PRÁCTICA DE REDES DE COMUNICACIONES

José Miguel Sánchez Almagro – josemiguel.sancheza@um.es

Francisco José Gambín Tortosa – franciscojose.gambin@um.es

Introducción

Este protocolo consta de dos autómatas, uno del cliente que desea conectarse al servidor de juegos, y otro de este último.

El autómata está formado por transiciones y estados que forman parte de la comunicación entre cliente y servidor. Además, cada autómata contiene al principio una serie de estados y transiciones correspondientes al proceso de obtención y envío/comprobación del token de autenticación.

Por lo tanto, usaremos dos protocolos de mensajes distintos: uno para la comunicación entre el cliente/servidor de juegos y el servidor de token; y otro para la comunicación entre el cliente y el servidor de juegos.

El servidor de juegos está formado salas que se crearán acorde a las necesidades del servidor. Cada una de ellas tiene un juego disponible para jugar con otros usuarios. Los juegos que hay disponibles y su correspondiente sala sólo cambia cuando el administrador del servidor realiza los cambios correspondientes, no es algo que vaya cambiando continuamente en el servidor ni de manera automática.

Los dos juegos que hay disponibles en el servidor son MathManager (una versión del mostrado por los profesores), y el clásico juego del BlackJack, pero sin cartas.

Diseño

Formato de los mensajes

Obtención de tokens

A través de este protocolo tanto cliente como servidor de juegos llevaran a cabo una consulta a un servidor externo confiable denominado servidor de token (bróker) para la obtención de un token. Este token servirá como autenticación del cliente frente al servidor de juegos.

Tras la obtención del token por parte del cliente y su posterior envío al servidor de juegos, este último solicitara otro token al broker y comprobara que ambos tokens fueron obtenidos en un instante muy próximo para poder validar la conexión.

Para esta solicitud y envío de tokens se llevarán a cabo dos propuestas de formato:

Mensaje binario multiformato

4 bytes	
Type (1 bit)	**#Parámetro (31 bits)**
- 0 get_token - 1 token	Valor del token, al tener 31 bits se podrán obtener 2^{31} ms

- Cuando el campo Type tenga un 0 se estará realizando una petición al broker para la obtención de un token.
- Cuando el campo Type tenga un 1 se estará enviando un token del broker al cliente o el servidor de juegos.
- En total se envían o reciben 4 bytes.

- **Ejemplos:**

get_token

0	

token

1	242222142

Comunicación Cliente-Servidor

A través de este protocolo se establece la comunicación entre el cliente y el servidor de juegos. El cliente será un usuario que desea conectarse al servidor con el fin de jugar con otros usuarios a un juego al mismo tiempo.

Para el desarrollo de esta comunicación se empleará el formato de Field:Value. El cual se caracteriza por tener un campo identificador (Field) y otro campo de valor (Value) separados por el símbolo ':'. A continuación se mostrara un ejemplo de cada uno de los cuatro tipos de mensaje implementados en el proyecto

Mensaje de control (NGControMessage)

Field	**Value**
OP_EXIT	\n

Mensaje de juego (NGGameMessage)

Field	**Value**
OP_ROOMS	MathManager 1 BlackJack 0

Mensaje de número (NGNumberMessage)

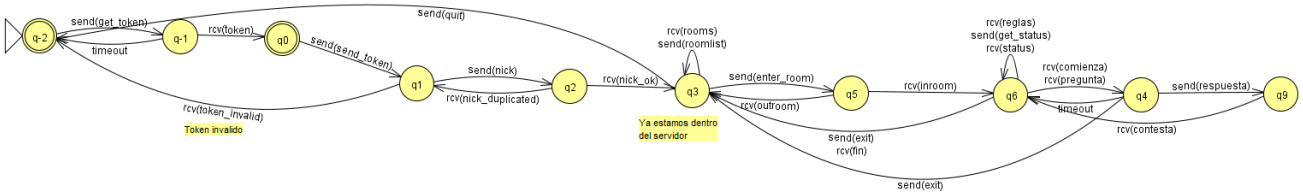
Field	**Value**
OP_SEND_TOKEN	25654

Mensaje de texto (NGTextMessage)

