Relazione Assignment #3

Di Casamenti Gianmaria, Castellucci Lucia e Pasini Luca

Simulazione agent-based con Attori (parte 1)

Per quanto riguarda la risoluzione della prima parte dell'assigment abbiamo deciso di partire da una soluzione Master-Worker per poi convertirla ad una architettura ad attori.

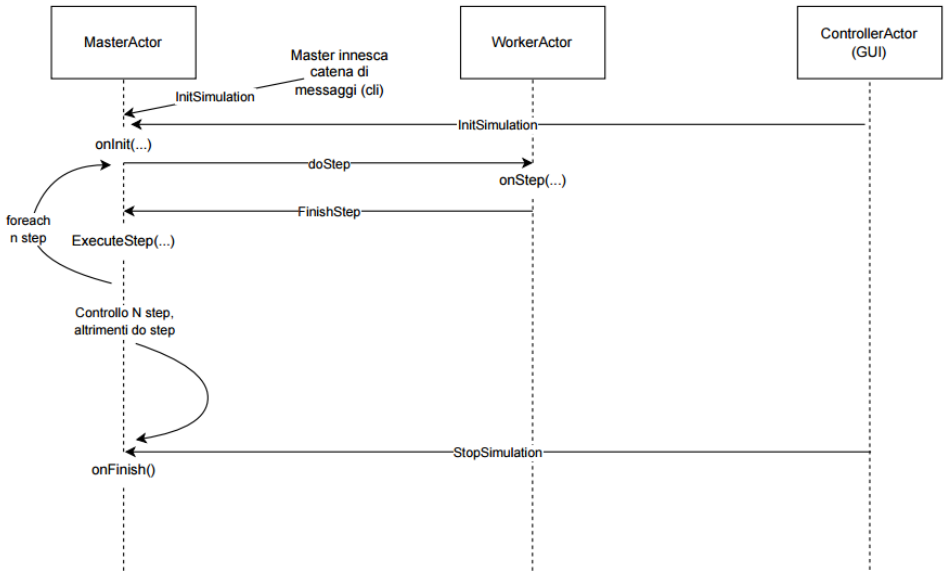
Sono stati sviluppati due attori:

- **Master Actor** è l'attore che ha il compito di gestire gli attori Worker e mandargli il messaggio **doStep**, la sua catena di messaggi viene innescata da linea di comando mandandosi il messaggio **InitSimulation**.

- **WorkerActor** è la classe degli attori che poi verranno istanziati ripetutamente per eseguire il metodo **onStep**().
  Una volta eseguito la step il worker manda un messaggio **FinishStep** al master che controllerà se tutti i worker hanno eseguito la step per andare a quella successiva.

Nella versione con la GUI abbiamo deciso di inserire un terzo attore per che rappresenta l'interfaccia grafica appunto e che avrà il compito di comunicare con il master.

- **ControllerActor** è l'attore relativo alla GUI, il suo ruolo è quello di mandare messaggi quando vengono premuti i Button di inizio e fine simulazione.



Sudoku Cooperativo (parte 2)

La seconda parte dell'assignment consiste nella realizzazione di un Sudoku Cooperativo. Nello specifico, si vuole realizzare una versione distribuita cooperativa del gioco del Sudoku ("Cooperative Sudoku"). L'applicazione deve permettere a giocatori in rete di poter creare dinamicamente nuove griglie da risolvere e/o partecipare alla risoluzione di griglie già create ancora non risolte.

Per poter immettere o cambiare un valore in una casella, un giocatore deve prima selezionare la casella. L'applicazione deve permettere a ogni giocatore di vedere – in modo consistente – sia lo stato della griglia, sia le caselle correntemente selezionate da parte degli altri giocatori. Ovvero: dati due eventi e1 ed e2 di modifica dello stato della griglia, se e1 happened before e2 per un giocatore, allora e1 happened before e2 per qualsiasi altro giocatore.

