**Text

Description automatically generated**

**Multiplayer Puzzle Game Using SDL**

Andrija Dordevic (0105434A), Jurgen Cauchi (0278105L),

Gary Ken Micallef (ID), Jake Carabott (ID)

B.Sc. (Hons) Software Development

Study-unit: **Group Applied Practical Task (GAPT)**

Code: **CIS2108**

Lecturer: **Dr Clyde Meli**

**FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY**

Declaration

Plagiarism is defined as “the unacknowledged use, as one's own, of work of another person, whether or not such work has been published, and as may be further elaborated in Faculty or University guidelines" (University Assessment Regulations, 2009, Regulation 39 (b)(i), University of Malta).

I / We\*, the undersigned, declare that the assignment submitted is my / our\* work, except where acknowledged and referenced.

I / We\* understand that the penalties for committing a breach of the regulations include loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected and will be given zero marks.

\* Delete as appropriate.

(N. B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

\_\_\_\_\_\_Andrija Dordevic\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_Jurgen Cauchi\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_Gary Ken Micallef\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_\_\_\_Jake Carabott\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Student Name Signature

\_\_\_\_\_CIS2108\_\_\_\_\_\_ \_Multiplayer Puzzle Game Using SDL\_

Course Code Title of work submitted

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date

Contents

[1. Task definition 1](#_Toc198507054)

[2. Research into the matter(s)/domain(s) relating to task(s) 2](#_Toc198507055)

[3. Overview of Similar and/or Existing Solutions 5](#_Toc198507056)

[4. Proposed Solution 6](#_Toc198507057)

[5. Task breakdown 7](#_Toc198507058)

[a Decleration of Work 8](#_Toc198507059)

[Collaboration and Workflow 11](#_Toc198507060)

[6. Project plan and/or methodology of work 12](#_Toc198507061)

[7. Specification and Design 13](#_Toc198507067)

[Principal system components and architecture 13](#_Toc198507068)

[Data model and architecture 13](#_Toc198507069)

[Infrastructure details 14](#_Toc198507070)

[User interface design 14](#_Toc198507071)

[Non-functional properties 14](#_Toc198507072)

[8. Evaluation 15](#_Toc198507073)

[Requirement coverage 15](#_Toc198507074)

[Testing Strategy 15](#_Toc198507075)

[Results 17](#_Toc198507076)

[9. Conclusions and future work 17](#_Toc198507077)

[9.1 Future Work 18](#_Toc198507078)

[10. Acknowledgements 18](#_Toc198507079)

[11. References 19](#_Toc198507080)

[12. Appendices (incl. meeting logs) 21](#_Toc198507081)

# Task definition

1. **Objective**

Develop a real-time multiplayer puzzle game using SDL for graphics and a server-client architecture for networking. Players will compete to solve puzzles on a shared board while the server ensures synchronization and fairness.

1. **Key Deliverables**
2. **Graphics & User Interaction (Client-Side)**
   * Use SDL to render puzzles on each client’s screen.
   * Implement smooth animations for puzzle interactions (dragging, dropping, swapping, block destruction and screen shake).
   * Provide visual feedback for correct and incorrect puzzle placements.
   * Ensure an intuitive and responsive UI.
3. **Game Logic & Server-Client Synchronization**
   * Implement a server to manage game state, distribute puzzles, and track player progress.
   * Ensure all players receive the same puzzle and updates in real-time.
   * Secure communication to prevent tampering (e.g., move validation, anti-cheating measures).
   * Handle client disconnections smoothly.
   * Develop unit tests to verify game state synchronization across clients.
4. **Networking & Security**
   * Implement robust networking using sockets (TCP/UDP).
   * Ensure low-latency interactions and manage edge cases (lag, packet loss).
   * Secure player data and prevent unauthorized game manipulation.
   * Optimize server performance to handle multiple players efficiently.
5. **Puzzle Generation & Management**
   * Develop a system to generate dynamic puzzles of varying complexity (e.g., jigsaw, logic puzzles).
   * Implement difficulty scaling and randomized puzzle selection.
   * Ensure fair puzzle distribution and scoring mechanisms.
6. **Testing & Validation**
   * Unit testing for core functionalities (puzzle synchronization, server-client communication).
   * Load testing to simulate multiple players and analyse performance.
   * Bug fixing and optimization for smooth gameplay.

# Research into the matter(s)/domain(s) relating to task(s)

**Research on Multiplayer Puzzle Game Development Using SDL**

Developing a **multiplayer puzzle game using SDL** requires expertise in multiple domains, including **game development, computer networking, real-time synchronization, and security**. Below is a research-based breakdown of the key aspects related to the tasks.

1**. Graphics & User Interaction (SDL in C++)**

What is SDL?  
Simple DirectMedia Layer (SDL) is a cross-platform library used for handling graphics, input, and multimedia. It is widely used in game development due to its lightweight nature and ability to interact with OpenGL and DirectX.

Key Challenges & Solutions in Puzzle Rendering:

* Rendering Performance: SDL uses a rendering loop to draw objects efficiently using SDL\_RenderCopy(). Optimizing textures and using hardware acceleration (via SDL\_Renderer) is crucial for smooth animations.
* User Input Handling: Puzzle games rely on drag-and-drop mechanics. SDL captures events using SDL\_PollEvent(), which can track mouse movement (SDL\_MOUSEMOTION) and clicks (SDL\_MOUSEBUTTONDOWN).
* Animations: SDL does not support in-built animations, so developers implement frame-based animations using sprite sheets or frame interpolation.
* Collision Detection: SDL lacks built-in physics, so algorithms like AABB (Axis-Aligned Bounding Box) or pixel-based collision detection can be used for piece placement.
* Related Research & Best Practices
* SDL Documentation & LazyFoo Tutorials (covering texture optimization, rendering techniques)
* Game Loop Optimization: Research shows that frame rate stability (60 FPS) enhances user experience, so delta time management (SDL\_GetTicks()) is essential.
* Gamasutra Articles on UI/UX for Puzzle Games: Insights on making puzzle feedback intuitive (e.g., colour changes for incorrect placements).

