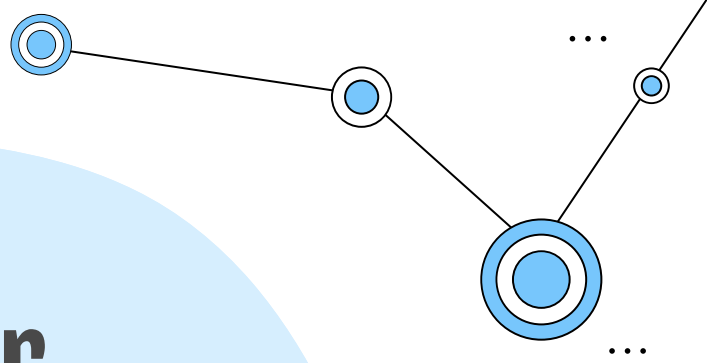
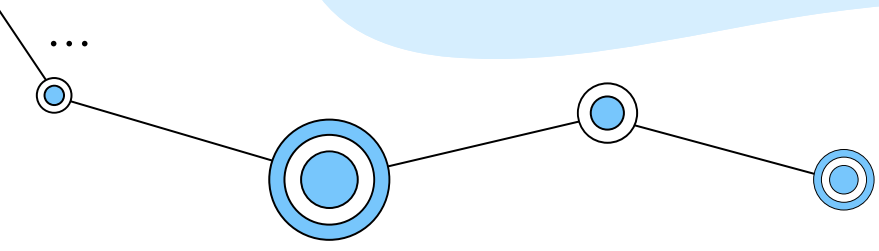


Multiplayer Minesweeper

Sistemi Distribuiti - aa. 20/21

Fabio Muratori (fabio.muratori2@studio.unibo.it)



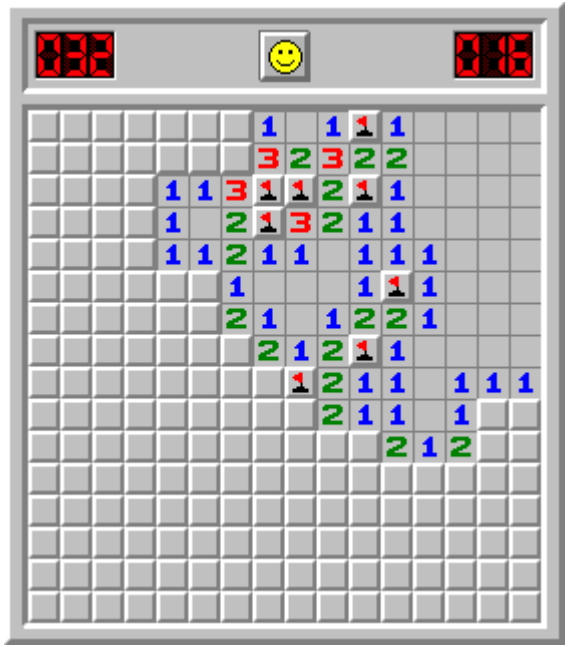
Obiettivo

Minesweeper è un classico gioco *single-player* diventato famoso negli anni 90' con Windows 3.1

Regole del gioco base

- E' data una griglia di gioco
- Il giocatore deve identificare tutte le caselle con mine
- La griglia di gioco stessa suggerisce la posizione delle mine

Si vuole realizzare una versione distribuita del gioco dove più partecipanti possono interagire in una stessa partita



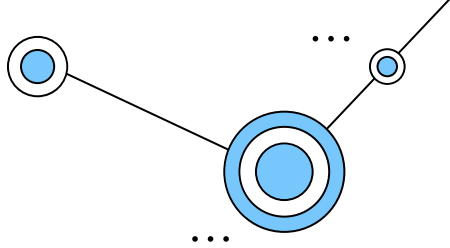
Architettura

Caratteristiche dell'architettura

- Stile architettura **Shared data space**
- Architettura client-server **centralizzata**
  - I dati condivisi sono mantenuti da server specifici
  - I client (giocatori) richiedono operazioni ai server
  - I server forniscono lo stato delle partite

