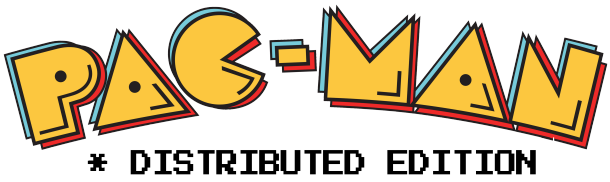
Università di Bologna

DISTRIBUTED SOFTWARE SYSTEMS



GIACOMO VALLORANI

Febbraio 2022

INDICE

1 INTRODUZIONE

Pac-Man è un famoso videogioco prodotto da Namco nel 1980 per gli arcade da sala. E' stato pubblicato per quasi la totalità dei computer, questo gli ha permesso di mantenere fino ad oggi la sua fama di classico dei videogiochi. Il gioco di basa sul far muovere una creatura sferica di colore giallo, chiamata Pac-Man, fancedole mangiare i puntini disseminati all'interno del labirinto. Nel far questo deve evitare di farsi toccare da quattro fantasmini per non morire. In alternativa, può mangiare delle "pillole" speciali che rovesciano la situazione rendendo vulnerabili i fantasmi, i quali invertono la loro marcia. In questo stato Pac-Man può fagocitare i fantasmini ed ottenere punti.

Lo scopo di questo progetto è quello di creare un clone multiplayer del gioco Pac-Man, in modo da poterlo eseguire in maniera distribuita su più dispositivi senza l'ausilio di un server per coordinare lo stato del sistema.

Dopo un'attenta ricerca è emerso che la piattaforma web ha una vasta scelta di librerie per la creazione di interfacce per giochi e permette, in maniera nativa, di gestire connessioni peer-to-peer utilizzando il protocollo *WebRTC*.
In seguito, è giunta la necessità di sincronizzare lo stato globale dei peer del sistema. Inizialmente sono stati presi in considerazioni algoritmi per gestire i problemi di mutua esclusione come: Ricart–Agrawala, Consenso Bizantino e Paxos.

- **RICART–AGRAWALA** [1], è un algoritmo che permette di ottenere la mutua esclusione in un sistema distribuito. E' un'estensione dell'algoritmo ideato da *Lamport* e utilizza un sistema di autorizzazione per acquisire la sezione critica. Il problema principale di questo approccio è la completa mancanza di fault tolleranze, che potrebbe essere risolta introducendo un timeout, ma che in un sistema realtime, come in questo caso, comprometterebbe l'esperienza d'uso rendendo il gioco ingiocabile.
- **CONSENSO BIZANTINO** o *Byzantine Fault Tolerance (BFT)* [2] è un algoritmo per stabilire il consenso (su un valore) in un sistema distribuito, anche nel caso in cui alcuni nodi falliscono o rispondono in maniera errata. Il principale difetto di questo approccio è che il numero di comunicazioni necessarie è esponenziale nel numero dei nodi. Anche questo non è adatto ad essere utilizzato su sistemi realtime.
- **PAXOS** [3] è un algoritmo di consenso che permette ai nodi di concordarsi su una proposta, senza richiedere ad un leader di coordinare la transazione. Come per il consenso bizantino la

disponibilità dei dati è bassa e porta a scartare anche questo approccio.

Nella creazione del gioco un requisito fondamentale è la disponibilità dei dati e la tolleranza la partizionamento. Nel primo caso, poiché i giocatori devono avere uno stato globale sempre disponibile, essendo un gioco altamente responsivo, ogni frame deve essere calcolato nel minor tempo possibile. In media deve essere eseguito a 60 frame al secondo, che equivalgono ad avere a disposizione 17 millisecondi per computare lo stato e renderizzare la scena. Inoltre, il sistema deve essere tollerante al partizionamento in quanto il gioco deve continuare anche se alcuni nodi si disconnettono o hanno una connessione instabile.

TEOREMA CAP: è impossibile che un servizio distribuito sia coerente, disponibile e tollerante al partizionamento nello stesso istante di tempo [4].

Come definito dal teorema CAP, l'unica proprietà che è impossibile garantire in questo caso è la coerenza. Di conseguenza, la scelta è ricaduta sull'utilizzare tecniche di *Eventual Consistency* per gestire lo stato del sistema.

EVENTUAL CONSISTENCY (EC): se non vengono apportati nuovi aggiornamenti ad un certo dato, alla fine tutti gli accessi a quel dato restituiranno l'ultimo valore aggiornato [5].

Usando questo approccio si viene a creare il problema di come gestire i conflitti generati dagli aggiornamenti simultanei dello stesso dato dai diversi peer. Esistono diverse tecniche ma la maggior parte sono ad-hoc e soggette ad errori, la letteratura offre poche indicazioni su come progettare un corretto sistema eventualmente coerente.

Marc Shapiro et al. [6] dimostrano come è possibile aggiornare uno stato condiviso in maniera deterministica senza avere conflitti introducendo il concetto di *Strong Eventual Consistency*.

STRONG EVENTUAL CONSISTENCY (SEC): tutti i nodi che hanno ricevuto lo stesso insieme di aggiornamenti (non ordinati) in un futuro raggiungeranno lo stesso stato [6].

La struttura dati che permette di implementare SEC è chiamata *Conflict-free Replicated Data Type (CRDT)*. Il modello SEC è valido e dimostrabile teoricamente [6]. Nei CRDT gli aggiornamenti avvengono in maniera asincrona e rimangono sempre disponibili nonostante la latenza e gli errori della rete.

