RoundWord

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

MATTEO BRUCATO e MIRO MANNINO
Università di Bologna

12 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 "VERIFICARE".
- Sia $w = w_1, w_2, \ldots, w_n$ la sua suddivisione in sillabe. Ad esempio, "VE", "RI", "FI", "CA", "RE".
- 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, "REGINA", "RECITA" e "RETICOLO" 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, gestione di stati condivisi. [... FARE BENE STA PARTE, ED ELENCARE BENE I REQUISITI SCRITTI NEL SITO].

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. Il peer 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*).

...

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. Questi stessi 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

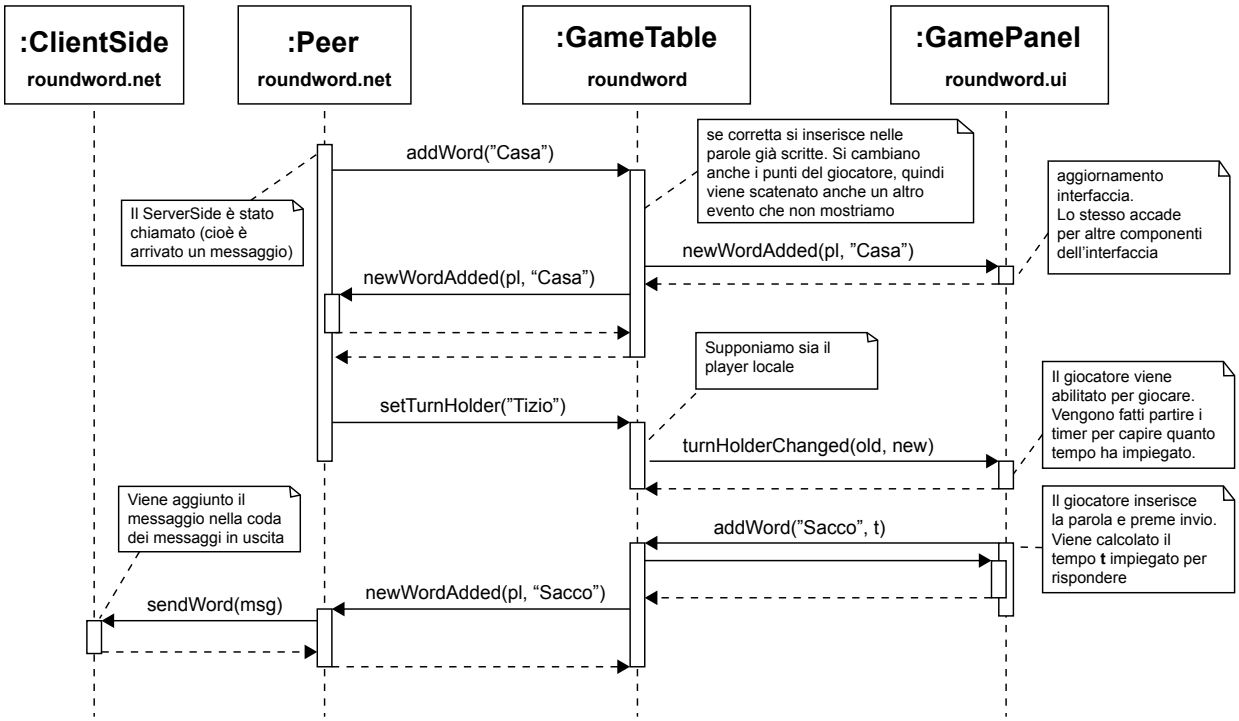

Figura 2:

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.

3.1.2 Comunicazione

Descrizione peer, clientside, serverside, ecc...

4 Valutazione

TODO: Dizionari divisi per sezioni, ad esempio tutte le parole che riguardano l’automobilismo. Si potrebbe ampliare il gioco in modo da restringere le parole che possono essere utilizzate a quelle che appartengono solamente ad una determinata categoria.

