Taller de Programación I (75.42)

Ejercicio 4 - Cargador remoto de firmware

ISO C++ / Posix Threads / BSD Sockets

Contenido

Objetivo

Introducir al alumno en los conceptos básicos de programación concurrente y sockets encapsulando las API del sistema operativo en objetos de C++.

Introducción

Ya hemos construído un monitor y controlador de estaciones de baja tensión para la empresa en donde trabajamos, pero el trabajo no termina acá. Resulta que el firmware de los PLC (Programmable Logical Controller) que controla el sistema SCADA fue programado a las apuradas y cada tanto es necesario actualizarlo. Seguir haciendo esta actualización en el sitio y a mano se está volviendo una tarea muy poco redituable.

Desarrollo

Tendremos que desarrollar una aplicación para poder enviar el firmware actualizado a través de la red TCP/IP a los PLC. Para esto aprovecharemos la codificación de S-Registers que ya hemos desarollado, ya que los PLC usan un microcontrolador motorola y algunos todavía pueden ser actualizados sólo utilizando una línea serie (y como recuerdan, la codificación de S-Registers puede ser transmitida por líneas serie). De esta manera se podrá compartir gran parte del código entre ambas implementaciones.

A cada PLC se le instalará un programa servidor que escuche conexiones en un determinado puerto y espere que se le transfiera a través del socket el contenido de la memoria del nuevo firmware en formato S-Registers. A medida que recibe los datos debe guardarlos en disco para luego hacer el upgrade del firmware cuando esté seguro que la transferencia fue hecho correctamente (no hay errores de checksum por ejemplo).

Por otro lado, debe construirse un cliente que se conecte a los PLC y les transfiera el firmware (en formato S-Registers, por supuesto).

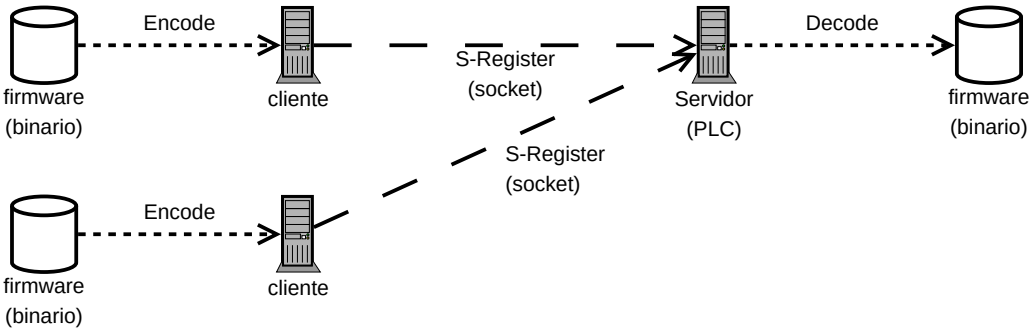

Figura 1: Esquema de la aplicación

Restricciones de diseño

La empresa ya aprendió una valiosa lección con los firmware defectuosos, por lo que quiere que esta aplicación esté bien hecha desde un comienzo y que sea simple y extensible (planean reutilizar el código para otras aplicaciones).

Es por esto que el programador debe seguir varios lineamientos de diseño para cumplir con los requerimientos de la empresa.

La aplicación se debe componer de dos módulos principales (el servidor y el cliente) quienes a su vez compartirán otros módulos más pequeños (socket y thread). Sin embargo la empresa quiere como primer entrega un prototipo simple que sirva para efectuar pruebas. Es por esto que aunque la naturaleza de la aplicación es que se componga de 2 procesos (cliente y servidor), se hará en uno sólo, pero sin embargo desde el diseño debe realizarse como si el cliente y el servidor corrieran en procesos distintos. Además el cliente no debe codificar los archivos en formato S-Registers porque ya se le proveerán los firmware en dicho formato.

Forma de invocación

Para realizar las pruebas sobre el protoripo se requiere que la forma de invocarlo sea la siguiente:

```
./tp archivo_entrada archivo_salida
```

La semántica es la misma que la del ejercicio1, pero el archivo de entrada debe ser leído por el módulo cliente, quien debe conectarse al servidor y envíara el archivo a través de un socket TCP. A su vez, el servidor debe aceptar nuevas conexiones y guardar el archivo recibido en disco.