2 CONFLICT-FREE REPLICATED DATA TYPE

Un *Conflict-free Replicated Data Type (CRT)* è un tipo di dato astratto, con un'interfaccia ben definita, progettato per essere replicato su più nodi e che presenta le seguenti proprietà:

(i) qualsiasi replica può essere modificata senza coordinarsi con altre repliche;
(ii) quando due repliche qualsiasi hanno ricevuto lo stesso insieme di aggiornamenti, raggiungono lo stesso stato, deterministicamente, adottando regole matematicamente valide per garantire la convergenza degli stati [7].

Esistono due tipologie di CRT, quelli che si sincronizzano sullo stato e quelli che si sincronizzano sulle operazioni.

2.1 Sincronizzazione basata sullo stato

Nella sincronizzazione basata sullo stato, le repliche si sincronizzano stabilendo connessioni bidirezionali (o unidirezionali), in cui entrambe le repliche (o una) inviano il proprio stato reciprocamente. Quando una replica riceve lo stato di un nodo, unisce lo stato ricevuto con il suo stato locale. Questi tipi di CRDT definiscono una funzione di unione per integrare lo stato della replica remota. È stato dimostrato [6] che tutte le repliche di un CRDT convergono se:

(i) i possibili stati sono parzialmente ordinati secondo l'operatore $\leqslant$ e formano un semireticolo;
(ii) un aggiornamento modifica lo stato s di una replica producendo un nuovo stato che è maggiore o uguale allo stato originario, cioè per ogni aggiornamento u si ha $s \leqslant u(s)$;
(iii) la funzione $merge$ produce l'unione dei due stati, per esempio da $merge(s1, s2)$ si ha $s1 \cup s2$.

Infine il grafo delle connessioni deve essere connesso per garantire che tutti gli aggiornamenti raggiungano tutte le repliche.

Di seguito una possibile implementazione di un contatore distribuito sincronizzando lo stato.

```
P = [0, 0, ...] // Payload
N = [0, 0, ...]
value() = ∑_{i=1}P[i] - ∑_{i=1}N[i]
increment() = P[myId]++
decrement() = N[myId]++
merge(s, s') = s ∪ s'
             = ([..., max(s.P[i], s'.P[i]), ...],
                [..., max(s.N[i], s'.N[i]), ...])
```

2.2 Sincronizzazione basata sull'operazione

Nella sincronizzazione basata sulle operazioni, le repliche convergono scambiandosi gli aggiornamenti reciprocamente. Quando un aggiornamento viene ricevuto, viene applicato allo stato della replica locale. I CRDT progettati per sincronizzarsi sulle operazioni devono definire, per ogni aggiornamento, una funzione generatore e una funzione che applica gli effetti dell'aggiornamento sullo stato.

La replica che invia l'aggiornamento genera, tramite la funzione generatore, la funzione che codifica gli effetti dell'operazione sullo stato. La replica ricevente eseguirà la funzione. È stato dimostrato [6] che se le operazioni vengono consegnate in ordine casuale, le repliche convergeranno nello stesso stato se le operazioni commutano. Se le operazioni possono essere eseguite più di una volta, allora devono essere idem-potenti.

2.3 Applicazioni

Di seguito alcuni impieghi nell'ambito enterprise.

- Il database Riak è stato uno dei primi ad aggiungere il supporto ai CRDT nel 2013 [8].
- Il database Apollo usato internamente da Facebook supporta i CRDT [9].
- Il sistema di storage Dynomite utilizzato da Netflix utilizza i CRDT internamente [10].
- Il database CosmosDB di Microsoft Azure può essere configurato per risolvere i conflitti utilizzando i CRDT [11].
- Redis Enterprise utilizza i CRDT per abilitare la replicazione multi-master tra datacenter geograficamente distribuiti [12].
- Elmerfs è un file system distribuito, con interfaccia FUSE, scritto in Rust. Elmerfs utilizza i CRDT per evitare conflitti strutturali [13].

3 LOGICA DEL GIOCO

Sono state apportate alcune semplificazioni alla logica originaria per poter adattare meglio il gioco ad un contesto distribuito. Di seguito vengono spiegate le regole del gioco.

- Il numero di punti disseminati nel labirinto è definito a priori e non è dipendente dal numero di giocatori.
- I fantasmini sono 4 indipendentemente dal numero di Pac-Man.
- Il numero massimo di giocatori, quindi di Pac-Man, è pari a 8.
- Ogni Pac-Man parte da una posizione differente nel labirinto.

- Nella fase iniziale vengono assegnati i fantasmini ai Pac-Man che sono connessi. Un Pac-Man può avere associati più fantasmini in base alla disponibilità. Se non ci sono abbastanza fantasmini alcuni Pac-Man rimarrano senza.
- Il fantasmino calcola la strada migliore per arrivare al suo Pac-Man target.
- Mangiando i power-pellet i ruoli si invertono e i fantasmini scappano dai Pac-Man e quest'ultimi possono fagocitarli guadagnando punti.
- Lo scopo del gioco è quello di mangiare più punti possibili e allo stesso tempo di non essere mangiati dai fantasmini.
- Se un giocatore chiude la pagina o cambia tab entra in uno stato offline e viene escluso temporanemante dal gioco.

4 SOFTWARE STACK

4.1 Svelte

Svelte è un compilatore JavaScript front-end gratuito e open-source. Le applicazioni scritte in Svelte non includono una libreria per la manipolazione del DOM, ma generano a tempo di compilazione il codice necessario a ciascun componente. Questo approccio permette di ridurre la dimensione degli script e offre prestazioni migliori sia all'avvio che a run-time [14]. L'intero progetto è stato strutturato sotto forma di classi e componenti Svelte, così facendo l'intera codebase è modulare e facilmente manutenibile.

4.2 TypeScript

Come linguaggio di programmazione è stato scelto TypeScript, un Super-set di JavaScript che basa le sue caratteristiche su ECMAScript 6. Il linguaggio estende la sintassi di JavaScript in modo che qualunque programma scritto in JavaScript sia anche in grado di funzionare con TypeScript senza nessuna modifica. È stato progettato per lo sviluppo di grandi applicazioni e deve essere compilato in JavaScript per poter essere interpretato da qualunque web browser [15].

