Bomberman - Final Report

Group U

Ryan Seys - 100817604
Martin Gingras - 100831823
Cassandra Perez - 100859183
Fady Ibrahim - 100876906

March 31, 2014

Table of Contents

1. Introduction....................3

2. Design Documentation / System Architecture & Behaviour....................3
    2.1 Overview Description....................3
    2.2. Communication Protocol....................4
        2.2.1 Purpose....................5
        2.2.2 Format....................5
        2.2.3 Constraint of Message Ordering....................6

3. System Design....................7
    3.1 Game Server Design....................8
    3.2 Spectator Client & Player Client Design....................9
    3.3 Key Design Choices....................10
    3.4 Key Concurrency Issues....................11
    3.5 Changes from Milestone 1 to 2....................11

4. Testing Strategy....................12
    4.1 Purpose....................12
    4.2 Overview....................12
    4.3 Functional Tests....................13
    4.4 Concurrency Tests....................13
    4.5 Performance/Scalability Tests....................14

5. Conclusion....................14

1. Introduction

Our group was comprised of 4 group members, named on the title page. Each member contributed to the success of the project in different ways. Martin contributed to the Server, Cassandra and Fady contributed to Java Client and design documentation (Class and Sequence diagrams), and Ryan contributed to the Testing Framework and NodeJS/WebKit Client. All members of the group contributed to the overall documentation of the project, and actively smashed bugs.

2. Design Documentation / System Architecture & Behaviour

The following will describe the overall architecture of the project and discuss the messaging protocol used in the system as well as reasoning behind some of these design decisions.

2.1 Overview Description

Our Bomberman project was designed into a collection of projects that together made up the entire system. Each of the projects had their own roles and responsibilities which are listed beside their name. These 4 projects were:

- UDP Server - Manages game state and all game logic, broadcasting messages to all the clients on an interval. Maintains a double buffer state of the game. Accepts control messages from the clients in the JSON format. See *2.2 Communication Protocol* for more information.
- Java Client - Spectator & Player client built into one GUI. Displays game state, takes keyboard controls to send game commands to the server once a connection has been initially established and a game has been started. All messages are sent in the JSON message format. See *2.2 Communication Protocol* for more information.
- Test Framework - Tests the functionality of the Java Client and Server together. The framework covers many different types of tests including functional tests, concurrency tests and scalability and performance tests. See *4. Testing Strategy* for more information.
- NodeJS Client - Fully-functioning client written with HTML, JavaScript and CSS, rendered in a WebKit browser/NodeJS hybrid system called node-webkit. Accomplishes all of the functionality that the Java Client accomplishes using standard NodeJS libraries and web technologies.

Other than the Test Framework, each other project can function completely separately from each other. This allows us to easily package the clients and server separately for painless package distribution.

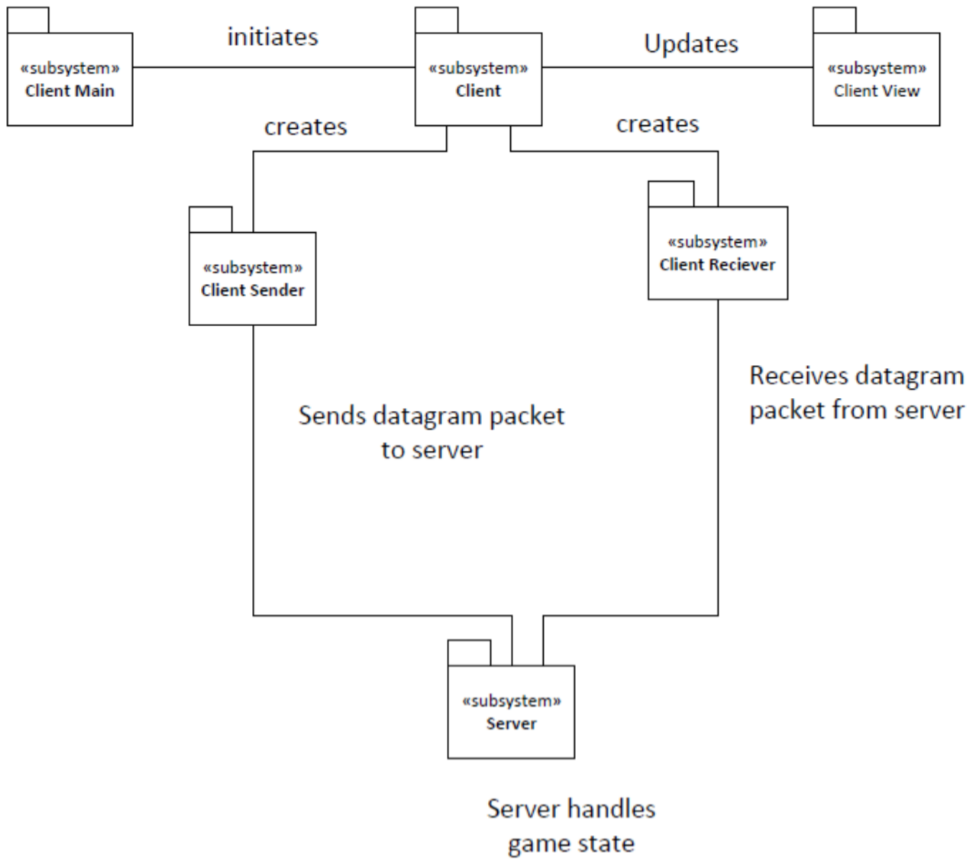

initiates
Updates
«subsystem» Client Main
«subsystem» Client
«subsystem» Client View
creates
creates
«subsystem» Client Sender
«subsystem» Client Reciever
Sends datagram packet to server
Receives datagram packet from server
«subsystem» Server
Server handles game state

Figure 1: Subsystem Diagram - A high level overview of the initial system design.

2.2. Communication Protocol

We chose to use JSON to encode our messages that would be sent to and from the server and clients. We used this protocol on the Java Server, Java Client and NodeJS Client. We required an additional JSON library to support JSON parsing in Java, but JavaScript has it built in, so incorporating it into our NodeJS Client was much easier.

2.2.1 Purpose

The reason for having a messaging protocol that is strictly agreed upon makes it easier to extend the functionality of the game, and create more clients that can communicate with the server as long as the messaging protocol is adhered to. This allows us to open our system to extensibility. The purpose of choosing JSON for our implementation is that we had experience using the messaging format coming from a web development background and thought that we could use this to create additional clients easily.

2.2.2 Format

For our system, as mentioned previously, the format of messages that we used were JSON strings. These are strings formatted in the JavaScript Object Notation. Coming from a JavaScript background, and thinking ahead that we wanted to create a client using JavaScript (which we did do), we decided to implement our messaging format as JSON to allow us to easily extend the client to JavaScript. Basically the JSON format is an object of key-value pairs. The keys are simple strings and the values can be objects, arrays, strings or numbers.

A simple connect message in JSON looks like this:

```
{"command":"join", "type":"player"}
```

You can “stringify” JSON objects allowing them to be easily transported over the UDP messaging protocol, used in our project.

You can also then “parse” the JSON strings back into JSON objects for easy object and attribute traversal/retrieval.

The game board broadcast is even sent using this messaging protocol, and the format allows for very easy interpretation of the messages to humans (no byte-level messages here).

