SISTEMES MULTIJUGADOR

Pràctica 2

Projecte: *Mossega'm* - Snake Multijugador

Alumnes:

- Federico Diaz (u1988492)
- Claudia Rebeca Hodoroga (u1988492)

Curs: 2025-2026

Índex

---

1. Concepte del Projecte

Mossega'm és una implementació multijugador del clàssic joc Snake, dissenyada per demostrar els principis fonamentals d'interactivitat en sistemes multijugador. El projecte se centra en la sincronització d'estat en temps real mitjançant HTTP Polling, la gestió de conflictes entre jugadors i la mesura de la latència de xarxa.

El joc permet a dos jugadors competir simultàniament en un tauler compartit de 40x40 cel·les. Cada jugador controla una serp que es mou automàticament i pot canviar de direcció mitjançant el teclat. L'objectiu és menjar fruites per créixer mentre s'evita col·lisionar amb les parets o amb la serp de l'oponent.

---

2. Estat Actual i Abast

2.1 Funcionalitats Implementades

El projecte actual compleix tots els requisits d'interactivitat especificats:

- **Accés remot:** Els jugadors poden connectar-se des de diferents dispositius dins de la mateixa xarxa local.
- **Interacció paral·lela:** Múltiples partides poden existir simultàniament sense interferències.
- **Sincronització d'estat:** El servidor manté l'estat de referència i tots els clients veuen el mateix estat del joc.
- **Detecció i resolució de conflictes:** El sistema identifica i resol situacions com jugadors competint per la mateixa fruita.
- **Mesura de latència:** El sistema calcula i mostra la latència de cada jugador en temps real.

2.2 Funcionalitats Pendents per a la Següent Iteració

Algunes funcionalitats s'han deixat per a una futura iteració que integrarà el mòdul de gestió d'usuaris:

Accés remot complet: Tot i que el sistema està preparat per funcionar en xarxa local, la configuració automàtica per a accés des d'internet es deixa per a la integració amb el sistema d'autenticació d'usuaris. Això permetria implementar mesures de seguretat adequades abans d'exposar el servidor públicament.

Predicció del client: S'ha considerat implementar predicció del moviment al client per compensar latències altes (>150ms), però s'ha decidit posposar aquesta funcionalitat. Afegeix complexitat significativa (cal implementar reconciliació d'estat quan hi ha discrepàncies); segon, el joc actual és perfectament jugable fins a latències de 100-150ms, que són típiques en xarxes locals. Per a una futura versió amb jugadors en ubicacions geogràfiques distants, aquesta funcionalitat seria prioritària.

Historial de partides: Actualment les partides finalitzades no es guarden. La taula `game_history` amb estadístiques detallades s'implementarà quan s'integri amb el sistema d'usuaris, permetent classificacions i seguiment de progressió.

---

3. Decisions Tècniques

3.1 Estructura de la Base de Dades

S'ha optat per SQLite com a base de dades per diverses raons pràctiques:

Simplicitat d'instal·lació: SQLite està integrat en PHP sense necessitat de serveis addicionals, facilitant el desplegament i les proves.

Persistència lleugera: Tot i ser una base de dades embeguda, ofereix persistència adequada per a l'abast del projecte. L'estat del joc sobreviu a reinicis del servidor.

Facilitat de depuració: Poder obrir el fitxer `.db` amb eines visuals facilita enormement la inspecció de l'estat durant el desenvolupament.

L'estructura principal es compon de dues taules:

Taula `game_state`: Emmagatzema l'estat complet de cada partida. La decisió més significativa ha estat guardar les serps i fruites com a JSON en comptes de taules relacionals. Aquesta decisió respon a:

- Simplificació de queries (un sol SELECT retorna tot l'estat)
- Estructura natural per a JavaScript (parsejat directe sense transformacions)
- Millor rendiment (menys JOINs)

El compromís és que no podem fer queries SQL sobre coordenades específiques, però això no és necessari ja que sempre operem sobre l'estat complet.

Taula `player_latency`: Registra les mesures de latència de cada jugador. Això permet calcular avantatges/desavantatges i podria utilitzar-se en el futur per a matchmaking basat en qualitat de connexió.

3.2 Càlcul de Moviments i Estat Autoritatiu

El servidor és completament autoritatiu: tots els càlculs de lògica de joc s'executen al servidor. Els clients són "tontos" i només renderitzen l'estat rebut.

Justificació: Aquest model garanteix que no hi pot haver trampes (impossible manipular el client per guanyar) i que tots els jugadors veuen exactament el mateix. El cost és que la latència és més perceptible, però per a un joc amb tick rate de 500ms (2 moviments per segon), aquest model és més que adequat.

