Protocolo NanoGames

Universidad de Murcia
2º Grado de Ingeniería Informática
Asignatura: Redes de Comunicaciones
Curso 2017/2018

Trabajo realizado por Nicolás Enrique Linares La Barba.

ÍNDICE

INTRODUCCIÓN

El objetivo de esta práctica consiste en el desarrollo y el diseño de dos protocolos de comunicación de nivel de aplicación en un sistema de juegos de red, denominado NanoGames. El sistema está formado por un conjunto de clientes que desean conectarse al servidor de juegos para hacer uso de alguno de los juegos disponibles, a los que llamaremos *rooms*, representado en la figura 1.

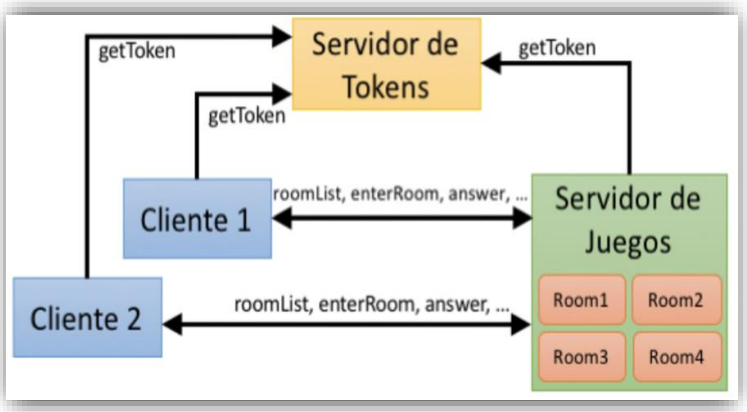


Figura 1. Esquema del sistema.

En primer lugar, se realiza la comunicación del cliente con el servidor de tokens o bróker mediante un protocolo UDP, para obtener una marca de tiempo en mili-segundos (token).

A continuación, se establece una comunicación mediante el protocolo TCP. Una vez obtenido el token, el servidor de juegos deberá validarlo, para ello le pide un token al bróker de la misma forma que lo hizo el cliente, y compara ambas marcas de tiempo: si el límite de tiempo permitido entre los tokens se supera, el servidor de juegos rechazará la conexión. A partir de este momento el usuario se encuentra conectado al servidor de juegos, y deberá ponerse un nombre de usuario que servirá para identificarlo durante toda la conexión.

Tras pedir la lista de rooms podremos elegir entre permanecer en la sala y entrar a una de las rooms, o terminar la conexión con el servidor, mediante el comando “quit”. Si decidimos entrar a una room, nos aparecerá su nombre y sus reglas, y estaremos en espera hasta que un jugador entre en la sala y pueda comenzar el juego. Al finalizar la partida podremos elegir entre salir de la room o volver a jugar si nuestro oponente también lo desea.

Se han diseñado dos juegos: RockPaperScissorsLizardSpock y MathQuestions. El primero es un juego sin turnos que consiste en el clásico “Piedra, papel o tijera” pero con dos opciones más que lo hacen más dinámico, y acabará cuando un jugador alcance la puntuación máxima. El segundo es un juego por turnos basado en preguntas sobre operaciones matemáticas, donde el jugador deberá resolver cada operación antes de que se acabe el tiempo y ganará el que haya respondido más preguntas correctamente.

ESTRUCTURA DEL PROYECTO

La siguiente figura ilustra cómo se organiza el código del proyecto y qué entidades llevan a cabo la comunicación entre sí.

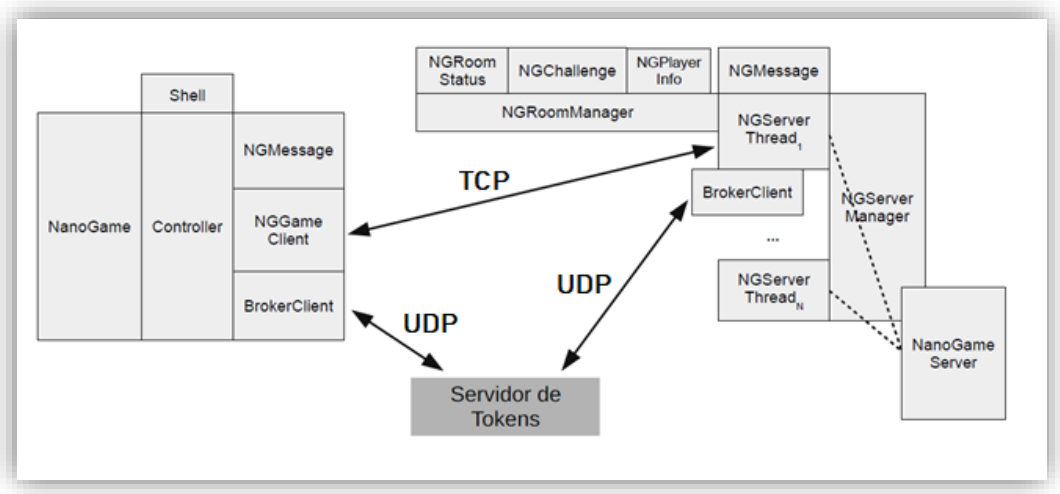


Figura 2. Estructura detallada del proyecto.