Come requisiti, si richiede di adottare una soluzione per cui:

- sia possibile partecipare dinamicamente alla risoluzione di una griglia
- qualsiasi giocatore possa entrare o uscire dinamicamente (anche a causa di crash), incluso chi ha creato la griglia.

Sudoku Cooperativo con Attori (parte 2A)

Tra le soluzioni ad attori e MOM si è decisa di adottare la prima in akka, data la maggiore confidenza dovuta anche alla risoluzione del punto precedente dell'assignment.

Per poter rispettare il requisito di decentralizzazione peer-to-peer dove ogni nodo ha le stesse capacità e responsabilità, abbiamo deciso di attuare le seguenti soluzioni:

Akka Clustering

Abbiamo utilizzato Akka Clustering perché è un toolkit e runtime per la costruzione di applicazioni concorrenti e distribuite in Scala e Java.

Akka Clustering consente la creazione di cluster di nodi che possono comunicare e lavorare insieme per fornire scalabilità, tolleranza ai guasti, e migliorare la distribuzione dei carichi di lavoro.

Tra i concetti di Akka Clustering utilizzati c'è:

- **Nodi:** Ogni nodo in un cluster Akka è una singola istanza di un'applicazione Akka, che può essere un processo JVM (Java Virtual Machine) separato.

- **Cluster:** Un gruppo di nodi che lavorano insieme, comunicando tramite messaggi. I nodi possono unirsi e lasciare il cluster dinamicamente.

- **Roles:** I nodi possono essere assegnati a uno o più "ruoli", che possono essere utilizzati per differenziare il comportamento o le responsabilità dei nodi all'interno del cluster.

All'interno della soluzione sono stati designati due attori principali:

GamesActor

GamesActor è un attore che viene istanziato solamente una volta e ha l'obbiettivo di gestire le partite, il core principale di questa gestione avviene attraverso la mappa games:

```
Map<Integer, Pair<ActorRef<PlayerActorContext>,
List<ActorRef<PlayerActorContext>>>> games = new HashMap<>();
```

Questa mappa è formata dall'id della partita come chiave e da un Pair con l'ActorRef del Leader e la lista di tutti i player che giocano a quella determinata partita, con questa mappa risulta più facile gestire le operazioni di Join, exit e di fault di un nodo nel cluster.

Da notare che all'interno di GamesActor ci sono solo riferimenti e associazioni alle partite ma non lo stato/dati della partita stessa, questo perché lo stato della partita è gestito dentro ai nodi player (ogni nodo ha il suo stato) questo per garantire una migliore soluzione peer-to-peer.

All'attore GamesActor arrivano i seguenti messaggi:

- StartNewSudoku: è il messaggio di gestione di una nuova partita, che si occupa di aggiungere una nuova associazione alla mappa.

- JoinInGrid è il messaggio di join in una partita gia esistente, aggiorna la lista di player partecipanti a quella partita nella mappa players.

- DeletePlayer: Delete player rimuove il riferimento del player nella mappa in modo da non ricevere più messaggi.

- ChangeLeader: ChangeLeader viene invocato quando a lasciare il gioco è un player leader e quindi bisognerà rieleggere un nuovo leader e aggiornare la mappa già esistente di quella partita.

- DeleteMatch: quando anche l'ultimo player abbandona la partita viene rimosso il corrispettivo record dalla mappa.

PlayerActor

PlayerActor è l'oggetto che gestisce la maggior parte della logica all'interno del sudoku, una volta creata una nuova partita il player acquisisce ruolo di leader, questo è utile perchè sarà proprio l'attore con tale ruolo a dover avvertire tutti i player nella partita che c'è stato un cambiamento dello stato nella griglia.

Ogni player si tiene in memoria lo stato della griglia perchè in caso di fault o exit da parte del nodo leader deve essere pronto a essere eletto nuovo leader.

