



UNIVERSITAT
POLITÈCNICA
DE VALÈNCIA



Actividad 8

Servidor y Cliente TCP/IP

Sistemas Informáticos Industriales (SII₁₂₁₆₄)

Contenido

1. [Introducción](#)
2. [Objetivo](#)
3. [Conceptos](#)
 - [Pila de Protocolos TCP/IP](#)
 - [Protocolos IP e ICMP](#)
 - [Protocolos de Transporte \(UDP, TCP\)](#)
 - [Arquitectura Cliente-Servidor](#)
 - [Clases de Qt para aplicaciones TCP/IP](#)
4. [Ejemplos](#)
 - [Ejemplo 1 – Cliente TCP – Conexión a la web de la UPV](#)
 - [Ejemplo 2 – Servidor TCP – Servidor de Números Aleatorios](#)
5. [Actividades](#)
 - [Actividad 1 – Cliente de Números Aleatorios](#)
 - [Actividad 2 – Servidor de Números Aleatorios v.2](#)
 - [Actividad 3 – Servidor de Fecha y Hora](#)
6. [Actividad del Proyecto](#)
7. [Criterios de Evaluación](#)

Introducción

En esta práctica veremos cómo comunicar ordenadores a través de la red utilizando el protocolo TCP/IP

Programaremos un servidor TCP que presta un servicio de generación de números aleatorios. Programaremos también un cliente TCP que se conecta a dicho servidor

Para que el cliente y servidor se comuniquen, definiremos un protocolo. La versión inicial de dicho protocolo será sencilla. Después definiremos nuevas versiones que incorporen servicios adicionales

Objetivo

1. Conocer los conceptos básicos del protocolo TCP/IP
2. Implementar una aplicación distribuida utilizando el protocolo TCP:
 - Implementar en Qt un servidor TCP
 - Implementar en Qt un cliente TCP

Conceptos

Videos

- Os recomiendo que veáis previamente los siguientes videos sobre redes:
 - <https://media.upv.es/#/portal/video/8e8edc2c-cede-4cd8-b8f3-c04851ede114>
 - <https://media.upv.es/#/portal/video/981d4bb0-021e-11e6-851a-656f7e06a374>
 - <https://youtu.be/4YCqius0CiQ>

Pila de Protocolos TCP/IP

TCP/IP es la pila de protocolos utilizada en Internet para establecer la comunicación entre diferentes máquinas

En esta pila intervienen principalmente los siguientes protocolos:

A nivel de red:

- IP
- ICMP

A nivel de transporte:

- UDP
- TCP

Pila de protocolos TCP/IP

Niveles	Protocolos	
Aplicación	(Web, Whatsapp, ...)	
Sockets API del sistema operativo. Interfaz entre en nivel de transporte y el de aplicación		
Transporte	TCP	UDP
Red	IP <div>ICMP</div>	
Enlace	Redes locales	
físico	Emisión de señales (por cable, WIFI, fibra óptica...)	

Protocolos IP e ICMP

IP es un protocolo a nivel de red que se encarga de transmitir y encaminar un paquete de datos (llamado *datagrama*) desde un origen hasta un destino en Internet

Existen dos versiones de IP:

- IPv4 (RFC 791): IP versión 4, es la versión original. Utiliza direcciones IP de 32 bits, normalmente representados de la forma: *192.168.0.3* (donde cada número es un byte)
- IPv6 (RFC 2460): IP versión 6. Diseñada para reemplazar a IPv4, utiliza direcciones IP de 128 bits, representados con una notación similar a: *2a00:1450:4003:806::200e*

IP se encarga sólo de la tarea de conseguir alcanzar el destino de un datagrama. Si IP detecta un error en un datagrama lo descarta y, si es posible, informa del error al dispositivo origen del datagrama, mediante el protocolo ICMP (RFC 792)

() RFC - Request for Comments. En el contexto del Internet Governance, son publicaciones del Internet Engineering Task Force (IETF) y del Internet Society (ISOC), los principales agentes de estandarización de Internet*

Protocolos de Transporte

Se encargan de identificar las aplicaciones que envían y/o transmiten un datagrama dentro de un dispositivo. Para realizar dicha identificación, TCP/IP utiliza los puertos, que son identificadores de 16 bits

Existen dos protocolos a nivel de transporte sobre IP

- UDP: Basado en la transmisión de bloques de bytes, también llamados datagramas
- TCP: Basado en la transmisión de un *stream* de bytes

UDP (RFC 768)

Ofrece el servicio de transmisión de un datagrama desde una aplicación origen a una aplicación destino, normalmente en dispositivos diferentes

El servicio no ofrece fiabilidad ni control de flujo. Es un servicio de tipo *Send and Pray* (enviar y rezar), ya que, a lo largo de la ruta hacia el destino, los datagramas pueden ser eliminados, duplicados, e incluso pueden llegar al destino en un orden temporal diferente a como se han emitido desde el dispositivo origen

En UDP no hay conexiones. El servidor responderá a todos los datagramas que reciba, independientemente de quién sea el otro extremo

TCP (RFC 793)

Ofrece un servicio de comunicación fiable con control de flujo a nivel de aplicación

La base de TCP es la transmisión de un *stream* de bytes desde una aplicación origen a una aplicación destino, normalmente, alojadas en máquinas diferentes en Internet. Por lo que es necesario establecer previamente una **conexión** entre las dos aplicaciones. Una vez establecida la conexión, la comunicación es bidireccional, por lo que TCP maneja dos *streams* por conexión, uno de cliente a servidor, y otro de servidor a cliente

TCP ofrece **fiabilidad**. Nos asegura que los datos son entregados en orden y sin duplicados a la aplicación destino en forma de un *stream* de bytes. Si no es posible, TCP informa del error a la aplicación

TCP ofrece **control de flujo**. Controla que la velocidad a la que la aplicación origen envía los bytes sea igual o inferior a la velocidad que la aplicación en el destino va consumiendo los datos

Cualquier imprevisto, como la pérdida de segmentos TCP o los duplicados son resueltos por TCP de forma transparente para las aplicaciones. Si TCP puede resolver un error, la comunicación no se interrumpe y las aplicaciones no se enteran de lo que haya podido pasar. Sólo pueden experimentar una ralentización de la conexión, pero los datos seguirán llegando

Los puertos en TCP y UDP

Los puertos son el mecanismo utilizado para identificar a las diferentes aplicaciones que utilizan TCP o UDP dentro de un dispositivo

Un puerto es un número de 16 bits (de 0 a 65535)

Los primeros 1024 puertos están reservados para servicios bien conocidos (como por ejemplo HTTP - Hypertext Transfer Protocol, en el puerto 80)

Los puertos UDP son independientes de los puertos TCP, por lo que un mismo identificador de puerto en TCP hace referencia a una aplicación distinta al mismo puerto en UDP

Se llama **endpoint** al par formado por dirección IP y puerto. Un *endpoint* identifica a una aplicación

Ejemplo: El servidor web de la UPV se encuentra en el *endpoint* 158.42.4.23:80

Nota: Una misma aplicación puede abrir varios *endpoints*. Por ejemplo, un navegador puede abrir varias páginas webs, para cada servidor web usará uno o varios *endpoints* diferentes

En TCP una conexión se identifica por la pareja de *endpoints* de las dos aplicaciones que estén conectadas. TCP permite, por lo tanto, gestionar varias conexiones a un mismo *endpoint*, siempre que el otro extremo de la conexión sea un *endpoint* distinto

Ejemplo: (158.42.20.20:**45432**, 158.42.4.23:80) y (158.42.20.20:**333**, 158.42.4.23:80) en TCP son dos conexiones completamente distintas, aunque sólo cambia el puerto del primer *endpoint* (marcado en negrita)

Sockets

La interfaz (la API - Application Programming Interface) que los sistemas operativos ofrecen a las aplicaciones para acceder a los protocolos TCP/IP son los **sockets**

Existen dos tipos distintos de sockets, uno para TCP y otro para UDP. Como existen diferencias en las APIs de los sockets según el sistema operativo, los lenguajes de programación de alto nivel implementan su propia API basada en *sockets* para simplificar la tarea al programador

En Qt usaremos las clases **QTcpServer** y **QTcpSocket**, para implementar un servidor y un cliente

Arquitectura Cliente-Servidor

Existen dos tipos principales de arquitecturas para aplicaciones en red:

- **La arquitectura Cliente-Servidor.** En este tipo de aplicaciones en red hay, al menos, un dispositivo que está siempre en funcionamiento, esperando que el resto de dispositivos establezcan comunicación con él. Este dispositivo es el **servidor**. Un servidor ofrece un servicio a los clientes que se conecten a él
 - Por ejemplo, un servidor web se encarga de entregar los objetos que los navegadores web le piden, generalmente páginas web
- **La arquitectura Peer-to-peer.** Todos los dispositivos ofrecen los mismos servicios al resto de dispositivos y todos ellos pueden hacer uso de dichos servicios, de ahí que no hablemos de servidores ni de clientes, sino de **pares** (o **peers**)
 - El tipo de aplicación más utilizada de este tipo son las aplicaciones de compartición de archivos, como es el caso de BitTorrent. A medida que un *peer* va descargándose bloques de un archivo, este *peer* puede enviar aquellos bloques que ya tenga completamente descargados a otros *peers* que se estén descargando el mismo archivo

(*) En esta práctica crearemos aplicaciones de tipo Cliente-Servidor

Clases de Qt para aplicaciones TCP

- QTcpServer

Esta clase se utiliza para crear un servidor TCP. Un objeto de esta clase abre un puerto al cual se podrán conectar los clientes. La tarea principal de QTcpServer será escuchar en dicho puerto, esperando peticiones de conexión.

