# RoundWord

Progetto per il Corso di Sistemi Distribuiti, a.a. 2012-2013

MATTEO BRUCATO e MIRO MANNINO
Università di Bologna

14 giugno 2013

**Sommario**

## Indice



## 1 Introduzione

La presente relazione tratta della realizzazione del progetto per il corso di sistemi distribuiti, dalla sua ideazione, alle scelte progettuali, agli aspetti implementativi, non senza includere difficoltà incontrate e ciò che abbiamo imparato da questa esperienza formativa.

Un gioco molto famoso tra i bambini di ogni età, ma giocato anche tra adulti senza limiti di età, consiste nel formare sequenze di parole collegate tra di esse attraverso sillabe. Il gioco è molto semplice e non richiede nessuna strumentazione né attrezzature particolari, e può esser giocato in qualunque contesto. Per giocare, basta avere una comune conoscenza del vocabolario italiano ed essere in grado di suddividere le parole in sillabe.

Non essendo a conoscenza del nome del gioco (nonostante lo abbiamo giocato sin da bambini), abbiamo deciso di chiamarlo *RoundWord* per il presente progetto.


Figura 1: Una screenshot del gioco dove Giacomo detiene il turno e pertanto deve inserire una parola. Notare anche che Marco si è ritirato, oppure ha avuto un crash.

## 1.1 Regole del gioco

All'inizio del gioco, i giocatori decidono di comune accordo una sequenza di gioco, ovvero l'ordine dei turni (giocando dal vivo, ci si mette spesso in cerchio). Ogni giocatore, al proprio turno, produrrà una singola parola. Il primo giocatore sceglie una parola qualunque, che deve essere *presente nel dizionario* italiano utilizzato dal gioco. Dal secondo giocatore in poi, scatta la regola per la scelta della parola:

- Sia $w$ la parola prodotta dal giocatore precedente. Ad esempio "VERIFICA".
- Sia $w = w_1, w_2, \ldots, w_n$ la sua suddivisione in sillabe. Ad esempio, "VE", "RI", "FI", "CA".
- Sia $w' = w'_1, w'_2, \ldots, w'_m$ la parola inserita dal giocatore corrente, e la sua suddivisione in sillabe.
- La parola $w'$ è *valida* se è presente nel dizionario e se $w_n = w'_1$, ovvero se la prima sillaba della nuova parola è uguale all'ultima sillaba della parola precedente. Ad esempio, "CARTA", "CARI" e "CARATTERISTICA" sono tutte parole valide, mentre "RIONE", "RESPIRARE", "MANO" sono tutte parole non valide.

Ciò quindi crea una sequenza di parole tutte collegate tra di essere per mezzo dell'ultima e della prima sillaba tra ogni coppia di parole.

Per rendere più interessante il gioco, non basta che una parola sia *valida* affinché si possano guadagnare dei punti per il proprio turno. Vi è un altro requisito fondamentale: la parola deve essere *nuova*. Ovvero, la parola inserita non deve essere stata inserita precedentemente da alcun giocatore, durante il corso della corrente partita. Quindi, ogni giocatore deve tenere memoria della sequenza di parole generate da tutti i giocatori, ed evitare di riproporre una parola già inserita precedentemente. Per aiutare l'utente a capire le parole che sono state inserite, vengono visualizzate le ultime 6 parole.

Per complicare ulteriormente il gioco, e per evitare che il gioco duri troppo, ogni giocatore ha un limite di tempo entro il quale può proporre la sua parola.

Ogni giocatore, al proprio turno, può guadagnare o perdere punti, secondo le seguenti regole:

- Se la parola non è presente nel dizionario perde 80 punti.
- Se la parola non è valida perde 80 punti.
- Se la parola è *valida*, ma non è *nuova*, perde 100 punti.
- Se il giocatore non è riuscito a produrre la parola entro il limite di tempo stabilito per un singolo turno, perde 60 punti.
- Se la parola è valida e non è stata proposta precedentemente durante la stessa partita, il giocatore guadagna un numero di punti che dipende dalle lettere utilizzate (e.g. 1 punto per la lettera A, 8 per la lettera G), dalla lunghezza della parola (i.e. 5 punti in più per ogni lettera inserita dopo la quinta), e dai millesimi di secondo impiegati (i.e. 0.02 punti per ogni millisecondo che l'utente aveva ancora a disposizione per rispondere).

