MUD: A Networked, Multi-User Database Service

by
Halvard K. Birkeland

Submitted to the Department of
Electrical Engineering and Computer Science

In Partial Fulfillment of the Requirements
for the Degree of

Bachelor of Science in Electrical Engineering
as the Massachusetts Institute of Technology

February 1990

Author [illegible]
Department of Electrical Engineering and Computer Science
September 22, 1989

Certified by [illegible]
Glorianna Davenport
Thesis Supervisor

Accepted by
Leonard A. Gould
Chairman, Department Committee on Undergraduate Thesis

MUD: A Networked, Multi-User Database Service

by

Halvard K. Birkeland
hkbirke@media-lab.media.mit.edu

Submitted to the Department of Electrical Engineering and Computer Science in partial fulfillment of the degree of Bachelor of Science in Electrical Engineering and Computer Science

Abstract

MUD is a multi-user, configurable, distributed, network-based database service designed for a workstation-based computing environment. The service is composed of secure primary control and data servers interfacing with various authenticated clients via TCP/IP. The service is secured by using a remote authentication protocol for both servers and clients.

This document presents a detailed specification and design for a MUD implementation. The discussed implementation uses CTREE for a local database and Kerberos for the remote authentication protocol. In addition, this document utilizes a multi-user, integrated editing system for a detailed example.

Thesis Supervisor:	Glorianna Davenport
Title:	Assistant Professor of Media Technology

Table of Contents

Acknowledgements

First and foremost, I would like to thank all of my friends, corroborators and fellow programmers for getting me through MIT because I certainly couldn't have done it without you. Thanks to Dan, Ezra, Mark, Susan, Scott, Mike, Pascal and Jenifer for I certainly wouldn't be here today without your unending support, confidence, patience and trust in me.

The Film/Video crew was great at seeing me finally write this after months of procrastination. Alan "Cool Moe Barf Jam" Lasky, Ben "RubinHarry" Rubin, Amy Bruckman, Brian Bradley, Jeff Johnson and Mike "Han Solo" Kobb were always there when needed. To fix my code, read this tome, replace the machines, graduate on time, and keep me writing once I started.

The night shift at the Media Lab deserves even more credit. Pascal Chesnais and Jenifer Tidwell finally shamed me into writing a thesis after talking about it for nearly a year. Dan "Luke Skywalker" Applebaum was endlessly patient teaching me about networks, servers and how to write (and not write) them. They sat through all of the discussions at the Haus and IHOP and never said I couldn't do it. Sylvain Morgaine, Ming Chen, Bob Sabiston, and the rest of the VLW students were always there, making me feel guilty about not writing the thesis. Without their constant prodding, I wouldn't have graduated.

Finally, I would like to thank Glorianna Davenport, my thesis advisor, for sticking by me through my indecision and procrastination until I finally decided on a thesis and wrote it. The third time was the charm.

A fond farewell and following seas to my friends at The Tech, Pershing Rifles, Project Athena, MTG, pika, NROTC, and, most of all, the Media Lab. It really was fun, no matter how much I bitched about life.

--hal
DarthVader@movies.mit.edu

Chapter 1: Introduction

MUD was originally intended as a specialized database server for VEdit, the Film/Video Integrated Video Editing System. The crucial element in an integrated editing system seems to be the database: its flexibility, speed, and utility. During the design, the project's scope gradually expanded from this small, dedicated server into a general, large-scale database service that VEdit could use.

This work is intended to present a design, specification, and sample applications of a large-scale, networked database server for a distributed computing environment. The proposed MUD unifies five significant design elements available in other network servers. First, configurable database formats and architectures permit client defined database structures. Second, distributed database services enable simultaneous access to different segments and large scale database integration. Third, centralized administration of the distributed database system aids in load sharing and error recovery. Fourth, apparent real-time response facilitates use by an interactive client. Finally, secure networked database system uses a trusted, remote authentication service.

The thesis begins in Chapter 2 by describing existing networked database systems, Moira, gdb and Sybase. In addition to the existing database servers, Kerberos, Project Athena's remote authentication service, is briefly described. The thesis then continues describing the Project Athena workstation computing environment and outlining the required hardware for a full MUD server implementation.

The heart of MUD is presented in chapters three through six, covering the design specifications, implementation, and use of the multi-user database system. Chapter 3 presents a brief overview of the system and a description of its major design requirements. A detailed discussion of a MUD design takes place in Chapter 4. The design includes the internal database layout, networking the database, database security, and client interface. Chapter 5 illustrates a MUD using a multi-user, virtual video editing system. Implementing a MUD lies at the focus of Chapter 6. This discussion ranges from coding a network layer atop an existing relational database to writing a client interface and error recovery techniques.

Finally, the limited results of the work are presented in Chapter 7. In conclusion, proposed future expansions and development of the MUD system are detailed in Chapter 8. These modifications include further improvements in server availability and error recovery along with backup facilities and a complete implementation.

Chapter 2: Background

Before embarking on MUD specification and design, let us review two existing networked database services because the proposed design attempts to combine the advantages of these available systems. In addition to the existing network services, it is useful to examine the project's available resources: both hardware and software. Let me begin with the motivation for MUD: VEdit.

A Brief Overview of an Integrated Virtual Editing System

VEdit is an integrated, multi-user video-editing system written using the X Windows System, Version 11 under Project Athena's derivative of the Berkeley UNIX™ operating system.[1] As an editing system, VEdit permits the user to hierarchically combine, edit and mix a variety of source material in an interactive environment with real-time, seamless playback [Birkeland, 1989].