El código desarrollado está formado por los siguientes paquetes y clases:

- Paquete nanoGames.broker_client:
  - BrokerClient.java: implementación de los sockets UDP para la comunicación con el Broker.
- Paquete nanoGames.client.application:
  - main_NanoGameClient.java: Clase main del cliente que tiene la secuencia de llamadas al controlador.
  - NGController.java: Implementación del controlador que estará encargado de coordinar la comunicación con el Broker y con el servidor de juegos en función de la fase en la que nos encontremos o en función de lo que el usuario introduzca a través del shell.
- Paquete nanoGames.client.comm:
  - NGGameClient.java: Plantea la funcionalidad necesaria para comunicar el cliente con el servidor de juegos.
- Paquete nanoGames.client.shell:
  - NGCommands.java: Clase de apoyo para facilitar la implementación del shell. Contiene la definición de los tipos de comandos y de los parámetros aceptados por los mismos.

- NGShell.java: Implementación del Shell para leer las entradas del cliente.

- Paquete nanoGames.message:
  - NGMessage.java: Superclase que proporciona la abstracción de un mensaje del protocolo entre el cliente y el servidor de juegos. Contiene métodos de utilidad para generar o interpretar mensajes.
    - NGControlMessage: Subclase que define la estructura de los mensajes de operación.
    - NGGameInfoMessage: Subclase que define la estructura de los mensajes de la información del juego.
    - NGStringMessage: Subclase que define la estructura de los mensajes de texto.
    - NGTokenMessage: Subclase que define la estructura de los mensajes para el envío de los tokens.

- Paquete nanoGames.server:
  - main_NanoGameServer.java: Clase main del servidor de juegos. Se encarga de crear el socket del servidor, aceptar conexiones y generar los threads correspondientes para atender las solicitudes entrantes.
  - NGServerThread.java: Clase que implementa el hilo que debe atender cada conexión entrante, la cual procede de un cliente.
  - NGServerManager.java: Clase utilizada para gestionar los distintos juegos que hay en el servidor, así como los distintos jugadores que están conectados. Un objeto de esta clase será compartido entre todos los hilos.
  - NGPlayerInfo.java: representa la información de un jugador durante una partida.

- Paquete nanoGames.server.roomManager:
  - NGRoomManager.java: Clase abstracta (superclase) que se utilizará como base a la hora de implementar la lógica concreta de cada juego.
  - NGChallenge.java: representa la información sobre un desafío.
  - NGRoomStatus.java: representa la información sobre el estado del juego.
  - MathQuestions.java: lógica del juego de preguntas rápidas sin turnos.
  - RoomRockPaperScissorsLizardSpock.java: lógica del juego con turnos de piedra, papel, tijera, lagarto, Spock.

DISEÑO DE LOS MENSAJES DEL PROTOCOLO

FORMATO DE LOS MENSAJES: LENGUAJE DE MARCAS

Se van a emplear cuatro tipos de mensajes para toda la implementación, los cuales tienen la siguiente estructura:

- NGControlMessage

```
<message>
        <operation> opcode </operation>
</message>
```

Este mensaje será utilizado en peticiones donde solo es necesario el código de la operación correspondiente, como pedir la lista de rooms o terminar la conexión.

- NGStringMessage

```
<message>
        <operation> opcode </operation>
        <text> string </text>
</message>
```

Este mensaje se utilizará siempre que pasemos una cadena al servidor, como cuando indicamos nuestro Nick o cuando contestamos a una pregunta.

- NGTokenMessage

```
<message>
        <operation> opcode </operation>
        <token> string </token>
</message>
```

Este mensaje solo será utilizado para el envío del token ya que convierte el tipo Long en String.

- NGGameInfoMessage

```
<message>
        <operation> opcode </operation>
        <game> string </game>
        <rules> string </rules>
</message>
```

Este mensaje es utilizado únicamente para pasar la información de la room al cliente cuando entramos a ésta, así podemos separar las cadenas y asignarlas a variables diferentes una vez recibido.

CÓDIGOS DE OPERACIÓN

Para poder distinguir entre cada operación es necesario definir los códigos de dichas operaciones, cada uno con un número asociado y un nombre con deducible funcionamiento:

OP_SEND_TOKEN = 1;
OP_TOKEN_OK = 2;
OP_TOKEN_NOT_OK = 3;

Para la tarea de verificación de token.

OP_SEND_NICK = 4;
OP_NICK_OK = 5;
OP_NICK_NOT_OK = 6;

Para la tarea de verificación de Nick.

OP_DESCRIPTION = 7;

Para el envío de la descripción de una room.

OP_REQ_ROOMLIST = 8;
OP_ROOMLIST = 9

Para solicitar la lista de rooms.

OP_DISCONNECT = 10;

Para solicitar la desconexión con el servidor.

OP_ENTER_GAME = 11;
OP_ENTER_GAME_OK = 12;
OP_ENTER_GAME_NOT_OK = 13;

Para el proceso de entrar a la room.

OP_GAME_AND_RULES = 14;

OP_STATUS = 15;

OP_CHALLENGE = 16;
OP_ANSWER = 17;
OP_ANSWER_REC = 18;
OP_WAIT_TURN = 19;
OP_TIMEOUT = 20;

OP_SCORE = 21;

OP_PLAY_AGAIN_OR_EXIT = 22;
OP_EXIT_GAME = 23;
OP_PLAY_AGAIN = 24;

Operaciones utilizadas durante el juego.

