Software Distribuït

Pràctica 1: Client-Servidor

Blai Ras i Albert Morales

Índex

1. Introducció

En la primera pràctica de l'assignatura hem realitzar un programa per jugar a "El Pòquer de Kuhn" de forma distribuïda, fent ús de *sockets* i *threads* de Java. A continuació trobareu la realització del projecte i les nostres conclusions.

2. Model Client-Servidor

Per realitzar aquest programa hem dissenyat un protocol basat en el model Client-Servidor. Concretament, tant Client com Servidor "es bloquegen" per llegir dades però no al escriure. D'aquesta manera, cadascun és independent de l'altre, però ambdós segueixen la lògica del protocol. Entre ells es comparteixen missatges (comandes) d'error, de resultat o d'informació.

3. Servidor

En primer lloc explicarem "la entitat" Servidor. Aquest té dos rols: de "banc" o "caixa" i de jugador. Dit d'una altre manera, el Servidor és l'encarregat d'establir una connexió amb varis clients (és a dir, de coordinar més d'una partida) a la vegada que fa el paper de jugador, un rol que és idèntic al del client connectat.

Per tant, Servidor està format per les següents classes:

1. **Server:** classe on trobem el *main* de Servidor. S'encarrega, per tant, de gestionar la correcte rebuda dels arguments d'inicialització, de crear l'instancia de *Logger* que farem servir durant tot el procés, i de realitzar, és clar, la connexió inicial d'un client: espera una sol·licitud de connexió, crea el *Socket*, espera una comanda d'inici (STRT) i finalment agafa l'*id* del client en qüestió. A continuació, crea el seu corresponent fil.
2. **Fil:** classe que representa un *thread.* Rep per paràmetres una comunicació, un identificador de partida i el *Logger* que ja em mencionat. S'inicia desde Server amb la comanda "*fil.start()*", la qual es refereix al seu mètode *run()*. Aquest mètode invoca una nova Partida.
3. **Partida:** aquesta és la classe que gestiona tot el transcurs d'una partida en el Servidor. Consisteix, per tant, en un *Switch-Case* on es gestionen tots els casos que poden succeir. Es poden veure tots els casos amb les seves conseqüències i el flux de tot el programa a la carpeta *Diagrames* que s'adjunta amb aquesta pràctica.
4. **LogicaPartida:** aquesta classe conté únicament un *enum* on estan inclosos els noms de tots els casos possibles. Cada estat d'aquesta *enum* et porta a un altre. En el cas de server, l'inicial és ANTE, i el final, END. Aquesta classe, per tant, conté també *getters i setters* d'estat a part d'un booleà "part" que estarà a *true* sempre i quant hi hagi partida.
5. **Comunicació:** tal i com el seu nom indica, en aquesta classe s'hi troben tots els mètodes de connexió entre Servidor i Client. Comunicació és la classe on hi derivem qualsevol enviament o rebuda d'informació, organitzada en mètodes generals (com *rebreComanda()*, que simplement llegeix un *string* de 4 bytes) o en mètodes específics, com *deal()* que reparteix aleatòriament les cartes i les retorna, a Client i a Servidor mateix.

6. **ComUtils:** per poder realitzar tot aquest enviament d'informació, necessitem la classe *comUtils*. De fet, la classe comunicació no envia res, sinó que ho deriva tot amb els paràmetres correctes a *comUtils*. A *comUtils* tenim el nivell més baix de comunicació: allà trobem els mètodes de llegir i escriure (*strings*, bytes, *chars*, *ints*...) més primitius que s'ajuden òbviament dels *InputStream* del *Socket*. La hem hagut de modificar per afegir mètodes necessaris exclusivament pel nostre protocol.

Resumint, Servidor és per una part l'administrador d'una partida i les seves accions que es duen a terme entre Client i jugador 2 (servidor) i per una altre té el rol de jugador 2 que és idèntic al de Client. Aleshores, la nostra implementació ha intentat diferenciar aquets dos rols (perquè no hi hagi "trampes") i per tant crear una estructura que separa aquestes dos facetes.