The editing system is an outgrowth of the New Orleans Project: a large, multi-media case study on urban development in New Orleans. This project includes a cinematic representation of New Orleans between 1983 and 1986 as well as a variety of supporting material drawn from newspapers, laws and statistics.[2] In order for the case study to become an effective courseware document, users need to access, explore, comment and rearrange the material. The problem with making New Orleans a courseware tool was that once this case study was catalogued, organized, and assembled there were only limited tools for authoring and exploring it.

One of the most powerful tools would permit the student to integrate original portions and other student's interpretations of the source material into

[1]The X Windows System is © Massachusetts Institute of Technology, 1989. UNIX is a trademark of Bell Laboratories.

[2]"A City in Transition: New Orleans, 1983-1986" is a case study of the effects of the World's Fair, produced by Professor Glorianna Davenport. For a complete description, see [DAVENPORT 1987].

his own documents. Imagine the beginning filmmaking student sitting down at the console, calling up the sequences that everyone in the class had made for yesterday's class and being able to compare not only the final presentations but also how they arrived there. The same system would support work in a film history class: maybe the student wants to see how a real editor put together a film; so he calls up that film and gets all of the annotated information of the editor, continuity person, camera assistant, and director: what each thought about individual shots, how they assembled the film, or maybe different revisions along the way. The final environment for such an editing system would be a video version of the Electronic Classroom where students could learn how to edit video in an interactive group atmosphere.[3] In a film production course, students could use VEdit to interactively edit their own work, retain previous edits and play them back seamlessly in real time.

The editing system relies heavily upon its databases for everything from edit information to transcripts and database layout. In each of the preceding three examples, the student needs access to a subset of the available material contained in the database. A hierarchical, object-oriented database service can organize the material, presenting the student with only the material he is interested in. But in order to successfully use VEdit in a video electronic classroom or in the Project Athena environment, students must be able to access the databases independently from and simultaneously with other students.

Existing Networked Databases

Before designing a new database server, three existing networked databases were examined. Sybase, gdb, and Moira were chosen because they were available under Project Athena's version of Berkeley UNIX. c-tree and several other commercially available network database servers including Oracle, DBase and Fourth Dimension were not considered as a network database service because they did not operate in the Project Athena environment.[4] It appears that the

[3]The Electronic Classroom is a Project Athena cluster dedicated to group instruction.[Belville, 1989]

[4]While the c-tree network server will not run under Berkeley UNIX, the single-user portion of the database package can run with minor modification.

speed of the ISAM relational database outweighs the generality present in the SQL databases.

Sybase

Sybase, as distributed on the NeXT,™ is the more promising of the two SQL relational databases. It is flexible, faster than Ingres, and can easily be used with NeXTStep, the graphical programming interface for the NeXT. Sybase, however, does not support database replication, multiple database servers, or on-line backup. In addition, information is stored in host byte order, precluding sharing physical information between servers. Finally, the minimum database size is two megabytes, substantially larger that the Project Athena home directory size of 1.2 megabytes.[Sybase, 1989] It is fine for a limited availability environment, but falls short when considered for a near one-hundred percent availability requirement as in MUD.

RTI Ingres

RTI Ingres is a common relational database package available for the Berkeley UNIX operating system. Ingres, while sufficient for some applications, poses significant problems for a highly interactive application like VEdit. Performance testing has been conducted using two different network interfaces which resulted in the same conclusion: Ingres is too slow. The other major problem is that Ingres runs on a limited hardware and operating system configurations.

Moira

Moira is Project Athena's dedicated network database service. It is consists of databases containing records of all registered users, filesystems, and much more.[5] Moira, also called Service Management System (SMS), is a network interface built on top of a relational database. Currently, SMS uses Ingres as the database application. While Moira may perform satisfactorily in its current role, it is not usable for a highly interactive, multi-user application like VEdit.

[5]The Moira Technical Plan is presented in [Levine, 1989].

In the past, performance testing has been done on both Ingres and SMS. With the database running on an unloaded DEC MicroVaxII with nine megabytes of random access memory (RAM), Ingres was shown to take one second to update a single field in a small, specified record in a large database. In addition, SMS imposed a minimal overhead for the query (less than five hundredths of a second).[Kohl, 1988] As Kohl concluded, the SMS protocol overhead is not excessive but the Ingres response time is too slow for an interactive system. Perhaps if a faster relational database were used, Moira would be more usable for an interactive environment.

gdb

gdb is another network interface built on top of a relational database which can best be described as a network abstraction for a local database. While one could theoretically use any available relational database, gdb has only been implemented for RTI Ingres version 3.[Mendelsohn 1986a] Unfortunately, it suffers from problems similar to Moira. The Film/Video Section of the MIT Media Laboratory has experimented using gdb and Ingres for movie database storage [Beauchamp, 1987] and found gdb inadequate. Simple queries into a relatively small database could take up to two and one-half minutes.[Davenport, 1989] It is logical to attribute the poor performance to Ingres because it was shown before to be exceedingly slow.