Da notare che, nel caso un giocatore abbia una perdita dei punti, la parola proposta non viene aggiunta alla lista delle parole utilizzate, il prossimo giocatore dovrà pertanto riprendere dalla parola dell'ultimo giocatore che era riuscito ad inserire una parola.

Un giocatore può ritirarsi in qualunque momento. In quel caso, il gioco continua tra i giocatori rimanenti, richiudendo il cerchio nella maniera naturale. Il gioco finisce quando un giocatore rimane da solo, oppure quando nessun giocatore riesce a produrre parole valide e nuove per un intero ciclo di gioco.

## 1.2 Obiettivi del progetto

L'obiettivo principale del presente progetto è la realizzazione del gioco in un ambiente distribuito, in cui varie entità distribuite (che chiameremo *peer*, vista la loro intrinseca natura client/server) comunicano tra loro attraverso una rete asincrona, non affidabile, come quella di Internet. In questo tipo di contesto sono necessari coordinazione tra i peer e gestione di stati condivisi. Infatti, lo stato del gioco, composto di elementi che devono essere condivisi tra i vari peer in maniera coerente, non può risiedere in un unico peer che funge da nodo centrale.

Sono state fatte delle semplificazioni per non dovere affrontare errori difficili da risolvere, che sono le stesse riportate dalle specifiche del progetto: per prima cosa si assume di avere delle comunicazioni affidabili (ovvero non è possibile che si guastino), ed i processi possono avere solamente guasti di tipo crash. Ciò significa che assumiamo sempre che un peer che non riusciamo a contattare, per un certo periodo di tempo, sia un peer che ha subito un crash, piuttosto che un peer che non riusciamo a contattare per via di un fallimento della rete, o per via di una temporanea indisponibilità del peer stesso.

Alcuni problemi correlati a queste assunzioni sono comunque stati risolti. Si pensi ad esempio che un peer che precedentemente dichiarato crashato, poiché lento o per ritardi di comunicazione, torna a comunicare con i peer della rete, viene comunque ignorato: il giocatore vedrà un messaggio che lo avverte del fatto che la propria connessione è troppo lenta e che quindi è stato cacciato fuori la partita, come avviene in molti giochi commerciali.

Inoltre, secondo le specifiche, come paradigma di comunicazione, è stato utilizzato remote invocation, in particolare il Remote Method Invocation (RMI) di Java. Ciò fornisce un modo semplice ed ad alto livello per comunicare tra i vari peer, che ha semplificato molto l'implementazione delle comunicazioni tra di essi, in quanto tale middleware permette di mascherare l'eterogeneità tra le varie macchine. Ciononostante non è sufficiente da solo per fornire tutte le funzionalità di un ambiente distribuito, come la coordinazione, la gestione degli errori e del tempo.

# 2 Aspetti progettuali

## 2.1 Interpretare RoundWord come un sistema distribuito

Interpretare questo gioco nell'ambito dei sistemi distribuiti non è particolarmente difficile. Infatti, ogni giocatore controllato da un *peer*, in una rete distribuita in stile peer-to-peer (senza overlay). I turni a ciclo suggeriscono l'idea di un protocollo in stile *token-ring*, dove il detentore del turno viene equiparato al *leader* attuale, e la leadership viene passata al prossimo peer allo scadere di ogni turno. Lo *stato condiviso* tra i peer partecipanti consiste in:

- La lista dei peer attivi (non crashati)
- La lista dei punteggi dei rispettivi giocatori
- La lista completa delle parole inserite dai giocatori.
- L'identità del detentore del turno attuale (ovvero del leader attuale)

## 2.2 Architettura astratta

Il sistema è composto di tre aree ben distinte: modello del gioco, controllori dei giocatori e comunicazione. La parte di modellazione del gioco si compone di tutte quelle classi che realizzano il gioco, senza prendere in considerazione il fatto che questo possa essere giocato in rete, e senza una concezione di interfaccia GUI. Tale modello ignora pertanto come i vari giocatori interagiscono, fornendo solamente dei metodi per cambiare lo stato del gioco, e generando eventi per segnalare tali cambiamenti. I controllori dei giocatori possono essere di tre tipi: un'interfaccia grafica che comunica con il giocatore reale permettendogli di poter giocare, un peer remoto che interagisce sfruttando la parte di comunicazione, o un giocatore automatico (principalmente utilizzato per debug, ma che comunque non si esclude possa essere utilizzato come un rimpiazzo per i giocatori che hanno subito un guasto).

