M2MIMud

Robert Whitcomb

November 8, 2004

Committee: Alan Kaminsky, Hans-Peter Bischof, Andrew Phelps

Abstract

M2MIMud is a simple MUD (multi-user dungeon) type game that is designed to run on an ad hoc network of devices without the presence of any central server. It runs over M2MI, a distributed object systems API that uses broadcasts and handles, rather than IP addresses, to communicate. This paper discusses the design and development of M2MIMud, delving into the issues unique to such a system. Among topics discussed are a survey of existing architectures, a discussion of M2MI, and talk about the unique aspects of this system. Also included in this paper are a player's manual and some design documentation. Overall, M2MIMud is an interesting study in the efforts to port existing types of software to an ad hoc environment.

1.0) Introduction

One of the earliest forms of computers games is the multi-user dungeon, commonly referred to as a MUD, which are simple (compared to many present games), text-based systems which usually create a role-playing environment for their users. Players assume the identity of someone else and play in a fantasy world according to the rules of the developers. With the proliferation of cheap, high-powered computer hardware, especially video cards, players today tend towards the MMORPG (massively multiplayer on line role-playing game). However, there are still many MUDs out there. Examples of these are Aetolia (http://www.aetolia.com), RetroMUD (http://www.retromud.org), and Threshold (http://www.thresholdrpg.org). While they do not usually have the large player base that an MMORPG has, MUDs are still important game systems.

With the advent of small, low cost computing devices like cellular phones, tablet PCs laptop computers, and personal digital assistants, a new arena has opened up for computer games. This is the arena of the ad hoc network. As opposed to standard networks, devices in an ad hoc network are highly mobile, with members coming and going at will. There are no dedicated lines, no routers, and generally no central servers. All communications are done on a point-to-point basis, and the members of the network usually do routing. Also, ad hoc networks, as the name implies, are temporary, forming at will by the interested parties and dispersing just as quickly. The networks have incredible potential; one such example is that of a network formed by investigators at a crash site, where they can use their PDAs to gather information and send it to the other investigators in a short time.

It may seem that an ad hoc environment is not suitable for an online game like a MUD. One of the key components of a MUD is the existence of a pervasive world. Users may log on and off at any time, even take a few a days break and come back. No matter what, their character is there, and the world in which they play in is there (provided that the MUD has not been taken offline). As such, all of these systems have a central server that users log into and play. An ad hoc network does not seem suitable for such a system.

M2MIMud is study of that claim, that an ad hoc network is not a suitable environment for a MUD. It is an attempt to create a pervasive, state based system on an environment that does not inherently support it. As such, there are many issues to be dealt with not normally found within a typical MUD. Many things taken for granted in a standard MUD cannot be in an ad hoc network. This paper presents M2MIMud, detailing the design of the system, discussing the issues and presenting the solutions.

2.0) Background

2.1) Client Server Architecture

Massively multiplayer games are very profitable for any company that is capable of building one and attracting customers. Typically, these games remain online for several years. Mythic Entertainment's *Dark Age of Camelot* has just reached its 3rd year of play. Sony's *Everquest* has been around since 1999. And, perhaps the true testament to the length of the online game is Electronic Arts' *Ultima Online*, which began in 1997. In

addition to being online for several years, these systems boast thousands of customers. In am recent example, Square-Enix recently announced that the number of registered characters has reached one and a half million. Although players can have more than on character per account, this still means roughly more than a half a million players. Their MUD counterparts, while they might have fewer numbers of customers, typically can compete with their more advanced brothers in terms of lengths of play. These games offer a lot, from questing to competing with other players. They offer different worlds, different rules, and different player classes. However, these games all share one thing in common: the basis for their architecture, which is the client server architecture.

In the client server architecture, the world resides on a large server. Players typically purchase a client of some sorts. They then use this client to connect to the server to play. Clients usually contain the graphics and sounds of the game, as well as the software needed to talk to the server so that clients can inform the server what the player is doing. The world, the characters, item database, monster information, and rest of the game data are stored on the central server. This is the type of architecture used by the majority of games on the market today.

This architecture has both benefits and disadvantages. Two chief advantages are synchronization and the safety of game information. One of the chief requirements of a game is that game data is secure. This helps to ensure that the game remains fair, so that a player can only become powerful within the confines of the game (i.e. prevent cheating). It also helps to keep players' data safe so that they know their character will have the same items, levels, etc as before. In terms of synchronization, there is ultimately only one system handling the state maintenance. The clients sent their commands to the server; the server executes, and then informs all interested parties about the change. Although there can be several clients, there is only one central server which is responsible for maintaining and updating the global state.

In terms of disadvantages, the biggest issue is that the server is a single point of failure. If the server goes down, the players cannot play the game. Given the nature of the game, this makes sense, but it can cause frustration on the part of the players. Also, it's possible for the server to become bogged down with requests and begin to delay its responses to the players, commonly referred to as lag. Overall, however, the client server architecture has been very successful for many game systems.

An excellent discussion of the basis of client-server architecture, as well as the issues that surround can be find in [1] (pages 94-108) and [17].

2.2) Real Time Strategy Games

Another form of architecture can be found by examining so called real time strategy (RTS) games. These games include titles like Microsoft's *Age of Empires* and Blizzard Entertainment's *Starcraft*. In an RTS, players usually choose a team to play. They then must gather resources, build up an army, and defeat their opponents. Unlike MMORPGs, they do not have character information to store. There is no character development. What's key, however, is their architecture. "... the *Age of Empires* series takes a different approach in which each peer runs the entire simulation and distributes the user's input to each peer. Consequently, all peers execute the same command at the same time and thus

remain consistent." (Bauer, 37) This form of architecture is advantageous, since players do no need a server. Rather, all they need is someone to establish a game that they can join. Once joined, there is no "leader" So, the player who created the game could easily leave and the game would continue (except that his team is no longer active).

This architecture has some advantages, mainly because of the lack of a single point of failure. As long as one system is running the simulation, the game will continue. However, this limits the scalability, because "scalability is limited by the computational power of the weakest peer, because each peer needs to run the simulation." (Bauer, 37) In the client server architecture, a slow client will not affect game play for the others, mainly because all commands are sent first to a central server; each system only computes the data relevant to its player, given the location and activity of the player. In the distributed setup, however, each peer computes all data for the game, which means that the game can only go as fast as the slowest peer. This of course, is a major disadvantage, since if people are playing with a person who has a very slow computer, it will mean they will experience slow downs.

Another disadvantage of such a system is that it generates a lot of network traffic, so that only a small number of players can play at a time.

2.3) M2MI

M2MI, which stands for Many-to-Many Invocation, is a new distributed object system API developed at the Rochester Institute of Technology by Professor Alan Kaminsky and Professor Hans-Peter Bischof. It is a broadcast system, whereby communication is done via method invocations on handles. Objects in this system implement an interface, and then export themselves. Other objects can then get these handles and execute functions on them. There are 3 types of handles. A unihandle refers to a single object, and an omnihandle refers to all objects that implement a given interface. A multihandle refers to an explicitly specified group of objects.

M2MI is optimal for usage in an ad hoc network because it does not need IP address information. Rather, all one device needs to talk to other devices is the appropriate handle, and then make the correct method calls. This eases development of an ad hoc system, where all aspects of the network can change at any given moment. More information about M2MI can be found in [2].

2.4) M2MIMud

A MUD is a game that requires a static server to work. Players need dedicated connections. It may seem difficult to attempt to write a MUD like system to run over an ad hoc network. However, what's important to realize that M2MIMud is not a typical MUD. It can't be, given the environment for which is intended. There is no central server, since there is no guarantee that a server will be up, or that devices will be in range of a central server. This presents a problem, since the central server is what stores game information as well as executing the user's commands on the system. Also, in an ad hoc network, players can be temporarily disconnected and then come back and they would be none the wiser. It seems like a daunting task, but it can be done. What needs to happen is to modify the view of what a MUD is given the type of network it was designed to run

over. M2MIMud is attempt to mimic the behavior of a MUD over an ad hoc network, taking into account the unique issues presented by such an environment.

2.4.1) Goals

There are quite a few goals of M2MIMud, which affect the design of the system. This section lists those goals.

a) First and foremost, there is no central server. There is no one unit that users log onto. All members of a session have an equal say in determining the global state of the world.

b) That being said, all operations in M2MIMud will be as close to atomic as is possible, and there will be only one object that can modify the state. This is done to help keep the states maintained. The idea is "one thing at a time". One state change occurs before another does. Only one object is capable is changing the state. This setup prevents race conditions, and it prevents the possibility that two changes could occur at the same time that conflict with one another.

c) For the most part, only the unit that controls the player can make changes to that player. This is done because the only thing that can be trusted when it comes to a player's situation is the associated with that player. There are two exceptions to this rule. These will be described later in Section 5.9

d) Another goal of M2MIMud is to keep it simple. The design and development of a full scale MUD is a complex process that can take years. The purpose of this project is to demonstrate that MUD type application can run in an ad hoc environment. As a result, only the basics of a MUD are covered. These basics include:

- Movement - the ability to move around the world.
- Communication - the ability to talk to other players
- Combat - the ability to fight monsters and other players
- State changes- the ability of players to modify their state.
- Interactions with NPCs - NPCs are non-playable characters, and for M2MIMud, these are the merchants.

Given these goals, it was quite a challenge to find a way to make M2MIMud run over an ad hoc environment. Research was done into the areas of state sharing in distributed systems. One potential candidate was found in [18]. However, this design pattern makes uses of a master unit to maintain the state. This is essentially the same a central server, something M2MIMud was designed to not have. Also, this pattern does not handle network partitioning, a problem discussed in Section 3.4.

3.0) Architecture

The basic design of M2MIMud is something very similar to *Age of Empires*, in that each

device runs a copy of the game, and is responsible for maintaining its version of the state. Unlike a classic MUD, where clients send commands to the server that in turn send the results of these commands to all other listeners, in M2MIMud all commands are sent to all other clients so that they are aware what players are doing to the state.

3.1) General Description

As stated in the beginning, M2MIMud is a MUD type application. This means that it is a role playing game in which users assume a class and play within world. It is entirely text based.

In M2MIMud, users play as player characters, which have several components to them. First and foremost is the player's class, which is a description of the type of character they are. In M2MIMud, there are two classes: the warrior, who is a melee fighter, and the mage, who casts spells. The warrior is physically tougher than a mage, can wear heavier armor, and carry bigger weapons. On the other hand, a mage cannot take much damage, and is limited to light armor and staves, daggers, and wands. However, since mages can cast spells, they can inflict greater damage on their targets and kill them much faster than a warrior. The second component is the player's experience level, which is a measure of growth. As a player increases in level, he or she can take more damage from a target as well as deal more damage to said target. In order to gain levels, a player must accumulate enough experience points, which is usually done by fighting certain targets within the world.

Players play in a large object referred to as a world, which represents the physical boundaries of where the player can go. A world is comprised of much smaller entities referred to as rooms, which can represent a physical room within a building, a street block, or the immediate area around a player while in a large field. There are three types of rooms in M2MIMud, grass, water, and woods. These refer to the description that is given when the user enters the room. Grass means that the player is in an area full of grass, like a meadow or clearing. Woods means that the player is surrounded by trees, and water means that the player is swimming in water.

In addition to players and rooms, worlds have other things in them. There are monsters, which are non-playable creatures that players may kill to acquire experience points, items, and money. In M2MIMud, monsters are mobile, which means they move, and non-aggressive, which means a monster will not attack a player unless the player attacks it first. Players may also fight other players. However, in M2MIMud, player versus player combat is not open, which means that one player may not just openly attack another player without warning. They must first issue a challenge. If accepted by the other player, the two will begin a duel, which can only end when one player has defeated the other player.

In addition to monsters, there are also merchants, which are also non-playable characters (NPCs). Merchants do not move, and they cannot be attacked. Much like their title implies, merchants will buy and sell items. There are three types of merchants: an armor merchant who sells protective gear for a player to wear to reduce damage, a weapons merchant who sell weapons and spells, and an item merchant who sells primarily healing potions a player can use to recover after a battle. These potions recover hit points, which

are a measure of how much damage a player can take. Merchants will also buy any item that the player wants to sell, whether it was obtained from defeating a monster or old equipment the player no longer wants. In the latter case, old equipment will be bought by a merchant at a fraction of the cost it was sold at.

Players may also make slight modifications to the world. In each non-water room, the player has the option to add a small pond. There can only be one pond per room, and all players will know if a pond has been dug in the room they are in. Players may also set up one house in the world. A player house has a specific owner, and other players will be informed of whose house is near them when they enter a room. There can be multiple houses in one room, but a player may set up only one house.

In order to play M2MIMud with other players, a player must either join or create a session. A session is a group of players all playing within the same world. They can communicate with each other, and they will see the actions of other members within a session. Sessions are comprised of a layout, which is a map of the world. A layout contains the dimension of the world as well as the type of each room. Sessions also have states, which contain information on who is in the session, where they are and what they are doing, where the monsters are and what they are doing, and the location of the ponds, houses, and merchants. Two sessions may have the same layout, but they will not have the same state.

To communicate with others, players have three options. The broadest form of communication is the yell, in which the user's message will be displayed to all members of the session. The next form of communication is the say message, in which the message is broadcast only to players that are in the same room as the player. Third is the tell message, which is a private message that is sent from one user directly to another. Because users can have the same name in M2MIMud, each character is assigned a special ID when they are created. Other players are informed of this ID, and it is used in some circumstances when it is unclear who the user wishes to interact with. For example, if there are 2 players with the name “Susan” in the session, a player who wants to send a tell message to one of them must indicate the ID of the Susan they want to talk to. Information about how this is done can be found in the User's Manual.

In order to do anything in M2MIMud, users must input their actions in the form of commands. A listing of the format of commands and what they do can be found in the User's Manual section of this report.

A typical game of M2MIMud would go something like this: the user starts up M2MIMud, and selects or creates a character. They will then join or create a session, in which other players can join. While playing, users can move freely about the map, talking to each other, digging ponds, setting up houses, fighting, and performing transactions with the merchants. When the user no longer wants to play he or she will disconnect from the session, and then shut down M2MIMud.