Availability

While RTI supports many hardware configurations, Ingres is available at Project Athena only on the Digital Equipment Corporation's VAX-11 architecture. More importantly, RTI has claimed that support for the Berkeley version of the UNIX operating system will be discontinued in the near future. In fact, Project Athena is currently searching for a replacement for Ingres as the supported relational database. It makes little sense to develop a new networked database service using a slow relational database which may shortly be replaced.

c-tree

An alternative to the SQL based databases presented above seems to be an ISAM database like c-tree. c-tree is a relational database package originally developed for the IBM PC which has been ported to run on a large variety of platforms: everything from DEC VAX-11 and MicroVax through IBM PC and PC/RT to Apple Macintosh.[FairCom, 1988] c-tree's major advantages are speed and portability. While the database may lack some of the generality present in an SQL database like Sybase and Ingres, it is extremely fast.

Portability is important in a workstation environment because there often is a mixture of vendors and products from one cluster to another. C-tree will not only run on many platforms but also can share databases between platforms. The sharing is difficult due to varying internal hardware differences between different workstations.

What c-tree loses in generality to an SQL database, it makes up for with raw speed. As shown below, it will perform simple queries, modifications, and additions many times faster than Ingres. The speed difference is exacerbated when c-tree is run on a faster machine. For example, a DECStation 3100 with only eight megabytes of core memory performs three to five times faster than the DEC MicroVAXII.

Test Type	**Ingres**	**c-tree**	
(time in secs)	**uVaxII**	**uVaxII**	**PMAX**
Add 100	100	6.2	3.2
Add 1000	1000	66	36
Change 1000	1050	23	4

Figure 1: Relative speed of Ingres and c-tree on various hardware platforms

Moira and gdb confirmed that the networked database service is rapid enough for an interactive application but they utilized relational databases which were too slow. As shown above, c-tree seems to be much faster than Ingres. Because c-tree is not an SQL database, a different server must be written and in the process it might as well combine the advantages present in Sybase and Moira with c-tree's speed.

Available Resources

MUD is designed for a workstation-based computing environment like Project Athena's. It follows the client-server model prevalent in remote services, like Galatea and Kerberos, present at Project Athena.

Workstation

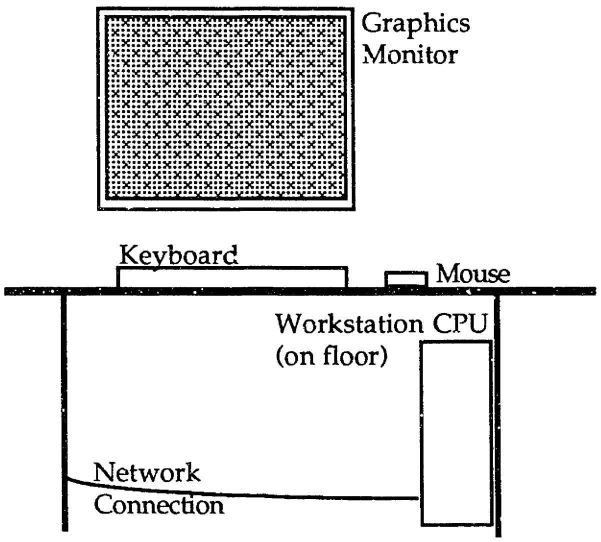


Figure 2: Standard Project Athena Workstation

A workstation is a publicly available, single-user computer designed to operate in a high-speed networked environment. While the workstation is a private computing resource, it relies on remote machines for file and many other remote services. At Project Athena, this machine is typically a monochrome DEC MicroVax II or IBM RT/PC running a local version of 4.3BSD UNIX (the Berkeley derivation of AT&T's UNIX operating system).

Networked Computing Environment

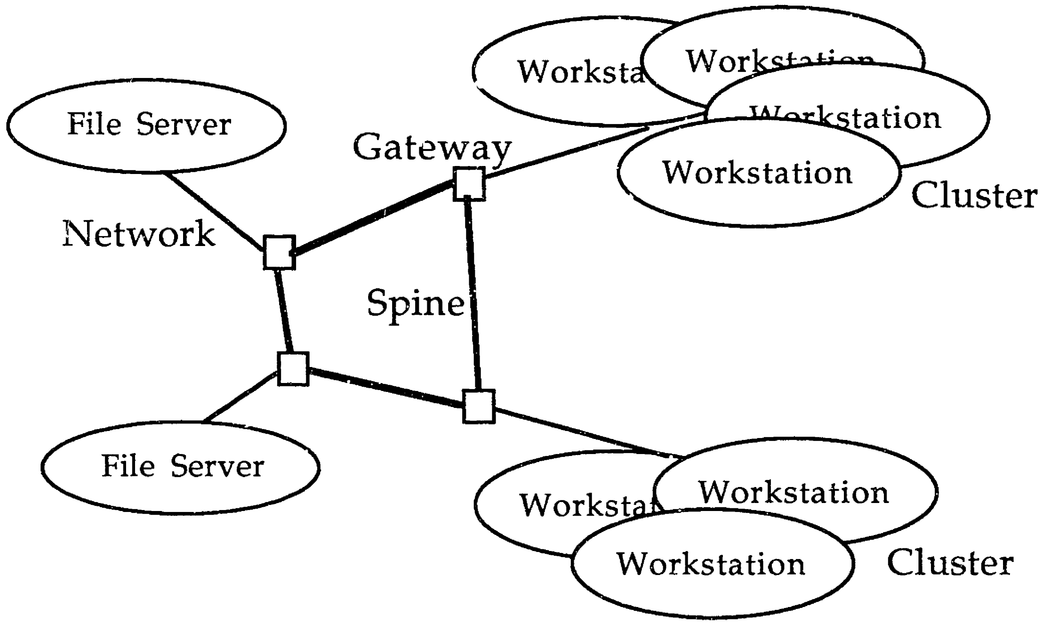


Figure 3: Networked Environment

In the past, large-scale distributed computing environments have been composed of several, isolated mainframe computers with many connecting terminals in each location. Recently, the small number of mainframes has been replaced by a large number of individual, networked workstations. Going along with the workstations have been a variety of distributed, network-wide services including the file, name, and authentication services shown above.