La parte di comunicazione ha come incarico quello di interagire con gli altri peer presenti nel sistema, in modo da comunicare le parole che sono state inserite dal giocatore locale, e comunicare al modello del gioco quali sono quelle inserite dagli altri peer. Ogni peer ha pertanto il compito di gestire un particolare giocatore, eseguendo delle azioni opportune per ogni tipo di evento generato dal gioco. Utilizza due parti distinte che lo completano: una parte che funge da client (chiamata *client side*) ed una parte che funge da server (chiamata *server side*). Il client side non è un entità che si occupa di mandare i messaggi, e non è altro che un thread fornito di una coda di messaggi in uscita. Il server side invece è la parte che si occupa di ricevere tali messaggi e di eseguire le azioni opportune, in base al tipo di messaggio ed ai dati in esso contenuti. Facendo un esempio per riassumere, ogni peer, per gestire l'evento del giocatore locale che ha inserito una parola, crea un messaggio (o più messaggi) specificando il peer destinatario, e quindi lo invia aggiungendolo alla coda di uscita del proprio client side; successivamente tale messaggio viene processato e viene inviato chiamando l'opportuna funzione del server side del destinatario.

## 2.3 Problemi affrontati

### 2.3.1 Inizializzazione della partita

La parte di inizializzazione della partita è stata progettata utilizzando un server centrale. Questo perché richiesto dalle specifiche del progetto in sé, ma è un assunzione abbastanza realistica: è necessario dare una disponibilità a giocare utilizzando un server noto (che abbia un indirizzo conosciuto).

Tale server, chiamato *registrar*, è un server HTTP che provvede a raccogliere gli indirizzi IP dei vari giocatori, le loro porte utilizzate ed i loro nickname. Tale server è configurato per creare partite di $N$ giocatori, e quindi lascia iniziare la partita dopo che sono state fatte $N$ richieste.

Al momento del raggiungimento delle $N$ richieste, il registrar manda, ad ogni giocatore, la lista degli utenti che parteciperanno alla partita. Tale lista $[p_1, p_2, .., p_N]$ è ordinata ed identica per tutti, stabilendo quindi quale sarà l'ordine con il quale avverranno i turni. Tale liste di giocatori arrivano ai vari peer in momenti differenti, e per questo motivo è necessario che tutti i peer siano a conoscenza del fatto che tutti gli altri abbiano ricevuto tale lista. Se così non fosse, il peer che detiene il turno, iniziando a giocare, potrebbe assumere falsamente che alcuni peer abbiano subito un guasto (questi prima di ricevere la lista dei giocatori non possono far nulla). Il peer che deve per primo iniziare a giocare (ha il turno), invia pertanto un messaggio di *hello*, attendendo che questo ritorni indietro. Nel caso in cui tale messaggio non dovesse ritornare (si utilizza un timeout), la partita viene annullata, per fare sì che se ne possa ricreare una nuova.

#### 2.3.2 Comunicazione

# 3 Aspetti implementativi

## 3.1 Architettura

### 3.1.1 Modellazione del gioco

Il gioco in sé viene rappresentato da poche classi: `GameTable`, `Word`, `Player` e `Dictionary`.

La classe `Word` rappresenta una parola del gioco. Consiste di una semplice stringa, ma ha numerosi metodi utili per scomporla in sillabe e determinarne il valore in punteggi. Inoltre essa è serializzabile, poiché deve poter essere inviata lungo la rete. La classe `Dictionary` rappresenta un dizionario, e contiene semplici metodi per il lookup di una parola. La classe `Player` rappresenta un giocatore. Esso ha un nickname, un punteggio, uno stato (attivo o non attivo), ed altre informazioni necessarie per il gioco. Esso può essere controllato da un giocatore reale presente sullo stesso peer, da un agente che gioca automaticamente, oppure controllato un peer remoto. La classe `GameTable` rappresenta il tavolo di gioco. Essa contiene informazioni quali: la lista delle parole che sono state inserite, come anche la lista dei giocatori, ed il giocatore che detiene il turno. Inoltre ha dei metodi per modificare lo stato del gioco, come quello per determinare chi è il prossimo detentore del turno, o per inserire una nuova parola. Tale classe non ha un'idea del fatto di come viene controllato un giocatore; seguendo il design pattern chiamato *Observer Pattern*, la classe è stata progettata per generare degli eventi, quali ad esempio l'inserimento di una nuova parola, il cambio del turno, o la terminazione del gioco. Sono poi presenti diversi listener che rispondono in maniera passiva a questi eventi seguendo delle azioni opportune. Queste stesse entità che fungono anche da listener, possono comunque avere dei ruoli attivi, come ad esempio l'inserimento di nuove parole.