Here's an example of the game board state being broadcast to a player:

```
{"game":{"height":10,"width":10,
 "board":[[0,8,0,0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0,0,0,0],
          [0,7,0,0,0,0,0,0,0,0],
          [0,0,0,8,0,0,0,0,0,0],
          [8,0,0,0,0,0,0,0,0,0],
          [0,0,0,0,0,0,0,8,8,0],
          [0,9,0,0,0,0,0,0,0,0],
          [0,0,1,0,0,0,0,0,0,0]]},"type":"broadcast"}
```

A brief overview of the messages that can be sent include:

Start the game:

```
{"command":"button", "pid": 1, "button":"start"}
```

Move a player up:

```
{"command":"move", "pid": 1, "direction":"up"}
```

Deploy a bomb:

```
{"command":"deploy", "pid": 1}
```

2.2.3 Constraint of Message Ordering

There is an inherent order in which the messages should be sent to the server. That is, you should not send a move message to the server before you have joined the server and started the game. We have implemented error checking into the logic of the server so that a client may send messages at any time and have it processed by the server, but depending on the current state of the server, the message may or may not be handled appropriately.

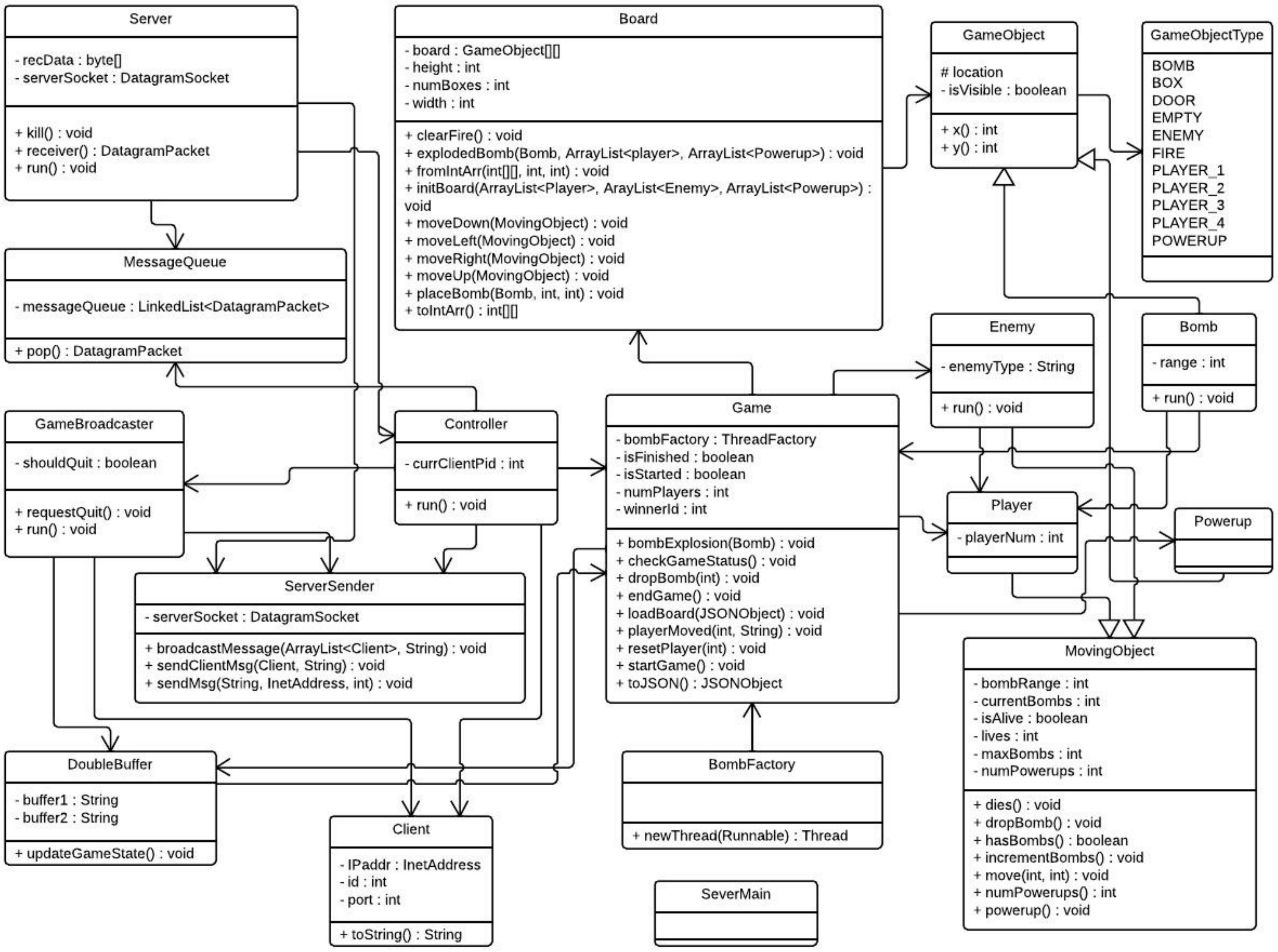
In the case of sending a move message to the server without previously connecting to the server, the server will not respond to your request (i.e. you will not receive a message back from the server) but the server will log a message saying that the command has been processed and ignored.