- QTcpSocket

Esta clase implementa la comunicación en forma de *stream* de bytes entre la aplicación actual y la aplicación a la que se encuentre conectada

- Los clientes utilizan un objeto de esta clase para establecer una conexión con un servidor
- Los servidores crean un objeto de esta clase por cada conexión entrante establecida

- QHostAddress

Esta clase se utiliza para definir direcciones IP, tanto en IPv4 como en IPv6

- quint16

Los puertos son números enteros sin signo de 16 bits

Servidor TCP: QTcpServer (1/3)

Para crear un servidor hemos de crear un objeto de la clase QTcpServer:

```
QTcpServer *servidorTCP = new QTcpServer();
```

Tened en cuenta que crear un objeto de tipo *servidorTCP* no abre ningún puerto TCP, por lo que el servidor aún no está en funcionamiento

Para poner en marcha el servidor hemos de invocar el método *listen()*:

```
servidorTCP->listen(QHostAddress::AnyIPv4, 5000);
```

La sentencia anterior pone en marcha el servidor TCP, escuchando todas las interfaces de red mediante el protocolo IPv4, en el puerto TCP 5000

El servidor funciona en segundo plano, por lo que el programa continuará a la espera, lleguen o no peticiones de conexión

Servidor TCP: QTcpServer (2/3)

QTcpServer gestiona en segundo plano una cola de conexiones entrantes. Cada vez que un cliente establezca con éxito una conexión a este servidor, dicha conexión se añadirá a la cola

Para saber si el servidor tiene conexiones entrantes en la cola se usa el método:

```
servidorTCP->hasPendingConnections()
```

que devolverá **true** si hay una o más peticiones de conexión en la cola

La señal **newConnection()**:

Un objeto de tipo *QTcpServer* emitirá una señal *newConnection()* cada vez que se añada un cliente a la cola de conexiones pendientes

Servidor TCP: QTcpServer (3/3)

Para atender a cada uno de los clientes que se conecten al servidor, es necesario crear un *socket* conectado a dicho cliente. QTcpServer nos crea dicho *socket*, ya conectado, con el método:

```
QTcpSocket *socket = server->nextPendingConnection();
```

Este método nos devuelve un *socket* conectado al primer cliente de la cola de conexiones entrantes, y lo quita de la cola

Más adelante veremos cómo podemos usar el objeto de tipo *QTcpSocket* para enviar y recibir datos

Cliente TCP: QTcpSocket

Para que un cliente pueda establecer la conexión con un servidor se usa directamente la clase QTcpSocket

Las siguientes sentencias establecen una conexión con el servidor web de la UPV:

```
QTcpSocket *socketCliente = new QTcpSocket();  
socketCliente->connectToHost("www.upv.es", 80);
```

QTcpSocket

La clase **QTcpSocket** se utiliza para enviar y recibir datos a través de una conexión TCP. Es una clase derivada de la clase *QIODevice* de la cual hereda los métodos de lectura y de escritura de datos. La forma de gestionar los *stream* de datos que se envían y que se reciben, es mediante búferes

Métodos de lectura (para recibir datos):

read, readData, readLine, readAll, ...

Métodos de escritura (para enviar datos):

write, writeData, ...

Métodos para conocer los parámetros de la conexión:

localAddress, localPort, peerAddress, peerName, peerPort

Otros métodos: *bytesAvailable, canReadLine, flush*

Señal **readyRead()**:

Cada vez que un objeto de tipo QTcpSocket recibe datos nuevos se emite la señal readyRead(). Los datos nuevos recibidos son almacenados en el búfer de recepción esperando a que el programa invoque algún método de lectura de datos

Ejemplos

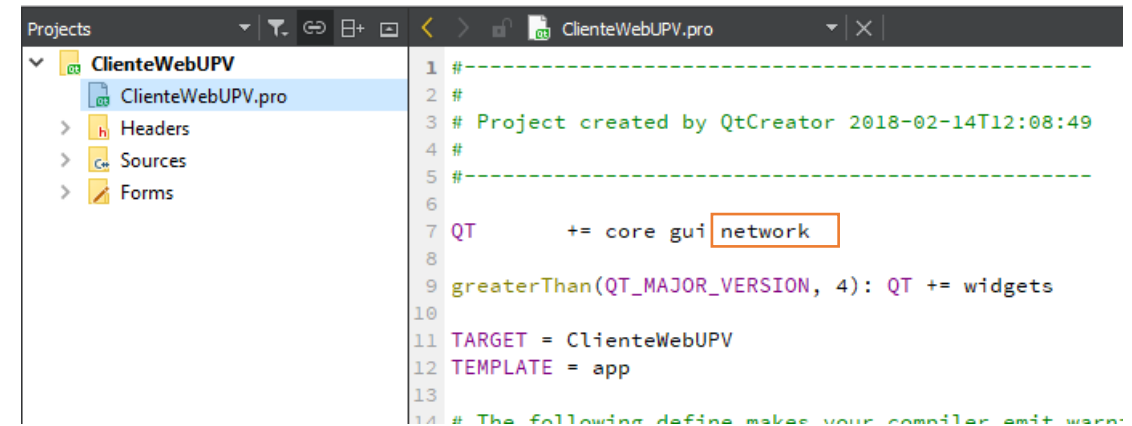
Ejemplo 1

Cliente TCP

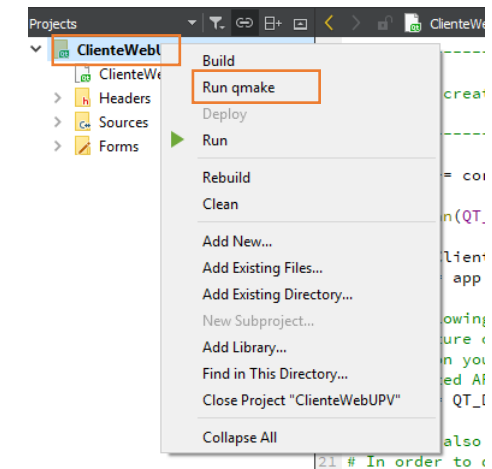
Conexión a la web de la UPV

Ejemplo 1: Creando un cliente TCP (1/6)

- Cread un nuevo proyecto de tipo *Qt Widgets Application*
 - Llamadlo “ClienteWebUPV”
- Editad el archivo “ClienteWebUPV.pro”
 - Añadid el símbolo “network” a la línea “QT += core gui”
(ver rectángulo rojo en imagen de arriba)
- Seleccionad el nombre del proyecto a la izquierda con el botón derecho del ratón y pulsad “run qmake”
(ver rectángulos rojos en imagen de abajo)

