Documentazione del backend

Federico Viscomi

10 luglio 2011

Indice

1 Introduzione **1**

2 Global table manager **1**

3 Local table manager **3**

4 Gtm ed ltm console UI **4**

5 Global chat **4**

6 Database Manager **4**

7 Single table manager e logica del gioco **4**

8 Local Bot **5**

9 Command line player **5**

10 Ristrutturazione della comunicazione e della mutua esclusione **5**

11 Test del solo backend **7**

1 Introduzione

Questo documento descrive quali sono state le varie iterazioni che hanno portato all'implementazione del backend e alcuni dettagli di quanto è stato sviluppato in ogni iterazione.

2 Global table manager

In questa iterazione si è implementato il global table manager o gtm. Il gtm ha diverse responsabilitá:

swarm : Gestire un insieme(chiamato in seguito swarm) di local table managers o ltm. In particolare gtm deve:

- Permettere di aggiungere e rimuovere ltm dallo swarm.
- Controllare la consistenza dello swarm. A questo scopo fa polling degli ltm chiamando

```
LocalTableManagerInterface.isAlive()
```

ogni

`GlobalTableManagerInterface.POLLING_DELAY`

millisecondi.

- Inoltrare le richieste delle servlet ad un ltm in modo da bilanciare il carico di lavoro tra gli ltm. A questo scopo gli ltm hanno associata una coppia di interi: il primo indica il numero di tavoli correntemente gestiti dall'ltm e il secondo indica il numero massimo di tavoli gestibili dall'ltm. Quando il gtm deve scegliere un ltm sul quale creare un nuovo tavolo, prende dallo swarm un ltm tra quelli che hanno un carico di lavoro piú basso. Il carico di lavoro o workload di un ltm è definito come il rapporto tra i tavoli gestiti e il numero massimo di tavoli gestibili.

Ricapitolando lo swarm deve mantere associazioni tra interfacce ltm e coppie di interi, deve fornire operazioni di:

- Aggiunta con chiave una interfaccia ltm.
- Rimozione con chiave una interfaccia ltm.
- Modifica del numero di tavoli gestiti da un ltm.
- Scelta di un ltm con workload minimo.

Si è deciso di usare un ArrayList di record:

```
interfaccia ltm, numero di tavoli gestiti, numero di tavoli gestibili
```

Tale lista viene mantenuta ordinata con workload decrescente. Questa struttura dati minimizza il tempo di esecuzione dell'operazione piú frequente cioè la scelta di un ltm. Sia n il numero di ltm allora le complessitá in tempo al caso pessimo delle operazioni dello swarm sono:

aggiunta : O(n) perché bisogna controllare che non ci siano due interfacce ltm uguali.

rimozione O(n) perché in questo caso non vengono confrontati i workload ma le interfacce.

modifica O(n) perché anche in questo caso non vengono confrontati i workload ma le interfacce.

scelta O(log(n)) perché: per scegliere un ltm con carico minimo basta prendere l'ultimo elemento della lista peró dopo bisogna incrementare il suo numero di tavoli gestiti e riordinare la lista.

tables : Mantenere alcune informazioni riguardo tutti i tavoli in Cupido. Tali informazioni sono tutte e sole quelle necessarie ad un giocatore che vuole unirsi ad una partita o che vuole vedere una partita e sono contenute nella classe

```
TableInfoForClient
```

Ogni tavolo nel gmt è identificato da un descrittore di tavolo cioè una istanza di

```
TableDescriptor
```

Il gtm deve memorizzare associazioni tra descrittori di tavolo e informazioni del tavolo, inoltre deve avere a disposizione le operazioni di:

- Aggiunta usando come chiave un descrittore di tavolo.
- Ricerca usando come chiave un descrittore di tavolo.
- Rimozione usando come chiave un descrittore di tavolo.
- Reperire una parte dei tavoli.