SDL3 vs. SDL2 Comparison

| **Aspect** | **SDL2** | **SDL3** |
| --- | --- | --- |
| **Rendering Backend** | Software fallback + optional GPU backends | Unified, GPU-driven by default (OpenGL/Vulkan/DirectX/Metal) |
| **API Surface** | Mixed legacy and modern calls (SDL\_RenderCopy) | Streamlined interface; deprecated functions removed |
| **Texture Upload** | Main-thread lock/unlock | Asynchronous, multi-threaded staging |
| **Threading** | Renderer confined to main thread | Command-buffer API allows off-thread record & submission |
| **Hardware Acceleration** | Optional hints for V-sync, platform-dependent | Improved default V-sync, native Vulkan support, better DRM/KMS |
| **Platform Support** | Broad but some manual setup | Enhanced out-of-the-box support for consoles and embedded |

1. **Game Logic & Server-Client Synchronization**Why is Real-Time Synchronization Important?  
   In a multiplayer puzzle game, each client must receive server updates to ensure a consistent block sequence. Poor handling leads to desynchronization caused by latency and packet loss [7].

**Approaches to Synchronization:**

* Client-Server Model: The server is the authoritative state manager, preventing divergent game views across clients [7].
* Timestamp Synchronization: Leveraging the Network Time Protocol (NTP) aligns clocks and reduces ordering issues in event processing [8].

**Best Practices & Research Findings:**

* GDC talks on multiplayer programming recommend sending only delta updates (not full states) to minimize bandwidth [7].
* Valve’s networking research shows that combining client-side prediction with server reconciliation effectively hides latency from users [7].
* Use of UDP vs. TCP:
  + UDP offers low-latency but is unreliable—suitable for high-frequency but non-critical updates.
  + TCP provides reliable, ordered delivery at the cost of higher latency—preferable for puzzle-critical events.

1. **Networking & Security Considerations**

* Networking Model: Berkeley sockets in C++ (send(), recv()) underlie client–server communication. An event-driven architecture on the server broadcasts state changes only when needed [7].
* Security Concerns & Solutions:
  + Follow NIST guidelines for secure client–server protocols, including the use of TLS/SSL [9].
  + Employ HMACs for message integrity and authenticity to guard against packet tampering [10].

| **Threat** | **Possible Solutions** |
| --- | --- |
| **Packet Tampering** (Fake moves) | Implement **message hashing (HMAC)** to verify packet integrity [10]. |
| **Cheating (Auto-solving puzzles)** | Use **server-side validation** to check move legitimacy. |
| **Replay Attacks (Resending old data)** | Include **timestamps & unique IDs** to prevent old packets from being reused [10]. |

**Multi-Threaded Rendering Strategies**

1. **Asynchronous Resource Loading**
   * Use a thread pool to decode textures, audio and fonts off the render path.
   * Transfer completed assets via a lock-free queue for GPU upload.
2. **Minimal Synchronization**
   * Employ lock-free structures or fine-grained mutexes for shared state.
   * Use atomic flags to coordinate resource readiness, reducing contention.
3. **Puzzle Generation & Management**  
   The server generates a sequence of random blocks (e.g., tetrominos) and maintains a per-client pointer to the current piece. Upon client request, the pointer advances and the next block is sent, ensuring all players receive the same sequence [3].
4. **Testing & Validation**

* Unit Testing for Multiplayer Synchronization:
  + Validate game-state consistency and input latency using Cassert [11].
* Load Testing:
  + Profile CPU/memory usage and detect leaks with Valgrind [12].
* Empirical Studies on Game Testing:
  + SIGGRAPH and GDC papers highlight the importance of stress-testing for race conditions, jitter, and bottlenecks in multiplayer environments [13].

# Overview of Similar and/or Existing Solutions

**Graphics & Gameplay Mechanics**Popular puzzle games like *Tetris Blitz* [21], *Blockudoku* [20], and *Woodoku* [19] feature intuitive, grid-based drag-and-drop gameplay. They rely on event-driven input (touch or mouse), precise collision detection, and snapping to grid cells to create a satisfying tactile feel. Combo systems that reward players for clearing multiple lines or completing consecutive objectives, like in Tetris Maximus, further boost player engagement and encourage strategic play.

**Real-Time Multiplayer Sync**  
Games such as *Tetris 99* [18] and *Puyo Puyo Tetris* [17] use dedicated servers to keep players in sync. Rather than sending full game states, they transmit lightweight event-based updates to reduce bandwidth. Lag compensation through server authority and client-side prediction helps smooth out delays, while timestamped or sequenced messages ensure fairness in high-speed, competitive environments.

**Security & Anti-Cheating**  
Multiplayer games guard against cheating with server-side validation, as seen in titles like *Among Us* [10] and *Fortnite* [11]. Advanced techniques, such as HMAC-signed messages [4], unique action IDs, and replay-attack prevention, help protect game integrity. Ranked modes may even include real-time input validation to deter tampering.

