DOCUMENTAZIONE IMPLEMENTAZIONE DI RETE DI MYSHELFIE

Generalità

Il modello del gioco è stato modificato mediante l'aggiunta del metodo adjacentItemsCheck(Player player) necessario al controllo delle adiacenze delle tessere Oggetto nella Libreria del giocatore passato come parametro, e relativamente al numero identificativo delle carte Obiettivo Comune uniformandole alle denominazioni delle risorse grafiche.

È stato deciso di implementare nella nostra versione di MyShelfie sia la connessione mediante protocollo RMI (Remote Method Invocation) che attraverso l'utilizzo della dinamica dei socket di rete. L'utilizzo dei socket ricalca il modo di funzionamento del protocollo RMI mediante l'utilizzo di un ClientSkeleton e di un ServerStub. Abbiamo inoltre aggiunto due funzionalità aggiuntive: la possibilità di giocare partite multiple sullo stesso server e la persistenza delle partite.

L'utente all'avvio della applicazione, dopo aver inserito dettagli circa il funzionamento del gioco in sé (quali nickname e numero di giocatori) può decidere a propria discrezione con quale tecnologia di rete connettersi all'app sul server.

Nota: avendo ancora da scrivere alcune parti di codice non abbiamo descritto il funzionamento delle meccaniche di rete mediante i sequence diagram UML ma descrivendolo a parole nelle seguenti pagine.

Prima connessione del client al server

All'avvio dell'app del client

1. Vengono richieste le informazioni sopracitate.
2. A seconda del tipo di tecnologia di connessione scelta, l'app del client avvia AppClientRMI o AppClientSocket passando le restanti informazioni precedentemente raccolte come parametri a queste.
3. Nel caso di connessione RMI
    a. AppClientRMI esegue il lookup del rmiregistry ricercando l'AppServer al quale connettersi
    b. Nel caso in cui si sia scelto di creare una nuova partita, AppClientRMI esegue il metodo connect() sul riferimento ad AppServer ottenuto dal precedente lookup. Questa chiamata restituisce un Server (di tipo ServerImpl) su cui verrà ospitata la

partita. Successivamente istanzia un nuovo Client (di tipo ClientImpl) passando anche il riferimento al Server di gioco appena ottenuto. Se invece ci si unisse ad una nuova partita, viene ancora eseguita la chiamata connect() ma, qualora non siano presenti partite in attesa di giocatori sull'AppServer, viene comunicato all'utente che deve creare lui una nuova partita. Il susseguirsi delle operazioni è poi il medesimo.

   c. La creazione di ClientImpl comporta la chiamata del metodo register() sul server di gioco che istanzia un nuovo modello e un nuovo controller oppure lega il client che desidera registrarsi al modello e al controller già esistenti qualora si entrasse in una partita in attesa di giocatori. Infine, viene eseguita la chiamata al metodo startGame() che, qualora non si sia raggiunto il numero di giocatori necessari ad avviare la partita, comunica agli utenti di aspettare finché tale numero non sia raggiunto. Altrimenti inizia la partita.

4. Nel caso di connessione mediante l'utilizzo di socket di rete
   a. AppClientSocket istanzia un ServerStub sull'indirizzo e porta dell'AppServer
   b. AppClientSocket istanzia un nuovo Client (di tipo ClientImpl) passando come parametro il ServerStub creato in precedenza.
   c. La creazione del ClientImpl comporta la chiamata del metodo register() sul ServerStub
   d. Il metodo register() su ServerStub prima crea il socket di rete sull'ip e la porta dell'AppServer, l'ObjectOutputStream e l'ObjectInputStream e poi, una volta creati, scrive sul canale di comunicazione i parametri relativi alle informazioni raccolte in fase di avvio dell'AppClient
   e. Il relativo ClientSkeleton in ascolto sul server riceve quanto scritto sul canale di rete dal ServerStub del client e esegue l'istanziazione del Server di gioco (di tipo ServerImpl) e la chiamata al metodo register() su questa istanza ottenuta. Le azioni del metodo sono le stesse del caso in cui la connessione sfruttasse il paradigma RMI, con la sola differenza che ad essere registrato non è l'istanza del ClientImpl (che vive sul client) ma il relativo ClientSkeleton.
   f. L'AppServer pone poi in ascolto sul canale di rete in un thread dedicato il ClientSkeleton appena registrato, mentre il Server di

gioco esegue la chiamata a startGame() che agisce in modo analogo al caso di connessione RMI con la sola differenza che l'update viene prima invocato sul ClientSkeleton, e successivamente questi scrivendo sul canale di comunicazione informa il corrispondente ServerStub lato client di attendere i giocatori rimanenti oppure che la partita si sta avviando. Il ServerStub propaga poi l'informazione al ClientImpl mediante l'opportuno metodo update() che a sua volta notifica la propria view di cominciarlo all'utente.