3.2) Commands

In M2MIMud, commands are sent to all other members of the session. More precisely, relevant commands are sent the other members. For example, one of the earliest

commands implemented was the “walk” command. This command attempts to move the user’s character on tile in the specified direction. The results of this are then sent to all other clients. For example, say that the user wishes to move one tile to the west. She will enter the command as “w west”. Her personal system will check to see that she can move west from her current location. If not, she is informed of this, and no broadcast is sent. However, if the move is valid, she is moved in that direction and her location is updated. Her personal system than bundles the movement data into an object that is broadcast to all other session members, which in turn use this information to move their copies of the player to the specified location. If applicable, it informs their user if the player is moving to or from their location, since it is typical in MUDs to inform players when other players move near them.

However, not all commands are sent to the session members. An example of such a command is the buy command, which allows the user to purchase items from a merchant. Other players do not care about the inventory of any player other than the ones they are controlling. They will eventually receive an updated copy of the player’s inventory, but it is not a necessary update since their players cannot affect another’s players inventory (i.e. one player cannot steal from another player). Therefore, the effects of the buy command are not transmitted since they do not affect the global state of the game.

In addition to commands such as buy, there are also some commands that are sent to only certain members. An example of this command type is the send command, which sends a message to a specific player. Also, all dueling related commands (challenge, accept, decline, and the attack commands) are only broadcast the necessary player. In this fashion, the systems attempts to reduce the size of the messages sent to the session, and only send needed commands.

3.3) Sessions

One approach to M2MIMud would be a global game that all players join. While a novel idea, it is limiting, since one of M2MIMUDs goal was customization of the world in which the users play. Therefore, rather than simply play in one large, global game, users play in smaller units referred to as sessions, which is a group of players playing together in one world. The world comprised of two components: the layout (or map) that indicates the size of the world and how the tiles are laid out, and the state, which contains information about the locations of the monsters, merchants, players, etc. One of the more challenging aspects of M2MIMud was solving the question of how to transmit this state to other players. Once a user has created a session, his copy of M2MIMud starts a timer that, upon timing out, causes the system to broadcast out his "session ad". This session ad contains the name of the session, the multihandle associated with it, and the current state of the session. Upon join a session, an incoming unit takes this state and uses it to build its own internal state.

A joining unit also takes upon itself the task of periodically broadcasting out its session ad. However to cut down on network traffic, if a unit receives an ad for its session, it will restart its timer. For example, if A is broadcasting out a session ad, and B joins the session, B will also start broadcasting out the session ad. However, if B receives a session ad broadcast from A, it will restart its own session ad timer. The idea is that each

unit has a copy of the state that is good enough for an incoming unit to use as a start. These session ads are also used for an internally by the members of a session for state maintenance, but that is discussed in another section.

The following is a more detailed description of how this process works. In the Communications package of M2MIMud, there is an interface known as GameDiscovery. This interface has one method: report. This function is inherited/implemented by a few interfaces and classes, the two most notable being the SessionAdvertiser (GameSystem implements this class) and the SessionFinder.

Report is always invoked on every object that implements it. What happens afterward depends. The SessionFinder will do one of two things. The report function carries with it a session ad for a game session. If this session is not already registered with the Session Finder, it will register it. However, if the ad has been registered, it will update its internal copy of the session ad for that session, and restarts the timer associated with that session. Each session ad has a timer associated with it to ensure that users will not be able to join a session that no longer exists. It is the responsibility of the members of the session to periodically report their existence to the Session Finder object.

For the SessionAdvertiser class, the report function is actually declared as abstract. Rather, the child class of the SessionAdvertiser, the GameSystem class, is what implements this function. When a report method is invoked on the GameSystem, the ad associated with the report method either belongs to the session that the system is in, or not. In the latter case, the report is simply ignored. In the former, two things happen. First, the GameSystem uses the ad associated with the invocation to perform state checking and maintenance (if need be). Second, it restarts its broadcast timer. The SessionFinder does not care about the unit that sent the ad, it only concerns itself with the session that ad pertains to. Therefore, an ad sent by any member of the session is enough to restart the timer associated with that ad.

3.4) State Maintenance

One of the challenges in M2MIMud is maintaining the global states to ensure that all members of a session play in the same world. Normal MUDs and MMORPGs have a central server that takes care of this. The clients send their information to the server, which in response makes the necessary changes and the somehow broadcasts out that information to other clients. A situation in which one client becomes unsynchronized is handled swiftly, since the central server serves as the final word. Whatever it decides is what happens. (There may be a rare occurrence in which the user contacts the customer service of the game and the employee overrides the server's decision, but since M2MIMud does not have a customer service that is not an option.) In M2MIMud, there is no central server. The global state is not stored on one machine, but is rather distributed among all members of a session. When one device becomes unsynchronized, there is no source unit it can turn to. It must rely on its own methods to help resolve the problem.

The most probable cause of unsynchronized states is a network partition, whereby some members of the session simply do not receive messages. The most probable cause of partition is that one device has moved outside of the transmission range of the other devices. This presents a problem, since players do not know that they are partitioned.

This can lead to some major state clashes, whereby 2 players could be listed as attacking the same monster. The solution to this utilizes the fact that each unit will periodically broadcast out its version of the state. In addition to resetting a timer, each time a session broadcast out it state, all other members check to see if their state matches the broadcast state. If so, the game assumes that everything is fine. If not, however, it performs a state healing procedure to bring its state back within a reasonable condition. Since, all session members have equal say in the state, any state broadcast by a member is important to look at when trying to determine the global state of the system. Therefore, if a local state is not the same a broadcast state, the local state must be synchronized with the broadcast state. Of course, it's not just a simple copy over. There are some rules (discussed later) that are applied, and the system is also capable of adjusting the broadcast timer's timeout value, so that it can broadcast a state out faster than other members. This method is not perfect, but it does help to keep the states within a reasonable range of equal, so that players have the necessary state information to help them play the game.

3.5) Game Data

Since there is no central server, all game data is stored on the unit itself. The user's character data, the class information, and the data about monsters and items are stored on the player's unit. Also stored are the maps the user has created and the states that he or she has played in (if these have been saved). These, of course present some issues, since this method allows for a skilled user to hack into these files and change them to benefit their character. More will be said about this in the design issues section, but it worth placing a small note here since the method in which game data reflects the fact that this is truly an peer to peer based game, since all units are responsible for their own characters and game data.

4.0) Design

As mentioned previously, the design and development of M2MIMud is influenced by RTS games such as *Age of Empires*, in that the user's command is, in some form or other, executed on every client in the session. Obviously, there are many ways to go about this. This section details the design of M2MIMud, going into the basic mode of operation, a description of the various packages, interfaces (both M2MI interfaces and normal interfaces), and more. There is also a discussion of various issues that arose during the implementation and the ways in which they were dealt with.

4.0.1) Design Assumptions

There are two assumptions that are important to discuss so that they can be kept in mind while talking about the design. The first assumption comes from M2MI itself, the second comes from the project.

a) Message delivery is mostly reliable

b) Messages from group members are secure.

The first assumption is a major part of the design. One of the most important aspects of M2MIMud is that it tries as best it can to keep the state in all members of a session in

sync. One of the issues, however, is that there is no central server, nor is there a governing device that determines what the state is. The global state is maintained by all members of a session. This has it advantages, as there is no need for an election algorithm, and if a device goes down, it's not a disaster. It does have its disadvantages, since if one of more devices suddenly stops receiving messages it goes out of sync. So the assumption is made that the majority of devices will receive messages. Partitions and dropped messages are rare. This is a reasonable assumption, since, like most ad hoc applications, the devices will geographically be close to each other. Unlike a normal MUD or an MMORPG, M2MIMud it primarily designed to be played by a few players close together, like in a library, classroom, or bar.

The second assumption needs to be made since the purpose of this project is the development of a MUD on an ad hoc network, not group security. Group security is a major topic in ad hoc development, and there are several projects devoted to it. For more information, see [14], [15], and [16]. However, what this assumption means is pretty simple. When a device receives a message, it assumes that the message is from a member of its group and that the device is who it claims to be. Only valid M2MIMud units can join a session, and, as such, all messages are assumed to be secure and valid.

4.0.2 Design Considerations

There are many elements that are commonly found in an MUD that are not found in M2MIMud. Also, there are several areas in which M2MIMud differs from a normal MUD. This is because there is no central server which users can turn to in order to resolve issues that come up during game play. This section discusses a few areas in which M2MIMud operates differently from a normal MUD.

One thing to reiterate is design goal (c) mentioned in section 2.01. That rules states that only the unit that controls a player may make changes to that player. The reason for this is simple. There is no central server, so the only thing a unit has to go by when making changes are the other units in the session. As such, it would be unwise and unfair to allow any unit to make changes to the players. This is because an unscrupulous player could hack the system and make changes to other players that harm these players while benefiting themselves. As such, only the unit that physically controls a player may make changes to that player. These changes include the position and combat situation of the player. With this in mind, the differences between M2MIMud and client-server game makes more sense.

First and foremost, there is no group based combat. In MMORPGs and MUDs, it is common to allow players to group together so that they may fight the same monster. This allows players the opportunity to fight monsters which are more powerful than them, and thus earn greater rewards than if they were solo. In M2MIMud, this is not the case. Only one player may attack a monster at any given time. This is due, primarily, to the lack of an aggression table. In group-based fighting, monsters will usually attack the player who has the highest aggression rating, commonly referred to as "aggro" or "hate" by players. The implementation of this is simple. As soon as combat begins, all members of the group are placed into a table with a rating value. As they perform actions, this rating changes. The player with the highest rating bears the brunt of the monster's attacks. As

mentioned, such a system does not exist in M2MIMud. This is because there is no central server. Such a table would need to be synchronized among all clients and update constantly. This would mean all units would need to be notified of a player's actions and thus adjust their aggro rating. This would cause an enormous amount of network traffic, and if a packet is lost or a network partition occurs, things could get ugly fast. For example, say that players A and B are fighting a monster, which is currently attacking player B. A partition occurs, and player A performs an action that increases his aggro rating to the point where the monster is attacking him. On one system player B kills the monster, whereas on the other, the monster kills player A. If the partition heals, the question becomes confusing as to what to do. Player A is no longer fighting the monster, so the player B's unit would have to remove him from combat, which could be bad if player B needed player A to kill the monster. This can lead to a rather confusing mess, as the systems have to work out what went wrong. As such, avoiding it all together prevents the problem.

Of course, this rule of "one player to one monster" also has implications on the synchronization process that occurs when a network partition has occurred and the unit is attempting to recover. In a normal set of events, if a player attacks a monster, a message is broadcast out to all units to place that monster into combat with the player. This remains the case until the unit that the player has killed the mob, the mob has killed the player, the player has run away from the mob, or the player times out. If any other player attempts to attack a monster before the second message is received, their unit prevents that from happening. However, if the system is partitioned, funny things can happen. For example, assume a partition has occurred and players A and B are now listed as attacking the same monster. When the partition heals (and for the sake of discussion, they are still attacking the monster), the player with the earliest timestamp wins when deciding who is really attacking the monster. When a player begins attacking a monster, the time is recorded. When perform a state healing process, these timestamps are checked, and the player with the earliest one "wins" in a manner of speaking. The player with the later is removed from combat (and is informed of this). This is done primarily to prevent kill stealing, whereby a player could partition, cause another player to timeout, return, and force the other player from combat.

Of course, this only occurs once the state healing process occurs. It's more than possible for a combat event to occur before the state has had a chance to heal. What happens depends on the event. Assume that player A is attacking a goblin. If his unit receives notification that player B begins attacking the same goblin, his unit performs an emergency broadcast in order to inform player B's unit that the mob has already been claimed. If the player receives a death notification (either the mob they are fighting dies or another player is killed by the mob), they are removed from combat with the mob. This is done because this is an indication that something is really wrong, and the system has no idea who attacked the mob first, so it gives the benefit of the doubt to the other player. Finally, if the system receives notification that the monster that its player is fighting has respawned, it performs an emergency broadcast to "claim" the mob before anyone else can.

Of course, there is also PvP combat as well. As opposed to monster combat, this is not time stamped. The rule is simple: once a player is in combat with another player, the will

remain in combat until the second player times out or informs the first player that they are not longer fighting. For example, assume players A and B are fighting. A partition occurs and player B ends her fight with player A, and then begins to fight player C. When the system recovers, player A will remain in combat with player B until player B informs player that she is no longer fighting him. If player A receives a state broadcast from player C, his unit will notice that this broadcast claims that player B is now fighting player C. The unit has no good reason to believe this is the case and several to believe this not the case (hackers, for example), so it ignores this. However, if it receives a broadcast from player B that player B is now fighting player C, the unit will remove its player (A) from combat. If player A receives notification that player B has been killed by another player, or has killed a player other than player A, then player A's unit will remove it from combat.

Two last things to note. In many RPGs it's common for there to be quests. These are something that a player does for some reward, like an item or experience. In M2MIMud, there are no quests because there is no way to verify that a world makes sense for a quest. For example, a quest the wants a player to kill a sheep in water while the moon is full would make no sense in a world in which there is no water. All that could really be done is to allow players to create their own quests, but that is beyond the scope of this project. A final thing is crafting. In many games, players are allowed to craft items that they can sell to other players, thus creating an economy run by players. This is not done in M2MIMud for two reasons. First, it's beyond the scope of this project. A player run economy is a difficult thing to design and get correct. See (1) for an excellent discussion of the issues with a player run economy. Second, as with quests, crafting requires certain things to be present in the world. Without a central server to verify that things are correct, this would be almost impossible to do. For example, assume the player needs to harvest materials from a forest. If the world they are in has no woods, there's not much they can do.

4.1) Packages

All things in M2MIMud, except for the main program itself, are a part of a package. There are a total of four packages, which work together to perform the various functions of M2MIMud. These packages are:

- Command – The command package is responsible for the process of taking a user's input and transforming it to a Command object, which contains the data needed by the system to perform the user's action.
- Communications – The communications packages contains the code that deals with transferring data between the various members. This package contains the M2MI interfaces, as well as the interfaces (and some partial implementations) that deal with the finding and reporting of game sessions.
- State – The state package contains the classes that maintain the personal state of the user.
- Game – The game package is the "master" package so to speak. It has the responsibility of using the other packages to perform the user's command. The

premier class here is the GameSystem class, which is the controller for the entire system.

This is but a brief overview of the packages of M2MIMud, to help set the groundwork. More of the various classes within the packages will be presented later.

4.2) Interfaces: M2MI

This section details the M2MI interfaces used in M2MIMud. These can be found in the Communications package

Interface Name: DiscoverableGame

Description: This is the interface for an object that is an M2MIMud game that can be discovered.