TODO: altri casi di terminazione, basati ad esempio sul fatto che un giocatore vince se doppia i punti degli altri ecc

TODO: rimpiazzi giocatori falliti

5 Conclusioni

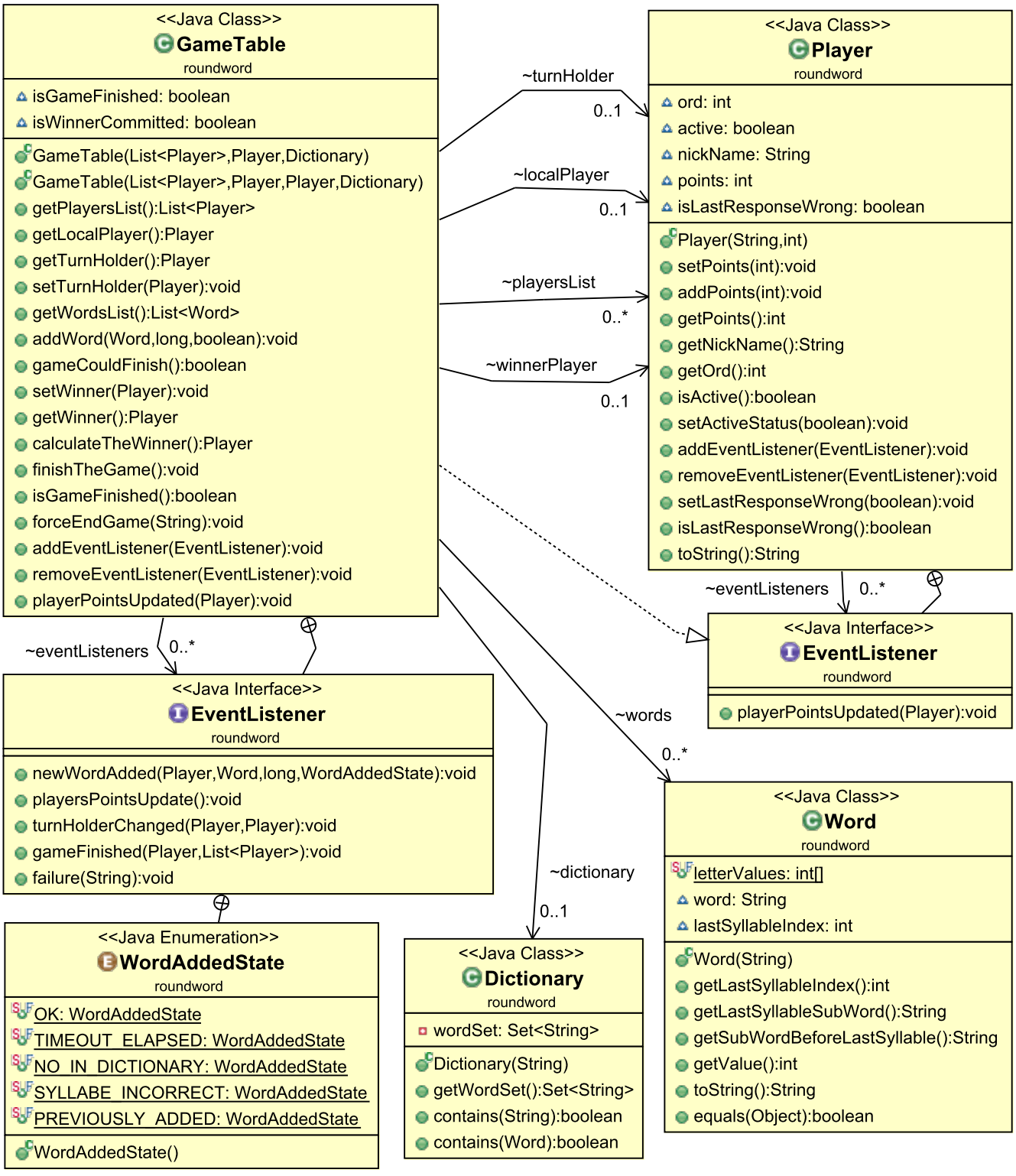
<<Java Class>>
GameTable
roundword
isGameFinished: boolean
isWinnerCommitted: boolean
GameTable(List<Player>,Player,Dictionary)
GameTable(List<Player>,Player,Player,Dictionary)
getPlayersList():List<Player>
getLocalPlayer():Player
getTurnHolder():Player
setTurnHolder(Player):void
getWordsList():List<Word>
addWord(Word,long,boolean):void
gameCouldFinish():boolean
setWinner(Player):void
getWinner():Player
calculateTheWinner():Player
finishTheGame():void
isGameFinished():boolean
forceEndGame(String):void
addEventListener(EventListener):void
removeEventListener(EventListener):void
playerPointsUpdated(Player):void
~turnHolder
0..1
~localPlayer
0..1
~playersList
0..*
~winnerPlayer
0..1
<<Java Class>>
Player
roundword
ord: int
active: boolean
nickName: String
points: int
isLastResponseWrong: boolean
Player(String,int)
setPoints(int):void
addPoints(int):void
getPoints():int
getNickName():String
getOrd():int
isActive():boolean
setActiveStatus(boolean):void
addEventListener(EventListener):void
removeEventListener(EventListener):void
setLastResponseWrong(boolean):void
isLastResponseWrong():boolean
toString():String
~eventListeners
0..*
<<Java Interface>>
EventListener
roundword
playerPointsUpdated(Player):void
~eventListeners
0..*
<<Java Interface>>
EventListener
roundword
newWordAdded(Player,Word,long,WordAddedState):void
playersPointsUpdate():void
turnHolderChanged(Player,Player):void
gameFinished(Player,List<Player>):void
failure(String):void
~words
0..*
<<Java Class>>
Word
roundword
letterValues: int[]
word: String
lastSyllableIndex: int
Word(String)
getLastSyllableIndex():int
getLastSyllableSubWord():String
getSubWordBeforeLastSyllable():String
getValue():int
toString():String
equals(Object):boolean
~dictionary
0..1
<<Java Enumeration>>
WordAddedState
roundword
OK: WordAddedState
TIMEOUT_ELAPSED: WordAddedState
NO_IN_DICTIONARY: WordAddedState
SYLLABE_INCORRECT: WordAddedState
PREVIOUSLY_ADDED: WordAddedState
WordAddedState()
<<Java Class>>
Dictionary
roundword
wordSet: Set<String>
Dictionary(String)
getWordSet():Set<String>
contains(String):boolean
contains(Word):boolean