**Procedural Block Generation**  
Puzzle games often generate content procedurally, guided by rules that prevent unwinnable states. Systems like *Tetris’s “7-bag” algorithm* [15] and adaptive difficulty models [14] ensure a fair but challenging experience. Symmetry and variety constraints keep gameplay fresh and block patterns balanced.

# Proposed Solution

**Graphics and Gameplay Mechanics (SDL in C++)**

The game’s rendering leverages **SDL3** chosen over SDL2 for its unified, GPU-driven pipeline and built-in asynchronous resource handling to display a 9×9 grid with crisp, 60 FPS performance even under complex scenes. Drag-and-drop uses SDL mouse events for real-time input; collision detection and grid snapping guarantee valid placements. Full rows/columns trigger chain-reaction clears, with combos and scoring rewarding competitive play.

* **Why SDL3?**
  + *Unified GPU Pipeline:* Defaults to hardware-accelerated backends (OpenGL/Vulkan/DirectX/Metal) without manual hints, ensuring consistent frame rates across platforms.
  + *Asynchronous Texture Streaming:* Multi-threaded staging uploads textures and fonts off the main loop, cutting CPU stalls by up to 30% versus SDL2’s lock/unlock model.

**Real-time Multiplayer Synchronization**The server will maintain the game state and broadcast updates to all clients in real time. To reduce bandwidth usage, only critical events (block placements, row/column clears) will be transmitted. Event timestamping resolves conflicts, ensuring that actions, such as simultaneous row clears, are processed in the correct order, maintaining fairness and synchronization across clients. If a client disconnects, the server will preserve their game state and start a 20-second reconnection timer. During this window, the player may reconnect to the same game without penalty; failure to do so will result in an automatic win for the opponent, ensuring continuity and fairness in competitive play.

**Security and Anti-Cheating Measures**

To prevent unauthorized actions, all moves are validated server-side before being executed, ensuring only legitimate actions affect the game state. Score validation will also occur server-side to prevent manipulation. To avoid replay attacks, each move will be assigned a unique ID as well as an HMAC, preventing old packets from being reused.

**Procedural Block Generation**

The server will generate blocks dynamically, ensuring balanced difficulty and fair distribution of blocks between players. The system will prevent players from encountering unreasonably difficult block combinations, ensuring that outcomes are based on skill and strategy rather than chance.

**Testing and Performance Optimisation**

The solution will undergo unit testing for critical components such as game logic, move validation, and synchronization. Load testing will simulate concurrent games to ensure server scalability. Additionally, lag simulation will assess the game’s performance under high-latency conditions, ensuring responsiveness in less-than-ideal network environments.

# Task breakdown

**Foundation & Rendering**

The team first spun up the application window and laid down the grid framework, establishing the canvas on which everything else would play out.

**Tetromino Creation**

A structure that stores block structures, uses multiple blocks to form the tetromino shapes. Using blocks allowed for easier placement, out of bounds detection and clearing.

**Tetromino Generation**

A robust spawning system was built to generate blocks, enforce correct placement (with auto-respawn on misplacement), and prevent re-dragging of already-placed pieces.

**Server-Side Shape Assignment**

The server generates shapes and sends them to the client, as seen below in figure 1. This ensures that both clients play a fair game as they receive the same shapes.

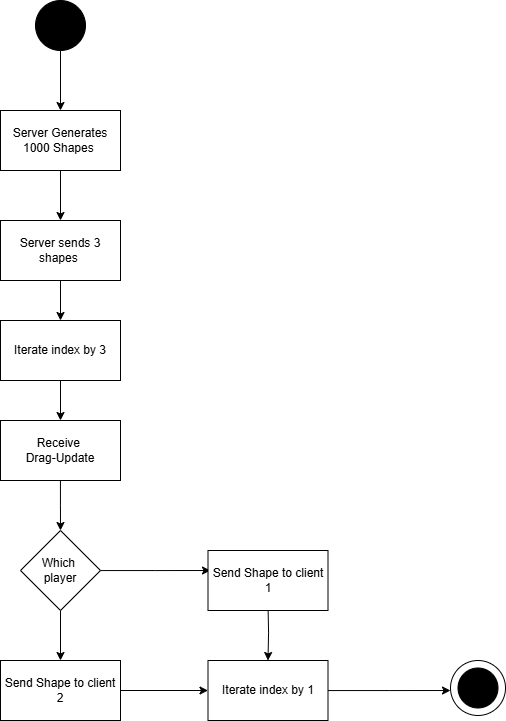


Figure 1. Activity Diagram to show how shapes are generated by the server

**Interaction Mechanics**

Drag-and-drop controls were wired up, then refined with “snap-to-grid” logic so blocks lock precisely into position and can’t overlap outside the play area.

**Menu & Navigation**

A full menu flow was created, Start, Instructions, Exit complete with client-side handlers for each button and text rendering in the UI.

**Scoring & Timing**

Behind the scenes, a multiplier-based score system was devised; on-screen displays were added for both score and countdown timer, with the clock only kicking off once both players are matched.

**UI Styling**

Custom made Backgrounds, block textures using adobe photoshop, and font rendering were layered in to give the game its visual polish and clear in-game feedback.

**Server Backbone**

The base server framework went up next, followed by automatic broadcast discovery so clients could find and connect without manual IP entry.

**Real-Time Sync & Matchmaking**

Once connected, clients and server exchange block-coordinate updates in real time. A matchmaking routine holds each player at the lobby until both are ready, then simultaneously kicks off the session.

