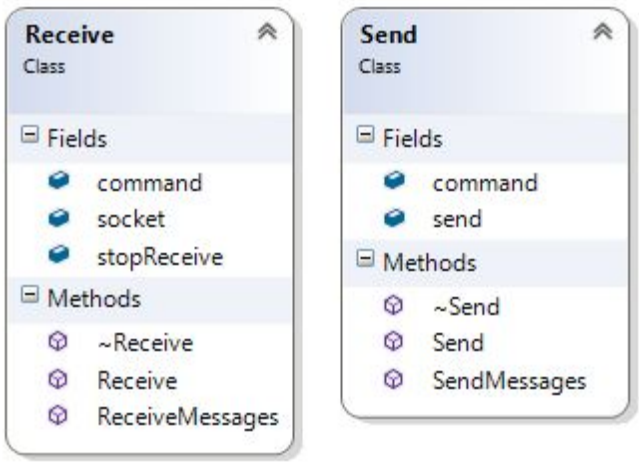
Practica 1 - Buzz

Poner en marcha el proyecto

1. Abrir 3 compiladores.
2. Establecer Server como proyecto de inicio en uno.
3. Ejecutar el servidor en modo Release (Problema en la librería.)
4. Establecer el Client como proyecto de inicio en los otros 2 compiladores.
5. Ejecutar los clientes en modo Release.
6. Para salir del servidor “Retroceso” y “Escape” en los clientes.

Diagrama de clases.



Reglas del juego.

Este juego es un concurso para 2 jugadores, una vez iniciada la partida aparecerá una pregunta y los jugadores deberán responder en menos de 10 segundos. Teclas “1” ”2” ”3” o ”4” en función de la respuesta.

Ganarán puntos todos los jugadores que contesten correctamente, el jugador que conteste con mayor velocidad recibirá la máxima puntuación, el segundo la segunda. Ejemplo: El jugador 1 responde en 2 segundos. el jugador 2 en 3’. Las puntuaciones seràn, 3 puntos y 2 puntos respectivamente.

Protocolo de comunicación.

El protocolo de comunicación consiste en la implementación de una cabecera de 3 dígitos enteros seguidos de una cadena de caracteres.

Type	SubType	Player	Message	Explicación
1	N	J	vacío	Jugador envía su N de respuesta identificado con su índice de jugador J
2	1	J	vacío	Servidor envía inició partida.
2	5	9	vacío	Servidor envía acaba partida.
2	3	J	vacío	Servidor envía final de tiempo
2	4	J	vacío	Servidor envía jugador J ya ha respondido.
3	N	J	vacío	Se envía nueva pregunta, indice N.
4	1	J	Nombre	Nombre de jugador con su índice J. Confirmación de cliente a servidor
5	N	J	vacío	Puntuaciones N de los jugadores J.
6	0	J	vacío	El cliente J se ha desconectado

Aplicación del protocolo a modo de ejemplo:

Los jugadores envían su nombre sin ningún índice : 410Manolo. Servidor se guarda los nombre y envía todos los nombres a cada cliente, con el índice actualizado: 412Manolo, cliente guarda todos los jugadores con sus índices actualizados.

Servidor envía el índice de la nueva pregunta, en este caso la pregunta 0: 300. Los clientes mandan su respuesta: 132. (Manolo ha respondido la 3). Servidor hace saber quien ha respondido a los clientes: 242 (Manolo ya ha respondido.)

El tiempo termina, servidor manda a los jugadores que ha terminado el tiempo: 230. Servidor manda las puntuaciones actualizadas. 532 (Manolo recibe 3 puntos.) 510 (Carmen recibe 1 punto) etc... Servidor manda la nueva pregunta: 310

Cuando todas las preguntas terminan o un jugador se desconecta servidor manda: 259. Todos los clientes ordenan las puntuaciones y muestran por pantalla el ganador.

Indica cómo guarda el servidor el estado del juego.

El servidor guarda su estado de juego en la variable state.

```
26  enum State {
27      send, // enviar paraula nova y que comenci partida
28      play, // mentres els jugadors estan escribint. comproba si sacaba el temps i si algú ha encertat la partida
29      points, // Envia les puntuacions als jugadors y actualitza els seus logs
30      win // el joc sacaba
31  };
```

```
143         switch (state) {
144         case send:
```

Indica cómo aceptas las conexiones en servidor, dónde se guardan y en qué parte del código puedo encontrarlo.

Las conexiones se guardan en un std vector de punteros sockets. Las conexiones se crean, aceptan y almacenan a medida que los jugadores van conectándose.

```
 99      //Accept per els dos jugadors
100
101     for (int i = 0; i < player.size(); i++)
102     {
103         if (listener.accept(*sockettmp) != sf::Socket::Done) {
104             std::cout << "Error al acceptar conexió" << std::endl;
105             return -1;
106         }
107         std::cout << "\n New user" << std::endl;
108         //sockettmp->setBlocking(false);
109         sockets.push_back(sockettmp);
110         sockettmp = new sf::TcpSocket;
111     }
112     listener.close();
```

Indica qué información guarda servidor por cada uno de los clientes y dónde lo almacena.