La caratteristica principale sono i tipi che forniscono un modo per descrivere la forma di un oggetto, inoltre, consentono di produrre una migliore documentazione e permettono di verificare il corretto funzionamento del codice prima di essere eseguito.

4.3 Three.js

Three.js è una libreria JavaScript cross-browser che fornisce una API specializzata per creare e visualizzare grafica 3D animata in un browser Web utilizzando WebGL [16]. Questa libreria è stata utilizzata per creare l'interfaccia principale del gioco con i principali elementi grafici, quali il labirinto, i fantasmini e i Pac-Man. Il tutto è stato reso molto più accattivante progettando l'ambiente di gioco in maniera tridimensionale.

4.4 Yjs

Yjs è un CRDT ad alte prestazioni per la creazione di applicazioni collaborative che si sincronizzano automaticamente. Espone il suo modello CRDT interno come tipi di dati condivisi che possono essere manipolati contemporaneamente. I tipi condivisi sono simili ai tipi di dati comuni come Map e Array, essi possono essere manipolati in maniera asincrona (anche offline) e possono essere uniti con altri automaticamente senza conflitti [17].

Yjs non fa discriminazioni sulla tecnologia di rete che si sta utilizzando. Finché alla fine arriveranno tutte le modifiche, i documenti verranno sincronizzati correttamente. L'ordine in cui vengono applicati gli aggiornamenti dei documenti non ha alcuna importanza. Yjs implementa una versione modificata dell'algoritmo descritto in [18], ulteriori informazioni sono disponibili nella documentazione [19].

4.4.1 Y-webrtc provider

Yjs ha un approccio modulare che permette di essere ampliato, in base al caso d'uso, utilizzando dei plugin. Uno di questi è *y-webrtc* [20]. Il suo compito è quello di gestire la comunicazione tra i diversi peer connessi attraverso il protocollo WebRTC, rendendo trasparente al programmatore l'intera procedura di aggiornamento dei CRDT.

5 ARCHITETTURA

5.1 Protocollo WebRTC

WebRTC non può creare connessioni senza una sorta di server nel mezzo, che viene detto server di segnalazione.

La configurazione di un endpoint di una connessione WebRTC è denominata *session description*. La descrizione include informazioni sul tipo di media inviato, il suo formato, il protocollo di trasferimento

utilizzato, l'indirizzo IP e la porta dell'endpoint. Queste informazioni vengono scambiate tramite *SDP* [21].

SESSION DESCRIPTION PROTOCOL (SDP): è un formato per descrivere sessioni di comunicazione multimediale al fine di annunciarsi ad altri host ed iniziare la sessione [22].

Oltre a scambiare informazioni sui media, i peer devono scambiare informazioni sulla connessione di rete. Questa tecnica è nota come *Interactive Connectivity Establishment (ICE)* e descrive in dettaglio tutti i metodi disponibili al peer per comunicare (direttamente o tramite un server TURN).

INTERACTIVE CONNECTIVITY ESTABLISHMENT (ICE): è una tecnica utilizzata per individuare la modalità con cui due computer possono comunicare tra di loro il più direttamente possibile nelle reti peer-to-peer [23].

TRAVERSAL USING RELAY NAT (TURN): se un host si trova dietro un NAT, in determinate situazioni può essere impossibile per quell'host comunicare direttamente con altri host (peer). In queste situazioni, è necessario che l'host utilizzi i servizi di un nodo intermedio che funge da relay di comunicazione. Il protocollo TURN consente all'host di controllare il funzionamento del relay e di scambiare pacchetti con i suoi peer usando il relay [24].

SIMPLE TRAVERSAL OF UDP THROUGH NATS (STUN): è un protocollo che consente alle applicazioni di scoprire la presenza e i tipi di NAT e firewall tra loro e la rete Internet pubblica. Fornisce inoltre la possibilità per le applicazioni di determinare gli indirizzi IP pubblici assegnati loro dal NAT [25].

Inoltre, prima di iniziare la sessione, l'host invierà una richiesta ad un server STUN che risponderà con l'indirizzo pubblico del client. Sulla base della sua accessibilità o non accessibilità dietro il NAT del router, deciderà se interpellare il server TURN. In questo progetto non è stato inserito nessun server TURN, pertanto i peer devono effettuare connessioni direttamente senza intermediari.

In figura 1 viene mostrato il diagramma di comunicazione di due peer tramite WebRTC. Mentre in figura 2 viene mostrato il diagramma di sequenza di una connessione WebRTC tra due peer.