Errores

El reporte de errores es igual que en el ejercicio 1 (al igual que las restricciones sobre el formato S-Registers). Recordemos:

- Se debe validar la no existencia de errores al manejar archivos. Ante cualquier error de manejo de archivos se deberá imprimir, como única leyenda, la cadena "Error de Acceso a Archivos", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 1.
- Se debe validar la correcta recepción de los registros, validando el campo de checksum. Ante cualquier error de checksum se deberá imprimir, como única leyenda, la cadena "Error de Checksum", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 2.
- Se debe validar la existencia y el correcto orden de los registros (recordar ignorar los desconocidos). Header opcional solo al principio, footer solo al final obligatorio. Ante cualquier error detectado, se deberá imprimir, como única leyenda, la cadena "Error de Formato", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 3.
- El programa deberá recibir dos parámetros, el archivo de entrada (archivo de texto) y el archivo de salida (archivo binario), en ese orden. Ante cualquier error detectado, se deberá imprimir, como única leyenda, la cadena "Error de Sintaxis", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 4.
- Luego de una ejecución exitosa, se deberá imprimir, como única leyenda, la cadena "Ejecucion Exitosa", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 0.

Además apareceran los siguientes posibles errores con la inclusión de threads y sockets:

- Se debe validar la no existencia de errores al manejar sockets. Ante cualquier error de manejo de sockets se deberá imprimir, como única leyenda, la cadena "Error de Sockets", seguida de un fin de línea. Asimismo, se debe salir con código de retorno 5.
- Se debe validar que el thread se haya creado con éxito. Ante un error al crear un thread debe imprimirse la leyenda "Error al Crear Thread" y salir con código de retorno 6.

Socket

El primer requisito es encapsular la API de sockets de manera tal de abstraernos del sistema operativo y de conseguir un uso más fácil de los sockets en situaciones simples. Para esto hay que crear una clase Socket que cumpla con la siguiente interfaz:

```
class Socket
{
  public:
    typedef unsigned short port_type;
    // Crea un socket que se conecta al servidor 'host', puerto 'port'
    Socket(const std::string& host, port_type port);
    // Crea un socket que escucha conexiones en el puerto 'port'
    explicit Socket(port_type port, backlog=5);
    // Envía datos por el socket
```

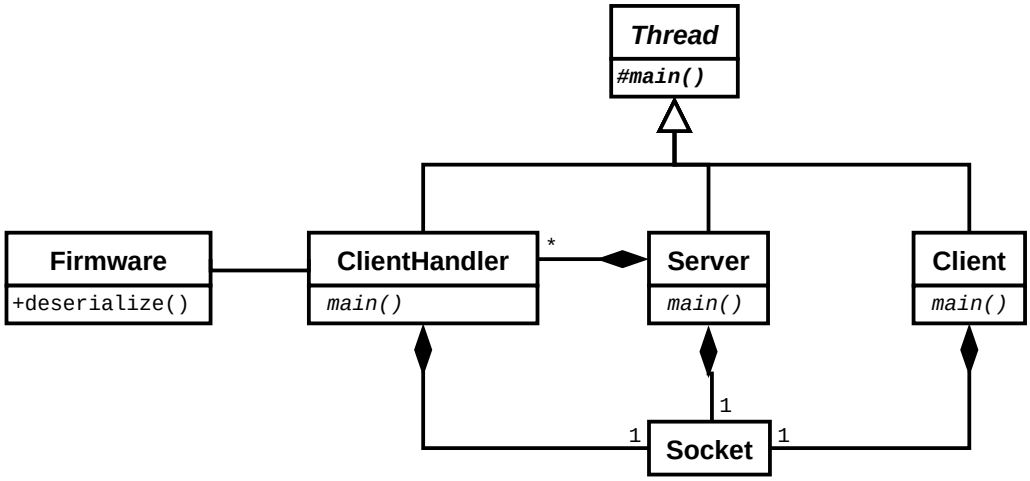

Figura 2: Diagrama simplificado con las clases principales

```
    int send(const std::string& buf);
    // Recibe datos del socket
    int recv(std::string& buf, size_t len);
    // Acepta unañueva conexión
    Socket* accept();
    // Para chequear si el socket es válido (por ej: if (socket) ...)
    operator bool () const;
    // Cierra y libera el socket
    ~Socket();

  private:
    // Constructor de copia y operator= ocultos (para prevenir descuidos)
    Socket(const Socket&);
    Socket& operator=(const Socket&);
    // Crea un socket a partir de un file descriptor (para el accept)
    Socket(int fd);
};
```

Los constructores deben conectar o dejar el socket listo para aceptar conexiones (según sea el caso). Luego de construído hay que chequear siempre si la construcción fue exitosa a través del operador bool (también puede implementarse con excepciones, queda a criterio del alumno, pero no se recomienda a menos que se tenga bien claro el uso de excepciones en constructores). Esta clase no está diseñada para ser heredada (se utiliza por composición o agregación), por lo tanto no tiene destructor virtual.

Esta interfaz es lo mínimo que se pide implementar, se pueden incluir otros métodos de ser necesario pero se recomienda conservarla lo más pequeña posible.

