SE Lab Project: Implementation and Testing Report

Team Members:

1. IMT2023026 Siddharth Kini
2. IMT2023010 Priyanshu Tiwari
3. IMT2023082 Gautam Kappagal

Github Link: https://github.com/d3monviking/multiplayer-platformer-se
README Link: https://github.com/d3monviking/multiplayer-platformer-se/blob/main/README.md

1. Introduction

multiplayer-platformer is a 2D multiplayer game built with a client–server setup. The goal is to let multiple players join the same world over a network, interact with shared objects, and stay synced in real time. The system splits everything cleanly: the client handles rendering and player input, while the server controls the game state and keeps everyone consistent. This setup ensures smooth gameplay and prevents desyncs or cheating.

This report explains the testing approach, tools, and workflow used for the Multiplayer Platformer project. The goal of our testing setup is to make sure both the server and client behave correctly, stay stable, and work reliably. Since this is a real-time multiplayer game, we rely on a mix of unit tests and integration tests: unit tests check small pieces of logic on their own, and integration tests make sure all the parts work properly together. We also give a brief overview of the different components that are a part of our project. Detailed instructions on how to setup and run the project along with the server and client testcases can be found in the README linked above.

2. Server Tests

The server, written in Java, is the authoritative source of the game state, so it goes through thorough testing to confirm that its logic, state handling, and networking are all working as expected.

Frameworks and Tools Used

- **JUnit 5** : the main testing framework.
- **Apache Maven** : handles dependencies and controls the build + test lifecycle.

Types of Tests

- **Unit Tests**
  These are quick, focused tests that check small pieces of logic in complete isolation. They cover things like player state changes, physics calculations (like gravity), and collision detection. They don't require a running server or any network setup. These files end with `*Test.java`.

- **Integration Tests**
  These tests are broader and check how multiple components work together. They test flows like the main server loop, how multiple clients connect and interact, and how the server receives input and sends out state updates. These tests end with `*IT.java` and are run during Maven's `verify` phase.

3. Client-Side Testing (C++)

The C++ client handles rendering, input, and player-side game logic, so it has its own testing setup to make sure everything works correctly and communicates properly with the server.

Frameworks and Tools Used

- **Google Test (GTest)** : the main C++ testing framework. CMake automatically downloads it via `FetchContent` during the build.
- **CMake** : manages the whole build, sets up dependencies like SFML and Boost, and creates the test executables.

Types of Tests

- **Unit Tests**
  These tests give fast feedback on the correctness of smaller client-side components. They check things like entity properties, player movement logic, and level/terrain structures. They run without touching rendering or networking.
- **Integration Tests**
  These tests look at how different client systems behave together. They're meant to catch issues in more complex features, like the client's physics simulation or the networking layer that sends input to the server and receives updates back.

4. Overview of Project Components

1. Client Module (C++)

The client handles everything the player sees and does. Its main jobs are:

- Rendering sprites, tiles, and world elements on the screen
- Capturing keyboard input
- Sending that input to the server at regular intervals
- Receiving updated player/object positions from the server and showing them visually

Basically, the client is the user-facing part of the game.

2. Server Module (Java)

The server is the authoritative part of the game and keeps track of the global state. It's responsible for:

- Handling incoming player connections
- Keeping track of player positions, actions, and world interactions
- Running game logic like movement, collisions, and updates
- Sending the updated world state to every connected client

This ensures that all players see the same consistent world and prevents cheating.

3. Networking Layer

The client and server talk using network sockets. The data flow looks like:

- **Client → Server:** input commands (move, jump, etc.)
- **Server → Client:** updated positions, animations, events

This networking layer keeps all players synced and makes the game feel responsive.

4. Game Assets

The project includes various asset files, such as:

- Character sprites
- Tile-based maps
- Backgrounds and world textures

These define how the game looks and how levels are laid out, making it easy to reuse or expand the world.

5. Game Logic

The core gameplay rules include:

- Player movement and physics (gravity, speed, jumping)
- Collision detection with platforms and boundaries
- Keeping multiple players synchronized across the network

While the client shows immediate movement to keep things smooth, the server confirms the real, authoritative state to maintain fairness.

Overall System Workflow

1. The server starts up and waits for connections.
2. Clients start the game and connect to the server’s IP.
3. Players press keys, and the client sends this input to the server.
4. The server updates the world state and broadcasts it back.
5. Clients render the final synchronized frame on screen.

This loop repeats continuously, providing real-time multiplayer gameplay.