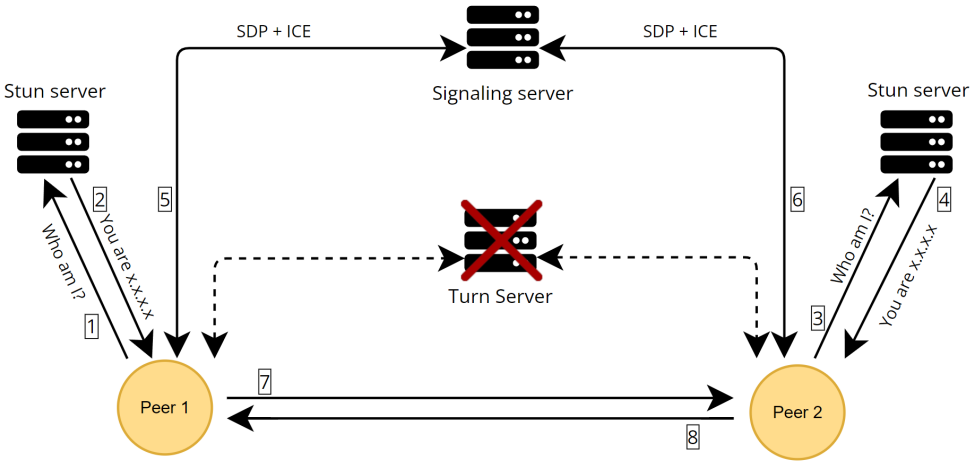


Figura 1: diagramma di comunicazione di due peer tramite WebRTC

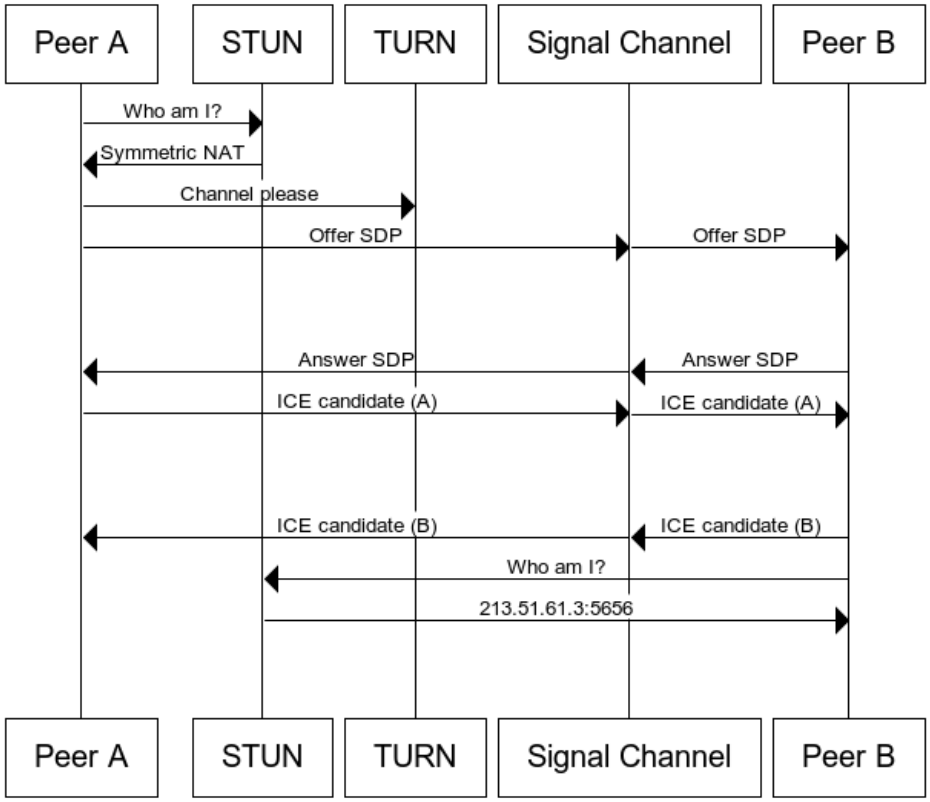


Figura 2: diagramma di sequenza di una connessione WebRTC [21].

5.2 Topologia della rete

I peer sono connessi tra loro formando una topologia full-mesh, cioè ogni peer ha una connessione diretta con tutti gli altri. Questo vincolo è imposto dalla libreria Yjs e in particolare dal plug-in y-webrtc.

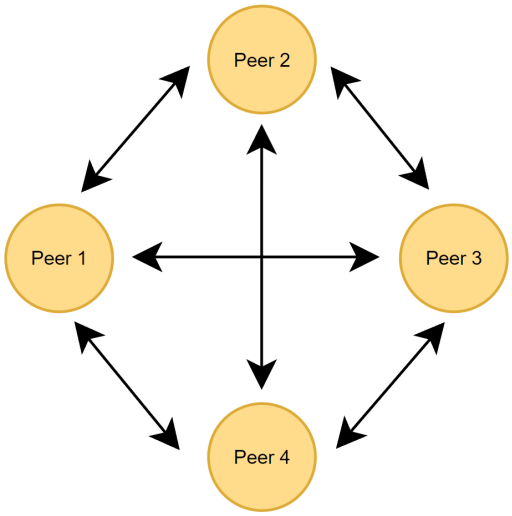


Figura 3: topologia della rete.

Il vantaggio principale di questa topologia è che le comunicazioni tra i nodi avvengono rapidamente, non ci sono nodi intermedi da contattare.

5.3 Diagramma delle classi

La figura 5 mostra il diagramma delle principali classi che compongono il gioco.

5.4 Diagramma dello stato

Inizialmente, tutti i peer che aprono la pagina web impostando lo stesso id della stanza stabiliscono una connessione diretta. A quel punto, dopo aver inserito il nickname e premuto il bottone START il gioco inizia. Si resta nello stato *Game started* fino a quando non vengono esaurite tutte le vite, la finestra perde il focus i si perde la connessione. Quando tutti i giocatori arrivano allo stato *Game over*, chiudono tutte le connessioni che hanno con gli altri e il gioco termina.

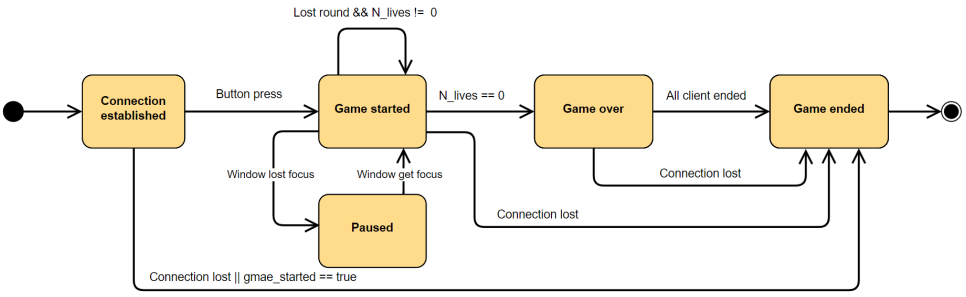


Figura 4: diagramma dello stato.