EJEMPLO DE INTERCAMBIO DE MENSAJES

Inicia la conexión con el servidor y entra a una room:

CLIENTE

```
<message>
        <operation> OP_SEND_TOKEN </operation>
        <token> 63500 </token>
</message>
```

SERVIDOR

```
<message>
        <operation> OP_TOKEN_OK </operation>
</message>
```

```
<message>
        <operation> OP_SEND_NICK </operation>
        <text> toni97 </text>
</message>
```

```
<message>
        <operation> OP_NICK_OK </operation>
</message>
```

```
<message>
        <operation> OP_REQ_ROOMLIST </operation>
</message>
```

```
<message>
        <operation> OP_ROOMLIST </operation>
        <text> (1) RockPaperScissorsLizardSpock [1/2]
                (2) MathQuestions  [2/2]   </text>
</message>
```

```
<message>
        <operation> OP_ENTER_GAME </operation>
        <text> 2 </text>
</message>
```

```
<message>
        <operation> OP_ENTER_GAME_NOT_OK </operation>
        <text> Limit of players reached </text>
</message>
```

```
<message>
        <operation> OP_ENTER_GAME </operation>
        <text> 1 </text>
</message>
```

```
<message>
        <operation> OP_ENTER_GAME_OK </operation>
        <text> You are in the room </text>
</message>
```

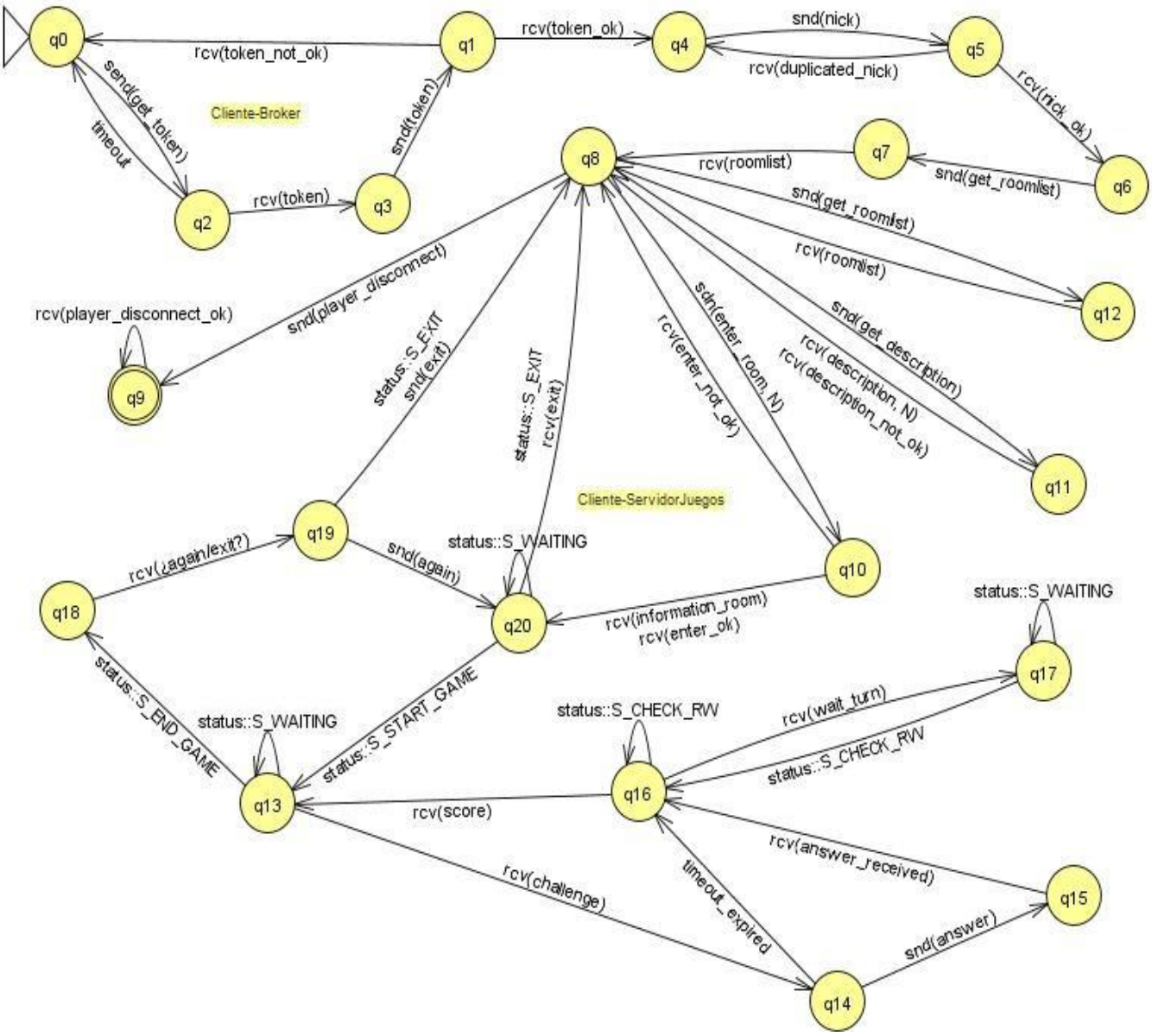
```
<message>
        <operation> OP_GAME_AND_RULES </operation>
        <game> RockPaperScissorsLizardSpock </game>
        <rules> rules... </rules>
</message>
```

DINÁMICA DEL SISTEMA

Los siguientes autómatas muestran el comportamiento del sistema y sus distintos estados para el cliente y el servidor.

