The Lost Vaults: Uneasy Alliance

An online multi-user Dungeon Client and Server.
Project Proposal for group 4

Felix Färjsjö (19911225-4678)
Jimmy Holm (19870928-0138)
Fredrik Larsson (19890422-0590)
Anna Nilsson (19910804-0628)
Philip Åkerfeldt (19920508-1335)

May 1, 2014
Version 1.0

Abstract

A proposal for implementing a concurrent server model as the basis for an online, multi-user dungeon role playing game in Scala.

Contents

1 Introduction **2**

I Design **3**

2 System Architecture **3**
2.1 Client-side 3
2.2 Server-side 3
2.2.1 Concurrency Model 4
2.2.2 Actor Model 4
2.2.3 Class Model 6
2.3 Network Messages 7

3 Game Design **7**

II Development Process **7**

4 Development Tools **7**

5 Process evaluation **7**

1 Introduction

The final product of this project will be a fully functional and interactive multiplayer online role playing game, or MORPG, called The Lost Vaults.

The main challenges that we expect to face are writing network code as well as learning a new programming language.

The relation between the concept of concurrency and our project is the ability to have more than one player active in the game at the same time and their interaction with each other.

The game is a semi-traditional MUD, in which users connect to a central server and get the ability of interacting with other players and take on quests in parties. The game takes place in one of two locations, the city and the dungeon - the city being where the players may recoup from previous adventures, restock on supplies and form parties to delve back into the dungeon. The second location is the dungeon, a procedurally generated set of rooms populated with monsters, traps and treasures. The dungeon is procedurally generated and instanced in such a way that each party of players has its own dungeon, uniquely generated when the party delves down into the magical caves underneath the city. Once inside, the party is given one or several quests they should take on while exploring.

A quest can be finding an object and bringing it back to the surface, killing a number of monsters or collecting a set amount of gold. The party works together to accomplish the goals, but at the same time only one of the party members reaps the rewards, making the party a tense and fragile alliance of rogues. Gold earned in the dungeon can be spent in the city on equipment and consumables to increase the player's chance of surviving in the dungeon and taking on more difficult and threatening monsters.

Part I
Design

2 System Architecture

The system architecture is in this document divided into Client-side archiecture and Server-side architecture. Being written in Scala, the application runs in the Java Virtual Machine and as such, both client and server should be able to run on any platform supporting Java VM 1.7.

2.1 Client-side

The design goal for the Lost Vaults client is to create something akin to a remote terminal, where no logic is performed beyond sending requests, and receiving and interpreting request responses. Ultimately, we aim to accomplish a client-side implementation that is easily extensible when the server's implementation of game logic increases, i.e. a system we can add to by simply adding new requests and responses. As seen in Figure 1 the client window is divided into several text areas displaying relevant information to the player. The main area is for regular server responses and will contain for example room descriptions, player chat messages, system information messages and similar. The right side of the client is reserved for persistent information that the player may need to quickly reference. The top division contains information about the player's own character, listing the player's health and combat stats as well as the group's remaining food.

The bottom om the window contains the command input field, where the player types in commands in accordance to a strict syntax listed under the server architecture below. Upon starting the client, the player is faced with the login window shown in the center of the mockup, where the player can enter their username and password as well as the server IP address to connect to.

The client GUI will be implemented using Java's Swing UI library and integrated into Scala to utilize Akka's TCP capabilities, as shown in figure 2.