In caso di selezione di una cella:

- Il player corrente riceve il messaggio **SelectCell** con la cella seleziona che a sua volta lo divulgherà al leader.
- Il leader una volta ricevuto il messaggio **LeaderCell**, si occuperà di avvertire tutti i giocatori all'interno della partita.
- **SelectForEveryone** è il messaggio che riceveranno tutti i player, ai quali reagiranno mostrando sul loro schermo la casella selezionata.

In caso di modifica al valore di una cella:

- Il player corrente riceve il messaggio **ChageCell** con la cella modificata che a sua volta lo divulgherà al leader.

- Il leader una volta ricevuto il messaggio **LeaderChange**, si occuperà di avvertire tutti i giocatori all'interno della partita.
- **ChangeCellOfEveryone** è il messaggio che riceveranno tutti i player.

In caso di join di un nuovo player in una partita già iniziata:

- Il caso di join di un nuovo player, il giocatore manderà un messaggio all'istanza del GamesActor che a sua volta avvertirà (**NotifyNewPlayer**) il leader di un nuovo player.
- Sarà il leader ad aggiornare il player entrante con il messaggio **SendData** a fornirgli lo stato della griglia.

In caso di exit da parte di un player:

- Questo messaggio **LeaveGame** viene inviato al player dopo aver fatto la richiesta di lasciare la partita.
- **DeletePlayer** verrà inviato a tutti i player in modo da aggiornare la loro lista di player che giocano alla griglia.
- In caso sia il leader a uscire dalla partita dovrà mandare un messaggio **ChangeLeader** sia al GamesActor che a tutti i player con il riferimento del nuovo player eletto come leader.

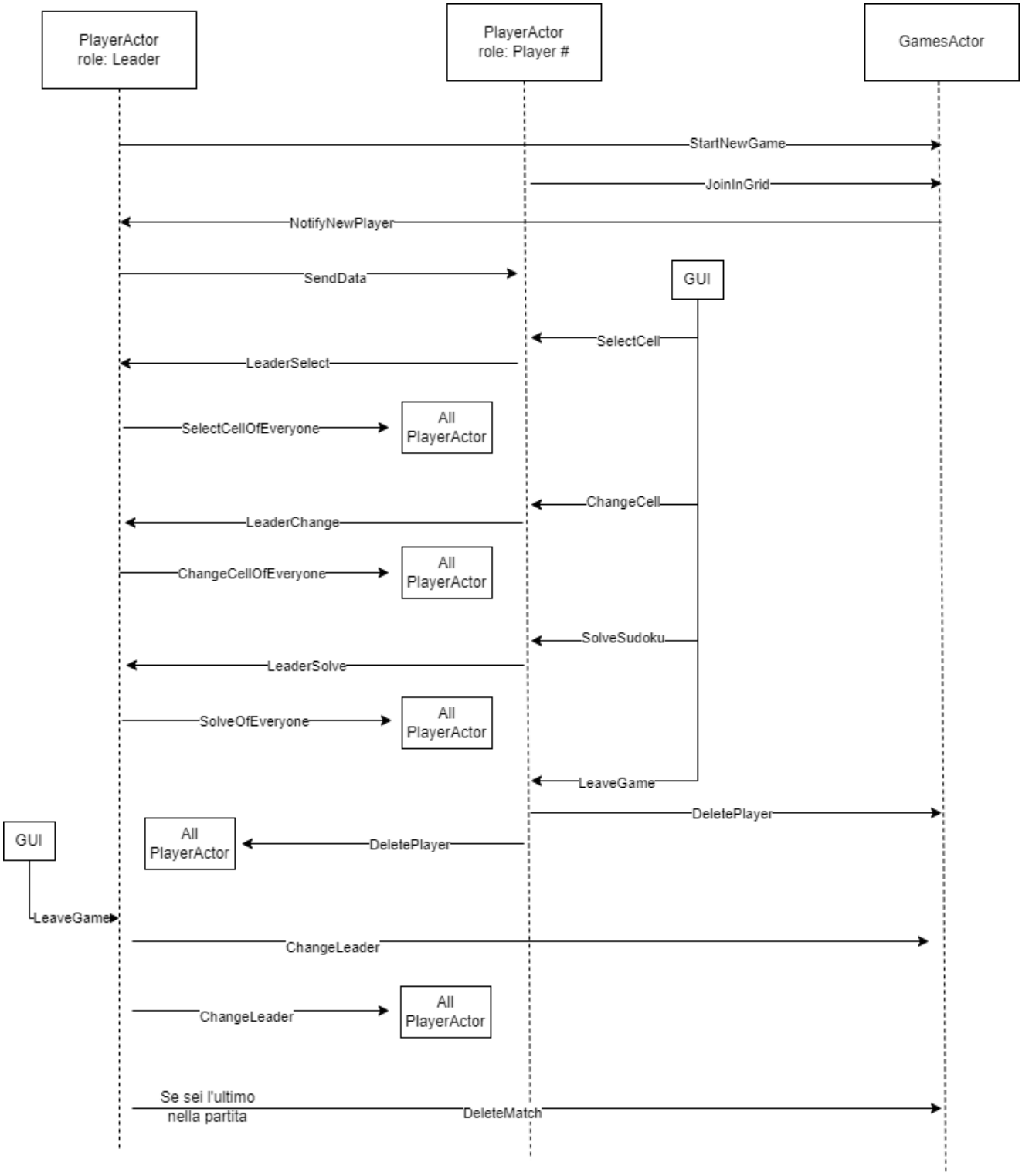
Gestione dei fallimenti dei nodi (crash)