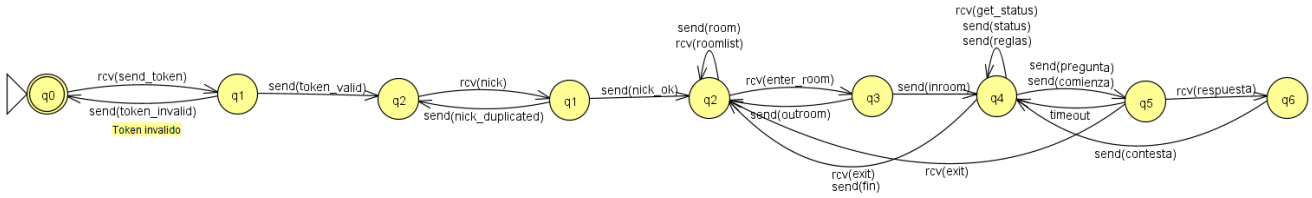
Field	**Value**
OP_NICK	Pedro

Autómatas

Autómata del cliente



Autómata del servidor



Detalles de la implementación

Implementación del formato de los mensajes

Los diferentes tipos de mensajes implementados en el juego se construyen a partir de la superclase NGMessage, y de esta forma abstraer cualquier tipo de mensaje necesario en el proyecto. Esta clase contiene una lista de constantes con todos los códigos de operación que se usaran para redirigir en función de su tipo a cada una de las subclases de NGMessage. Además esta clase contiene un método por cada tipo de mensaje que sea necesario crear.

Las diferentes subclases de NGMessage serán las encargadas de construir los diferentes tipos de mensajes que serán necesarios para el desarrollo del servidor de juegos. Estas clases tienen una implementación similar, y variaran en función de la necesidad de información que se necesita que contenga el mensaje que se quiere construir.

Sublcases:

NGControlMessage:

Clase que creará los distintos tipos de mensajes de control necesarios en el proyecto. Este tipo de mensajes solo estarán formados por el campo type. Estos mensajes se emplearan para llevar un control sobre cada uno de los cambios que se están llevando a cabo en el transcurso del juego, por ejemplo Nick_duplicated para conocer que el nick con el que se están intentando autentificar el cliente ya está siendo usado.

NGGameMessage:

Esta clase estará formada por un campo String para distinguir su tipo y por un HashMap<String, String>. Este tipo de mensaje se empleará para enviar información de cada una de las salas disponibles en el servidor de juegos. De tal forma las claves del mapa serán los identificadores de cada una de las salas, y su valor será información relativa a cada una de las salas, por ejemplo el número de jugadores de dicha sala.

NGNumberMessage:

Clase formada por el campo Type y por un campo de tipo long donde se almacenará un valor numérico. Esta clase construirá los mensajes con el token que el cliente envíe al servidor para poder acceder al servidor de juegos.

NGTextMessage:

Este tipo de mensajes serán usados para enviar tanto cliente como servidor aspectos relativos con la mecánica de los distintos juegos. Esta clase se empleara por ejemplo para que el servidor envíe mensajes con las preguntas de un juego, o para que el cliente envíe mensajes con la respuesta a una determinada pregunta. Esta clase estará formada por un campo String que contendrá el tipo de mensaje, y por un segundo campo también de tipo String donde se almacenará el texto que será enviado en el mensaje.

Mecanismo de gestión de salas

Las salas se gestionan de manera dinámica, es decir, en cada momento el servidor solo tiene en ejecución las salas que necesita.

Cuando el servidor arranca, no inicia ninguna sala por defecto, espera la solicitud por parte de los usuarios de entrar a una sala para crearla. De hecho, por muchos usuarios que haya conectados, si ninguno solicita entrar a una sala, ésta no es creada.

Si la sala de un juego se llena de jugadores, y un otro usuario quiere jugar, se creará otra sala completamente independiente a la que ya hay creada, para albergar un nuevo juego con distintos jugadores. De esta manera podemos tener de 0 a infinitas salas, no existe límite en el número de usuarios que el servidor puede albergar.

Cuando un jugador sale de una sala, esa sala se destruye automáticamente.

Implementación

Toda la gestión de las salas se realiza en la clase *NGServerManager*. Esta clase tiene dos Mapas que almacenan los usuarios y las salas. Cuando se crea un objeto de esta clase se inicializan ambos mapas. Al mapa de las salas se añade cada sala que tiene el servidor, pero sin crear ninguna sala, por lo que la lista asociada a cada sala se encuentra a *null*.

Cuando se añade un jugador con el método *addPlayer*, se comprueba si ese jugador se encuentra ya en el servidor, y si no, se añade. Con *removePlayer* eliminamos el jugador pasado como parámetro.

Cuando un jugador solicita entrar a una sala con *enterRoom*, se comprueba si no hay ninguna creada de ese juego. En ese caso se crea y se añade el jugador. Si ya hay alguna sala de ese juego creada y se puede unir un jugador, se añade. Si alguna de estas condiciones no se cumple creamos una nueva y lo añadimos a esta.

Cuando un jugador solicita dejar una sala con *leaveRoom*, la sala debe destruirse y echar a todos los jugadores que se encuentran en ella, para ello llama al método *removePlayer* de la sala, que se encarga de echar a todos los jugadores (lo veremos más adelante), y acto seguido elimina la sala.