Functions:

- request() – The request function informs this object to reduce the time until it's next ad transmission; in effect, this causes the unit to report itself faster than usual.

Interface Name: GameDiscovery

Description: This is the interface for an object that listens for M2MIMud game objects to report themselves.

Functions:

- report(SessionAd) – This is called by a game unit to report its existence to this object. The SessionAd object contains the name of the session, the numbers of players in it, and the multihandle used to communicate with the session.

Interface Name: Game

Description: This is the most important interface, as this is a game. The members of a session implement this interface and use this interface to communicate their actions to each other.

Functions:

- yell(message) - Causes the message to be displayed on all members' units.
- joinSession(player) – Informs the unit that the player is joining the session.
- leaveSession(player) – Informs the unit that the player has left the session.
- notifyPlayerTimeout(playerId) – Informs the system that the player with the given id has timed out.
- processMove(MoveData) – Informs the system to process the move, done either by a player or a monster; in effect this updates the object's location on the game

board and informs the unit's player if he or she at either of the 2 locations (from and to).

- say(message) – Causes the message to be printed in the display if this unit's user is in the location specified in the message object.
- setPond(location, playerid) – Causes a pond to be placed location specified. The playerid is included so that if this unit's player is at the location, he or she will see a message.
- notifyTimePassage(newTime) – Informs this unit to update its time value to the specified value.
- warp(playerid, from, to) – Inform this unit to remove the player from the location and place them in the to location. This differs from a move in that to can be anywhere on the board.
- merchantBroadcast(merchantType, location) – Informs this unit to perform a merchant broadcast. What this does is refresh the appropriate merchant's timer and the prints out a message if this unit's player is at the specified location.
- addHouse(theHouse) – Informs the unit to set up a player owned house. The theHouse object contains the owner's id, permission settings, and location.
- registerAttacker(loc, theKey, playerId) – This informs the system to set the given playerid as in combat with the monster specified by the theKey value. This causes a message to be displayed if the user is in the same tile, and prevents the unit's player from attacking the monster.
- declineDuel(reason, pId) – Declines a duel for the given reason. The pId of the unit is included.
- printAttack(damage) – This is a two-fold function. It is called when the unit's player is in combat with another player. This function displays how much damage the other player does, as well as reduces the (unit's) player's hit points by that value
- requestDuel(pId) – Informs this unit that the player with the id "pId" has issued a duel challenge.
- acceptDuel(pId) – Informs the unit that player with the id has accepted the duel challenge.
- notifyFight(p1, p2, loc) – Registers that the players, p1 and p2, are now fighting each other at the given loc.
- notifymonsterDeath(loc, pId) – Informs this unit that the player with the given pId has killed his or her target. This causes the monster to be registered as dead on the unit and starts up the respawn timer.
- notifymonsterRespawn(key) – Informs this unit that the monster with the given key value has respawned at the location where it was killed.
- notifyPlayerDeath(id, loc) – Informs this unit that the player has been killed by

his or her target at the given location. This places the player back at location at (0,0). If the player was fighting a monster, this clears out the monster's target and restarts the movement timer.

- updateState(theState) – Causes this unit to update its state with the given state.
- refreshplayer(playerId) –Causes this unit to refresh the timeout timer of the player with the given id.

It may seem that the Game interface has quite a bit of work to do, but that is by design. The intent of this is that the object that implements it is the main controller of the entire system, which is responsible for using the other objects of the system to do the user's requests. Therefore it will have the most work to do. The desire for a main controller is because the operations of the system must be as close to atomic as is possible, so the goal is to have one object modifying the state, doing one thing at a time.

4.3) Interfaces: Normal

In addition to the M2MI interfaces of the system, there are normal Java interfaces. These are presented here to further describe how the system is organized.

Interface Name: CommandExec

Description: This interface is used for an object that can process the user's string command. The idea is that this object uses the Parser object located in the Command package to transform the user's input, regardless of where it comes from and creates a Command object, which it will then pass the to the GameSystem class's execute command for processing.

Functions

- executeCommand(command) – This takes in the user's command, generally in the form of a string, and executes it.

Interface Name: GameDiscoveryListener

Description: As mentioned in the section about M2MI interfaces, the report function, located in the GameDiscovery interface, is invoked periodically by all sessions to report their existence and transfer information. When that information changes, there needs to be a way to report that back to the system so that, if need be, the user is informed of any changes. That is what this interface is for, an object the GameDiscovery implementer can use to report back the M2MIMud when data about available sessions change.

Functions:

- newSessionAdded(theSession) – This is invoked when a new session is identified, which occurs when a session ad is reported that was not previously stored.
- sessionLeft(sessionName) – This is called when a registered session's timer timeouts before a session ad is reported. This means that the session has not

verified its existence, and, in all probability, no longer exists.

- sessionCountChanged(handle, count) – In addition to the name of the session, the numbers of players in the session is also reported. This value will change as players enter and leave a session. This function is called when that occurs.
- sessionStateChanged(handle, newState) – Each SessionAd object passed to the report object contains the full state of the session as perceived by the unit that generated the ad. This serves a twofold purpose: it gives units that are joining the session something to work with, and is used for state maintenance. When the state is changed, this needs to be reported back to the system to ensure it has the most current value of the state. This function does that.

Interface Name: GCWindowListener

Description: The "GCWindow" in the title of this class refers to the Game Choice Window, which is displayed when the user goes to find a game. This interface is for the listener of that window, to inform the system what session the user selected, if any.

Functions:

- sessionSelected(theAd) – This function is called when the user has either selected a session from the list or has hit the cancel button. A cancel is indicated by a null value for theAd.

Other Interfaces

Four other interfaces are used to indicate when a timed event has happened. These interfaces are:

- PlayerCacheListener, which is a listener for objects that need to be notified when a player has timed out and needs to be removed from the session.
- TimeListener, which is a listener for objects that are notified when the time changes.
- MerchantListener, which is a listener that waits for the merchant to indicate that it is time to perform a merchant broadcast, where the merchant "advertises" his or her wares.
- monsterListener, which is an interface that deals with when monsters move and respawn, as well as when they attack their target. (This last part is unique, as only the unit that is attacking the monster receives this notification.)

From these descriptions of the interfaces, a basis for the design of the system has been laid. The CommandExec is responsible for processing the user's input. The Game is responsible for executing the user's actions and reporting those actions to the session in general, if need be. The various other M2MI interfaces are responsible for finding other sessions and reporting them to the user. What follows now is a more detailed description of the design, including a presentation of the overall design as well as some of the class

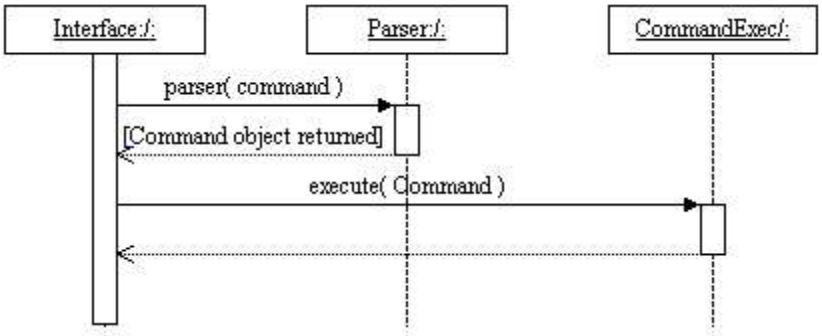
interactions.

4.4) Sequence Diagrams

This section describes some of the important class interactions of M2MIMud. Not all collaborations are presented, only the most interesting ones. A brief description of the sequence is given followed by the diagram itself, and then finally some comments. One thing to note: the printMessage invocations in several of these are omitted because they are not very useful in seeing the interactions.

4.4.1) Command Processing

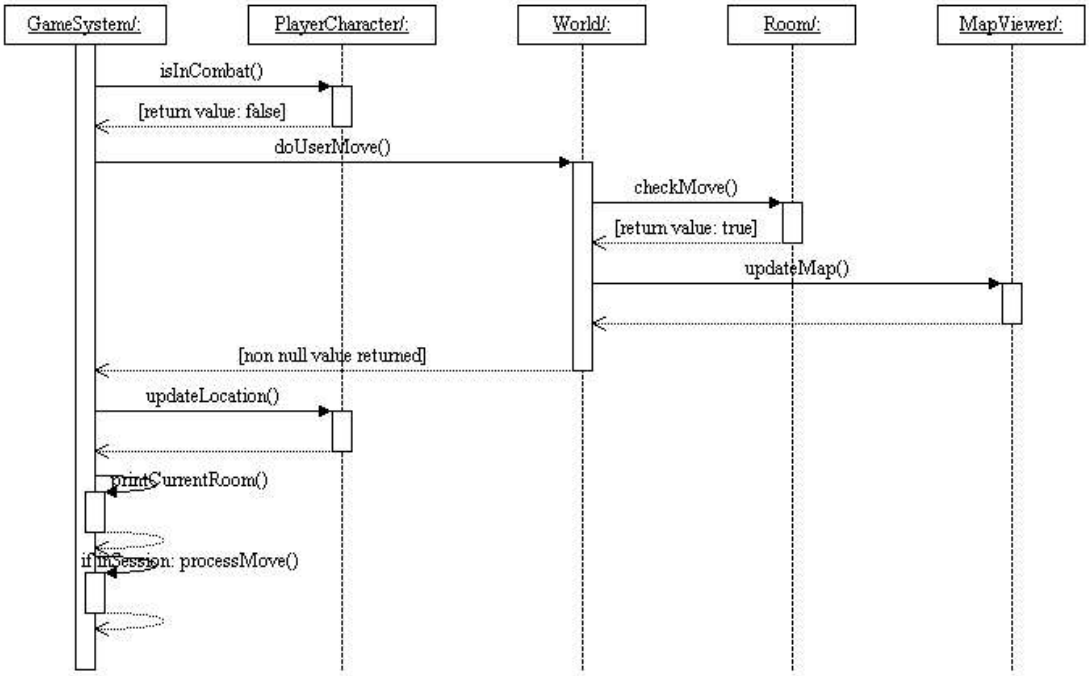
This is perhaps the most important collaboration diagram in the system. This is how a user enters in command into M2MIMud. It is the sequence of events that always occurs when the user performs an action.



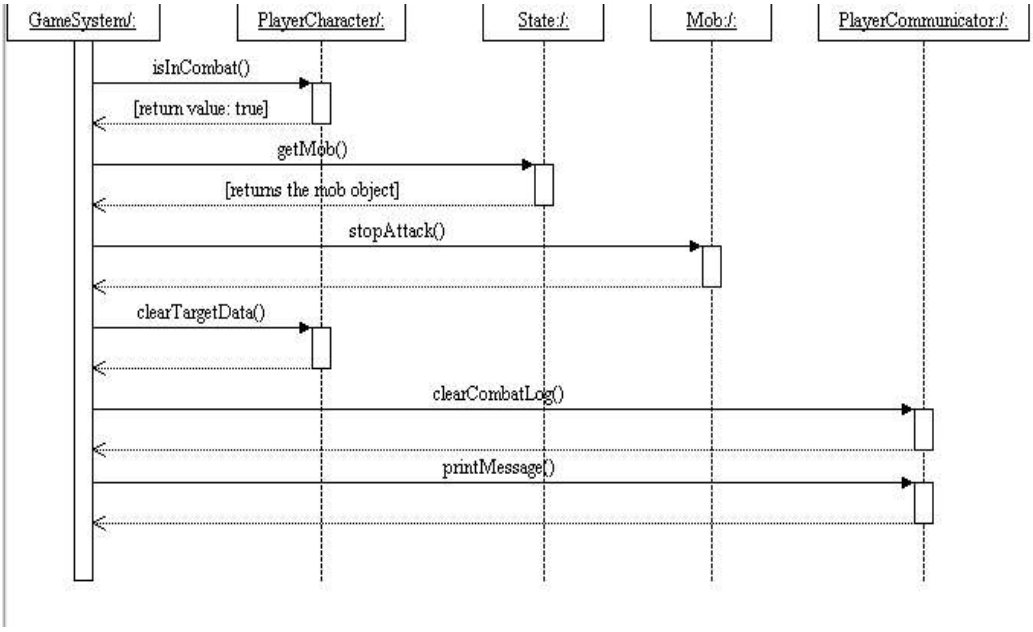
If the command is an invalid command, the GameSystem does not execute it, since that object expects all commands to be valid. This helps to reduce the amount of processing that object has to do.

4.4.2) Movement

This is one of the most basic actions in M2MIMud: moving around the map. It was also the first action implemented. Obviously the command processing sequence occurs first.



The above is a basic move. If the move is not valid or the user is fighting another player, an error message is printed out. A more complex example occurs when the user is fighting a monster, since the fight has to be stopped:

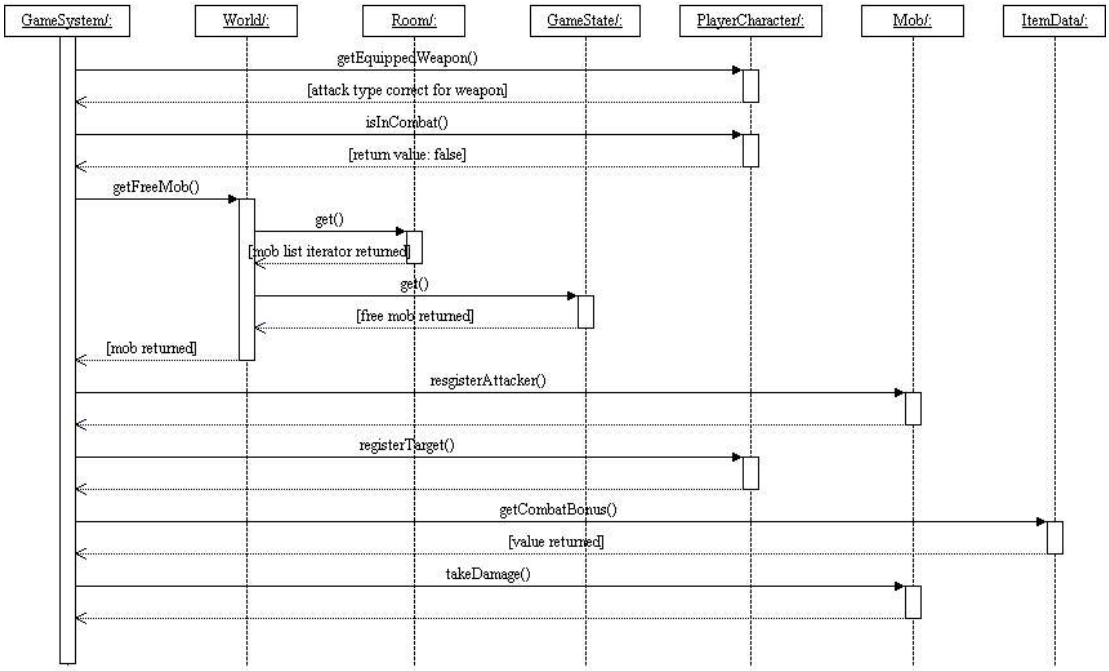


After this sequence of events the move is checked for validity and the system executes its action accordingly. It's important to note that all that the rooms contain, in terms of players, is the id of the players that are currently in the room. So, when a unit receives a command to update a player, all the world object does is remove the id from the "from" room and place it into the "to" room. However, units do not place their own user's id into a room, so this does not happen on the user's device.