Al fine di mostrare la fault tolerance della nostra applicazione abbiamo inserito a schermo un taso "Test crash" che una volta premuto lancerà un'eccezione al rispettivo playerActor, il quale smetterà di funzionare. Grazie al metodo" watchWith" di akka gli altri nodi si accorgeranno subito del malfunzionamento e lo tratteranno come un nodo che è uscito volontariamente dalla partita, in caso succeda al leader questo verrà sostituito così che la partita potrà continuare senza problemi.

Funzionamento

Nell'immagine sottostante è possibile analizzare il comportamento dei vari messaggi tra attori.

Leader e Player# sono istanze della stessa classe PlayerActor.



Sudoku Cooperativo con Java RMI (parte 2B)

Per realizzare una soluzione che utilizzasse Java RMI si è deciso di adottare un approccio molto minimale, sfruttando a pieno la semplicità di utilizzo del paradigma ad oggetti delegando tutta la parte di gestione del contesto distribuito al middleware.

E' stata utilizzato un approccio centralizzato, nello specifico si è fatto uso di un'architettura client-server, dove il server, rappresentato dall'oggetto SudokuServer, funge da coordinatore per tutte le griglie di gioco e per i relativi giocatori mentre il client, rappresentato dall'oggetto SudokuPlayer, ha come unico compito quello di tenere traccia della griglia di gioco del player rappresentato e aggiornarla mano a mano che la GUI viene aggiornata. La griglia di gioco è rappresentata da un oggetto (non remoto ma serializzabile) Grid, che contiene a sua volta una matrice di interi che rappresenta la griglia di partenza, un'altra matrice di interi che rappresenta la griglia corrente e una matrice di booleani che rappresenta le celle selezionate (o meno).

Entrando nello specifico...

Server

I ruoli del server sono i seguenti:

- Crea la griglia iniziale nel caso di creazione di una nuova griglia di gioco, memorizzandone l'autore (che nell'architettura rappresenta una specie di leader, dal quale i nuovi giocatori entranti ereditano la griglia di gioco)
- Crea la griglia di gioco nel caso di ingresso di un giocatore in una partita già esistente (che, come sopra anticipato, viene ereditata dal player autore della griglia)
- Notifica tutti i players dell'ingresso/uscita di altri players dalla partita, eliminando la partita nel caso in cui non vi sia più alcun player all'interno di essa
- Elegge un nuovo autore in caso di uscita del player autore della griglia
- Notifica tutti i players dei cambiamenti sulla griglia
- Notifica tutti i players nel caso di verifica del sudoku, sia nel caso in cui il sudoku sia corretto sia nel caso in cui non lo sia