Implementació del tick: El servidor calcula moviments cada 500ms. Hem escollit aquesta velocitat després de diverses proves:

- 250ms era massa ràpid i feia el joc estressant
- 1000ms era massa lent i feia el joc avorrit
- 500ms ofereix un equilibri que permet reacció sense ser frenètic

El càlcul es fa dins de l'endpoint `get_state`. Quan un client fa polling, el servidor verifica si han passat 500ms des de l'últim moviment. Si és així, calcula la nova posició de totes les serps, detecta col·lisions i actualitza l'estat a la base de dades abans de retornar la resposta.

3.3 Delegació de Decisions

Totes les decisions crítiques es prenen al servidor:

- **Validació de direccions:** El servidor verifica que no es pugui fer marxa enrere (direcció oposada). Si un client envia una direcció invàlida, s'ignora sense afectar el joc, i es mostra un missatge.

- **Detecció de col·lisions:** El servidor és l'única autoritat per determinar si hi ha hagut col·lisió. Això evita situacions en què dos clients podrien tenir interpretacions diferents.

- **Assignació de fruites:** El servidor decideix qui menja cada fruita basant-se en l'ordre de processament.

L'única "decisió" que pren el client és quin input enviar, i fins i tot això està limitat per un cooldown de 100ms per prevenir spam.

3.4 Mesura i Ús de la Latència

La latència es mesura mitjançant pings periòdics cada 2 segons:

Procediment: El client envia el seu timestamp al servidor. El servidor calcula la diferència i la guarda a la base de dades. La fórmula utilitzada és:

Latència (ms) = (Timestamp_Servidor - Timestamp_Client) × 500

El factor 500 ve de convertir segons a mil·lisegons (×1000) i dividir per 2 perquè només mesurem el viatge d'anada.

Ús actual: La latència es mostra a la interfície amb codificació per colors:

- Verd (<50ms): Excel·lent
- Groc (50-150ms): Acceptable
- Vermell (>150ms): Problemàtic

També es calcula i mostra l'avantatge/desavantatge respecte a l'oponent (si la diferència supera els 30ms).

Ús futur: Tot i que actualment la latència és només informativa, la infraestructura està preparada per a:

- Compensació de latència en resolució de conflictes (actualment el jugador 1 té prioritat sempre, està pendent de millorar)
- Matchmaking basat en qualitat de connexió
- Ajust dinàmic de la freqüència de polling segons latència

3.5 Limitació a Dos Jugadors

La decisió de limitar el joc a exactament dos jugadors és per l'arquitectura i la simplicitat de comprenció:

La base de dades té columnes específiques per a `player1_*` i `player2_*`. Suportar N jugadors requeriria:

- Redissenyar l'esquema amb una taula `players` separada
- Reescriure la lògica de col·lisions (actualment assumeix dues serps)
- Modificar el sistema de puntuació (necessitaríem ranking en lloc de head-to-head)

Amb dos jugadors és més senzill entendre i implementar els conceptes d'interactivitat. Tots els conflictes possibles són més fàcils de raonar i verificar.

Futura: Una extensió a N jugadors és totalment viable i està planificada com a millora futura, però requeriria canvis substancials que van més enllà de l'abast d'aquesta pràctica centrada en interactivitat.

3.6 Freqüència de Polling

S'ha escollit una freqüència de polling de 200ms (5 vegades per segon) després d'analitzar diverses opcions:

Anàlisi:

- **50ms (20 Hz):** Generaria 40 requests/segon amb dos jugadors. Massa overhead quan el servidor només actualitza cada 500ms.
- **100ms (10 Hz):** Encara excessiu. Gastaríem recursos capturant el mateix estat múltiples vegades.
- **200ms (5 Hz):** Equilibri òptim. Captura tots els canvis del servidor amb marge, sense overhead innecessari.
- **500ms (2 Hz):** Arriscat. Hi ha possibilitat de perdre un tick del servidor si el polling no està perfectament sincronitzat.

La relació ideal és polling a 2.5× la freqüència del tick del servidor, que és exactament el que tenim.

Mesures: En proves amb 10 partides simultànies (20 jugadors):

- Requests/segon: ~100
- Ús de CPU: 5-10%
- Latència de resposta del servidor: 5-15ms

Aquestes mètriques confirmen que la freqüència escollida és sostenible.

3.7 Enviament d'Informació al Servidor

Els clients només envien dos tipus de missatges:

Canvis de direcció (`set_direction`): Quan el jugador prem una tecla, s'envia la nova direcció al servidor. No s'envia la posició ni res més, només la intenció. El servidor decideix si aplicar-la.

Pings (`ping`): Periòdicament per mesurar latència. Aquests no afecten l'estat del joc.

Fent això aconseguim que el client no pugui mentir sobre la seva posició i requests molt petits (~100 bytes cadascun)

El sistema de cua de direccions (`next_direction`) permet que no es perdin inputs entre ticks: si el jugador prem una tecla just després d'un moviment, aquesta es guarda i s'aplica en el següent tick.