Thread

También se pide encapsular la API de thread de manera tal de que sea multiplataforma y fácil de usar para casos simples. La interfaz mínima requerida es la siguiente:

```
class Thread
```

```
{
  public:
    // Constructor queda a criterio de las subclases.
    // Lanza el thread
    virtual void run();
    // Pide al thread que finalice sus tareas lo antes posible
    virtual void stop();
    // Indica si el thread está corriendo oño
    virtual bool running() const;
    // Espera a que el thread termine realmente
    virtual void join();
    // Espera que el Thread termine y lo libera
    virtual ~Thread();
    // Duerme el thread actual una cierta cantidad de milisegundos
    static void sleep(unsigned long msecs);

  protected:
    // Método virtual puro que deben implementar las subclases
    virtual void main() = 0;

  private:
    // Método estático (función plana) que realmente ejecuta el thread
    // (tomará el puntero this como argumento siempre).
    static void* static_run(void* arg);
    // Constructor de copia y operator= ocultos (para prevenir descuidos)
    Thread(const Thread&);
    Thread& operator=(const Thread&);
};
```

Esta clase debe utilizarse siempre a través de una subclase, ya que es virtual pura. La subclase debe definir el método main(), quien debe encargarse de proveer algún mecanismo para terminar (si hubiera un loop).

Además se debe proveer una clase Mutex, primitiva básica para sincronización:

```
class Mutex
{
  public:
    // Constructor
    Mutex();
    // Obtiene el mutex
    int lock();
    // Libera el mutex
    int unlock();
    // Destruye el mutex
    ~Mutex();

  private:
    // Constructor de copia y operator= ocultos (para prevenir descuidos)
    Mutex(const Mutex&);
    Mutex& operator=(const Mutex&);
};
```

Además se pide implementar un técnica de programación conocida como RAII (Resource Adquisition Is Initialization) que consiste en asegurar que cuando un recurso es adquirido, va a ser liberado automáticamente al finalizar el scope:

```
class Locking
{
  public:
    // Constructor, toma una referencia al objeto a .ªsegurar-
    Lock(Mutex& m);
    // Libera el objeto
    ~Lock();

  private:
    // Constructor de copia y operator= ocultos (para prevenir descuidos)
    Locking(const Locking&);
    Locking& operator=(const Locking&);
};
```

Esta clase nos asegura que todo Mutex que sea adquirido se libere al finalizar el scope, permitiendo además simplificar el uso de mutex. Por ejemplo:

```
int getX()
{
  Locking(mutex);
  return x;
  // Cuando se termine el scope se liberará automáticamente el mutex.
}
```

Si no utilizáramos esta técnica habría que tener una variable temporal y sería mucho más propenso a errores:

```
int getX()
{
  mutex.lock();
  int tmp = x;
  mutex.unlock();
  return tmp;
}
```

Servidor

El servidor debe ser una clase que herede de Thread y que use un Socket. Debe aceptar en el socket nueva conexiones y atender cada una a su vez en un nuevo Thread (clase ClientHandler), de manera de poder seguir aceptando conexiones mientras procesa las conexiones existentes. El contenido del loop que acepta conexiones quedaría algo así:

```
Socket* s = socket.accept();
if (s)
{
  ClientHandler* ch =ñew ClientHandler(s /*, otros parámetros */);
  ch->run();
}
```

(este es, por supuesto, un código incompleto, pseudocódigo si se quiere, sólo para ilustrar la *semántica* del problema)

Además debe separarse en una clase Firmware la funcionalidad propia del firmware. En este caso sólo debe deserializarse (convertir de formato S-Registers a binario), pero en el futuro esta clase también

tendría el método para serializar (convertir de binario a S-Registers). Esta clase debe encapsular el código de conversión programado en el ejercicio 1.

Cliente

El cliente es bastante más simple que el servidor. También debe heredar de Thread y utilizar un Socket, pero todo lo que debe hacer es enviar a través del socket el archivo .s (se presume que los archivos serán convertidos previamente para ahorrar tiempo de procesamiento).

Estructura del programa prototipo

El prototipo no debe ser más complejo que lo siguiente (código ilustrativo):

```
int main(int argc, char* argv[])
{
  // validación de parámetros
  Cliente c(argv[1], "localhost", PUERTO);
  Servidor s(argv[2], PUERTO);
  s.run();
  c.run();
  c.join();
  s.stop();
  s.join();
  if (c.status() != OK)
    return c.status();
  if (s.status() != OK)
    return s.status();
  return 0;
}
```

El puerto a usar queda a elección del alumno, pero deben recordar que los puertos por debajo del 1024 son puertos privilegiados y por lo tanto no deben usarse. También es recomendable no elegir algún puerto que sea utilizado por alguna aplicación con una popularidad considerable.