Figura 3:

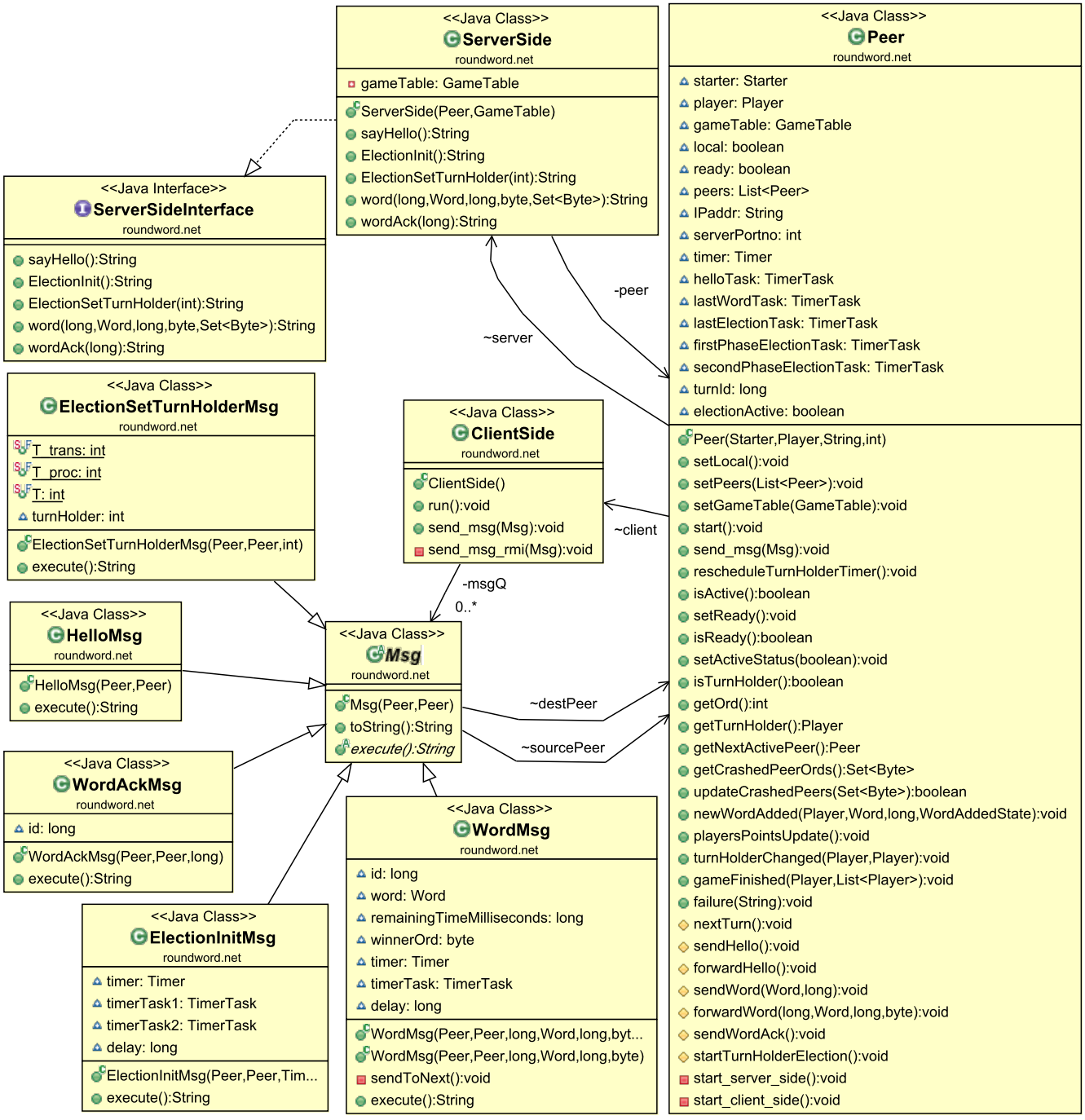
<<Java Class>>
ServerSide
roundword.net
gameTable: GameTable
ServerSide(Peer,GameTable)
sayHello():String
ElectionInit():String
ElectionSetTurnHolder(int):String
word(long,Word,long,byte,Set<Byte>):String
wordAck(long):String
<<Java Class>>
Peer
roundword.net
starter: Starter
player: Player
gameTable: GameTable
local: boolean
ready: boolean
peers: List<Peer>
IPaddr: String
serverPortno: int
timer: Timer
helloTask: TimerTask
lastWordTask: TimerTask
lastElectionTask: TimerTask
firstPhaseElectionTask: TimerTask
secondPhaseElectionTask: TimerTask
turnId: long
electionActive: boolean
Peer(Starter,Player,String,int)
setLocal():void
setPeers(List<Peer>):void
setGameTable(GameTable):void
start():void
send_msg(Msg):void
rescheduleTurnHolderTimer():void
isActive():boolean
setReady():void
isReady():boolean
setActiveStatus(boolean):void
isTurnHolder():boolean
getOrd():int
getTurnHolder():Player
getNextActivePeer():Peer
getCrashedPeerOrds():Set<Byte>
updateCrashedPeers(Set<Byte>):boolean
newWordAdded(Player,Word,long,WordAddedState):void
playersPointsUpdate():void
turnHolderChanged(Player,Player):void
gameFinished(Player,List<Player>):void
failure(String):void
nextTurn():void
sendHello():void
forwardHello():void
sendWord(Word,long):void
forwardWord(long,Word,long,byte):void