Galatea: The Network Video Device Server

One of these distributed network services is Galatea, a network service designed to access remote video devices.[Applebaum, 1989] Galatea provides a sample, distributable client-server implementation using both TCP/IP and UDP connections. This multi-user networked video device service could be used as a basis to develop a secure networked database service.

Kerberos: A Remote Authentication Service

Another important network service provided by Project Athena is a remote authentication service called Kerberos. Kerberos is a trusted, third-party service which provides private key encryption which can authenticate a remote client to a server. [Steiner, 1988b] For example, Kerberos can be used to authenticate a user (client) on an insecure workstation to his file server (remote service) by providing a unique number (key) known only to the user and the server.

Chapter 3: MUD Requirements

This chapter presents a brief description of the requirements for a multi-user database server. It begins with an overview of a distributed computing environment. Then, the specification is presented for the secure, multi-user, definable database architecture.

Operating Environment

The MUD is designed for a distributed network environment like MIT's Project Athena. The classical computing environment is a few large mainframe computers each with many text terminals for users. Instead, Project Athena has more than five hundred workstations located throughout the MIT campus. Users no longer use a centralized machine but instead communicate with many other machines using a variety of servers. MUD is designed to match this environment by using several separate machines to provide the networked database service.

Database Functionality

Perhaps the most important question at this point is what does MUD have to offer that other, existing database servers do not. Its functionality can be defined in two major areas: distributed environment and flexible structure. The distributed environment permits multiple simultaneous connections and distributed processing and storage in a secure environment. While the environment encourages large scale database development, it is the flexible, definable database format that forms the heart of MUD's functionality.

Multi-User Capability

The single most important requirement of a multi-user database is for it to be truly multi-user, meaning that many users can access the database simultaneously. While true simultaneous access is not possible in all circumstances without creating a read-write race conditions, database access shall be as simultaneous as possible. This simultaneatity becomes important during complicated multi-stage queries because other clients should not be blocked from accessing the database for the entire period.

Multi-Server Capability

Eventually, as the size of the client population and data size grows, more than one server will be needed to maintain a reasonable level of performance. Having more than one server also improves simultaneatity through parallelism: while one server is conducting a search across a segment of the MUD, the other servers are free to access their respective segments. Additional servers increase the reliability and availability of the MUD system.

Security

Security, in light of recent computer-based attacks on network services, may be the most important aspect of any networked database server. The importance of security is only magnified by Project Athena's public, workstation based environment. Fortunately, Project Athena provides Kerberos, a remote, secure, authentication service, for user validation [Steiner, 1988a].

To ensure a secure operating environment, MUD requires that not only all clients authenticate themselves with the server but that each individual server authenticate with the master server. The mutual authentications are designed to ensure that neither a fake client nor database server can be brought into the MUD.

Definable Formats

At the database level, definable formats are a significant advantage over a dedicated network database like Moira. By allowing the client application to specify the architecture and contents of a database allows the number and classes of data objects to expand without modifying the database server. Definable formats allow the best of both worlds: server conducted searches on a client defined database.

In a traditional, dedicated database server, the application has two options: use an existing record type and conduct searches on the server side or use a new type and conduct searches on the client side. The first choice allows a new data type only through a server modification. The second choice enacts a severe speed penalty as the entire search set must be transmitted to the client and not just the match set.

Apparent Real-Time Operation

The initial target application is the interactive video editing system. In order for such highly interactive software to be usable, the access delay when communicating with the database must be minimized. Ideally, the database query and retrieval would happen in less than one second, but this delay could be several seconds without greatly affecting the client. The current delay of more than two minutes using Ingres and gdb is wholly unacceptable.

Distributed Storage

MUD's storage architecture should match its operating environment. Existing databases were designed for timesharing hosts, large mini or mainframe computers utilized by many users, as opposed to a single-user workstation. They stored the data in a single centralized location because file storage was only local to the timesharing machine. Since the workstation environment consists of many remote storage locations (file servers) and remote file systems, it only makes sense to use them.

MUD should place the databases in an appropriate remote location (file locker) for three major reasons. First, the workstation has very limited local storage which is routinely erased. Second, a user will rarely use the same workstation on a routine basis because the workstation concept assumes that all stations are identical and that a user will choose whichever is available. Third, splitting the databases across filesystems allows each person's or group's data to effect their own quota. Users will be in control of how much data they can store locally and will not have to worry about another user's usage. Placing files in a consistent remote location, home directories for user databases and course file systems for group projects, will permit the user to access the databases from any Project Athena workstation.

Chapter 4: Designing a MUD

Presented in this chapter is one design for a MUD. It is not intended to be the sole design method but simply one possible design. The design begins with a database level discussion about internal database organization: the object and the template. The objects are combined to form the database segment. Then these segments are assembled to form the database hierarchy. The dual server arrangement, primary and database, is introduced to control the segmented database. Finally, an authentication protocol is explained to ensure mutual authentication of clients and servers.

The Database Object

The object forms the atomic unit for a MUD database. The complexity of these objects varies widely from a few simple to multiple variable length fields. Let us examine two sample objects from VEdit that illustrate the flexibility present in MUD databases.