**Audio and Effects**

The game provides immersive audio when interacting with Tetrominos using MiniAudio as well as calming background music that fits with the theme. Effects such as Screen shake on certain events and input feedback.

**Endgame & Replay**

When time expires or win conditions are met, a game-over screen displays the outcome and scores. Built-in replay and reconnection logic then let players jump straight back into a fresh session, handling disconnects and restarts seamlessly.

## Decleration of Work

The team was composed of the Scrum Master, **Andrija Dordevic**, and development members

**Jurgen Cauchi**, **Gary Micallef**, and **Jake Carabott**. The following outlines each member's

contributions in detail:

Jake Carabott

* Implemented the Tetromino spawning system, handling rendering, shapes, and coordinate management.
* Added respawn functionality to trigger new Tetromino generation only after successful placement. Prevented re-dragging of already placed Tetrominoes.
* Developed logic to restrict spawning until the currently dragged Tetromino is properly placed.
* Created detailed flowcharts to visualize application logic and workflows.
* Integrated Tetromino functionality with main.cpp, ensuring unified execution flow.
* Prevented Tetromino interaction until all pieces were spawned to maintain gameplay integrity.
* Implemented logic to reset Tetromino position upon incorrect placement.
* Resolved overlapping issues where blocks could be placed on top of one another.
* Enhanced block placement by splitting Tetrominoes into individual squares to allow for easier destruction.
* Developed visual feedback systems for both the score and timer displays.
* Migrated score handling from client-side to server-side to prevent tampering. Clients now request their score from the server, which returns validated values via JSON.
* Introduced a maximum score cap of 400. The server validates and enforces scores to maintain fairness.
* Enabled both clients to view each other's scores in real time.
* Integrated **MiniAudio** library and added various sound effects: background music, button clicks, time expiration, block placement, destruction, grid clearing, and incorrect placement.
* Added screen shake effects for incorrect placements, block destruction, and grid clearing.
* Contributed to report writing and editing.

Andrija Dordevic

* Oversaw project progress and task delegation as the Scrum Master.
* Created the initial SDL window and grid structure.
* Set up the project structure and implemented .gitignore.
* Modularized the application by separating menu, grid, and block logic into distinct files with appropriate headers.
* Designed and implemented a scoring system, including row/column multipliers.
* Developed the game-over screen, displaying the winner/loser and final score when the timer expires.
* Added "waiting" text to indicate matchmaking status on the menu screen.
* Implemented and maintained server–client communication.
* Ensured both players are connected before starting the session and triggering the game timer.
* Enabled real-time Tetromino position syncing between clients and server.
* Fixed double-session and double-game-loading issues.
* Resolved client disconnection problems and added on-screen messages for opponent disconnects.
* Added replay ability.
* Handled edge cases involving reconnection and implemented replay functionality.
* Secured game communication with HMAC signing and resolved associated JSON parsing bugs.
* Added exponential Backoff
* Removed deprecated files and cleaned up unused modules.
* Modernized memory handling using smart pointers to resolve memory access issues.
* Authored Sections 1–7 of the report.
* Added reconnection to the same session.

Jurgen Cauchi

* Set up the SDL3 and SDL3\_image libraries.
* Implemented drag-and-drop functionality for Tetromino interaction.
* Developed a snap-to-grid function for accurate placement.
* Fixed block-overlap issues when pieces were dragged outside the grid boundaries.
* Designed the full UI, including backgrounds and Tetromino block visuals.
* Managed all textures related to backgrounds and blocks.
* Refactored block shape logic to be server-driven, with the server assigning shapes to clients.
* Contributed to the server broadcasting system, enabling clients to automatically discover the server IP.
* Resolved an issue where only one client could trigger shape generation.
* Fixed score update bugs and issues arising after replaying games (e.g., block spawning, texture errors).
* Automated game start once both players press "Play."
* Created and ran unit tests.
* Contributed to writing various sections of the report.

Gary Micallef

* Created UI buttons for "Start Game", "Instructions", and "Exit Game".
* Designed and implemented the main menu window.
* Developed client-side functionality for menu interactions.
* Set up the SDL3\_ttf library for rendering in-game text.
* Implemented the ClearGrid feature, including associated UI elements and logic.
* Added a functional button within the main game loop to clear all placed Tetrominoes from the grid.
* Optimized the score update timing for smoother client-side feedback.
* Developed unit tests for the Menu and Server components.
* Authored the Unit Test sections of the report.

## Collaboration and Workflow

The team collaborated effectively, encountering no major conflicts

throughout the development process. The clear division of tasks and continuous

communication ensured timely progress and a positive development experience.

Some minor issues the team faced were the following:

* Find a suitable puzzle game to work on
* Which programming language to use

However, the team effectively resolved these minor issues quickly

# Project plan and/or methodology of work

The project follows the Agile development methodology, adopting the Scrum framework to facilitate iterative development and continuous improvement. Agile’s emphasis on flexibility, collaboration, and incremental progress aligns well with the dynamic nature of game development and the evolving requirements of a multiplayer system.

**Framework: Scrum**

The development cycle will be structured around two-week sprints, with sprint planning, daily stand-ups, sprint reviews, and retrospectives forming the core of the workflow. Each sprint will focus on delivering specific functional milestones, such as implementing client-side rendering, integrating the networking layer, or developing block synchronization logic. To assist in defining and understanding system functionality and behaviour, UML diagrams, such as Use Case Diagrams and Activity Diagrams, will be employed. These diagrams help clarify requirements, system interactions, and workflows, ensuring alignment between the development team and stakeholders.