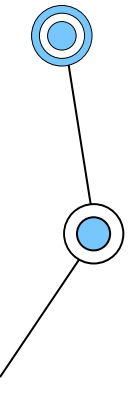
- Architettura a **microservizi** e decomposta in servizi indipendenti per
  - Deployment dell'applicazione
  - Gestione della fase pre-partita
    - Raggruppare giocatori
    - Gestire l'inizio delle partite
  - Gestione delle partite

Architettura - Componenti



L'architettura del sistema si suddivide in 3 componenti principali

- *Frontend Service* - Server per distribuire applicazioni web
- *Sessions Service* - Server per gestire sessioni di match-making
  - I giocatori possono organizzare sessioni di gioco
  - Solamente quando una sessione è piena la partita può iniziare
  - Diverse politiche di "*inizio partita*" possono essere implementate
- *Games Service* - Server per gestire partite
  - Mantiene lo stato delle partite (giocatori connessi, griglia)
  - Implementa le politiche di accesso e fornitura dello stato di gioco



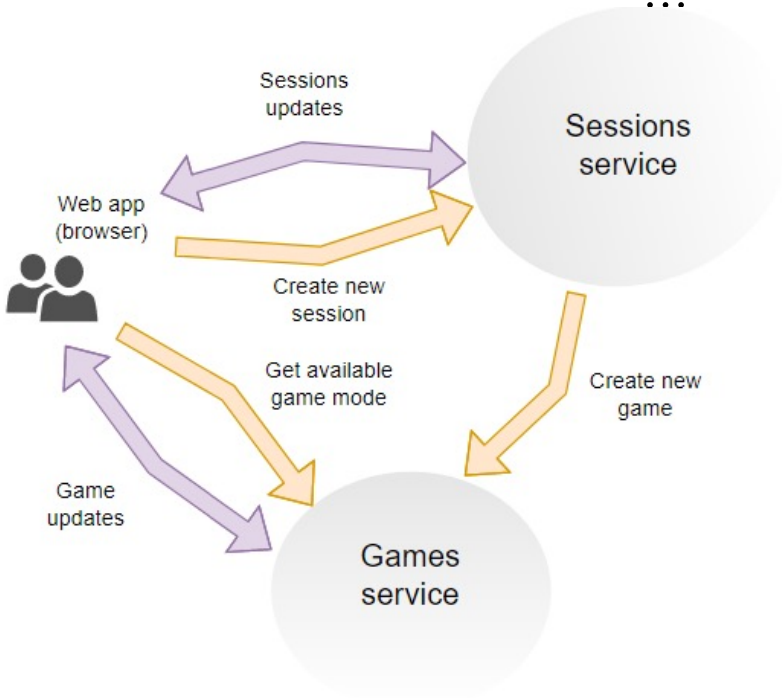
I giocatori possono interagire con il sistema tramite web browser.

Ogni servizio è indipendente dagli altri e gestisce un singolo aspetto dell'intera infrastruttura
In caso di *failure* di un servizio, gli altri componenti non sono compromessi

Architettura – Comunicazione

I componenti scambiano informazioni sfruttando 2 tipologie di canali di comunicazione:

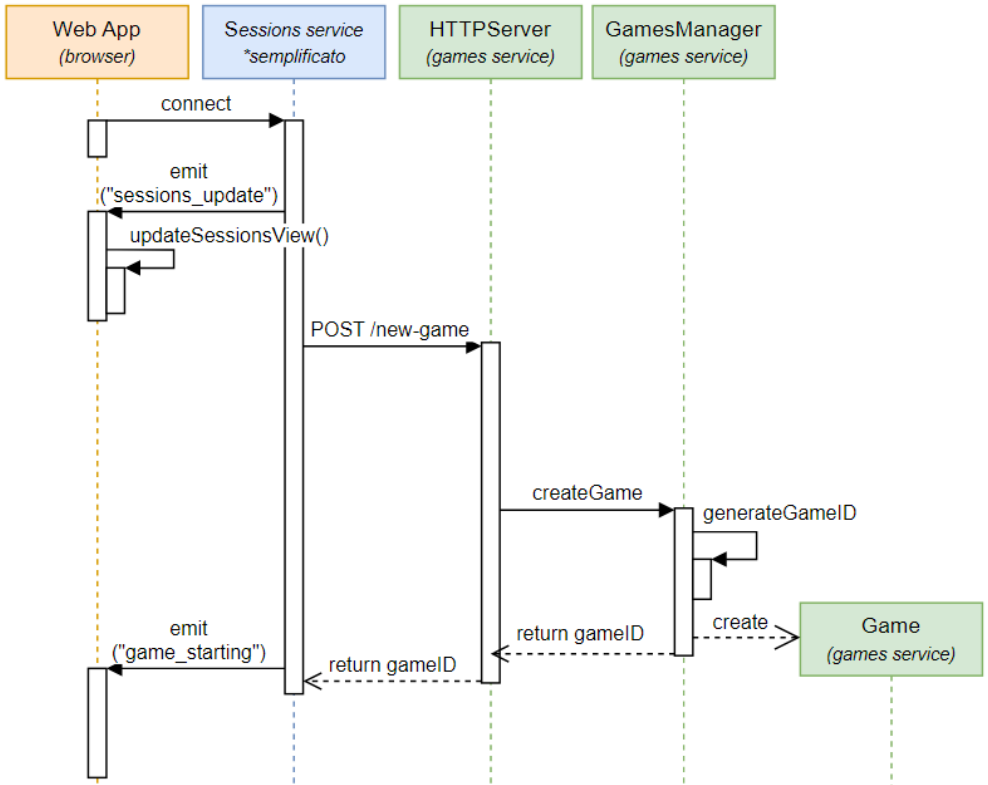
- REST APIs
  - Richiedere informazioni su sessioni attive
  - Richiedere la creazione di sessioni
  - Richiedere l'inizio di una partita
- Socket
  - Comunicare l'evoluzione delle partite dal server ai giocatori
  - Inviare richieste di azioni (visita cella, segnala possibile bandiera)



Architettura - Interazioni 1/2

INIZIALIZZAZIONE DI UNA PARTITA

1. I giocatori si connettono a *Sessions service* e ricevono aggiornamenti sullo stato della sessione
2. Non appena le condizioni lo permettono, *Sessions service* richiede la creazione di una partita (*server as a client*)
    - In seguito comunica a tutti i giocatori l'id della partita
3. I giocatori possono quindi connettersi al *games service* utilizzando questo ID



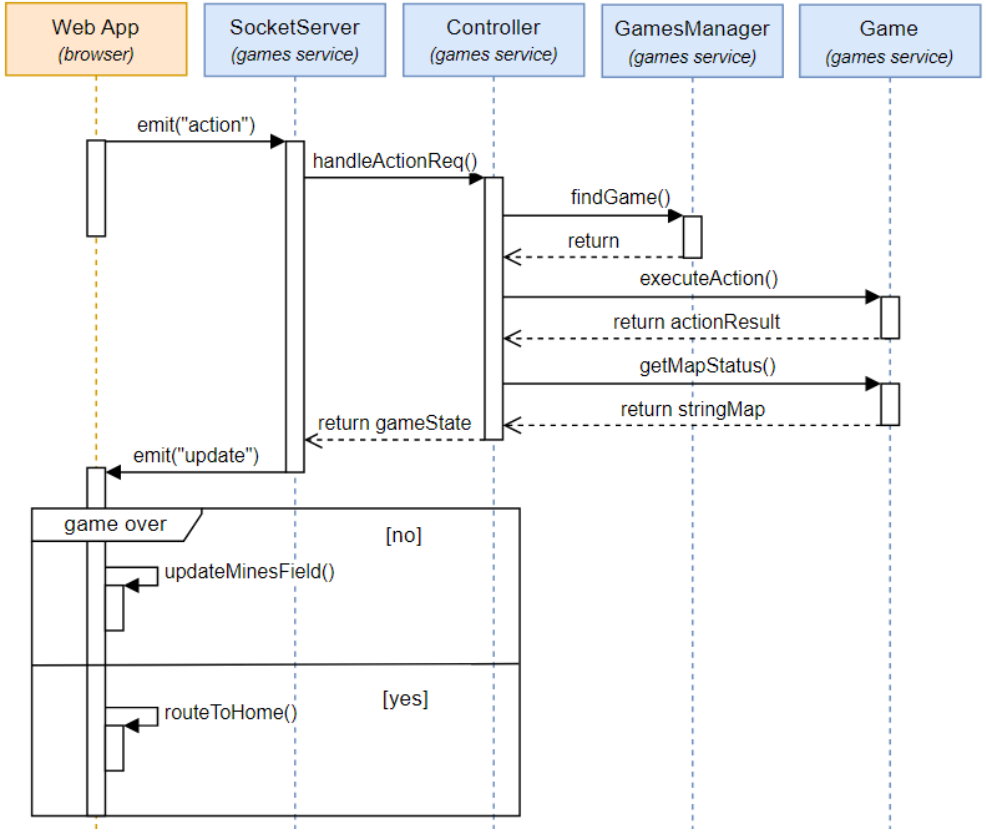
Architettura - Interazioni 2/2

AZIONI DI GIOCO

1. L'utente richiede l'esecuzione di un'azione
2. Il *games service* riceve il messaggio
3. L'operazione è eseguita
4. L'utente viene aggiornato del risultato

L'utente non attende la risposta ad una richiesta

L'utente riceve lo stato della partita solamente quando esso viene modificato



Implementazione 1/2 – Gestione dei dati

Lo stato delle partite è completamente gestito da un singolo server

- Ogni evento provoca l'invio del nuovo stato ai client
- I Client si occupano della rappresentazione grafica

Monitor Java gestire dati acceduti concorrentemente

- Le azioni richieste sono ricevute tramite socket
- I monitor assicurano un'esecuzione atomica delle azioni su una singola partita
- Più azioni possono essere eseguite parallelamente su più partite

Richieste a API Rest sono gestite da Event loop e thread worker

Similmente il Sessions Service gestisce lo stato delle sessioni pre-partita

Implementazione 2/2 – Comunicazione

Nella gestione dello stato di sessioni e partite è necessario raggruppare utenti:

- **PROBLEMA 1 - Un giocatore deve ricevere solo eventi sulla sua partita**

Inoltre, il servizio dedicato alla gestione delle sessioni deve fornire canali bidirezionali per

1. Fornire una visione sulle sessioni aperte agli utenti che non sono connessi a sessioni
2. Aggiornare lo stato della sessione a cui l'utente è connesso

- **PROBLEMA 1 - Se un utente è connesso ad una sessione, non deve ricevere informazioni sulle altre sessioni**
- **PROBLEMA 2 - Il server deve fornire 2 canali di comunicazione distinti**

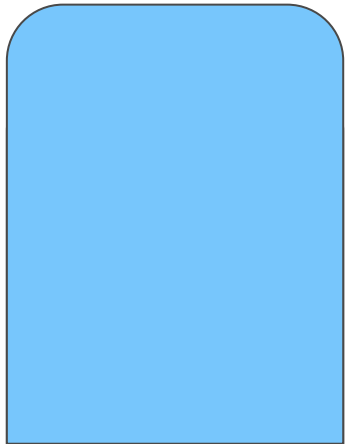
Nell'implementazione delle socket è stato usato Socket.IO

- **SOLUZIONE 1** - Implementa un concetto di *room* (stanze) per raggruppare i giocatori
- **SOLUZIONE 2** - Implementa un concetto di *namespace* per fornire servizi differenti
    - canali di comunicazione distinti su una singola socket

CAP Theorem

Network Partitioning tolerance

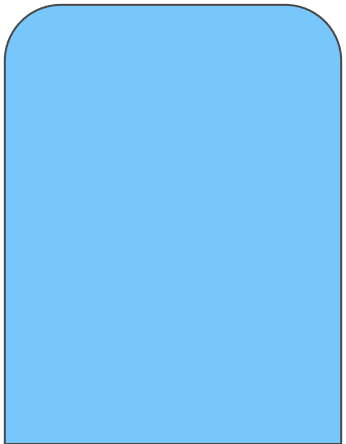
Il sistema resiste in caso di perdita di messaggi potenzialmente a discapito della consistenza dei dati



Availability

Ogni richiesta viene eseguita e se lo stato del gioco cambia, i giocatori sono aggiornati.

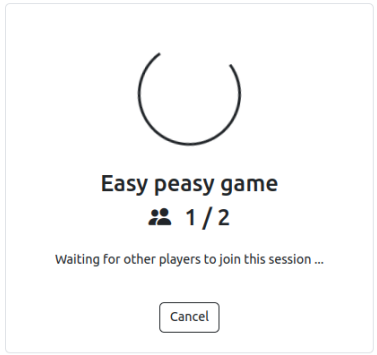
In casi di errore, il server risponde sempre

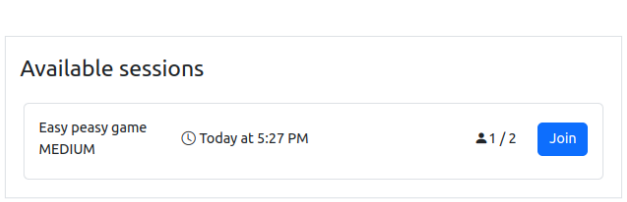


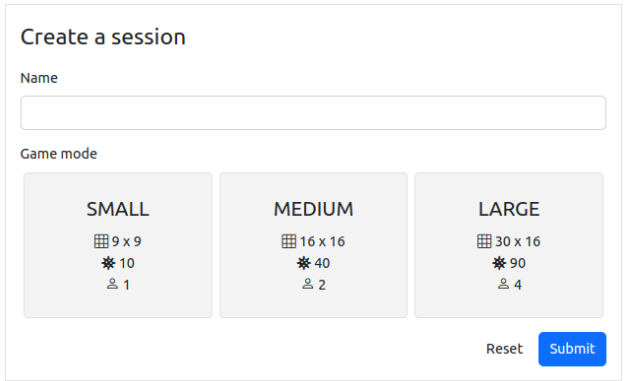
Consistency

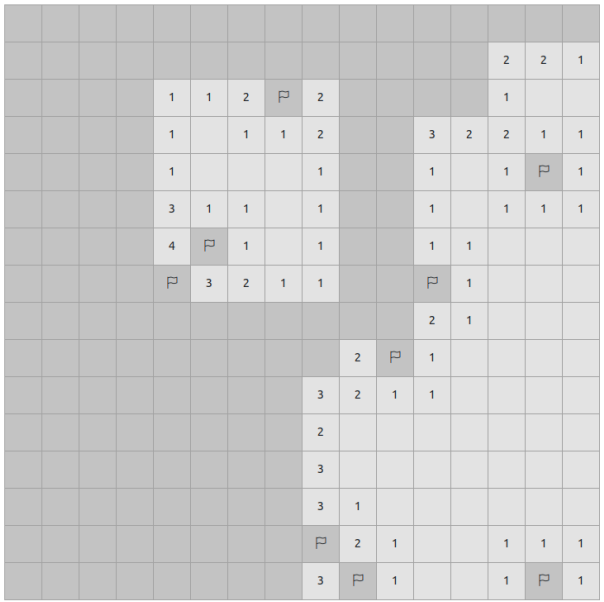
I giocatori vedono sempre uno stato corretto del gioco
ma...
non sono stati usati meccanismi per assicurare la visualizzazione dello stato più aggiornato (vector clock)

Applicazione fina









Sviluppi futuri

Espandere il sistema con più giochi

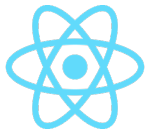
Espandere il sistema con più modalità di gioco

Visualizzare il cursore di ogni giocatore durante la partita

Integrare una chat nell'architettura a microservizi

Deployment effettivo del sistema

Tecnologie usate



GitLab

Thanks!

fabio.muratori2@studio.unibo.it