<u>Dinamica delle connessioni durante lo svolgimento di un turno generico</u>

Quando il numero di giocatori della partita è stato raggiunto, l'invocazione del metodo startGame() indipendentemente dalla tecnologia di connessione tra il server e il client, notifica tutti gli utenti che il numero necessario di giocatori è stato raggiunto e invoca un metodo dedicato alla scelta casuale del primo giocatore. Il modo in cui viene scelto è affine alle dinamiche di gioco di fine turno, indi per cui si può assumere la descrizione delle connessioni di un turno generico senza necessariamente indicare il primo.

1. Nel caso di connessione RMI

Nel modello il currentPlayer è cambiato rispetto al turno precedente. Viene effettuata la seguente catena di chiamate (nota che game è il modello del gioco)

i. Da setCurrentPlayer(Player player) (inizio)
ii. setChangedAndNotifyListener(game)
iii. notifyObserver(game)
iv. Per ogni ClientImpl registrato come osservatore viene chiamato il metodo update(new GameView(), clientID) con GameView vista immutabile del modello e clientID identificativo univoco del giocatore che deve effettuare questo turno
v. Se il ClientImpl ha identificativo di gioco diverso da quello del giocatore in turno, viene chiamato il metodo update(game.getGameBoard().getGameGrid()) sulla view del giocatore che mostra a schermo la plancia di gioco corrispondente all'inizio del turno del nuovo giocatore.

vi. Se il ClientImpl ha identificativo di gioco uguale a quello del giocatore in turno, viene chiamato il metodo update(game) sulla view del giocatore che a sua volta esegue il metodo run() che mostra le opzioni di gioco disponibili
vii. Vengono chiesti al giocatore il numero di tessere Oggetto da prelevare dalla Plancia, il loro ordine di inserimento e la colonna della propria Libreria in cui inserirle.
viii. A questo punto viene chiamato il metodo setChangedAndNotifyListener(coords) con coords lista di coordinate delle tessere scelte già ordinata secondo l'ordine di inserimento dettato dall'utente. Si ha quindi una sotto-catena di chiamate
    1. NotifyObserver(this, coords) con this questo ClientImpl
    2. Viene eseguito il metodo update(this, coords) sull'unico osservatore di questa view (ossia il Server di gioco)
    3. Il metodo update(this, coords) sul ServerImpl di gioco chiama il medesimo update del controller
    4. Il controller, eseguendo questo metodo update(), controlla se il ClientImpl è registrato sul Server di gioco come utilizzatore di un Player e che sia il Client di turno. Successivamente esegue la chiamata game.getCurrentPlayer().pickItems(coords, game.getGameBoard.getGameGrid(), game.getValidGrid()). I parametri sono le coordinate delle tessere Oggetto che l'utente vuole prendere, la griglia della Plancia e la matrice che identifica se le posizioni indicate sono valide
    5. Il metodo pickItems(...) esegue le operazioni necessarie a realizzare quanto indicato dal giocatore
    6. Qualora si verificasse un errore (posizioni invalide o già scelte ecc...) viene lanciata un'eccezione raccolta dal controller, il quale esegue una chiamata al metodo update(String) del Client ricevuto come parametro, setta come prossimo giocatore il

giocatore del turno corrente che ha sbagliato, facendogli ripetere il turno.

7. Qualora tutto andasse correttamente, qui si interrompe la catena di chiamate relativa alla scelta delle tessere Oggetto.

ix. Viene ora chiamato il metodo setChangedAndListener(column) con column il numero della colonna selezionata in cui inserire le tessere Oggetto. Si ha quindi una sotto-catena di chiamate

