SUBE

Ejercicio N°3

Objetivos	● Diseño y construcción de sistemas con acceso distribuido ● Encapsulación de Threads y Sockets en Clases ● Definición de protocolos de comunicación ● Protección de los recursos compartidos ● Uso de buenas prácticas de programación en C++
Instancias de Entrega	**Entrega 1:** clase 8 (03/10/2017). **Entrega 2:** clase 10 (17/10/2017).
Temas de Repaso	● Definición de clases en C++ ● Contenedores de STL ● Excepciones / RAII ● Move Semantics ● Sockets ● Threads
Criterios de Evaluación	● Criterios de ejercicios anteriores ● Eficiencia del protocolo de comunicaciones definido ● Control de paquetes completos en el envío y recepción por Sockets ● Atención de varios clientes de forma simultánea ● Eliminación de clientes desconectados de forma controlada

Índice

Introducción

Para mejorar la eficiencia de las tarjetas SUBE, se nos pide implementar un cliente y servidor que simule este sistema de pago.

Descripción

Las tarjetas SUBE son tarjetas RFID que almacenan un saldo, en pesos, que el usuario puede utilizar para pagar, por ejemplo, boletos de colectivo, tren, etc. El saldo de estas tarjetas se encuentra bajo una firma digital (mecanismo de criptografía), de forma tal que la recarga de las mismas no pueda ser alterada fácilmente, mientras que los pagos realizados pueden almacenarse en la tarjeta sin necesidad de una firma.

En nuestra aplicación, leeremos un archivo binario que representa la salida del lector RFID, en el que se encuentran varias lecturas de tarjetas SUBE.

Formato de Línea de Comandos

Servidor

El modo servidor recibirá

1. El puerto por el que escuchará conexiones

Por ejemplo

```
./server 8080
```

Cliente

El modo cliente recibirá

1. La ip del servidor a conectarse
2. El puerto al cual debe conectarse
3. El path del archivo binario con las transacciones

Por ejemplo

```
./cliente 127.0.0.1 8080 data.bin
```

Códigos de Retorno

En cualquier caso, tanto cliente como servidor deben retornar 0.

Entrada y Salida

Cliente

Las transacciones se ingresarán en texto binario mediante un archivo pasado por parámetro.

El formato de las lecturas en el archivo binario es el siguiente:

```
<metadata><tarjeta>[<monto>]
```

- **metadata**: Campo de 16 bits con el siguiente formato: 5 bits de “checksum” de la tarjeta, 5 bits de “checksum” del monto, 3 bits de operación, 3 bits en 0 que son ignorados.

- **tarjeta**: Campo de 32 bits sin signo que representa el id de la tarjeta. Obligatorio en todas las transacciones.

- **monto**: Campo de 32 bits con signo que representa el monto (en pesos) asociado a la operación. El campo no se encuentra presente en las operaciones que no lo requieren, como al momento de registrar la tarjeta o de consultar saldo.

Todos los campos se encuentran en big endian.

Metadata

Los primeros 5 bits de este campo representan un “checksum” de la tarjeta. Este checksum se calcula sumando todos los bits en ‘1’ del campo tarjeta. Por ejemplo, la tarjeta número 20 (`0x00000014` en big endian), posee los bits 3 y 5 (comenzando desde la derecha) en 1, por lo que su checksum será 2, es decir `00010`.
La misma lógica se aplica al monto. En caso de que el la cantidad de ‘1’s sea 32, el checksum será 0 (es decir, en el caso que la tarjeta sea `0xFFFFFFFF` o que el monto, al ser con signo, sea -1)
La operación es un número de 3 bits que pueden tomar los siguientes valores:

- `000`: Agregar monto
- `001`: Forzar agregar monto
- `010`: Preguntar saldo
- `011`: Registrar tarjeta
- `100`: Setear saldo

De esta forma puedo codificar, por ejemplo, que quiero agregar un monto de 100 a la tarjeta 520 de la siguiente forma:

```
500 = 0x01F4 -> Checksum = 6 = 0b00110
100 = 0x0064 -> Checksum = 3 = 0b00011
Op = 000
```

Metadata `= 00110 00011 000 000 = 0x30C0`

Transaccion final `= 30C0 0000 01F4 0000 0064`

Las operaciones de registrar tarjeta y preguntar el saldo no requieren un monto para realizarse, motivo por el cual la transacción sólo poseerá metadata y número de tarjeta. En estos casos, para el cálculo del checksum se asume monto igual a 0.

