3.6. Networking

3.6.1. Introducción

La evolución de Internet como sistema global de comunicaciones ha tenido un **impacto muy significativo en la creación de videojuegos**. De hecho, hoy en día se puede afirmar que la mayoría de los juegos que se distribuyen incorporan algún tipo de funcionalidad en red o características *online* que permiten, entre otras cosas, jugar con amigos al mismo juego desde distintas ubicaciones físicas. En concreto, Internet ha permitido, entre otros aspectos, integrar en el mundo de los videojuegos posibilidades como las siguientes:

- Aparición de videojuegos con modos de juego exclusivos para múltiples jugadores en red, como por ejemplo *World of Warcraft*™.[28]
- Distribución masiva de juegos, considerando especialmente el caso de las plataformas móviles, como por ejemplo los *smartphones*.
- Actualización automática de juegos para garantizar el contenido más actual o la correción de *bugs* existentes.
- Aparición de nuevos modelos de negocio basados en la distribución de juegos gratuitos y la posterior monetización a través de sistemas de publicidad.

Evidentemente, el juego en red introduce una **complejidad adicional** a la hora de desarrollar un videojuego. Si, por ejemplo, dos amigos juegan *online* al mismo juego, entonces los dos deberían tener una representación lo más similar posible del estado del juego para poder jugar de una manera adecuada. En otras palabras, es desable minimizar el tiempo que transcurre desde que el jugador A interactúa con el juego (por ejemplo, moviendo su personaje) hasta que el jugador B observa el resultado de dicha interacción (por ejemplo, observar al personaje en la posición actual).

En esta sección se introducen algunas consideraciones de diseño especialmente relevantes y se discute el uso de **sockets** como herramienta básica de comunicación en juego en red. Posteriormente, se estudia una posible solución de integración de funcionalidad básica *online* con **Haxe y NME**.

[28] http://es.wikipedia.org/wiki/World_of_Warcraft

3.6.2. Consideraciones iniciales de diseño

Desde el punto de vista de la red, los juegos multi-jugador que se desarrollan en tiempo real, esto es, varios jugadores jugando de forma simultánea, son los más exigentes. Esto se debe a que su diseño y desarrollo tiene que considerar diversas cuestiones:

- **Sincronización**. La identificación de las acciones que ocurren en juegos de tiempo real requiere de una gran eficiencia para proporcionar al usuario una buena experiencia. En este contexto, se debe diseñar el sistema de transmisión y sincronización así como los turnos de red para que el juego sea capaz de evolucionar sin problemas.
- **Gestión de actualizaciones**. La identificación de las actualizaciones de información y la dispersión de dicha información es esencial desde el punto de vista del jugador para que, en cada momento, todos las partes involucradas tengan la información necesaria y actualizada del resto.
- **Determinismo**. La consistencia es crítica para asegurar que todas las partes del videojuego disponen de la misma información.

Generalmente, el **diseño del modo multi-jugador** de un juego se aborda desde el principio [3]. En otras palabras, el modo *online* de un juego no suele integrarse como un añadido más en las últimas etapas de desarrollo, sino que representa un aspecto crucial que se aborda desde el principio. De hecho, una práctica bastante extendida consiste en abordar el modo *single player* o de un único jugador (*offline*) como un caso particular de la versión multi-jugador del juego.

En este contexto, se debe identificar, desde las primeras fases del diseño del juego, qué información se va a distribuir para que todas las partes involucradas tengan la información necesaria que garantice la correcta evolución del juego.

Al más bajo nivel, es necesario diseñar un protocolo de comunicaciones que, típicamente mediante el uso de sockets, permita transmitir toda la información necesaria en el momento oportuno. Este protocolo debe definir idealmente los siguientes aspectos:

- **Sintaxis**: qué información y cómo se estructura la información a transmitir. Esta especificación define la estructura de los mensajes a transmitir y recibir, su longitud, los campos de los mensajes, etc. El resultado de esta fase de diseño debe ser una serie de estructuras a transmitir y recibir a través de un punto de comunicación.

