Lab 4. Aplicación distribuida segura

La aplicación CallSystem que se describe en esta memoria esta preparada para trabajar con un servidor y múltiples clientes de manera distribuida.Pero, una vez desconectado un cliente, este no podrá volver a conectarse en la misma sesión de servidor. Téngase en cuenta a la hora de testear la aplicación. El motivo de esto se explicara mas adelante.

Para ejecutar la aplicación de manera distribuida, no hay que modificar el código fuente. El servidor se arranca siempre igual, mientras que en el cliente, se nos pedirá por pantalla la ip del cliente y después la ip del servidor a la que conectarse. Los puertos están fijos. Si se quiere probar en local, simplemente en vez de la ip escriba localhost.

Interfaces ICE

Necesitaremos usar dos interfaces. Una que modele las funciones que ejecuta el cliente hacia el servidor y otra interfaz que modele las funciones que implementa el cliente y que puede invocar el servidor.

UserManager es la interfaz que implementa el servidor e invoca el cliente .Hay que destacar las funciones de connect y disconnect, su utilidad se vera mas adelante. Como tenemos múltiples clientes en la función avisar consumo, tenemos indicar el la ip del cliente en la cual queremos la notificación de un usuario determinado.

```
module CallSystem {
    interface UserManager{
        int darAlta(int dni);
        int comprarMinutos(int dni,int minutos);
        int avisarConsumo(int dni,int minutesAlertThreshold,string myip);
        int connect(string myip,string port);
        int disconnect(string myip);
    };
};
```

AlertSystem es la interfaz que implementa el cliente y tiene dos funciones. Estas las invoca el servidor para notificar al cliente las alertas de consumo y cuando va a apagarse de manera controlada.

```
module CallSystem {
    interface AlertSystem{
        void consumAlert(int dni,int minutesAlertThreshold);
        void shutDown();
    };
};
```

Entre las dos interfaces estamos usando dos tipos ice, int y string. Una string en ice,la convertimos en c++ a un objeto de tipo string. En el caso de un int en ice, puedes interpretarlo en c++ de varias

maneras. En nuestro caso, guardamos el tipo ice int y lo convertimos en un unsigned int. Nótese que simplemente estamos modificando la manera en la que se interpreta la variable, no su contenido.

Arquitectura del Sistema

El objeto principal de ice, es el comunicador. Es el objeto que gestiona la comunicación en un determinado programa. Se ha descubierto que solo se puede tener un comunicador, independientemente del numero de conexiones que se implemente. Este se declarara de manera global. En la parte de cliente tendremos dos hebras. Una que hará de cliente ice y se conectara al adaptador del servidor, y otra que creara un servidor en el mismo comunicador e iniciara un adaptador nuevo al que el servidor podrá conectarse e invocar funciones.

En la parte servidora tenemos un esquema similar, con pequeñas matizaciones. En el arranque del servidor solo se inicializa la hebra servidora (implementada en la hebra main en este caso). Cada vez que un cliente nuevo se conecta, este invoca la función remota connect de servidor. El servidor añadirá una nuevo cliente al vector de clientes y creara un hebra cliente con la ip y el puerto proporcionado por el cliente. Cuando el cliente se desconecta lo eliminara del vector y cerrara la hebra cliente para esa maquina concreta. Esto permite a nuestra aplicación tener mas de un cliente conectado a la vez. Como el comunicador ice es global y común a todos, cuando un cliente se desconecta y quiere reconectarse, este falla puesto que no hacemos limpieza en el comunicador cuando eliminaos un cliente. En una futura versión se podría implementar.

También capturamos la excepción Ice: :ConnectionRefusedException y la manejamos. Así notificamos al cliente si el servidor esta caído a la hora de conectarse y le avisamos, en vez de recibir una excepción. La aplicación se cierra igualmente.

```
 } catch (const Ice::ConnectionRefusedException& ex) {
        cout << "CallSystem no esta disponible en estos momentos, porfavor
intentelo mas tarde." << endl;
        cout << "Si el problema persiste contacte con el servicio de atencion al
cliente"<<endl;
        status = 1;
  }
```

Con esto ya tenemos una comunicación bidireccional con cada cliente.