Si è scelto di memorizzare le informazioni dei tavoli in una

```
LinkedHashMap<TableDescriptor, TableInfoForClient>
```

L'hash è calcolato sui campi del descrittore del tavolo. In particolare il descrittore del tavolo contiene: una stringa che identifica un ltm, un intero che identifica il tavolo all'interno dell'ltm. In questo modo le prime tre operazioni richiedono tempo costante. La quarta operazione restituisce una quantitá di tavoli non superiore a

```
GlobalTableManagerInterface.MAX_TABLE_LIST_SIZE
```

I tavoli da restituire vengono copiati in una istanza di

```
ArrayList<TableInfoForClient>
```

e vengono scelti in modo casuale. In questo modo la dimensione dei dati inviati al client è costante. Al caso pessimo la complessitá in tempo di questa operazione è lineare nel numero di tavoli perché per scegliere una parte dei valori da una tabella hash in modo uniformemente distribuito bisogna eventualmente iterare su tutti essi. Si è scelto di non fornire una operazione per reperire tutti i tavoli perché il numero dei tavoli potrebbe essere troppo grande e quindi anche il traffico di rete conseguente. Un'altra strategia presa in considerazione è quella di un iteratore remoto ma è stata scartata perché genera troppe chiamate remote e perché la lista dei tavoli puó cambiare tra due chiamate.

Gtm è usato attraverso RMI quindi il multithreading è gestito in automatico dalla JVM. La mutua eclusione sul gtm è gestita semplicemente usando metodi

```
synchronized
```

E' stata presa in considerazione un'altra strategia di mutua eclusione meno restrittiva cioè: tutte le operazioni hanno accesso esclusivo all gtm con l'eccezione che si permettono piú operazioni in lettura allo stesso tempo. Tuttavia questa soluzione presenta il problema della starvation delle operazioni di scrittura quindi si è deciso di non adottarla.

3 Local table manager

In questa iterazione si è implementato il local table manager o ltm. Ltm si occupa di una porzione dei tavoli di Cupido. In particolare ltm fornisce le operazioni seguenti:

- Creare componenti table chiamate in seguito single table manager o stm.
- Rimuovere un stm quando una partita finisce.
- Cercare un stm quando un giocatore vuole unirsi ad una partita o guardare una partita.

Ltm mantiene in una tabella hash gli stm indicizzati da un intero che identifica l'stm in modo univoco nell'ltm. In questo modo le operazioni dell'ltm hanno complessitá in tempo costante. Per quanto riguarda il multithreading e la mutua eclusione valgono le stesse cose dette per il gtm. All'avvio l'ltm legge dal file di configurazione

```
localTableManager.config
```

le seguenti variabili:

- `MAX_TABLE`: specifica il massimo numero di tavoli che questo ltm puo' gestire. Il valore di default é

`LocalTableManagerInterface.DEFAULT_MAX_TABLE`

- `RMI_REGISTRY_ADDRESS`: specifica l'indirizzo del registro rmi sul quale cercare il gmt. Il valore di default é

  `LocalTableManagerInterface.DEFAULT_RMI_REGISTRY_ADDRESS`

- `DATABASE_ADDRESS`: specifica l'indirizzo del database. Il valore di default é

  `DatabaseInterface.DEFAULT_DATABASE_ADDRESS`

4 Gtm ed ltm console UI

In questa iterazione sono stati implementati due interpreti di comandi da console minimali per gtm ed ltm. I comandi forniti permettono di vedere lo stato interno del gtm e degli ltm. Quindi questi interpreti hanno una valenza per lo piú in fase di debug e test. Potrebbero essere usati anche in un deploy reale per operazioni di manutenzione ma si è scelto di non farlo perché non serve: in particolare non ci sono requisiti che vietano di riavviare il gtm o gli ltm in fase di esecuzione.

5 Global chat

In questa iterazione è stata implementata la chat globale lato backend. La chat globale è un buffer di messaggi di dimensione

`GlobalChatInterface.MESSAGE_NUMBER`

Gli utenti di Cupido possono solo scrivere messaggi o reperire un certo numero di ultimi messaggi. Quindi i client fanno polling della chat globale e non ricevono notifiche quando un client invia un messaggio nella chat globale. Questo perché si prevede la possibilitá di avere un numero elevato di utenti e quindi non conviene che la chat globale memorizzi un interfaccia di notifica per ognuno di essi e che invii notifiche a questa interfaccia ogni volta che arriva un nuovo messaggio nella chat globale.