- **Semántica**: qué significa la información transmitida y cómo interpretarla una vez recibida. La interpretación de la información se realiza mediante el *parsing* o procesamiento de los mensajes transmitidos e interpretando la información recibida.
- **Temporización**: el modelo de temporización expresa la secuencia de mensajes que se deben recibir y transmitir en función de los mensajes recibidos y enviados con anterioridad, teniendo en cuenta la información que se necesite transmitir. La temporización y la semántica generalmente se traducen en una máquina de estados cuyas transiciones vienen determinadas por los mensajes recibidos, los mensajes transmitidos y la información contenida en los mismos.

El protocolo de un videojuego debe especificar la sintaxis, semántica y temporización.

Un aspecto determinante en el diseño del modo multi-jugador consiste en definir **qué información es necesaria transmitir**. Esta pregunta es específica del videojuego a desarrollar y, por lo tanto, la respuesta variará de un juego a otro. Sin embargo, de forma genérica, se han de identificar los siguientes aspectos:

- Aquella información vinculada al estado de una entidad, como por ejemplo su posición en el espacio 3D y su orientación.
- Información relativa a la lógica del juego, siempre y cuando sea necesario su transmisión en red.
- Aquellos eventos que un jugador puede realizar y que afectan al resto de jugadores y al estado del propio juego.

La eficiencia en la transmisión y procesamiento de la información de red, así como el objetivo de minimizar la información transmitida, deben guiar el diseño y la implementación de la parte de *networking*.

De forma paralela al diseño del protocolo de comunicación de un juego, es necesario decidir la **estructura lógica** de la parte de *networking*. Generalmente existen las siguientes alternativas viables:

- **Arquitectura P2P** (*peer-to-peer*): en este modelo cada usuario comparte la información con todos los usuarios integrados en una partida en curso.
- **Arquitectura cliente-servidor**: en este modelo todos los usuarios envían la información a un servidor central que se encarga de redistribuirla.

A continuación se discute el caso particular de los sockets TCP/IP como herramienta básica para enviar y recibir información.

3.6.3. Sockets TCP/IP

La programación con sockets es la programación en red de más bajo nivel que un desarrollador de videojuegos generalmente realizará. Por encima de los sockets básicos, que se discutirán a continuación, es posible utilizar bibliotecas y *middlewares* de comunicaciones de más alto nivel. Sin embargo, estas herramientas aportan un nivel más alto de abstracción aumentando, a priori, la productividad y la calidad del código a costa, en algunas ocasiones, de una pérdida de eficiencia.

Conceptualmente, un socket es un **punto de comunicación** con un proceso que permite comunicarse con él utilizando, en función del tipo de socket utilizado, una serie de protocolos de comunicación. En función del rol que asuma el proceso, se pueden distinguir dos tipos de programas:

- **Cliente**, entidad que solicita a un servidor un servicio.
- **Servidor**, entidad atiende peticiones de los clientes.

En el **contexto de NME** y el lenguaje de programación Haxe existen diversas opciones desde el punto de vista de la implementación de un sistema cliente/servidor basado en sockets. Un ejemplo muy interesante está representado por la clase *ThreadServer*[29], la cual se puede utilizar en Haxe para generar la parte servidora en Neko.

Neko es un lenguaje de programación y un entorno de ejecución creado por Nicolas Cannase y que está soportado por Haxe. En otras palabras, Haxe puede generar *bytecode* para Neko. De hecho, cuando NME dio sus primeros pasos, fue diseñado para proporcionar soporte gráfico, sonido y otra funcionalidad multimedia para la máquina virtual de Neko. Sin embargo, el rendimiento no fue el esperado inicialmente y, a partir de ahí, surgió C++ como *target* por defecto para aplicaciones de escritorio desarrolladas con Haxe.

[29] http://haxe.org/doc/neko/threadserver

La **clase ThreadServer** se puede utilizar para crear, de una manera sencilla, servidores multi-hilo en los que cada hilo puede manejar un número arbitrario de peticiones o sockets. Sin embargo, el código necesario para el tratamiento de dichas peticiones se encapsula en un único hilo, simplificando el mismo y evitando así la integración de mecanismos de sincronización para evitar inconsistencias.

Por otra parte, esta clase se puede parametrizar para trabajar con distintos tipos de datos, típicamente asociados a las estructuras *cliente* y *mensaje*. El siguiente listado de código muestra las estructuras *Client* (línea 1-3) y *Message* (líneas 5-7), las cuales encapsulan, respectivamente, el identificador del cliente y el contenido del mensaje.

