Práctica 2.2: Serialización y Replicación de Objetos

Objetivos

En esta práctica estudiaremos algunas de las técnicas de serialización de objetos que permiten enviarlos por la red. Además implementaremos una técnica de replicación sencilla de objetos usando un modelo centralizado. Como resultado de esta práctica obtendremos una librería que nos permitirá implementar servidores genéricos.

Contenidos

Preparación del entorno para la práctica

La práctica debe realizarse en el puesto físico del laboratorio, sin necesidad de arrancar ninguna máquina virtual. Las pruebas de funcionamiento de los programas se harán con las IP's configuradas en la máquina del laboratorio, normalmente la dirección de `loopback`.

El código fuente para realizar la práctica está disponible en el siguiente repositorio de github: https://github.com/rsmontero/rvr/archive/refs/tags/release1.0.zip Descomprimir el contenido y cambiar a la carpeta práctica 2.2.

Serialización de Objetos

En primer lugar vamos a definir un interfaz que deberán implementar todos los objetos que se envíen por la red.

Ejercicio 1. Estudiar el archivo Serializable.h proporcionado.

Ejercicio 2. Usando la técnica de serialización binaria definir una clase jugador que implemente el interfaz anterior. La clase Jugador tendrá los siguientes atributos:

Listado 1

```
...
private:
  int16_t pos_x;
  int16_t pos_y;

  char name[MAX_NAME];

  static const size_t MAX_NAME = 20;
...
```

Hacer un program que cree un objeto con valores arbitrarios (p. ej. `Jugador one("player one", 12, 345);`) y los escriba en un fichero. El fichero se manejará con las llamadas open(2), write(2) y close(2). Observar el contenido del fichero, usar el comando od con las opciones -sa (`od -sa data`) ¿qué hace el comando od? ¿qué relación hay entre la salida del comando y la serialización?

Ejercicio 3. Para comprobar el correcto funcionamiento del método que *des-serializa* el objeto

extender el programa para que reconstruya el objeto a partir del fichero y lo muestre por pantalla.

Replicación de Objetos

El objetivo de esta sección es implementar un servidor UDP de mensajería que emplea un mecanismo de replicación de mensajes sencillo. Las características del servidor son:

- **Gestión de la sesión**. Cada cliente establecerá la conexión mediante un mensaje especial de inicio de conexión (LOGIN). El servidor mantendrá un lista de los clientes conectados (p. ej. std::vector). De la misma forma el mensaje LOGOUT eliminará al cliente de la lista.
- **Tratamiento de mensaje.** El thread principal del servidor estará dedicado a recibir mensajes de red, si estos son de tipo MESSAGE se reenviará a todos los clientes registrados (menos el que lo envía).
- **Clientes.** Los clientes estarán representados por su conexión (socket).

El protocolo de aplicación

El servidor de mensajería usará un protocolo de aplicación simple basado en el intercambio de mensajes con la siguiente estructura:

`TIPO` (1 byte, tipo `uint8_t`). 0: LOGIN, 1:MESSAGE, 2:LOGOUT
`NICK` (8 bytes, tipo `char[8]`). El apodo incluye los caracteres de final de cadena '\0'
`MENSAGE` (80 bytes, tipo `char[80]`). El mensaje incluye el carácter final de cadena '\0'

Nota: Se recomienda probar cada ejercicio desarrollando pequeños tests que usen la lógica programada.

Ejercicio 4. Estudiar el archivo `Socket.h` proporcionado y completar la implementación de la clase Socket en Socket.cc. La clase Socket representa una abstracción de los socket UDP del sistema que incluye métodos para enviar y recibir objetos serializables.

```
class Socket
{
public:
    static const int32_t MAX_MESSAGE_SIZE = 32768;

    Socket(const char * address, const char * port);

    Socket(struct sockaddr * _sa, socklen_t _sa_len):sd(-1), sa(*_sa),
        sa_len(_sa_len){};

    virtual ~Socket(){};

    int recv(Serializable &obj, Socket * &sock);

    int send(Serializable& obj, const Socket& sock);
```

```
    int bind()
    {
        return ::bind(sd, (const struct sockaddr *) &sa, sa_len);
    }

    friend std::ostream& operator<<(std::ostream& os, const Socket& dt);

    friend bool operator== (const Socket &s1, const Socket &s2);

protected:

    int sd;

    struct sockaddr sa;
    socklen_t       sa_len;
};
```

Ejercicio 5. Implementar la serialización del mensaje de chat implementado en la clase `ChatMessage`, ver el archivo `Chat.h`. El mensaje simplemente contendrá el tipo, el nick del usuario y el mensaje. Por simplicidad, se asumirá un tamaño fijo para ambos.

Ejercicio 6. Implementar el servidor de chat (`ChatServer`). El servidor tiene un Socket para comunicarse con los clientes y una lista de clientes conectados. El bucle principal que realizará el tratamiento de cada mensaje recibido y se implementará en el método `do_messages()`.

```
class ChatServer
{
public:
    ChatServer(const char * s, const char * p): socket(s, p)
    {
        socket.bind();
    };

    void do_messages();

private:
    std::vector<Socket *> clients;

    Socket socket;
};
```

Ejercicio 7. Implementar el cliente de chat (`ChatClient`) que usará dos threads uno para leer de la entrada estándar y enviar los mensajes; y otro que recibirá e imprimirá los mensajes (menos los suyos).

```
class ChatClient
{
public:
    ChatClient(const char * s, const char * p, const char * n):socket(s, p),
        nick(n){};

    void login();
```

```
    void logout();

    void input_thread();

    void net_thread();

private:
    Socket socket;

    std::string nick;
};
```

Ejercicio 8. Comprobar el funcionamiento del servidor y clientes. El proyecto se puede compilar con el archivo Makefile proporcionado.