Client

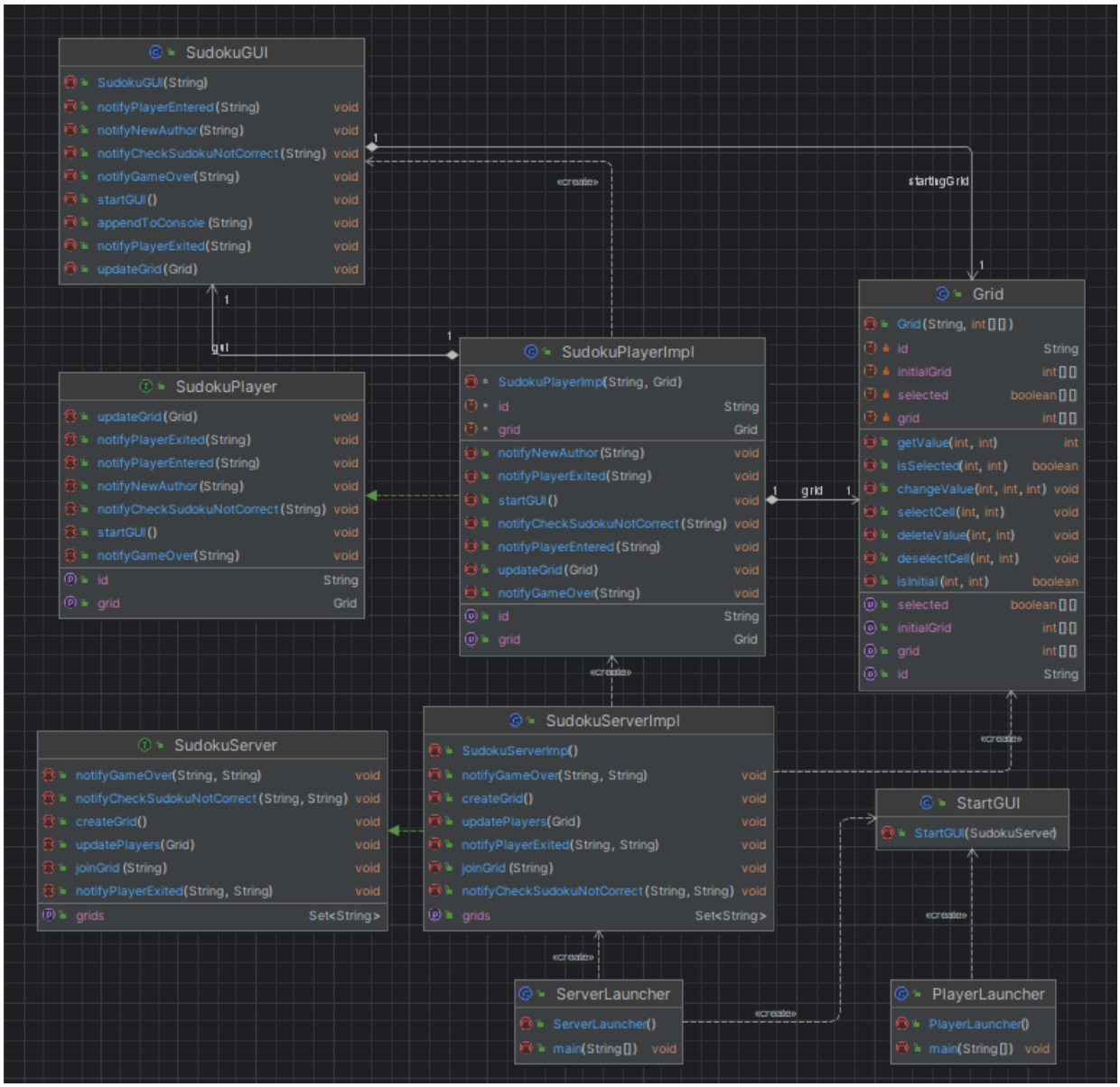
I ruoli del client sono i seguenti:

- Crea la GUI di gioco relativa al player
- Propaga i messaggi ricevuti dal server verso la GUI

la concorrenza rispetto all'accesso agli oggetti distribuiti è stata gestita mettendo "synchronized" su ogni metodo ritenuto passibile di conflitti. Tale soluzione ha peggiorato le performance del sistema ma ha garantito la consistency delle griglie, così come da specifiche di progetto.