4.4.3) Combat

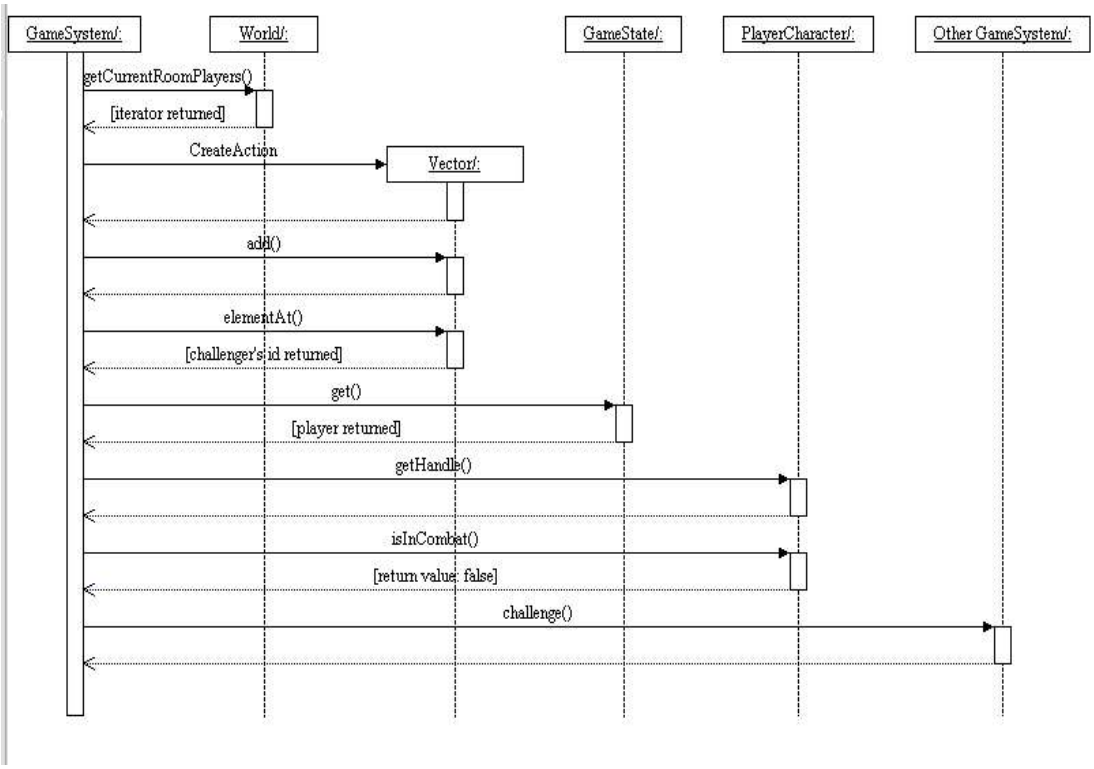
Another big aspect of M2MIMud is combat. These sequence diagrams show the events that happen to let a user fight.

The first is what happens when a user goes to attack a monster.

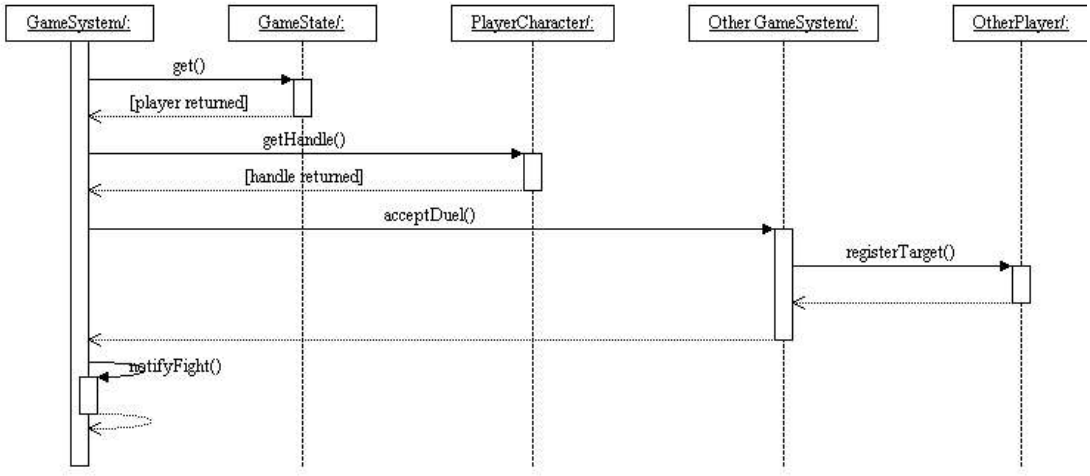


The first thing that is checked is to make sure that the user can perform the requested attack. Players are not allowed to slash or thrust with just their bare hands. After that, the system checks for a free monster, which is a monster that has the name given, is in the room, and is not under attack. Once that occurs, the player and the monster are placed into combat, and this is broadcast out to everyone. The final act of the system is to damage the monster and start some timers (this is not shown).

Player versus player has 2 steps: a challenge and an answer. This is the challenge diagram.



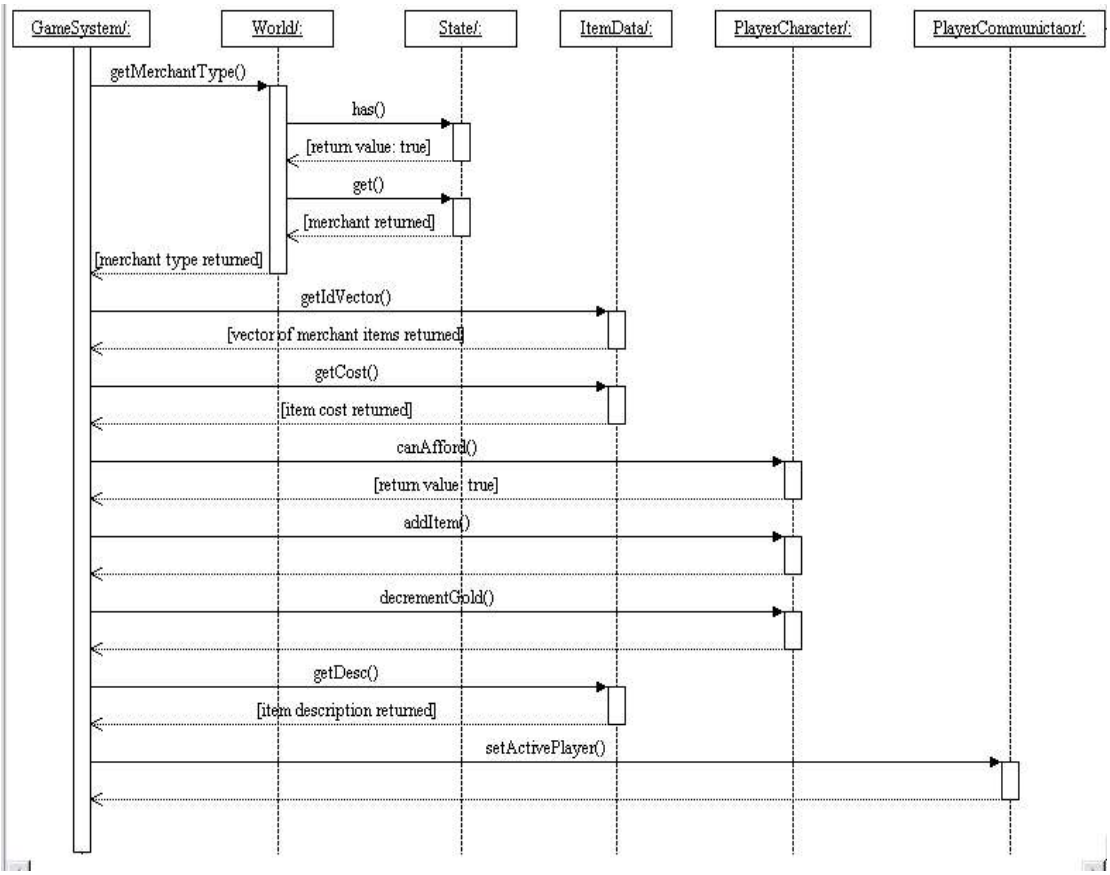
The system gets the name of the player the user wants to challenge, and then places the ids of all characters with that name into a vector. If that vector has a size of 0 or greater than 1, an error is reported back, since that means the target player is not in that room, or there are two or more players with that name present (minus the user, of course). If the user wants to challenge a person, that person must be the only player with that name in the room. If there is only one player of that name, the challenge is issued.



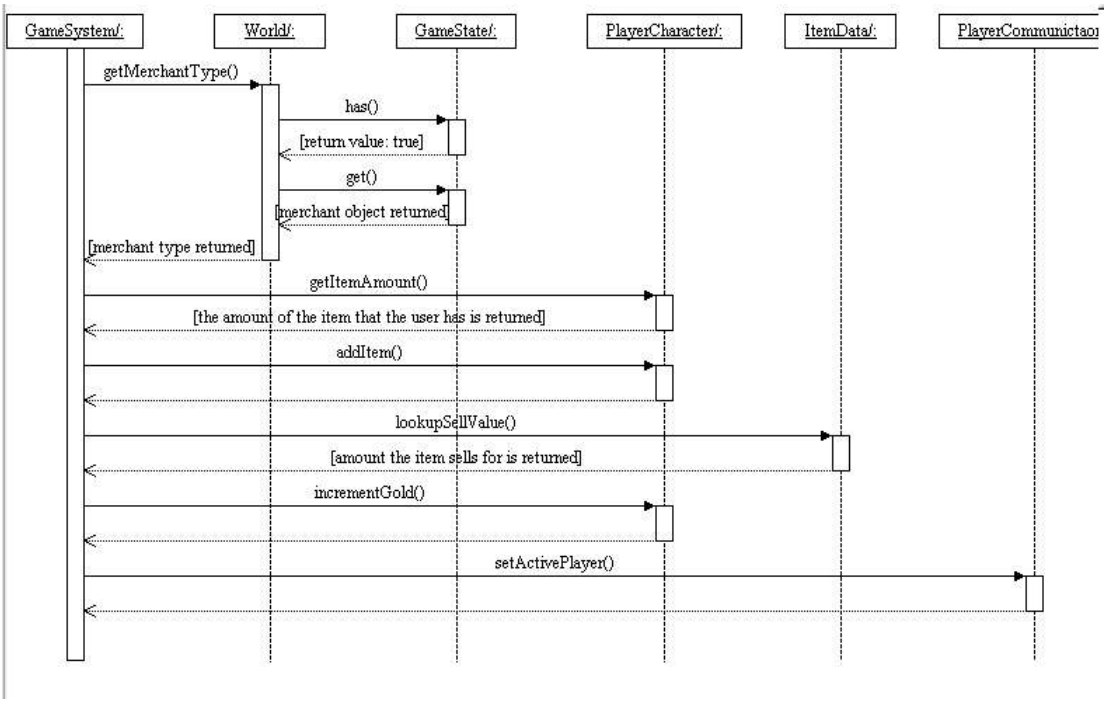
The notify fight command in this is called on the session, which makes the other unit pull out the players fighting and making the register the target.

4.4.4) Buying and Selling

The last major action that the system performs is the purchasing and selling of items.



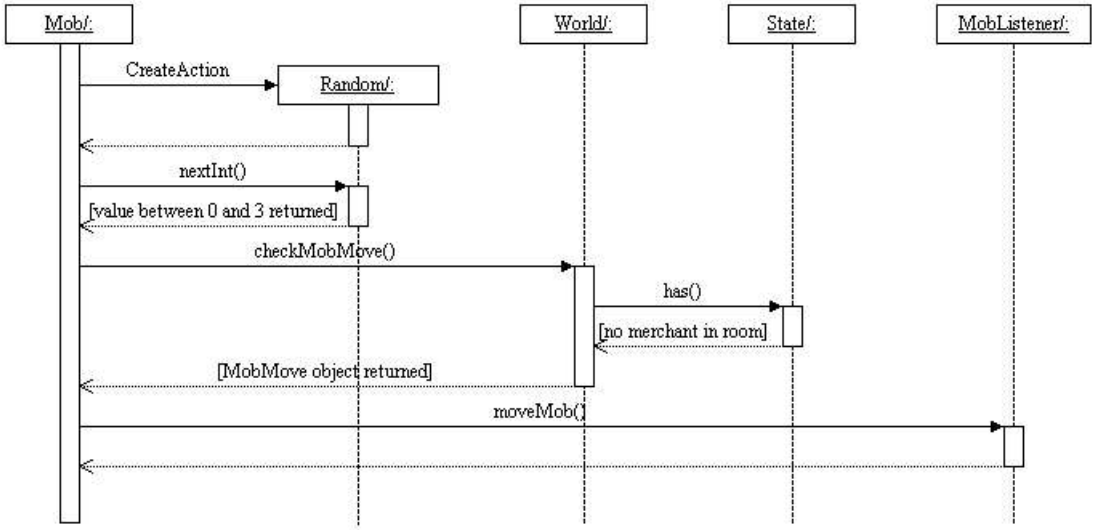
The ItemData class, which stores information about the items in the games, keeps track of what merchant sells what item. All the system has to do is pass the type of the merchant the user is buying from, and the ItemData class will return the items. Therefore, merchants do not have to keep track of the items they carry. This also uses the setActivePlayer function in the PlayerCommunicator class, which displays some information about the player. This is called because the user's gold is always displayed, and when they buy or sell something, it has to be updated with the new amount.