The Source A Simple, Fixed-Length Object

Perhaps the simplest object class currently implemented is the "Source" object which provides background information about a particular piece of source material. As shown in Figure 4, the source is a fixed length, 128 byte record. The object defines not only the source material's name, type, and description but also assigns it an icon and unique id for future references.

Source:

key:	unsigned long
Name:	char31
Icon:	Tuple
Source Type:	unsigned long
Description:	FileReference

Figure 4: The Source Object

The Edit: A Variable Length Object

The edit object is an excellent example of a complicated object containing variable length elements. As shown below, the edit object has three major parts: display information, ancestral information, and the EDL representation.

Edit:

key:	unsigned long
Shot:	Tuple
Parent:	Tuple
Mode:	StateArray
NumberClips:	unsigned long
<Clips>:	Clip
Number Switches:	unsigned long
<Switches>:	Switch

Figure 5: The Edit Object

The first segment, the display representation, describes how the shot appears to the user. In addition to an array of flags which define the edit's on-screen appearance, it also includes a pointer to the edit's icon (a portion of a video frame). The ancestral information is a pointer to another database object. It contains information entered when the movie was logged including continuity information, notes, comments, transcript, original edit, and a technical description of the camera shot. The final segment contains the variable length representation of the EDL. In brief, the EDL consists of two major parts: "clips" and "switches."[6] The segment begins with the number of each type of event followed by a sequential listing of all of them. Typically, a straightforward shot will be one clip and two switches. A complicated sequence, however, might contain twenty clips and over sixty switches.

Object Template

The object template tells the database what fields, and their respective types, are present in a given object class. This content and formatting information consists of a series of tuples, one for each field in the object. A tuple

[6]The VEdit representation for an EDL is described in Appendix B.

is comprised of three fields: the name, type, and formatting information. The name is a NULL-terminated string used as the field's title. The type is a four byte signed integer used to denote what type of data the field is associated with.[7] The format field consists of 32 single-bit flags to convey information about not only the appearance of the field but also whether the field is editable and what classes of characters are permitted for input.[8] For example, the tuple for a person's surname will contain "Surname:" in the name field, "Char32" in the type field, and "Editable | ASCII | Right Justified" in the format field.

Template:

key:	unsigned long
Name:	char31
Icon:	Tuple
Object Class:	unsigned long
Number Fields:	unsigned long
<Fields>:	Field

Figure 6: Object Template

The Individual Database

An individual database is divided into three major portions. First, the database information segment which contains the generic information about this particular database including name, ownership, protection, and location. Second, the template segment contains not only the templates used for each object class but also location and ancestral information for that particular object class. Finally, the remaining segment, the data segment, contains the actual information contained in the database.

[7]A complete list of field types is available in Appendix B.

[8]The complete list of formatting and content information is also available in Appendix B.

Database:

key:	unsigned long
Name:	char127
Parent:	Tuple
Icon:	Tuple
State:	StateArray
numGroups:	unsigned long
<Groups>:	Tuple
<Access>:	StateArray
numLocations:	unsigned long
<Location>:	File Reference

Figure 7: Database Object

Database Information Segment

The first portion of the database record is the information segment containing the name, ownership, protection, and physical location of the database. Each access list entry is a tuple in the group list and a corresponding state array containing the actual protection. This arbitrarily long list permits flexible, configurable access similaı to the Andrew File System.[TransArc, 1989] The remaining list, the locations, give the real locations for the database as a series of tuples. The first tuple in this list is the synchronizing site, other locations must forward the write requests to this site.

Template

A template contains the formatting information for a given object class. It consists of a series of field tuples preceded by the number of fields in the object class. A field tuple consists of the field type, name, and flags. The field type describes how the data in the field is formatted and is chosen from signed numeric, unsigned numeric, character string, numeric string, and object pointer. The field name is a fifty-four character string containing the field title. The final element of the tuple is the flag information, an array of flags controlling field display and modification. The controls include whether the entry is displayed, the value is editable, the hierarchy is expanded, and whether a pointer should be followed by default.

Template:

key:	unsigned long
Name:	char31
Icon:	Tuple
Object Class:	unsigned long
Number Fields:	unsigned long
<Fields>:	Field

Figure 8: Sample Template Structure

Data

The final element of the database is the information itself. Because the individual object classes may be widely variant, the object classes are isolated from one another. This segregation results in a series of record lists with each list corresponding to the entries for a particular object class.

Database Structure

MUD allows this information to be split amongst several databases. These elements of the overall database, called segments, fall into two classes: static and dynamic. In order to use these segments, MUD combines them into a single homogeneous environment.

Static and Dynamic Databases

Databases fall into two general categories: static and dynamic. A dynamic database is designed to be continually modified. All of the records within the database are volatile. The static database, or master database, changes very slowly with time. While it is not impossible to change the data in a static database, these changes should be limited to minor corrections.

More importantly, important content related information should not be changed in a static database because references into a static database are links. This means that the information is not actually stored with the reference but is looked up in the static database. For example, entire records should not be removed from a static database without first verifying that nothing has been linked to it. On the other hand, changing a typographical or data entry error automatically changes it in all referring records.

Combining Databases

The important consideration when combining several databases is that local information replaces parental information in a controlled manner. The user must be able to selectively view not only the local information but all additional information contained in parent records. On the other hand, the user should not rely on information contained in dynamic databases because it may be deleted or modified at any time. The solution seems to be varying the inheritance path between a parent database types.