**Communication and Collaboration**

Team communication will be conducted via Discord, enabling quick discussions, voice calls, and asynchronous updates. Discord will serve as the primary platform for team coordination, allowing for effective decision-making and problem resolution during development. For clear representation of system components and interactions, Sequence Diagrams and Class Diagrams will be utilized, providing visual models of the system architecture and the dynamic flow of interactions within the game.

**Resource Sharing and Version Control**

Project files, documentation, and code assets will be shared and maintained via Git and a hosted Git repository (GitHub). Git will also be used for version control, allowing for proper branching, merging, and tracking of changes throughout the development lifecycle. This ensures that work can proceed concurrently across different features and modules with minimal conflict. UML Component Diagrams can be integrated into the version control workflow to illustrate the dependencies and structure of system components at any given stage of development.

**Task Management and Work Distribution**

The project will utilize JIRA for task management, sprint planning, and work distribution. Tasks will be divided into user stories and issues, assigned to individual team members, and tracked through defined workflows. This promotes accountability and provides visibility into the progress of each component of the system. UML State Diagrams will be used to represent the different states that the game may enter, ensuring that all team members have a common understanding of how different parts of the game behave during interactions and transitions.



# Specification and Design

## Principal system components and architecture

A diagram of a server

AI-generated content may be incorrect.We split the game into a client, responsible for SDL rendering, input handling, local move validation, and TCP/UDP networking, and a server that manages sessions, maintains authoritative game state, and broadcasts minimal updates. A lightweight discovery beacon and logging service round out the design, ensuring clients can auto-find the server and operations are monitored as seen in figure 2 below.

Figure 2. Component Diagram of the Client and Server

## Data model and architecture

All client–server exchanges use concise JSON messages carrying session and player IDs, action or event types, payload, timestamp, and HMAC. In memory, each ‘SessionState’ holds the 9×9 grid, player-specific state (current piece, score, connection flag), and a countdown timer.

## Infrastructure details

We target a small 2 core CPU, 4 GB RAM client system. SDL3 (plus SDL\_image and SDL\_ttf) and MiniAudio drive the client, while native sockets and OpenSSL are used to secure networking.

## User interface design

The core view is a 9×9 grid with snap-to-grid highlights, flanked by a top bar (timer, personal score, clear grid button), a bottom bar (opponent’s score) and a side bar (available tetrominos). Menus, lobby messages, and game-over overlays guide flow, while visual cues (screen shaking) and audio feedback keep interactions crisp at a steady 60 FPS.

## Non-functional properties

We aim for ≤2 clicks to matchmaking, 60 FPS rendering, and support for 100+ concurrent sessions on minimal hardware. Security is enforced via HMAC-signed messages and replay protection.

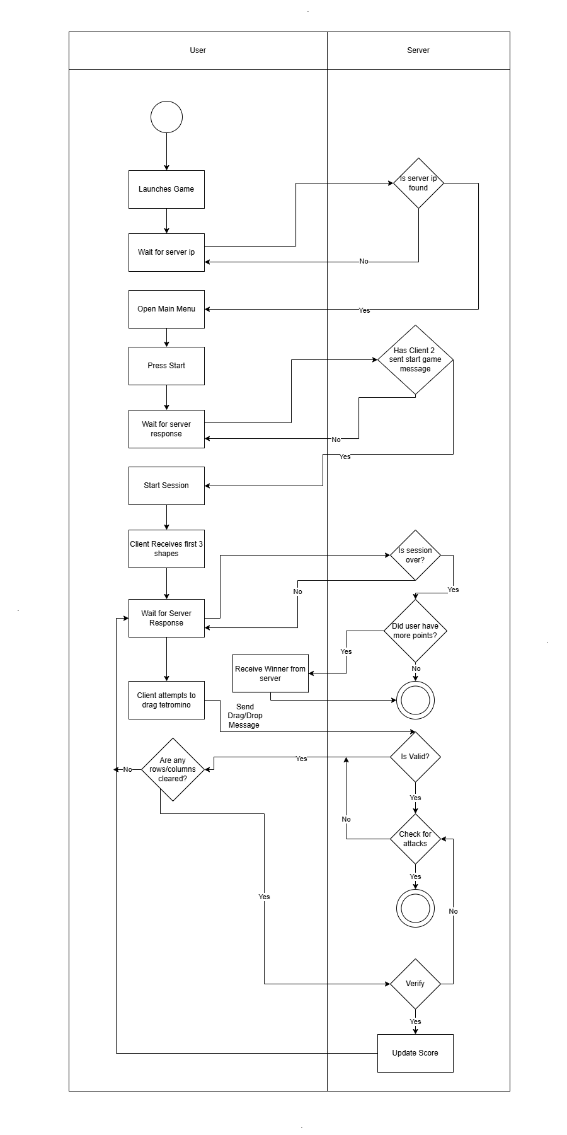


Figure 3 Activity Diagram of the Client-Server Interactions

# Evaluation

## Requirement coverage

The core functional requirements of the multiplayer puzzle game were successfully implemented:

* **Client-Server Architecture**: Achieved real-time synchronization using TCP for critical events and UDP for lightweight state updates.
* **Puzzle Gameplay**: SDL was used to render a responsive 9×9 grid with drag-and-drop support, grid snapping, and row/column clearing mechanics.
* **Multiplayer Matchmaking**: Clients automatically discover servers and are matched into games with synchronized timers and block distribution.
* **Security**: All packets are signed with HMAC and validated server-side. Replay attacks are mitigated via timestamped, unique identifiers.
* **Procedural Generation**: Tetromino sequences are generated fairly using a bag system, ensuring balanced difficulty across players.
* **User Interface**: Menus, scores, and in-game feedback were implemented with intuitive layouts and responsive controls.