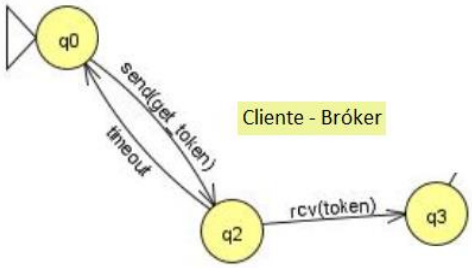
AUTÓMATA DEL CLIENTE

A continuación, se muestra el autómata completo para el proceso Cliente:



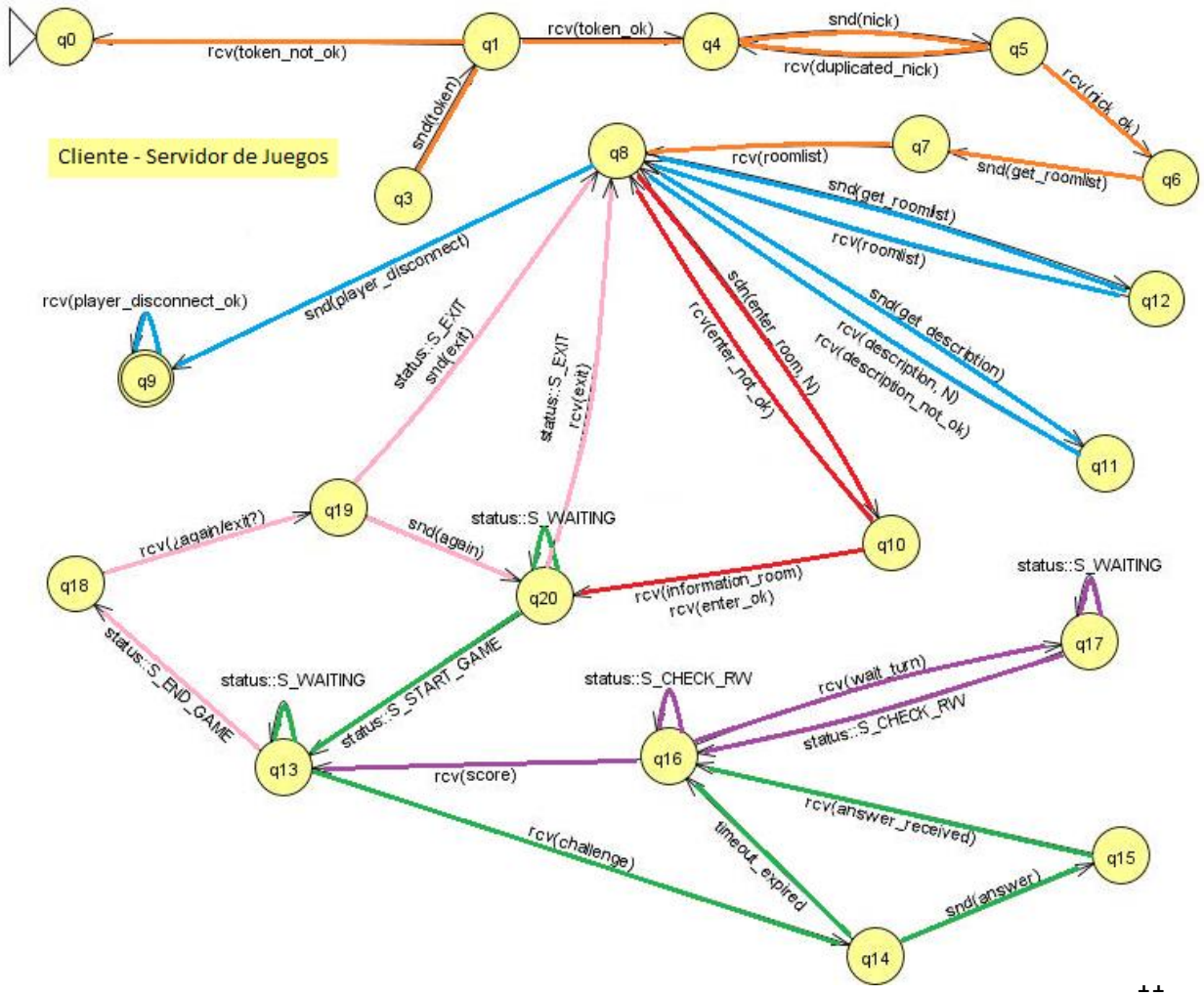
El autómata se puede dividir en dos partes:

1. La comunicación entre el Cliente y el Bróker para obtener el token, utilizando el protocolo de transporte UDP para enviar un paquete SNMP:



Nada más iniciar el cliente, es la primera tarea que se lleva a cabo (automáticamente). Al enviar la petición al servidor nos contestará con un mensaje que contiene una marca de tiempo, esta marca es el tiempo trascurrido desde que el servicio SNMP fue iniciado.

2. La comunicación entre el Cliente y el Servidor de juegos, mediante el envío de paquetes con el protocolo de transporte TCP (para mejor comprensión y facilidad en la explicación hemos resaltado con colores las transiciones de los estados):