The merchant type is not important when the user is selling an item, however, it is needed here because a value of –1 means that there is no merchant in that tile. So it is called. Another note is the usage of the addItem function. One of the parameters of this function is an integer multiplier that is used to adjust the amount value stored in the Player object. So, a negative value passed into this function will decrement the amount of the items in the player's inventory. The reason the system needs to get the amount of the item that the user has is because it doesn't just blindly sell the item back. If the user specifies he or she wants to sell more of an item than they have, the system will sell as much of it as it can. So, of the user wants to sell 5 hats, but they only have 3, they will receive the amount of gold equivalent to 3 hats, and all hats will be removed from their inventory.

4.4.5) Monster Movement

Monster movement is not really a user action, being a timer-activated event. However, unlike other such events, which simple make a function call or set a variable, this one is a bit more involved.



The world has to perform some checking, as it needs to make sure that the direction specified is correct and that there are no merchants (the merchants set up a warding field that prevents monsters from approaching and attacking them). The direction is specified via an integer between 0 and 3, corresponding to the 4 points on a compass.

The actual move is simple, removing the monster's id from one room and placing it in another, and printing out s message as need be.

4.5) Design Comments

Perhaps the most striking thing about this is the fact that so much is done via the GameSystem. This is intentional. It's fairly important that the state remains consistent for the game unit, so the decision was made to basically have one thing touch the state. The GameSystem is the controller. For the most part the other objects hold state information. The GameSystem takes this information and uses it to perform the operation it needs to do. By having only object perform the actions, this reduces the amount of coordination. Of course, this has some drawbacks. The GameSystem class is large, since it implements the Game interface as well as the execute function. It is doing a lot of work, and thus is will be large. The other problem is that this creates a single point of failure. This isn't really a problem per se, since if the GameSystem fails, that means something went wrong and the user can't continue.

5.0) Implementation

This section describes some of the way M2MIMud actually does its work.

5.1) Object Interactions

Object interactions within M2MIMud are not particularly complex. For communication purposes, the GameSystem object that each unit has does all interactions. Messages are broadcast and received by this object, which keeps the rule that the GameSystem object is the central controller for the game. (In reality, however, it is actually a Game object, but since that is an interface that the GameSystem implements, all communications are done through that.)

There is one exception to this rule, and those deal with how a session is discovered. When the user enters the "find games" command to search for games, an object known as a SessionFinder is exported. This object listens for a session broadcast, and when it hears one, it reports to its listener (GameDiscoveryListener) what it is has found. The listener, in turn, reports back the GameChooserWindow and causes it to display the session information for the user to see.

For all other things, however, the communication interactions are done from GameSystem to GameSystem. When the user performs a move, for example, his GameSystem invokes the processMove() function on the session's multihandle. This, in turn, causes all the other GameSystems to perform this function and update where the user is in that world.

For command processing, the interactions are primarily done by the Parser and the Interface class objects. Through the MudClient window, the Interface obtains the user's command. It passes this to the Parser object, which returns back a Command object. If it is a valid command, this object is in turned passed to the GameSystem object for processing.

Some of the command are basic, and require nothing more than the GameSystem object to use the PlayerCommunicator object to print out a message. Some commands require minimal interaction. The look command, for example, requires the GameSystem command to use the World object to obtain a description of the user's current room and then display it. Still others require a massive amount of interaction among the various classes of the system. However, the GameSystem object does not actually work with many objects. The two primary objects it works with are the PlayerCharacter object and the World object to perform its work. These in turn call other objects to get their work done.

The one object that almost never calls other objects, other than its internal data members, is the GameState object. This is because this object is more of a repository object that stores data. Other object access it, but it itself very rarely makes calls to other objects of the system.

An example of a complex object interaction would be the move command. When the user requests this command, the MudClient passes the command string, "w east" to the Interface object. This, in turn, passes the string to the Parser object, which returns back a

Command object, which is then passed to the GameSystem. Upon receiving this command, the GameSystem asks the World to check if the direction listed is valid. The World object, in turn, asks the Room object that the user is currently in to see if it has an exit in that direction. If so, the GameSystem asks the World to move the player. The World does this by removing the user from the old room and then placing them into the new room. Once this is done, the player's location is updated in the GameState object by the World object. The GameSystem then requests the new room's description that it then prints out to the user. If the GameSystem is in a session, it will invoke the processMove function. This causes the other GameSystems to request their World objects to move the user and update his location. The other systems may or may not print out a message depending on the locations. Another thing to mention is that if the user was fighting a monster and decides to move to stop fighting, the GameSystems will also use the Mob object to stop the mob from attacking, clear out it target information and then restart its movement.

An example of a command where the GameSystem does not use the World object is when the user buys something from a merchant. In this case, the GameSystem uses the ItemData to determine the cost of the user's purchase as well as check if the item is valid for the merchant type in the same room as the player. If these are fine, it checks the player object to see if he or she has enough money. If so, it reduces the user's gold amount and modifies their inventory to reflect their purchase.

5.2) Language

The language of choice for M2MIMud is Java, because M2MI itself is written in Java.

5.3) Master Controller and Atomic Functions

By design, the GameSystem class is the master class for the entire system. The entire user input is executed by it, and it alone implements the functions for the Game interface. The desire is to have one single entity controlling the state of the system. Allowing only one object to change the state makes things simple to do. There is no need for it to wait for another object to finish its job and release any resources; rather, once the GameSystem is done with one task, it can go about doing another. However, for the GameSystem to keep the state as consistent as possible, it can't simply start one task, then start another before finishing it's first task. Therefore its functions should be atomic. To do this, the system employs a locking mechanism embodied in two functions: attemptLock() and unlock(), which use Java's wait() and notifyAll() functions (located in the Object class) along with a variable to try and lock the system. When an M2MI method is invoked, it locks the system, and once it is finished, it unlocks it. To help avoid race conditions and keep things in order, the Java keyword *synchronized* is employed.

5.4) Input Parsing

When the user's input is given to the system, it is transformed into a Command object and passed to the GameSystem object to control. This is done via the Parser object. This is accomplished via pattern matching. The user's input is compared to large number of

regular expression. When a matching expression is found, the user's command is parsed according to the command that expression embodied, and passed back the parser object, which gives that to the GameSystem to execute. All the parser does is to check the syntax of the command, it does not check the semantics of the command, as that is left to the GameSystem to look at.

5.5) Player Data

A player's data, the items they store, their stats, etc. are all stored on the device itself. Of course, this is really the only way to do store character data because there is not central server. To store the data, the PlayerCharacter implements the Externalizable interface. It is then written directly out to the disk of the system. Of course, such an approach to storing data has problems, and those are mentioned later. But suffice to say, in a system like this, there really is no other way to do this.

5.6) Item and Monster Data

Much like player data, the information about monsters and items are also stored on the game unit itself. However, unlike the player data, the objects aren't written out in object form. Rather, the information is stored in a special database-like object. These are the MonsterData and ItemData classes. Essentially, these are objects that have several hash maps in which entries are keyed according to the item or monster they belong to.

When a monster is created, its information is pulled directly from the data classes and a monster object is created. This is because although two monsters might be of the same type, they have information unique to them, like their location, number of hit points remaining, and their current target. So, each monster is a unique, independent object that pulls its initialization information from the same object. Items on the other hand, are not so. When a user has an item, he or she really has the id of the item. There is no "short sword" object so to speak. Rather the information about the item is pulled from the ItemData object as needed. This is done because an item is the same, regardless of who has it. Player A's short sword is the same as Player B's. The only difference lies in who owns it, but items do not store who has them. Rather it is up to the PlayerCharacter object to manage what items it has and how many it has. There is no need for independent item objects.

Of course, this may call into question how merchants operate. Merchants do not have an inventory of the items they sell, either objects or item ids. There is no need. All a merchant needs to know is where it is and what type it is. The ItemData class is capable of taking in a merchant type and returning back a list of items that merchant can sell. A player's inventory is really just a hash map where the key is the item id and the value is the amount of that item they hold. When a user buys or sells an item this value is adjusted accordingly.

For monsters that drop items, they store the id of item that they drop, and, upon death, the user gains that id in their inventory.

Again, such an approach intrinsically has issues with it. These will be discussed later in Section 8.

5.7) The World

In M2MIMud, each unit stores a copy of the world that it is playing in. The world itself is actually quite simple. It is nothing more than a hash map that contains Room object keyed by their location. Rooms store the ids of the players, monsters, and houses that are currently in, an integer code that tells what type of room it is, and whether or not it has a pond. It has no knowledge if it has a merchant in it. Merchants are actually stored in another hash map keyed by their location. When the user enters into a room, the location of the room is checked against the hash map to see if a value is keyed on that location. If so, the appropriate merchant description is appended to the description of the room. Rooms also contain functions to add and remove ids and ponds. When a change is made to a room, the World class gets the room from the location and calls the needed function.

5.8) Combat

Combat in M2MIMud is of two types: player versus monster (commonly referred to as PvE for Player vs. Environment) and player versus player (PvP). In terms of implementation, the two systems are the same. There are some games, most noticeably Mythic Entertainment's *Dark Age of Camelot*, which offers the 2 forms of combat but have different systems for both, but such a setup is far beyond the scope of this project.

For PvE combat, it's very simple. A player starts attacking a monster, which in turn, starts to fight back. After its initial attack, the monster starts a timer, which upon a timeout, the monster invokes the attack function on its listener. When the user does damage, that value is passed to the monster, and it's hit points are reduced. If the monster's hit points are reduced to zero or below, it is killed. Upon death, the monster begins another its respawn timer. While this timer is counting down, the monster does not move and cannot be attacked. Players will see the corpse of the monster if they enter the room where it is. When the timer times out, the monster's hit points are reset to their maximum value, and it begins moving again.

Of course, there are two other ways to end combat: the monster kills its attacker, or the player runs away from the monster. In both cases, the monster stops attacking, is healed back to full hit points and restarts its movement timer. If the attacker is killed, the player loses some experience and is placed back to location (0,0). A player's death is notified by a specific function, whereas a player running away from a monster is packaged into the movement command that is generated when the player moves. In both cases, the monster can be attacked again immediately. One thing to note about a PvE combat: after the initial broadcast that the player has entered into combat with a monster, there is no other broadcast about the fight until it is over. It is common in many online games to see messages like "Player A kicks the monster!" i.e. a play by play of the fight. Because this would cause a lot of network traffic, M2MIMud does not do this. Instead if a player enters a room when a fight is happening, they are shown a message informing them the player is fighting the monster.

Player versus player combat is a little different. There is no open PvP combat, which means a player can start attacking another player with no warning. Rather, to fight someone, a player must first challenge them. The challenged player can accept or decline. There are several ways a user can decline: they can move out of the room,

logout (or timeout), or start fighting a monster. If the user accepts, however, combat begins. Much like monsters, all players not involved in combat see if the fact that two players are fighting each other, no play-by-play messages are sent. For the players involved in the combat, however, they see when they attack and when their opponent attacks them. These messages are transmitted directly to the players. M2MIMud has a relatively simple and straightforward process for doing damage. A weapon and armor both have combat bonuses. The amount of damage a weapon does is that bonus, and the amount that piece of armor protects from is that bonus. So, it a sword has a bonus value of 9, it will take off 9 hit points off a player with no armor equipped. If a piece of armor has a bonus of 4, it will protect the player from 4 points of damage. So, if the player uses the sword with a bonus of 9 against a player wearing armor with a bonus of 4, they will hit the player for 9 – 4 = 5 hit points of damage. When a player attacks, the base damage value is sent to the other player's unit, which calculates how much total damage is done and sends that back to the other unit for display purposes.

Once a duel has started it cannot end until one player dies (or time outs), or surrenders using the surrender command. Players may not leave the room they are fighting in. Once a player is killed, the player is warped back to location (0,0) and resurrected. Other units are notified when one player kills another player. One final note: only one duel challenge may be issued to a player. If player A issues a challenge to player B, and then player C tries to challenge player B, C will receive an error message.

5.9) States and Player

In order to maintain the state of a session, each unit periodically broadcasts out its version of the state embedded into a session ad. In addition to helping to maintain states, these ads serve another purpose; they give units that are joining the session a state to work with. When a unit joins a session, it merges its state information with that of the current session ad it has, provided, of course, it shares the same map as the session. If not, it simply overwrites it current state with that of the session. If, however, the unit that is joining has the same map as that of the session it is joining, that means that its states can be merged with that of the session. In this case, the joining unit tells all members of the state to update their state information with its state information. The session members merge their information with that of the incoming state, while the incoming state merges its information with that of the session's state. Essentially, what this does is to add the unit's player to the session as well as any ponds or houses he has that the session does not already have.

The maintenance of the state is straightforward. When a unit receives an ad for the session it is in, it checks to see if its state is equal to that of the ad. If not, it merges its information with that of the ad. For houses and ponds, all it does is make sure that the locations are the same. Monsters and players are a bit harder. If a monster is not in combat, all it needs to do is to make sure the locations of the monsters are the same. However, if they are in combat, there is a problem. However, it should be stated first that the only thing that can change the information of a player is the session ad from the unit that owns that player. What this means is that the only (other) player the unit checks in the session ad's state is that of the unit which sent the ad (the ad also has the id of the player it's unit owns). The unit also compares its local copy of its player with the copy in

the session ad. If they are different, it sets a flag to do an emergency broadcast (i.e. it broadcasts out its ad a significantly shorter interval than normal). However, in terms of other players, it only checks the player for the unit that sent the ad. From a design standpoint, this enforces the rule that the only thing that can change the state of a player is the unit that owns it. From an implementation standpoint, this reduces the amount of work that a unit has to when resolving state differences. There are 2 exceptions to this rule. In the case where the session ad has a player that the unit does not, this player is added and placed where the session's ad claims it is. The idea is to get the player registered into the unit. The second exception deals with mob combat. Here, whom ever the mob says it is fighting (if its fighting) is placed into combat with it, regardless of whom the session ad originated from.

The biggest issue in the state resolution mechanism revolves around combat. This is because there are so many different things that can go wrong. First and foremost, the ad is given the benefit of the doubt. So if the session's ad state that a monster is not in combat, but the local copy of the state says it is, the local copy's monster is removed from combat. If the local copy of the monster is not in combat, but the session's state is, than the monster is placed into combat with the target. If the monsters are in combat with two different targets, that's deferred to the section that handles players, where timestamps are employed to take care of it.