Otras hebras

El código también cuenta con un dos hebras mas en el cliente. Una de ellas es la hebra consumo que simplemente se crea y se destruye cada vez que el usuario lo pide. Esta va modificando datos del vector de usuario. La otra hebra que tenemos en marcha es la hebra monitora, esta gestiona la eliminación de usuarios y lanza los avisos. Cuando detecta un usuario al que le toca un aviso

determinado, incluye su dni en un vector de avisos, que las hebras cliente comprueban para lanzar el aviso si ese usuario forma parte de su cliente. Esta hebra no se destruye. Se crea en la inicialización y usando una variable de condición se pone en espera pasiva.

Como se puede observar se usa mucha memoria compartida en el programa, y no hemos hablado de sincronización todavía, pero claramente los accesos al vector de datos de usuarios, al vector de maquinas y al vector de avisos, deben estar sincronizados.

```
static vector<client_t> datos_cl;
static vector<machine_client_t> maquinas_avisos_cl;
static vector<unsigned int> avisos_cl;
```

Mecanismos de sincronización

Aparte de las hebras anteriormente descritas, cada vez que se invoca un función remota,esta la ejecuta el servidor en una hebra independiente. Un motivo mas por el que necesitamos proteger las secciones criticas correspondientes a la lectura y escritura de los 3 vectores descritos arriba.

En el caso del vector de maquinas y el de avisos, estos vectores tiene un tamaño muy reducido(Inferior a 20). Por tanto con una simple protección global con un mutex nos valdría para sincronizar los accesos. Como el array es corto, la penalización en eficiencia de un solo mutex es prácticamente despreciable, con la potencia actual de computo.

En el caso del array de usuarios este puede ser mas grande, por tanto lo mas eficiente probablemente seria una sincronización por bloques del array. Proteger posición a posición es muy complejo y usar un solo mutex puede ralentizar el sistema. Por tanto, esta es una solución equilibrada. Un numero adecuado podría estar alrededor de 1000 usuarios por bloque. Como nosotros no llegamos a esa cifra y tenemos solo un bloque, hemos usado solo un mutex.

Por ultimo, vamos a explicar un poco mas detalladamente como hemos modelado el arranque y espera de la hebra monitora. Hemos usado una variablede condición y el mutex del array de datos. Usamos este y no creamos uno nuevo ya que en cada ejecución de la hebra monitora necesitamos obtener este mutex para poder leer y escribir en el array de usuarios. En la hebra tendremos la siguiente construcción:

```
  while(1){
        pthread_mutex_lock( &_mutex );
        while(monitora_status==0){
            pthread_cond_wait(&_var_cond, &_mutex);
        }
        //CODIGO HEBRA MONITORA
          pthread_mutex_unlock( &_mutex );
    }
```

Si hay mas de dos usuarios ejecutara, si no entrara en el bucle y esperara a que reciba una señal.

Esta señal la genera la funcion dardeAlta cuando detecta que hay mas de dos usuarios y la hebra esta parada. La propia hebra monitora se desactiva cuando borra usuarios. Esto lo hace poniendo la variable de condición a 0.

Smart Pointer

Un puntero en C es un variable que almacena una direción de memoria. Sobre esta direción de memoria podremos opera como si de un valor se tratara.

Un smart pointer es una etiqueta a un objeto o variable. Son maneras alternativas de llamar a el mismo objeto o variable. Sobre un smart pointer no podremos operar o cambiar el tipo de datos al que se refiere, en C si podiamos

La ventaja reside cuando queremos hacer paso por referencia. En C usamos punteros para realizar paso por referencia, con los errores que puede conllevar. Un puntero puede estar señalando a una estructura de datos distinta a la que esperamos, mientras que una referencia esta asociada a un tipo de datos en concreto. Ese tipo de errores saltaran en tiempo de compilación y no en tiempo de ejecución, con las ventajas que ello conlleva.