1. notifyObserver(this, column) con this questo ClientImpl
2. Viene eseguito il metodo update(this, column) sull'unico osservatore di questa view, ossia il ServerImpl di gioco
3. Il metodo update(this, column) sul ServerImpl di gioco chiama il medesimo update del controller
4. Il controller, eseguendo questo metodo update(), controlla se il ClientImpl è registrato sul Server di gioco come utilizzatore di un Player e che sia il Client di turno. Successivamente esegue la chiamata game.getCurrentPlayer().putItemsInShelf(column)
5. Il metodo putItemsInShelf(...) esegue le operazioni necessarie
6. Qualora la colonna selezionata fosse piena viene lanciata una un'eccezione raccolta dal controller, il quale chiama il metodo update(String) sul ClientImpl registrato sul server relativo al giocatore di turno che ha inserito le tessere in una colonna invalida.
7. Qualora tutto andasse correttamente, il controller chiama il metodo manageTurn(ClientImpl) sulla propria istanza di TurnHandler passando come parametro il Client che ha appena eseguito il turno.
8. Il metodo manageTurn(ClientImpl) esegue i controlli di fine turno relativi ai punti da carte Obiettivo Comune, Fine Partita, eventuale riempimento della Plancia ecc... Alla fine, esegue un update(String) all'utente di turno notificandolo della fine del proprio

turno. Poi chiama il metodo nextTurn(Player) con parametro il giocatore che ha appena finito il turno di gioco.

9. Il metodo nextTurn(Player), a meno che non sia la fine della partita, si calcola chi deve essere il prossimo giocatore, chiama il metodo setCurrentPlayer(otherPlayer) sul modello, con otherPlayer il successivo giocatore.
10. Il metodo setCurrentPlayer(Player) impone il nuovo giocatore di turno (fine), ripetendo il ciclo di chiamate per il prossimo turno.

2. Nel caso di connessione tramite socket
    a. La catena di chiamate è identica al caso di connessione con RMI fino al punto iii)
        i. Per ogni ClientSkeleton registrato come osservatore viene chiamato il metodo update(new GameView(), clientID) con GameView vista immutabile del modello e clientID identificativo univoco del giocatore che deve effettuare questo turno
        ii. Il metodo update(GameView, int) di ClientSkeleton esegue in ordine oos.writeObject(GameView), flushAndReset(oos), oos.writeObjecy(int), flushAndReset(oos) scrivendoli sul canale di rete
        iii. In ascolto sull'altro lato è posto il relativo ServerStub lato client in attesa di messaggi in arrivo. Qualora il ois.readObject() non lancia eccezioni, verifica se il messaggio ricevuto sia effettivamente una GameView e solo allora esegue ancora una volta un ois.readObject() scansionando un Integer dal canale di rete (il clientID inviato assieme alla GameView). Esegue quindi il metodo update(GameView, int) sul relativo ClientImpl convertendo l'Integer a int.
        iv. La catena di chiamate prosegue come indicato dal punto v) al capo 1) punto viii) del protocollo RMI
        v. Viene eseguito il metodo update(client, coords) sull'unico osservatore di questa view, ossia il ServerStub.

vi. Il metodo update(client, coords) di ServerStub esegue in cascata oos.writeObject(client), flushAndReset(oos), oos.writeObject(coords), flushAndReset(oos)
vii. Dall'altro lato del canale di rete è in ascolto il corrispondente ClientSkeleton in attesa di messaggi sulla rete. Qualora il ois.readObject() non lanci un'eccezione, il metodo discrimina se il messaggio letto è effettivamente un Client oppure una List<int[]>. In tali casi viene eseguita una macchina a stati in modo tale da avere salvati sia il Client che la lista prima di effettuare la chiamata update(client, coords) sul relativo server.
viii. Le successive chiamate sono le stesse dei punti 3) sottosezione vii) a 1) sottosezione ix) del protocollo RMI
ix. Viene eseguito il metodo update(client, integer) sull'unico osservatore di questa view, ossia il ServerStub
x. Il metodo update(Client, Integer) di ServerStub esegue in cascata oos.writeObject(Client), flushAndReset(oos), oos.writeObject(integer), flushAndReset(oos)
xi. Dall'altro lato del canale di rete è in ascolto il corrispondente ClientSkeleton in attesa di messaggi sulla rete. Qualora il ois.readObject() non lanci un'eccezione, il metodo discrimina se il messaggio letto è effettivamente un Client oppure un Integer. In tali casi viene eseguita una macchina a stati in modo tale da avere salvati sia il Client che l'Integer prima di effettuare la chiamata update(Client, Integer) sul relativo server.
xii. Le chiamate successive sono le stesse descritte dal punto 3) sottosezione ix) al punto 10) sottosezione ix) del protocollo RMI

