fork y procesos pesados, puesto que el socket multiplexor debe compartirse.

El código del verificador de checksum es:

```
unsigned char 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 >> 24);
    cc += (sum >> 16) & 0x000000ff;
    cc += (sum >> 8) & 0x000000ff;
    cc += sum & 0x000000ff;

    return( (unsigned char) ~ (cc & 0x000000ff) );
}
```

Para calcular el checksum de un paquete se utiliza todo el header y los datos. En el header, se debe usar un checksum 0 al calcular el checksum (por razones obvias, el checksum no verifica al checksum).

4. Stop and Wait

El multiplexor debe implementar un Stop and Wait por conexión, para recuperarse de los errores del medio. El timeout de transmisión lo pueden fijar a un segundo. Los números de secuencia son de un byte y deben volver a 0 al superar 255. Mientras un paquete de datos no logra recibir su ACK respectivo, no deben aceptar más datos desde esa aplicación.

Para implementar esto, deben manejar un buffer de transmisión y los números de secuencia para cada cliente conectado por separado.

5. Ejecución y Debugging

En `dichato.dcc.uchile.cl` hay un servidor en el puerto 3000 ejecutando este protocolo y que se conecta con un servidor de la tarea2 que corre también en `dichato`.

Pueden usar los clientes de la tarea2, modificándolos para que envíen el primer byte con el ruido y deben construir el `mp_client` en esta tarea.

Si van a probar todo el sistema en un solo PC, les dejo los ejecutables en la página del curso. El cliente provisto acepta los argumentos:

```
./tcp_client -n noise host port
```

La implementación del ruido es como sigue:

- Ruido 0 Sin pérdidas de ningún tipo
- Ruido 1-9 Pérdidas de paquetes
- Ruido 10-19 Pérdidas de paquetes y de bytes
- Ruido 20-100 Pérdidas de paquetes, de bytes y alteración de información

El ruido siempre implica una probabilidad creciente de errores y afecta tanto al envío como a la recepción de datos en esa conexión.