2.2 Server-side

The server-side system architecture is further divided into the sections Concurrency model, Actor Model and Class Model. The first describes our choice of concurrency model and the reasoning behind it, the second describes the func-

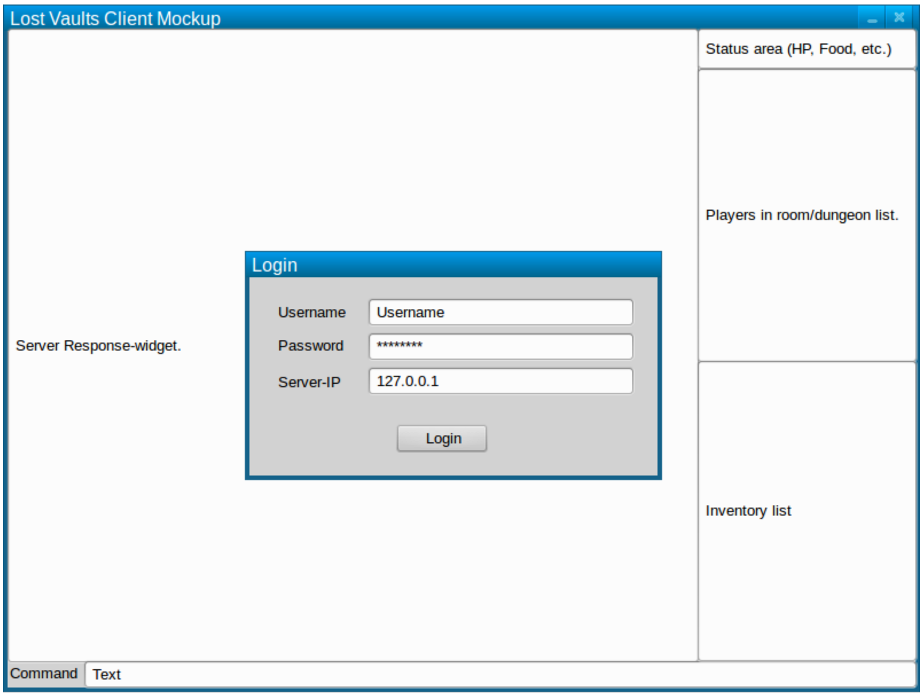

Figure 1: User Interface mockup of the Lost Vaults Client.

Figure 2: [TODO: Insert Client-system architecture figure]

tion of the five major actor types of the Lost Vaults server and the final gives a more in-depth description of the class relationships used by the server.

2.2.1 Concurrency Model

We are opting to use the Actor Model for the project as we during our initial design discussion managed to isolate behaviour into several smaller and largely independent sections, which could with ease be made to share data only through message passing and thus avoiding the problem of deadlocks by removing shared data from the equation.

In order to implement this kind of system with the greatest efficiency, we have chosen Scala as the language for the server as scalability is one of the fundamental design goals of the language. We then decided to use Akka as our actor library and to make use of Akka's TCP library for network communication. Akka's TCP library and its ease of use also guided our decision towards choosing Scala as the language of choice even for the client, as mentioned in the previous section.

2.2.2 Actor Model

The server side is divided up into several processes using Akka's actor library to enforce the actor concurrency model in accordance with figure 3. Only two

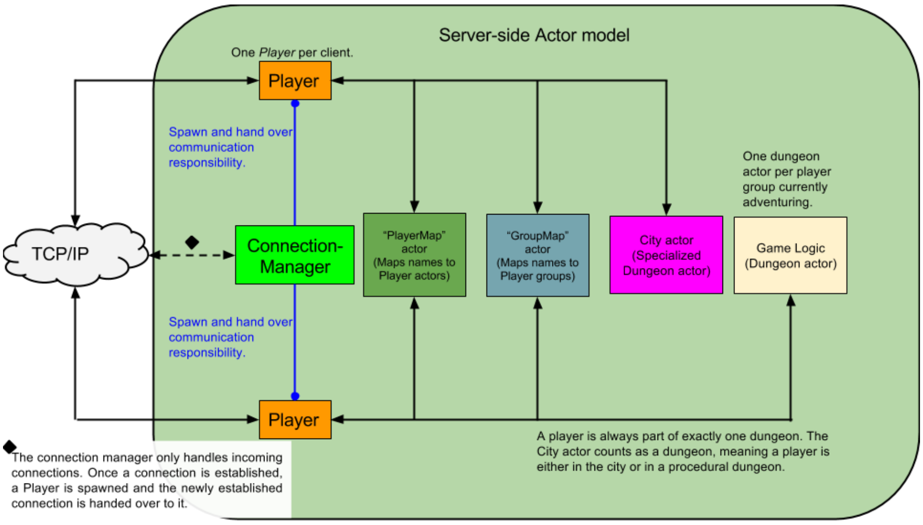

Figure 3: Diagram of the server actors' interactions.