<u>Dinamica delle connessioni nel caso di fine partita</u>

Si prenda come riferimento di inizio l'esecuzione del metodo nextTurn() a fine turno

1. Nel caso si siano verificate le condizioni per la fine della partita (calcolate ogni turno dal turnChecker) viene eseguita una chiamata al metodo gameOverHandler()

2. Il metodo sbriga operazioni di servizio (quali aggiunta di punti sulle carte obiettivo personale, i punti per le adiacenze ecc...) per poi dichiarare un vincitore. A questo punto
3. Sia nel caso di connessione RMI che mediante socket si ha la seguente catena di chiamate (nota: esse differiscono per il punto c.)
    a. Esegue la chiamata setStatus(String) per ogni giocatore del modello con parametro il messaggio di fine partita (punti, vincitore, saluti ecc...)
    b. Il metodo setStatus(String) chiama il metodo setChangedAndNotifyListener(String)
    c. La catena di chiamate coincide con quella per l'update di una stringa.
    d. Quando viene ricevuto dal ClientImpl, si verifica se è il messaggio di fine partita mediante l'uso di un marker
    e. Si chiama l'update(String) del messaggio di fine partita sulla view, la quale mostrerà il messaggio suddetto a schermo
    f. Viene eseguita una chiamata di System.exit, facendo terminare il Client.

<u>Dinamica delle connessioni nel caso di notifica mediante Stringa come parametro</u>

Si prenda come riferimento la notifica di una stringa dal server al client in quanto il viceversa viene eseguito una volta sola (per la notifica del nickname) e ha svolgimento analogo.

1. Viene chiamato il metodo setChangedAndNotifyListener(String)
2. Viene chiamato il metodo notifyObservers(String)
3. Nel caso di connessione RMI
    a. Viene eseguita la chiamata del metodo update(String) sul riferimento a ClientImpl registrato come osservatore
    b. Nel metodo update(String) su ClientImpl vengono discriminate tre casistiche:
        i. Qualora il messaggio fosse quello di fine partita vengono eseguite in cascata due chiamate rispettivamente sul messaggio ricevuto dal server e una seconda con un messaggio di "FINE PARTITA" sul metodo update(String)

della view del ClientImpl. Viene allora eseguita una chiamata a System.exit

ii. Se il messaggio è una notifica di “colonna selezionata sbagliata” viene eseguito l’update(Integer) sulla view del ClientImpl con parametro la colonna 0

iii. Qualora il messaggio non rientri in uno dei casi precedenti viene eseguita una chiamata del metodo update(String) sulla view del ClientImpl

c. Il metodo update(String) della view esegue una System.out.println per mostrare a schermo il messaggio corrispondente. Qualora questa stringa contenga una sottostringa “Try again” significa che le tessere Oggetto precedentemente scelte dell’utente in questione non sono valide, viene quindi eseguita una chiamata a this.gameActionOnGameboard() che ripropone al giocatore una nuova scelta delle tessere Oggetto.

d. Il metodo update(Integer) ricevendo come parametro un “Integer = 0” mostra a schermo la scritta “Invalid column selection” ed esegue una chiamata al metodo gameActionOnShelf() che ripropone all’utente una nuova scelta della colonna in cui inserire le tessere Oggetto precedentemente selezionate.

4. Nel caso di connessione basata su socket
    a. Viene chiamato il metodo update(String) sul ClientSkeleton corrispondente
    b. Viene eseguita la chiamata oos.writeObject(msg) con parametro la stringa ricevuta dal metodo update(String) chiamato precedentemente il quale scrive sul canale di rete la stringa msg, eseguendo poi il relativo flush and reset
    c. Dall’altro lato del canale di rete, il corrispondente serverStub lato client è in ascolto mediante il metodo receive(Client) sullo stesso canale in attesa di messaggi dalla rete. Se eseguendo un ois.readObject() viene effettivamente letto qualcosa dal canale, allora si discrimina se ciò sia effettivamente una stringa. Qualora lo sia, verrà eseguita la chiamata al metodo update(String) sul ClientImpl di riferimento. Nel caso in cui la stringa fosse un messaggio di “non presenza di partite disponibili sul server”, dopo la notifica al ClientImpl del messaggio, verrà eseguito un System.exit (in quanto non essendoci partite in attesa di giocatori

si presuppone che l’utente riavvii l’applicazione per crearne lui una nuova).

d. Il proseguimento della catena di chiamate è il medesimo del punto b) della versione con protocollo RMI