Implementación de la lógica del juego

Se han desarrollado dos juegos, uno es un concurso de matemáticas, enviando la misma pregunta a todos los jugadores, y aquel que consiga más puntos gana. Otro es el BlackJack simplificado, con el objetivo de llegar a veintiún puntos, sumando puntos de 1 a 9 cada ronda.

MathChallenge

Este juego, al igual que ocurre con el BlackJack, se inicia en el método *registerPlayer*, cuando se alcanza el número de usuarios necesario para jugar. Se actualiza el estado del juego indicando los jugadores que hay en la sala y se inicia el primer challenge.

En el método *checkChallenge* se comprueba si se cumplen las condiciones para lanzar una nueva pregunta, o la primera: Que se cumpla el número necesario de jugadores, que todos hayan contestado, y que no se haya alcanzado el número máximo de rondas. Si se cumplen las condiciones se llama al método *nuevaOperación*, que lanza una operación matemática a resolver de manera aleatoria, y almacena su resultado. Esta pregunta es enviada a todos los jugadores.

Cuando un jugador no contesta, no se le suman puntos y se actualiza el estado indicando la puntuación actual de los jugadores.

Cuando sí contesta, se comprueba si la respuesta es válida y si es correcta. Si no es válida se toma como incorrecta y se actualiza el estado. Si es correcta se le suman dos puntos si ha sido la primera respuesta correcta, y uno si no ha sido la primera. Se actualiza el estado del jugador con esa indicación y el estado del juego con las puntuaciones actualizadas.

El método *checkStatus* devolverá el *status* actual del juego salvo que haya terminado. En ese caso calcula al ganador (si hay empate se escoge de manera aleatoria) e imprime los resultados finales. Además, pone a verdadera la variable que indica el final del juego, por lo tanto, el jugador que primero lo compruebe saldrá de la sala, y ésta sacará al resto de jugadores. Esto lo realiza *removePlayer*, que anuncia que el juego ha acabado y coloca la variable *Jugando* de todos los jugadores a falso, por lo que irán saliendo de la sala poco a poco hasta vaciarla, que será entonces eliminada.

BlackJack

Este se trata de un juego por turnos, por lo que las preguntas no se enviarán a todos los jugadores. En *registerPlayer* se reparten las cartas a todos los jugadores (gracias al método *repartirCartas*) y se activa la variable *nuevoChallenge*, para lanzar la pregunta a un jugador de si desea coger nueva carta. Conteste o no, la pregunta va pasando de jugador en jugador hasta que el juego acaba. Por lo tanto, *checkChallenge*, solo ejecutará acciones cuando se deba preguntar a un jugador. La pregunta se realiza con el método *nuevaCarta*.

Si un jugador no contesta, su estado cambia a PLANTADO, y se comprueba si ha acabado el juego.

Si hay respuesta, y ésta es afirmativa, se reparte carta a ese jugador, se comprueba si ha acabado el juego, y en caso contrario se pregunta al siguiente jugador. Si ese jugador no quiere carta se PLANTA y se pregunta al siguiente jugador si el juego no ha acabado. Si al dar una nueva carta a un jugador, la puntuación de éste es mayor que 21, el estado de ese jugador será SOBREPASA y deja de participar en el juego. Si su puntuación es 21, el juego habrá acabado al tener ya ganador.

comprobarFin irá preguntando jugador a jugador si sigue JUGANDO. Si hay más de un jugador jugando, el juego continúa, si solo hay uno jugando, pero hay usuarios que se han plantado, también continúa, y en el resto de casos el juego ha acabado.

Cuando eso ocurre, *checkStatus* calcula al ganador o a los ganadores. Se imprimen por pantalla todos los jugadores con su puntos finales y se indica si son ganadores o no, y la variable *eliminar* se pone a *true*, para dar paso a la eliminación de la sala, que se produce de manera análoga al juego *MathChallenge*.

Conclusiones

La dificultad encontrada en la práctica no se considera elevada, pero si hay un alto grado de desconocimiento, tanto a la hora de saber qué hacer en determinadas situaciones, o para saber cómo funciona gran parte del proyecto.

Se ha necesitado continuamente ayuda del profesorado para resolver múltiples dudas y problemas, y muchas veces la respuesta obtenida era ambigua o incierta.

Sin embargo, la temática de la práctica hace que sea llevadera y motivadora, incluyendo más alumno en el proyecto y haciéndolo más partícipe, desarrollando también la práctica de manera personal y otorgándole personalidad.

Por lo tanto, la idea es buena, pero la puesta en marcha ha resultado ser mala. No obstante, una vez sea revisada y se mejore, será más gratificante tanto para alumnos como para profesores.