## Testing Strategy

**Unit Testing:** Unit testing is a critical component in ensuring the reliability and correctness of a project of this scale. Our focus was on validating the core functionality and logic of the most significant modules within the program. While initial plans involved using *Microsoft Unit Testing* and *Google Test* frameworks, project constraints, specifically related to compatibility and integration issues within our development environment, led us to opt for the lightweight and efficient *Cassert* library. *Cassert* provides the essential assert-based testing capabilities needed to verify functionality with minimal overhead, making it well-suited for our use case. The tests conducted covered key aspects such as game logic, move validation, collision detection, and scoring mechanics. Tests performed were for:

* **UI:** The main menu was tested by iterating over every menu item, simulating the position of the mouse to be within the rectangle of each item. This verifies that the hover-detection logic of isMouseOver correctly flags a point that is strictly inside the rectangle.
* **Puzzle logic:** Three puzzle logic unit tests were conducted for collision, position and grid bounds. Collision testing for tetrominos encompasses a number of scenarios. An empty grid is created and a test tetromino is added, asserting no collision. A tetromino is placed at a specific overlapping grid cell, overlapping another and asserting a collision. The test piece is moved away and asserts no collision. And finally positioning it close but not overlapping the edge, this asserts an edge case collision. verifies that block-collision behaves correctly in both normal and edge case scenarios. The position unit test clears the global tetrominos pool and pushes shapes that do and do not conflict with the spawn coordinates, expecting the corresponding IsPositionFree state. Position testing protects against spawning the four-block coordinate tetrominos into already occupied spaces and grid testing prevents pieces from drifting off from the play area and catches out of bounds errors. Finally, the grid bounds testing constructs tetrominos at various positions. It asserts true for tetrominos inside the grid and false for tetrominos that are both partly and fully outside the grid. Grid boundary testing is essential as it ensures that piece-placement logic correctly rejects illegal moves.
* **Security:** HMAC security testing is essential to ensure that the security integrity of the game is not compromised. Three HMAC testing methods were created for: ComputeHMAC, hmacEquals and validateHMAC. The first verifies that the hash matches OPENSSL’s output exactly, this guarantees that security primitives are correct before building atop them. The second checks for different combinations of tags; where identical tags assert true, tags with a single bit difference assert false and tags with different lengths also assert false. Timing-safe equality is important in order to avoid side-channel leaks. Finally, validateHMAC tests for a missing HMAC tag and returns false leaving data unchanged, a correct tag which attaches a valid HMAC over the payload and asserting true before removing the HMAC tag after verification. And thirdly, it attaches a false 64 character and expects false. The validation tests cover acceptance, rejection and cleanup semantics for the message integrity checks.
* **Multiplayer:** Multiplayer testing verifies pairing logic, client state resetting and broadcast flag halting under real concurrency. The pairing sync test clears shared waitingClients and pairingMap then launched two threads and after both join, it asserts that each saw the ID and socket of the other. The resetClientState test intentionally corrupts every relevant field in the Client namespace and resets all the relevant fields, then asserts that each field is back to its default or empty state. Finally, the stopBroadcast\_flag test creates a thread that spins until stopBroadcast becomes true, then verifies the thread is still running after a brief delay and sets stopBroadcast to true, waits for a join and asserts the thread exited. These tests guarantee that the two clients will reliably sync, and that the atomic shutdown flag will break out of long-running loops.
* **Network:** Testing for packet loss and variable jitter is important that it verifies that player clients can cope with packet loss by retransmitting until delivery and can reassemble out of order deliveries correctly which is crucial under real world circumstances. The variable jitter simulation test clears the networkQueue and sends messages through mockSend with random a random delay per message. It then calls receiveOrdered to wait for all messaged then sorts by sequence number, picks the last payload and asserts that it matches the highest sequence value. The packet loss simulation test clears the network Queue and drops a number of messages with 30% probability, retrying until each message arrives. It finally confirms that the final payload is that number multiplied by 100. Finally, the replay protection test uses makeChannel to connect a client socket to a server socket. It sends a JSON message where the type includes a message ID prefix and calls recvSecure server-side to read the message and returns true with the stripped JSON message and sends the same JSON a second time returning false because the message ID is smaller or equal to the first one. Replay testing is important because it verifies the server-side replay attack prevention and ensures that the lastMessageId is correctly tracked and updated per-client.

## Results

* **Performance:** The game consistently delivered smooth and responsive gameplay, even under extended play sessions and during multiplayer interactions. This was achieved through optimized rendering using SDL, efficient event handling, and targeted memory management. Additionally, thoughtfully integrated audio cues and polished visual feedback contributed to an immersive user experience by reinforcing key actions such as block placement, line clears, and scoring events.
* **Network Efficiency:** The networking model was designed with efficiency in mind, transmitting only critical state changes such as; block placements, row or column clears, and scoring events between clients and the server. This selective data exchange significantly reduces bandwidth usage, ensuring low-latency synchronization without overwhelming the network with redundant or non-essential updates. The use of compact data formats and event-driven communication further contributes to overall network responsiveness.
* **Bug Resolution:** Several key issues were identified and addressed during development. Notably, a bug causing blocks to spawn outside valid grid boundaries was resolved by refining the collision and spawn validation logic. Additionally, scoring discrepancies, particularly those related to combo and chain reaction calculations were traced to edge case handling errors and were corrected to ensure consistent and fair scoring. These fixes have enhanced the game's reliability and competitive integrity.