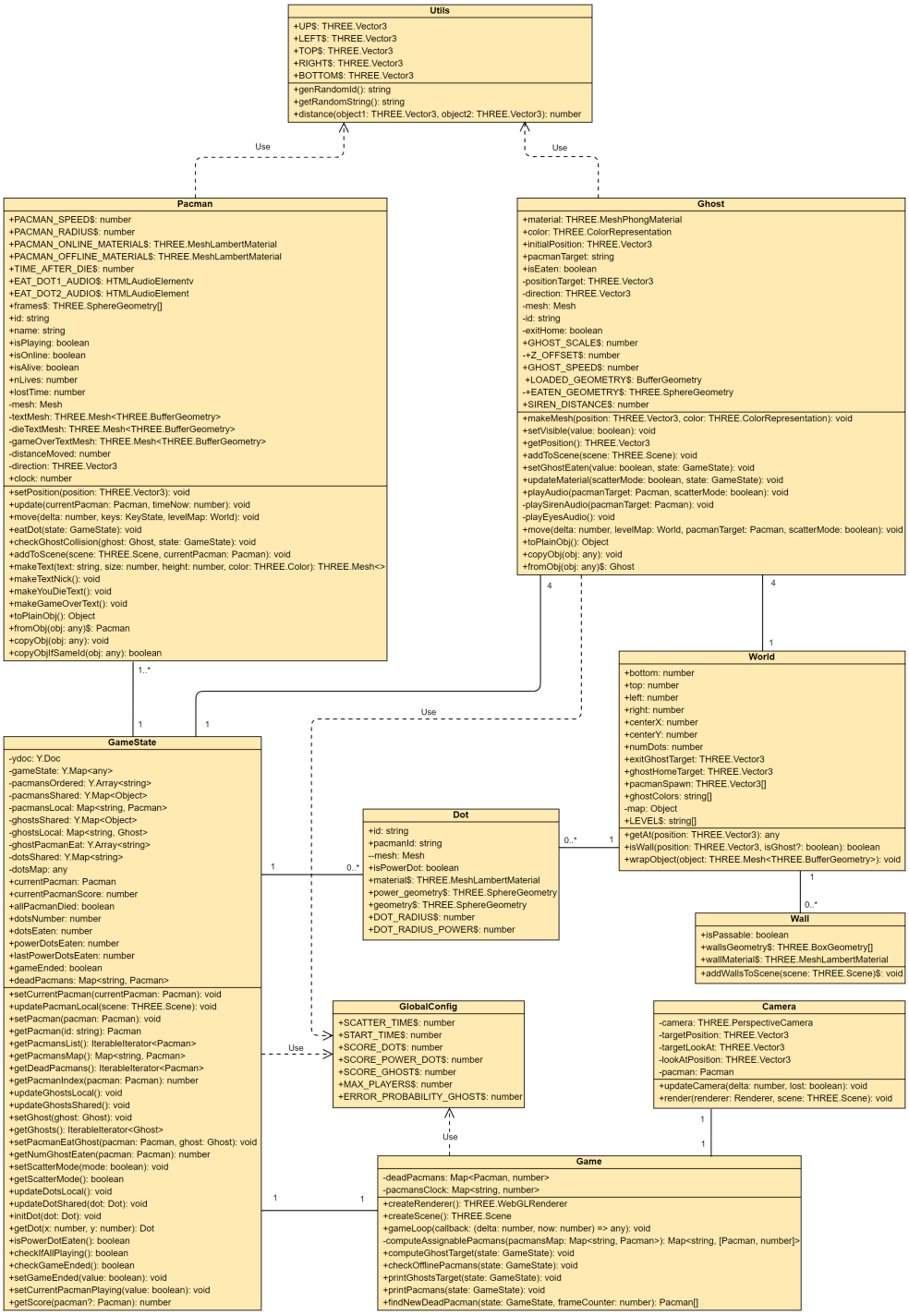


Figura 5: diagramma delle classi.

5.5 Struttura dei CRDT

Una parte molto importante della progettazione riguarda il come strutturare i CRDT, in quanto durante il gioco lo stato deve rimanere il più coerente possibile e soprattutto alla fine tutti i giocatori devono visualizzare lo stesso numero di punti senza discrepanze. Tutti i CRDT e i metodi che li gestiscono sono definiti nella classe `GameState`.

`pacmansOrdered: Y.Array<string> // Array<pac_id>`

Array che contiene gli identificatori dei Pac-Man presenti nel sistema. Viene utilizzato per mantenere un ordinamento dei giocatori, questo tornerà utile quando sarà assegnata loro una posizione nel labirinto.

`pacmansShared: Y.Map<Object> // Map<pac_id, pac_object>`

Mappa che associa gli identificatori dei Pac-Man al loro oggetto nel gioco. Durante il gioco ogni Pac-Man modifica gli attributi solo del proprio oggetto.

`ghostsShared: Y.Map<Object> // Map<ghost_id, ghost_object>`

Mappa che associa gli identificatori dei fantasmini al loro oggetto nel gioco. Durante il gioco ogni oggetto fantasmino viene aggiornato dal rispettivo peer che comanda il Pac-Man che gli è stato associato.

`dotsShared: Y.Map<string> // Map<dot_id, pac_id>`

Mappa che associa l'identificatore del punto all'identificatore del Pac-Man. Un punto è stato mangiato dal Pac-Man se compare come chiave nella mappa. In questo modo lo score di ogni giocatore rimane coerente per tutta la durata del gioco.