6 Database Manager

In questa iterazione è stata implementata la classe DatabaseManager che fornisce le operazioni descritte in SRS.

7 Single table manager e logica del gioco

In questa iterazione è stato implementato stm. Stm si occupa di una singola partita. In particolare stm:

- gestisce giocatori e bot: permette ad un giocatore di unirsi alla partita di lasciare la partita e di giocare, permette al creatore del tavolo di aggiungere bot, sostituisce un giocatore con un bot se questo lascia la partita dopo l'inizio, notifica i giocatori di quello che succede nella partita.
- gestisce i visitatori: permette ad un visitatore di aggiungersi alla partita o di lasciare la partita, invia ai visitatori le notifiche di quello che succede nella partita.

- gestisce le carte: da le carte e controlla che le mosse dei giocatori siano valide.
- gestisce i punteggi dei giocatori.

8 Local Bot

In questa iterazione è stato implementato un bot, cioè un giocatore automatico. Quando il bot riceve le carte, le ordina dalla piú alta alla piú bassa e poi in base al seme. Quando deve passare le carte sceglie le prime tre che ha. Quando deve giocare una carta sceglie la prima carta valida che ha.

9 Command line player

In questa iterazione è stato implementato un interprete di comandi testuale per un utente di Cupido. L'interprete fornisce una ampia gamma di comandi: creare un tavolo, ottenere la lista di tutti i tavoli, unirsi ad una partita, vedere una partita e giocare una partita. Per una descrizione dei comandi si veda la documentazione della classe

```
PlayerConsoleUI
```

I comandi possono essere presi da console o da un file di input. Usando un file di input si puó far giocare una partita a degli utenti in modo completamente automatico.

10 Ristrutturazione della comunicazione e della mutua esclusione

In questa iterazione è stata re-implementata la comunicazione fra le varie componenti del backend per evitare deadlock.

In particolare, all'inizio di questa iterazione tutte le notifiche venivano inviate in modo sincrono tramite RMI, e all'arrivo di una notifica, essa veniva gestita direttamente nel metodo chiamato attraverso RMI. Questo significa che, ad esempio, quando una servlet chiamava con RMI un metodo del tavolo, esso inviava le notifiche relative ed aspettava che fossero gestite prima di restituire il controllo alla servlet.

Questa implementazione dava vari problemi: le servlet non potevano chiamare metodi del tavolo durante la gestione di una notifica (perché i metodi del tavolo sono synchronized e si sarebbe creato un deadlock). Inoltre, quando una servlet chiamava un metodo del tavolo, il tavolo non poteva notificare la stessa servlet, perché questo avrebbe causato problemi di concorrenza all'interno della servlet.

Per risolvere questi problemi, è stata aggiunta una ActionQueue ai tavoli e ai bot, per gestire rispettivamente le notifiche in uscita e in ingresso.

ActionQueue è una classe creata durante questa iterazione. Questa classe supporta l'accodamento di azioni e le esegue in un secondo momento, in un thread dedicato.

Ogni tavolo ora usa una ActionQueue per le notifiche in uscita: al posto di inviare le notifiche alle servlet (o ai bot) direttamente, esse vengono accodate nella ActionQueue ed eseguite dopo aver restituito il controllo alla servlet (o al bot).

I bot usano anch'essi una ActionQueue, ma per le notifiche in entrata: in questo modo, quando l'ActionQueue del tavolo invia le notifiche, non deve aspettare che esse vengano gestite dai bot, cosa che potrebbe comportare delle chiamate al tavolo da parte dei bot (ad esempio, quando il bot deve giocare una carta). Il diagramma di struttura composita della parte del backend che gesiste una partita è in figura 1.