L'entità più semplice, chiamata `FakePlayer`, che rappresenta un agente che gioca automaticamente, consiste ad esempio in un semplice loop, che inserisce una parola, e che rimane in attesa fin tanto che il `GameTable` non genera un evento, in modo da segnalare che tale agente è il detentore del prossimo turno.

Al contrario, un giocatore reale ha tanti oggetti, che rappresentano varie componenti dell'interfaccia. Ognuno di essi può fungere anche da listener, in modo da poter aggiornare le varie informazioni che gli competono. Inoltre, l'interfaccia ha anche un ruolo attivo, poiché inserisce le parole che sono state scritte dall'utente, calcolando anche il tempo impiegato per farlo.

Più complicato è invece un giocatore controllato da un peer remoto. Il `Peer`, che funge anche da listener, può reagire ad un evento, quale ad esempio l'inserimento di una nuova parola, mandando dei messaggi per informare anche agli altri peer di tale evento. Può poi inserire nuove parole sulla base dei messaggi ricevuti dagli altri peer, cambiare lo stato di un giocatore (ad esempio a causa di un crash), oppure decidere chi possederà il prossimo turno in base sullo stato degli altri peer.

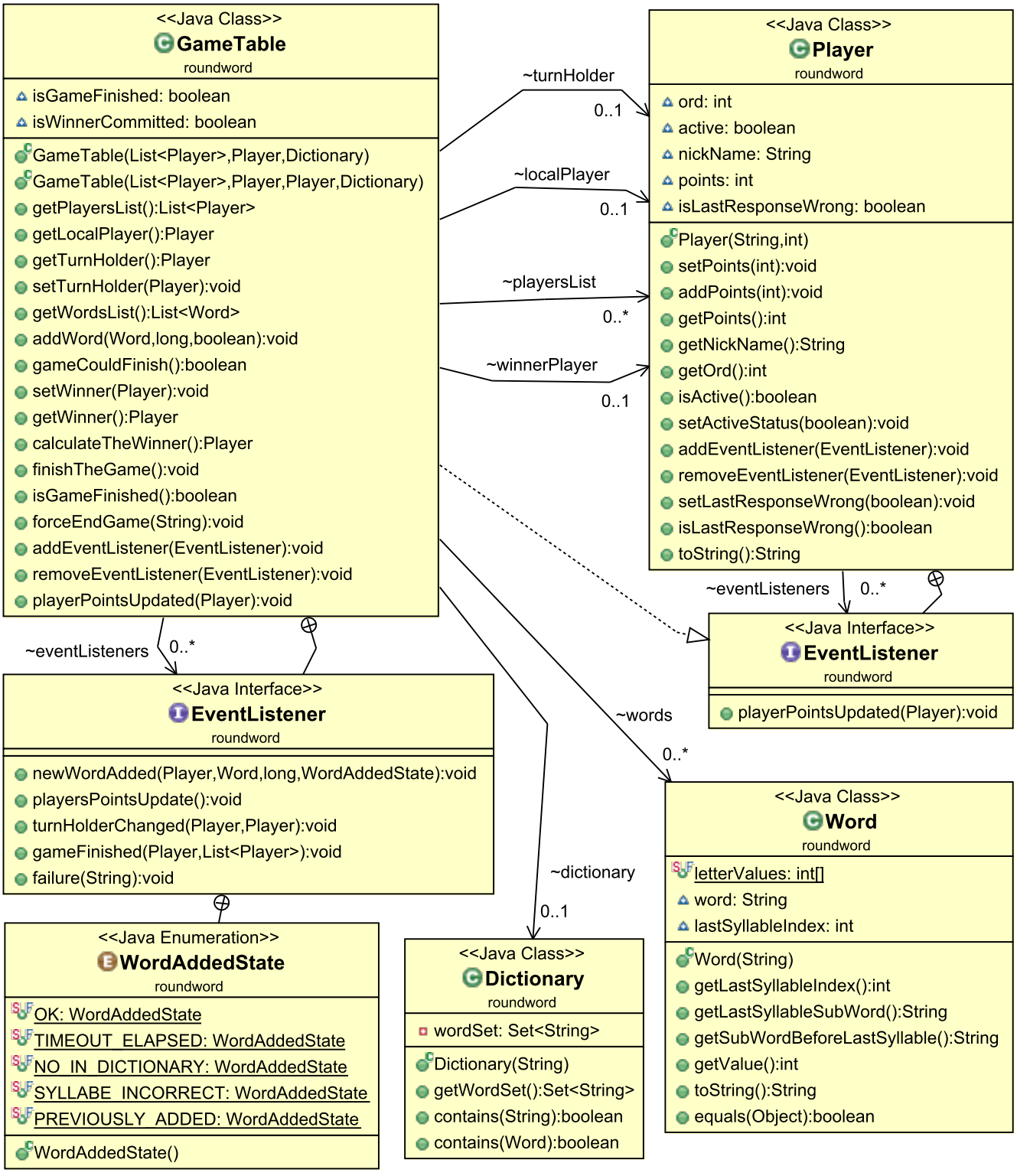


Figura 2: Il diagramma delle classi di tutta la parte di modellazione del gioco, escludendo quindi tutte le classi per far partire il gioco, le classi di comunicazione, le classi per i giocatori automatici, e le classi per l'interfaccia con l'utente.

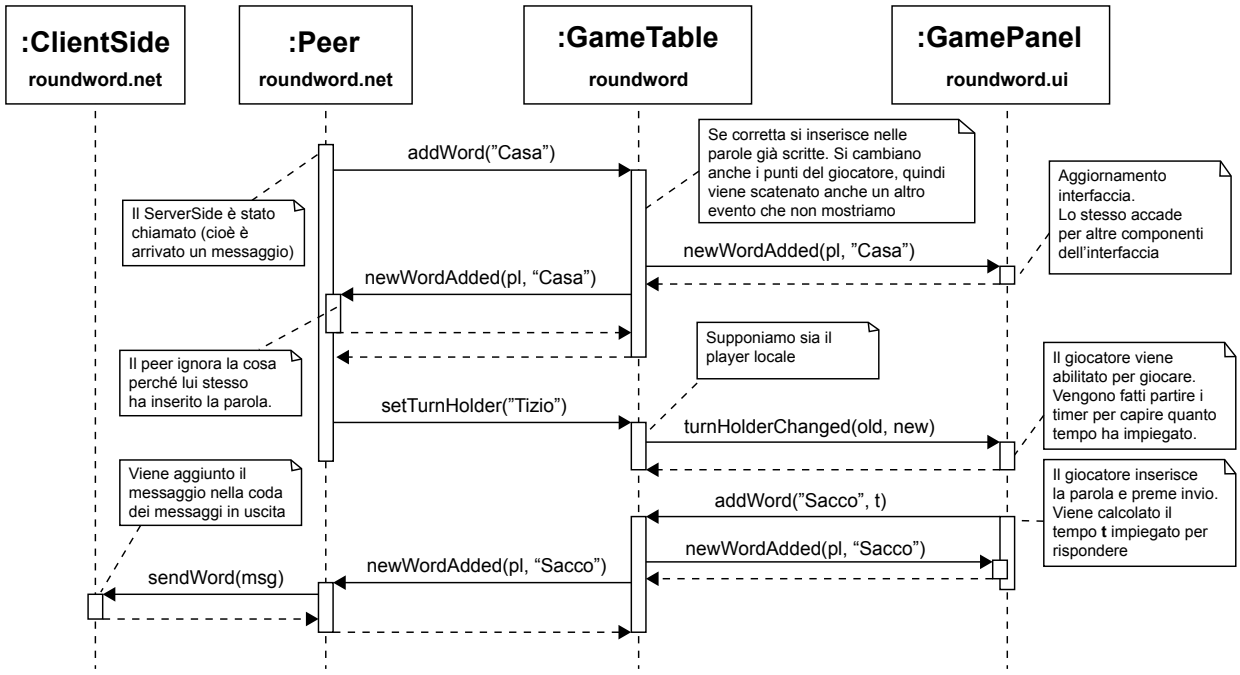


Figura 3:

In figura 3.1.1 possiamo osservare il diagramma delle classi di tutta la parte di modellazione del gioco. Questa, come si può intuire, comprende solamente una piccola parte dell'intera applicazione. che corrisponde agli elementi che abbiamo cercato di descrivere in maniera approfondita. Da notare che anche il `Player` può generare un evento (che avviene al cambio del punteggio), ma questi vengono esclusivamente intercettati dal `GameTable` che a sua volta genera l'evento `playerPointsUpdate()`. Quest'ultimo serve solamente alla parte dell'interfaccia per aggiornare visivamente il punteggio dei giocatori.

### 3.1.2 Comunicazione

I `Peer` è un entità che controlla un determinato giocatore (`Player`). Esso, al momento della creazione di un gioco, si registra come listener per intercettare gli eventi del `GameTable`. Fatto ciò, ogni volta che l'utente locale ad esempio inserisce una parola, crea un messaggio di tipo `WordMsg` e lo aggiunge alla coda dei messaggi di uscita del `ClientSide` chiamando il metodo `send_msg()`.

Il `ClientSide` è un thread con metodi e campi in più rispetto ad un normale thread, nasce e muore insieme al peer, provvedendo alla gestione dell'invio dei messaggi del peer. Si compone di una coda: è stato deciso di utilizzare la `BlockingQueue`, ma senza limitazioni nella dimensione, per fornire una coda che potesse essere utilizzata da più thread (il thread del peer e quello del `ClientSide`). Il metodo `send_msg_rmi(Msg)` altro non fa che “eseguire” il messaggio, raccogliendo tutti gli eventuali errori che possono essere generati da Java RMI.

Utilizziamo il termine “eseguire” poiché un messaggio, in generale, è in realtà una classe che contiene un metodo `execute()` che chiama il metodo remoto presente nel server side del peer destinatario. Ogni tipo di messaggio esegue quindi un metodo differente, e i parametri del messaggio vengono utilizzati come argomenti per i metodi che vengono richiamati.

Il `ServerSide` è invece una classe, istanziata solamente una volta per ogni peer, che costituisce l'insieme dei metodi che possono essere richiamati da ogni messaggio. Facendo un esempio, il metodo `execute()`, del messaggio di tipo `WordMsg`, esegue il metodo del `ServerSide` del destinatario chiamato `word()`, specificando come argomenti quelli propri del messaggio: chi lo ha generato (chi detiene il turno), la parola inserita, il tempo impiegato per inserire la parola, se esiste un vincitore (cioè la partita può terminare), e la lista dei peer morti che si sono accumulati lungo il percorso.

C'è quindi un disaccoppiamento tra il peer e l'invio dei messaggi. Questo poiché, per implementare il progetto, è stato utilizzato *Java RMI*, e di conseguenza inviare un messaggio equivale ad eseguire una funzione in un oggetto remoto: nel nostro caso è il client side ad eseguire la funzione nel server side del destinatario. Detto ciò è stato deciso di creare un thread separato (il client side) che si occupasse solo dell'invio dei messaggi, in modo da non bloccare l'intera applicazione per lunghi periodi (si pensi all'attesa causata da un peer che non risponde, oppure ad un messaggio che altro non è che una chiamata ricorsiva). In tal modo, il peer che deve solamente fare un forward a $P_2$ del messaggio proveniente da un peer $P_1$, all'interno del metodo nel server side (invocato da $P_1$), creerà un messaggio analogo a quello generato da $P_1$, cambiando solamente il destinatario, ed incodandolo al proprio client side, con il risultato che il client side di $P_1$ termina immediatamente, senza aspettare che $P_2$ riceva il messaggio.

Nel seguente estratto, molto semplificato, del codice del `ClientSide` possiamo vedere quali siano le azioni base intraprese dal metodo `execute()` di un messaggio di tipo `WordMsg`.

```
registry = LocateRegistry.getRegistry(destPeer.IPaddr, destPeer.serverPortno);
stub = registry.lookup("ServerSide");
response = stub.word(id, word, time, winnerId, crashedPeer);
```

In figura 3.1.2 possiamo osservare il comportamento della parte di comunicazione, messo in relazione con tutta la parte di modellazione del gioco. Le classi dell'interfaccia mostrate nel diagramma non sono state menzionate poiché inutili ai fini del progetto di sistemi distribuiti, possiamo solamente dire che il `GamePanel` è una componente dell'interfaccia che si occupa di inserire le parole da parte dell'utente.