types of actors communicate directly with the network through TCP/IP, the Connection Manager actor and the Player actors. The Connection Manager actor serves as the entry point for incoming connections, and is in charge of spawning Player actors as new top-level actors before registering them as the newly established connection's listener. As the listener, the Player actor will receive and send data over the network through the established connection. The Player actor is in charge of all things related to the clients, such as the specific player's attributes, is in charge of verifying login attempts through communication with the PlayerMap and it is through the Player actors that the other actors can send messages over the network when necessary.

The PlayerMap actor and GroupMap actor serves two specific utility functions, the first of which is to map a string, the names of all connected players, to the actors controlling the player with that name. Through this actor, another actor can quickly and easily send messages to a specific actor based on its name - an important feature for commands acting on other players, such as *WHISPER*. It also allows a quick and simple way for the Player actor to ensure no more than one player of a given name exists on the server at any given time.

The GroupMap actor performs a similar function as PlayerMap, however it links player names with player groups instead of player actors. A player group, as described in the game design section of this document, is a collection of one or more players who together can enter a dungeon that is procedurally generated for them. The purpose of this actor is to facilitate forming and joining other groups of players. Using the GroupMap, it is enough to join a player by name to be included into that player's group, or have a new group formed for them.

The fourth type of actor used by the Lost Vaults server is the Dungeon actor, the actor performing all game logic. There is a special case of Dungeon actor which is called the City actor, which is a Dungeon actor that is persistent over the lifetime of the server and has special behaviour to handle the socializing aspect of chatting and forming groups as well as trading with NPC merchants. The ordinary case of the Dungeon Actor is in charge of generating and handling a procedural dungeon of multiple, interconnected rooms layed out in a two-dimensional grid. It is in these dungeon rooms that players will have the ability to complete quests, find treasure, etc. The Dungeon actor is in charge of coordinating all the players that are currently present in the dungeon, and make sure that commands such as *SAY* are relayed to the relevant players. With the exception of the City actor, the Dungeon actors are in charge of their own lifespan after having been spawned by the City actor, and a Dungeon actor will stop itself once there are no more players within it.

2.2.3 Class Model

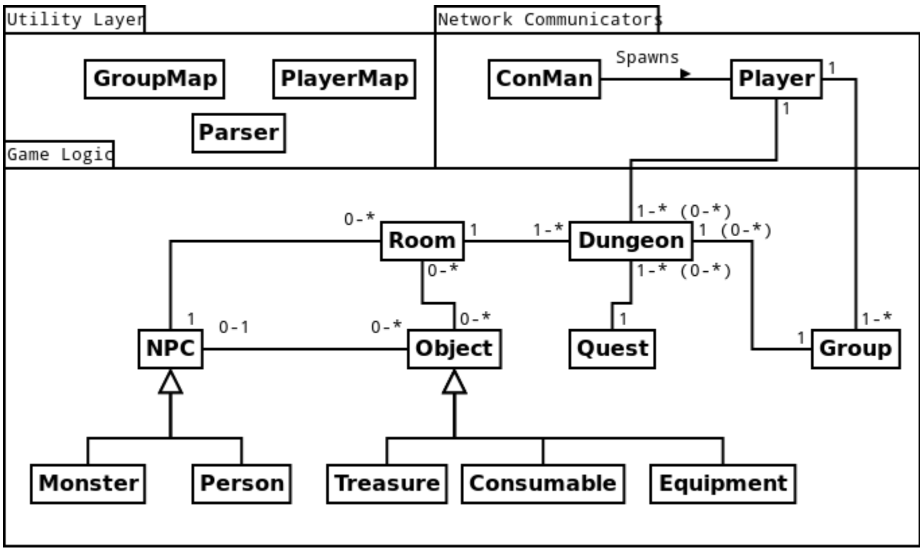

Figure 4: The server's class relationship model.