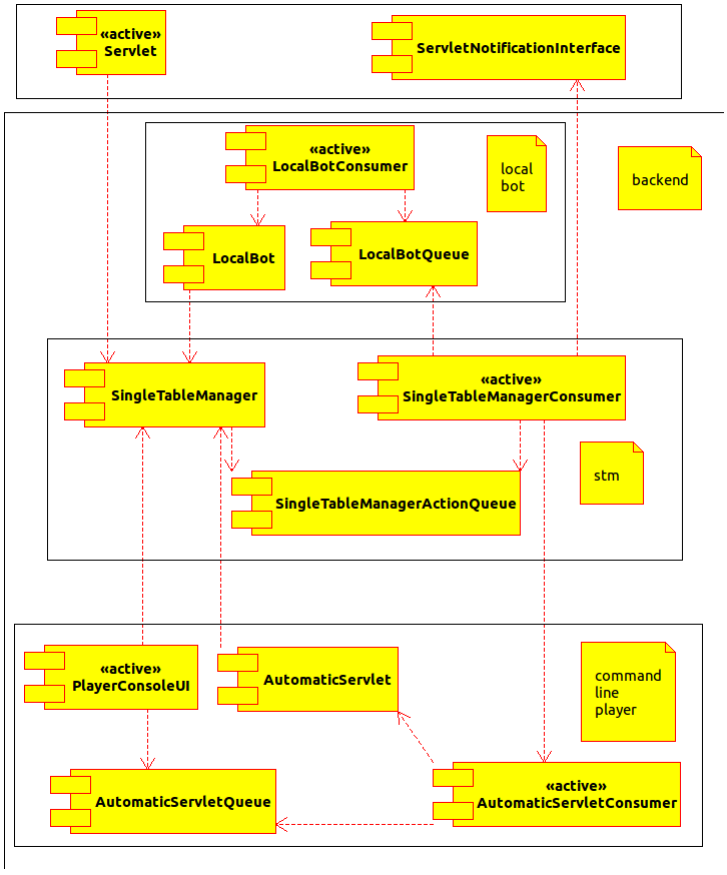

Figura 1: Struttura composita di stm

11 Test del solo backend

La cartella cupiboBackendImpl/test contiene i file per i test automatici del backend. Sono stati eseguiti i seguenti cinque test automatici sul backend:

- Un giocatore crea un tavolo, aggiunge tre bot e gioca. Questo test si esegue con il comando

  `./cupidoBackendImpl/test/test1`

  il quale esegue un giocatore a riga di comando che legge i comandi dal file

  `./cupidoBackendImpl/test/createTableAndAddThreeBot`

- Un giocatore crae un tavolo e aggiungie due bot, in seguito un altro giocatore si unisce alla partita. Questo test si esegue con il comando

  `./cupidoBackendImpl/test/test2`

  il quale esegue due giocatori a riga di comando che leggono i comandi dai file:

  `./cupidoBackendImpl/test/createTableAndAddTwoBot, ./cupidoBackendImpl/test/joinATable`

- Un giocatore crea un tavolo, aggiunge tre bot e gioca, nel frattempo un visitatore guarda la partita. Questo test si esegue con il comando

  `./cupidoBackendImpl/test/test3`

  il quale esegue un giocatore come nel primo test e in piú esegue un visitatore da riga di comando che legge i comandi dal file

  `./cupidoBackendImpl/test/view`

- Un giocatore crea un tavolo, aggiunge tre bot, gioca per un po e poi abbanona la partita. Questo test si esegue con il comando

  `./cupidoBackendImpl/test/test4`

  il quale esegue un visitatore ed un giocatore da riga di comandi che legge i comandi dal file

  `./cupidoBackendImpl/test/createTableAndAddThreeBotLeave`

- Un giocatore crea un tavolo e aggiunge due bot, nel frattempo un altro giocatore si unisce alla partita e in seguito i due giocatori giocano. Dopo qualche turno il giocatore non creatore lascia la partita e viene rimpiazzato da un bot. Questo test si esegue con il comando

  `./cupidoBackendImpl/test/test5`

Tutti i test sono andati a buon fine.