MIRE: A Multi-user Information

Retrieval Environment

by

Erik A. Kay

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the degree of

Bachelor of Science in Computer Science and Engineering

at the Massachusetts Institute of Technology

May 1992

Copyright Erik A. Kay 1992

The author hereby grants M.I.T. permission to reproduce
and to distribute copies of this thesis document in whole or in part.

Author _____

Department of Electrical Engineering and Computer Science
May 15, 1992

Certified by _____

Walter Bender
Thesis Supervisor

Accepted by _____

Leonard A. Gould
Chairman, Department Committee on Undergraduate Theses

Abstract 7
9
9
9
9
9
Dedication 9

CHAPTER 1 *Introduction 11*

Subject 11
Scope 12
Communities 12
Interface 13
Purpose 15
Previous Work 15
Development 17
Overview 17

CHAPTER 2 *MUDs as serious media 19*

CHAPTER 3 *News 23*

CHAPTER 4 *User Modeling 27*

CHAPTER 5 *What are MUDs? 31*

CHAPTER 6 *A Sample run through MIRE 37*

CHAPTER 7 *The MIRE driver / parser 47*

Selection of driver 47
Cleaning up the driver 49
Planning 49
Doing it 50
MIRE additions 52
Design and implementation 52
Future work 59

CHAPTER 8 *The MIRE world (mudlib) 63*

Influence of the driver 63
Starting the project 64
Design goals and their implementation 65
MIRE specific additions 69

CHAPTER 9 *Future Work 73*

Powerful clients 73
Multiple MIREs 74

Public discussion forums 75
Volume and noise level reduction 76
Community Tailored News 78
More powerful user monitoring interactions for feedback 78
New drivers 79

CHAPTER 10 *Conclusion 81*

Acknowledgments 85
Bibliography 87

APPENDIX A *Abbreviations and other information 89*

Acronyms 89
MUD names 90
Character Names 90

CHAPTER 2 *An LPC tutorial for C programmers 93*

What is LPC? 93
What makes it object-oriented? 94
What does C have that LPC doesn't? 96
Limitations of LPC as a language 96
What does LPC have that C doesn't? 97
Runtime Features 97
Object Oriented Features 98
Strings and Arrays (Mappings) 101
Miscellaneous Additions 103
Scope of message efuns and `catch_tell` *109*

CHAPTER 3 *MIRE Commands 111*

User Commands 111
Developer Commands 112
Developer File Commands 112
Soul Commands 113

MIRE: A Multi-user Information Retrieval Environment

by

Erik A. Kay

Submitted to the

Department of Electrical Engineering and Computer Science

May 15, 1992

In Partial Fulfillment of the Requirements for the Degree of

Bachelor of Science in Computer Science and Engineering

Thesis Advisor: W alter Bender

Abstract

MIRE is a system for multi-user news and information retrieval. It addresses two primary objectives: user interface and the world community's access to and exchange of information in a group environment. The goal is to produce an environment in which people can quickly and easily obtain information which is particularly relevant to their own interests, and to allow a coherent exchange of ideas between people throughout the world. It consists of two separate parts: the driver and the mudlib. The mudlib is the code which describes the world in which users interact and

the driver is the program which accepts connections and parses and executes the mudlib code.

Dedication

This thesis is dedicated to the residents of ThiRD EAst and all of my other friends at MIT. Without you, I would have surely gone insane.

CHAPTER 1

Introduction

Subject

Today we find ourselves well into the information age. Now past the initial start-up phase, we are faced with a whole new set of problems. Rather than trying to get more information from around the world into our computers, we find ourselves trying to find ways to manage that information. This problem is far from one dimensional. There are many issues being looked at. Primarily , I will address two of these issues: the problem of user interface as it relates to productivity and information management, and the problem of news and the world community . Specifically , I've designed an environment called MIRE which allows multiple users log in and communicating with each other , read their personalized newspapers who are

interested in reading their own personalized newspapers, as well as getting news which is customized to communities of people. MIRE tackles the idea of user interface from a slightly new angle and introduces the idea of being able to access and share community information in a simple way .

Scope

Communities

As things currently stand, most news / information retrieval systems are geared towards a single user . An example of this is a newspaper or an evening television newscast. These are both single user interfaces. No interaction, discussion or sharing of these ideas occurs between people directly through either of them. Also, all of the content within it is tailored to be appealing and interesting to the largest variety of users of the systems. There is no way to customize what you see to a particular set of interests. There are a number of environments which have been dealing with some of the multi-user interaction. One of these is USENET , an international network of discussion groups. Other examples include various electronic bulletin board systems such as Prodigy , CompuServe and the WELL. One problem that many of these systems have is that they are primarily store and forward interfaces. There is very little, if any , direct interface or discussion between users, making debates and discussions with

immediate feedback and interaction more difficult. The system I implemented allows for more direct interaction between users, and more flexibility in the kind and quantity of information received. I will be using user and group modeling techniques through a system known as Doppelgänger[1]. This will allow users to get information which is tailored directly to their own personal interests rather than being in a canned format intended to be general enough to be pleasing to a large number of people. It is also capable of providing more flexible types of discussion groups such as groups which are based upon a common set of interests rather than based around a particular subject.

Interface

A great deal of research is being done on graphical user interfaces (GUI). GUIs have become popular because they are easy to use, allowing novice computer users to learn how to use them more quickly . Examples, of GUIs can be seen on many popular computer systems today , most notably on Macinotsh and NeXT computers. GUIs achieve this by applying a physical metaphor to operating system features. For example, copying a file into a directory is represented by dragging a sheet of paper into a file folder , and destroying or removing a file is represented by dragging it into a trash can. Natural metaphors such as these make operations more intuitive to novice users. However , not everyone is jumping onto the GUI bandwagon; some are fans of command line interfaces (CLI) such as those of MS-DOS or

UNIX. These interfaces are popular primarily because they can be fast and powerful once mastered. They also have a much richer vocabulary than GUIs and are therefore more expressive. For example it is very difficult to say “erase every file in this directory which begins with the letter c” in a GUI, while it is very easy to say the same thing in a CLI. The major problem that people have with CLIs is that they are complicated and hard to learn. The reason that CLIs are more difficult to learn is that there is no direct metaphor between commands and their results. Also, the sentence structure of commands in CLIs is not closely related to English in most cases, so it may be confusing to the novice. For example, it might take a novice some time to learn that in UNIX “rm c*” means “remove all files beginning with the letter c”. An alternative approach could be taken to CLIs which would make a sort of compromise between GUIs and CLIs. This system would provide a physical metaphor which was accessible via the command line, through commands which are very similar to spoken English. For example, a user could say “put the document in the folder” to copy a file into a directory . Such an interface already exists in modern text adventure games which were among the first computer games written and were later popularized by Infocom and in MUDs (Multi User Dungeons or Multi User Dimension). I took an existing MUD system, and modified it, removing the game elements, and keeping its rich CLI, and virtual world,

multi-user capabilities. Finally, I built a system for information retrieval and exchange.

Purpose

There are many real world applications for these ideas. Some that I envision are an environment for group software development, a networked conferencing system, or a world in which users could share news and information and have public forums and discussions based on that information across national and international boundaries. I focused on the latter of these ideas, developing a news and information retrieval system which can be accessed, shared and discussed by the many users in a completely physical metaphor of that interaction.

Previous Work

Unfortunately, almost no previous research has gone into MUDs as. The majority of research on MUDs has been on their social aspects. This is because MUDs as they stand now are primarily games and places for social interaction; very few offer other services. MUD research so far has focused on two topics: anonymity, and electronic relationships.

MUD connections are for the most part anonymous, and thus a lot of possibilities emerge. For example, Pavel Curtis observes about his MUD experiences, “it appears that the vast majority of players are male, and the vast majority of them choose to present themselves as such. Some males, however , taking advantage of the relative rarity of females in MUDs, present themselves as female” and goes on to say that some of these people go “so far as to entice male-presenting players into sexually explicit discussions and interactions.”[4] Amy Bruckman goes into further detail, finding that this is not an uncommon thing to do, and that in fact, many of the people who gender switch are actually female, and do so to avoid unwanted sexual attention.[5]

The second common area of focus in MUD related research is on relationships formed through a MUD. Through anonymity , people seem less inhibited on MUDs and normally shy people find that they can communicate with others more easily . In a related field, the Internet Relay Chat, similar phenomenon to MUDs are noted as well. Elizabeth Reid notes in a paper on the IRC one user's feelings on this: “by nature I'm shy ... wouldn't talk about such things if you met me face to face, thus the network is good” [6] As a result, it is common for friendships and relationships to develop between people who never meet in real life. Some people who have met

through MUDs have even wound up getting married. One player even commented that he believes that he MUD friendships “are deeper and have better quality” than real life friendships.[5]

However , such psychological and social phenomenon are beyond the scope of this work. T o the best of my knowledge, study of MUDs in contexts other than as games or social environments is almost non-existent at this point.

Development

I have taken an existing LPMUD parser and modified it to provide an interface into the doppelgänger user modeling system and newsdaemon, a local news retrieval service. Both systems are programs developed at the MIT Media Lab to aid in research of electronic transfer and display of news. In addition, I have built a highly structured base world for users to read their news and to interact with Doppelgänger as well as other users. The implementation section will cover these two stages in more detail.

Overview

Chapter two discusses why MUDs are currently looked at as simply games and social forums, and why they have a bad reputation among system administrators world-wide. Chapter three defines news, and the context in

which I am using it. The next chapter explores user modeling and how it can interact with news and MIRE. Following that chapter is a more formal definition and example of what a MUD really is, and how it can be and is used. In chapter six, an example run through MIRE is given. The following two chapters describe the implementation of the driver and mudlib components of MIRE. The paper ends with a look at future possible applications of MIRE, and the conclusion.

CHAPTER 2

MUDs as serious media

One problem with the ideas presented here is that MUDs are not taken seriously as a medium. They are at best considered a harmless game, and at worst considered a destructive force on the Internet.[1]

As stated previously , up to this point, MUDs really have not been used as anything more than as a game or a social forum. However , that is changing. I know of two groups which are working on making a conferencing system out of a MUD. [2] This is something that MUDs naturally excel at because of the ease at which one can express feelings and emotions as well as plain text. In addition, there already exists a MUD which is a kind of

1. The Internet is an international wide area network, connecting universities and businesses alike.
2. One is being developed as a project at TMI, and another is being done at Xerox PARC[4]

MUD university . The purpose of this MUD, The MUD Institute (TMI), is to teach people how to program MUDs. Rather than fight goblins and rescue princesses, or merely socializing, users program on the MUD, attend MUD classes and work on developing the next generation mudlib or MUD driver .

Some consider MUDs to be destructive. This is primarily a problem with the resources consumed by MUDs. Almost all MUDs take up a fairly substantial amount of disk space (10-20 megabytes is usually a minimum) and a large amount of system memory when running (usually 5-10 megabytes). In addition, many use a large percentage of CPU resources. For these reasons, many system administrators will not allow MUDs to be run on their systems. Since the primary audience of MUDs at this point is college undergraduates and high school students, not many of them have access to their own machines on which they can run a MUD. As a result, many of these people run MUDs covertly on unused or rarely used machines at their school. These unauthorized MUDs are often the target of system administrator anger . This is understandable due to the large number of underground MUDs in existence.