A similar process will occur to every move command received after the game has finished. This ensures proper concurrency states and scenarios are met. If a player spams the keyboard, sending 4 up moves to the server, but after the second up move, the player hits an enemy and dies, the 2 remaining up commands in the message queue will be processed like any other message but they will be logged and ignored due to the game over state that the server is in.

3. System Design

The following will discuss the client and server design as well as key design choices made and some issues we encountered along the way related to our design and how we fixed these issues.

3.1 Game Server Design



Our Server is has many moving parts, from a Controller thread that handles all game logic changes from incoming messages, to the MessageQueue which holds messages that come from the clients. There's a Broadcaster which broadcasts the game state on an interval using the DoubleBuffer to hold the game state in one buffer and update the game state in the other buffer, before switching. All of the GameObjects are classes, including Door, Player, Enemy, Powerup, which are stored on the Board in the Game object. Bombs are threads because they have a fixed life-time before they explode, and they are created using the BombFactory.

All of the game logic is handled by the Server. The state of the server, once running, is only modified by messages that it receives on the port that it listens to using the UDP protocol. It broadcasts all game state changes to the clients which properly connect and register on the server.

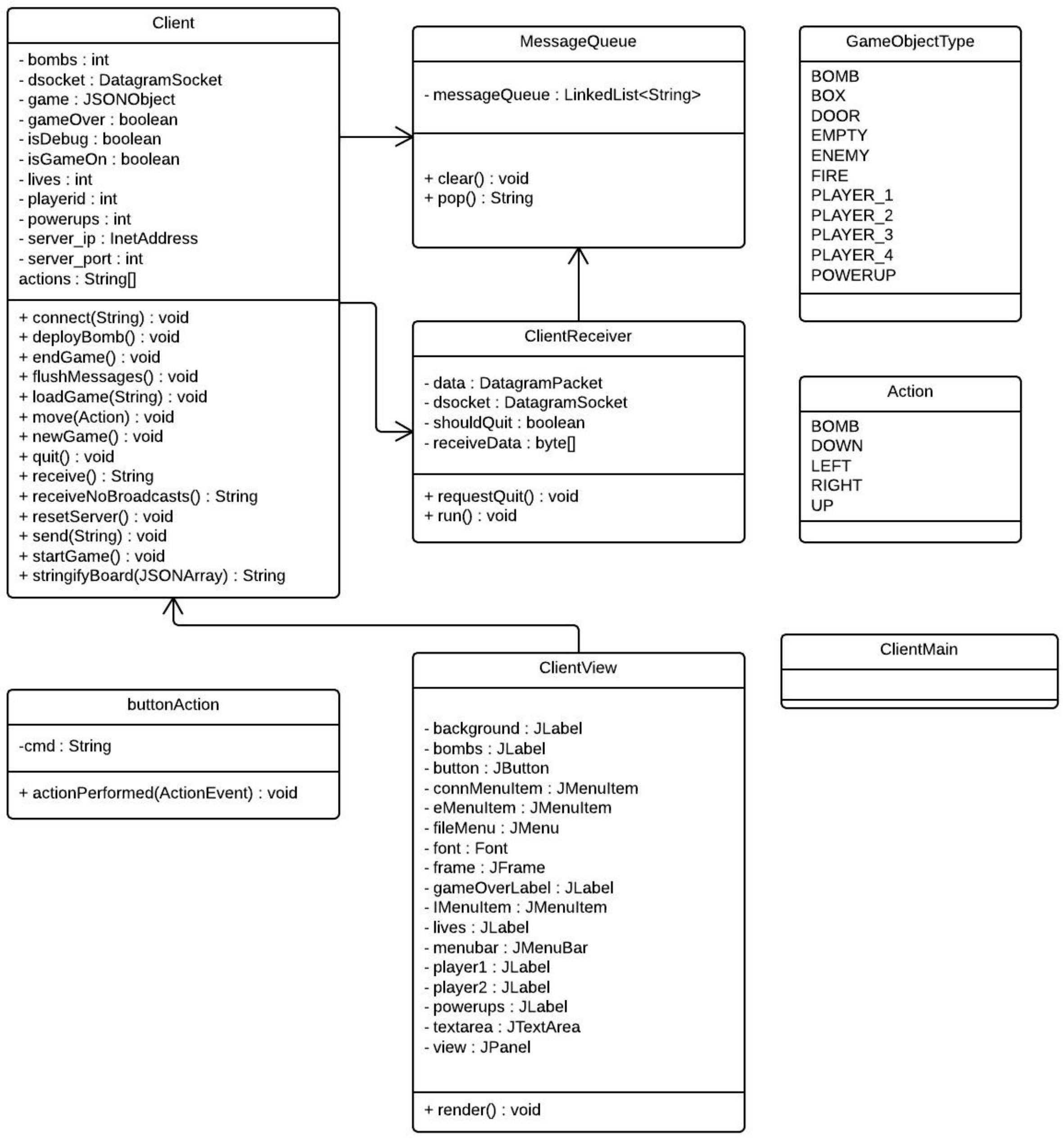
3.2 Spectator Client & Player Client Design

Our Spectator and Player client are implemented in the same project called Client. We believed that implementing both into the same system would eliminate redundancies and streamline our development process throughout the progression of the milestones. Our Client is built using Java, and it receives messages on the same socket that it uses to send messages to the Server. This allows the Server and Client to communicate back and forth regarding the state of the game.

Our Client uses a MessageQueue much like the server does for handling received messages. This way it can process the received messages in a timely manner rather than relying on the UDP protocol to queue all the messages for it. This way, we can be sure that every message we receive will be immediately added to a message queue, ready for processing.

Our client is said to be very “dumb” in the sense that other than knowledge of the message protocol and how to render the game state and what messages to send, the client knows nothing more about how the game is implemented. This allows the client to be completely agnostic toward the server, and vice-versa. The client simply receives messages, parses them as JSON, follows a few simple steps to figure out what kind of message it is (e.g. broadcast, game over, player join) and then render its view according to that new state. The Client also listens for keystrokes from the user and binds these keys to specific commands (e.g. move up, move down, drop bomb), which are simple strings, to be sent to the server.

The class diagram for the client can be seen on the next page.



3.3 Key Design Choices

One of the key design choices that we think makes our system more robust is our decision to use JSON as the messaging protocol. Rather than reinvent the wheel or possibly create a messaging protocol that is unreadable at a glance or possibly incomplete or inconsistent, we decided to go with a tried and true messaging format of JSON. This design decision made things very easy to get working once we figured out how to use the Java library for JSON objects. This decision also allowed us to implement an entire JavaScript client in under a day, based on the fact that JavaScript has JSON support built directly into the language. Using JSON has saved us a lot of time and potential headaches and we believe it is a great design decision.

