ID1217 Concurrent Programming

Project - EarlyTiger

Simon Ström - simstr@kth.se
Charlie Lindviken - lindvi@kth.se

Instuctions

To run the game you have to:

1. Start a server
    a. Open a terminal and navigate to the /bin/ folder of the project
    b. Run the command: java pps/et/server/Server
    The server then creates a server socket listening on port 4711

2. Start any numbers of clients
    a. Open a terminal and navigate to the /bin/ folder of the project
    b. Run the command: java pps/et/client/Client <HOST> <PORT> <YOUR_NICK>
    Note: *You have to set that java can access your network. Else you won't be able to connect to the server.*
3. Bombs away!

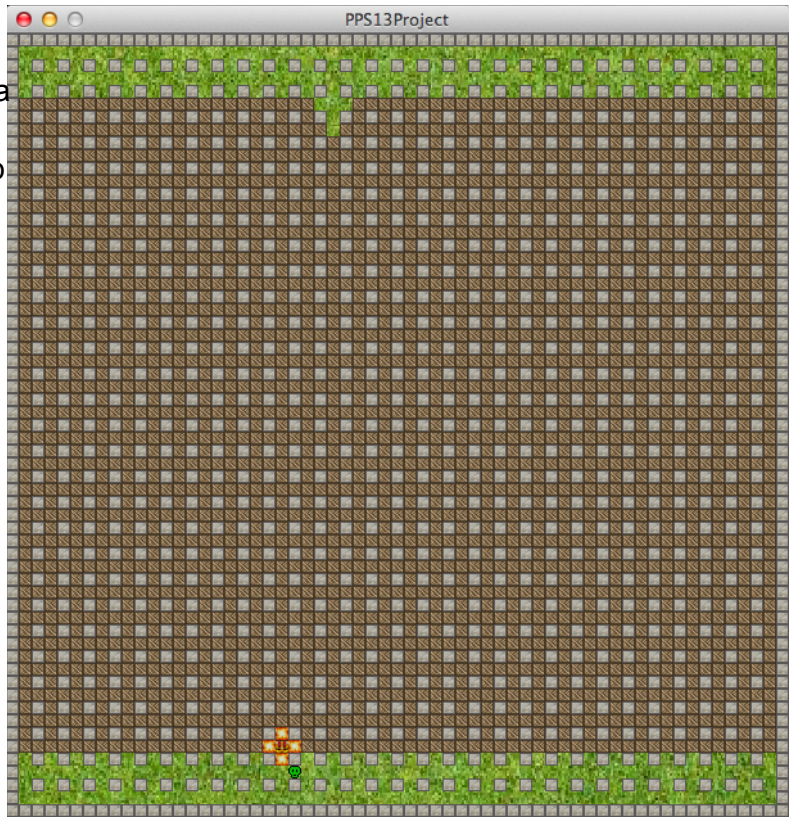
The game

Our project consisted of implementing a server and client based game.

Main idea

Our main idea was to create a shoot-'em up game with a 2d view of the game. We later decided to change the game idea for an easier game to play, but still uses the same complexity of the server.

The final version of the game is a multiplayer bomberman[1]. Where as a player you log in to a server and the can move around on predefined map and drop bombs to find upgrades. You win when you manage to hit the opponent or opponents.



The game map is 61 x 61 steps big. Everything in the game has the size of 1 x 1. This allows up to 61 players to join each team before the spawning area gets filled up. However there is no restriction on how many players there can be, except the maximum connections Java's serversocket can handle.

There is numerous of different items in the game, all with different properties. This allows us to add more items if we would like to.

The game engine has been modeled that it is possible to create any type of game if you like to. We have been implementing everything with a top-down design. Which means every part of the game handles one thing.

[1] http://en.wikipedia.org/wiki/Bomberman

This allows us to implement and delete parts of the game without anything else breaking. So it is possible to change the whole game idea to something else without having to rewrite most of the codebase.

Environments

The game was implemented with the Eclipse IDE and have only been tested on Ubuntu and OS X operating systems. It should work for any OS that have Java installed.

