SETUP INSTRUCTIONS

IDE Used: Eclipse

Setup Instruction

Please import the following Java source files to a new project in Eclipse.
In order to run the program run the source files “B_TestDriver” and “BombermanServerMain”. The “B_TestDriver” is the client and the “BombermanServerMain” is the server part. Test cases have been included in either of client and network parts of the program.

Java Source Files:

'package sysc3303.networking':

ClientNetworkHandler.java

DoubleBuffer.java

NetworkHandler.java

Sendable.java

ServerNetworkHandler.java

Subscriber.java

‘package sysc3303.project.BomermanNetworkGame’:

B_ClientMain.java

B_ClientNetworkHandler.java

B_NetworkPacket.java

B_NetworkPlayerController.java

B_ServerMain.java

B_ServerNetworkHandler.java

B_TestDriver.java

B_TestServerNetwork.java

U_BombData.java

U_ClientData.java

U_PlayerData.java

U_WorldData.java

'package sysc3303.project.BombermanGame':

B_Controller.java

B_Packet.java

B_Player.java

B_PlayerController.java

B_View.java

Bomb.java

Entity.java

GridGenerator.java

GridObject.java

PlayerCommand.java

PlayerCommandType.java

PlayerName.java

Point.java

Powerup.java

Utils.java

ViewRenderer.java

World.java

WorldActionOutcome.java

General Description:

- This program uses UDP method of communication.
- Every client send an array of player, command and objects (list of changes to make) to the server. The server goes through changes and applies to world and send a new char[][] (world grid) to all clients that are subscribers.
- Communication protocol: "BombermanServerNetworkHandler" is how the program communicate changes.
- The main threading solution runs inside the "BombermanServerMain". It creates a logger and creates the thread and joins them inside of it.

Description of Types and their Methods:

Network Handler<Send, Receive>

Receive:
First the server tries to receive from a socket (this process loops forever) and once it receives the data, it stores it in the receive write buffer.

Receive Thread Method:
This method calls preprocessed packet, receives from socket and locks on receive write buffer and writes the data to buffer. This method is synchronized.

Sender Thread Method:
This Method grabs send read buffer and synchronizes on the buffer and parses to turn the sent object into byte array.

Send Data Method:
This method takes an array list of data type send and synchronizes on send write buffer. It loops through data and adds it to the buffer. At the end it swaps the send buffer and takes data from outside class and adds to send write buffer.

Get Data Method:
This is a public method which is called from outside and locks on receive buffer and returns the data. At the end it swaps buffer list.

Bomberman Server Network Handler:

This type adds everything that server handler didn't add. Type "S" as world and "R" as player command. It also implements all abstract methods.

Bomberman Client Network Handler:

This type extends client network handler by typing "S" for PlayerCommandArray and "R" for char[]][]. This type also implements abstract methods.

Entity
This is the base class for a component with a dynamic position within the world. It has a name, a character and X and Y positions.

Diagrams:

- UML Diagram.pdf
- Diagram.pdf
- Sequence Diagram.pdf

Extensions:

No additional features has been added to the game (No one can learn how to program for Android in two weeks when they have a life) but from an engineering standpoint, it can be said that server is programmed in an abstract way that it handles multiple inputs and creates logs.

Test Framework:

In order to test the project that was built, the following tests were provided in order to examine the written code.
Whole tests were built on the "junit.Assert" libraries. These libraries were utility classes that assist in validating arguments. This library is useful for identifying programmer errors early and clearly at runtime. Couple of certain methods were used inside each of the tests which are as following:

1- AssertEquals() : This method asserts that two object are equal.
2- AssertNull(): This method asserts that an object is null.
3- AssertArrayEquals(): Asserts that two arrays are the same and if not an AssertionError is thrown.
4- Fail(): It fails a test with no message. Inserted intentionally.
5- AssertSame(): Asserts that two objects refer to the same object.

For each type certain tests were provided in order to examine the written code:

Server concurrency:

1- Network initialization:
In order to test if the network has been initialized, first we use the methods inside that certain class in order to see whether the network has been initialized. Also we compare the received data with the data that has been gathered from network. This allows us to see first if the data has been sent and two, if they have been fully received (Packets have been dropped or not).

2- Run Test:
This type of test allows us to run multiple threads in order to see if the server part of the program works when under answering to multiple requests. First we should check whether commands have been received or if the server is waiting for them to arrive. Second, we make sure that the server is accepting requests by not shutting down. Third, it accepts players as well as accepting the commands that each player is sending. Fourth allows us to check whether the position of the players on the map is the same with the movement that they have made. Finally we check whether the received data is the same as the grabbed data from the network. It also checks whether the server is capable of actually updating and stopping the network. The whole purpose of this test was to see how the server handles multiple threads. If it

is one, then it just gets executed after being read from the buffer. If there is more than one, then the order of the execution will be First In First Out (FIFO). Handling of the concurrency of the server will be based on which package has priority over the other ones.

NetworkHandler:

1- Sending/Receiving:
In order to see if the network part of our program is starting, first we have to check whether our sending and receiving threads are up and running and second we have to check that they are being sent and receive continuously. In other methods we check for packets of data to being parsley sent and received and they should be checked whether they match the returning data. In the last step we check for sending and receiving methods and whether the packets are not null and if the preprocessed Packets match the type that are being needed.