The inheritance rules are different between a static and a dynamic database. If the user incorporates a record from a dynamic database into his personal environment, then the entire record is copied over. Therefore, all of the information is locally available to the user. If the parent record is changed, the local record still contains all of the original information. If the record had been copied from a static database, then only the link would have been copied. In this case, a change in the parent record will be propagated to the local record.

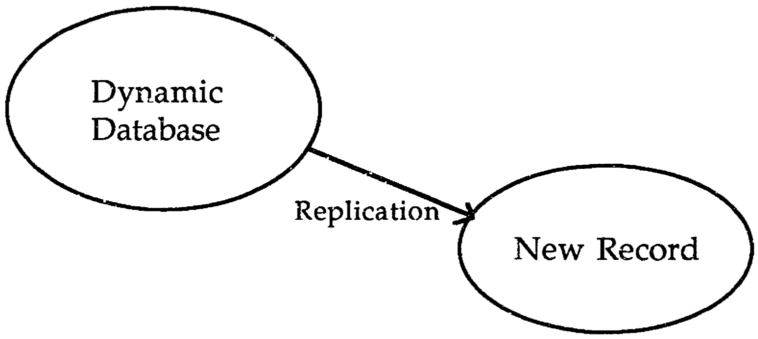


Figure 9: Inheritance Rules for Dynamic Databases

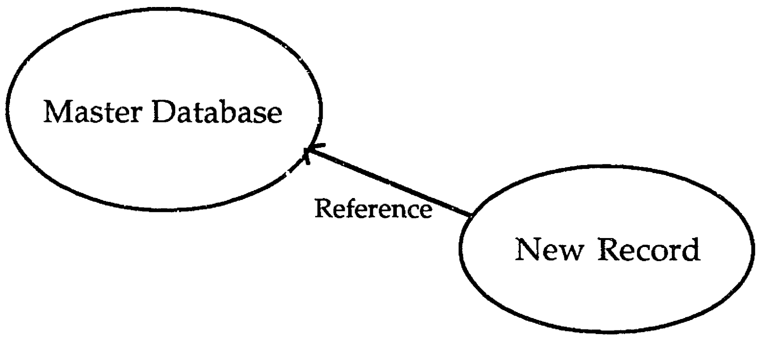


Figure 10: Inheritance Rules for Master Databases

Networked Database Design

The MUD incorporates two separate servers for the network interface. The primary server oversees the database as a whole, maintaining state and status information while the database server performs the actual data retrieval, sorting, and storage operations. The dual server arrangement increases the reliability, flexibility, and scale of a MUD implementation.

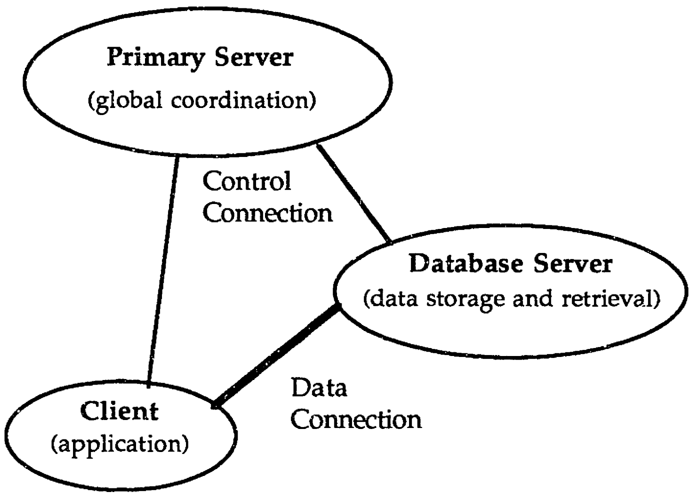


Figure 11: MUD Layout

Primary Server

The primary server oversees the MUD operation. Its role is composed of four distinct parts: maintaining state, controlling database servers, authenticating clients and servers, and aiding in error recovery. While none of these functions are essential to an initial implementation, each task becomes important when building a large scale, networked database.

State Maintenance

The first task for the primary server is maintaining the MUD state. MUD's internal state is divided into three major categories: client and database server connections, database layout, and statistics. Client and remote server connections describe the status of all of the remote clients and servers using the MUD. The database layout describes not only the MUD database environment but also which database server is controlling each segment of the MUD. The final element of internal state is gathering statistics on various operations, load factors, response times, and server availability. While the statistics are not directly usable by the MUD, they can be invaluable in indicating which portion of the server to optimize and what typical usages of the MUD are.

Security

The primary server's next task is to authenticate all of the connections, both remote database servers and new clients. As described below in protection, verifying all remote connections is crucial in a networked environment. A secondary facet of authentication is providing a secure acknowledgement to a remote server to attest to the authenticity of a newly requested connection. Basically, the primary server distributes the initial client or server authentication to any necessary remote locations.

Database Structure

The primary server's third role is to distribute the MUD activities amongst the available database servers depending on load and resource availability. If the database is split amongst several database servers, the primary server needs to coordinate the overall organization of the database, assign backup database segments. An outgrowth of the assignment is reassigning database segments when either the segment or the database becomes unavailable.

Combining and Maintaining Logs

Logs have two general purposes: statistics and recovery. The primary servers have two separate roles in logging. First, they combine the individual database server logs into a single, system-wide log. Second, the primary server maintains its own logs regarding database servers, client connections, database distribution, and overall activity.