Listado 3.64: Clase Server. Tipos de datos.

```
1 typedef Client = {
2   var id : Int;
3 }
4
5 typedef Message = {
6   var str : String;
7 }
```

Para poder utilizar esta clase, al menos es necesario sobreescribir las funciones *clientConnected()*, *readClientMessage()* y *clientMessage*, además de definir las estructuras *Client* y *Message*. Todas estas funciones están afectadas por el modificador *dynamic* con el objetivo de garantizar la interoperabilidad con distintas plataformas.

El siguiente listado muestra una posible implementación de la función *clientConnected()*. Básicamente, esta función genera un identificador aleatorio para el cliente conectado y muestra información de la conexión.

Listado 3.65: Clase Server. Función clientConnected().

```
1  class Server extends ThreadServer<Client, Message> {
2
3    // Crea un nuevo cliente.
4    override function clientConnected ( s : Socket ) : Client {
5      var num = Std.random(100);
6      Lib.println("Cliente " + num + " desde " + s.peer());
7      return { id: num };
8    }
9
10   // Restos de funciones...
11
12 }
```

A continuación, la función *readClientMessage()* es la responsable de devolver una dupla formada por el contenido del mensaje y el tamaño del mismo. Note cómo en las líneas 8-11 se utiliza un bucle para leer el contenido del mensaje de manera incremental.

Listado 3.66: Clase Server. Función readClientMessage().

```
1  // Lee el mensaje de un cliente concreto.
2  override function readClientMessage (c:Client,
3                       buf:Bytes,
4                       pos:Int, len:Int) {
5    // ¿Mensaje completo?
6    var complete = false;
7    var cpos = pos;
8    while (cpos < (pos+len) && !complete) {
9      complete = (buf.get(cpos) == 46);
10     cpos++;
11   }
12
13   // Si no llega un mensaje completo, readClientMessage retorna.
14   if( !complete )
15     return null;
16
17   // Devuelve el mensaje completo.
18   var msg:String = buf.readString(pos, cpos-pos);
19   return {msg: {str: msg}, bytes: cpos-pos};
20 }
```

Finalmente, la función *clientMessage()* es la responsable de gestionar el mensaje recibido por parte del servidor. En el siguiente listado se muestra una implementación trivial en la que simplemente se imprime dicho mensaje.

Listado 3.67: Clase Server. Función clientMessage().

```
1 // Gestiona el mensaje del cliente.
2 override function clientMessage ( c : Client, msg : Message ) {
3   Lib.println(c.id + " envio: " + msg.str);
4 }
```

Como se introdujo anteriormente, los sockets representan puntos de comunicación entre pares. La información básica para establecer estos puntos de comunicación consiste en definir la dirección del host y el puerto de escucha. En el siguiente listado se muestra la instanciación de un objeto de la clase *Server*, la cual hereda de *ThreadServer*. Note cómo es necesario invocar a la función *run()* para indicar dicha información.

Listado 3.68: Clase Server. Función main().

```
1 public static function main() {
2   var server:Server = new Server();
3   server.run("localhost", 1234);
4 }
```

En el lado del cliente, el código es mucho más sencillo ya que sólo es necesario invocar la funcionalidad de la parte servidora.

Listado 3.69: Clase Cliente.

```
1  class Client {
2
3    public static function main () {
4        Lib.println("Abriendo socket...");
5        var sock = new Socket();
6        sock.connect(new Host("localhost"), 1234);
7
8        Lib.println("Enviando mensajes...");   Sys.sleep(.1);
9        sock.write("Mensaje de test.");        Sys.sleep(.3);
10       sock.write("Otro mensaje de test.");   Sys.sleep(.3);
11       sock.write("Y otro mensaje de test.");
12
13       sock.close(); Lib.println("Cliente finalizado.");
14   }
15
16 }
```

Para compilar el código fuente, tanto del cliente como del servidor, para Neko es necesario ejecutar los siguientes comandos:

```
$ haxe -neko server.n -main Server
$ haxe -neko client.n -main Client
```

Para ejecutar el servidor, es necesario ejecutar el siguiente comando:

```
$ neko server.n
```