Existe un Struct llamada player, con todos los atributos que los jugadores poseen (nombre, índice y puntuación) tanto servidor como cliente van actualizando el mismo Struct a medida que el juego va avanzando.

```
74      // Crear players per guardar la info
75      std::vector<Player> player(MAX_USERS);
76      for (int i = 0; i < player.size(); i++)
77      {
78          player[i]._num = i;
79      }
```

Indica qué información guarda el cliente del juego y dónde la guarda.

Existe un Struct llamada player, con todos los atributos que los jugadores poseen, tanto servidor como cliente van actualizando el mismo Struct a medida que el juego va avanzando.

```
74      std::string name; // Nom auxiliar al crea el player
75      int _indexClient; //Index del client
76
77      std::vector<Player> player(MAX_USERS);
78      for (int i = 0; i < MAX_USERS; i++)
79      {
80          player[i]._num = i;
81      }
```

Cliente también guarda el índice i el nombre del propio jugador en una variable local para diferenciarse del resto de jugadores.

Indica qué información guarda el cliente de cada uno de los demás jugadores y dónde la guarda.

Todos los clientes guardan la misma información para todos los jugadores (Índice, nombre, puntuación y un booleano de seguimiento). Esta información está almacenada en el Struct player.

```
74      std::string name; // Nom auxiliar al crea el player
75      int _indexClient; //Index del client
76
77      std::vector<Player> player(MAX_USERS);
78      for (int i = 0; i < MAX_USERS; i++)
79      {
80          player[i]._num = i;
81      }
```

Indica si estás utilizando threading, sockets non-blocking o una combinación de ambas, el motivo y en qué partes del código.

Estamos usando sockets non-blocking porque para nosotros son más cómodos que los threads.

a. En cliente

En el estado que espera la nueva pregunta cambiamos los sockets a Blocking para que se queden esperando el mensaje del servidor para avanzar.

```
227             socket.setBlocking(true);
```

Una vez han recibido el índice de pregunta y se ponen en el estado de Play, cambiamos los sockets a non-Blocking.

```
239                     socket.setBlocking(false);
```

Cuando llegamos al momento que esperan al servidor para actualizar la puntuación volvemos a ponerlos en blocking.

```
330                 socket.setBlocking(true);
```

b. En servidor

Durante el loop del servidor, los sockets se mantienen non blocking. En los casos que tengamos que enviar la misma información a todos los jugadores, usamos la función sendAll que nos permite, también, decidir si queremos enviar la información con sockets blocking o non blocking.

```
50  void sendAll(Send* sender, std::vector<sf::TcpSocket*> sockets, bool block = false) {
51      for (int i = 0; i < sockets.size(); i++)
52      {
53          if (block) sockets[i]->setBlocking(true);
54          sender->send = sockets[i];
55          sender->SendMessages();
56          if (block) sockets[i]->setBlocking(false);
57      }
58  }
```

Indica dónde haces el disconnect de los sockets.

a. En cliente

Antes de terminar el programa

```
411     socket.disconnect();
412     return 0;
413 }
```

b. En servidor

Después de decirle a los jugadores que se ha terminado el juego.

```
248             case win:
249                     command = protocol.CreateMessage(2, 5, 9, "");
250                     sendAll(&sender, sockets, true);
251                     for (int i = 0; i < sockets.size(); i++)
252                     {
253                         sockets[i]->disconnect();
254                         delete sockets[i];
255                     }
256                     serverOn = false;
257                 break;
258             }
259     }
260     return 0;
261 }
```

Indicar si estás utilizando RAW data o SFML Packets para enviar/recibir y porqué.

Estamos utilizando Raw data, ya que proximamente haremos UDP y RAW data se aproxima más a la metodología de mensajes de UDP.

Indica en qué punto del código se envían/reciben cada uno de los mensajes del protocolo de comunicación.