- Naranja: Corresponde a las tareas de verificación del token y el Nick, además de la solicitud de la roomlist, los cuales deben realizarse en ese orden. Esto significa que, por ejemplo, no podremos cerrar la conexión con el servidor hasta no haber pedido una lista de los juegos disponibles. En caso de no seguir el orden del autómata el cliente nos avisará de la acción que toca en dicho momento.
- Azul: Corresponde a las acciones que podremos realizar antes de entrar a una room. Ya sea pedir la descripción de un juego o volver a pedir la lista de rooms, las cuales podremos realizar sin orden alguno y las veces que sean necesarias. O por el contrario, cerrar la conexión con el servidor, el cual informa con un mensaje que se ha cerrado la conexión.
- Rojo: Proceso de entrar a un juego, que puede ser rechazado si la room no existe o si se encuentra ya llena (esto se puede ver también en el mensaje de la roomlist que lleva un contador de los jugadores que hay en el momento en que realizamos la solicitud). Al recibir la lista de rooms, cada una tendrá un número asociado que el cliente utilizará en este momento para indicar a cuál entrar. Una vez que hayamos entrado, se nos enviará automáticamente un mensaje que contiene el nombre del juego y las reglas del mismo.
- Verde: Una vez que hemos entrado a la room elegida, si no se encuentra nadie dentro, deberemos esperar (S_WAITING) a que un jugador entre, y en ese momento comenzará el juego automáticamente (S_START_GAME), recibiendo el challenge los dos jugadores a la vez en el caso del juego sin turnos, o quedando uno en espera (S_WAITING) en el caso del juego por turnos. Y contestando, o no (timeout), al challenge.

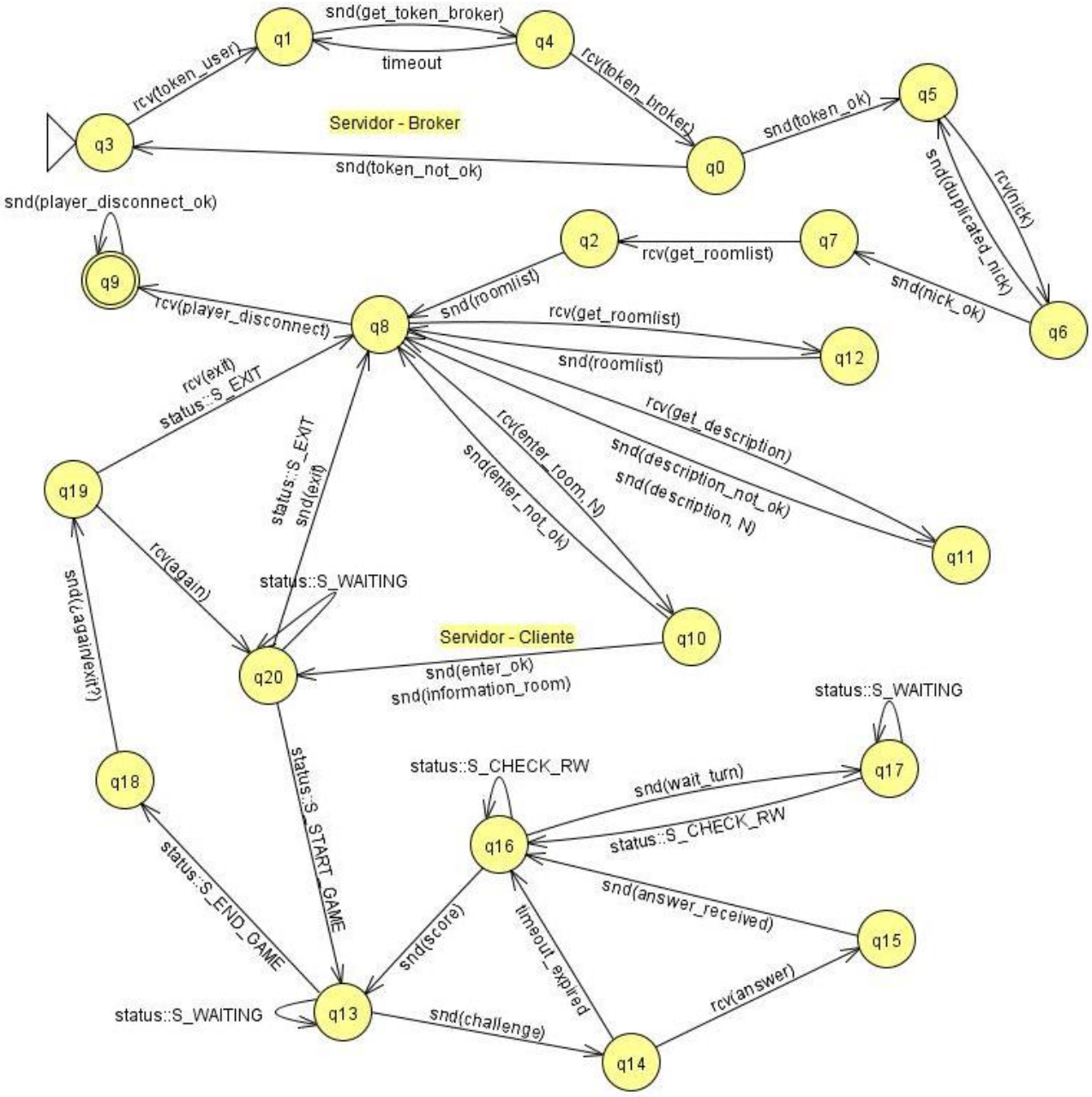
Estas transiciones "status" corresponden al estado global de la room, que los hemos añadido para que se entienda cuándo va cambiando a lo largo del juego. Estos estados serán explicados más adelante.