Figure 4 show the relationship between the different classes of the Lost Vaults server implementation and the multiplicity of relevant relationships. They are divided into three categories; the Utility Layer, the Network Communicators and the Game Logic. The Utility Layer provide supporting functionality to the other classes and exactly one of each class should exist at any given time. The Network Communicators are the ones that relate back to the network through Akka's TCP library. Of the two, ConMan, the connection manager class, is to be considered a singleton and only one instance should exist at any given time. The Player class however has one instance for each open connection.

The Game Logic section contains all the internal classes as well as the Dungeon class. The Dungeon class is the bridge between the Network Communicators and the Game Logic and it is only through a Dungeon class that a Player can perform any actions apart from the server-wide *WHISPER* command. The Dungeon class keeps track of explored and unexplored rooms, quests, groups, objects and NPCs. The Dungeon also keeps track of all Players currently connected and associated with the given dungeon.

2.3 Network Messages

Message	Direction	Description
SAY [msg]	Client → Server	Chat message sent from client to server.
SAY [name] [msg]	Client ← Server	The Chat message sent by [name] and relayed to the rele
WHISPER [to] [msg]	Client → Server	A private message sent from a player to another player n
WHISPER [from] [to] [msg]	Client ← Server	A private message sent from a player named [from], to ar
LOGIN [name]	Client → Server	A request to sign onto the server using [name] as the pla
LOGINOK	Client ← Server	Response from server that a login request has been accep
LOGINFAIL	Client ← Server	Response from server that a login request has been denie
[TODO:	Insert the rest	of the messages.]

3 Game Design

[TODO: Describe game design here!]

Part II
Development Process

4 Development Tools

For this project we intend to use a number of tools that we see fit for the project. Since we will be working in Scala we will be writing our code in Eclipse. The reason we chose Eclipse was because it is, according to us, by far the easiest way to develop code when writing in Java or Scala. Since all of the development of the code will be done in Eclipse there will be no need to worry about using Make or a similar tool to automatically build software since that feature is already a pre-existing part of the program.

To handle the version control we intend to use the reliable and well known tool Git. We will most certainly use the repository previously created by our group to host the files for the project. The reason we chose Git was mainly because all of the members of the project are well acquainted with the tool and we have been encouraged to use the tool in previous projects. For the testing of our code we will use JUnit since we will be working in an environment that is very similar to Java.

For assembling our documentation we will use Scaladoc since it is a tool we are all familiar with and it's also pleasant to be able to put our efforts elsewhere and focus on more important parts.

5 Process evaluation

We applied the Osborn principles for brainstorming with a small addition to the process. In the brainstorming process we went around the table and each group member had to add at least a couple of ideas before we let anyone who had an idea add them to the list of possible ideas. This led to many ideas which might lead to other group members coming up with new ideas from the proposed idea.

The initial brainstorming went as expected with a long list of possible project ideas as the result. No ideas were criticised during the brainstorming and all the group members' ideas were shared. Nothing specific could be improved about our brainstorming session.

During the brainstorming session three main project ideas were presented. The first idea was to make an online multiplayer game and the two others were simulation programs, one to simulate the traffic situation in a city and one to simulate an ecosystem. We divided the group into three focus groups that looked deeper into these three options. After this, each group were given time to present their respective fields, and after discussing pros and cons for the potential of each idea, we decided to choose the online multiplayer game. The reasons for this was that the complexity was intriguing, and that there was potential to develop the game further in case of extra time. Some of the interesting parts of the online game, besides being the most fun, was to learn about networking and our chosen programming language, Scala.

Focusing on quantity in the first stage of brainstorming led to a good brainstorming process with plenty of ideas and doing so we found that it was easier to decide on a high quality idea when ideas could be put on the table and refined by the group as a whole, rather than feeling that any idea had to be fully formed before being suggested.