6 IMPLEMENTAZIONE

L'entry-point del gioco è la componente `App.svelte`, nella quale viene creato il `WebrtcProvider` impostando: il numero massimo di connessioni, il nome della stanza, una password per criptare la connessione, il server STUN e quello di signaling (di default sono impostati dei server di pubblico dominio). `WebrtcProvider` ha il compito di gestire le connessioni e la sincronizzazione dei CRDT.

Fatto ciò, viene visualizzato il menu che permette di inserire il proprio nickname.

Figura 6: schermata inserimento nickname.

Una volta premuto su *GO*, viene creato il `game loop` e viene generato il proprio Pac-Man che verrà poi inserito nello stato globale (`pacmansOrdered` e `pacmansShared`). Inoltre, vengono aggiunti alla scena i fantasmini, i muri che formano il labirinto e i relativi punti. Man mano che i giocatori si collegano i CRDT vengono popolati e sincronizzati in maniera automatica.

Figura 7: lista dei giocatori connessi.

A questo punto, dopo aver premuto *START GAME*, viene aggiornato l'attributo `game_started` nel CRDT globale, e viene chiusa la connessione con il signling server.

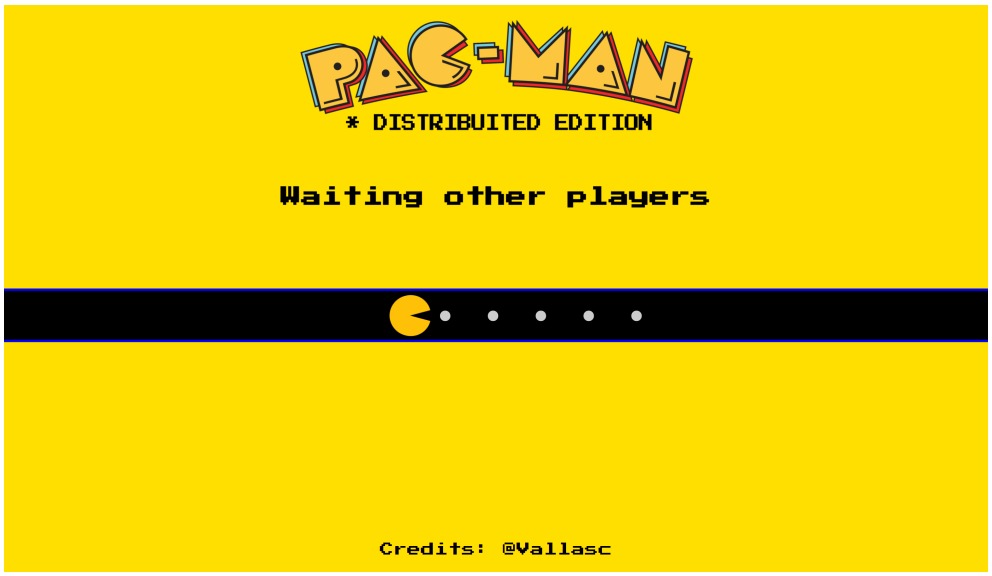


Figura 8: schermata di attesa degli altri giocatori.

Dopo aver ricevuto l'aggiornamento e dopo aver controllato l'attributo `game_started`, gli altri peer stabiliscono che la partita è iniziata. Se in un questa fase un giocatore provasse a collegarsi verrebbe automaticamente disconnesso. Durante la fase di caricamento, ogni Pac-Man calcola gli avversari che devono essere assegnati ai fantasmi. La regola generale è che ogni Pac-Man può avere associati al massimo $\lfloor \frac{\#ghosts}{\#pacmans} \rfloor$ fantasmini. Finché non viene rispettato questo criterio vengono ricalcolati, ogni secondo, e vengono aggiornati nello stato condiviso.

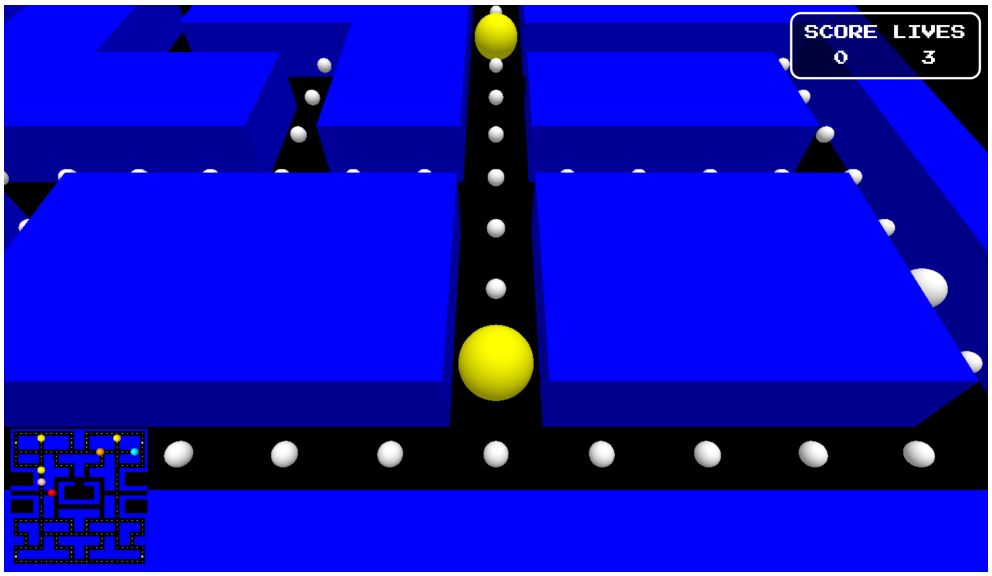


Figura 9: posizione iniziale Pac-Man .

Figura 10: schermata GAME OVER.