- Morado: Cuando contestamos a una pregunta, esperaremos a que el otro jugador responda también por lo que el jugador pasará a un estado de espera (S_WAITING). Cuando responda el oponente la room cambia de estado para comprobar quién ha ganado la ronda (S_CHECK_ROUND_WINNER), comprobación que la realiza el último en contestar.
- Rosa: Corresponde a la terminación de un juego. Cuando hemos contestado a todas las preguntas el juego vuelve a cambiar de estado (S_END_GAME) para comprobar el ganador del juego. Por último, pregunta a los jugadores si desean volver a jugar o salir.
    - Si ambos salen volvemos al estado marcado en azul.
    - Si uno elige jugar, se quedará esperando en el estado verde hasta que responda el otro. En este momento puede ocurrir que el otro jugador elija salir de la room, lo que provocará que el jugador que estaba esperando salga también. O si el segundo jugador elige jugar de nuevo, se comienza una nueva partida.

Los estados de la room siempre van a ser modificados por el último jugador que realiza una misma acción, como en el caso de responder a un challenge: no podremos comprobar un ganador hasta que el último jugador no responda. O por ejemplo el punto antes mencionado, hasta que el segundo jugador no decida si quiere volver a jugar, el otro se queda esperando.

AUTÓMATA DEL SERVIDOR

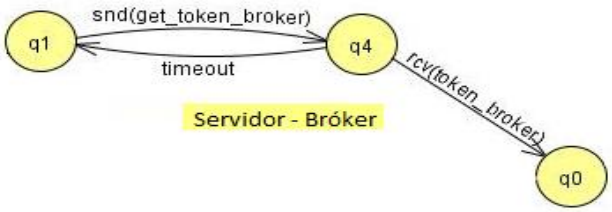
A continuación, se muestra el autómata completo para el Servidor de juegos, el cual es homólogo al del Cliente, con una pequeña diferencia:



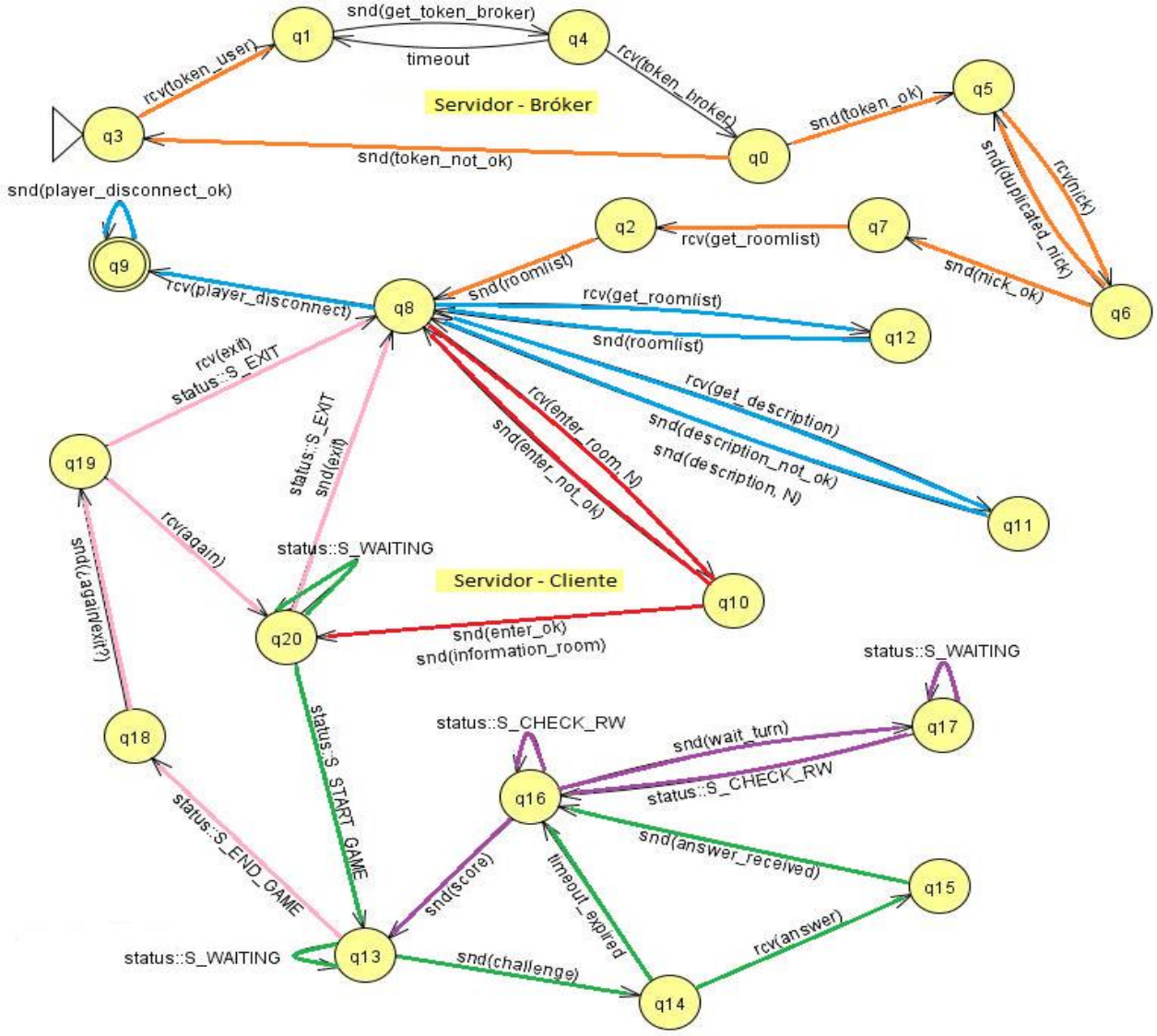
La diferencia con el autómata anterior es que el Servidor lo primero que va a hacer es recibir un token por parte del Cliente, y a continuación llevará a cabo un proceso de verificación, que consiste en pedir otro token al Bróker y comprobar que la diferencia de tiempo entre ambas marcas de tiempo no supere un valor.