Il `ClientSide` ed il `ServerSide`, come anche il `Peer` vengono eseguiti in thread differenti. Detto ciò è stato molto importante gestire la concorrenza, sopratutto quando vengono modificati i campi del `Peer`. In particolare sono stati utilizzati i meccanismi di locking messi a disposizione da Java, per garantire che nessun thread potesse compiere delle operazioni contemporaneamente sul peer. Questa decisione non è limitante in termini di efficienza, poiché tali sono stati fatti in modo da non comprendere i ritardi di trasmissione della rete, ma solamente in modo da comprendere solamente le piccole porzioni che modificano o leggono i valori del peer.

Per quanto riguarda invece l'implementazione dei timeout, è stata utilizzata la classe messa a disposizione da Java, chiamata `Timer`. Questa, memorizzata all'interno del `Peer`, provvede ad eseguire determinate task dopo un certo periodo di tempo. Tali task (chiamati da Java `TimerTask`) possono comunque essere annullati, ad esempio in seguito ad una ricezione di un messaggio. Ogni task viene comunque memorizzato all'interno del `Peer` stesso, in modo che ogni metodo (del `Peer`, `ClientSide` o `ServerSide`) possa crearli, e salvandoli abbiamo la possibilità (anche da parte di altri metodi) di poterli annullare. I task utilizzati sono i seguenti:

- `helloTask`: per catturare il fatto che il messaggio *hello* non è ritornato indietro.
- `lastWordTask`: per catturare la morte dei forwarder del messaggio *word*.
- `lastElectionTask`: per catturare la morte del peer che detiene il turno attuale.
- `firstPhaseElectionTask`: per catturare la morte dei peer candidati durante l'elezione.
- `secondPhaseElectionTask`: per catturare la morte del peer eletto nella seconda fase dell'elezione.