The logs serve three major roles: audit trace, statistics, and recovery. The logs could be compiled, sorted and filtered using a database administration client. In addition, the logs can be used for a primitive audit trace to determine which portion of the database have been entered, modified or deleted. The final, and most important, role of the log files is to permit a database to recover from a catastrophic crash.

Error Recovery

The remaining task for the primary server is to aid in error recovery. The major reason for the dual server arrangement is for error recovery. By including a supervising server in the database architecture, the supervisor can reallocate the resources among all of the workers (the database servers) without the clients noticing. While there may be a decrease in performance during the reorganization, the clients should not experience total failure. This error recovery permits almost one hundred percent database availability except when the primary server becomes unavailable for extended periods.

Database Servers

Perhaps the most important quality for a database in a networked, multi-user environment is network support. Network database server support seems to exist in two flavors: specialized and raw. A raw network interface simulates a low-level local interface for the application. The specialized network interface behaves like a UNIX server allowing the application to make high-level requests and returning the results. While the raw database server seems to be the preferred option, the faster specialized server can be used by encoding additional information within the database.

Raw Database Server

The raw network interface possesses several important advantages and one very significant disadvantage. The raw interface is much more flexible in the general case because the application directly controls the database. The interface simply pipes the application's requests to the database and returns the results. However, the raw interface is much slower because it transmits the intermediate results back to the application. The performance for the raw interface may be orders of magnitude slower than for a specialized interface when performing large complex searches across multiple databases because the results for each part are transmitted to the client application for comparison.

Specialized Database Server

The specialized network interface has several disadvantages but two very important advantages for the multi-user environment. The specialized server is inherently less flexible than the raw server. In addition, the database implementation is split between the client and the server because the relational database functionality is now in the server and not the client. Placing the relational database manipulation in the server requires server development to continue throughout the project because as new functionality is needed it must be built into the server as opposed to the client under a raw database server arrangement. Building the database functionality into the server results in the major advantage for the specialized server — speed. Since the server is only returning the matches to complex searches, much less data is being transmitted across the network and the apparent search speed is dramatically increased.

The specialized server can be made more flexible by permitting client definable database structure. In other words, the database format is not hard coded into the server but is determined at run-time. The database formatting information, templates, should be able to submitted by the client or some existing segment of the database. By downloading the template into the database server, queries still be performed server-side for a client defined database maintaining the major advantage of the specialized server.

Maintaining Logs

A database server must also maintain its own versions of journal files. Here, the logs contain command histories, client connections, and database modifications. The purpose of these logs is very similar to the primary server: statistics and error recovery. The log files are routinely used by the database servers for recovering flushed queries and restoring a replicated database to the current state.

MUD Redundancy

One of the more difficult issues to address in limited time and coding effort lies in database maintenance during system errors. MUD improves the reliability of a network database service in three areas: replicated databases, multiple database servers, and multiple primary servers. Database replication results in multiple physical copies of the information. Multiple database servers permit several hosts to provide the service for a given database. Finally, multiple primary servers permit the MUD to survive a primary server crash.

Replicated Databases

Maintaining multiple copies of the data have two major advantages. First, it allows the database to survive a file server going off-line. Second, it provides instant and zero-loss recovery from a catastrophic file system error. The penalty for this redundancy is additional overhead in maintaining the database during operation and doubled file storage requirements. Depending upon the role of MUD, the database redundancy may be or may not be worth implementing.

The dual server arrangement permits database replication without a complicated overhead. A replicated database is one which is maintained by several, separate database servers. While any of the replications may be used for retrieval operations, only the primary copy may be used for storage. The client may be connected to any of the servers, but all write requests will be transparently forwarded to the primary server. The additional overhead, present on the write request, is a retransmission to the primary server. The database server for the read/write copy, however, will then have to update each of the remote copies after control is returned to the client. Depending on implementation specifics and database structure, the delay from the write operation until full propagation should be between five seconds and one minute.

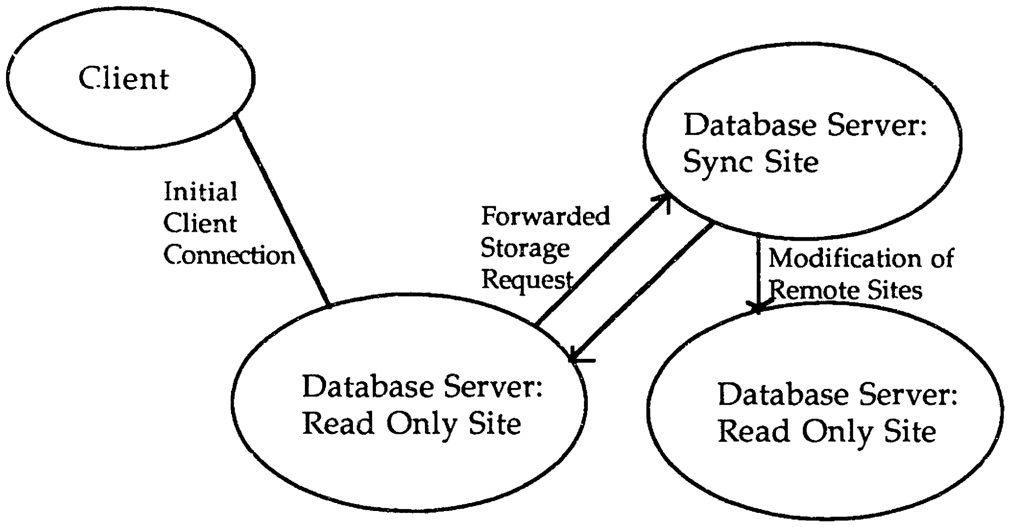

Client
Initial Client Connection
Forwarded Storage Request
Database Server: Sync Site
Modification of Remote Sites
Database Server: Read Only Site
Database Server: Read Only Site

Figure 12: Replicated MUD Database

Multiple Database Servers

From a redundancy standpoint, having backup servers allows the entire database to remain available in the event of a server going off-line. The segments of the MUD database served by an individual server are controlled by the primary servers. For the proposed example, VEdit, continuous availability is not very important, but if the database was being used for user and system level services one hundred percent availability is extremely important.