Como en el caso anterior se va a dividir en dos partes:

1. La comunicación entre el Servidor y el Bróker para obtener el token de la misma manera que para el Cliente:



2. La comunicación entre el Servidor de juegos y el Cliente. Vamos a emplear los colores de las transiciones del autómata Cliente en las posiciones homólogas de este autómata para que se puedan ir comparando con facilidad.



ASPECTOS DE IMPLEMENTACIÓN

FORMATO DE LOS MENSAJES

Toda la implementación de los mensajes para la comunicación entre clientes y servidor se encuentra en el paquete *es.um.redes.nanoGames.message.*

Mediante la clase padre NGMessage podremos definir funciones comunes para todos los tipos de mensajes, como es la función *getStringMessage(),* que será utilizada por todas las demás clases específicas para obtener cada tipo de mensaje con su correspondiente formato (todos los enlaces llevan al mismo apartado):

NGTokenMessage

NGControlMessage

NGGameInfoMesaage

NGStringMessage

También tendremos la función utilizada para interpretar el mensaje recibido por el socket llamada *readMessageFromSocket()* que una vez recibido el mensaje lo convierte a String y utilizando las clases Pattern y Matcher haremos una previa comprobación del código de operación del mensaje, para saber a qué tipo de formato de mensaje corresponde, el cual seleccionamos con la estructura switch-case, y poder leer adecuadamente los demás campos con *readFromString()*.

Debemos recalcar que la expresión regular que nos proporcionan no permite leer cadenas con salto de línea, por lo que hemos añadido la siguiente expresión regular para facilitar el envío de mensajes multilínea: "<(\\w+?)>((.|\\n)*?)</\\1>"

Por último, tendremos los métodos *makeXMessage()*, utilizados tanto por el servidor como por el cliente, para crear cada tipo de mensaje, mencionados anteriormente, a los cuales se les pasarán los parámetros correspondientes a su estructura, donde todos tendrán el código de operación y otros campos.

ENVÍO Y RECEPCIÓN DE MENSAJES

❖ SERVIDOR DE JUEGOS

El ServerThread va a enviar los mensajes mediante un método llamado *sendMss()* al que le pasaremos el string y el código de operación que debe tener el mensaje para poder ser identificado. De esta manera, vamos a seleccionar son la estructura Switch-case un formato de mensaje para crearlo y poder enviarlo por el DataOutputStream. Cuando el mensaje que enviamos es siempre el mismo, el parámetro "message" va a contener una cadena vacía, pues no se va a utilizar, como puede ser en la operación OP_SCORE o en OP_GAME_AND_RULES.

Para la recepción de mensajes, tanto fuera como dentro de la room, se van a utilizar dos métodos principales: *messageController()* y *messageControllerGame()* respectivamente. Estos métodos van a recibir el mensaje por el DataInputStream y según el código de operación ejecutarán un procedimiento. Además, tendremos un método para recibir la respuesta a los challenge *receiveAnswer()*, que devolverá la respuesta para poder ser usada en *processNewChallenge()*.

❖ CLIENTE

El cliente va a enviar y recibir todos los mensajes a través de la clase NGGameClient. Se ha implementado un método para cada comunicación:

- Solicitud y recepción: como el método *verifyToken()* donde enviará el token y se quedará esperando la respuesta del server; con esta misma estructura de envío y respuesta tenemos los métodos: *registerNick(), requestRoomList(), requestDescription()* y *requestEnterGame()*.
- Mensaje sin respuesta: que consta de un único mensaje por parte del cliente, compuesto por los métodos: *sendAnswer(), sendExitGame(), sendPlayAgain()* y *sendDisconnect().*
- Recepción de mensajes: el método *checkGameMessage()* nos será de utilidad para recibir mensajes mientras estamos dentro de una room.

Esta última función será llamada desde el método *processGameMessage()* de la clase NGController, que extraerá del mensaje el código de operación para extraer los campos del mensaje y poder mostrarlo por pantalla o bien realizar una acción determinada.

MECANISMO DE GESTIÓN DE SALAS

Para explicar la gestión de las salas debemos tener clara la estructura del servidor, compuesta por tres pilares fundamentales, las clases:

- ServerThread: donde se lleva a cabo la comunicación directa entre el cliente y el servidor, así como de la lógica del autómata mostrado anteriormente.
- ServerManager: va a ser utilizada por cada ServerThread para gestionar los jugadores que se encuentran conectados y los juegos que existen en el servidor.
- RoomManager: de la que derivan todos los juegos, con métodos y variables en común para todas las salas.

Una vez hecha esta aclaración, podemos pasar a la explicación de la clase ServerManager:

Esta clase va a contener una Lista de todos los jugadores que están conectados con el servidor, actualizando la lista "players" cuando un jugador se va o inicia una conexión. También contiene una estructura Hash para almacenar los juegos, de esta manera podemos asociar un número a cada sala y que el jugador pueda decidir una acción sobre ésta indicando esa clave (este valor se mostrará cuando el cliente solicite la lista de rooms).

Cuando el cliente envíe su Nick y se confirme que es válido con *checkNick()*, en este momento se creará un objeto NGplayerInfo con el respectivo Nick y otros parámetros, y a su vez, será añadido a la lista "players" con la llamada al método *addPlayer()*.