When a player begins attacking a monster, the action is time stamped. Each unit that receives the attack information also receives the timestamp value of when that monster started its current battle. This is done for the sake of the state resolution. Specifically, this is done for the case when two players are listed as fighting the same monster. When this occurs, the timestamps are checked, and the one with earliest timestamp wins. So if player A and player B are registered as fighting the same monster, but player A started fighting before player B did, than player A wins and is set to be in combat with the monster. If the timestamps are the same, combat is broken off.

PvP combat is different, and a bit harder to deal with. Again, the thing to keep in mind is the fact that the state resolution function only deals with two players: the unit's player and the player of the unit who sent out the session ad. This is in conjunction with the rule that only the game unit that controls the player can make changes to that player. As with the majority of the state resolution process, the benefit of the doubt is given to the incoming session ad.

What this means, is that unless it deals in some fashion with the unit's player, the data assumed in the session ad is correct in regards to the activity of its player. (The logic is that this ad is directly from the unit that owns the player, which is the only thing that can be trusted.) Therefore, if the session ad claims that its player is fighting another player, then the local copy of the session ad player is updated. If it claims that the player is not fighting, that's changed as well. (For players not in combat, they are simply warped to wherever the ad says they are.) However, things get complicated when the session ad claims that its player is doing something that affects the unit's player. In this case, the unit is given a higher priority. There are two cases to consider. The first is when the player claims to be fighting the same player that the unit's player is fighting. In this case, the command is ignored. The idea here is that a partition happened, and there's no way to verify what's what, so the unit will assume the last thing is correct. The other case is

when the session's ad player is the player that my unit believes it is fighting; yet the session ad says it is no longer fighting the player. In this case, the unit's player is removed from combat with the session ad's player, and the session ad's player is changed to fighting whomever he says he is. (If the player is not found, it is pulled from the session ad's version of the player cache.) The reasoning for this course of action deals with the fact that the only unit that can really be trusted when it comes to determining the state of a player is the unit that controls it. So, when a unit claims it is fighting someone, any unit processing that information has no choice but to trust it (so long as it does not cause their player to go into combat).

Player themselves are stored in what is known as the player cache, which is an object which exists outside of the state of world. If the state needs information about the player, it will pass the player's id to the player cache and get back the needed information. The player cache is essentially two hash maps, both keyed on the id of the player; one hash map holds the actual character data, whereas the other one holds the timer that goes off when the player timeouts. To prevent a timeout, a player has a heart beat object that periodically broadcast out the facts that is player is still active in the session. A player can only timeout if a partition occurs and is not healed with a certain amount of time.

6.0) User Manual

6.1) Startup

The first step to running M2MIMud is to install and run M2MI itself. Details of this can be found at the M2MI website, located at http://www.cs.rit.edu/~anhinga. Once this has been installed, the user must unjar the Game.jar file. This archive contains the 5 directories. To play the game, the user must enter into the M2MIMud directory, and execute the command "java M2MIMud".

6.2) Character Creation

Upon loading up M2MIMud, the user will be presented with the welcoming screen, which states that they must first load or create a character. If this is the user's first time playing, they must execute the "create" command, and specify a gender, name, and class for their character. If they have a previously saved character, they may execute a load command, giving the name of the character, which will load a previously saved character. Once a player is loaded, they will place into the default world, which is a 100 by 100 grid of grass tiles.

6.3) Worlds

The area in which a player plays is referred to as a world. A world has two components to it: a layout and a state. A layout is the map of the world, whereas the state is the location of all the various entities within the world. As such, a player is allowed to load up either a layout or a state.

6.4) Layout

The layout (or configuration) it the map of the world, it essentially tells the game what the tiles are. Each tile may be flagged as grass, trees, or fields. Users may create and use any layout they wish to use. To do this, a utility program is included with M2MIMud. This is the MapCreation program. To use this, users must first define what their world looks in a configuration file. This file takes the following form:

- Map Name
- Dimensions: x y
- Tile specifications

33 The first line of the file is name of the map to be generated. The second line is the dimension of the world. The x value indicates the length of the world, whereas y indicates the height of the world; in other worlds, they will make the world an x by y grid. The remainder of the file is the tile specifications. By default, each tile is marked as a grass tile. However, users can mark them as either water or woods tiles. This is done using the directives enclosing two sets of coordinates in a water or wood directive. This will then cause the system to mark all tiles within the range of those coordinates as being marked as the specified tile type. For example, say the user has the follow configuration file:

playtest

5 5

water

0 0 2 2

water

wood

3 3 4 4

wood

Example configuration file

This will create the following file:

playtest

5 5

g g g t t

g g g t t

w w w g g

w w w g g

w w w g g

Contents of playtest.map, the file generated from the example configuration file

When the user loads this map into the game using the load command, it will recreate the world they are currently into one which reflects this world. The state of this world will

be empty, which means it has no ponds or houses in it.

The purpose of this system is mainly to give users a visual cue as to how their map is setup on a large scale. This allows them to tweak their world as they wish, so that they may design a world that they want to play.

Loading a map is done via a load command, where the user enters in the command “load map <map-name>”. This will load the map into the game and reconfigure the world to use this map. Also, the state of the system is cleared, so a user will start out in an empty world. If the user wishes to save the state they are currently in, they must save it beforehand.

6.5) States

The second component to the world is the state, which has information specific to this world, such as the location of the merchants, ponds, houses, monsters, and the players. Much like a map, a state can save and loaded. When a player is first loaded in to the game, they are placed into the default world, and its associated state. The default layout is 100 by 100 grid of grass tiles. It is a valid layout, so the world associated with it is valid as well. As a result it has a state that can be modified. This state is referred to as the default state. This is a special state, one that always exists. If deleted, the system will regenerate it. Its layout cannot be changed. The default state serves as a common basis for all the units, as it is hard coded into the system.

States can be loaded and saved, much like a map. The rules are as follows. When a new layout is loaded, the resulting state is empty. It also has no name. In order to save a state, a name must first be specified. This is done via the “save world <name>” command. A state cannot be given the same name as a preexisting state. For example, if a user wishes to save a state as “water” but already has a (different) state named “water”, he must first delete or rename the old state and then save the other one. The same state may be saved under multiple names, by simply executing the save command specified. Once a state has been given a name, it can be saved by using the “save” command which will overwrite the existing state.

If the user is not in a session, they are in single player mode. In a single player mode, there are no monsters or merchants in the world. Players can add and remove ponds from any tile they wish (except water tiles). They may also create one player owned house. If a player does not like their house's location, the player must first remove it from its current location and then place it at the new location. Players may also remove any other player’s house that they may have obtained as a result of joining a session. Also a player can only save their world when they are in single player mode. Once in a session, a player may not save until they have left. While in a session, a player may add any pond and set their house down, but they may not remove ponds or change the location of their house. These can only be done in a single player mode. However, players can fight monsters and each other, as well as interact with merchants.

6.6) Characters

When the game is first loaded, the player must choose a character to play. However, this is not set in stone. At any time in single player mode, a player may create a new character or load another one. Their current character is automatically saved, and the new character is load or created. While in a session, however, players may not change their characters. There is no limit on how many characters a player may have (outside of hardware limitations). Information about how to create and load characters can be found in the command list section.

6.7) Creating a Session

Once the character has been selected, and the game is using the desired state, the user can create a session, which will spawn the monsters and the merchants. These will also others to join the world and play in it. Executing the "game <name> command does this.

6.8) Joining a Session

Once the character has been loaded, the player may find any session by typing in the find command. Once this happens, the main window will become disabled and the session finder window will appear. Eventually, sessions will appear on the screen as well as how many players are currently in them. The user then highlights the session and then pushes the join button to join the session. They will join the session.

Some important things to note, the user must make sure to save their previous state before joining a new one. This is because there is no guarantee than the session they are joining is using a compatible state. If the user is using the same layout as the session, the system will merge the user's state information with the session's state information. The session will then get any houses and ponds the user had. However, if the session is not using the same layout, than the user's state will be overwritten by the session's state. No attempts to save the state will be made. In other words, if the maps are the same, than the users and session's state information will be saved, otherwise, the session's state information will overwrite the user's state information. Therefore, it is recommended that users always save their state before joining a session if they wish to keep their state.

Once in a state, a user may interact with other players, perform transactions at the merchants, and fight monsters. One thing to note, each session has a layout associated with it. At anytime, a user may save this map using the "save map <name>" command. However, this will only work if there is no other map of the same name. This will allow user to store map of the session they are in without having to save the session's state information.

6.9) Leaving a Session

To leave a session, a user must not be in combat with monster or a player. They must then execute the leave command. When a player leaves, his character is removed from the session. However, only his or her character is removed. Their state information (i.e., house and ponds) is not removed. On the player's side, they, too retain all state information they acquired from the session. Obviously all other players are removed

from their unit. However, merchant and monster information is also purged. Effectively, the player is in single player mode, with one major difference: they are not allowed to create a new session, nor can they find other sessions to join. Once a player has left a session, they must completely shut down M2MIMud and restart if before attempting to deal with sessions again. However, before they shut down, they can modify the state they are in and save it for future use.

6.10) Communications

One of the incentives to play on-line games is the ability to interact with other players. M2MIMud offers three forms of communications to allow players to talk with each other. The first form of communication is the basic "say" message. This is accomplished by typing in "s <message>". This will cause the message to print in the display of all players who are in the same tile as the player. The next form of communication is the yell. This has the same format as say, except that "s" is a "y". This will result the message to be displayed to all members of the session. The third and final form of communication is the send message. This is a private message, and the message will only display on the player the user indicated (the message will also display on the user's window as well).

6.11) Movement

Movement in M2MIMud is done on a room-to-room basis. Players move from one room to another via the "walk" command ("w" for short). When the user enters into a room, they will see a message telling what directions are valid exits for the room. To move in that direction, the user must type in "walk <direction>". This will move them in that direction and display the contents of the room. At any time, the user may type in the "map" command to open a window that will display a text map. This map will display the current coordinates of the user, as well as the tile type for all the rooms surrounding the user's location. One this map "G" means grass, "T" means trees, "W" means water, and "X" means a tile the user may not enter.

6.12) Transactions

Buying and selling are an important part of any RPG, and M2MIMud is no exception. Merchants can be found only in a session, as they do not appear in single player mode. There are three types of merchants, item, weapon, and armor. They can be recognized by their description. These merchants provide wares to players as well a place to sell off unwanted items for cash. To see what a merchant has to sell, a player must be in the same tile as them, and type in the command "list wares". This will display what the merchant has for sale.

To purchase an item, the user must be in a tile that contains the appropriate merchant for the item the user wishes to buy. To purchase an item, the user must specify 2 things in addition to the fact that they want to buy an item. They must specify the item id and the quantity. When the user executes the "list wares" command, they will see something very similar to the following:

healPot Potion of minor healing 80g

The first characters are the item's id, the second is the item description, and the third is the cost of the item. For the user to, for example, buy 3 potions, they would execute the command "buy healPot 3". If the user has at least 240g, they will purchase the item, and 3 potions will be added to their inventory. If they do not have the necessary amount of gold, they will receive a warning message, and they will not buy anything. Selling an item is basically the same. However, the user does not need to be on a specific merchant tile to sell an item. All merchants will buy any items the user does not wish to hold on to. Much like selling an item, the user must specify the item id and the amount of the item they wish to buy. To sell back the three potions the user purchased above, they would execute the command "sell healPot 3". If they user does not have the amount they specify, the system will sell back the greatest amount it can. So, if by accident the user types in "sell healPot 4", they will only receive cash for three, and the items will be removed from their inventory. Items do not sell back for the exact amount they were purchased for.

6.13) Inventory and Equipment

When a user purchases an item, it is placed into their inventory. Conversely, when a user sells an item, it is sold from their inventory. To determine what a user has in his or her inventory, they can execute the "inventory" command to get a listing of what items they have in their inventory. When they do this, they will see the id of the item, its description, and the amount of the item they have.

Equipment refers the items the user can equip. These take two forms: armor and weapons. Weapons increase the amount of melee damage a user does, as well as enable the slash and thrust commands. Armor, on the other hand reduces the amount of damage a user takes from a melee attack. Both types of these items can be purchased from a merchant. To equip an item, the user must execute the "equip" command, specifying with it the id of the item they wish to equip. If they cannot equip the item, they are informed of this. In M2MIMud, there are three slots that can be equipped. These are the body, one hand, and 2-hand slot. If the user has an item in a slot, and they execute the equip command, that item is removed and placed back into their inventory. This also happens if the user equips a different kind of weapon. If the user, for example, is wielding a long sword, which is a one handed weapon, and then purchases and equips a bastard sword, which is a two handed sword (and two hand only), then the long sword is unequipped and placed back into the user's inventory. Only items that are in the user's inventory can be sold.

Some items can be used, like potions. To use an item, the command is "use <item id>". The item will be used and removed from the player's inventory.

6.14) Houses and Ponds

The worlds are not static, as the users are allowed to change them. The two things that a user can do in a tile is to dig a pond or set up a house. Ponds can only be placed in grass or wood tiles, and there can only be one pond per tile. In order to dig a pond, a player will execute the "dig pond" command, which will they setup a pond in that tile.

To create a house, the user must execute the create house command, which takes the form "createhouse <desc>". A user may only have one house a session. If they leave a session and come back, their house will still be there. In the event that a user's layout is the same as the session they are joining, and the system is merging his or her state with that of the sessions, than any houses for players that the state does not have are added. For example, if player B has a house by player D, and is joining a session with players A and C, and their session layouts are identical, players A and C will gain player D's house in their state. The only exception to this rule is the player's own house. If the player has a house in both the state's session and their personal session, but the locations are different, then the house will be placed at the location in the player's state. For example, say player A has joined a session, and placed his house at location (1,0). He then leaves the session, and moves his houses to location (3,3); if he rejoins the session, than all players will now see his house at (3,3) rather than (1,0). Note that once as house is placed down while in a session, it cannot be removed. In single player mode, the user may remove any houses they wish to, and move their house around. While the user is in a session, however, they may not remove any houses, even their own.

To enter into a house, a user must type in the "enter house <id>" command, where <id> is the id of the player who owns the house. While in a house, a user may see the description of the house, as well as the any other players in that house. They may not fight or move, though, until they leave the house.