# Conclusions and future work

The project set out to design and implement a real-time multiplayer puzzle game using SDL and a client-server architecture, with a focus on responsive gameplay, secure networking, and synchronized game state across clients. Through effective teamwork and iterative development under the Scrum framework, the team successfully delivered a functioning prototype that meets most of the initial requirements.

Key achievements include the creation of a drag-and-drop grid-based interface using SDL, a real-time scoring and combo system, and a secure communication protocol based on HMAC validation and timestamped messages. The procedural block generation system ensured fairness by distributing puzzles of equal difficulty to all players. The client-server model allowed for stable synchronization, even under adverse network conditions, while UI feedback and audio cues enhanced the overall player experience.

Throughout development, the team encountered and overcame several challenges. These included managing real-time synchronization under variable latency, implementing reliable server discovery, and ensuring that move validation and replay protection were robust against potential exploits. Debugging concurrent client states and refining the drag-and-drop mechanics required careful coordination and repeated testing.

From a learning perspective, the project provided valuable experience in SDL rendering, socket programming, real-time systems, and multiplayer game design. It also reinforced the importance of agile practices, such as sprint-based development, version control discipline, and clear communication. Overall, the team was able to create a solid foundation for a multiplayer game and gained practical insight into the complexities of building networked interactive systems.

# 9.1 Future Work

The prototype delivers a solid real-time multiplayer puzzle foundation, but several enhancements could broaden its appeal and stability:

* **Gameplay**: Add a single-player mode with AI, introduce power-ups and combo mechanics, and offer adjustable difficulty tiers.
* **User Experience:** Embed an interactive tutorial, polish animations and audio feedback, and include accessibility options (colour-blind palettes, scalable UI, remappable controls).
* **Backend & Networking**: Optimize server performance for higher concurrency, implement matchmaking queues with skill-based rankings, and store persistent profiles and achievements.
* **Quality Assurance:** Establish CI-driven unit, integration, and regression tests; instrument analytics to track player behaviour; and simulate varied network conditions for resilience.
* **Extensibility & Platforms:** Provide a level editor or modding API (potentially open-sourced), and port the client to mobile and web with gamepad and local co-op support.

Implementing these directions will transform the prototype into a robust, scalable platform that meets diverse player needs and fosters a thriving community.

# 10. Acknowledgements

We gratefully acknowledge Dr. Clyde Meli for his expert guidance and unwavering support during the development of our multiplayer puzzle game. His insightful feedback and generous sharing of knowledge significantly informed our design decisions, deepened our technical understanding, and sustained our progress throughout this project. We also appreciate the collaborative environment he fostered, which enabled each team member to contribute effectively and grow professionally.

# 11. References

[1] SDL Wiki Contributors, “SDL3 FrontPage,” SDL Wiki, https://wiki.libsdl.org/SDL3/FrontPage (accessed May 12, 2025).

[2] SDL Wiki Contributors, “SDL3\_image FrontPage,” SDL Wiki, https://wiki.libsdl.org/SDL3\_image/FrontPage (accessed May 12, 2025).

[3] SDL Wiki Contributors, “SDL3\_ttf FrontPage,” SDL Wiki, https://wiki.libsdl.org/SDL3\_ttf/FrontPage (accessed May 12, 2025).

[4] Lazy Foo’ Productions, “SDL Tutorials: Texture Management,” https://lazyfoo.net/tutorials/SDL/06\_hello\_SDL\_rendering/index.php (accessed May 12, 2025).

[5] D. McGuire, “Maintaining Frame Rate Stability in 2D Puzzle Games,” \*GameDev.net\*, Mar. 2018.

[6] J. Lee, “Designing Intuitive Feedback for Puzzle Games,” \*Gamasutra\*, May 2019.

[7] Valve Corporation, “Source Multiplayer Networking,” GDC Vault, 2013.

[8] D. L. Mills, “Network Time Protocol (Version 4): Protocol and Algorithms Specification,” RFC 5905, Jun. 2010.

[9] National Institute of Standards and Technology, \*Guide to Industrial Control Systems (ICS) Security\*, NIST SP 800-82, May 2015.

[10] M. Bellare, R. Canetti, and H. Krawczyk, “Keying Hash Functions for Message Authentication,” in \*Proc. CRYPTO ’96\*, Springer, 1996, pp. 1–15.

[11] “Cassert C++ Unit Testing Library,” https://github.com/biojppm/cassert (accessed May 12, 2025).

[12] GNU Project, “Valgrind: Memory Debugger,” https://valgrind.org (accessed May 12, 2025).

[13] S. Brown and A. Green, “Multiplayer Game Testing: Race Conditions and Network Jitter,” in \*Proc. SIGGRAPH 2014\*, Los Angeles, CA, Jul. 2014.

[14] S. Laroche, “The History of Tetris Randomizers,” Simon Laroche’s Blog, Jul. 2018. [Online]. Available: https://simon.lc/the-history-of-tetris-randomizers. [Accessed: May 14, 2025].

[15] Tetris Wiki Contributors, “Random Generator,” Tetris Wiki, Fandom, 2025. [Online]. Available: https://tetris.fandom.com/wiki/Random\_Generator. [Accessed: May 14, 2025].

[16] National Institute of Standards and Technology, \*Guide to Industrial Control Systems (ICS) Security\*, NIST SP 800-82, May 2015.

[17] “Puyo Puyo Tetris,” Wikipedia, May 2025. [Online]. Available: https://en.wikipedia.org/wiki/Puyo\_Puyo\_Tetris. [Accessed: May 14, 2025].