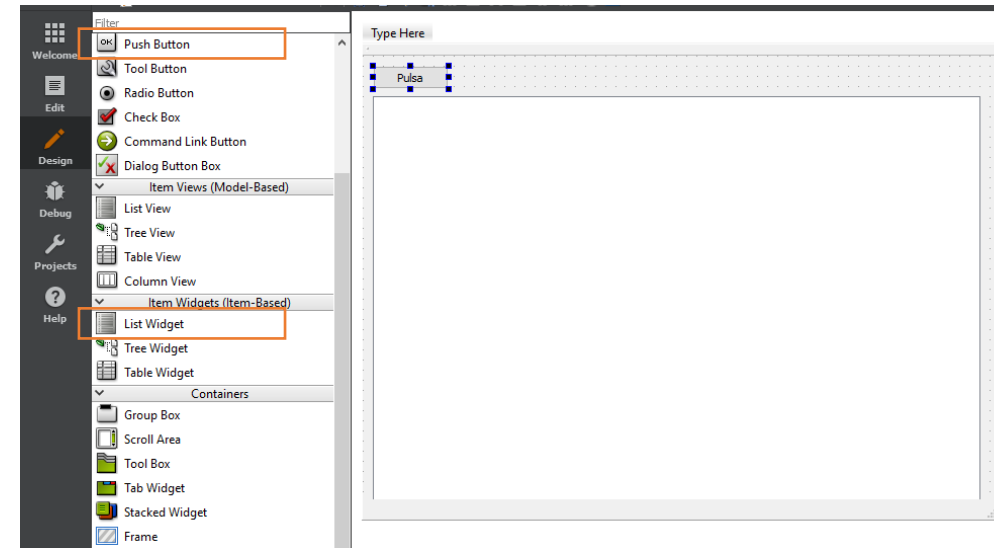


```
1 #-----
2 #
3 # Project created by QtCreator 2018-02-14T12:08:49
4 #
5 #-----
6
7 QT      += core gui network
8
9 greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
10
11 TARGET = ClienteWebUPV
12 TEMPLATE = app
13
14 # The following defines make your compiler emit warn
```



Ejemplo 1: Creando un cliente TCP (2/6)

- Añadid un botón “Push Button” y un “List Widget” a la ventana principal
 - El objetivo es que cuando pulsemos el botón, el programa enviará una petición a la web de la UPV, y mostrará en el “List Widget” los datos recibidos
- Ahora vamos a conectar la señal “clicked()” del botón *PushButton* a un slot utilizando la ayuda que Qt nos ofrece para ello:
 - Sobre el botón que hemos añadido, pulsad con el botón derecho del ratón y seleccionad en el menú: “go to slot...”
 - Seleccionad “Clicked()” y pulsad “OK”
 - Nos genera automáticamente un nuevo *slot* en la clase *MainWindow* llamado “on_pushButton_clicked()” y lo conecta a la señal “clicked()”, por lo que no tenemos que programar manualmente la llamada a “connect” para esta conexión entre la señal y el *slot*



Ejemplo 1: Creando un cliente TCP (3/6)

- Editad el archivo “mainwindow.h” y añadid un nuevo *slot*:
`void datosDisponibles();`

Añadid también: `#include <QTcpSocket>` al principio del archivo

- Añadid también, en la zona privada de la clase MainWindow, un puntero a un objeto de la clase QTcpSocket
- Editad el archivo “mainwindow.cpp” y añadid el slot anterior como un nuevo método

En la imagen inferior podéis ver el contenido de “mainwindow.cpp” antes de empezar a programar los slots “datosDisponibles()” y “on_pushButton_clicked()”

```
void MainWindow::datosDisponibles()
{

}

void MainWindow::on_pushButton_clicked()
{

}
```



```
mainwindow.h
MainWindow

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpSocket>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = nullptr);
    ~MainWindow();

private slots:
    void on_pushButton_clicked();

    void datosDisponibles();

private:
    Ui::MainWindow *ui;
    QTcpSocket *socket;
};

#endif // MAINWINDOW_H
```

Ejemplo 1: Creando un cliente TCP (4/6)

- Editad el método “on_pushButton_clicked()”

```
void MainWindow::on_pushButton_clicked()
{
    ui->listWidget->clear();

    socket = new QTcpSocket();

    if (!socket) return; // Si no se puede crear un socket termina el método.

    // Conectamos el socket para que active el SLOT "datosDisponibles" cada vez que reciba datos.
    connect(socket, SIGNAL(readyRead()), this, SLOT(datosDisponibles()));

    // Conectamos el objeto socket para que sea destruido cuando se cierre la conexión.
    connect(socket, SIGNAL(disconnected()), socket, SLOT(deleteLater()));

    // Establecemos la conexión
    socket->connectToHost("www.upv.es", 80);

    if (socket->isOpen())
    {
        // Preparamos la petición web, usando el protocolo HTTP 1.0
        QString str = "GET / HTTP/1.0\r\n";
        str += "host: www.upv.es\r\n";
        str += "Connection: close\r\n\r\n";

        //Esperamos a que se establezca la conexión.
        socket->waitForConnected();

        //Enviamos el string en formato utf8 (compatible con el servidor web)
        socket->write(str.toUtf8());

        // Para asegurarnos que la petición se envía inmediatamente invocamos el método flush.
        socket->flush();
    }
}
```

Ejemplo 1: Creando un cliente TCP (5/6)

- Editad el método “datosDisponibles()”

```
void MainWindow::datosDisponibles()
{
    // "sender()" nos permite manejar varias conexiones TCP en paralelo con un único slot.
    // Aunque en este ejemplo no usemos varias conexiones, viene bien habituarse a su uso.
    QTcpSocket *socket = static_cast<QTcpSocket *>(sender());

    // Bucle para leer todas las líneas recibidas en un único segmento TCP
    while (socket->canReadLine())
    {
        // Leemos una línea (readLine) y eliminamos los espacios en blanco
        // al principio y al final (trimmed)
        QByteArray buffer = socket->readLine().trimmed();

        // Convertimos el Array de Bytes en QString, suponiendo que el formato
        // de caracteres es utf8
        QString linea = QString::fromUtf8(buffer);

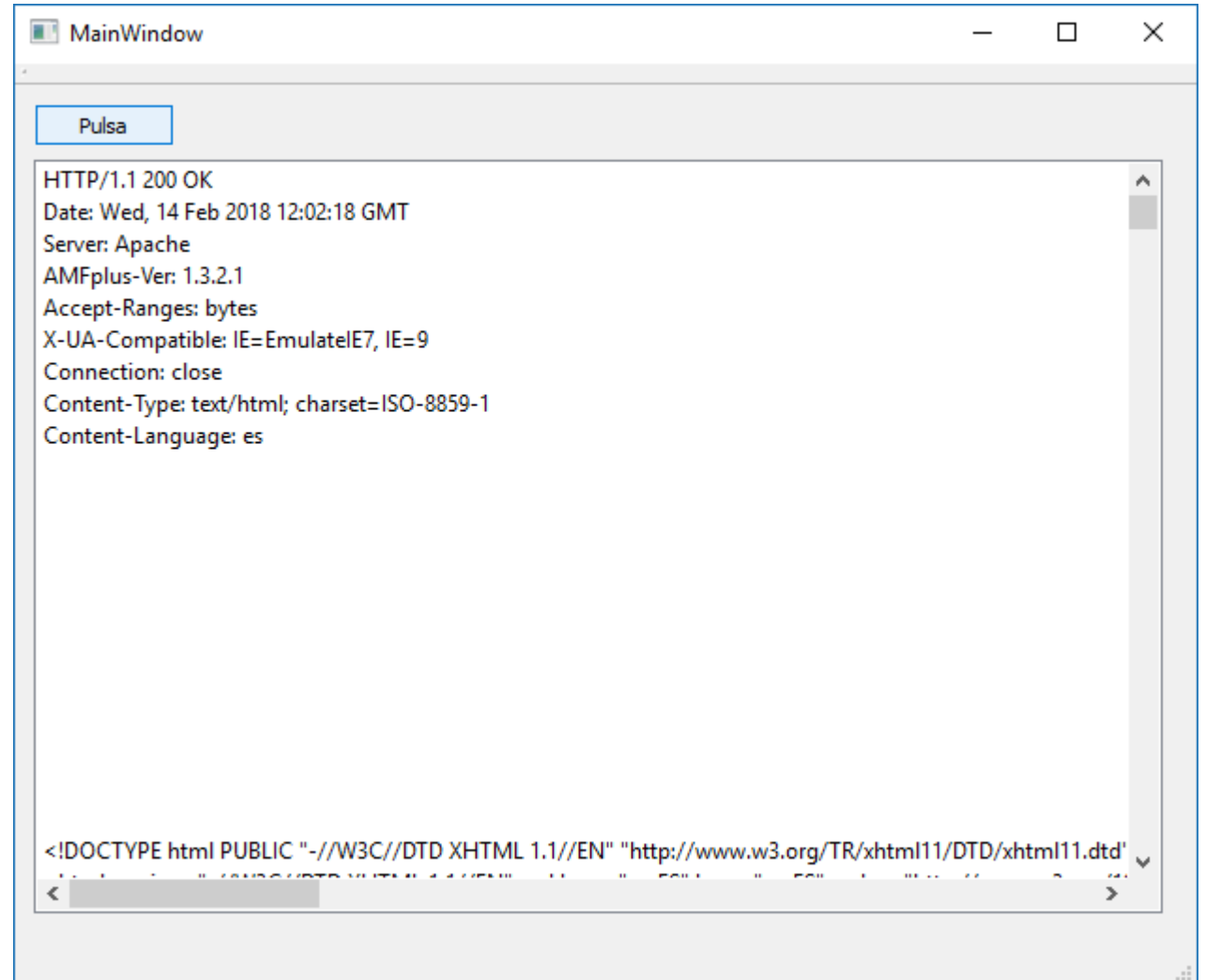
        // Añadimos la línea de texto a listWidget
        ui->listWidget->addItem(linea);
    }
}
```