Del mismo modo, para ejecutar el cliente, es necesario ejecutar el siguiente comando:

```
$ neko client.n
```

Una posible salida del servidor es la siguiente:

```
Cliente 13 desde { host => 127.0.0.1, port => 56216 }
13 envio: Mensaje de test.
13 envio: Otro mensaje de test.
13 envio: Y otro mensaje de test.
Cliente 13 desconectado
```

Una posible salida del cliente es la siguiente:

```
Abriendo socket...
Enviando mensajes...
Cliente finalizado.
```

3.6.4. Gestión on-line de récords en Tic-Tac-Toe

En esta sección se discute la integración de un servidor de registro de récords para el juego de tres en raya o tic-tac-toe. El principal objetivo perseguido es el de estudiar cómo se puede **integrar funcionalidad on-line** mediante el uso de de sockets. El término *récord* se ha utilizado en este caso de estudio para reflejar la situación en la que el jugador gana a la máquina, de manera independiente a la dificultad y tiempo empleado. En este contexto, el servidor será responsable de manejar, como se discutirá más adelante, una lista ordenada con los récords registrados.

Por una parte, se ha implementado un servidor utilizando Haxe y la clase *ThreadServer* proporcionada en la API de Neko. Este servidor es una versión adaptada del servidor discutido en la sección anterior.

Por otra parte, en el propio juego se ha incluido un **módulo de comunicaciones** que tiene como principal responsabilidad tanto el envío de récords al servidor como la obtención de las mejores puntuaciones. Dichas puntuaciones aparecen en la ventana principal del propio juego.

Servidor de récords

Como se ha comentado anteriormente, el servidor de récords es una versión modificada del servidor de la sección 3.6.3, prestando especial atención a los tipos de mensajes necesarios para comunicar el juego con el servidor y viceversa.

El siguiente listado de código muestra la implementación de la función *clientMessage()*, responsable de procesar el mensaje recibido de un cliente. En función de su contenido se distingues dos casos posibles:

- Mensaje de **solicitud de récords**, identificado por el servidor de manera sencilla si el propio mensaje comienza por el token *request* (líneas 14-18). En este caso, el mensaje recibido tendrá el formato $request : host : puerto$ para que el servidor pueda extraer tanto el host como el puerto del cliente y establecer un socket para enviar el listado de récords (ver función *sendRecords()* en la línea 17).
- Mensaje de **registro de un récord**, el cual contiene la información del nuevo récord de acuerdo al formato $jugador : dificultad : tiempo$ (líneas 20-24).

Listado 3.70: Clase Server. Función clientMessage().

```
1  class Server extends ThreadServer<Client, Message> {
2
3    private var _records:Array<String>;
4
5    // Resto de código...
6
7    // Gestiona el mensaje del cliente.
8    override function clientMessage (c : Client, msg : Message) : Void
         {
9      var msg_rec = msg.str.substr(0, msg.str.length - 1);
10     var tokens = msg_rec.split(":");
11
12     // Petición de récords.
13     // msg ->request:host:port
14     if (tokens[0] == "request") {
15       Lib.println(c.id + " solicita el registro de rcords.");
16       // Envía records actualizados al cliente que envió el suyo.
17       sendRecords(tokens[1], Std.parseInt(tokens[2]));
18     }
19     // Registro de un nuevo récord.
20     else {
21       Lib.println(c.id + " registra un nuevo record: " + msg.str);
22       // Añade el récord.
23       addRecord(msg.str.substr(0, msg.str.length - 1));
24     }
25   }
26
27 }
```

La **gestión de récords** se delega en las funciones del siguiente listado. La primera de ellas, *addRecord()* (líneas 2-5), añade el récord recibido como argumento a la lista de récords para, posteriormente, reordenarla. El criterio de ordenación se define en la función *sort()* (línea 4) y basa en:

- Primero los récords con un nivel de dificultad más elevado, es decir, *impossible*, *medium* y *easy*.
- Si el nivel de dificultad es el mismo para dos récords, entonces aparecerá primero el que tenga un tiempo inferior (se derrotó a la máquina en un tiempo menor).

La figura 3.45 muestra una captura de pantalla de la interfaz gráfica del juego en la que se resalta la parte de **visualización de récords**. Como se puede apreciar, el número máximo de récords mostrados está limitado, actualmente, a 5. En este contexto, el servidor maneja una lista con todos los récords registrados, pero sólo enviará a un cliente los n primeros (actualmente 5).