6.15) Combat

Combat in M2MIMud takes two forms: a player versus a monster and a player versus another player. Monsters in M2MIMud are non aggressive; they will not fight unless they are attacked. They will simply move from tile to tile. To attack a monster, the player must be in the same tile as the monster they wish to fight, and then execute an attack command, specifying the type of attack they wish to perform as well as the name of the monster. Without a weapon equipped, a player may only "kick" or "punch" a monster, with one the player may also "slash" and "thrust" a monster. Once a monster has been engaged, a player only needs to execute the non-target version of those command, i.e. they do not need to specify the name of the monster, since the system already knows what they are fighting. If the user is a mage, they may choose to cast spells on the monster instead of fighting it with a weapon. To cast a spell, a user must have the spell's scroll in their inventory, and they must be of the right level. Spell scrolls can be purchased from a weapons merchant. Once the use has the spell they wish to cast, they type in "cast <spell> <target>", which casts a spell on the target. Once this has been done, the user simply types in "cast <spell>" to cast again. Like weapons, all spells have a cool down timer, during which the user may not move or attack again. Once this period is over, a user may choose to melee or cast another spell on their target.

A fight between a player and a monster is one to one: a player can only fight one monster at a time, and a monster can attack only one player at a time. Attempting to fight a monster that is already engaged in combat with another player will result in an error message. If the player kills the monster, he or she is award some experience, gold, and any loot the monster may have been holding on to. If they are killed however, they lose experience and are warped back to location (0,0). The monster, on the other hand, is

restored back to full health, and begins to move around the map again, able to be attacked by other players. If a player is losing to a monster and they want to run away, they simply need to move one tile away. The monster will not chase them; rather, it will disengage combat, be restored to full health, and begin roaming again.

Player versus player (PvP) combat is different. In M2MIMud, there is no open PvP combat. To fight another player, the user must challenge that person using the challenge command. This is done by the command "challenge <user name>". (If there is more than one person of the specified name, the challenger must ask the player he or she wishes to fight to move to another tile.) Once the command has been issued, several things go into play. Only one challenge can be done at a time. If player A challenges player B, then player C may not challenge either A or B. The challenger may not be fighting a monster until the challenge is resolved. If either player leaves the room or leaves the game, the challenge is cancelled. Also, the challenged player may also decline the challenge. However, if the player accepts the challenge, the duel begins. Neither player may leave the room where they are fighting. The duel will end once one player kills or surrenders to the other player. Players are also not allowed to switch weapons mid duel, they must fight the person with the gear they had at the beginning of the duel. Much like fighting a monster, once another one player kills a player, he or she is warped back to location (0,0), and is fully refreshed. However, the individual who won is not healed back to full, and must obtain a potion to heal. There's an important note to make here. Once a player is killed, the fight has ended. If the defeated player wishes to fight again, the player must challenge the other player that defeated them in order to fight again.

Regardless of the type of combat, a player cannot just simply type in the attack command over and over. Each weapon has a delay associated with it. After an attack has been executed, there is a set amount of time during which a player may not attack. For example, a long sword has a delay of 3.3 seconds. After the player attacks with this weapon, he or she must wait 3.3 seconds to attack again. This is to simulate the fact that after using a weapon, a person must reposition himself or herself to use the weapon again.

6.16) Friend's Listing and the Lookup List

Unlike typical MUDs, M2MIMud does not prevent multiple players from choosing the same name. This can present problems to players, since it can be hard to determine who is who. Therefore, there are some tools in place designed to help users select the right player.

First and foremost is the lookup list. Certain commands require the user to specify another user. If there are more than one players of that name, they may receive a message telling them to perform a lookup. (The only exception to this is the challenge command.) To do this, the user must type in the command "lookup <name>". This will search through the players, and then list all players of the name into a listing, along with their unique player id. Once this command is executed, the user must execute the l-version of the command, along with the index number of the player on the list. For more information, check the command version.

The second tool available to players is friend's listing. This tool allows players to keep track of certain other players. Friends are added via the "add" command (this may be a lookup command). Once a player's friend is added to the listing, they will be informed when the player enters and leaves the session. They can also execute a "check friends" command to see who is in the session with them. The purpose of the friend's listing is to allow a user to keep track of certain players so they know that the "Delmania" that is in the room with is indeed the player they think it is.

6.17) Command List

The following is a listing of the commands available in M2MIMud and what their function is:

create male|female <name> <class>: This will create and save a level 1 character with the given <name> of the specified <class>, and set to the sex specified. Current valid values for <class> are "fighter" and "mage".

walk/w <direction>: Attempts to move the player in the <direction> specified.

quit/q: Shut down M2MIMud. Cannot be used while in a session.

yell/y <message>: Prints the <message> to all players in the session.

look/lo: Prints out the contents of the current room

list/li: Prints out some information (name, level, class, and location) of the player.

game <name>: Create a game with the given <name>.

find: Displays the find game window so that a user may join a session.

leave: Leaves the session that the user is in.

who: Displays character information for all players in the session

say/s <message>: Prints the <message> on the display of all the players in the same room as the user.

dig pond: Attempts to dig a pond in the tile that the user is in.

createhouse <desc>: Creates the user's house in the given tile. The <desc> is the description the users will see when they enter a house.

time: Prints the current time of day it is.

list wares: Causes a merchant to print out the item they are selling

buy <id> <quantity>: Attempts to buy <quantity> of the item with the given <id>.

sell <id> <quantity>: Attempts to sell <quantity> of the item with the given <id>.

inventory/inv: Prints out the user's inventory.

send <name> <message>: Sends the user with the given <name> the <message>.

add <name>: Adds the player with the <name> to the user's friends' listing.

lookup <name>: Performs a lookup on the <name>

check friends: Prints out if the players in the player's friends' listing are in the same session as the player.

llist: Prints the results of the lookup

lsend <index> <message>: Sends the player at position <index>s on the lookup listing the given message.
For example if the lookup listing has the following:

(0) Crystal 12

(1) Crystal 13 Then "lsend 1 hello" will send the message "hello" to the Crystal with the id of 13.

ladd <index>: Adds the player at the <index> position of the lookup listing to the player's friends' listing.

lcheck <index>: Determines if the player at the given <index> of the lookup listing is in the player's friends' listing.

remove <index>: Removes the player at the given <index> from the player's friends' listing. Note: <index> here refers to the index on the friends' listing, not the lookup listing.

equip <id>: Attempts to equip the item with the given <id>.

punch/kick/slash/kick/pu/ki/sl/th <name>: Performs the specified attack on the monster of the given <name>. Note that slash and thrust may only be performed with a weapon equipped.

punch/kick/slash/kick/pu/ki/sl/th: Performs the attack on the player's current target

accept: If the player has been issued a challenge, this will accept it.

decline: If the player has been issued a challenge, this will decline it.

challenge <name>: Challenges the player with the <name> to a duel.

map: Opens the map window.

use <id>: Attempts to use the item with the given <id>. The user must have the item in their inventory.

load world/player/map <name>: Attempts to load the given entity with the given <name> . What happens is as follows:

- World: Loads the state with the given name. The layout of the world may or may not change depending on the layout used in the world loaded.
- Player: Loads the player of the name into the current state.
- Map: Clears the current, and reconfigures the world to use the layout found in the file.

save world/map <name>: Saves object given the specified <name> world: This saves the state of the world, as well as the layout. The name give appears at the top of the screen

map: This saves the layout of the world into a .map file of the given name. This can then be used to create a new world of the given map with no state information.

Note: If there exists either a map or state file with the same name, the system will not save the data.

save: This is a quick save, so to speak. Once a "save world <name>" command has been executed, the user can use this to save the state information to the file they previously specified. So, of the user had executed "save state water", which creates a file named "water.dat" in the states directory, executing a "save" command will overwrite that file.

remove pond: This removes the pond from the room. Cannot be executed while in a session.

remove house <index>: When the contents of a room a displayed, a list of all the houses in the room is displayed, each with an index number next to them. Executing this command will remove the house at the specified index. For example, if the room has the following houses:

(0) Crystal's house (13)

(1) Crystal's house (14)

Executing "remove house 1" will remove the house of the player with id 14.

surrender: If the player is in a battle with another player, this will immediately end the battle.

cast <spell> <target>: This will attempt to cast the <spell> on the player's target. This can only be used to initiate combat with a monster. If the player is already engaged with a target, they must use the non-target version of this command.

cast <spell>: This is used to cast a spell on the user's current target

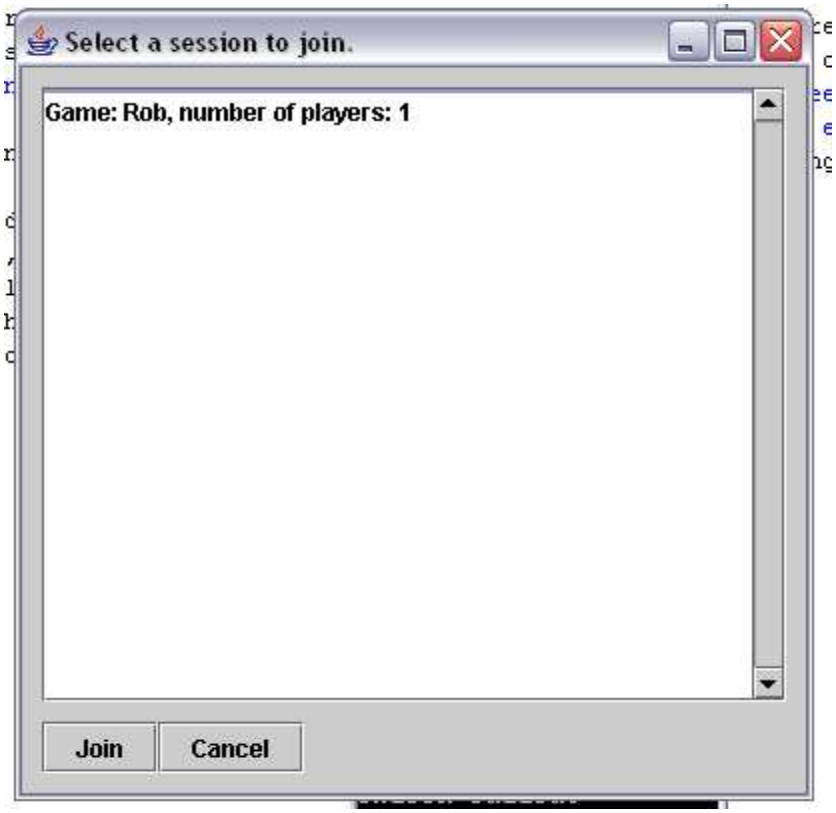
enter house <id>: This command is used to enter into a house who's owner has the given id.

leave house: If the player is in a house, this will cause them to exit the house.

7.0) Windows

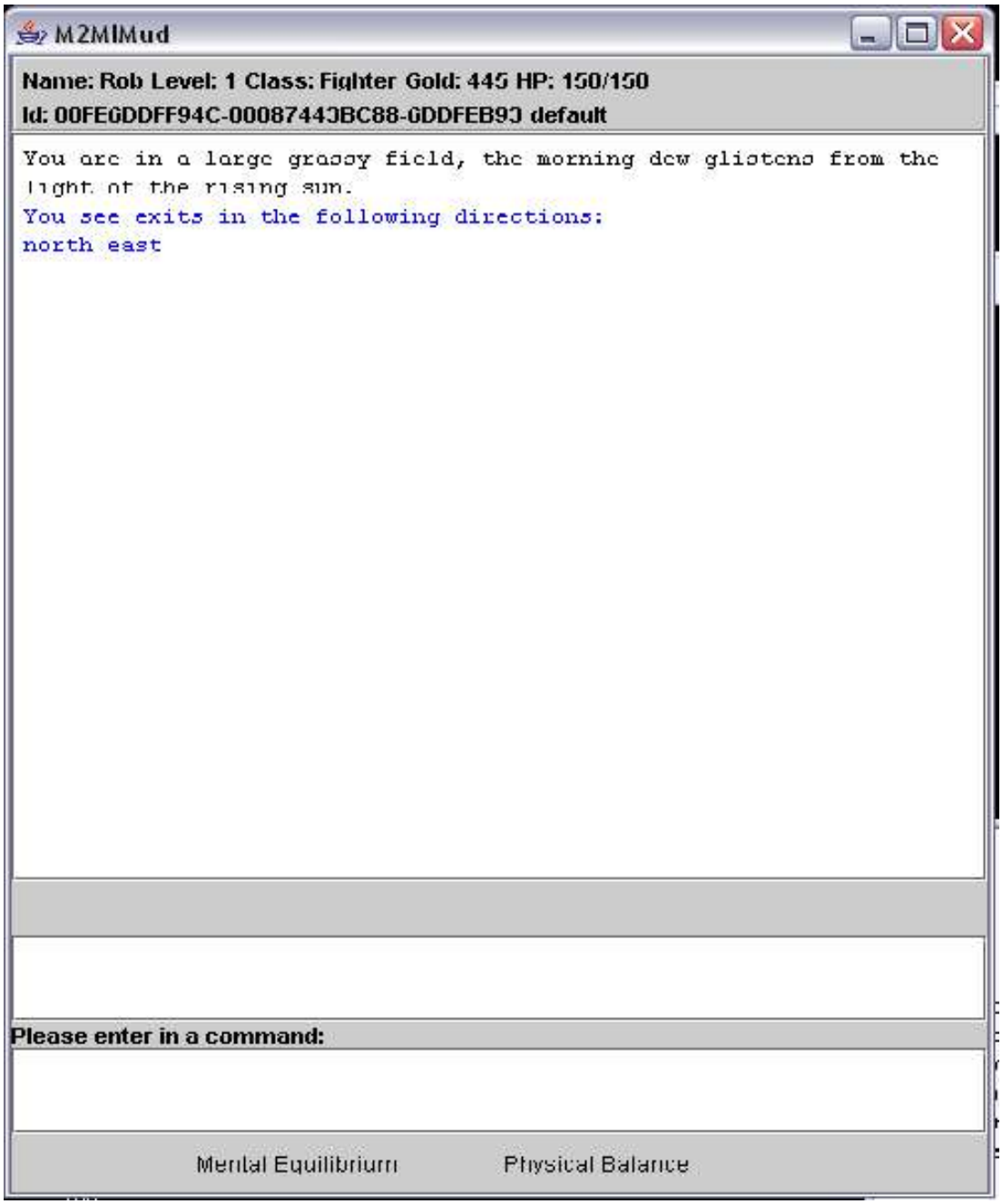
The following is a description of the windows used in M2MIMud

7.1) Find Games Window



This is the find window. Sessions will appear in the text area. The user can then click in the join button to join the session, or cancel to return to single player mode.