Ejemplo 1: Creando un cliente TCP (6/6)

- Compilad y ejecutad el programa:

Si todo va bien, al pulsar el botón veremos la respuesta que el servidor web de la UPV envía a los navegadores web

Lo que realmente estamos viendo es el protocolo HTTP/1.1, un espacio en blanco y el documento principal de la página web



Ejemplo 2

Servidor TCP

Servidor de Números Aleatorios

Ejemplo 2: Servidor de números aleatorios (1/23)

Funciones de un servidor

- Esperar conexiones de clientes

El servidor abre, al menos, un puerto TCP, a través del cual se mantiene en escucha a la espera de recibir peticiones de conexión por parte de algún cliente

- Atender una conexión establecida

Cuando el servidor acepte una conexión entrante, deberá establecerse un diálogo entre servidor y cliente. A las reglas que definen ese diálogo se le llama **protocolo**

Los servidores deben tener la capacidad de atender a varios clientes a la vez

Para implementar dichas funciones, vamos a definir dos *clases* diferentes

Para nuestro ejemplo dichas clases serán **RandomServer** y **RandomHandler**

- *RandomServer* se encargará de aceptar las conexiones entrantes
- *RandomHandler* se encargará de atender a un cliente. Ésta es la clase que implementa el protocolo entre cliente y servidor

Ejemplo 2: Servidor de números aleatorios (2/23)

Protocolo

Define los mensajes que van a intercambiar el cliente y el servidor. Podría decirse que un *protocolo* es el algoritmo de una aplicación en red

En una aplicación cliente-servidor, el protocolo se centra en la “conversación” entre un único cliente y el servidor (aunque el servidor atienda a varios clientes simultáneamente)

Protocolo del servidor *RandomServer*

Todos los datos se enviarán como cadena de caracteres (string, char *) en formato **utf8**. Los números deberán convertirse a su representación en base 10

Una vez establecida la conexión entre un cliente y el servidor, se deben seguir los siguientes pasos:

1. El servidor envía una línea indicando el nombre del servicio y la versión:
`RandomServer/1.0`
El servidor permanecerá a la espera de que el cliente le haga una petición. La versión 1.0 sólo implementa una función que consiste en enviar un número aleatorio
2. El cliente enviará la palabra “RND” (o “rnd”) con un salto de línea al final: “RND\r\n”
3. Si el servidor recibe la petición correcta, responderá enviando un número aleatorio en base 10 en formato *utf8*. Con un salto de línea al final
Ejemplo: 17480\r\n
4. Si el servidor recibe una petición incorrecta, responderá enviando un mensaje de error: “ERROR 500 Bad request\r\n”
5. Repetir indefinidamente los pasos del 2 al 4 hasta que se corte la conexión

El servidor debe permitir que la línea enviada por el cliente tenga espacios en blanco antes y después de la palabra “RND”

La clase QString implementa la función *trimmed()* que elimina todos los caracteres considerados espacios en blanco que la cadena tenga al principio y al final. Los espacios en blanco que estén entremedias no son alterados. (Entre otros, son considerados espacios en blanco los caracteres: ' ', '\t', '\r', '\n' (espacio, tabulador, retorno de carro, salto de línea))

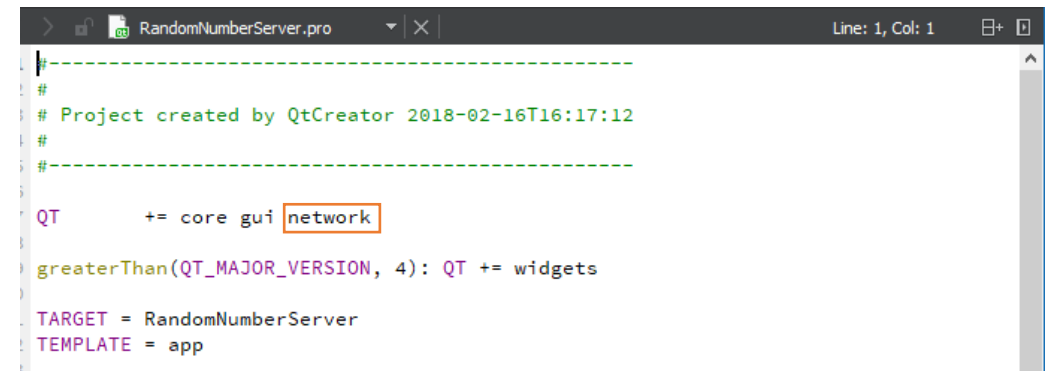
Las siguientes líneas de ejemplo son correctas (las comillas no serán enviadas por el cliente, se han incluido en los ejemplos para ver donde empieza y donde acaba la línea):

- “ RND \r\n”
- “rnd \r\n”
- “ rnd\r\n”

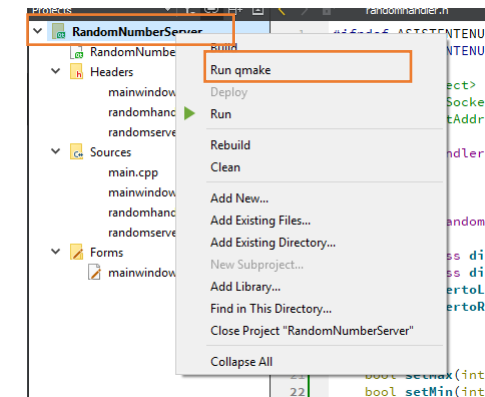
Ejemplo 2: Servidor de números aleatorios (3/23)

En nuestro ejemplo vamos a crear un servidor que enviará al cliente un número generado aleatoriamente, al que llamaremos **RandomNumberServer**

- Cread un nuevo proyecto de tipo *Qt Widgets Application*
 - Llamadlo “RandomNumberServer”
- Editad el archivo “RandomNumberServer.pro”
 - Añadid el símbolo “network” a la línea “QT += core gui” (ver en la imagen)
- Seleccionad el nombre del proyecto a la izquierda con el botón derecho del ratón y pulsad “run qmake”

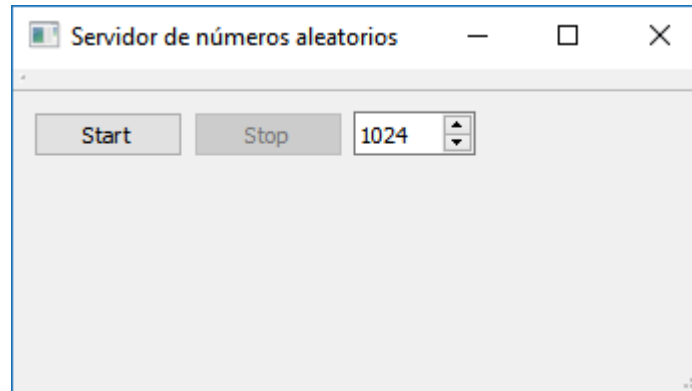


```
#-----  
# Project created by QtCreator 2018-02-16T16:17:12  
#-----  
  
QT       += core gui network  
  
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets  
  
TARGET = RandomNumberServer  
TEMPLATE = app
```



Ejemplo 2: Servidor de números aleatorios (4/23)

- Diseñad una ventana para la aplicación que sea similar a la siguiente:



Los nombres de los botones elegidos para el ejemplo son: ***pushButton_start***, ***pushButton_stop***
El nombre elegido para el spinBox es: ***spinBox_puerto***

Fijaos en que el botón ***pushButton_stop*** empieza con la propiedad “enabled” a “false”

Los límites del ***spinBox_puerto*** deben comprender el rango [1024, 65535]

En este ejemplo vamos a utilizar el método ***qDebug()*** para hacer un seguimiento de la actividad del servidor, por lo que los mensajes aparecerán en la ventana *Application Output*, que está en la parte inferior del entorno de programación *Qt Creator*