Una vez que el jugador se ha registrado, siguiendo el autómata, pedirá la lista de rooms y el hilo, atendiendo a la petición, va a comunicarse con el manager para que le proporcione la lista de rooms mediante *getRoomList()*, que recorrerá el mapa concatenando las salas para generar la cadena que enviará el hilo. A continuación, si el jugador pide una descripción de alguna de las salas, el manager consultará la room indicada por el usuario para obtener la descripción desde una de sus variables.

En este momento el jugador puede ingresar en una de las rooms. Para ello indicará a la que quiere entrar, se comprobará siempre la existencia de la sala para evitar problemas con el método *checkRoomExists(),* y si esto es correcto intentamos meter al jugador en la sala, pues es posible que ya se encuentre llena. Para cada caso se enviará un mensaje específico al cliente, establecido en la clase ServerThread, para que esté informado de la situación.

Mientras el cliente se encuentre jugando en una sala solo va a hacer uso del método *checkStatus(),* que va a ser utilizado para comprobar en qué estado se encuentra el juego en un instante, y si ha cambiado, será devuelto para ser enviado al cliente.

Para terminar, se han implementado los métodos para borrar a un jugador, tanto de la lista de jugadores "players" con *removePlayer()* porque se haya terminado la conexión, como de la room si decide salir de ella con *leaveRoom().*

LÓGICA DEL JUEGO

Antes de comenzar con la explicación detallada de cada procedimiento implicado en el funcionamiento del juego, debemos explicar la idea principal de cómo se va a desarrollar.

Cada juego va a tener un estado global que se va a ir actualizando en determinados momentos del juego. Es un objeto de la clase NGRoomStatus, compuesto por un número asociado a un estado y un string que será enviado a los jugadores:

- S_WAITING = 0: utilizado para un estado de espera, donde se quedará comprobando continuamente si el estado de la room cambia para poder informarlo al jugador, por ejemplo, cuando entramos en un juego, debemos esperar a que cambie a S_START_GAME.
- S_START_GAME = 1: este estado implica que ya podemos jugar al juego y se actualiza cuando entra el último jugador. Cuando pasamos a este estado se enviará un mensaje específico al jugador definido por cada juego dentro de su clase. En nuestro caso se envía el oponente de cada jugador.
- S_CHECK_ROUND_WINNER = 2: se utiliza en el juego RockPaperScissors… cuando el último jugador contesta, actualiza el estado para mandar la jugada realizada, ya sea un empate o que la piedra le ganó a las tijeras, por ejemplo.
- S_END_GAME = 3: cuando llegamos a la puntuación máxima posible o hemos contestado todas las preguntas, el último jugador actualiza a este estado para indicar el ganador del juego.
- S_NOTIFY_EXIT = 4: gracias a este estado podemos controlar si un jugador ha salido del juego y poder notificarlo al otro jugador.
- S_SKIP_STATUS = 5: este estado es utilizado en el juego MathQuestions para omitir la comprobación del estado, pues no será enviado ningún mensaje específico de la acción realizada en una ronda.

Todos estos estados serán comprobados con el método *sendStatus()* del ServerThread, el cual va a comprobar con el método *checkStatus()* de NGServerManager, si el estado de la room ha cambiado para enviárselo al jugador. En cierto modo, sincroniza a los jugadores para que si uno contesta a una pregunta no ejecute otros métodos que no tocan.

❖ FUERA DE LA ROOM

Vamos a explicar el funcionamiento de la clase ServerThread cuando estamos fuera de una room. En el método *run()* de este hilo, una vez verificado el token y el Nick según el autómata, vamos a entrar en un bucle que se va a ejecutar indefinidamente hasta que decidamos terminar la conexión (variable “connectionAlive” a false) o se produzca un error de conexión. Este bucle va a ejecutar el método *messageControler()* que va a estar a la espera de recibir un mensaje por parte del cliente. Los tipos de mensaje que puede

recibir son aquellos que son válidos fuera de la room: entrar a un juego (enter N), desconectarse (quit), pedir la roomlist (roomlist) o una descripción de una sala (description N). Todo esto corresponde al estado (q8) del autómata SERVIDOR.

Cuando el mensaje recibido corresponda al de entrar a un juego y se confirme que podemos entrar, se pone la variable "room_ok" a true para pasar a ejecutar el método *processRoomMessage()* que significa que ya nos encontramos dentro del juego, concretamente en el estado (q20) del autómata.

- ❖ DENTRO DE LA ROOM

El método *processRoomMessage()* va a comenzar enviando un mensaje con las reglas del juego y se quedará esperando en *sendStatus()* hasta que entre el oponente jugador. Cuando esto ocurra, entraremos en el bucle del juego donde cada iteración del bucle corresponde a una ronda, con el siguiente orden de acciones:

1. Comprobamos si es nuestro turno, con *CheckTurn()*, si el juego no tiene turnos no hará nada.
2. Mandamos un nuevo challenge, con *processNewChallenge(),* que esperará la respuesta del cliente o por el contrario saltará el timeout.
3. Esperamos a que cambie el estado del juego con *sendStatus()*.
4. Enviamos las puntuaciones de la ronda con *sendScore().*
5. Comprobamos si hemos llegado al final del juego con *checkGameWinner(),* en su caso:
    a. Se procesa el ganador con un nuevo estado y se envía con *sendStatus()*.
    b. Se envía otro mensaje preguntando si quieren jugar de nuevo o salir de la partida, gestionando la respuesta con *messageControllerGame()*.
6. Si la partida termina y desean salir, *processRoomMessage()* termina su ejecución y volveríamos al método comentado en la sección anterior.