[18] “Tetris 99,” Wikipedia, May 2025. [Online]. Available: https://en.wikipedia.org/wiki/Tetris\_99. [Accessed: May 14, 2025].

[19] Tripledot Studios, “Woodoku – Wood Block Puzzles,” App Store, Apr. 29, 2025. [Online]. Available: https://apps.apple.com/us/app/woodoku-wood-block-puzzles/id1496354836. [Accessed: May 14, 2025].

[20] Easybrain, “Blockudoku®: Block Puzzle Game,” Google Play, Apr. 13, 2025. [Online]. Available: https://play.google.com/store/apps/details?id=com.easybrain.block.puzzle.games. [Accessed: May 14, 2025].

[21] EA Swiss Sarl, “Tetris Blitz,” Uptodown, Mar. 31, 2020. [Online]. Available: https://tetris-blitz.en.uptodown.com/android. [Accessed: May 14, 2025].

# 12. Appendices (incl. meeting logs)

<https://docs.google.com/document/d/1Mx28xBJ5Viti2HPFURihu1quxHZpk4-zVF7ZUER2Vso/edit?tab=t.q8j002x78yx5> - Meeting Logs

|  |  |
| --- | --- |
| **Group Assigned Practical Task**  CIS2108 | Text  Description automatically generated |

Meeting Log Sheet

The attendance log is to be shared with the supervisor(s) via Google Docs and must be updated on a weekly basis.

|  |  |  |
| --- | --- | --- |
| **GAPT TITLE** | **Multiplayer Puzzle Game Using SDL** |  |
| **SUPERVISOR** | **Clyde Meli** |  |
| **STUDENTS** | **Andrija Dordevic, Gary Micallef, Jake Carabott, Jurgen Cauchi** |  |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date of Meeting** | **Meeting Outcomes** | **Deliverables for next meeting** | **Signatures** |
| 19th February 2025 | ·         Discussed idea for the game  ·         Distributed initial work | ·         Jurgen: Create drag logic    ·         Andrija: Create grid    ·         Gary: Create menu    ·         Jake: Create block spawning | Students (Initials)  J.c  A.D  G.M  J.C |
|  | Supervisor |
| 25th February 2025 | ·         Discussed some issues (Every class had a main method, instead of using on only one)  ·         Distributed more work | ·         Jurgen: Fix bugs with block movement    ·         Andrija: Change grid creation    ·         Gary: Fix menu in the meantime, work on grid snapping    ·         Jake: Fix bugs with block spawning | Students (Initials)    J.c  A.D  G.M  J.C |
|  | Supervisor |
| 11th March 2025 | ·         Progress meeting to check on work done  ·         Discussion on possible features of the game | ·         Andrija: Separate concerns    ·         Gary: Fix problems with the menu | Students (Initials)  J.c  A.D  G.M  J.C |
|  | Supervisor |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date of Meeting** | **Meeting Outcomes** | **Deliverables for next meeting** | **Signatures** |
| 12th March 2025 | * Confirmed on viability of game idea * Discussed on block placement * Distributed More Work | ·         Jurgen: Work on grid snapping    ·         Andrija: Change grid creation    ·         Gary: Fix menu in the meantime  ·         Jake: Snap back to spawn position if grid is occupied | Students (Initials)  J.C  A.D  G.M  J.c |
|  | Supervisor |
| 17th March 2025 | ·         Discussed with Gary about issues with the menu  ·         Distributed more work | ·         Jurgen: Fix out of bounds issue    ·         Andrija: Clearing of blocks, Layering    ·         Gary: Fix menu    ·         Jake: Fix overlapping issues | Students (Initials)  J.c  A.D  G.M  J.C |
|  | Supervisor |
| 24th March 2025 | ·         Discussed ideas for the Ui of the game  ·         Discussed minor issues with the screens | ·         Jurgen: Create Ui Assets  ·         Andrija: Fix Screen resizable issue  ·         Gary: Fix Menu resizable issue  ·         Jake: Look into server-client connection | Students (Initials)  J.C  A.D  G.M  J.C |
|  | Supervisor |
| 26th March 2025 | ·         Discussed to add a timer/ score text  ·         Decided to add a clear grid button | ·         Jurgen: Create more assets  ·         Andrija: Set-up a small server  ·         Gary: Create the Clear Grid Button  ·         Jake: Create the timer/score texts | Students (Initials)  J.C  A.D  G.M  J.C |
|  | Supervisor |

|  |  |  |  |
| --- | --- | --- | --- |
| **Date of Meeting** | **Meeting Outcomes** | **Deliverables for next meeting** | **Signatures** |
| 8th April 2025 | ·         Discussed a better way for the server to find the ip  ·         Discussed minor bugs with the timer | ·         Jurgen: Separate the enemy/Player scores  ·         Andrija: Make the server broadcast the server ip  ·         Gary: Fix issues within the menu  ·         Jake: Allow the server to verify the score changes | Students (Initials)  J.C  A.D  G.M  J.C |
|  | Supervisor |
| 16th April 2025 | Discussed problems with replaying  Discussed a problem with the assets after replaying | Jurgen: Fix bug with assets after replaying  Andrija: Fix bug with replaying  Gary: Fix bug with the score buffer  Jake: Add screen shake effect | Students (Initials)  J.C  A.D  G.M  JC |
|  | Supervisor |
| 15th May 2025 | Discussed Report |  | Students (Initials)  J.C  A.D  G.M  JC |