Ejemplo 2: Servidor de números aleatorios (5/23)

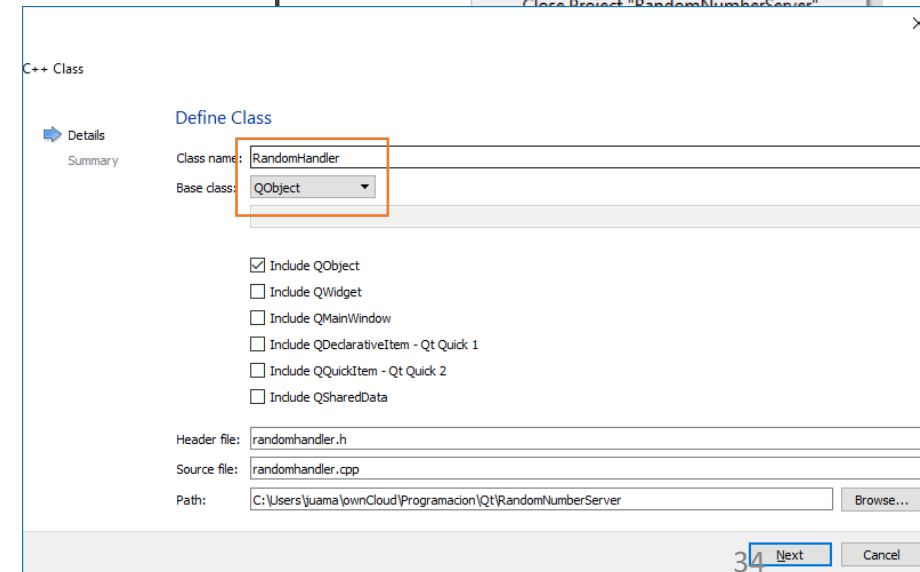
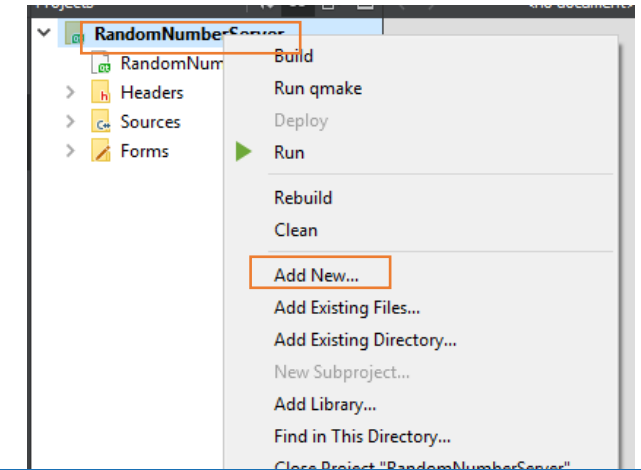
- Comencemos por la clase **RandomHandler**

Vamos a suponer que cada vez que un cliente se conecta al servidor, el servidor crea un objeto de la clase *RandomHandler*

- Cread una nueva clase para este proyecto llamada RandomHandler con el asistente de Qt Creator

Ver imagen (Botón derecho del ratón sobre el nombre del proyecto → Add New...)

- No olvidéis indicar que la clase base es *QObject*



Ejemplo 2: Servidor de números aleatorios (6/23)

- Vamos a editar el archivo “randomhandler.h”

1. Añadiremos un parámetro al constructor:

El constructor recibirá un socket TCP conectado a un cliente. Eso lo veremos más adelante

```
explicit RandomHandler(QTcpSocket *socket, QObject *parent = nullptr);
```

2. Añadiremos al final del archivo, una zona privada:

- Definiremos un campo para almacenar la referencia al objeto de tipo *QTcpSocket* que reciba el constructor
- Definiremos los métodos y los campos que usaremos para generar números aleatorios:
 - Los campos ***_min*** y ***_max***, de tipo entero, que guardarán el rango dentro del cual se generará el número aleatorio. (El número aleatorio generado ***x***, cumplirá la condición ***_min ≤ x < _max***)
 - Una función para inicializar el generador de números aleatorios y otra función para obtener el siguiente número aleatorio

```
private:
    QTcpSocket *_socket;
    int _max = RAND_MAX;
    int _min = 0;

    void inicializa_qrand();
    int rand();
};

#endif // ASISTENTENUMEROSALEATORIOS_H
```

Ejemplo 2: Servidor de números aleatorios (7/23)

- Seguimos editando el archivo “randomhandler.h”
 1. Crearemos la zona pública de nuestra clase con los siguientes métodos (hemos incluido el constructor):

```
class RandomHandler : public QObject
{
    Q_OBJECT
public:
    explicit RandomHandler(QTcpSocket *socket, QObject *parent = nullptr);

    // Métodos del socket TCP
    QHostAddress direccionIpLocal();
    QHostAddress direccionIpRemota();
    quint16 puertoLocal();
    quint16 puertoRemoto();
    bool estaAbierto();
    bool cerrar();

    // Métodos del campo _max
    int max();
    bool setMax(int m);

    // Métodos del campo _min
    int min();
    bool setMin(int m);
};
```

2. Crearemos una *señal* para informar cuando se cierre la conexión
3. Crearemos un *slot* para leer los datos que nos envía el cliente

```
signals:
    void desconectado();

protected slots:
    void datosDisponibles();
```

Ejemplo 2: Servidor de números aleatorios (8/23)

- Así queda “randomhandler.h”

Observad que al principio de este archivo se encuentran tres líneas *#include* con las clases que va a utilizar este objeto

```
1  #ifndef ASISTENTENUMEROSALEATORIOS_H
2  #define ASISTENTENUMEROSALEATORIOS_H
3
4  #include <QObject>
5  #include <QTcpSocket>
6  #include <QHostAddress>
7
8  class RandomHandler : public QObject
9  {
10     Q_OBJECT
11 public:
12     explicit RandomHandler(QTcpSocket *socket, QObject *parent = nullptr);
13
14     // Métodos del socket TCP
15     QHostAddress direccionIpLocal();
16     QHostAddress direccionIpRemota();
17     quint16 puertoLocal();
18     quint16 puertoRemoto();
19     bool estaAbierto();
20     bool cerrar();
21
22     // Métodos del campo _max
23     int max();
24     bool setMax(int m);
25
26     // Métodos del campo _min
27     int min();
28     bool setMin(int m);
29
30
31 signals:
32     void desconectado();
33
34 protected slots:
35     void datosDisponibles();
36
37 private:
38     QTcpSocket *_socket;
39     int _max = RAND_MAX;
40     int _min = 0;
41
42     void inicializa_qrand();
43     int rand();
44 };
45
46 #endif // ASISTENTENUMEROSALEATORIOS_H
47
```

Ejemplo 2: Servidor de números aleatorios (9/23)

- Editamos el archivo “randomhandler.cpp”
 1. Programamos el constructor:

Algunos de los métodos que usa el constructor aún no los hemos programado. Lo haremos más adelante
 2. Añadid también `#include <QDateTime>`, ya que esta clase se usará para inicializar el generador de números aleatorios

```
> randomhandler.cpp # RandomHandler::RandomHandler(QTcpSocket *, QObject *) Line: 24, Col: 70
1  #include "randomhandler.h"
2  #include <QDateTime>
3
4  RandomHandler::RandomHandler(QTcpSocket *socket, QObject *parent) : QObject(parent)
5  {
6      _socket = socket;
7
8      if (estaAbierto())
9      {
10         // Conectamos la señal al slot para leer datos cuando lleguen al socket.
11         connect(socket, SIGNAL(readyRead()), this, SLOT(datosDisponibles()));
12
13         // Conectamos las señales para emitir la señal "desconectado()" cuando se cierre el socket.
14         connect(socket, SIGNAL(disconnected()), this, SIGNAL(desconectado()));
15
16         // Inicializamos el generador de números aleatorios
17         inicializa_grand();
18
19         // Enviamos el primer mensaje del protocolo
20         QString saludo = "RandomServer/1.0\r\n";
21         _socket->write(saludo.toUtf8());
22         _socket->flush();
23
24         // Escribimos un mensaje de depuración en la salida estándar.
25         qDebug().noquote() << "Enviado al cliente:" << saludo.trimmed();
26     }
27 }
```

Ejemplo 2: Servidor de números aleatorios (10/23)

- “randomhandler.cpp”

Programamos los métodos del *socket*:

Los que extraen información del socket y el método para cerrar la conexión

```
28
29  QHostAddress RandomHandler::direccionIpLocal()
30  {
31      return _socket->localAddress();
32  }
33  QHostAddress RandomHandler::direccionIpRemota()
34  {
35      return _socket->peerAddress();
36  }
37  quint16 RandomHandler::puertoLocal()
38  {
39      return _socket->localPort();
40  }
41  quint16 RandomHandler::puertoRemoto()
42  {
43      return _socket->peerPort();
44  }
45  |
46  bool RandomHandler::estaAbierto()
47  {
48      return _socket && _socket->isOpen();
49  }
50
51  bool RandomHandler::cerrar()
52  {
53      if (!_socket) return false;
54      _socket->close();
55      return true;
56  }
57
```

Ejemplo 2: Servidor de números aleatorios (11/23)

- “randomhandler.cpp”

Programamos los métodos que encapsulan el campo `_max`

```
58
59 ▼ int RandomHandler::max()
60 {
61     return _max;
62 }
63
64 ▼ bool RandomHandler::setMax(int m)
65 {
66 ▼     if (m > min() && m <= RAND_MAX)
67     {
68         _max = m;
69         return true;
70     }
71     return false;
72 }
73
```

Programamos los métodos que encapsulan el campo `_min`

```
73 |
74 ▼ int RandomHandler::min()
75 {
76     return _min;
77 }
78 |
79 ▼ bool RandomHandler::setMin(int m)
80 {
81 ▼     if (m < max() && m >= 0)
82     {
83         _min = m;
84         return true;
85     }
86     return false;
87 }
88
```


Ejemplo 2: Servidor de números aleatorios (12/23)

- “randomhandler.cpp”

Programamos los métodos privados que generan números aleatorios

- Para inicializar el generador de números aleatorios utilizamos el día y la hora actuales (en milisegundos desde el 1 de enero de 1970). Utilizamos dicho número como semilla para el generador de números aleatorios de Qt
- El método *rand()* genera un número aleatorio x tal que $_{min} \leq x < _{max}$

```
118
119 void RandomHandler::inicializa_grand()
120 {
121     uint aux = (uint)QDateTime::currentDateTimeUtc().toMSecsSinceEpoch();
122     qsrand(aux);
123 }
124
125 int RandomHandler::rand()
126 {
127     return (qrand() % (_max - _min)) + _min;
128 }
129
```

Nota : Los generadores de números aleatorios de los lenguajes de programación se basan en generar una secuencia de números de gran tamaño (por ejemplo, de 64 bits, o de 128 bits). Al último número de gran tamaño generado se le llama semilla. El número que la función **qrand()** devuelve son sólo unos pocos bits de la semilla, por ejemplo los 16 bits de menor peso. La función **qrand()** calcula la siguiente semilla con un algoritmo que cumpla que, estadísticamente, la secuencia de números de 16 bits obtenida se comporta como si fuesen números generados aleatoriamente

El programador puede asignar una semilla usando la función **qsrand(int)**. De esta forma, la secuencia de números aleatorios obtenida es siempre la misma (algo muy útil para depurar errores)

En nuestro caso, la semilla que le asignamos depende de los milisegundos que marque el reloj del sistema, lo que hace que la semilla asignada sea impredecible, y cada vez que pongamos en marcha nuestro servidor generará una secuencia de números aleatorios totalmente diferente

Ejemplo 2: Servidor de números aleatorios (13/23)

- “randomhandler.cpp”

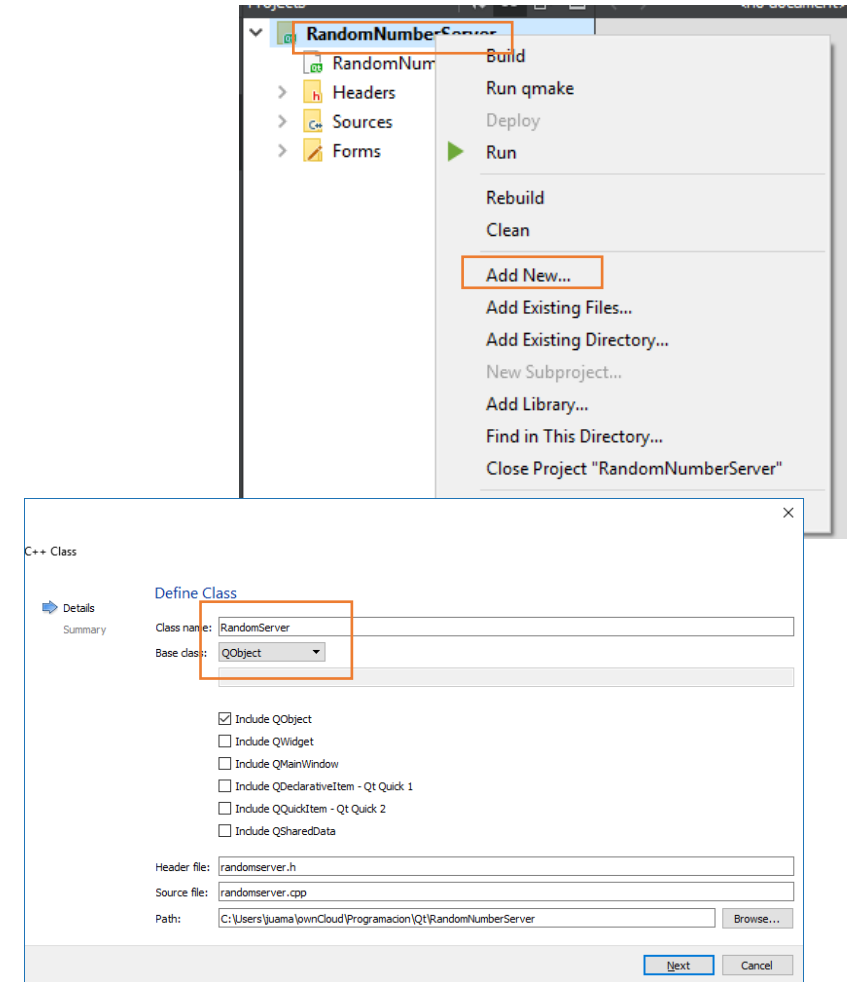
Programamos el *slot* **datosDisponibles()**. Este *slot* es invocado cada vez que el socket recibe datos desde el cliente

Tened en cuenta que en un mismo segmento de datos pueden ser enviadas varias líneas, por lo que debemos programar un bucle que compruebe si quedan líneas por procesar

```
90
91 void RandomHandler::datosDisponibles()
92 {
93     while(!_socket->canReadLine())
94     {
95         // Leemos una línea del socket y eliminamos los espacios en blanco al principio y al final de la línea (trimmed())
96         QByteArray buffer = _socket->readLine().trimmed();
97         QString linea = QString::fromUtf8(buffer);
98
99         // Mensaje de traza
100         qDebug().noquote().nospace() << "Petición de " << direccionIpRemota().toString() << ":" << puertoRemoto() << " # " << linea;
101
102         QString reply;
103
104         if (linea == "RND" || linea == "rnd")
105         {
106             // Si la petición es correcta la respuesta será un número aleatorio (con salto de línea al final)
107             reply = QString().number(rand()) + "\r\n";
108         }
109         else
110         {
111             // Si la petición es incorrecta la respuesta será un mensaje de error.
112             reply = "ERROR 500 Bad request\r\n";
113         }
114
115         // Enviamos la respuesta.
116         _socket->write(reply.toUtf8());
117         _socket->flush();
118
119         qDebug().noquote() << "Respondiendo:" << reply.trimmed();
120
121     }
122 }
```

Ejemplo 2: Servidor de números aleatorios (14/23)

- Vamos a crear la clase **RandomServer**
- Cread una nueva clase llamada RandomServer con el asistente de Qt Creator
Ver imagen (Botón derecho del ratón sobre el nombre del proyecto → Add New...)
- No os olvidéis de indicar que la clase base es *QObject*



Ejemplo 2: Servidor de números aleatorios (15/23)

- Editad “randomserver.h”

- Incluirá:

- Los “includes” necesarios para usar las clases definidas por Qt y la clase *RandomHandler* programada por nosotros
 - Un constructor al cual le pasaremos el número de puerto a través del cual el servidor aceptará las conexiones de los clientes
 - Una parte pública con tres métodos, además del constructor
 - Dos slots:
 - *conexionNueva()* será invocado cuando se conecte un cliente nuevo
 - *clienteDesconectado()* será invocado cuando se cierre la conexión con un cliente
 - El método para modificar el puerto no será público, ya que una vez que el servidor esté escuchando, el puerto no se puede modificar
 - En la zona privada tenemos un campo para almacenar el puerto y una referencia al objeto de tipo *QTcpServer* que usamos para crear el servidor

```
#ifndef SERVIDORDENUMEROSALEATORIOS_H
#define SERVIDORDENUMEROSALEATORIOS_H

#include <QObject>
#include <QTcpServer>
#include <QHostAddress>
#include "randomhandler.h"

class RandomServer : public QObject
{
    Q_OBJECT
public:
    explicit RandomServer(quint16 p = 33333, QObject *parent = nullptr);

    quint16 puerto();

    bool estaEscuchando();
    bool cerrar();

signals:

public slots:
    void conexionNueva();
    void clienteDesconectado();

protected:
    bool setPuerto(quint16 p);

private:
    quint16 _puerto;
    QTcpServer *_servidor;
};

#endif // SERVIDORDENUMEROSALEATORIOS_H
```

Ejemplo 2: Servidor de números aleatorios (16/23)

- Editad “randomserver.cpp”

Programamos el constructor:

Básicamente, creamos un objeto de tipo `QTcpServer()` para que escuche a través del puerto del servidor y conectamos la señal `newConnection()` al `slot conexionNueva()` para capturar las conexiones entrantes

```
RandomServer::RandomServer(quint16 p, QObject *parent) : QObject(parent)
{
    // Asignamos el puerto al campo privado.
    setPuerto(p);

    // Creamos un nuevo servidor
    _servidor = new QTcpServer();

    // Ponemos el servidor a escuchar a través del puerto asignado. Usaremos únicamente IPv4.
    if (_servidor->listen(QHostAddress::AnyIPv4, puerto()))
    {
        // Si el servidor ha podido abrir el puerto conectamos las señales con los slots
        connect(_servidor, SIGNAL(newConnection()), this, SLOT(conexionNueva()));
        connect(_servidor, SIGNAL(destroyed(QObject*)), this, SLOT(deleteLater()));
    }
    else
    {
        // Si el servidor no ha podido abrir el puerto lo destruimos.
        // Esta situación suele darse cuando hay otro servidor usando el mismo número de puerto.
        delete _servidor;
    }
}
```

Ejemplo 2: Servidor de números aleatorios (17/23)

- Editad “randomserver.cpp”

Encapsulamos el campo `_puerto`

Recordad que `setPuerto` no es un método público, ya que el puerto no se puede cambiar cuando el servidor esté activo

```
quint16 RandomServer::puerto()
{
    return _puerto;
}

bool RandomServer::setPuerto(quint16 p)
{
    _puerto = p;
    return true;
}
```

Creamos un método para cerrar el servidor y otro para conocer el estado del servidor

```
bool RandomServer::cerrar()
{
    if (!_servidor) return false;
    _servidor->close();
    delete _servidor;
    return true;
}

bool RandomServer::estaEscuchando()
{
    return _servidor && _servidor->isListening();
}
```

Ejemplo 2: Servidor de números aleatorios (18/23)

- Editad “randomserver.cpp”

Creamos el *slot* que acepta las nuevas conexiones de los cliente y crea un objeto de tipo *RandomHandler* por conexión

```
void RandomServer::conexionNueva()
{
    // Mientras queden conexiones pendientes las aceptaremos y crearemos objetos de tipo RandomHandler (una por conexión)
    while (_servidor->hasPendingConnections())
    {
        // Obtenemos la siguiente conexión establecida con el primer cliente de la cola de conexiones pendientes.
        QTcpSocket *socket = _servidor->nextPendingConnection();

        // Creamos un objeto de tipo RandomHandler para atender a la nueva conexión.
        RandomHandler *conn = new RandomHandler(socket);

        // Mostramos un mensaje de traza.
        qDebug().noquote().nospace() << "Conexión nueva " << conn->direccionIpRemota().toString() << ":" << conn->puertoRemoto();

        // Conectamos la señal desconectado() del objeto RandomHandler, con el slot clienteDesconectado() del servidor.
        // Así detectaremos cuando se desconecta cada cliente.
        connect(conn, SIGNAL(desconectado()), this, SLOT(clienteDesconectado()));
    }
}
```

Observad que, para cada conexión establecida, se crea un objeto nuevo de la clase *RandomHandler*. Además, para cada objeto creado, se conecta la señal *desconectado()* del objeto al *slot clienteDesconectado()* del servidor. Así el servidor puede capturar cuando un cliente se ha desconectado

Ejemplo 2: Servidor de números aleatorios (19/23)

- Editad “randomserver.cpp”

Programamos el *slot* que detecta las desconexiones de los clientes

En nuestro ejemplo, su única tarea será mostrar un mensaje de traza a través de la salida por defecto

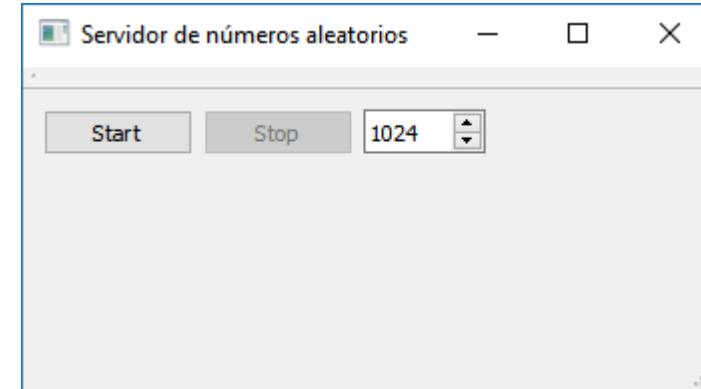
```
void RandomServer::clienteDesconectado()
{
    // Obtenemos el cliente que se ha desconectado llamando al método sender().
    RandomHandler *conn = (RandomHandler *)sender();

    // Mostramos un mensaje de traza.
    qDebug().noquote().nospace() << "Cliente desconectado: " << conn->direccionIpRemota().toString() << ":" << conn->puertoRemoto();

    // Le decimos a Qt que borre el objeto cuanto antes.
    conn->deleteLater();
}
```


Ejemplo 2: Servidor de números aleatorios (20/23)

- Vamos a modificar el comportamiento de los controles de nuestra aplicación

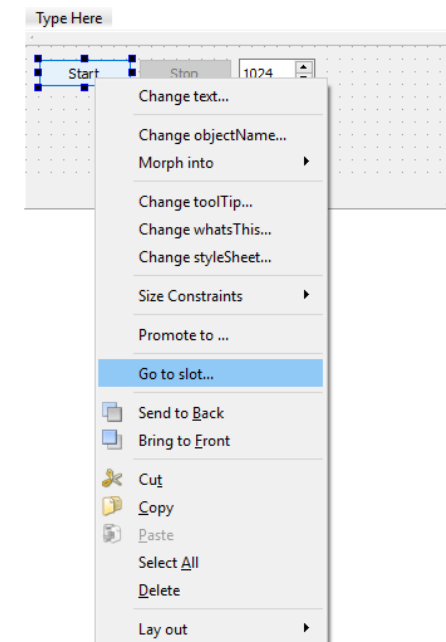


- Pulsando con el botón derecho del ratón sobre cada botón, seleccionad “Go to slot...”
- En la ventana que aparece, seleccionad la señal *clicked()* y pulsad OK
- Nos creará dos *slots* en el archivo “mainwindow.cpp” que deberemos programar:

```
void MainWindow::on_pushButton_start_clicked()
```

```
void MainWindow::on_pushButton_stop_clicked()
```

- Ambos *slots* los crea ya conectados a las señales *clicked()* de los botones, por lo que no tenemos que usar la función *connect*



Ejemplo 2: Servidor de números aleatorios (21/23)

- Modificando “mainwindow.h”
 - Añadid la línea #include “randomserver.h”

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "randomserver.h"
```

- Añadid a la zona privada un campo para la clase **RandomServer**:

```
private:
    Ui::MainWindow *ui;

    RandomServer *servidor;
};

#endif // MAINWINDOW_H
```

Ejemplo 2: Servidor de números aleatorios (22/23)

- Modificando “mainwindow.cpp”

```
void MainWindow::on_pushButton_start_clicked()
{
    // Desactivamos todos los controles de la ventana que estén activados.
    ui->pushButton_start->setEnabled(false);
    ui->spinBox_puerto->setEnabled(false);