---

4. Resolució de Conflictes

El sistema identifica i resol diversos tipus de conflictes que poden sorgir en el joc multijugador.

4.1 Competició per la Mateixa Fruita

Situació: Ambdós jugadors mouen les seves serps cap a la mateixa fruita en el mateix tick del servidor.

Resolució implementada: Ordre de processament determinista. El servidor sempre processa primer el moviment del Jugador 1. Si la seva nova posició conté una fruita, aquesta es consumeix i s'elimina del array abans de processar el Jugador 2.

Aquest mètode és simple i consistent. El Jugador 1 sempre té un petit avantatge (ser "host"), cosa que s'ha de modificar.

Alternativa considerada: Utilitzar la latència per decidir el guanyador (qui té millor connexió s'emporta la fruita). Es va descartar perquè:

- Afegeix complexitat sense benefici clar
- Pot ser injust si les latències són molt dispars
- És menys predictible per als jugadors

Millora futura: Quan s'implementi matchmaking, es podria emparellar jugadors amb latències similars per minimitzar l'impacte d'aquest ordre de processament. Per no afavorir a cap dels 2 jugadors, si es troben amb una latència similar dins d'un rang, la decisió serà aleatoria.

4.2 Col·lisió Frontal (Head-to-Head)

Situació: Les capçaleres de dues serps es mouen a la mateixa cel·la en el mateix tick.

Resolució implementada: El mateix ordre de processament s'aplica aquí. El Jugador 1 es mou primer; si col·lisiona amb el Jugador 2 (que encara està a la posició anterior), el Jugador 1 perd. Si el Jugador 1 sobreviu, s'avalua el moviment del Jugador 2.

Resultat pràctic: En una col·lisió frontal real, si ambdós es mouen un cap a l'altre:

1. El Jugador 1 es mou i col·lisiona amb la posició actual del Jugador 2: Jugador 1 perd
2. Cas contrari: El Jugador 2 es mou i pot col·lisionar amb la nova posició del Jugador 1: Jugador 2 perd

Aquest sistema garanteix que sempre hi ha un guanyador clar. Mai hi ha empat, cosa que simplifica la lògica del joc.

4.3 Fruita a la Paret

Situació: Una fruita apareix en una cel·la de la cantonada i un jugador intenta menjar-la, la qual cosa el faria col·lisionar amb la paret.

Resolució implementada: L'ordre de verificació dins de la funció `move_snake` és crític:

1. Es verifica si la nova posició conté una fruita i es marca com a "menjada"

2. Es verifica si la posició és vàlida (dins del tauler)
3. Si hi ha col·lisió amb paret, es retorna l'estat de col·lisió

El jugador pot menjar fruites situades exactament a les cel·les (0, y), (39, y), (x, 0) o (x, 39) sense morir. La seva serp creix normalment.

Aquesta decisió millora la jugabilitat. És frustrant que una fruita sigui "impossible" de menjar. A més, les fruites a la vorera són més arriscades (menys espai per maniobrar després), així que hi ha un equilibri de risc-recompensa.

Implementació tècnica: Això es va aconseguir reorganitzant l'ordre de les verificacions a la funció `move_snake`. Inicialment, es verificava primer la col·lisió amb la paret, cosa que feia impossible menjar fruites de la vora.

4.4 Reversió de Direcció

Situació: Un jugador intenta moure's en la direcció oposada a la seva direcció actual (per exemple, està movent-se cap amunt i prem la tecla "avall").

Resolució implementada: El servidor valida cada canvi de direcció abans d'aplicar-lo. Si detecta que la nova direcció és oposada a l'actual, ignora la petició i respon amb `{"success": true, "ignored": true}`.

El client mostra un missatge visual breu ("No pots revertir!") però el joc continua sense interrupcions.

Prevenir la reversió és essencial en Snake. Sense aquesta validació, un jugador podria "suïcidar-se" accidentalment amb un sol error de tecla. La validació al servidor (no només al client) assegura que fins i tot manipulant el client no es pot fer trampa.

4.5 Spawn de Fruites en Posicions Ocupades

Situació: Quan una fruita es consumeix, cal generar-ne una de nova. Existeix el risc que aparegui sobre una serp.

Resolució implementada: La funció `random_fruits` rep com a paràmetre un array d'exclusió amb totes les cel·les ocupades per serps. Genera posicions aleatòries i les verifica contra aquest array. Si hi ha col·lisió, torna a intentar-ho.

Límit de seguretat: Després de 100 intents fallits, la funció retorna menys fruites de les sol·licitades. Això prevé loops infinits en partides on el tauler està quasi ple.

Aquest mètode és simple i eficient. Amb un tauler de 40×40 (1600 cel·les) i serps de fins a ~50 cel·les, la probabilitat de generar múltiples col·lisions consecutives és baixa. En proves amb serps molt llargues (>100 segments), mai s'ha arribat al límit de 100 intents.