Durante la partita vengono catturati gli eventi di pressione dei tasti (W, A, S,e D) e si agisce di conseguenza sulla posizione e sulla direzione del Pac-Man. Ad ogni ciclo del game loop vengono ricalcolate:

- posizioni dei Pac-Man nemici;
- posizioni dei fantasmini;
- eventuali collisioni del Pac-Man corrente con i fantasmini;
- eventuali collisioni con i punti;
- posizione della camera in base alla posizione del Pac-Man.

Sono stati implementati meccanismi di awareness che consentono ai Pac-Man di stabilire se un dato peer è offline. In pratica, ogni Pac-Man ha un contatore che viene incrementato ad ogni frame, gli altri in gioco controllano, ogni 60 frame, che questo contatore si sia incrementato. Se così non fosse, il Pac-Man viene etichettato come offline e viene aggiornato lo stato condiviso.

Dopo aver perso le vite a disposizione il Pac-Man entra nello stato di game-over (figura 10). Qui dopo aver ricalcolato i target dei fantasmini è possibile visualizzare gli altri giocatori finché tutti non hanno finito il gioco.

Infine, vengono visualizzati i punteggi di tutti i giocatori in ordine decrescente (figura 11) e la connessione WebRTC viene terminata.

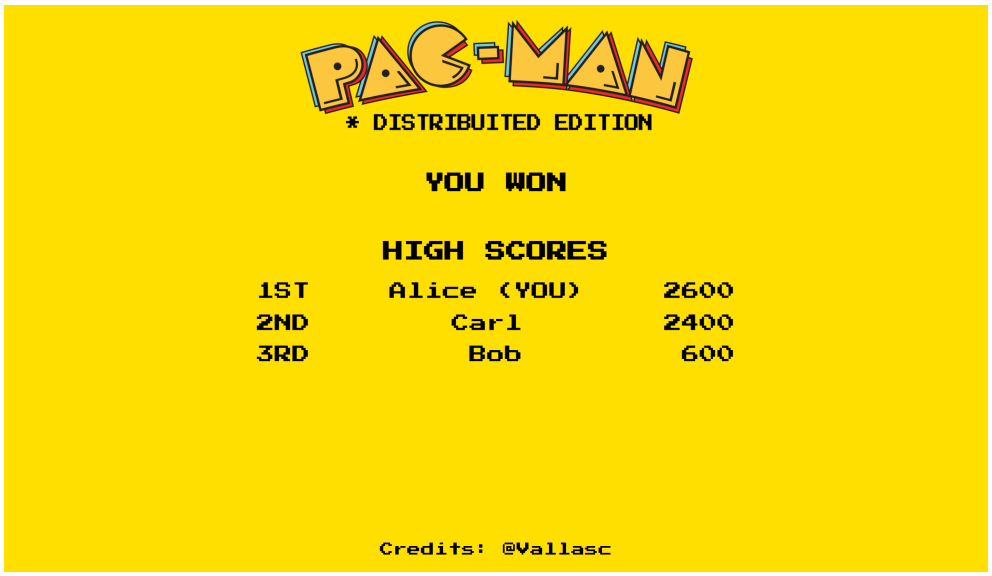


Figura 11: schermata dei punteggi.

7 VALUTAZIONI

7.1 Valutazione empirica

Durante lo sviluppo del software è stato possibile valutare in maniera empirica il comportamento del sistema. Da quattro peer in su l'overhead delle connessioni si comincia a notare, soprattutto quando si utilizzano dispositivi poco performanti (es. smartphone), e i fantasmini non gestiti dal Pac-Man corrente si muovono a scatti. La partita nel complesso é giocabile senza grossi problemi. In generale, il frame-rate è stabile nel tempo, ad eccezione delle rare volte in cui *Yjs* deve sincronizzare i CRDT o deve effettuare operazioni di garbage collection.

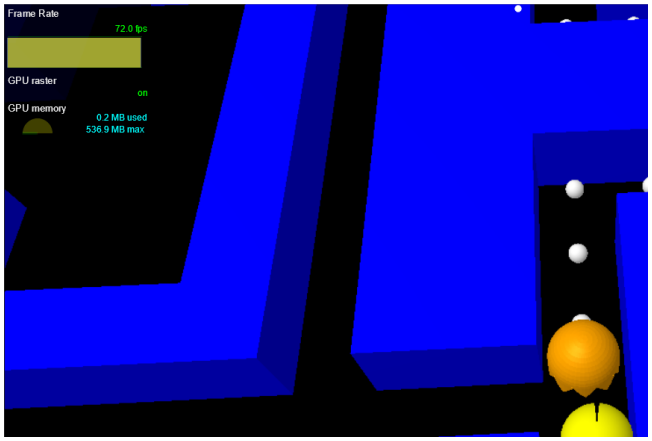


Figura 12: test frame rate.

7.2 Tolleranza ai guasti

Il sistema è altamente fault-tollerant, riesce a tollerare un numero di guasti pari ad $n-1$ (n numero di peer) e l'utente rimasto continuerebbe a giocare normalmente. Anche eventuali disconnessioni temporanee sono gestite correttamente ricalcolando i Pac-Man target dei fantasmini ed escludendo temporaneamente il peer dal gioco.

Nel caso ottimo i giocatori terminano tutti con lo stesso stato. In quello pessimo, per qualche errore non riescono a risincronizzare i CRDT, lo stato finale è diverso, ma comunque coerente nella loro versione locale del gioco.

7.3 Memoria utilizzata

I CRDT occupano notoriamente una gran quantità di spazio in quanto mantengono tutto lo storico delle operazioni. *Yjs* integra un meccanismo di garbage collection che elimina dal CRDT, quando possibile, le operazioni ridondanti o non più necessarie, e fa l'unione delle strutture dati interne. Nello specifico, nel progetto, è molto importante attivare questa funzionalità perché ad ogni frame ogni pacman aggiorna i CRDT incrementandone la dimensione. Nei test effettuati (figura 13) lo heap del programma è rimasto stabile a 90MB indipendentemente dal numero di giocatori.