Another design decision that we made that has made our system stand out is our decision to separate the entire project into 4 separate projects. As mentioned eariler, this makes our system incredibly easy to package into separate pieces, meaning we can distribute the clients while maintaining complete control of the server code. This means that there are no dependencies between the server and client code, other than through the agreed upon message format mentioned earlier.

3.4 Key Concurrency Issues

At the beginning we experienced concurrency issues around the game state, but once we implemented the double buffer, those issued went away. Following that, when we first implemented bombs as individual threads, there were numerous concurrency issues around updating the game state and swapping the double buffers. At first our system required our player to move before the updates would show in the game state. Once we implemented the broadcaster, our player no longer had to move around the board to receive updates about the game state. We actually had an unneccesary safe-guard implemented at first for sending messages from the client socket. We found that this outgoing message queue was not neccesary and that the socket would always handle the send message appropriately without any concurrency issues. We removed this safe guard and have seen no regressions since.

There are also no known concurrency issues in the system right now.

3.5 Changes from Milestone 1 to 2

There were numerous changes made to the design of the system between Milestone 1 and 2. In Milestone 2, we added a Double Buffer to store the game state in two buffers and allow the game state to be updated in one buffer while being read from the other buffer.

We also added a broadcaster to the Server, creating more overhead for the Server but more reliability for the system overall. Now if a game state packet is lost, the clients will receive the game state again in 100ms.

We also moved the entire Test Framework out of the Client and into its own project conveniently called TestFramework. This allowed us to observe both the client and server state during tests. It also allowed us to use JUnit tests more freely to test the entire system as a whole.

4. Testing Strategy

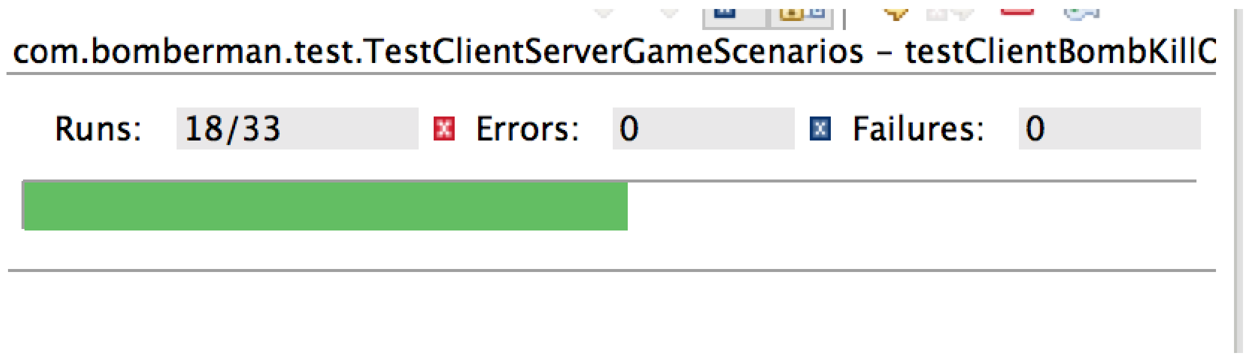
Testing is fundamental to our entire Bomberman system. It ensures a quicker development process and smooth running game.

4.1 Purpose

The purpose of our testing framework was to test the core functionality and some edge cases of the Java Server and Java Client. The reason we want to do this is so that we can iterate more quickly on the implementation of the client and server while being confident that it remains in a functional state. The testing framework allows us to undertake TDD, test-driven development.

4.2 Overview

Overall our testing framework was created on top of the JUnit testing framework available in Eclipse. This allows us to take advantage of the robust capabilities of JUnit, such as SetUp methods, Teardown methods, and quickly allow us to see if all tests pass, and if not, run all failed test with one extra click. Using JUnit really streamlined our workflow and allowed us to accomplish more in less time. Below is a screenshot of the Test Framework in action, using JUnit to provide us with instant feedback.


com.bomberman.test.TestClientServerGameScenarios – testClientBombKillC

Runs: 18/33 Errors: 0 Failures: 0

Figure 2: Test Framework uses JUnit to run tests, showing an overview of test pass/fail rate.