7.2) Main Window



This is the main window of M2MIMud. The top of screen contains character information in this order:

- Name: the name of the character the player is using
- Level: the current experience level the character is
- Class: the class the character is
- Gold: the amount of gold the player currently has

- Hit point information: Current HP/Maximum HP

The second line contains the following information:

- Id: The unique id of the player
- State name: The name of the state the player is using. If this is empty, then the user must execute the “save world name” command to given this variable a value. Afterwards, executing a “save” command will save the state information using that name.

The second component is the main feedback window. This text area displays all the information for a user, including a description of the current room, the wares of a merchant, any messages the user receives, and more. The second text field is the combat window, which displays information about the damage the player does to their target and receives. The third text field is the command window, where users enter in commands. The final lines are whether or not the person has physical balance or mental equilibrium, which indicate when a user can attack.

7.3) Map Window

This is the map window. The current time of the game is displayed at the top, and the player’s current coordinates are displayed at the bottom. The “P” in the center is the player’s location, and 8 character values around it refer to the 8 tiles that surround the player’s current location. In this case, 5 of the tiles are tiles that the player cannot move onto, and the remaining 3 are grass tiles.

8.0) Future Work

As with all things, M2MIMud is not perfect, and being that it is a relatively new idea, it would be hard to claim that it is the best way to do things. This section deals with what can be improved and what can be added.

First and foremost, one of the most significant areas for improvement revolves around the fact that there is no central server. In the majority of MUDs and MMORPGs, game data is stored on the central server. This prevents players from cheating and doing things like artificially raising their level or giving themselves equipment they did not obtain through

the game's processes. However, this option is not available in M2MIMud, since there is no central server. All game data is stored on the unit that is running the game. This is a major security issue, since there really is no way to prevent a player from hacking their data. The best one can do is to slow down the process. On way possible way would be to add a level of encryption and decryption to the saving and loading aspect of the program. When the system goes to save data, it could encrypt it. And when it is loaded back up, is could decrypt. Added to the fact that the system does not use any standard file formats, this can help to prevent cheating. Of course, such a method is not fail proof. A user could reverse engineer the game and use that information to decode the files. There really is way to stop a determined cheater from cheating, since the game data is not stored on a central server protected by firewalls, passwords, and other such tools. Even then, however, it's not 100% safe. There are stories in many MMORPGs how users found way to modify their character data or manage to enable the developer commands. Perhaps one of the most famous examples is when a group of hackers broke into the servers for Ubisoft's *Shadowbane*. Upon doing this, they changed all of their character data, and moved several players and player run cities into the bottom of the ocean.

The point is there really is not failsafe method of stopping hackers. In fact in [1], Bartle makes a most interesting point, you can't even be sure that a person is connecting to your server using your client. Most MMORPGs and MUDs rely upon the fact that users do not have direct access to their character information. In M2MIMud, that is not an option.

Another possible improvement would be to store the item and monster data in actual databases as opposed to specialty classes. Of course, this is just a matter of preference, as the actual specialty objects work just fine. However, it is an area for possible improvements.

Another issue is security. Right now, there is no way for M2MIMud to guarantee that a message sent is by a member of a legitimate M2MIMud session. However, this really ad hoc group security, which is far beyond the scope of this project. Plus, there are some other areas of research that are dealing with just that issue. See [14], [15], and [16] for more information.

One of the biggest areas for improvement would be in the combat system. Right now, M2MIMud uses a simple system for calculating damage. In fact, it’s really there to show that combat is capable in such a system. However, a good game will have a more complex system that takes into account things like the weapon type, the player’s class and stats, the armor type of the player, and more. A good combat system is complex and fair, and that is probably the one of the significant improvements that could be made to M2MIMud.

Significant improvements could be made to the command parsing aspect as well. Right now, command parsing is simple: the system merely checks the user’s input against a large set of regular expressions, and the extracts the information as needed. As a result, the parser class has a large set of Pattern type variables to hold. Also, the parse function is rather large. One possible way to fix this would be find a yacc/lex equivalent for Java, or use Java’s Jack package. Although research was done into this it was decided to stick with the pattern matching because it achieved the desired result, which was to transform a string into a command object.

Another area of improvements would be the actual MUD aspect of the game. As discussed, such things like crafting and quests are not currently present in the project. This does not mean they could not exist, but rather there would need to be a way for a user to dynamically setup the quests and crafting situations to match the world they designed. A tool could be created, for example, that have a database of available quests that the user could select from and modify based on their world. For crafting, it would be more of a challenge, but it could be done in that there is also a database of items and the ingredients needed to create them, which can be adjusted based upon the setup of the world.

More than a game application, M2MIMud is also an ad hoc application. As a result, there is an opportunity for more work to be done in that aspect of the system. M2MIMud is a study of state maintenance in an ad hoc environment. Members of a session have a world in which they play, and they need to keep that world as consistent as is possible to play the game correctly and fairly. To accomplish this, each system broadcasts out when it makes a state change, and periodically systems broadcast out their version of the state so that others may examine it. This creates an enormous amount of communication, and really limits scalability. Of course, most ad hoc applications do not have a lot of simultaneous users, but this is still and area of concern. One possible solution would be to remove the state broadcast and focus more on state change message. The system would have to be intelligent enough, and the message descriptive enough, so that it could detect when its state is no longer correct and can use the messages to resynchronize itself with the rest of the world. This would lead to a lot more complexity on the side of the system, but it may help t reduce network traffic.

Another area of research deals with the fact that there has to be some commons basis in M2MIMud, regardless of the session. Player classes, items, merchants, and NPCs all have to be the same to ensure consistent play. There are two ways to do this: one would be to hardcode this information, which increases the size of the program, but it ensures users cannot change it. Another way would be pass this information with the game files, but not provide the users the tools to change it. A skillful user could use something like a hex editor to change this information. There has been one suggestion, that of encrypting the files on the disk. But there is a potential for research to be done into safeguarding this information from users, and detecting when a user has changed the information and how to fix it. This is particularly a challenge because there is no central server, so the systems would need some manner to access another system's file, verify they are correct, and change them. It also relies on the fact that the checker itself hasn't been corrupted by the user, which brings it back to the original question of how to safeguard the data from a user.

9.0) Summary

M2MIMud is a MUD type application that runs on an ad hoc network using an IP-less broadcast form of communication developed at the Rochester Institute of Technology. It is an exercise in state maintenance in ad hoc environments, as well as a study of how to take existing applications and port them over to an ad hoc network.

This project was easily the most challenging project I have ever work on, but it was also

the most fun and the most rewarding. I learned quite a bit on this project. First and foremost, this was my first large-scale project in which I used the Java programming language. Previously, much of my work was done in C/C++. However, since M2MI was written in Java, applications that utilize it must also be written in Java. So, this project really helped to increase my knowledge and skill of Java.

Second, this was my first ad hoc application. Previously, I had written either client-server or non-networked applications. M2MIMud was the first type of applications of its kind that I have written. As a result, I gained much insight into how to write such an application. I had never dealt with the issue of partitioning before, and while such a thing could occur in a normal network, I have never had to deal with it. So, one of the challenges of this project was designing an algorithm that would help to reduce the effects of a network partition once it healed, and yet would be fair to the players. Also, the lack of a central server presented a unique challenge because there was no single source to coordinate the actions of the players, and there was no way to perform name checking so that users have a unique identifier. Of course, I found solutions to these, as the M2MIMud library has an Eoid class, which, for my purposes, was capable of generating the unique number I needed. For coordinating the actives of the players, I borrowed the concepts I had learned by playing real time strategy games, and that was to simply have all members run the game at the same time. Of course, this causes problems, but in the end, I felt this was the best solution.

In addition to being my first ad hoc application, M2MIMud was something else: my first game application. I have a love for computer games of all kinds, and I have always been interested in trying to develop one. M2MIMud was my first chance to put what I had learned from playing various console RPGs, MUDS, and MMORPGs to good use, and to develop my own game. As anyone who has written one can attest, developing a game is no small task. In addition to the game system material, there is also a need to be able to create an interesting world that players will want to play in. While M2MIMud focuses mainly on the game system aspects of the MUD (i.e. command interpretation, state maintenance, etc), I did look into the possibilities of adding some more story related idea. However, in the end, the question was simply: how much? How much content should I add? There was a lot, but in the end, I added the content that I thought was important to the overall project, things like monsters, room descriptions, player classes, and items. There is a wealth of more to add, like permanent villages, more NPCs, quests, more player classes, crafting skills, etc. Of course, of these, only a limited number could actually be added effectively. Regardless of the session, the items would have to remain the same. A player would not like to go to two separate worlds and then suddenly discover that his or her spells do no work in one world because they are not defined there. Same with the player classes, these have to remain consistent. They should be flexible enough to be dynamically added (i.e. not hard coded) but protected so that players cannot change them. In my project using specialty objects to store the information and simply include with the game files does this.

The intent is to make it easy for me; the developer, to add new thins while developing, but prevents users from changing or adding new material. So, in the end, more items and player classes would not add new things. Adding towns and more NPC would be similar: it would just simple be adding more on to the system, but nothing really new. So in the

end, I focused more on the game mechanics. I did some research into existing code bases and projects, and found nothing that I thought was useful, so I wrote my own. The most relevant one I found was [4]. So I was writing an MUD from scratch in an environment that is still fairly new, and I got it to work. It may not best the absolute best implementation, but it works, and I feel that is a huge accomplishment.

I also feel that M2MIMud is a testament to the usability of M2MI. For the most part, I did not have to worry about the underlying communications. I could focus my work more on the actual game aspect of the project. By abstracting away the communications and changing it into a method calls; these significantly reduced the complexity of the project. I was able to write a more natural program, one that made uses of method calls to do work, not packaging and sending messages. In the end, I felt that M2MI allowed me to write a better program.

I would like to call M2MIMud a success, because, on a personal level, it was great introduction into the world of game development, as well ad hoc applications. It took a lot of work, but in the end, I think it was worth, as I had a lot of fun, and I felt that I really accomplished something by developing this program.

Glossary of Terms

RPG: Stands for “role playing game”. This refers to a game type that is popular among many computer users. Players assume the role of one or more characters and will usually play in some world.

MUD: Stands for multi-user dungeon. An early, text-based, computer RPG. There are still many run today. Some of the more prominent ones are Aetolia and Threshold.

RTS: Stands for real-time strategy. A type of game where a user chooses a team and then builds up an army (in real time) to defeat their opponent.

MMORPG: Stands for massively multi player online role-playing game. These can be seen as an evolution from a MUD. Both a role-playing games where the user assumes the role of a character and plays in some world. Unlike a MUD, however, MMORPGs usually have several servers each with thousands of players on them. They are also graphical games, and rely on text messages primary for communication.

M2MI: Stands for Many-to-Many Invocation. A distributed object system API for ad hoc networks developed at the Rochester Institute of Technology. It is an ip-less, broadcast API that uses interfaces and handles to send messages.

NPC: Stands for non-playable character. This refers to any character in the game, other than monsters, that are not controlled by a player. An example of this would be the merchants.

Session: A group of M2MIMud players who are all playing in the same world.

State: The current setup of a world. The state contains the map of the world, as well as the location and activities of the players, monsters, ponds, house, and merchants.

Layout: Also known as the map. The layout specifies the dimensions of a world as well as the

World: The area in which a session takes place. In M2MIMud, a world is an n x m grid of rooms that have a certain type. Users may move throughout these tiles, but they may not go off the edge tiles.

Room: A single tile in the world, a room represents the immediate area surrounding the user.

Player Class: A class is a player's designation of what type of character they are. In M2MIMud, there are 2 player classes, the melee combat oriented fighter and the spell casting oriented mage.

Player Character: A character in the game that is controlled by a user.

Monster: A computer controlled mobile entity that users can fight to earn experience points, gold, and items. These entities are non-aggressive and will only fight a user if the user attacks them first.

Merchant: A non-mobile entity that sells and buys item to/from users.

Experience Level: A mark of growth. At each experience level a player has more hit points and access to more spells if they are a mage.

Experience Points: The amount of growth a player has into their current level. Experience points are gained by fighting monsters, and when the user has enough, they will advance to the next level of growth.

References and Resources

1. Richard Bartle. Designing Virtual Worlds New Riders Publishing Indianapolis, Indiana 2004

2. Alan Kaminsky and Hans-Peter Bischof. *"Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems*." 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002), Onward! track, Seattle, Washington, USA, November 2002.

3. Daniel Bauer, Sean Rooney, and Paolo Scotton. *"Network Infrastructure for Massively Distributed Games."* Proceedings of the 1st workshop on Network and system support for game. Bruanschweig, Germany 2002

4. M2MI *http://www.cs.rit.edu/~anhinga*

5. P2PMud*http://www.p2pmud.com*

6. Aetolia *http://www.aetolia.com*

7. Threshold *http://www.thresholdrpg.com*

8. RetroMUD *http://www.retromud.org*

9. Final Fantasy Online *http://www.playonline.com/ff11us/index.shtml*

10. Dark Age of Camelot *http://www.darkageofcamelot.com*

11. Everquest *http://eqlive.station.sony.com/*

12. Shadowbane *http://www.shadowbane.com/us/index.php*

13. Ultima Online *http://www.uo.com*

14. Yair Amir, Giuseppe Ateniese, Damian Hasse, Yongdae Kim, Cristina Nita-Rotaru, Theo Schlossnagle, John Schultz, Jonathan Stanton, and Gene Tsudik. "*Secure group communication in asynchronous networks with failures: Integration and experiments*." The 20th International Conference on Distributed Computing Systems, pages 330-, April 2000

15. Yongdae Kim, Adrian Perrig, and Gene Tsudik. "*Simple and fault-tolerant key agreement for dynamic collaborative groups."* Proceedings of the 7th ACM conference on Computer and communications security, pages 235-244, 2000.

16. Michael Hurwicz. *Emerging technology: Peer-to-peer networking securit*y. Network Magazine, 434:155-, 2001.

17. Sergio Caltagrione, Matthew Keys, Bryan Sclief, and Mary Jane Willshire. *Architecture for a Massively Multiplayer Online Roleplaying Game Engine.* Journal of Computing Sciences in College. pages 105-116 December 2002.

18. Nayeem Islam and Murthy Devarakonda. *An Essential Design Pattern for Fault-Tolerant Distributed State Sharing.* Communications of the ACM. Vol 39, Issue 10. pages 65-74. October, 1996.