DoubleBuffer:

1- Swapping Buffer:
Since the reading and writing buffers are being read in order (First they write into the buffer, then they swap and the system reads the buffer). In order to do this part we had to make sure that the assertion for equality of buffers fail since they should not be the same thing. If this test fails it shows that the swapping actually works between two buffers (and when one gets read it gets flushed so that it would be available to store data again) and it does the reading and writing in order on different buffers. Synchronization is being used in the code to not allow the corruption of data while one is being read and the other gets written to. This issue also arises from the fact that ArrayDeques can grow extensively and don't have any size limits.

2- Testing reading and writing on buffers:

In order to test reading and writing on buffers we have to check whether buffers (both reading and writing) actually match the data that is being returned. By concatenating the read and write in an array in order and comparing this final array with the return value of our data we can conclude

whether the sent and received data are the same as the data that is being processed. Also by checking the sent data and that the data that is being written (and vice versa, by checking the receiving data and the data that is being sent) we can see that whether they are equal and conclude the fact that buffers work perfectly.

Attempts to break the game down:

According to the third milestone, test cases should be made purposefully so that they break the game down in order to take care of the bugs.

System under load:

1- Packet loss:
In order to find if the packets are lost and if it matters, this implementation on system was done:
If too many packets are all sent at once from all players, then the packet loss is bound to happen at some point. This could affect the total outcome of the game. For example, if two players drop bombs near one another and one of them gets lost, the lost bomb will not detonate so that the detonated bomb might kill one player. On some other level a move may not be recorded at all since the PLAYERCOMMAND is not sent. Overall it all depends which of the packets get lost. If the server gets blasted with 3000 packets from different machines (rather than local host) it might lose 10-20% of the data. The code that was written for this test goes through all the sent and received bytes and checks if they all match. This test was also done in the actual code of the game "ClienNetworkHandler" which gets a copy of the sent and received data for later comparison. But in the written testing code, the packets loss code has two sides. One side is server and one side is the client. The client tries to send different number of packets under different conditions (different test cases) and the server receives them. Depending on the percentage number of lost packets it outputs a number to showcase this:
   - Light usage: System performs normally with minimum number of packet loss and maximum speed. Every update on the world happens in real time.
   - Normal usage: System performs normally when all players are joined and play in normal manner, as expected. Package drops could happen (like 1%) due to network latency that is out of control of the users if they're playing on remote machines.

- Heavy usage: If there are too many number of spectators and too many packages are being sent (for example all players start trolling) couple of things happen: Spectators start to see players jump around instead of moving smoothly over a course of time and player packages might be dropped. For example in one instance (packages are numbered and if one of them is dropped the server moves on, eg. From 5 to 7 if there is no 6) a bomb may not go off but it was dropped by the player. Also, if the players are playing on different machines number of dropped packages could go up which results in angry players and them leaving the game. Local host still has the advantage of latency which hugely helps him becoming a winner since he's not delayed.

2- Same machine vs own machine:
As it was discussed earlier, if all players are being deployed on the same machine the issue of lag becomes obsolete. Otherwise on different machines, this issue will show itself. Due to network latency, the local always has the advantage of not losing packets or getting disconnected. What happens when one player repeatedly sends out bunch of moves is that the server receives all the moves (which are labeled so the importance of receiving them in order is out of the question) and it processes each move in orderly fashion. Depending on the network latency there could be lag. On a busy day at lab, the latency issue shows itself since there are a lot of sent packets and some of them even get lost (Especially on Thursday, March 27th). On the other days, the issue of lag was not severe and did not change the outcome of the game. Time latencies are as follows:
Min: 15 ms.
Max: 1000 ms.
Ave: 100 ms.
These tests were done on different machines (lab and owned laptops). Average time might vary on different situations.

3- Duplicated commands:
The server code is written in a way that it automatically disregard the duplicated commands in order to prevent trolls from cheating in the game. For example if one person just clicks on the dropping bomb button and holds it, the server only captures one of these commands instead of all since only one bomb can be dropped at one place.

4- Game initialization testing:
This code is written to test if all elements of the game work properly. First the network gets initialized by adding one spectator and then the server gets the address and port of this spectator. Next, the world gets initialized by generating a grid and then the spectator becomes a player in this world. Next, since the player has joined the game, the controller object gets initialized and the players will be added to the game. Since then players will be updated when every move happens and the world gets updated accordingly.

Performance:

Low Load vs High Load and Local vs Remote:

Both of these tests also depend on one another. For example, if low load test is done on the local machine the game runs perfectly with minimum delay. But if the high load is being run on remote machines then number of dropped packages will go up and the latency will take its toll on the game. Overall there are 4 different cases in which test cases are being shown:

Low Load on Local:

- Min Latency: 15 ms
- Max Latency: 50 ms
- Ave Latency: ~30 ms

High Load on Local:

- Min Latency: 50 ms
- Max Latency : 100 ms
- Ave Latency: ~75ms

Low Load on Remote:

- Min Latency: 300 ms
- Max Latency: 700 ms
- Ave Latency: ~450 ms

High Load on Remote:

- Min Latency: 700 ms
- Max Latency: 1500 ms
- Ave Latency: ~900 ms