sendWordAck():void
startTurnHolderElection():void
start_server_side():void
start_client_side():void
<<Java Interface>>
ServerSideInterface
roundword.net
sayHello():String
ElectionInit():String
ElectionSetTurnHolder(int):String
word(long,Word,long,byte,Set<Byte>):String
wordAck(long):String
-peer
~server
<<Java Class>>
ElectionSetTurnHolderMsg
roundword.net
T_trans: int
T_proc: int
T: int
turnHolder: int
ElectionSetTurnHolderMsg(Peer,Peer,int)
execute():String
<<Java Class>>
ClientSide
roundword.net
ClientSide()
run():void
send_msg(Msg):void
send_msg_rmi(Msg):void
~client
-msgQ
0..*
<<Java Class>>
HelloMsg
roundword.net
HelloMsg(Peer,Peer)
execute():String
<<Java Class>>
Msg
roundword.net
Msg(Peer,Peer)
toString():String
execute():String
~destPeer
~sourcePeer
<<Java Class>>
WordAckMsg
roundword.net
id: long
WordAckMsg(Peer,Peer,long)
execute():String
<<Java Class>>
WordMsg
roundword.net
id: long
word: Word
remainingTimeMilliseconds: long
winnerOrd: byte
timer: Timer
timerTask: TimerTask
delay: long
WordMsg(Peer,Peer,long,Word,long,byt...
WordMsg(Peer,Peer,long,Word,long,byte)
sendToNext():void
execute():String
<<Java Class>>
ElectionInitMsg
roundword.net
timer: Timer
timerTask1: TimerTask
timerTask2: TimerTask
delay: long
ElectionInitMsg(Peer,Peer,Tim...
execute():String

Figura 4: