Introduction

The proliferation of small, wireless devices such as personal digital assistants and cellular phones has given rise to a new type of computer network, which is referred to an ad hoc network This type of network is peer to peer network, in which members of the network talk to each other directly, rather than going through some kind of server. These types of networks are more dynamic then standard networks because members may join and leave at will, but there is no server that takes care of this aspect. So a resource that was available may suddenly become unavailable, but there is no direct to notify members of this happening. As a result, these networks are very unreliable, since resources may become available and disappear with no notice. Such a network, however, is very flexible, since there is no server, and is very portable, since the systems that use these networks are usually located in small, portable, wireless devices such as personal digital assistants (PDA's) and cellular phones.

Project Purpose

The purpose of this project is to design and implement a small scale MUD type game that runs on the M2MI system developed by Professor Alan Kaminsky and Professor Hans-Peter Bischof at the Rochester Institute of Technology. This system is provides a mechanism for IP-less communications by utilizing broadcasting. This makes it ideal for an ad hoc network environment, where an IP based communication scheme is not optimal due to the dynamic nature of members involved in the network.
On a much larger sense, the goal behind this project is see how well a persistent state program works on an ad hoc network. What makes a MUD type program unique is that, unlike word processors that store there data once the program is shut down, a MUD continues even though the user has left. What this means is that if a user leaves, and then comes back, they must be updated with any changes that occur in the world, and if possible, their character should return to the world in the same place they left (is possible). Thus, a MUD must remain persistent over many clients, not just one as is the case with a word processing system. This, of course presents many interesting issues that this project seeks to handle.

Application Domain

This project involves two application domains, ad hoc networking and persistent state programs. A persistent state program is one that maintains its state upon program termination, and allows users to restore this state once the program is reloaded into memory. An example of this could be a word processing system that, upon loading, allows user the option of loading the last opened document with the cursor at the last known position before the document was closed. Another example of this type of program would be a video game of sorts, one where users play a character through a story line, and there are various points at which the program stores data about the character and where they are. When the user starts the game up, they are given the chance to load up this data and continue playing from where they left off. Many role-playing games (rpg's) use this system, because to complete the game requires many days of play.

One of the earliest examples of a persistent state program is a multi user dungeon role-playing game, or MUD for short. A MUD is game in which a user assumes the identity of some mythical character, and goes out into a virtual world where they fight computer controlled opponents (referred to as PvE – player versus environment) for money to purchase items and equipment with and experience which is used to advance them in levels. Each new level increases the character's statistics and gives them access to more powerful weapon styles and spells. Some MUDS have a setup where players can attack and fight other players (referred to as PvP – player versus player) combat. They are referred to as role playing because more often then not, users are asked to role play their character, which means they don't refer to events outside the scope of the game, and they way they act and communicate with other players is affected by their role.

Functional Specification

The game to be designed will be as simple as is necessary to illustrate the methods used by the underlying system to handle the issues that arise as a result of being on an ad hoc network. The development of large scale MUD is project in and of itself. However, users will have option in how what they do.

Design Goals

User friendly – While a text based system, the game should still be user friendly so that if the user makes mistakes, the game informs them of this and what steps need to be taken to correct the mistake or perform the action correctly.

Platform Independent – So long as the user have the Java Virtual Machine installed on their system, they may play the game

Object oriented – The game will be written in Java, and follow the principles of good object oriented design.

Character Creation

Users will be allowed to create on character with which to play. While large scale MUD's usually allow users to create more than one, this game is intended to run on a small computing device where space is limited. Thus, to conserve space, users will be allowed to create one character. Two characters with the same names will be allowed, as each user will have a unique ID to differentiate them from other players with the same name. Each character starts with 100 gold pieces to use as money. Each character will have 3 slots on them for wearing equipment. These slots are right hand, left hand and body. Additionally, each character will have a 10 slot inventory which with which they may store items. Gold does not take up space in the character's inventory, but will remain a separate amount. To simulate a true MUD, each character may raise experience levels, up to level 20. Doing will increase the players "stats", which are the values for a character's strength, constitution, agility, and intelligence.

The Worlds

Upon creation, each user will be placed into a 10 by 10 grid that is their world. The center of this world will have a bind spot, a place a user returns to if they die or the area in which they were disappears, and a merchant that sells items. This merchant will sell armor, weapons, a healing herb, which heals a character's hit points, a magic restoration potion used to restore a character's magic points, and a scroll which returns the user to their bind point. A user may designate a tile as their house, and provide a short description of it, and indicate whether or no other users may enter into their house. Additionally, a user may indicate if a tile is full of water, a grassy field, or full or trees. In this fashion, users may create fields, bodies of water, and forests. They may also indicate what directions a user may move from this tile. The only exceptions are the border tiles, in which the direction leading out cannot be flagged as not reachable. When joining a set of larger users, an individual world is added to the other worlds to form a much larger world. As a result, users may travel between the worlds seamlessly. If the world in which a user disappears, then they are returned to their bind spot in their own world. Upon reentering the world, the game check to see if the world the user left is still available. If it is, the user is returned to that world, if not, they enter the game at their bind point. Users may also place up to 10 computer controlled mobile enemies, or mobs, so that other users can attack them.

Movement

Users may move in one of the four following directions: north, east, south, and west, depending on how the tile they are laid out.

Interactions

User may communicate with each other via two methods. One is a basic "say" system in which any user on the same grid will see the message. The other is via a send system, in which only the sender and the receiver will see the message. Users will be able to trade items they carry with other users. They will also be able to purchase items from the world's merchant. Each merchant will carry 10 of the items in their inventory, and upon depletion, the user will be asked if they wish to restock their merchant.

Combat

Developing a full-scale combat model for any MUD is a complex process that usually takes into account a variety of factors. Thus, the combat system in this game will be very simple. User will be able to attack the computer-controlled enemies and hit for the same damage. To attack other players, users will have to issue a command, which the other user must accept. To simulate a experience level system, a user must defeat as many mobs as the level to which they are aiming for. To acquire level 10, a user must be at level 9 and defeat 10 monsters. Defeating another player will result in the victor being able to acquire the loser's gold. Damage done is determined by the opposing player's character type and the user's stat values.

Commands

All commands will be entered into a command prompt. Users must enter in each command exactly or enter in a shortcut command. There are no graphical commands of any kind.

Security

The environment in which the game runs is assumed to be secure.

Design

As stated, the game will be designed according to object-oriented principles. The system will written using the Java programming language, and will therefore run on any system that has the Java Virtual Machine installed. Additionally, the target system is anything that can be classified as a small device, but more specifically, a personal digital assistant.

Because this system is trying to model a client server program on an ad hoc network, there are a number of issues that must be addressed.

Naming

On the surface, the problem of names seems to be a trivial matter. Each character will have a unique id differentiating it from other characters with the same name. However, where this causes a problem is in the personal communication system. The way this is usually done is that the user would type in something like /send <user> <message>. If there is more then one user with the same time, then all with this name would potentially receive this message. One way to solve this problem would be to have a friends list of sort, in which a user stores the names and ID's of other players, and can only send message to people on this list. Of course, this is restrictive, since the user must first add this person to his or her list, but it does solve the problem when the user wants to send a message to an individual person. Another problem, of course, is the case in which the user is adding a new friend that has the same character name as another one. In this case, the user will be asked to modify the name so that it is unique. It should be pointed out that a player must get the permission of the player they wish to add to their list. This has nothing to do with naming, but it a way to inform user's of who has them on their friend's list. Upon adding a player to a person's friend list, the information about the person who started the transaction will be added to the other person's list. When a user sends a message, both their name and ID is sent to the person, so that the system will look up the ID of the person and display that name, not the actual name. For space considerations, users will be able to have at most 20 friends in their list.

Command Processing

To reduce the size of data stored by an individual game, all commands are processed by the game that has the player in it. The world the user is in is not downloaded into the process they have started, but rather the two systems will work together to obtain and process commands. The user's process will obtain the commands and send them to the host process. The host processes in turn processes the command and sends out the result all other process that have players in it. This causes network traffic, but it is the most efficient way to ensure that all players are playing in the same world, and is reduces the amount of data a game unit most store. This means that games unit will need to maintain a list, called the active player list, of who is in them. Since this number could potentially be infinite, the maximum number of players allowed into a world is 20.

Data

All character related data will be stored locally on a single device, and will not be transferred over if the unit disappears. However, all data related to a world will be shared and updated on all systems, so that if a unit leaves, another world can take over that world. However, the character associated with that world will disappear.

Joining a Group

This is a group-based game, so a user will have to specifically join a group. To allow for this, each device will periodically send out a heartbeat which informs all other devices that it exists and what group it is a member of, if any. A device that is already in a group will ignore this heartbeat; unless it is a member of the group the heartbeat came from, in which case this heartbeat also has some other data in it.

If a device is not a part of any group, and it receives such a beat, the user will be given the option of joining the group. Once the user joins, every device receives a copy of this world, as well as how to get to it through the existing current world.

Leaving the Group

When a user decides to leave the world, they just sign out of the group. No other action is taken by the system. Their world remains in the group.

However, upon leaving a group, a user may flag his or her world as inactive, which means they have no plans to return in the near future. If space ever becomes an issue, the system as a whole will decide on which inactive worlds it will purge to free up space.

Control

A device owns one world, but it may control multiple worlds. What this means is that every world has a source device from which it comes. Additionally, if this device is present in the game, it must handle this world. However, since a world never leaves the game unless it is actively purged, if the owner of the world leaves or disappears, then another device must take control. This is achieved by using a timeout. When a device sends out a request for a change within the world, it begins a timeout. What happens is the controlling device for the world will receive the change request, make the change, and then broadcast out the modified world. This cancels the timeout. However, if the timeout succeeds, this device will take control of the world, make the change and then broadcast the changed world along with the fact that is has taken control of the world.
There is, of course, a potential problem with multiple timeouts. To handle this, once, if a device is notified of another change request for the world, it will reset its timer at with a larger value. This is an attempt to ensure that the last known device to request a change will take over the world if the owner of the world is no longer present.
If a device leaves a group and then comes back, it must use the world it left in that group to play the game.

Group Partitioning

Group partitioning is quite a interesting situation to deal with. There are two issues to handle once a partition heals: control and conflicting worlds. Control isn't very hard. If the owner of the world is present once the partition heals, it will assume control of its world. If it is not present, then each participant which controls a world will communicate with the other participants to determine who has held the world the longest, and then whomever that is becomes the controller for that world. This is accomplished by nature that every device keeps a record of the time and date at which is took control of a specific world. If control of the world is taken from a device, then it erases this timestamp.

To handle different worlds, a hybridization technique will be used. The idea is based on the fact that changes within a world will be isolated to a few things. What each system will do is to compare the various versions of its world to determine what changes have to be made. Any non-conflicting change will remain intact, while any conflicting change will be resolved to the best of the ability of the process. If possible, user whom these changes affect will be notified of this occurrence. Since the hybridization process takes time, the world will be effectively frozen while it is happening. When this occurs, and changes happening in the world are immediately ended, and no changes will be allowed while the world is frozen, with the exception of movement. Any tile that has been changed will display the appropriate text, while any tile that needs to be worked on will display text notifying that tile is in the process of being changed. Once the hybridization technique has been accomplished, the world is sent out and updated. During this time, a special signal is sent out to the processes so that they are aware that the world is still in existence. While requested changes are ignored, the timeout process is still in effect, so this signal serves to cancel the timers to ensure that no other world gains control of the world while it is being hybridized.

Deliverables

For this project, I will deliver the following:

- Source code for the project.
- A binary executable image that can be run on any system with the Java Virtual Machine.
- Full design documentation, using Rational Rose.
- A user manual that describes how to play the game.
- A final report detailing how the project went, problems that arose, and how I overcame them.

Tentative Schedule

- September 15, 2003 Research and Design Phase Beginscd pu
- October 20, 2003 Implementation Phase Begins
- February 16, 2004 Project Completed
- February 27, 2004 Project Defense

Reference List

Bischof Hans-Peter and Kaminsky, Alan. “ 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.

Intention of Paper:

This paper describes M2MI, the communications mechanism the system will be using to transfer data to and from all participants in the game. It describes, in detail, how M2MI works, and explains many of the tools that will be utilized in the project, such as omnihandles and multihandles. It is the basis on which many of the techniques I intend to use will be based.

Caltagirone, Sergio, Keys, Matthew, Schlief Bryan, and Willshire Mary Jane. "Architecture for a Massively Multiplayer Online Role Playing Game Engine." *The Journal of Computing in Small Colleges:* Volume 18, Issue 2 (December 2002)

Intention of Paper:

The main point of this paper is to prevent what the authors feel is a very good software architecture for a MMORPG. They set for themselves 6 goals, security, scalability, maintainability, load balancing, minimal network traffic, and the good performance of the client application. Essentially the architecture is a very modularized setup with the modules performing different tasks. One important concept here is that of the publisher subscriber model, in which only clients that need to be updated are. This has some interest to the proposed game, since this might help to reduce network traffic. However the main purpose of this paper is a comparison and contrast piece, which provides information that can be used to show the unique issues with an ad hoc game as compared to the normal model, which this paper presents.

Chiandy, K. Mani and Lamport Leslie. "Distributed Snapshots: Determining Global States of Shared Systems". *ACM Transactions on Computer Systems (TOCS)***:** Volume 3, Issue 1 (February 1985

Intention of Paper:

This paper presents an algorithm that allows the distributed components to determine the overall global state in a distributed system. Basically, each processor has a channel to the other processors within the system, and between them there a re variety of states depending on where messages are between the 2 processors. How the algorithm works is to record the states of the 2 processors and then the stats of the channel itself (message in transit or not for example). This information is then passed among all other so that they may determine the global state if they need to. Here, the issue is that there is no direct connection between the processors since my project uses M2MI, however it may be possible to alter this algorithm to help when dealing with network partitioning.

Cronin, Eric, Fildtrup, Burton, Kurn, Anthony, and Jamin, Sugih. "An Efficient Synchronization Mechanism for Mirrored Game Architectures" *Proceedings of the first workshop on Network and system support for games* Bruanschweig, Germany, April 16-17 2002

Intention of Paper:

The main of this paper was to present a technique for state synchronization called Trailing State Synchronization. How this works is that each game process runs several instances of the game itself, which update at different intervals. So the primary state would update immediately, while a second state would delay 5 seconds before updating, and another would wait 10 seconds. If there are any inconsistencies, then the process can use these different states to resynchronize itself. This sounds like a good idea to look into, as it could make things like network partitioning much easier to handle. However, one problem is that the paper assumes a sort of server architecture where there is no central server, rather the game is distributed among a variety of other servers. This is kind of a problem since the game I am designing runs on an ad hoc network. However, it is still something to look at and think about a similar approach.

Devarakonda, Murthy and Islam, Nayeem. "An Essential Design Pattern for Fault-Tolerant Distributed State Sharing" *Communications of the ACM:* Volume 39, Issue 10 (October 1996)

Intention of Paper:

The paper presents a design pattern for a distributed state sharing application, and has been used in IBM for several applications. The name of the pattern is the Recoverable Distributor, and it is consists of global and local state managers as well as global and local fault handlers. As expected these pieces work together to handle failures and try to ensure the global and local states are consistent. However, there are two issues, one is that network partitioning is beyond the scope, and the other is that this pattern uses an election process to determine who runs the global failure handler. However, this pattern is a good one, which could be modified to handle my program. If nothing else, however, it will give insight into a possible way to design my system.

Griwodz, Carsten. "State Replication for Multiplayer Games" *Proceedings of the first workshop on Network and system support for games* Bruanschweig, Germany, April 16-17 2002

Intention of Paper:

The point of this paper is another hybrid server approach to the idea of a distributed game. Clients connect to proxies that run an overall game state, but rather than sending out all messages these proxies work to figure out which messages are local, and which need to be broadcast to the rest of the players. Obviously, since there are no servers in my game, this technique won't necessarily work, however, this paper does present some other idea. Chief of these are an precedence ordering of message so the most important ones get sent out and processed first, and the other idea is of message packing, where several message are packed into one transmission. These are potential idea to take into consideration while designing the system.