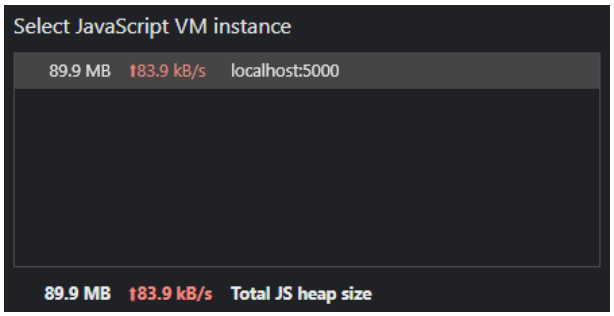


Figura 13: test frame rate.

7.4 Cheaters

Non è stato implementato nessun approccio per evitare che un peer possa imbrogliare. Visto che tutti i giocatori hanno una loro copia dello stato e possono modificarla a piacimento è molto difficile capire se un giocatore sta barando. Se anche fosse identificato il dispositivo impostore, tutti gli altri peer dovrebbero chiudere la connessione con lo stesso, il che inevitabilmente porterebbe all'utilizzo di algoritmi di consenso distribuito, ma come è stato visto in (1) non è attuabile in contesti real-time.

7.5 Test di carico

Dai test effettuati emergono i limiti di questo approccio, il gioco risulta fluido con al massimo 5 giocatori connessi sulla stessa rete. Aumentando i Pac-Man si percepiscono dei rallentamenti nell'interfaccia e nell'aggiornamento dello stato condiviso. Questo problema è causato dall'overhead che hanno le multiple connessioni WebRTC gestite dal browser.

8 CONCLUSIONI

Nel presente documento viene mostrato come è stato realizzato il gioco Pac-Man in modo da poter essere eseguito in un contesto distribuito. In particolare, sono stati utilizzati i *Conflict-free Replicated Data Type* per gestire lo stato del sistema implementando il concetto di *Strong Eventual Consistency*. Come visto in precedenza, il gioco è funzionante ma presenta alcune limitazioni (cheaters, rallentamenti, ecc.). In definitiva, applicazioni distribuite di questo genere possono essere realizzate sapendo che con l'aumentare dei giocatori il sistema subirà inevitabilmente dei rallentamenti impattando sulla user experience.

L'intero codice è disponibile alla repository GitHub del progetto[1].

1 https://github.com/Vallasc/Distributed-PacMan

BIBLIOGRAFIA

[1] Glenn Ricart and Ashok K Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the ACM*, 24(1):9–17, 1981.

[2] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OSDI*, volume 99, pages 173–186, 1999.

[3] Leslie Lamport et al. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[4] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343477–343502. Portland, OR, 2000.

[5] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, jan 2009.

[6] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free Replicated Data Types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS 2011 - 13th International Symposium Stabilization, Safety, and Security of Distributed Systems*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400, Grenoble, France, October 2011. Springer.

[7] Nuno Preguiça. Conflict-free replicated data types: An overview, 2018.

[8] Developing with riak kv. https://docs.riak.com/riak/kv/latest/developing/data-types/index.html, (ultimo accesso 1 Febbraio, 2022).

[9] Facebook announces apollo. https://dzone.com/articles/facebook-announces-apollo-qcon, 2014 (ultimo accesso 1 Febbraio, 2022).

[10] Anti-entropy using crdts on ha datastores. https://archive.qconsf.com/sf2019/presentation/modern-cs, 2019 (ultimo accesso 1 Febbraio, 2022).

[11] Azure cosmos db: Pushing the frontier of globally distributed databases. https://azure.microsoft.com/en-us/blog/azure-cosmos-db-pushing-the-frontier-of-globally-distributed-databases/, 2018 (ultimo accesso 1 Febbraio, 2022).

[12] Active-active geo-distribution (crdts-based). https://redis.com/redis-enterprise/technology/active-active-geo-distribution/, (ultimo accesso 1 Febbraio, 2022).

[13] elmerfs. https://github.com/scality/elmerfs, (ultimo accesso 1 Febbraio, 2022).

[14] Svelte. https://svelte.dev/, (ultimo accesso 1 Febbraio, 2022).

[15] Typescript official page. https://www.typescriptlang.org/, (ultimo accesso 1 Febbraio, 2022).

[16] Three.js. https://threejs.org/, (ultimo accesso 1 Febbraio, 2022).

[17] Yjs doc. https://docs.yjs.dev/, (ultimo accesso 1 Febbraio, 2022).

[18] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. pages 39–49, 11 2016.

[19] Yjs internals. https://github.com/yjs/yjs/blob/main/INTERNALS.md, (ultimo accesso 1 Febbraio, 2022).

[20] y-webrtc. https://github.com/yjs/y-webrtc, (ultimo accesso 1 Febbraio, 2022).

[21] developer.mozilla.org. https://developer.mozilla.org/en-US/docs/Web/API/WebRTC_API/Connectivity, (ultimo accesso 1 Febbraio, 2022).

[22] Mark Handley, Van Jacobson, Colin Perkins, et al. Sdp: session description protocol, 1998.

[23] Jonathan Rosenberg et al. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. Technical report, RFC 5245, April, 2010.

[24] Rohan Mahy, Philip Matthews, and Jonathan Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). Technical report, RFC 5766, 2010.

[25] Jonathan Rosenberg, Joel Weinberger, Christian Huitema, and Rohan Mahy. Stun-simple traversal of user datagram protocol (udp) through network address translators (nats). 2003.