Ejemplo:

Registrar tarjeta 4

```
4 = 0x0004 -> Checksum = 1 = 0b00001
No hay monto -> Checksum = 0
Op = 011
```

Metadata `= 00001 00000 011 000 = 0x0818`

Transaccion final `= 0818 0000 0004`

Tip: para trabajar con bits, utilizar operadores bitwise (a lo C) o usar la la biblioteca `<bitset>`[1]

Mensajes de error

En caso de que el checksum en el metadata no coincida con el calculado, se imprime por salida standard el código `E00001`: transacción inválida de la siguiente forma

```
<transacción erronea> -> E00001
```

Ejemplo:

Transaccion leida `= 0818 0000 0003`

Metadata `= 0x0818 = 00001 00000 011 000`

Operación `= 011 =` Registrar
Checksum tarjeta: `00001` = 1
Tarjeta leida: 0003
Checksum calculado = suma de unos en 0b0011 = 2

Esta transacción no se envía al servidor, y se escribe en la salida standard

```
R00000003 -> E00001
```

Servidor

El servidor deberá esperar por consola que se ingrese el caracter **q**. Cuando lo recibe, debe terminar ordenadamente, cerrando conexiones y liberando recursos. En paralelo debe atender conexiones entrantes.

Mediante la salida standard se imprimirá la siguiente información, cada línea finaliza con un **\n**:

```
<transacción recibida> -> <respuesta a enviar>\n
```

En caso de que se produzca un error, la salida será por la salida de error standard

```
<transacción recibida> -> <codigo de error>\n
```

Ejemplo

El servidor inicia y recibe un pedido de registrar la tarjeta 12345

Salida standard: `R0000012345 -> R0000012345`

El servidor recibe un pedido de agregar 54321 pesos a la tarjeta 12345
Luego recibe un pedido de agregar un gasto de 50000 pesos a la misma tarjeta

Salida standard:
`A00000123450000054321 -> A00000123450000054321`

`A0000012345-000050000 -> A00000123450000004321`

El servidor recibe un pedido de registrar nuevamente la tarjeta 12345. Envía el mensaje de error E00004, tarjeta existe.

Salida error: `R0000012345 -> E00004`

Mensajes de error

Los mensajes de error que puede enviar el servidor son

Tarjeta inválida: `E00002` - la tarjeta utilizada en la transacción no existe.
Monto inválido: `E00003` - el monto a agregar no es válido.
Tarjeta existente: `E00004` - la tarjeta que se intenta registrar ya fue registrada previamente.

Protocolo de comunicación

Formato del mensaje

Los comandos se enviarán utilizando texto plano de largo semi-fijo.

Envío de transacciones

Los comandos serán enviados en mensajes de texto con el siguiente formato

```
<cod-comando><tarjeta>[<monto>]
```

Donde `cod-comando` es el código del comando a enviar.

Los códigos de comando son los siguientes:

- `A`: Agregar monto
- `F`: Forzar agregar monto
- `P`: Consultar monto (Poll/Preguntar)
- `R`: Registrar tarjeta
- `S`: Asignar monto (Set)

Se envía el caracter que determina el comando y a continuación la tarjeta y monto en texto plano, en formato decimal, con ancho 10, rellenado con ceros.

Tip: ver la biblioteca `<iomanip>`[2]

Se asegura que no habrá montos menores a `-999999999`

Ejemplos:

Registrar tarjeta 124: `R0000000124`
Agregar 1200 pesos a tarjeta 124: `A00000001240000001200`

Restricciones

La siguiente es una lista de restricciones técnicas exigidas por el cliente:

1. El sistema debe desarrollarse en ISO C++11.
2. Está prohibido el uso de variables globales.
3. La aplicación debe soportar clientes en simultáneo.

Referencias

[1] std::bitset: http://en.cppreference.com/w/cpp/utility/bitset
[2] std::iomanip: http://en.cppreference.com/w/cpp/header/iomanip