Listado 3.71: Clase Server. Gestión de récords.

```
1  // Añade un nuevo record.
2  public function addRecord (newRecord:String) : Void {
3    _records.push(newRecord);
4    _records.sort(sort);
5  }
6
7  // Envía los récords al cliente.
8  public function sendRecords (host:String, port:Int) : Void {
9    var sock = new Socket();
10   sock.connect(new Host(host), port);
11
12   Lib.println("Enviando records..."); Sys.sleep(.1);
13   sock.write(getUpdatedRecords());
14
15   sock.close();
16 }
17
18 // Devuelve un String con los 5 mejores récords.
19 public function getUpdatedRecords (numRecords:Int = 5) : String {
20   return buildRecordsStr(_records.slice(0, numRecords));
21 }
```

Por otra parte, el **envío de récords** al cliente está gestionado por la función *sendRecords()* (líneas 8-16). En esencia, esta función establece un socket con el cliente del juego y le envía una representación textual del registro actual de récords (ver función *getUpdatedRecords()* en las líneas 19-21).

Antes de discutir la parte de comunicaciones del cliente, es decir, del jugador, es importante resaltar que el servidor ha de estar arrancado para registrar y enviar récords. Para ello, simplemente es necesario compilarlo y ejecutarlo mediante los siguientes comandos:

```
$ haxe -neko server.n -main Server
$ neko server.n
```

La clase NetworkManager

Para centralizar la comunicación de la parte cliente, es decir, del juego, se ha diseñado la clase *NetworkManager*. Básicamente, está clase representa el único **punto de comunicación** centralizado para interactuar con el servidor. Para ello, esta clase implementa el patrón *Singleton*.

El siguiente listado muestra tanto las variables miembro (líneas 4-14) como el **método constructor** (líneas 16-29) de la clase *NetworkManager*. Las variables miembro comprenden la información básica de comunicación, tanto de la parte cliente como del servidor, y los sockets que se utilizan como puntos básicos de envío y recepción de información.

Listado 3.72: Clase NetworkManager. Variables miembro y constructor.

```
1  class NetworkManager {
2
3    // Variable estática para implementar Singleton.
4    private static var _instance:NetworkManager;
5
6    private var _server:String;              // Servidor de récords.
7    private var _port:Int;                   // Puerto del servidor.
8    private var _client:String;              // Host del cliente.
9    private var _clientPort:Int;             // Puerto del cliente.
10   private var _socketServer:Socket;        // Socket del server.
11   private var _socketClient:Socket;        // Socket de escucha.
12   private var _socketIncoming:Socket;      // Socket de peticiones.
13   private var _networkingThread:Thread;    // Hilo gestión récords.
14   private var _recordsStr:String;          // String con los récords.
15
16   private function new () {
17     _server = "localhost";
18     _port = 2048;
19     _client = "localhost";
20     _clientPort = 1024;
21     _socketServer = new Socket();
22     _socketClient = new Socket();
23     _socketClient.bind(new Host(_client), _clientPort);
24     _socketClient.listen(100);
25     _recordsStr = "";
26
27     // Hilo para atender peticiones del servidor de récords.
28     _networkingThread = Thread.create(networkingThread);
29   }
30
31   // ...
32
33 }
```

En este punto en concreto, resulta esencial discutir cómo la parte del cliente y la del servidor interactúan. Por una parte, el servidor puede recibir información mediante un socket específico (línea 21) que, por defecto, reside en el puerto 2048 (línea 18). Esto es necesario para que un cliente pueda enviar un nuevo récord o solicitar el listado de mejores récords.

Sin embargo, la parte cliente, es decir, el propio juego, también tiene la **necesidad de recibir información del servidor**. En concreto, necesita esta funcionalidad para recibir el listado de récords. Esta situación tiene dos consecuencias importantes:

1. El cliente también utiliza un socket para recibir información del servidor. En las líneas 22-24 se muestra el código para instanciar un socket que reciba información por el puerto 1024.
2. Debido a que los sockets utilizados son bloqueantes, se utiliza un **hilo de control adicional** para atender la recepción de dicha in-

Figura 3.45: Captura de pantalla del juego Tic-Tac-Toe. En el recuadro rojo se enmarca la parte de visualización de récords.

formación. La línea 28 muestra cómo se crea un hilo de la clase *cpp.vm.Thread*[30], cuya funcionalidad está integrada en la función *networkingThread()*.