    // Creamos un objeto de la clase RandomServer en el puerto indicado
    // por el spinBox
    servidor = new RandomServer(ui->spinBox_puerto->value());

    if (servidor->estaEscuchando())
    {
        // Si ha sido posible poner en marcha el servidor

        qDebug() <<"Servidor escuchando en el puerto:" << servidor->puerto();

        ui->pushButton_stop->setEnabled(true);
    }
    else
    {
        // Si NO ha sido posible poner en marcha el servidor

        ui->pushButton_start->setEnabled(true);
        ui->spinBox_puerto->setEnabled(true);
        delete servidor;

        qDebug() << "No se puede poner en marcha el servidor. Seguramente el puerto esté ocupado.";
    }
}
```

Ejemplo 2: Servidor de números aleatorios (23/23)

- Modificando “mainwindow.cpp”

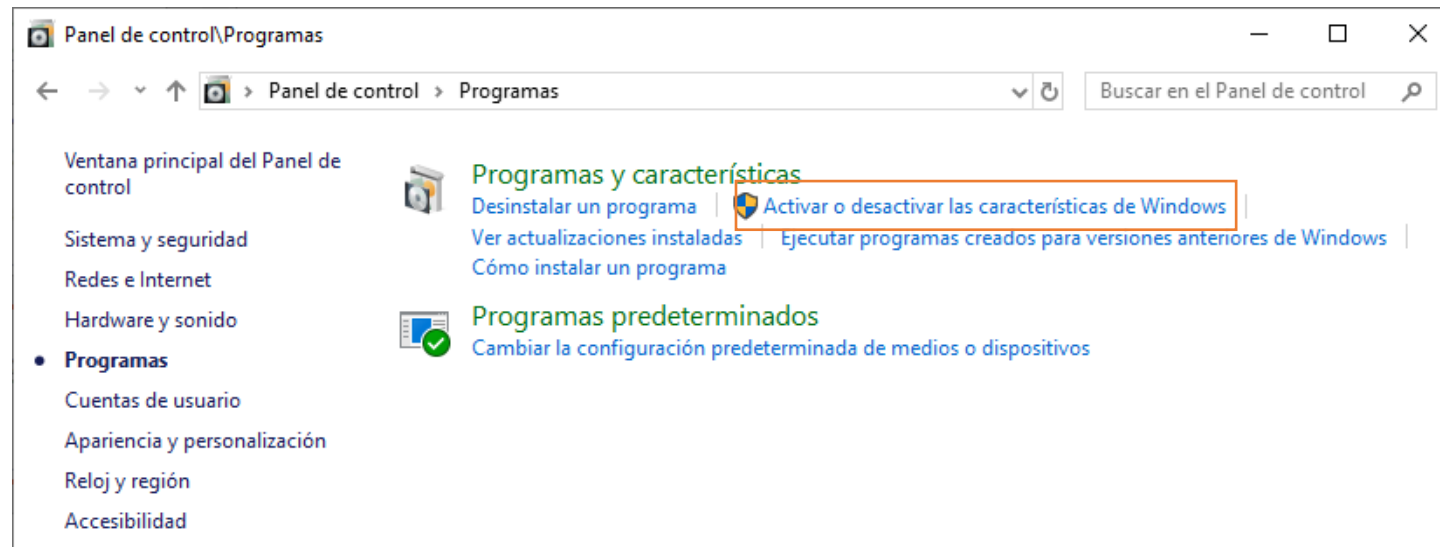
```
void MainWindow::on_pushButton_stop_clicked()
{
    ui->pushButton_stop->setEnabled(false);

    if (servidor)
    {
        servidor->cerrar();
        delete servidor;
    }

    ui->pushButton_start->setEnabled(true);
    ui->spinBox_puerto->setEnabled(true);
    qDebug() << "Servidor cerrado.";
}
```

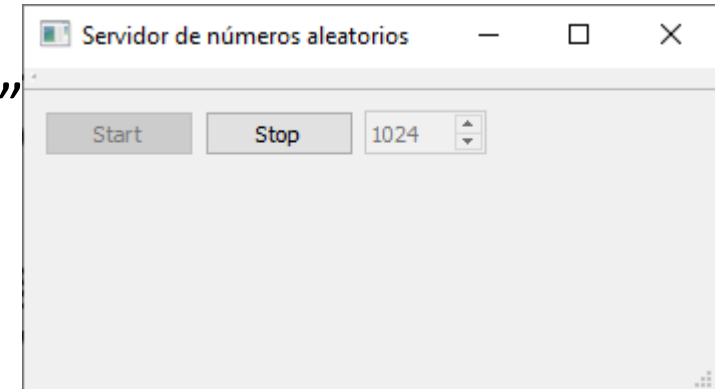
Prueba del servidor (1/2)

- Como nuestro servidor funciona en modo texto, podemos utilizar el comando “telnet” como cliente para probarlo
- Por defecto, “telnet” no está instalado en Windows pero podéis instalarlo en el panel de control, en “Programas y Características”, “Activar o desactivar las características de Windows”



Prueba del servidor (2/2)

- Poned en marcha el servidor y pulsad el botón “start”
- Abrid el “símbolo de sistema” (cmd.exe)
- ejecutad:
Telnet localhost 1024



Si habéis usado un puerto distinto en vuestro servidor, en lugar de 1024 poned el puerto que hayáis elegido

- Si todo va bien, nuestro servidor nos enviará el saludo: “RandomServer/1.0”
- Escribid: **RND** y pulsad *enter*
 - Si todo va bien nuestro servidor nos enviará un número aleatorio
- Probad a escribir cualquier otra palabra y pulsad *enter*. El servidor debería devolver un mensaje de error

Actividades

Actividad 1 – Cliente de Números Aleatorios

Programad un cliente con una interfaz similar al cliente del ejemplo 1

- El cliente se conectará al servidor de números aleatorios
- El cliente utilizará la clase *QTimer* de Qt para enviar una petición de un número aleatorio al servidor y mostrarlo en la ventana cada 5 segundos

Actividad 2 – RandomServer v.2 (1-2)

1. Modificad el servidor *RandomServer* para crear la versión 2.0 del protocolo
 - El servidor deberá enviar la cadena de texto “RandomServer/2.0” como saludo inicial
 - El servidor debe ser compatible con la versión 1.0, por lo que deberá implementar la orden “RND”
2. Modificad el servidor para que admita las siguientes peticiones (además de “RND”)
 - “MIN”. Sin parámetros. Devolverá el valor mínimo del rango de los números aleatorios
 - “MAX”. Sin parámetros. Devolverá el valor máximo del rango de los números aleatorios
 - “SETMIN <num>”. El parámetro <num> será un número en decimal. Cambiará el valor del mínimo
 - Si tiene éxito, el servidor devolverá el valor <num>
 - Si falla devolverá el valor del mínimo
 - “SETMAX <num>”. El parámetro <num> será un número en decimal. Cambiará el valor del máximo
 - Si tiene éxito, el servidor devolverá el valor <num>
 - Si falla devolverá el valor del máximo
 - “RNDFLOAT”. Devolverá un número aleatorio en el rango: [0.0, 1.0[
 - Utilizad para ello la fórmula: **grand() / (double)RAND_MAX**

Actividad 2 – RandomServer v.2 (2-2)

3. El servidor deberá enviar una respuesta a los clientes para todas las posibles operaciones
 - Todas las respuestas se enviarán como cadena de caracteres en formato *utf8*. Los números se enviarán en base 10
 - Para las operaciones de consulta o las de generación de números aleatorios se deberá enviar como respuesta el número
 - Para las operaciones de modificación se deberá enviar como respuesta si la modificación se ha llevado a cabo o no
 - Todas las respuestas del servidor deben terminar con un salto de línea: “\r\n”
4. El servidor deberá generar códigos y mensajes de error distintos para los siguientes casos:
 - El número de parámetros de una petición es incorrecto
 - Un parámetro numérico no está en el formato correcto
 - La orden no existe

Actividad 3 – Servidor de Fecha y Hora

Implementad una aplicación en red de tipo Cliente-Servidor para consultar la fecha y la hora del servidor

- Deberéis implementar tanto el cliente, como el servidor
- Deberéis diseñar el protocolo que permita realizar las siguientes funciones
- En el servidor...
 - El servidor debe admitir operaciones para obtener: la hora y la fecha, sólo la hora, sólo la fecha, y el día de la semana
 - (Opcional) El servidor debe admitir también una petición de alarma, de tal forma que el servidor enviará dos respuestas:
 - La primera aceptando o rechazando la petición de alarma
 - En el caso de aceptar la alarma, se enviará una segunda respuesta cuando venza la alarma
- En el cliente...
 - Debe tener varios botones, uno para cada una de las funciones anteriores, y debe mostrar la respuesta en un cuadro de texto o similar