4.6 Inputs Múltiples entre Ticks

Situació: Un jugador prem diverses tecles en menys de 500ms (el temps entre ticks del servidor).

Resolució implementada: Sistema de cua amb "última entrada guanya". Cada jugador té dos camps a la base de dades:

- `player_direction`: Direcció actual (s'està aplicant ara)
- `player_next_direction`: Direcció en cua (s'aplicarà en el següent tick)

Quan arriba un nou input, sobreescriu `next_direction`. En el següent tick, `next_direction` es copia a `direction`.

Cooldown addicional: Al client hi ha un cooldown de 100ms entre canvis de direcció per prevenir spam accidental.

Justificació: Aquest sistema permet que els jugadors amb reflexos ràpids puguin "preparar" el seu pròxim moviment sense perdre inputs, però evita que múltiples inputs acumulats generin comportaments inesperats.

---

5. Millores Futures

5.1 Matchmaking Basat en Latència

Proposta: Quan un jugador vol unir-se a una partida, en lloc de mostrar totes les partides disponibles, mostrar preferentment aquelles on l'host té una latència similar.

Implementació:

1. Quan un jugador crea una partida, es registra la seva latència típica (mitjana de les últimes 10 mesures)
2. Quan un jugador intenta unir-se, es calcula la seva latència actual
3. El servidor ordena les partides disponibles per similitud de latència
4. Es mostren les 5 partides més equilibrades

Benefici: Redueix l'avantatge inherent del Jugador 1 en resolució de conflictes. Partides més justes.

Complexitat: Baixa. Només requereix un camp addicional a `game_state` i modificar el query de `list_games`.

5.2 Unificació de Controls

Proposta: En lloc d'assignar WASD al Jugador 1 i fletxes al Jugador 2, permetre que ambdós jugadors configurin els seus controls preferits abans de començar.

Benefici: Millor accessibilitat. Alguns jugadors prefereixen fletxes, altres WASD. Alguns portàtils tenen disposicions de teclat incòmodes per a un dels esquemes.

Implementació: Pantalla de configuració al lobby on cada jugador pot seleccionar el seu esquema de controls. Es guarda a una cookie del navegador per persistir entre partides.

5.3 Hosting en Serveis Cloud (Azure)

Proposta: Desplegar el joc en Azure App Service o similar per permetre accés global sense necessitat de configuració de xarxa local.

Consideracions:

- Requereix migració de SQLite a Azure SQL Database o Cosmos DB
- Necessita implementar autenticació robusta abans d'exposar públicament
- Azure ofereix tier gratuït per a projectes educatius

Benefici: Els jugadors podrien jugar des de qualsevol lloc sense restriccions de xarxa. Facilitaria proves i demos.

5.4 Suport per a Més de Dos Jugadors

Proposta: Expandir el joc per suportar 3-8 jugadors simultanis.

Canvis necessaris:

- Redissenyar esquema de base de dades amb taula `players` separada
- Modificar lògica de col·lisions per avaluar N serps
- Implementar sistema de puntuació amb ranking
- Ampliar tauler (potser 60×60) per acomodar més serps
- UI més complexa per mostrar múltiples puntuacions

Desafiament: Amb més jugadors, la resolució de conflictes es complica exponencialment. Caldria repensar l'estratègia actual d'ordre de processament.

5.5 Millores d'UX/UI

Diverses millores detectades durant les proves però ajornades per centrar-se en funcionalitats core:

Feedback visual millorat:

- Animació quan una serp menja una fruita
- Efecte "d'explosió" quan hi ha col·lisió
- Rastre lleuger darrere de la capçalera per indicar direcció

Informació contextual:

- Tooltip sobre els indicadors de latència explicant què significa

- Tutorial interactiu per a nous jugadors
- Indicador visual de quin jugador ets

Opcions de personalització:

- Temes de color per al tauler
- Opcions d'accessibilitat (mida de cel·les, contrast alt)

Històric i estadístiques:

- Taula de classificació global
- Gràfic de progressió de puntuació

Aquestes millores, tot i no ser crítiques per a la funcionalitat core, millorarien significativament l'experiència d'usuari i serien prioritàries en una versió comercial del joc.

---

6. Conclusions

Aquest projecte ha proporcionat una comprensió pràctica dels desafiaments reals del desenvolupament multijugador. La necessitat de pensar en termes de "què passa quan..." (dos jugadors fan X simultàniament, un jugador perd connexió, etc.) ha desenvolupat una mentalitat de "disseny defensiu" valuosa per a qualsevol desenvolupament de software.

La decisió de prioritzar la funcionalitat core i deixar millores per a futures iteracions ha estat encertada, permetent centrar-se en els conceptes fonamentals sense distraccions.