We broke our tests down into three (3) main categories:

- Functional tests
- Concurrency tests
- Performance/Scalability tests

4.3 Functional Tests

Functionality tests are used as sanity tests, making sure game state is updated properly in the base and edge cases. All game logic related to the Bomberman game itself is tested through the functional tests. The following is a list of the *some* of the currently implemented functional tests (this is not a complete list):

- ***testConnectTwoPlayers()*** - Connect two players. Ensure connection success.
- ***testClientLoadBoard()*** - Load a game from file and test that load succeeded.
- ***testClientMoveUp()*** - Send the command for move up, check board state to ensure that player moved up in the game.
- ***testClientExitDoor()*** - Exit the door in game and ensure that game is finished.
- ***testClientMoveDownIntoBox()*** - Move down into box and ensure that you don't move.
- ***testClientPickUpPowerup()*** - Ensure that when you pick up powerup, you gain a powerup as a player
- ***testClientTwoPlayersColliding()*** - Ensure that two players colliding ends the game.

There is obviously a great number of tests to cover many more actions and scenarios in the game. Collectively these tests ensure that overall bomberman is functioning as it should. The nature of these tests are to test larger pieces of functionality so they end up touching a significant amount of code when they execute.

4.4 Concurrency Tests

The concurrency tests were created to test the lower-level parts of the server and client, specifically related to the intercommunication between specific parts, the use of certain data structures (e.g. DoubleBuffer) and the synchronization between these data structures to ensure data protection at a fine-grain level of detail.

One such low-level test that tests that the double buffer is functioning properly is ***testGetBufferStateTwice()*** - Check the double buffer state. This ensures that the double buffer maintains a proper state of the system at all times, regardless of which buffer is being used.

We also implemented tests that ensure certain timing is taken into consideration for some events. For example, bombs do not explode immediately, they have a timeout, so we needed to test that we can concurrently count down a bomb and move players so that they can avoid it's blast. One such test that accomplishes this is ***testClientPlayerTwoBombAvoid()*** - Deploy bomb in range of killing player, run away and ensure that both players are safe after bomb deploys.

4.5 Performance/Scalability Tests

The performance and scalability tests are more subjective tests about how performant we want the system to be. Scalability tests in our system are easy to implement but difficult to interpret. Because we have a broadcaster that is limited to broadcasting the game state at a certain interval, we are bottlenecked by that system in receiving up-to-date game states in a timely manner. In our current implementation we have the server broadcasting the game state every 100 milliseconds, which is fast enough that a human cannot tell the difference and react fast enough, but when we are scaling the system and throw a lot of messages at it, that 100ms really starts to show.

One such test we implemented is ***testScalabilityMove1000()* -** Connect to a game and immediately send 1000 up + down commands to the server (2000 commands in total). After waiting 1 second, the server should be caught up processing all the commands, so get the next game state and check it against your initial game state (it shouldn't have changed). This test has a 1 second wait period required to process all the server commands. Unfortunately the server isn't responsive enough to process them all immediately (perhaps this is also a limit on the network), however now with this test in-place, we can be sure that if the test starts to fail, our server performance has declined, and likewise, if the test continues to pass our system performance is still standing strong.

5. Conclusion

Overall, this bomberman project encapsulates many different aspects of concurrency and synchronization problems in a real time system. Using UDP messaging, we implemented a client-server architecture that incorporates the MVC design pattern, all while handling synchronization across many objects including a message queue, and double buffer game state. Our testing framework tests all aspects of the system, including functionality, concurrency and scalability, ensuring that the entire system is running smoothly to provide you with a great Bomberman experience.