In questo modo vi è sicuramente una lack in termini di fault tolerance, ma tale problema è dovuto all'architettura client-server che di sua natura ripone la maggior parte delle responsabilità nel server. Tuttavia, è stato scelto questo tipo di approccio per evidenziare la semplicità di utilizzo di uno strumento che rende pressoché trasparente il contesto distribuito al programmatore.

Di seguito un UML che rappresenta l'implementazione della soluzione sopra descritta:


SudokuGUI
SudokuGUI(String)
notifyPlayerEntered(String) void
notifyNewAuthor(String) void
notifyCheckSudokuNotCorrect(String) void
notifyGameOver(String) void
startGUI() void
appendToConsole(String) void
notifyPlayerExited(String) void
updateGrid(Grid) void
SudokuPlayer
updateGrid(Grid) void
notifyPlayerExited(String) void
notifyPlayerEntered(String) void
notifyNewAuthor(String) void
notifyCheckSudokuNotCorrect(String) void
startGUI() void
notifyGameOver(String) void
id String
grid Grid
SudokuPlayerImpl
SudokuPlayerImp(String, Grid)
id String
grid Grid
notifyNewAuthor(String) void
notifyPlayerExited(String) void
startGUI() void
notifyCheckSudokuNotCorrect(String) void
notifyPlayerEntered(String) void
updateGrid(Grid) void
notifyGameOver(String) void
id String
grid Grid
Grid
Grid(String, int[][])
id String
initialGrid int[][]
selected boolean[][]
grid int[][]
getValue(int, int) int
isSelected(int, int) boolean
changeValue(int, int, int) void
selectCell(int, int) void
deleteValue(int, int) void
deselectCell(int, int) void
isInitial(int, int) boolean
selected boolean[][]
initialGrid int[][]
grid int[][]
id String
SudokuServer
notifyGameOver(String, String) void
notifyCheckSudokuNotCorrect(String, String) void
createGrid() void
updatePlayers(Grid) void
joinGrid(String) void
notifyPlayerExited(String, String) void
grids Set<String>
SudokuServerImpl
SudokuServerImpl()
notifyGameOver(String, String) void
createGrid() void
updatePlayers(Grid) void
notifyPlayerExited(String, String) void
joinGrid(String) void
notifyCheckSudokuNotCorrect(String, String) void
grids Set<String>
StartGUI
StartGUI(SudokuServer)
ServerLauncher
ServerLauncher()
main(String[]) void
PlayerLauncher
PlayerLauncher()
main(String[]) void
«create»
startBigGrid
gui
grid
1

Part 3

Si vuole implementare in “Go”, il gioco "Guess the Number".

Il gioco consiste in un certo numero di N agenti *giocatori* (bot) concorrenti che devono indovinare il numero estratto a caso da un agente *oracolo*.

Il numero estratto dall'oracolo deve essere compreso fra 0 e MAX, dove MAX è un parametro in ingresso dell'applicazione.

Ad ogni turno, i giocatori devono provare ad indovinare il numero estratto dall'oracolo, proponendo un valore.
Se il valore proposto coincide con quello dell'oracolo, l'oracolo dichiara il vincitore inviando un messaggio di vittoria al player vincente e di sconfitta al resto dei giocatori, concludendo così il gioco.
Se il numero non coincide, allora l'oracolo invia una risposta al giocatore con un suggerimento, relativo al fatto che il numero da indovinare sia maggiore o minore di quello proposto. Ad ogni turno tutti i giocatori devono proporre il proprio tentativo, una e una sola volta.