Precisamente, el siguiente listado muestra la implementación de esta función. Note cómo el hilo se bloquea mediante la función *waitForRead()* de la clase *Socket* hasta que reciba información del servidor. Cuando recibe la lista de récords, la lee (línea 11) y actualiza la variable que los contiene.

[30] Se ha optado por utilizar la clase Thread del paquete cpp debido a la inexistencia de una clase más general.

Listado 3.73: Clase NetworkManager. Función networkingThread().

```
1  // Función con el código del hilo auxiliar
2  // para manejar las conexiones entrantes del servidor.
3  private function networkingThread() {
4    while (true) {
5      try {
6        // Acepta un nuevo cliente.
7        _socketIncoming = _socketClient.accept();
8        // Se bloquea a la espera de datos...
9        _socketIncoming.waitForRead();
10       // Lectura de récords.
11       _recordsStr = _socketIncoming.input.readLine();
12     }
13     // Captura excepción cuando llega a fin de buffer.
14     catch (exception : Eof) {
15       _socketIncoming.shutdown(true, true);
16     }
17   }
18 }
```

Por otra parte, la **solicitud de récords** al servidor se realiza a través de la función *getRecordsFromServer()* de la clase *NetworkManager*. Básicamente, esta función genera un mensaje específico para formalizar la petición (línea 4), concretando el host y el puerto del cliente, y lo envía mediante la escritura en el socket al servidor (línea 5).

Listado 3.74: NetworkManager. Función getRecordsFromServer().

```
1  // Solicita la lista de récords al servidor.
2  public function getRecordsFromServer () {
3    try {
4      var msg:String = "request:" + _client + ":" + _clientPort + ".";
5      _socketServer.write(msg);
6    }
7    catch (unknown : Dynamic) {
8      trace("Error de conexion con " + _server);
9    }
10 }
```

Finalmente, el siguiente listado muestra la implementación de la función *sendRecordToServer()*, cuya responsabilidad es, como su nombre indica, **enviar un nuevo récord** al servidor. Para ello, en esta función se construye el mensaje que contiene el récord (líneas 6-9) y, de nuevo, se envía a través del socket que conecta con el servidor (línea 12). Note cómo después de enviar el récord al servidor se obtiene la lista actualizada del servidor (línea 19) para poder mostrarla en la interfaz gráfica.

Integrando la funcionalidad online en Tic-Tac-Toe

La integración de la funcionalidad online desde la lógica de Tic-Tac-Toe se simplifica enormemente gracias a la existencia de la clase *NetworkManager*. En esencia, sólo es necesario **invocar a la función adecuada** cuando se requiera enviar un nuevo récord. Por ejemplo, el siguiente listado de código muestra un fragmento de código que permite enviar dicho récord cuando el jugador humano gana una partida.

Listado 3.75: Envío de un nuevo récord a través del NetworkManager.

```
1 // Envío del nuevo récord.
2 if (_board.getWinner() == Player_O) {
3   _newRecord = true;
4   NetworkManager.getInstance()
5     .sendRecordToServer(_name.text,
6             getDifficulty(),
7             Std.int((Lib.getTimer() - _newGameTime) / 1000));
8  }
```

A la hora de renderizar los récords registrados en la interfaz gráfica, sólo es necesario acceder a la copia actualizada que maneja el *NetworkManager*, como se puede apreciar en el siguiente listado.

Listado 3.76: Obtención de récords a través del NetworkManager.

```
1  // Obtiene los récords del NetworkManager,
2  // el cual los recuperó previamente del servidor.
3  var recordsServer = NetworkManager.getInstance().getRecords();
4
5  // Separación de récords.
6  var records:Array<String> = recordsServer.split(";");
7  var i:Int = 1;
8  // Muestra los 5 primeros registros.
9  for (r in records) {
10   var aux:Array<String> = r.split(":");
11   // Rellena String auxiliar con la info del récord i.
12   ++i;
13 }
```