In addition to system resources, many network administrators believe that MUDs consume far too much network bandwidth, since the purpose of the networks on which the MUDs run is research and academics. Many network administrators have even banned MUDs on their local network

because of this. Realistically , at sites which have high bandwidth connections to the Internet and within their own network, bandwidth is not much of a problem. It becomes a problem at smaller sites which have slow network connections, where their already slow network connections can be slowed down significantly by an active MUD.

Many system and network administrators have tried to convince their counterparts nationwide to have a global ban of MUDs across the network. Recently in fact, there was a discussion on the USENET net news group rec.games.mud where a letter from an Australian system administrator to other system administrators across Australia. The letter contained a list of statistics which showed that MUDs are the second highest source of packet flow across the Australian / United States internet link, in an attempt to convince them to ban MUDs at their school. He tried to convince others that if this situation persisted, NASA (who heavily funds the link) might decide that the network was not being used for research, and would withdraw funding as a result [1].

In response to these kinds of feelings, many MUD users become hostile, and send originators of these messages a deluge of "flame mail" [2]. Of

1. Note: after the initial writing of this section, the AARNET (Australia's network administration group), officially recommended to users of the network that they ban MUDs at their sites.

2. "flaming" is a phenomenon common on USENET "in which users attack other persons' views far more quickly and violently than would occur with any other medium. Because users are safely hidden behind their terminal, and cannot see who they are talking to, standard social customs concerning conversation do not seem to apply" [10]

course, this only increases the tension between the two groups, and the mood is increasingly tense.

CHAPTER 3

News

The flow of information produced today has far exceeded anyone's capacity to read and understand it all. "More than 100 trillion letterforms[1] are printed on paper every day."[8] A relatively small percentage of this originates on electronic news feeds which are flowing with information at an almost constant pace. The Media Lab alone receives Associated Press, New York Times, United Press International, Wall Street Journal, USA Today, and USENET news feeds, all electronically. These feeds give us literally thousands of individual articles each day.

Since we use all of these feeds simultaneously to build electronic newspapers, we have to deal with a large scale information filtering problem.

1. "Letterforms"..."are letters, numbers, and other symbols that people can read"[8]

Which stories are important? Certainly , my newspaper isn' t going to contain all of the thousands of articles that are available. Instead, we build personalized newspapers, where the stories selected are determined by an individual user 's likes and dislikes. This model for news is quite a bit different than the one that most people are used to. Currently people receive the newspaper once per day , laid out in what newspaper editors consider to be the most interesting or important to the most number of people, rather than tailored to individuals. V iewers watch television news broadcasts in a sequential, predefined order determined by the networks to be the best for the average viewer at the same time each night. News doesn' t just happen at 6:30 every night, it is continuous. Current news distribution technologies do not do a good job of getting news to people in a timely manner , unless it is of critically important.

When it becomes possible for people to receive more than just one or two local newspapers, and three or four television news broadcasts transferred electronically into their homes, this model must change. As W alter Bender said in his Straw-Man proposal to the newspaper industry , "in order to have maximum impact, the newspaper industry must provide timely delivery while facilitating personalization and advertising acuity ."[2] Advertising acuity is simply advertising which is relevant to the viewer / reader . Currently advertisements are rarely relevant to what individuals really

want or need, where in reality , there are probably quite a number of things a given person does need, and would like to see advertisements about.

CHAPTER 4

User Modeling

A large portion of the work in MIRE concerns user modeling. User modeling is building a computer representation of a user . This is not a graphical representation, but rather a representation of a person' s preferences, likes, dislikes, behavior , schedule, beliefs and knowledge. The idea is that given a sufficiently thorough user model of someone, a computer could make inferences about new things that this person might like or dislike, or projections about where a person might be located on a given day at a given time, or even ideas about what sort of activities a person might want to do in the future.

The system I based my work around is called Doppelgänger [1]. Doppelgänger is unlike most user modeling systems in that it can get its informa-

tion about users in a passive way from many sensors. For example, in conjunction with a program called Badger[1 1] and some electronic identification badges that many people in the Media Lab wear , it is possible to determine a user 's location within the lab during a given time. So Doppelgänger could use this information to know that someone always has a Monday afternoon meeting at one o'clock, and possibly tell a program to remind him of it, without him ever telling Doppelgänger directly that this is what he does.

Doppelgänger can receive direct feedback about a user 's likes and dislikes. It maintains a weighted keyword database to assimilate the users' s combined likes and dislikes. A user could tell Doppelgänger "I like cheese", and it will enter cheese into its database with a positive weight, or boost an existing value for cheese if it is already in the database. The system can be used in a similar manner to record dislikes.

In addition to direct user models, Doppelgänger extends this notion to communities. A user can build up stereotypes of likes and dislikes about a group of people. For example, there could be communities describing the group "MIT undergraduates", or "gourmet food lovers", or "men". These groups are not necessarily based on one particular topic of interest, but rather a common base of shared interests. This is an aid in initially creating user models, because it can determine automatically from sensor data,

or could be told what communities a user is involved in, or resembles most. For example, a recent periodical described a letter from one of its readers, Matt. Matt wrote to say that “he’ s into sports” and that in fact, “He’ s a member of the Conklin Y outh Sports Club.” [9] If this information was put into Doppelgänger , it could quickly build a rather descriptive user model for Matt, simply by boosting sports, inheriting from the sports lovers community , and more specifically , from the Conklin Y outh Sports Club community (if one existed). At this stage, Doppelgänger already has a lot of information about Matt.

Several tools have been developed which make use of this data specifically as it relates to news. One tool is called editor and is, as the name implies, an electronic newspaper editor . Based on a steady stream of news, editor builds prioritized lists of articles for specific user and community models. Given this information, various programs developed here at the Media Lab such as newspace (personalized paper with automatic graphical layout, supporting multi-media), epaper (an email ascii personal newspaper), pspaper (a PostScript, automatically laid out newspaper), fpaper (just like epaper , with built in Doppelgänger feedback capabilities) and of course MIRE, build personalized newspapers for given models.

CHAPTER 5

What are MUDs?

A MUD is a virtual world in which allows many users to login and interact within it. This world is analogous to the real world in many ways. A user is a "person," walking down a virtual road. There are buildings, trees, mailboxes, or whatever sort of object one would expect in the style or genre of the world they are in. (spaceships, magic swords, offices, pencils, what have you) Users interact with the MUD via a simple natural language(i.e. "get pencil", "go north", "put the third bagel in the wastebasket", etc.). MUDs contain ways to easily create and modify the virtual objects that users interact with. They can tailor the world to their own personal likes and dislikes; building rooms, objects and even autonomous "robots." This can also be seen in the fact that the

world is not static. It can constantly change and react to what users do. Some MUDs have simulated weather, night / day, and even small ecological systems within them. Since they are completely programmable, anything is possible.

Within a MUD, you might find various small subsystems, such as a store, a post office, or a local newspaper . Since it is a programmable virtual world, any kind of interface into these systems is possible. For example, a store could be designed which is like a giant vending machine which a user must insert coins into, or it could have a storekeeper robot with who users could bargain with to try to get a better price.

Each user sees other users as separate objects with which they can interact. Users can walk from room to virtual room, finding objects, and other users in each room. If the users Jon and Sally are logged on to a MUD, they can communicate with each other and even exchange virtual items. This communication is different from standard types computer communication (i.e. talk, write, bulletin boards) in two major ways: (1) each user is in a virtual location within the world of the MUD, and (2) each user is a simulated person with a subset of a person' s physical abilities (i.e. smile, laugh, cry , walk, shout, etc.). This allows communication to be more life-like, and natural. For example, to talk to a group of people, a user might walk into the same room that they're all in, and simple start talking. Since they're all in the same room, they can all hear . To say something privately

to someone, a user might whisper a message. W ider varieties of meaning can be conveyed through the use of special "emotion" commands on MUDs. For example, a user could "smile", or "grimace" in reaction to what someone says to them. This technique allows a lot of (sometimes subtle) information to be passed quickly .

A standard MUD interaction might look something like this:

```
Jon enters the room.
Sally says: "Hi Jon!"
Jon says: "Hi Sally! How's it going?"
Sally says: "Not bad. Hey, I wanted to invite you to my
party. Here's an invitation."
Sally gives Jon an invitation.
Jon smiles happily.
Jon says: "cool! I'll be there. I have to run now. Talk to
you later."
Sally says: "great. I'll see you there." Sally waves to
Jon.
Jon waves good-bye.
Jon leaves out the door.
```

From Sally's perspective, this is how the previous interaction would have occurred:

```
Jon enters the room.
> say Hi Jon!
You say: "Hi Jon!"
Jon says: "Hi Sally! How's it going?"
> say Not bad. Hey I wanted to invite you to my party.
Here's an invitation.
You say: "Not bad. Hey, I wanted to invite you to my
party. Here's an invitation."
```

```
> give invitation to jon
You give an invitation to Jon.
Jon smiles happily.
Jon says: "cool! I'll be there. I have to run now. Talk to
you later."
> say great. I'll see you there.
You say: "great. I'll see you there."
> wave jon
You wave to Jon.
Jon waves good-bye.
Jon leaves out the door.
```

MUDs are based on single-user text adventure games in the early 70's such as ADVENT by Crowther and Woods [9], and more popular versions of the same by Infocom, such as Zork [9]. The only two major extensions from these adventure games to MUDs of today are described in more detail above: (1) they have many users interacting with the same world, simultaneously (2) they are extensible by users.

Today, there are two primary variants of MUDs being used. Although there are many different MUD programs, most can be categorized as either an adventure MUD or a social MUD. Adventure MUDs are primarily multi-user versions of text adventure games, featuring quests, puzzles and monsters to kill. Meanwhile, social MUDs are primarily a place for people to socialize. Both types of MUDs have adventure and social aspects. The type is determined as to where the emphasis is in the particular MUD. MUDs which are considered social usu-

ally have limited programming facilities and are based on simple object databases. These include TinyMUDs, MUSHs and MUSEs. Adventure style MUDs are usually LPMUDs, DikuMUDs or AberMUDs.