L'ordine con cui i giocatori devono inviare la proposta ad ogni turno non è deterministico: ad ogni turno l'oracolo segnala a tutti i giocatori quando è possibile inviare il proprio tentativo e alla ricezione del messaggio, i giocatori devono concorrentemente inviare il proprio valore. L'oracolo considererà i valori in ordine di arrivo.

Soluzione:

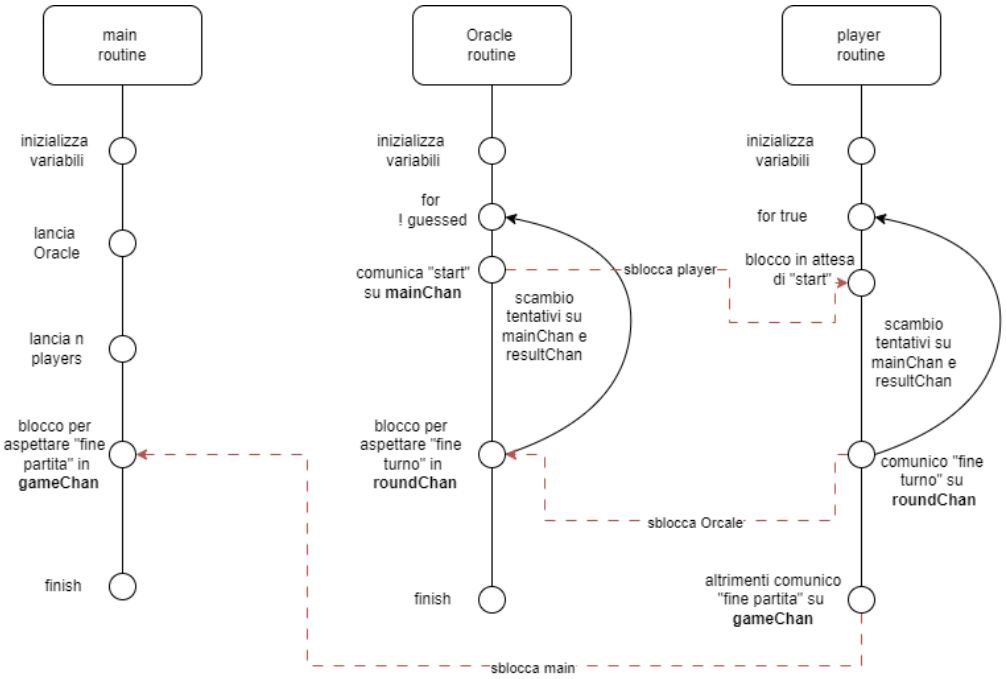
Per la gestione della concorrenza e la sincronizzazione dei turni ho deciso di utilizzare i channel di Go che permettono alle goroutine di comunicare tra loro e sincronizzare l'esecuzione passando dati.

I channel di Go sono **bloccanti** per natura, il che significa che l'invio o la ricezione su un channel ferma la goroutine finché l'operazione non può essere completata.

Per la risoluzione di questo problema abbiamo sfruttato la funzione di ricezione bloccante cioè se una goroutine prova a ricevere un dato da un channel vuoto, rimarrà bloccata finché un dato non sarà disponibile sul channel.

Come indicato nella figura sottostante, sono stati utilizzati 4 differenti canali:

- **mainChan** canale principale dove si scambiano lo stato.
- **attemptChan** canale dei risultati dove si scambiano i tentativi.
- **roundChan** canale per la sincronizzazione dei turni.
- **gameChan** canale per la sincronizzazione della partita.



Osservazioni:
Durante lo svolgimento di una prima soluzione erano stati utilizzati solamente due canali e la gestione di fine turno e fine partita avveniva grazie a due differenti sync.waitGroup che venivano sfruttati come “barriere” ma con tante istanze di players e le funzioni bloccanti dei canali in maniera randomica davano problemi.