4. Client

Aquesta entitat té l'objectiu de connectar-se a un servidor i realitzar partides com si es tractes del seu oponent. En conseqüència, està format per les següents classes:

1. **Client:** classe amb el *main* del programa. Realitza el procés de la connexió amb el servidor. Si aquesta es fa efectiva, crea un nou fil on es gestionarà cada partida per aquell client.
2. **FilClient:** tal i com el seu nom indica, aquesta classe representa un *thread* d'un client. Té un mètode *run*() que és cridat des de Client. En ell, es crida l'execució d'una nova partida.
3. **Partida:** aquesta classe representa una partida des del punt de vista de Client. En ella hi ha un mètode concret, anomenat *jugant* que coordina qualsevol estat que es pot donar en una partida. Aquets estats estan representats en la classe *LogicaPartida*. Partida de client i partida de Server són gairebé idèntiques, ja que òbviament comparteixen estats ja que comparteixen protocol. Concretament, algunes de les diferències són:
    a. Server és qui gestiona el torn. Client és qui el rep.
    b. Partida de Server és qui calcula qui és el guanyador de la partida. Client és qui ho rep.
4. **LogicaPartida:** aquesta classe és idèntica a LogicaPartida de Server, ja que com ja hem dit, els estats per Client i Server són els mateixos.
5. **Comunicació:** aquesta classe té el mateix objectiu que Comunicació de Server: gestionar l'enviament o rebuda de missatges per part de Server. Per tant, comparteixen mètodes, sobretot degut a que comparteixen estats (per exemple, l'enviament de la comanda *BETT* la fan tan Server com Client, amb els mateixos paràmetres. Per tant, el mètode que l'envia és idèntic). Per realitzar aquest enviament o aquesta rebuda de missatges, Comunicació compta amb *ComUtils*.
6. **ComUtils:** aquesta classe és pràcticament idèntica a *comUtils* de Server. Té la mateixa funcionalitat: gestionar la comunicació entre les dos entitats al més baix nivell (rebuda de bytes i la seva conversió).

Resumint, Client és l'entitat que es connecta a Servidor i no sap el procés que hi ha darrera. Ell és qui rep les dades del joc (els seus diners, el guanyador, etc...) i confia en que aquestes són correctes. Client és també la "part humana" del model (tot i que ell creu que està jugant contra un adversari) i per tant és qui decideix quan desconnectar-se, quan incrementar l'aposta o deixar-ho córrer, si vol seguir jugant, etc.

5. Modes de Joc

a. Cas automàtic (tan Client com Servidor): les decisions de CHECK, RAISE, FOLD o CALL seran preses de manera aleatòria. Concretament, fem ús de la classe *Random* de java la qual en proporciona un número enter aleatori. Aquest número enter li hem donat un significat, és a dir, el relacionem amb una acció. Per exemple, 1 vol dir CHECK o 2 vol dir RAISE. D'aquesta manera, no seguim cap lògica: les decisions són aleatòries. Hem decidit que per *testejar* aquest mode es preguntarà per consola quantes partides (seguides) es voldran fer. Un cop acaba aquest nombre especificat de partides, es donarà la opció de continuar jugant una per una.

b. Cas d'IA (tan Client com Servidor): en aquest cas hem considerat que l'estratègia òptima és aquella que et minimitza pèrdues. Dit d'una altre manera, la nostra IA és la següent:
   i. Si tens una K, sempre faràs CALL. Ja no contemplo aquest cas en les següents frases:
   ii. Si comences i tens una J: faràs *fold*. Si tens una Q: faràs bet.
   iii. Si no has començat i tens una J, sempre faràs *fold*. Si tens una Q, si t'han fet *raise* sempre faràs *fold*.

c. Cas manual (Client): en aquest cas, el Client és una "persona humana", és a dir, es un humà qui tria les accions de CALL, FOLD o RISE. Les decisions, per tant, es mostren per línia de comandes.

6. Sessió de test

Per fer una posta en comú dels treballs del nostre grup de pràctiques, el dia 21 de Març és va realitzar una sessió de test en la que els diferents grups comprovàvem el funcionament dels nostres clients i servidors. En aquella data teníem desenvolupat un Client i un Servidor (*uni-thread)* funcionals, amb l'excepció de:

- SHOW estava implementat de manera primitiva, és a dir, s'imprimia un missatge per comanda i mostrava el pot total.
- END, per tant, tampoc estava complet, és a dir, no et preguntava si volies fer una altre partida. Simplement, acaba l'execució de la partida (i per tant del programa).
- No hi havia modes de joc: Servidor era automàtic sempre (sense IA) i Client era manual (et preguntava per consola de comandes sempre).
- Servidor era *uni-thread* (en veritat no, podies fer més d'una partida, però no estava "controlat" sota una mètode "*crearFils"* amb un número de partida, etc.)
- No es tractaven tots els errors segons el protocol

Vam realitzar tests amb aquells grups que usaven un ordinador de l'aula i usaven Linux. Els resultats, van ser els següents (abans, però, cal deixar clar que entre "nosaltres" el programa ens funcionava, amb ordinadors diferents. Hi ha un Log que ho demostra):

- No vam aconseguir acabar cap partida tant com a Servidors com Clients.
- El motiu principal era que l'ID de client l'enviàvem com un missatge de 4 bytes, i són 3. El nostre servidor, per tant, també esperava 4 bytes al llegir l'ID (i no 3). Això provocava que la connexió fos efectiva però que la partida no comences mai. Es pot veure en el log del grup 8.
- De totes maneres, amb el grup 11 vam aconseguir començar una partida fent nosaltres de servidor (ja que també enviava 4). No vam passar del primer estat (l'ANTE), ja que en comptes de rebre BETT rebíem "note". Sembla que l'error el tenia el grup 11. Es pot veure el flux de l'execució (el *Log*) en la carpeta "*Logs*" de la nostra pràctica. L'hem anomenat "*ServerWrongTest-11.log*".
- Amb el grup 5 vam poder comprovar (nosaltres com a servidor) que enviaven malament l'STRT ja que aconseguíem connectar-nos però no rebíem la seva senyal STRT. Després de molts *prints*, vam veure que l'enviaven massa d'hora.

Les conclusions de la fase de test van ser clares: havíem d'anar més en compte de quines comandes son de 4, 3 o 1 byte. Com a client no vam aconseguir cap connexió ja que enviaven 4 bytes. Ves a saber si hagués funcionat si haguéssim enviat 3! També ens va servir per adornar-nos de que el fil de client s'havia de crear just després d'aconseguir la connexió. En aquell dia, el creàvem després de rebre l'STRT i el *balance*.

7. Tracte d'errors

Per tractar els errors hem creat un parell de funcions, una per visualitzar el missatge de l'error i poder saber perquè ha fallat per consola i un altra que l'envia a l'altre punt de la connexió. Els errors que hem definit són els especificats al protocol, 4XX els generats pel client i 5XX pel servidor.

Per tal de saber si hi ha hagut un error en la comunicació, cada vegada que llegim una comanda mirem si ens ha arribat una de tipus ERRO. En cas afirmatiu analitzem el tipus d'error del que es tracta i el tractem de la manera que s'escau. Si al llegir una comanda no és cap de les determinades al nostre protocol o no pot ser possible que hagi arribat una en concret, el que fem és generar un error de malformació de la comanda o un de comanda inesperada. Els errors de *timeout* es produeixen quan no s'ha pogut rebre una comanda.

8. Conclusions

En aquesta primera pràctica de Software Distribuït hem aconseguit realitzar un programa per jugar al Poker Kuhn de forma distribuïda sota el model Client-Servidor amb el llenguatge de programació JAVA.

Concretament, hem realitzar un Servidor amb el rol de jugador i "caixa" que permet la connexió de més d'un client, i per tant, la realització de més d'una partida en temps real. A més a més, aquest Servidor "té memòria" en el sentit de que recorda les dades del seus clients sempre i quan "no es desconnecti" de la corrent (es deixi d'executar el programa). Amb "recordar les dades dels seus clients" ens referim a que guarda el seu ID i els seus diners.

En segon lloc, hem realitzar “un” Client el qual és capaç de connectar-se a un Servidor i simular una partida de Pòquer amb la informació que s’escau: els diners que posseeixes, la carta que has rebut, la carta del contrincant, el guanyador de la partida... Aquest Client es pot emular com una màquina automàtica (mode automàtic), com una màquina automàtica però “intel·ligent” (mode d’estratègia òptima) i finalment com un humà (les decisions son triades per una tercera persona a través de l’indicador de comandes).
Realitzar aquest joc ens ha servit per veure com funcionen els *Threads* i *Sockets* de JAVA, quina estructura segueix el model Client-Servidor i com funciona una aplicació distribuïda simple.