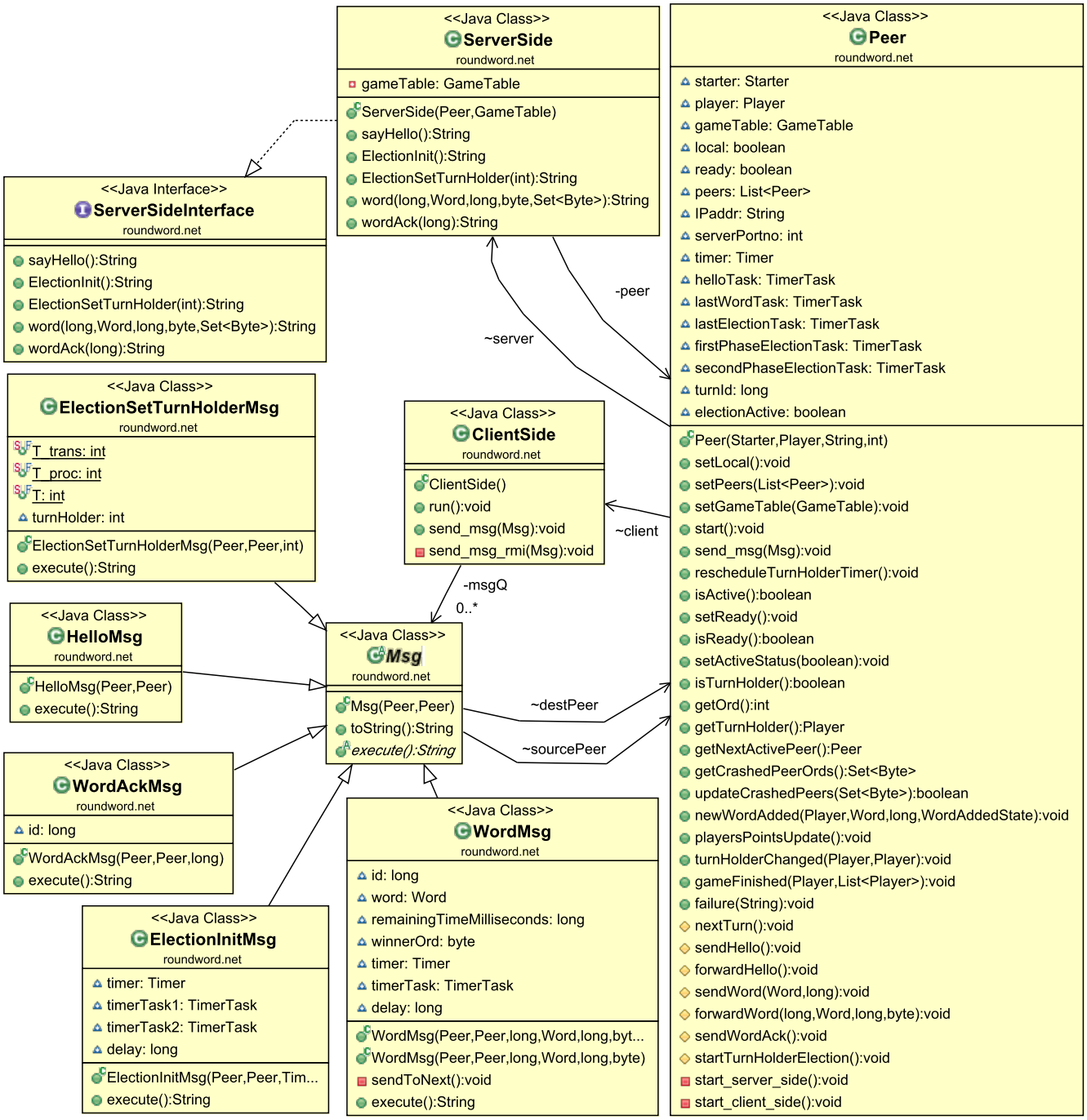


Figura 4: Il diagramma delle classi di tutta la parte di comunicazione del gioco. Da notare `ServerSideInterface` che estende la classe `Remote`, e quindi viene solamente utilizzata per Java RMI, in modo che i client side dei vari peer possano chiamare in remoto i metodi del `ServerSide`.

Ognuno di questi viene creato e distrutto in momenti differenti. Ad esempio `lastWordTask` viene creato al momento dell'invio di un messaggio di *word*, e ricreato quando invece si riceve un nuovo messaggio.

TODO: altro?

# 4 Valutazione

Il gioco che è stato implementato segue le caratteristiche del classico gioco che ha avuto tanto successo, chiamato *Ruzzle*, versione virtuale del noto gioco *Il Paroliere*. Quest'ultimo, molto semplice, si basa anch'esso sulla conoscenza approfondita del vocabolario italiano, viene giocato da più giocatori, e costituisce quindi una sfida da parte del giocatore che in caso di vittoria lo induce a credere di essere un buon conoscitore della lingua italiana, e di conseguenza aumenta lo stimolo da parte dello stesso. Allo stesso modo si è pensato quale potesse essere un gioco con le stesse caratteristiche, che potesse essere anche implementato come progetto per questo corso di Sistemi Distribuiti. Tale gioco è molto citato nei libri ed in giro per il Web, ma viene pressoché giocato senza l'ausilio di un computer, o comunque senza l'ausilio di un entità completamente automatica che controlli la validità delle parole (ad esempio si utilizzano dei forum con dei moderatori umani). Questo probabilmente perché è molto difficile riuscire a costruire un algoritmo che faccia una sillabazione perfetta, per via delle tante eccezioni che sono presenti. Per questo motivo è già stato un passo avanti riuscire ad creare qualcosa di innovativo in questo senso, riuscendo a superare tali difficoltà.

TODO: Valutazioni a livello di comunicazione?

# 5 Conclusioni

TODO: ...

Sarebbe interessante portare avanti questo progetto anche in ambiente mobile, dove sicuramente un gioco simile può avere successo. Per quanto riguarda la giocabilità ci sono diverse proposte e migliorie. Prima di tutto potrebbe essere più divertente creare dizionari divisi per sezioni, in modo da limitare l'inserimento delle parole solamente a quelle di una determinata categoria scelta all'inizio del gioco. Inoltre, sarebbe interessante, in seguito ad uno studio di usabilità decidere se prevedere casi di terminazione che potrebbero divertire di più i giocatori, come ad esempio la vittoria di un giocatore che realizza moltissimi punti con una sola parola. Poi potrebbe essere anche utile sfruttare tutta la parte già realizzata dei giocatori automatici per rimpiazzare i giocatori che hanno avuto un crash.