CHAPTER 6

A Sample run through MIRE

Here is a sample run through MIRE, as a demonstration of how it behaves and how it would be used.

```
> telnet dearth 2000
Trying 18.85.1.51...
Connected to dearth.
Escape character is '^]'.


mire (MudOS driver 0.8.14)

login: erik
Password:
Inventory restored.
This is Erik's workroom. He hasn't redecorated
this room yet, so it is empty at the moment.
 Obvious exits are: out.
Erik's mailbox
```

```
bulletin board [0 messages]
> look at mailbox
This is your mailbox. You open it and look at the mail
inside...
+ 1 [Wayfarer] Hi there
> read 1
(Message 1)

To: erik
From: Wayfarer
Subject: Hi there
Date: Sun May 10 03:09:51 1992
----
Hey, did you read that article in the paper about the guy who
supposedly tried to poison someone's nasal spray to kill
them? I know it's not really funny in reality, but I just
about died laughing when I read the headline:

MAN FOUND INNOCENT FOR TRYING TO KILL MAN WITH NASAL SPRAY

The images this conjured up were hilarious! Anyway, maybe
I'll catch you logged on later today. Talk to you later.

Wayfarer
> reply
To: wayfarer
Subject: Re: Hi there
To end message type '.'. To abort type ~q. To use ed type ~e.
_____________________________________________________________
hmm... my personal paper didn't have that article. Maybe I
should tell Doppelganger that I like nasal spray huh? Did you
save it? Maybe you could show me that article. I'd like to
see it.

Hope to see you on-line.

Erik
.
```

```
Message sent!
> inventory
You are carrying (weight: 0):
 Erik's personal newspaper
> look at newspaper
This is Erik's personal newspaper. (model erikkay)
page: 1 [100 articles on 7 pages]
 [1] Quirks in the News
 [2] Russian Space Nuclear Reactors Arrive in U.S.
 [3] UPI NEWS AT A GLANCE
 [4] Names in the News
 [5] Stocks end higher in Tokyo
 [6] Top video rentals
 [7] PM MN In Brief
 [8] PM Wi Minn. In Brief
 [9] Grandson in Chaney film clan killed
 [10] Special for Pacific Daily News-Guam
 [11] British Columbia Briefs
 [12] Roddy McDowall: the picture of Hollywood success
 [13] Cineplex Odeon loses $13.9 million
 [14] Daily Life in Kabul: Emerging Islam and a Fight for
Flour
 [15] Reserved seating offered by GC theaters in Dallas-Fort
Worth
> read 1

Erik's newspaper, page 1
Source: clari.news.interest.quirks
Quirks in the News
by: clarinews@clarinet.com

Put it all together, it spells ...
 NEW YORK (UPI) -- With the second Sunday in May approaching,
thoughts come back to good old Mom and all that wonderful
advice she has to offer.
```

A Mothers Day survey commissioned by DiSaronno Amaretto asked 500 adults around the country what they considered their mothers' best advice.

``Always wear clean underwear in case you get in an accident,'' took first place with 15 percent.

``Stand up straight, don't hunch,'' was second with 14 percent of those surveyed.

``Be good to the people you meet on the way up -- they're the same people you'll met on the way down'' had 12 percent with fourth place going to ``Don't talk to strangers'' and fifth to ``Don't talk with your mouth full.''[1]

> say oops!
You say: oops!
> look at board
You can set up new notes with the command 'note headline'.
Read a note with 'read num', and remove an old note with 'remove num'.
The bulletin board has no messages.
> note REMEMBER!
Editing message.
To end message type '.'. To abort type ~q. To use ed type ~e.

Remember to call Mom for Mother's day!
.
Posted.
> A hole opens in the ceiling, and Wayfarer falls out of it.
Wayfarer smiles at you.
Wayfarer says: dude!
>say Hi there... how's it goin?
You say: Hi there... how's it goin?
>say btw... I couldn't find that article on nasal spray, did you save it by chance?
You say: btw... I couldn't find that article on nasal spray, did you save it by chance?

---

1. the original story was longer, but was truncated here for clarity

```
Wayfarer pouts.
Wayfarer says: nope... sorry
Wayfarer says: figured you would have gotten it.
> say that's alright
You say: that's alright
> say I'll just tell doppelganger that I like nasal spray. If
some other news service puts the article out again, then I'll
be sure to get it!
You say: I'll just tell doppelganger that I like nasal spray.
If some other news service puts the article out again, then
I'll be sure to get it!
Wayfarer falls down laughing.
> tell doppelganger I like nasal spray a lot.
Doppelganger tells you: Ok. I'm boosting "nasal" and "spray"
for you.[1]
> smile
You smile happily.
> say ok
You say: ok
Wayfarer says: hey man... your room is boring! why don't you
make it look cool?
> look
This is Erik's workroom. He hasn't redecorated
this room yet, so it is empty at the moment.
 Obvious exits are: out.
Wayfarer is thesing night and day
Erik's mailbox
bulletin board [1 messages]
> think
You think carefully.
> say in all honesty, I don't know how
You say: in all honesty, I don't know how
Wayfarer says: just type 'help workroom'. remember, you can
get help about just about anything here
> say right...
```

1. This conversation with Doppelgänger was simulated. All other parts of this transcript were from an actual MIRE session.

```
You say: right...
> say I should have thought about that
You say: I should have thought about that
> help workroom
-----------------------------------------------
Help for workroom (Object Help)
-----------------------------------------------
You are in a default MIRE workroom. The workroom is your home
on MIRE. It is where you can keep your newspaper, read your
mail, write notes on your bulletin board, and have private
conversations with users. It supports the following com-
mands:

lock lock your workroom to outside people other
   than yourself.
unlock unlock your workroom.
redecorate change the basic description of your
   workroom.
> redecorate
Just type in the description you want your room to have.
To end message type '.'. To abort type ~q. To use ed type ~e.
```

```
You have entered the palace belonging to Erik. This is his
throne room, and all who enter, must kneel before the throne
of the great Erik. The room is filled with gold and fabulous
jewels, a sign of Erik's obvious wealth and influence.
.
You finish redecorating your room. Doesn't it look Maah-
valous?
> look
You have entered the palace belonging to Erik. This is his
throne room, and all who enter, must kneel before the throne
of the great Erik. The room is filled with gold and fabulous
jewels, a sign of Erik's obvious wealth and influence.
 Obvious exits are: out.
Wayfarer is thesing night and day
Erik's mailbox
bulletin board [1 messages]
```

```
> grin
You grin evilly.
Wayfarer says: a bit pretentious aren't we?
Wayfarer pokes you in the ribs.
> nod
You nod solemnly.
> say but of course
You say: but of course
Wayfarer says: hey... I wanted to show you this article in my
paper...
Wayfarer says: come to my workroom, and I'll show it to you
> smile
You smile happily.
> say ok
You say: ok
Wayfarer's mother comes and drags him home.
You are summoned to appear before Wayfarer.
> say hey!
You say: hey!
Wayfarer grins evilly.
Wayfarer says: well... you were taking too long
> look
This is a small, and cluttered office. Piles of paper and
dirty laundry lie all over the floor. Over in the corner you
see what you assume is a desk, but you can't be sure, because
the only thing you can tell about it is that it is a slightly
larger pile of papers. There also appears to be a black com-
puter sitting amidst the mess of papers. This mess can only
mean one thing: Wayfarer has been working on his thesis and
not on cleaning his room.
 Obvious exits are: out.
Wayfarer is thesing night and day
Wayfarer's mailbox
bulletin board [3 messages]
Wayfarer says: type "customize paper"
> customize paper
Newspaper customized to user model wayfarer.
Wayfarer says: turn to page 7
```

```
Wayfarer says: article 99
> say ok
You say: ok
> page 7
This is Erik's personal newspaper. (model wayfarer)
page: 7 [100 articles on 7 pages]
 [91] Cardinal urges Filipinos guard against vote fraud
 [92] L.A. police quash protest; residents seek government
aid
 [93] Brown Concludes Two-Day Swing Through State
 [94] Wilder, Warner Hear Pleas for Urban Funding
 [95] L.A. police quash protest; residents seek government
aid
 [96] Louisiana native killed in LA rioting given rural
funeral
 [97] Bank severs Parretti from MGM-Pathe
 [98] Nation's retailers report lackluster April sales
 [99] Rodney King's words put to music
 [100] More than 150,000 people attend Cojuangco rally
> read 99

Erik's newspaper, page 7
Source: clari.news.features
Rodney King's words put to music
by: clarinews@clarinet.com

 BALTIMORE (UPI) -- Brent Hardesty took Rodney King's pleas
for an end to the violence and put them to music to offer ``a
song of healing for the nation in the aftermath of what hap-
pened'' in Los Angeles. ``Love Will Find a Way'' debuted Fri-
day on a local radio station with lyrics by King and music by
Hardesty.[1]

> smirk
Riker would be proud!
```

1. Rest of article deleted for clarity

```
Wayfarer says: can you believe that people are already cash-
ing in on this thing?
Wayfarer says: I mean... here we have a really nasty thing
that happened, and people are making money off of it
Wayfarer sighs deeply.
> say yeah... he really sounded full of it, didn't he?
You say: yeah... he really sounded full of it, didn't he?
Wayfarer nods solemnly.
Wayfarer says: yep
> say anyway, I have to go. Just thought I'd check in
You say: anyway, I have to go. Just thought I'd check in
Wayfarer says: ok!
Wayfarer says: talk to you later man
Wayfarer waves.
> wave
You wave.
> quit
Come back again soon!.
Inventory saved.
Connection closed by foreign host.
dearth:[~]
>
```

Jon sez: put some notes about the seesion in italics on the right.

walter sez: observations here?

CHAPTER 7

The MIRE driver / parser

The driver is the underlying program that accepts remote connections from users, and interprets the world database and user actions, to make a MUD run.

Selection of driver

During the planning stage, I reviewed the needs of my driver . It needed to be flexible, powerful, easy to create a world in, and provide all basic MUD services (parsing, multi-user network connectivity , etc.). In addition, it needed to be able to talk to the Doppelgänger and newsdaemon services that I would use to retrieve news, and to provide a mudlib code interface to access these services. Looking at these requirements, I had two choices: write my own driver completely from scratch or take an existing driver

and modify it to deal with Doppelgänger and newsdaemon. I decided that while writing a driver from scratch would probably lead to an underlying architecture which would lend itself more to MIRE's special needs, but would take too long to get into a usable state, given the time constraints of the project, so I decided to modify an existing driver.

The next question was which driver to take and modify. The requirement here was that it must be fairly straightforward to modify, and have a relatively flexible architecture which would allow the game to be modified so that it was not a game anymore (still like a game, but things like monsters, combat, treasure, etc. should not be tied into the driver).[1] There were a number of existing drivers to choose from (LPMUD, TinyMUD, MUCK, MUSH, UnterMUD, DikuMUD, MUSE, MUF, and several others). My choice was mainly influenced by the fact that I had previous experience with the LPMUD driver, and thus was already familiar with the underlying design of it. In addition, I was already an experienced LPC[2] programmer[3], and thus world creation would also be easier. Finally, LPMUD was simply the most flexible and easy to modify of the drivers I looked at.

1. This is not to say that combat or similar fantasy concepts have no place in MIRE. For example, virtually 'killing' a person is a sure way to convey certain information to that person.

2. See appendix C

3. LPC is an interpreted, object-oriented C-like language (similar to both objective C and C++ in several ways) which is used to create / describe / modify the world in which users interact

Cleaning up the driver

Planning

The first stage of modifying the driver was to make it as versatile and flexible as I could make it. The version of the parser that I was working with (I started with 3.0.48), was still a beta release which was constantly changing, with numerous bugs that needed to be fixed. When the driver made it to release 3.0.50, the author (Lars Pensj|) slowed his development of the underlying driver code, and stopped responding to bug fixes and feature additions to the driver (several months later , he released version 3.1 and said that he would make no more modifications other than bug fixes). However , my driver was still falling short in a number of categories. It still had a number of rather serious bugs, and was missing a few features which I found frustrating. Specifically , I was working on multiple versions of the underlying mudlib (the LPC source code for the MIRE world) for debugging purposes. I wanted to be able to use the same driver for both mudlibs and run them on the same machine, but on separate port numbers. However , this flexibility of design didn' t exist. In addition, hard-coded pathnames for files were common throughout the driver . The driver design was too closely tied to a specific layout of the mudlib. Since I wanted a completely different layout for the mudlib than the one hard-coded into the driver , this needed to be fixed as well. Another thing I wanted to provide was a way to talk to smart clients. Specifically , I wanted

the ability to send formatted text, graphics, and even pointers to video information to a client if I wanted. Again, at the driver level, this didn' t exist.

In addition, there were a number of small convenience functions which I found to be missing from the driver . Specifically here, I wanted to have a way to get single character input and a way to directly notify the mudlib if a user broke his connection to the driver unexpectedly .

Doing it

I began working on a number of these subsystems on my own. My first project was runtime configuration. The driver had an include file, config.h, which was filled with compile-time configuration constants. I replaced this system with a customizable configuration file which was parsed at runtime. This system gives the user complete control over all of these parameters, so it was easy to quickly modify one parameter and debug the system under a host of different configurations quickly and easily . In addition, it was now possible to use the same binary to run multiple worlds with completely different needs and configurations on the same machine. This was an invaluable aid in debugging for me.

At this point, I met with a group of other people who were also doing some work on the driver , including some people on a MUD called The Mud Insti-

tute. It is an LPMUD devoted to teaching people how to program in LPC, and it became a testing ground for some modified game drivers. They were doing some work that was interesting to me, including many bug fixes to the driver , mappings (associative arrays) and the addition of printf and sprintf functions. I decided that for at least my generic modifications to the driver , we were working towards similar goals, and I worked in tandem with them. In addition, they provided additional feedback and testing of my modifications. The majority of this underlying work was done by two people plus myself: Whiplash and T ruilkan. Also L ynscar provided the sprintf package which Whiplash incorporated into our combined work. [1]

After approximately a month of sporadic work with T ruilkan and Whiplash, the driver got to the stage where I felt that it was much more stable than before, a bit more efficient, and had increased functionality and flexibility , and that any more additions made at that level were unnecessary to the final project. At this point I began to work on the additions which would make the driver into a MIRE driver , and not simply an enhanced LPMUD driver .

1. Since that time, several others have contributed work to this base underlying driver: Huthar, Jacques, Shadowhawk and Sulam

MIRE additions

Design and implementation

The first step was to identify all of the additions which were necessary to the driver. I needed to be able to fetch articles, to get a list of article id's in sorted order in accordance to a user model, and a way to provide feedback to Doppelgänger.

This is where the main problem in the driver design appeared. The problem is that the driver, although it supports connections from multiple users, it is not multi-threaded. It simply polls every user for input and acts on that input sequentially, does a bunch of internal functions, and then loops back to polling every user one at a time. The problem with this strategy is that if a particular operation by a user or the driver takes any significant length of time, every user on the MUD is stalled until that operation is completed. To alleviate this problem, the driver has a upper limit on the number of LPC instructions it will execute in a single cycle without giving an error. However, this analysis is based on number of instructions, and not total evaluation time. So if there is a single LPC efun [1] which takes a long period of time, the driver will devote all of its resources to it and make every other user wait. The advantage to developing it this way is that it is

1. An efun is an external, or driver function. This means that it is defined in the driver, and that any object can use it. C programmers might think of it as a library function.

much more portable since there are no system-specific multi-threading functions used in the system. Also, the driver was developed on a SUN, which currently has no C threads package bundled with it. The disadvantage is that functions which take long periods of time are not usable.

This is more than just a minor inconvenience for the development of MIRE. Specifically several of the functions I need to execute can potentially take a long time: fetching articles and fetching a sorted list of articles. Fetching articles is in general very fast, but under some circumstances can block for a long time (if the newsdaemon is rebuilding its database or it is swamped with a large number of requests or submissions). However, fetching the sorted list of article headers is very slow. It can take several minutes just to get the top one hundred article id's for a user if he has a very complex user model.

I was faced with three options:

1. build a multi-threaded system
2. preload every article and model list in advance, and disallow new loading except for on start-up
3. make every user wait when one user makes a lengthy request.

I decided that option three was not the right answer. It would work, but would make the system completely slow. The second option would work, but it wasn't flexible enough, and would have been very difficult to debug since the driver would have a very lengthy start-up time. This left me with the first option which would be more difficult to program, and would not be

portable from architecture to architecture, but was the cleanest implementation of the three; leaving me with a driver which was easy to debug and flexible and powerful for users.

This selection forced my decision on which machine I would be developing the driver on as well. Of the machines that I had available to me, the only ones that had any type of multi-threaded capabilities were the NeXT computers in the Garden. [1] By preference and experience the NeXT is almost always my machine of choice, so this did not pose a problem. The NeXTs have Carnegie Mellon's cthreads package, which were developed in conjunction with MACH, the kernel used on NeXT computers. Cthreads is a library of functions for doing multi-threaded programming. It features multiple threads, mutex locking, and condition signaling.

The primary design consideration was to make the system so that the driver itself would never get hung during any operation. Even if it were performing CPU intensive article searches, the users of MIRE should never wind up waiting.

My original design had two threads of execution. The first thread would be the primary driver thread. This would be identical to the original driver process. Ideally, the driver would have been split up into smaller threads

1. The section of the Media Lab in which I work

(perhaps one for each user). This would not be possible without a significant changes to the driver since it currently relies on a number of global variables which would cause serious problems in most useful threaded splits of the driver . The second thread is the electronic editor and article fetching thread. It does the job of fetching the sorted list of article id' s from Doppelgänger , and then getting those articles from newsdaemon.

Since the primary thread interprets the efuns which make editor and article requests, and the second thread handles these requests, the next step to consider was communication between the two threads. The secondary thread waits until it receives a signal from the primary thread. The efuns in the primary thread don' t actually fetch the article, or the list of article id's, but look to see if they were already loaded into memory . The efuns return the values if they are loaded, otherwise they add a request to a queue and send a signal to the other thread. If the secondary thread is waiting, this causes it to start processing the queue. If not, it simply does nothing. All the secondary thread does is wait for a signal, and then process a list of requests in a queue, loading the requested data into memory .

This is a fairly simple design, and it gave me something good to start with. I built the thread structure so that the primary process created and detached the MIRE thread, and initialized a lock and a condition, the queue arrays, and the storage variables. It then went into condition wait. I then wrote a couple of functions to test the process. One would fetch an

article from newsdaemon given an article id, and the other would fetch a sorted list of article id's for a given user, given a user model name. These worked as planned, so I wrote basic efuns to access them. The first, `editor_list`, calls the Doppelgänger library function editor with the username given and returned a sorted list of article id's, sorted by model preference. The next function, `fetch_article`, does just what it says: it tries to fetch an article from newsdaemon. Actually, as was mentioned before, neither of these functions did their actions directly, but rather through a queue. The only real problem with these efuns was that the data they returned internally, was in the form of C structures: multi-part data. So I had to convert these from structures to arrays since LPC has no structures. This makes the behavior of the efuns not very programmatically clean, since it relies on programmers knowing the order of the data within the arrays. These were small problems, and the efuns worked with very little debugging. An additional efun, `remove_editor_list`, was added so that users could remove their `editor_list` from the internal cache and force it to be reloaded so they could get the most up to date news for their model. This is extremely useful when users are updating their models frequently and want to see immediate feedback. Also, it allows users to guarantee that they have the most up-to-date news.

The next efuns which were written were so that feedback could be sent to doppelgänger from within MIRE. `doppel_mod` was the first of these and

would allow direct modification of a user model. It takes a model, a keyword, and a modifier . This caused the weight of the keyword to be changed by the amount of the modifier (either positively or negatively) directly in the model. The next efun, `find_keywords`, was designed to be used hand in hand with this. The function takes a buffer of text and calculates the number and weight of keywords within an article. Combined, these two efuns allow the user to tell doppelgänger that he likes the article as a whole, and boosts every keyword. Separate, they allow the user to modify individual keywords and observe what effect boosting an entire article would have.

After putting these efuns all in place, I found that there were some problems with the strategy that I had chosen. The problem was that editor requests take an extremely long time, and the queue system processed all requests in one queue before moving on to the other . So in other words, I had a long start-up time before I could actually get articles, because when I initially started up MIRE, I had it automatically preload article id lists for various models. So until all of these calls to editor were being made, I was unable to get any articles from newsdaemon.

The design I came up with to counter this was simply another extension on the same principle I was working with. I just split up the fetching thread into two parts: one which got article lists, and one which got articles. When either queue was added to, it activated the appropriate thread. This seemed to work fine, however , it occasionally got into a deadlock problem

with threads. I believe it was simply related to the fact that I was using two related underlying systems simultaneously which were not designed to be reentrant [1] and thread-safe. The system seemed to work well, but intermittently the article fetching thread would suddenly lock up. Since I had two threads fetching articles (the actual article fetching, and editor which was building the sorted article list) perhaps there was some conflict where both threads were simultaneously trying to talk to newsdaemon. With a little work, I'm sure that this deadlock could have been worked around; instead, I chose an easier solution due to time constraints. Ideally , I think that this is the solution.

Enter sub-design number three. This was a move back to my first design, with one addition: editor requests were handled only one at a time, and then all article requests were handled. This way , the long wait for initial retrieval was alleviated. There is still a small delay in the system, but for the most part, this design forced the two subsystems to share the time resources more equitably .

1. able to be called recursively without side-effects

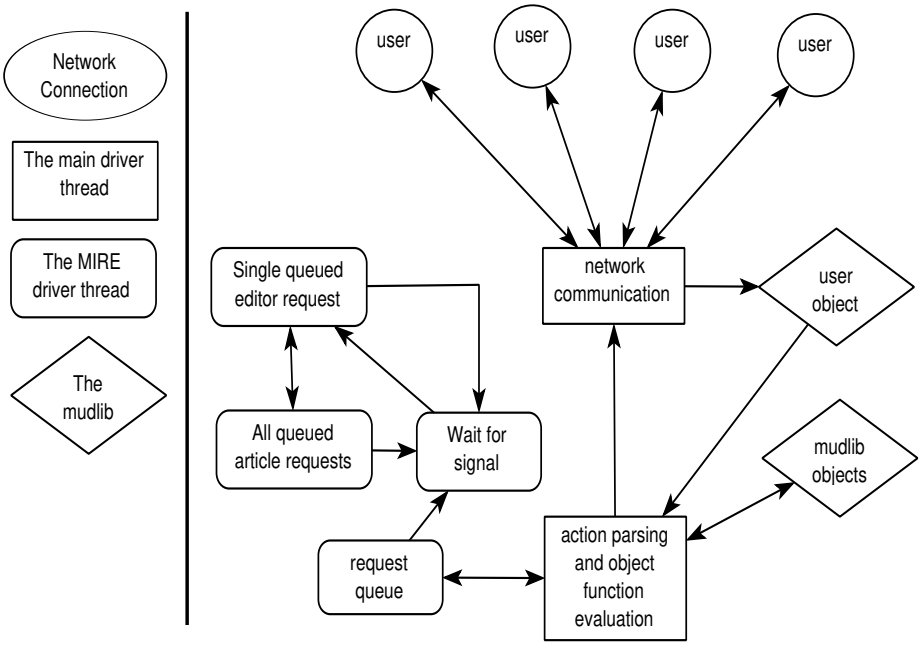

Diagram of MIRE input evaluation

Future work

There are a number of efuns which would be ideal to add to the system in the future. Specifically , a `fetch_model` efun which allowed users to look at their user models directly , and thus be able to modify specific parts of it as they saw fit. A `submit_article` efun would also be useful so that users could write their own articles which would be submitted to newsdaemon as articles for other users. Finally , some sort of model creation facility

would be a nice enhancement. One in which groups of people could get together and just build a new community would be ideal.

Also, the efuns which I wrote are neither very fast nor memory efficient. All searches are done in $O(n)$ time, and take up much more memory than is necessary. Increasing the efficiency of these functions would lead to a much more usable system.

One further addition would be to thread a few other efuns which are time consuming. `find_keywords` takes a long time on long articles. This is a problem that I don't have a solution for at the moment.

Article fetching done in the second thread using the queuing technique described above. All articles that are loaded, are cached locally in the MIRE driver. I did this originally because I believed that fetching large articles could block, and cause wait states throughout the whole system. This might be true in some extreme cases, but in practice I found newsdaemon to be very fast, and the blocking problem not to be a significant issue. I would like to replace this article fetching system in use now with an uncached system, making direct calls to newsdaemon on each request. This would probably improve memory performance in the long run, and would give more time to thread to fetch editor lists. I still don't know which design is better in this case, but I believe that it would be a good idea to at least experiment with this alternate approach.

One problem that I ran across with the threaded / queued request system is that I was unable to notify mudlib objects of completion of their requests. I had to rely on the objects polling for data occasionally to see if their requests were completed. The problem was that if I called a function in an object from the first thread, there was a moderate chance that I would get data conflicts (the other thread might be evaluating a function in that same object). In the news fetching code, I used mutexes [1] to solve data locking problems, but in this case it was a little more difficult. I had no good way of setting up mutexes to prevent data stomping because I did not know in advance which objects would be calling the fetch efuns. I would like the system to be able to notify objects directly upon completion of their requests.

One point which shouldn't be overlooked is the fact that these threaded solutions are being proposed for entirely the incorrect reasons. The reason they were necessary is because of the inherent design flaws in the driver that I chose. It is not multi-threaded, so any time-consuming process performed by one user causes every other user to wait for that one operation to finish. If one were to follow my approach for all functions which could potentially take a noticeable length of time, the game would suddenly be filled with a thread per efun. The correct approach is to make the driver inherently multi-threaded, and to give each user its own thread of control

1. a special flag which is used to mark data as being used by another thread

so that they can in fact perform lengthy operations without stopping everyone else. Though it would stop them, it would have no effect on others. This would be the clean and correct solution, but time constraints did not allow me to take this route.

CHAPTER 8

The MIRE world (mudlib)

Influence of the driver

A lot of people have been programming MUDs for a number of years now , creating a lot of MUD code. Most people who are starting a new MUD take existing code, and use it as the basis for their world, making modifications as they see fit. I would have done this myself; however , there were three major problems that I faced:

1. Most mudlibs[1] were so closely tied to the game for which they were originally designed, that trying to use them for even moderately different reasons required almost a full rewrite.
2. Most MUD code is written by people who haven't had formal training as programmers. As a result, it is usually hard to read, inefficient and very hard to extend.

1. the mudlib is the LPC code which defines an individual MUD

3. The driver that I had chosen for MIRE's base, did not have a good base mudlib anywhere. The driver was newly released (within a couple of months) and was still changing. It had not quite settled down yet, so very few people were writing mudlibs for it.

Given these problems, I decided that it would be better to take the time to work on a good underlying mudlib than to try to adapt someone else' s, since theirs would either be incompatible with the current version of the driver , or in a basic development stage. Furthermore, it would almost certainly be so intertwined into a game that it would be difficult to modify . So the state of the driver influenced the approach that I would be taking. Because of its relatively new arrival, the only real solution was to write a new mudlib from scratch.

Starting the project

When I started the project, this part of the project was actually well underway. As a hobby on the side, I had begun working on a MUD on my own in November . After a couple of weeks, I had recruited several friends to help me with the project. Since I am using our work as a basis for MIRE, some code is there as a result of their help. Huthar was my co-administrator of the initial project and he has the next highest contribution of code. There are other pieces of code from Mr .X, Imaginos and Sirrobin in the mudlib. So when I decided to base my thesis on this work in December , a good deal of work had already been completed by myself and Huthar . So let' s go back

to the actual start of the project before I began working on my thesis, because the initial design goals were the same. I wanted a mudlib which was generic, simple, powerful, and easy to debug and modify .

Design goals and their implementation

- Generic

What I wanted was a base mudlib which could be applied to any MUD. It wouldn't have anything specific to the game element of the MUD within it, and it wouldn't have anything which was world-specific. For example, no combat, magic or skills (elements commonly found in game mudlibs) would be in the base mudlib. This would mean that the code I wrote for the base mudlib could be taken and applied to any situation I wanted. Specifically , it meant that I could both be working on a game and MIRE, and use the same groundwork for the project.

- Simple

Where simplicity affects the design of the mudlib is in the object hierarchy . The idea is to make sure that no one object is more complicated than necessary , and that inheritance is used to its maximum abilities. So each object would be designed to contain minimum functionality , and larger objects would be made by inheriting from lots of these tiny objects.

The other target of simplicity in design was file system layout. An LPMUD mudlib is a collection of object files in a set of directories. The parent direc-

tory for the mudlib appears as '/' in the MUD filesystem. I went through many different file system layouts, but this is the one I finally arrived at:

```
/                   // the root directory
  ->adm             // administrative files
    ->data          //data files (mail, bulletin boards, etc.)
    ->etc           //configuration files
    ->log           //log files
    ->news          //news and announcements for users
    ->obj           //administrative objects
      ->simul_efun  //simulated efun files
    ->tmp           //tmp dir is primarily for the swap file
  ->attic           // outdated mudlib files
  ->bin             // user commands, plus other automatic
                    // programs
    ->adm           // admin-level commands (shutdown, etc)
      ->daemon      //daemon-like objects (newsd, fingerd, etc)
    ->dev           //developer-level commands
      ->file        //file access commands for devs
    ->extras        //misc commands meant to be added
                    // individually rather than as a part of a
                    //path
    ->soul          // 'soul' or 'feeling' commands
    ->user          //user-level commands
  ->d               // 'domain' directory (domains are sets of rooms)
    ->developer     // the developer work domain
    ->mire          //the mire domain
    ->standard      //the standard or default domain
  ->obj             // the standard set of clonable objects
  ->tmp             // temp space for stuff
  ->u               // user home directories
  ->usr             // programming stuff
```

```
->doc          // documentation for efuns, lfuns and help
->include      // header files
->lib          //standard object building blocks
  ->living     //objects for 'living' objects
    ->user     //    pieces for the user object
  ->room       //room objects
```

This is a fairly simple hierarchy . It leaves room for logical expansion in /usr/lib which is simply the object inheritance tree and in /d which is for all of the rooms which make up the worlds. All of the other directories are fairly static as is.

- Easy to debug and modify

This ties closely into the simplicity of organization. The more that the objects were split up and modularized, and the better the inheritance structure, the easier debugging was, and the easier it was to modify the mudlib.

One key example of this is user commands. T raditionally the way a user is given a ‘command’ (that is, an action that they can perform) is for the user or ‘player ’ object to `"add_action"` verbs to a player . This causes a function to be called whenever a user types that verb at the beginning of a sentence. T o each user has a list of verbs that are trapped by itself or objects in their possession or objects in the same room as them. For example, to implement the verb ‘look’ (as in ‘look at door ’), traditionally , one would have this line in the user object: `add_action ("look_function",`

`"look");` This calls the function `look_function` in the user object whenever a user types 'look' at the beginning of a line. However, this means that every command that you want a user to have as a default has to be in the user object. This makes one very large object which is difficult to maintain, update, and debug, and which takes a lot of memory.

We took an alternative approach in the mudlib by trapping all verbs generically through the user object, and then pass it through a preprocessor (so that things like nicknames, aliases and history were possible) [1] and then delegated each command off to a smaller object. Each command winds up being its own separate object. That way, every user shares the same object for commands, and only commands which are used are actually loaded into memory. It also makes command addition trivial, since all you need to do is add an extra file. You could even have multiple people modifying user commands simultaneously since they don't have anything to do with each other. One further feature of this design is that debugging becomes much easier when each function isn't in the user object. Since every user depends on the user object for all of their actions, modifying it directly can cause serious problems if and when errors are made. However, since the functions are independent of the user object, breaking them or adding new ones only affect that particular function and not the whole user object. [2]

1. nicknames allow users to use shortcuts for usernames, aliases are shortcuts for commands, and history is a record of every command a user has typed

One other example of this modularity is in the help system. Rather than using a centralized help database as is most common, the help system is distributed. There is a central help command which asks objects to tell it what their help information is. So for example, if I say 'help newspaper ', the help system looks for a newspaper in my possession, and asks it to display its help. Alternately , when I say 'help look' it might instead find the user command 'look' and ask it for help about itself. This means that there is no centralized database that is necessary to maintain. Help winds up being a lot more flexible, and a lot less complicated this way .[1]

MIRE specific additions

The MIRE specific additions were quite small to the mudlib. As I mentioned above, it was designed to be generic and to fit for many purposes, and it succeeded at that. Specifically for MIRE, I made the following custom objects: newspaper , news room, customizable office, mailbox, user registration system, newsdaemon and a group of rooms, objects and automatons which make up the MIRE world proper .

2. The inspiration for this idea came from Buddha, Sulam and Cynic at TMI

1. Mr.X helped clean up some initial design problems, and helped with the implementation of this system

The newspaper uses all of the MIRE-specific efuns. It builds a newspaper from a user model and allows them to read their news, and to provide direct feedback to Doppelgänger about their news. It also has the ability to talk to a news room so that it can determine the model associated with the room, and to customize itself to that model.

The news room is simply a subclass of the standard room object which has a user / community model associated with it. This is designed to work in conjunction with the newspaper object so that users can customize the news they get to the room they're in.

The customizable office is also a subclass of the basic room object. Its purpose is to provide a home office for users to enter and leave the world from, and to read their personal mail, and keep around notes, etc. It is also a place to talk to other users in private, or to compose long messages or posts to discussion boards. In most LPMUDs, workrooms are written by the people themselves in LPC. Since MIRE is not intended for only programmers, but an individualized office is somewhat important to get users into the creation aspect of the world.I wrote the office so that it could be "redecorated" with no coding. In addition, to support privacy , an individual can lock his office so that other users cannot enter it.

To give users the ability to send and receive private email within MIRE, each office contains a mailbox. This is a personal mailbox tailored to the

user: only that user can use it to send or read email. Since all of the users in my test group are in a local net and can just as easily send net-mail, this isn't entirely necessary, but it rounds out the communication spectra available in MIRE.

Rather than create users' accounts by hand as needed, I developed a user registration system, where the user types in a username. If this is not a valid name, the system attempts to try to register them as a new user. The registration system asks a number of questions which allow it to make a registration file, which another program, `nu` (for new user creation), uses to create the user accounts automatically.

The last real support object specific to MIRE that I made was the newsdaemon. This shouldn't be confused with the much larger real program which fetches news for users from an inverted dictionary database. Rather, it automatically pre-fetches a list of models in the background on start-up, so that users' papers will be pre-loaded when they log on.

Finally there is the MIRE domain. This is the set of rooms / automatons / objects which make up the actual MIRE universe. The setup that I made was fairly simple, and merely stands as a demonstration of what is possible. However, it provides users with their own personal rooms, rooms which give them access to community models, and an example direct interface to Doppelgänger.[1]

1. For an example of the MIRE world, see appendix B

CHAPTER 9

Future W ork

Powerful clients

Currently , my interface to MIRE is strictly text based. This need not always be the case. The potential exists here for me to contradict my earlier convictions along the lines of CLIs and suggest that a GUI to MIRE would be a good thing. So I will skirt this issue temporarily and deal strictly with information and not interface.

Text, although very descriptive, is definitely not the best way to describe all things. This is especially true in news, where information frequently is an image (photos, maps, diagrams, etc.), audio, or video. I could build a text-based client which interpreted the data stream from the MUD and looked for special sequences denoting graphics or video. Huthar and I

developed some base support for this functionality in the driver . Due to time constraints, no example was implemented.

To get back to the issue that I skirted before, I do not believe that having a GUI connected on top of a CLI is a bad thing. As long as the CLI is still available as a primary form of interaction, no power is lost, and there is a potential gain of some useful abilities. For example, there are many different kinds of messages in MIRE: descriptions, conversations, news, etc. Perhaps an intelligent graphical client could have multiple windows for each of these kinds of output. A graphical client could automatically be building up a graphical map of the MIRE world. Rather than having to remember complex coordinates, paths or room-names, a user could simply click on a desired room, and it could either teleport the user there or navigate a walking path back to the room. While I believe that the MUD style of CLI gives you inherently more power and relative ease of use, there are some things that GUIs do better and faster , and having a combination of the two interfaces could lead to a 'best of both worlds' situation.

Multiple MIREs

The possibility of multiple MIREs existing and talking to each other could lead to some interesting possibilities. This is the direction which things would have to proceed towards at this stage, if they were to proceed at all.

Primarily because if MIRE is to be thought of as a public information access system, the potential users include the world population. Obviously a single-server model would not work in that scenario. The more users MIRE would get, the more individual community sites it would have. How the MIREs would be connected and would talk to each other is another issue entirely . Regardless of how that happens, the following possibilities become either new or more interesting ideas when multiple communities are added.

Public discussion forums

One further extension of MIRE would be to build a system of public forums similar in function to the USENET in which people talk about topics with a broad group of users. One could walk into the AI room and hold debates and discussions there with people from around the world in a kind of forum. This possibility already exists within the MIRE world since these tailor -made rooms are already set up and ready to go.

Since this is more of a direct analogy to the USENET and other discussion groups, it is useful to address the problems with USENET , and maybe look at why public forums in MIRE would be better . Since the whole environment is programmable, forums can be created quickly and easily in many different forms. They could be moderated, in the form of a discussion panel or whatever organizational structure you want to have happen. Forums

could be held in real time, avoiding the propagation delay problem in the USENET . Such forums such as this could be logged in full, to provide information to users who didn' t attend. Of course, the traditional style of message posting and follow-ups would exist in concurrence with these real-time discussion groups. Users could still "post" information for group consumption, these articles could be treated simply as another news source.

Volume and noise level reduction

One problem that the public forum side of the program might introduce is one of volume and the high amount of "noise" and "flame" which goes with it. These are problems which are common now on the USENET , and will have to be addressed soon in some fashion, as more and more people begin to use it. Even now , the number of people involved in many discussion groups has become so large that many of the 1400+ USENET groups in existence have more than one hundred new messages per day . Since the majority of messages are noise or flame messages, finding useful information within that high volume is difficult at best.

With a combination of the direct feedback this system has and the public discussion and posting areas in a system of many MIRE servers, a new way of dealing with distribution problems would become possible. This idea is a bottom-up filtering system. Distribution of all articles would

begin at a local level, where what a user writes is distributed to the people in his direct area (on his local MIRE server). In each local area, the system can use the feedback to determine which users' articles are liked and read the most. The more a person's article is read, the more popular he could be considered. This popularity could be rated using some sort of point system which increases and decreases over time depending upon the frequency and positive or negative feedback about one's postings. So, users who become more popular would begin to have their posts distributed on a larger scale, where they could be further evaluated and rated. This would result in a national or international distribution of only articles which a large segment of the population finds informative or interesting, and thus the large scale signal to noise ratio in such arenas would improve.

The obvious danger to this system is the squashing of the minority opinion. News, although less noisy, could be definitely biased towards the opinion of the majority. I say 'could' and not 'would' since typically the minority opinion attracts enough interest from the majority that it is still read. Potentially there might be no problem with minority interests being distributed. Another problem is the generic assumption that simply because a user has written fifty good articles, that the fifty-first will be good as well. There is no guarantee. It would take time for modifications to people's 'level' to take place. During that time, either lots of users could have been

flooded with a lot of bad articles, or a large number of users could be missing out on an incredible first-time article.

Community T ailored News

Another possibility in a system of multiple MIREs is that each system would have a certain amount of 'local' news, which is local to the specific region that the MIRE server serves. The benefit of an environment like this is that people could quickly and easily get news which is more tailored towards their homes, and by the previous filtering idea, it would only filter up as long as the news was still somewhat local, since local news is rarely interesting to outsiders. Also, it provides people with a good way to keep in touch with their homes. For example, a student who is away at college could simply login remotely to their hometown MIRE server , and get all of the local news, as well as be able to talk to friends and family from home.

More powerful user monitoring interactions for feedback

The combination of the MUD interface and the types of information being managed could lead to other interesting possibilities, specifically in the way that Doppelgänger gets information about a user 's likes and dislikes. There may be a more direct way to obtain user feedback from articles. For

example, there are many of simple physical metaphors for how to get rid of an article after it has been read, which could supply the user modeling system with valuable information. For example, a user could crumple an article up and throw it away , or even burn it. If the user wanted to save it, he might fold it up and put it in his pocket for later perusal. Each of these actions could tell the system something about a user 's interests, likes and dislikes. Information could also be gathered directly from the public forums. In addition to knowing that you burned an offending article, the program could monitor your conversation, and determine exactly what about the article you didn' t like.

New drivers

The driver that I used is still lacking in many respects. A driver either more directly tailored for MIRE, or a more powerful and general driver would be desirable. A multi-threaded driver with supported for distributed objects and intercommunication between other drivers would be a nice improvement over the current driver .

CHAPTER 10

Conclusion

Some might look at MIRE and decide immediately that work based on top of a game is frivolous and has no real application to real work. It might be interesting and even thought provoking, but as far as being used in the real work environment, many might believe that more mainstream programs and operating systems will continue to do what they're doing now. This is perhaps the major limiting factor for MIRE: it is based on top of a game, so some won't take it seriously. However, if you forget that one fact about the system, it is easy to make some very interesting observations about the potential about such a system.

Let's take a look at where user interface is headed, and compare it with what already exists. More and more, the emphasis in user interface is for

making things easier for the end user . This is where the idea for a GUI came from, and why the Macintosh computer is so popular today . However , GUIs have sort of hit a stabilizing point. There have been no significant changes to how GUIs work in the last five to ten years. So what' s the next step in interface? A number of computer vendors either have available or are working on development versions of speech recognition systems (Simon Says for the NeXT , Caspar for Macintosh). So how do they expect a user will interact with these things? W ill they say "double click on the icon"? or "rm -rf *~"? Probably not. People will want to speak in their native language or close to it. They'll want to say things like "send mail to Jon" or "throw paper in the trash". If you look at MIRE, this sort of interaction is already there at the text level. W ith speech recognition, the upgrade would be near seamless. For other interfaces, this is definitely not the case. GUIs would require a complete language parsing, and document / application association system.

Another direction that computing is headed is towards networking, and group projects through networking. Most high tech businesses now have large networks installed in their companies connecting their computers together . Even many other types of businesses are moving towards networks. Most simply use the networks as a way to share data and hardware. The revolution here is happening with project work across the networks; people are using exactly the same information in many different

departments at exactly the same time. This is something which is designed into MUDs like MIRE to begin with. Multiple users log on and share the same information, and interact in the same world. If all of their software was within the context of a MUDs virtual world, they could all directly interact with each other and share the same data simply and easily. In other words, it' s already there.

A popular buzzword these days in computer science is "virtual reality ." The idea is to have a computer generated world in which you can walk around in and manipulate computer generated objects, and look at computer generated scenery . People are saying that this will provide better ways of training people to do work, and may even provide a simpler interface to dealing with computers since they could do things they would normally do in the real physical world, as a computer representation. For example, you could write a virtual note on virtual paper with a virtual pen, rather than typing into a word processor . Or you could put a virtual letter into a virtual mailbox rather than using a complicated computer email program. This is exactly what a MUD is, minus the graphics. When virtual reality becomes more than just a pipe dream, the same principles which govern MUDs today will apply to it as well.

I'm not the only one who believes that MUDs are more than games. Right now, there are a number of other projects being worked on other than what I've done, which use MUDs as something other than a plaything. These

primarily involve conferencing tools (TMI, Xerox P ARC). This is a logical first step for MUDs since this is something they're especially good at. Their ability to allow you to convey feelings in a simple manner , in addition to text, is a very powerful tool in communicating. Until network bandwidths become high enough for us to send voice and / or video in reasonable quantities, MUDs will most likely remain the one form of network communication where feelings are conveyed directly .

The basic idea here is that what' s in a MUD interface, is where I think computers are trying to go to right now . They won' t get there for a number of years, but in the meanwhile, MUDs will be here and will keep expanding in capabilities and size and tasks. MUDs currently provide simple metaphors for accessing data and communicating electronically with users. They are easy to use and allow group productivity to be exploited. They provide an elegant way for the world community to interact and share information. They can provide a simple interface into almost any task imaginable, as MIRE does with news retrieval and user modeling. MUDs today are a simple form of the future of computing tomorrow .

Acknowledgments

Klee Dienes for some of the original inspirations for MIRE, and telling me when things sucked (and for making me print his name in huge type)

Walter Bender for supporting me and believing in me, even when I wasn't producing anything

Pascal Chesnais for being a pain in the ass, someone to talk to when life got a little stressful, and for always being there to *bite me!*

Stephen Iannce Without you, MIRE might not have happened. You helped make this project fun.

Jon Orwant for countless hours of helping me debug and work through Doppelgänger and editor, and your nigh-invulnerable wit.

Mom and Dad Even though I don't say it, I love you very much. Thank you for your long distance support of me. When I didn't believe in myself, you guys were always there doing it for me.

Laura Juliano for being a shoulder to lean on

John Garnett for being the second half of the daring driver duo!

Stephanie Tai and John Conger for being two of my best friends

Alan Blount and Nathan Abramson for all the fun times in the lab

Alex Austin for reminding me that I still have friends at home, and for writing some damn good code!

Kerry Connel and Sue Boerth for keeping in touch

Lars Pensj| for the original driver. Lars, although I spend a lot of time flaming about problems I had with the driver, remember that I still think it's the best one out there.

Adam Beeman and James Waldrop for making TMI happen

Len Kawamoto for spending lots of tedious hours converting the soul

Bibliography

[1] *The Doppelgänger User Modeling System*, Jon Orwant, orwant@media-lab.media.mit.edu. MIT Media Lab. Published in the 1991 IJCAI Workshop on Modelling for Intelligent Interaction

[2] *The Newspaper of the Future: A Straw-Man Proposal in Four Parts*. Walter Bender (walter@media-lab.media.mit.edu), Muriel Cooper, Glorianna Davenport, Ken Haase, Nicholas Negroponte. Presented at the Symposium on the Future of Newspapers at the MIT Media Laboratory July 24-25, 1991

[3] *News and Movies in the 50 Megabit Living Room*. Andrew Lippman and Walter Bender. MIT Media Laboratory

[4] *Mudding: Social Phenomenon in Text-Based Virtual Realities*. Paul Curtis, Xerox PARC, 1991

[5] *Identity Workshop: Emergent Social and Psychological Phenomenon in Text-Based Virtual Reality.* Amy Bruckman, asb@media-lab, MIT Media Laboratory

[6] *Electropolis: Communication and Community on Internet Relay Chat*, Elizabeth M. Reid, emr@munagin.ee.mu.oz.au. Honours Thesis 1991, University of Melbourne, Department of History

[7] *Burger King, Kids Club Adventures*, Volume 3, Issue 3. (Matt, age 6)

[8] *Digital Typography: An Introduction to Type and Composition for Computer System Design*, Richard Rubenstein, Digital Equipment Corporation. Addison-Wesley Publishing Company, 1988

[9] *The New Hacker's Dictionary.* Eric Raymond. Cambridge, MA: MIT Press, 1991

[10] *Bury USENET.* Steve Steinberg. Intertek, 1992.

[11] *Activity Server: A Model for Everyday Office Activities*. Sanjay Manandhar, MIT Media Laboratory, 1989.

[12] *Newspace: Mass Media and Personal Computing.* Walter Bender, Håkon Lie, Jonathon Orwant, Laura Teodosio, Nathan Abramson, Electronic Publishing Group, MIT Media Lab. 1991

APPENDIX A

Abbreviations and other information

Acronyms

MUD - Multi-User Dungeon, or Multi-User Dimension. The original.
LPMUD - Lars Pensj|'s MUD
LPC - Lars Pensj|'s C. The interpreted language used in LPMUDs
TinyMUD - First 'social' MUD.
MUCK - a variant on TinyMUD
DikuMUD - yet another MUD
MUSH - yet another MUD
MUSE - yet another MUD
MOO - MUD, Object Oriented (yet another MUD)
MUF - Multi-User Forth (yet another MUD)
TMI - The Mud Institute

OD - Overdrive

MUD names

The following MUDs are referenced frequently in this paper, and were influential in this project.

The Mud Institute
Portals
Overdrive

Character Names

MUD name	**Real Name**	**MUDS**
Wayfarer	Erik Kay	MIRE, TMI, Portals, Overdrive
Huthar	Stephan Iannce	TMI, Portals, Overdrive
Truilkan	John Garnett	TMI
Mr.X	Glenn Austin	Portals
Buddha	Adam Beeman	TMI

MUD name	Real Name	MUDS
Sirrobin	Len Kawamoto	Portals, Overdrive
Imaginos	Trevor Baker	Overdrive, Portals

These MUD usernames are used frequently throughout this paper in place of real names.

CHAPTER 2

An LPC tutorial for C programmers

Note: This appendix assumes that the reader is familiar with C. Its intent is so that the reader can understand the design issues in the mudlib section and can read and understand the mudlib code.

Second Note: This appendix refers to LPC as implemented in MudOS version 0.8.14. This is a direct modification of the LPMud 3.1.2 parser written by Lars Pensj|, and was written by Erik Kay (Wayfarer), Jeremy Radlow (Whiplash), Stephan Iannce (Huthar), etc. There are some language differences between the two, which are marked with (some appropriate symbol).

What is LPC?

LPC is Lars Pensj|' s C language. It is an interpreted, object-oriented version of C. It is interpreted in the sense that no executable need be produced, and the driver doesn' t have to be restarted every time you want to modify code. However , it isn' t linearly interpreted. It is compiled into an

internal stack machine language which is much simpler and faster to execute directly .

What makes it object-oriented?

To be truly object-oriented, a language must have at least three features: polymorphism, inheritence and encapsulation.

Every file you “compile” is its own object. It has its own variable and function name space, and you can instantiate or “clone” new instances of each object, which each have their own copies of each of the instance variables. They are distinct objects with distinct data. In addition, each object has a contructor and a destructor . In other words, a function is called when an object is created and when it is destroyed. (in LPC, these are create() and remove(), respectively [1]). There is also a way to restrict access to data within an object. Data can be defined as public or private. To get at data in another object, you generally need to go through accessor functions. Thus data can’t be modified by other code in unexpected ways. These features define encapsulation.

Each object can inherit all of their functionality from other objects. There are some easy examples of this in computer graphics. I could write a ‘point’

1. remove isn’t actually called automatically on destruction. Instead, it is called manually.

object which defines an x,y coordinate on a screen. I could then write a 'polygon' object which defines a geometric shape. This object would inherit from 'point', declaring that to be its origin, and would contain a list of connected lines which form a polygon. Further , I could define a 'square' object which would be a 'polygon' with only four sides which would only allow sides which are of equal length at right angles to each other to be added. So we have an inheritence structure which looks like this: point->polygon->square. W e could also define a triangle, and have an inheritence tree which looked like this: point->polygon->{square,triangle}. In addition, LPC has multiple inheritence, so any one object can inherit properties from any number of objects. So continuing from the above example, let' s say I make a text object, which simply contains a pointer to a string of text. Now I can make a text square object which inherits from both the text object and the square object. This object would take the text we have in the text part of the object and place it within the confines of the defined square. It shares the features and data of both objects that it inherits from.

Polymorphism is the fact that every object can have the same function names and respond differently to them. When I call init() in one object, it doesn' t have to behave the same way as another object. Each object only responds to its own functions.

What does C have that LPC doesn' t?

It's easiest just to enumerate these:

1. strong type checking --- LPC does have some strong type checking, but only at compile-time and only for functions within itself or efuns. It doesn't do proper type assignment or type checking for function calls between separate objects.
2. function checking --- LPC doesn't give an error when calling a function in another object that doesn't exist. If the other object doesn't have that function, it simply treats it as if the function returned NULL. This is because LPC is a runtime language, and it does all of its type checking at compile-time.
3. unsigned, float, double, long, char, void* data types - LPC doesn't recognize any of these datatypes. (some are replaced by others)
4. struct, typedef, enum
5. the standard C library --- LPC efuns are very limited in scope. They tend to be strict data manipulation and text display. All file access is very specialized. There is no low-level system access.
6. static local variables
7. global variables
8. any kind of system dependent features (assembly language constructs, etc)

Limitations of LPC as a language

Again, an enumeration is simplest here.

1. LPC has no representation of real or imaginary numbers
2. LPC has no direct access to C libraries. You have to modify the driver to allow access to the C library. In a lot of ways, this is a feature of the language. Specifically, it makes it less of a security risk to a system, since it doesn't give a programmer any kind of low-level access to the machine which could bypass the driver. Although it limits LPC's power and flexibility, this is usually a good thing.

3. Very little runtime checking.

What does LPC have that C doesn't?

This is a much more involved section. The major enhancements can be categorized into three major groups: runtime features, object-oriented features, string and array features, and miscellaneous additions.

Runtime Features

These features are ones gained because of LPC's interpreted and dynamically bound system. Like Objective-C, LPC is dynamically bound. This means that all objects are of one type: object. Even though they can be different types of objects entirely, they are treated the same way. So when the objects are "compiled," functions and their arguments are bound together into the stack machine code, but a function call to a remote object and the object that is its receiver aren't bound together until run-time. While this inhibits type-checking and function checking (see above), it also allows a lot more flexibility in object messaging. For example, which function is called in a receiver and even the receiver itself can be decided on the fly. This works hand in hand with polymorphism. Each object can have the same method, but you can decide at runtime which of those objects you actually want to call, and get entirely different results. So in effect, you have runtime polymorphism.

Remote object function calls look a lot like pointer references to structure elements in C: `object-><function_name>()`. You might have the following code segment:

```
void get_realname(object user)
{
  string name;
  name = (string)user->query_realname();
  /* or: name = (string)call_other(user,"query_realname"); */
  write ((string)user->query_cap_name() + "'s real name is
"+name+".\n");
}
```

This calls the function `query_realname` in the object user. This is actually just syntactic sugar for the efun `call_other` as shown in comments below it. All remote function calls are of a special type "unknown". It is illegal to assign things of type unknown to any variable or to use them in expressions, nor is it possible to declare variables to be of type unknown. So you must forcibly cast unknown expressions to the type that you are assigning them to. No actual type checking is done here as stated above, and if you cast to the wrong type, you might get some strange results.

Object Oriented Features

A number of these features have already been discussed above. However, no examples of how these features are actually used were given. Polymorphism has already been shown to some extent with the demonstration of remote function calls.

- inheritence

To cause your object to inherit from a given object, you use the `inherit` keyword: `inherit "<object-name>";`. For example, to use the text square object from up above:

```
// text_square.c
inherit "/usr/lib/square";
inherit "/usr/lib/text";
```

This causes text_square to inherit from both the text object and the square object whose source files are in the mudlib directory /usr/lib.

Another syntactical addition resulting from inheritence is being able to reference functions which are defined in a parent object. To call a function in a parent object, the form is `<parent-name>::<function-name>()`. For example, to call the function `query_text()` which is defined in the parent object `text`, you would say `text::query_text()`.

- privacy and encapsulation

Variables and functions can be declared as private or public. Any object which inherits from that object cannot get access to private variables or functions, and external objects can't call private functions in other objects. For finer control, a function can be declared as static which allows inheriting objects access, but prevents remote objects from calling it. This allows you to control (e.g. encapsulate) data, by giving selective access to it. So, for example, I could make this object:

```
// password.c
private string password;
static void set_password (string arg)
{
  if (!arg || strlen(arg) < 6)
    {
      write ("Password must be longer than 6 characters.\n");
      return;
    }
  password = arg;
}
```

This objects prevents anything other than itself from looking directly at the password variable. Inherited objects and remote objects can't get to the information. To modify it, an inheriting object must go through the function `set_password` which has some safety checks built into it. Remote objects can't call set_password since the function is declared as static.

When you inherit from an object, you can do so privately, insuring that you don't accidentally overwrite data in the parent object, or unwittingly call functions you don't want to. To do this, simply prepend 'private' before each inherit line. This is where the keyword 'public' comes into play. Public variables and functions override private inheritance. This is the only use of 'public' in the language.

Finally, functions can be declared as nomask, which prevents inheriting objects from redefining important functions. For example, if I want to make sure that nobody overrides my set_password function because I

want to guarantee that all objects which inherit from it have passwords which are longer than six characters, I simply declare the function to be:

```
nomask static void set_password (string arg);
```

Strings and Arrays (Mappings)

Strings and arrays can all be dynamically allocated and automatically freed. You don't need to statically allocate space prior to use. Strings can grow as needed. Allocating ahead of time may speed things up a little, but it is unnecessary and rarely done in LPC.

Strings and arrays can be added. For example, I can say `string a = "foo" + "bar";` or, in the case of arrays: `string *a = (({"foo","bar"})); string *b = (({"baz","quux"})); string *c = b + c;` This would make `c` equal to the union of those two arrays. Arrays also support subtraction, making a copy of the first array minus the the intersection of the first and the second.

Range is another operation which is common with strings and arrays. Specifying a subrange of an array creats a new array. For example, let's say we have an array foo which has nine elements in it. I can set bar equal to the first three elements by saying `bar = foo[0..2];` This same operation works for strings, treating each character as an element in an array. You might be tempted to say at this point "well, why haven't you just been

treating them like arrays? In C, strings are just arrays of characters, right? And LPC is supposed to be just like C, right?" Well, this is one of those exceptions. LPC doesn't have the char datatype, so you can't modify individual elements of a string. You can access individual elements (they are treated as intagers), but you can't modify them.

An addition which belongs here if nowhere else is a new data type called mappings. Mappings are associative arrays. Rather than having an ordered integer index into an array of elements, you have a non-ordered mixed type index into an array of elements. So if I wanted to keep a database of usernames and passwords, I could make a mapping like this:

```
mapping password;
password["wayfarer"] = "biteme";
password["huthar"] = "snoopy";
password["neskaya"] = "syphilis";
```

and then I could make a password checking function which would work like this:

```
int check_password (string username, string pass)
{
  if (pass == password[username])
    {
      write ("password accepted\n");
      return 1;
    }
  else
    {
      write ("password incorrect\n");
```

```
    return 0;
  }
}
```

Miscellaneous Additions

- C++ style comments

C++ style comments have been added to LPC. The start of the comment is delimited by // and then end by the end of the current line of text.

- Mixed

LPC has another datatype called mixed which is similar to ANSI C' s void * datatype. It can literally be anything, and is the only other datatype besides "unknown" which can be cast to another value.

- Shadows

Objects can be made to "shadow" other objects. This means that they are sort of a bridge through which all function calls to an object are made. This is useful in temporarily modifying the behavior of an object without having to subclass it and create a new version. It provides a sort of "runtime inheritance" feature. Functions protected by 'nomask' cannot be overshadowed.

- Varargs

Vararg support is built into LPC. This allows functions which are labeled as varargs to take any number of arguments as a legal function call, but it must actually define the maximum potential list of arguments in its prototype if it wants to actually be able to look at the data. It is not nearly as flexible as varargs in C, but it is much simpler to use.

- Archiving objects

Every object can save and restore its current state to a file. There are two efuns built into the driver which support this: restore_object() and save_object(). Both take a filename as an argument (a .o extention is appended to the end of the filename). All non-static variables in the object get saved and restored. Static variables are ignored for the purposes of archiving.

- Efuns and Simul Efuns

Efuns are hard coded functions in the driver which are essentially equivalent to the standard C library in purpose. However , most functions are at a much higher level than the C library , dealing mainly in data manipulation and talking to other objects as opposed to dealing with talking to the OS, kernel, filesystem and network.

There is a facility called simul efuns which gives the ability to write simulated efuns, which aren' t really in the driver itself, but rather exist in the mudlib itself. These can be used to modify the behavior of existing efuns in the driver as well, because simul efuns take precedence over efuns. So that

simul efuns can be bypassed on certain occasions, a special "superclass" was provided called "efun", so any object can say efun::<efun-name> and be guaranteed to call the actual driver efun rather than the simul efun version if one exists.

- Security and the Master Object

There is a central object called the "master object" whose job it is to manage security and provide the backbone for the mud. Almost all communication between the driver and the rest of the mudlib takes places in the master object. A strong effort was made to try to eliminate all mudlib dependencies in the driver other than ones made to the master object. (those were reduced some, but still exist) The master object has two primary jobs: startup and security .

The master object is the only object which is loaded when the driver is started. This means that it has almost complete control over how the mud behaves. The function epilog() in master .c is what is called on launch.

LPC provides a certain measure of file security built in. Any efuns which provide filesystem access perform security checks to see if the object calling the function is allowed to perform it as requested.

For security , the master object has two jobs. The first is to determine who is the "creator" of an object. Who is the creator of the object influences what the object has permission to do. This is decided in the second part

which is determining valid reads and writes. Specifically , whenever any efun which provides file access functionality is called, valid_read and / or valid_write are called with the creator of the calling object. From that information, the master object decides whether or not to allow that object to do the requested operation.

- Containment

LPC has the concept of object containment built in. That is, any object can store other objects within it for free. This makes things which are common in MUDs like rooms easy . In addition, the concept of a person "holding" an object can be easily represented like this. There are efuns provided which will give the contents of an object, as well as an efun to test if one object is within another or to find the object that an object is contained within. These functions really have no place in a language spec, but they make MUD coding much simpler .

- Actions and interactive objects

Objects can be interactive, meaning that they can have a user connected directly to them. Which means that anything the user types gets passed through a list of "actions" associated with that object. Actions are verbs (strings) which have a function associated with them. T o add an action to an object, you simply call the efun add_action (function, verb). Then,

whenever the user connected to the object types that verb as the first word in a line, the associated function is added. add_action is a special efun in that it relies on state of the driver to do its work. When called, it doesn't associate a verb with the object that it's in, but rather with object returned by the efun `this_player()`. `this_player()` simply returns the interactive object whose command is currently being evaluated. There is one more wrinkle in this whole arrangement. Whenever an object is moved from one object to another, the function `init()` is called within the receiving object if it exists. This is the key element in understanding how the mudlib works. For example, this is how room exits work. If a room wants to have "north" as an exit, it does an `add_action` of the verb "north" to a function which will cause the user to move to the desired destination room. Or, a broom might have the verb "sweep" associated with it, so when a user enters a room with a broom, or picks a broom up, he gets that verb, and the appropriate function is called when he types "sweep". In the case of a name conflict for verbs, the last one added takes precedence. These functions return an integer, which indicates success or failure. When a zero is returned (failure), either the next verb with the same name is called, or an error message is printed (this error message is by default "What?", but can be changed by using the efun `notify_fail()`).

Non-interactive objects can pretend to be interactive via the use of `enable_commands()`. This adds actions to them in a similar way to interac-

tive objects, and allows them to be considered “living” (there are a number of efuns which look for and try to identify living objects). This is useful for making automatons (monsters, servants, secretaries, annoying grad students, or whatever).

Other special functions similar to `this_player()` are `this_object()` which returns the current object, and previous_object() which returns the object which did a call_other into this object if that is apropriate. If not, it simply returns `this_object()`.

- Heartbeats and `call_out`

The LPmud driver has timed events, so that you can specify a function to be called after some number of seconds. This is accomplished through the `call_out`(function, time) efun. For repeating timed events, rather than forcing repeated calls to call_out, there is a facility known as heartbeats. If you turn a heartbeat on in an object, a function in that object (heartbeat) is called every ~3 seconds. This is turned on and off through the `set_heartbeat()` efun.

- `input_to` and `get_char`

These two functions are for trapping user input before it gets parsed by the driver . They force the next line (or the next character) typed by the user to be sent to the function specified. This allows objects to be interactive, so

that they can ask questions etc, and not be limited by the action / object paradigm.

Scope of message efuns and `catch_tell`

There are a suite of efuns designed to display messages to users. These all have different scopes and ranges, which are fairly simple.

TABLE 1.

`write`()	sends message to `this_player`()
`say`()	sends message to all objects that are in the same object that `this_player`() is, minus `this_player()`.
`tell_object`()	the object specified
`tell_room`()	all objects in the object specified
`shout`()	everybody connected to the mud
`message`()	all of the above plus many other combinations can be synthesized

Object in the above table means interactive and living objects. When a living, non-interactive object receives a message, it is passed to a function in the object called `catch_tell`() if it exists. This allows automatons to respond to various user actions.

CHAPTER 3

MIRE Commands

User Commands

These are the simple user commands which are usable anywhere in MIRE. They are primarily miscellaneous utilities which allow typing shortcuts, but also includes communication and help. Most commands that users use in the world are defined by individual rooms and objects.

TABLE 2.

alias	art	biff	brief	bug
describe	do	drop	follow	followers
get	give	heartbeat	help	history
idea	inventory	kill	look	mail
new_body	nickname	passwd	put	quit
remove	save	say	set	shout
tell	time	typo	unalias	unnickname
unset	watch	wear	whisper	who

Developer Commands

These commands are programming and debugging aids for MIRE programmers.

TABLE 3.

addextra	authors	call	callouts	cgive
children	clone	clr	commands	del
dest	dev	dforce	du	dump
echo	echoall	echoto	emote	farlook
finger	force	function	gauge	goto
grab	home	idle	inherit	invis
ksweep	load	log	man	mud
netdead	news	origin	path	people
pforce	remextra	run	sforce	si
snoop	su	summon	sweep	tellmsg
test	trust	version	vis	which

Developer File Commands

These UNIX-like commands provide access to the filesystem. They are intended for MIRE developers.

TABLE 4.

cat	cd	cp	ed2	grep
gupdate	ls	mkdir	more	mv
new_master	pwd	rm	rmdir	rmw

Soul Commands

These commands provide easy macros to allow users to emote. They simply echo text to users in the same room indicating that a user did what the command implies.

TABLE 5.

ack	airguitar	applaud	beam	blush
bounce	bow	burp	cackle	censor
chuckle	clap	comfort	cough	cringe
cry	cuddle	curtsey	dance	duh
excellent	fart	flip	fondle	french
frown	gasp	giggle	glare	grin
groan	grope	grovel	growl	headbang
headbutt	hiccup	highfive	hug	kick
kiss	knee	laugh	lick	love
melvin	moan	moon	mumble	mutter
nibble	nod	not	peer	point
poke	ponder	pout	puke	purr
raise	ruffle	scream	shake	shiver
shrug	sigh	sing	slam	slap
smile	smirk	snap	sneeze	snicker
sniff	snore	snuggle	spit	squee
squeeze	stare	strut	stutter	sulk
tackle	thank	think	twiddle	wave
whistle	wiggle	wink	yawn	