The game use only standard libraries.

Problems

From start we knew we would have difficulties with synchronization between clients.

Here is some of the problems we encountered and in the next section we will provide solutions to these problems.

Network

As we have a server that handles connections between multiple clients we needed a stable and reliable way to maintain TCP connections. We looked into both java.net sockets and RMI for this.

As the game is played in real-time and not turned based, we had to make the server broadcast all changes in the game to everyone more or less instantly. Slightest delay from the server would affect the gameplay for someone or everyone.

We decided to go with the standard java sockets in our implementation.

Next set of problems were on the server side of the game. Since the server does most of the heavy-lifting of the game logic we put most of our time into that part.

Multi-threading

The server has to run and maintain numerous threads and keep everything synced. At first we tried just give every part of the server a thread but when we scaled it up a bit we noticed it wasn't a good solution to the problem.

We had to come up with something better and that could process requests much faster, but still reliable.

The last major problem we had to overcome was the synchronization between clients.

The game map being the most important thing to be synchronized. It would be disastrous if not everyone has the same map.

Solutions

Network

As we decided to use Java sockets instead of RMI we had to construct the protocol ourselves and we came up with simple messages that gets passed around the clients and the server. The following table shows some of the instructions we use to control the game.

nick newnick	changes the players nickname
move <direction>	tries to move the client to that direction. Directions are: u, d, l, r for Up, Down, Left, Right respectively.
build :what at :x :y	build an object at the given locations. Objects are either mine or box, which can be pickup by the user.

When a client acts (move in a direction or builds an object) the client updates the gamestate locally and transmits the action to the server. The server updates the internal gamestate and transmits the result to all active clients.

However, if the action was invalid or illegal, the server will notified the client thus correcting the

problem.

Multithreading

Having just one thread to manage all the game logic proved to be slow and affected the gameplay. The obvious solution to this problem was to implement a bag-of-task solution for the server side tasks. Clients do not have a bag of tasks.

The server has a bag-of-task for all incoming commands sent from the clients which a certain amount of consumer threads will process. The default number of consumer threads is set to 3 but can be changed to more if needed. All incoming messages are processed and made to tasks. Thus all aspects of the server is executed in forms of tasks. Moving a player becomes a task.

There are other forms of tasks such as blowing up a mine, change user nick or Change team. All tasks are classes in the package *pps.et.server.tasks*. The reason we implemented one class per action was to make the code more readable and easily understood. Since every task is an atomic action, when you want to change something it's fairly obvious where to look.

Synchronization

Most of the issues with synchronization resolved themselves when we implemented the bag-of-task for the server.

The game had to distributed to all players and all changes had to be made for everyone. The gamehandler class is implemented as a monitor class for the project. Through its **synchronized** functions and the use of Java **Concurrent Data Structures** all threads on both the server and the clients can update the local gamestate without creating conflicts.

Structure

The project is divided into different packages. Where each package handles a specific part of the game.

pps.et.logic: Contains all logic of gameplay such as gamemap etc.
pps.et.logic.entity: All things that can be built. Mines etc, but also walls which are not destructable.
pps.et.client: All client code.
pps.et.server: Server code. Includes TaskHandler which runs consumer tasks.
pps.et.server.tasks: All tasks are defined as classes here.

The gamemap is a hashmap of all entities. Where the key is a Point, (x and y coordiantes). Thus we can only have one entity per point. Using it like this we could have a really large map size.

The structure is actually a ConcurrentHashMap. Thus, it's threadsafe.

Achieved knowledge

We've gained a lot of knowledge about how java sockets and which problems can arise when using streams. Like that even though you're using the print-line function to the stream the received message can contain multiple lines, so you need to handle messages in a special way.

Other than that we've gained knowledge in how threads and synchronization works in java. Allowing us to make this kind of project forced us to implement everything from scratch and in the end resulted in a better understanding of concurrent programming and its real applications.

By letting us choose a project of our own design instead of the predefined projects, we actually spent more time on the project. The task we defined for ourselves was a lot more fun and interesting and therefore we learned much more (BONKERS BANANAS)