Servidor:

Función que gestiona el multienvío.

```
50  void sendAll(Send* sender, std::vector<sf::TcpSocket*> sockets, bool block = false) { // per misatges iguals que s'envien a tots el jugadors
51      for (int i = 0; i < sockets.size(); i++)                          // podriam ficar un bool de parametre si volem que es faci blocking
52      {
53          if (block) sockets[i]->setBlocking(true);
54          sender->send = sockets[i];
55          sender->SendMessages();
56          if (block) sockets[i]->setBlocking(false);
57      }
58  }
59
```

```
113     //Rebre noms dels jugadors
114     for (int i = 0; i < player.size(); i++)
115     {
116         receiver.socket = sockets[i];
117         if (receiver.ReceiveMessages()) {
118             player[i]._name = protocol.GetWord(command);
119         }
120     }
```

```
122     for (int i = 0; i < player.size(); i++)
123     {
124         command = protocol.CreateMessage(4, 0, player[i]._num, player[i]._name);
125         sendAll(&sender, sockets, true);
126     }
```

```
151                 command = protocol.CreateMessage(3, questionIndex, 0, ""); // crear misatge amb index de la pregunta random
152                 sendAll(&sender, sockets, true); // enviar a tots els jugadors el command amb el index de la pregunta random
```

```
164                     receiver.socket = sockets[i];
165                     if (receiver.ReceiveMessages()) { // si rep misatge
166                         if (protocol.GetType(command) == 1) { // si rep resposta e jugador
167                             if (protocol.GetSubType(command) == question.correctAnswer) { // si es resposta correcta
168                                 player[i]._score += sumScore; // suma punts depenent de l'ordre en que ha respos
169                                 if (sumScore > 1) sumScore--; // si tots responen be, el 3er i el 4rt guanyen 1 punt
170                                 playerChecks[i] = 1; // per saber que ha respos
171                             }
172                             command = protocol.CreateMessage(2, 4, i, "");
173                             sendAll(&sender, sockets); // enviar a tots la resposta que ha donat el jugador i
```

```
183                     command = protocol.CreateMessage(2, 3, 0, ""); // s'ha acabat el temps
184                     sendAll(&sender, sockets, true);
```

```
191                 command = protocol.CreateMessage(2,3,0,""); // s'ha acabat el temps
192                 sendAll(&sender, sockets, true);
```

```
201                     command = protocol.CreateMessage(5, player[i]._score, player[i]._num, "");
202                     sendAll(&sender, sockets); // enviar a tots les putnuacions actualitzades
```

```
208             for (int i = 0; i < sockets.size(); i++)
209             {
210                 if (playerChecks[i] != 1) { // si el jugador i ya ha enviado respueste, se le omite
211                     receiver.socket = sockets[i];
212                     if (receiver.ReceiveMessages()) { // si rep misatge
```

```
232         case win:
233                 command = protocol.CreateMessage(2, 5, 9, ""); // enviem que s'acaba la partida,
234                 sendAll(&sender, sockets, true);
```

Cliente:

```
190     //Enviem el nostre nom
191     command = protocol.CreateMessage(4,0,0,name);
192     sender.SendMessages();
```

```
194     receiver.ReceiveMessages(); //Rebem el primer nom
195     if (name == protocol.GetWord(command)) {
196         _indexClient = protocol.GetPlayer(command);
197     }
198     player[protocol.GetPlayer(command)]._name = protocol.GetWord(command);
199     Jugador1.setString(protocol.GetWord(command));
```

```
229             if (receiver.ReceiveMessages()) {
230                 if (protocol.GetType(command) == 3) {
231                     questionIndex = protocol.GetSubType(command); // agafar index de la pregunta
```

```
267                     command = protocol.CreateMessage(6, 0, _indexClient, ""); // enviem missatge de adios
268                     sender.SendMessages(); //enviem missatge
```

```
274                         command = protocol.CreateMessage(1, 0, _indexClient, ""); //enviem resposta 1
275                         sender.SendMessages(); //enviem missatge
```

```
299                 if (receiver.ReceiveMessages()) {//Rebem missatge durant el gameplay
300                     if (protocol.GetType(command) == 2) { //missatge de servidor
301                         switch (protocol.GetSubType(command)) {
```

```
334                 if (receiver.ReceiveMessages()) {
335                     if (playerChecks[protocol.GetPlayer(command)] == 0) {
336                         player[protocol.GetPlayer(command)]._score = protocol.GetSubType(command);
337                         playerChecks[protocol.GetPlayer(command)] = 1;
```

```
342                     command = protocol.CreateMessage(4, 0, _indexClient, "");
343                     sender.SendMessages();
```

Dificultades que has encontrado y conclusiones.

Han habido bastantes dificultades, como por ejemplo todo el tema de ir seteando los sockets de blocking a non-blocking en diferentes fases del juego. Por otro lado, los mayores problemas han venido porque en algunas partes del programa cliente y servidor se desincronizan y provocaba que se encontrarán en diferentes fases del juego.

También han habido problemas con el mismo compilador, Visual Studio, ya que en algunos momentos se corrompía el programa y todos los mensajes llegaban mal, (Para solucionarlo hemos tenido que hacer Clean).

Uno de los principales impedimentos ha sido tener que abrir 3 programas para hacer test, resulta ser bastante engorroso. Además de que el juego solo lo hemos podido probar en local y abrir 3 compiladores ralentiza mucho el ordenador. Es paradójico que los juegos de la asignatura de juegos en redes solo funcionen en redes locales.