Multiple Primary Servers

If there were more than one primary server arranged in a primary-secondary arrangement, when one goes off-line the other can keep the MUD running. Unfortunately, this adds yet another communication requirement (primary to secondary) in an already complicated system because the primary servers must share the database control information. A simple solution to sharing information is to use the MUD itself: storing the database information inside the MUD database permits it to be replicated across several sites. In addition to improved reliability, the additional primary servers can improve the overall performance by sharing the centralized operations. If MUD service is supposed to be uninterruptable, however, then the backup primary server is essential.

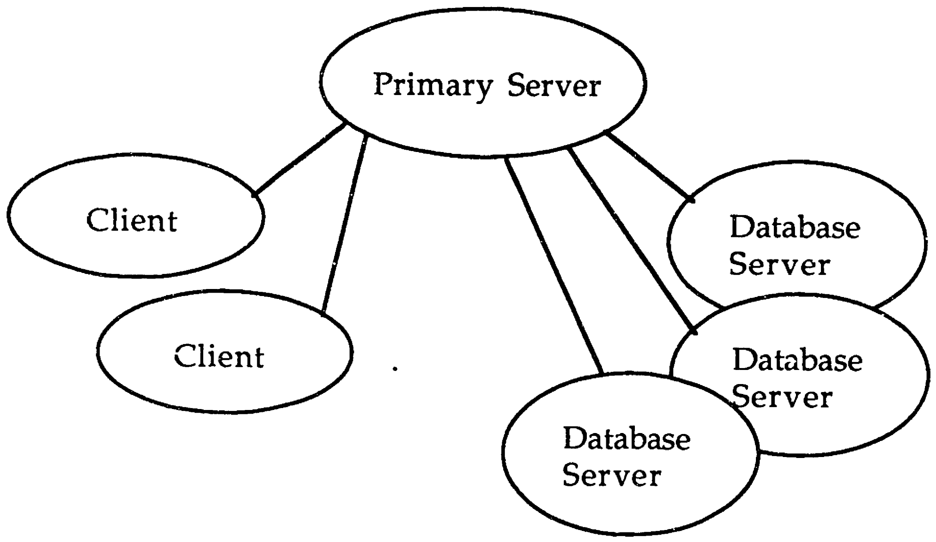


Figure 13: MUD Service with a Single Primary Server

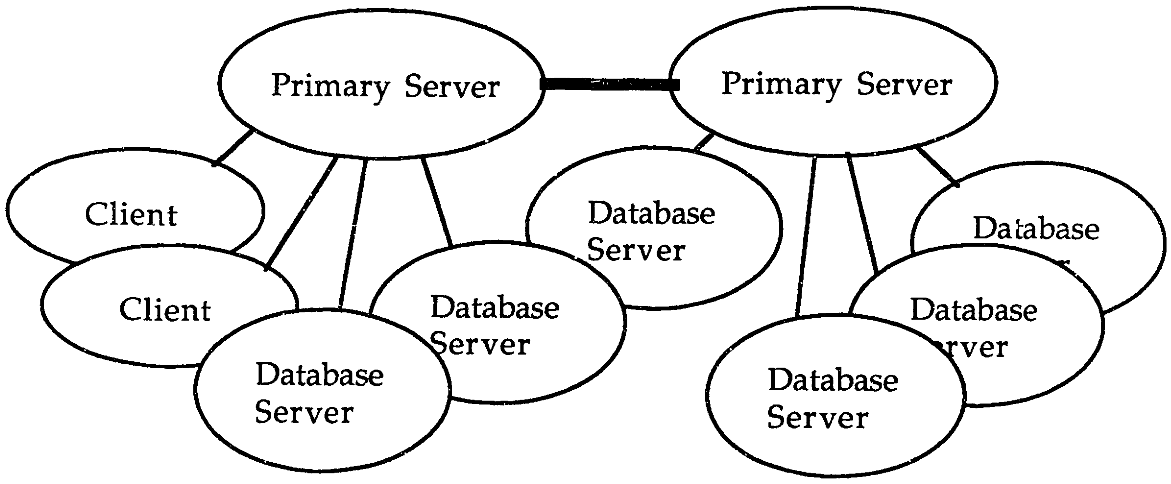


Figure 14: MUD Service with Multiple Primary Servers

Automatic Backups

The final redundancy issue is automatic backups. Given the volatile nature of portions of the database, it may be desirable to implement a MUD backup utility which would save the contents of a database every night. The MUD can support this by making a temporary read-only replication of the database and then dumping that to tape. While the author has no intention of describing how to design or implement such a facility, it is important to realize that internal backups may be a very desirable feature.

Overcoming Performance Penalties

Unfortunately, the dual server arrangement has some associated performance penalties. These performance hits take the form of additional overhead required for a given transaction because the database request must be gatewayed through the primary server to the appropriate database server. If the returned information also passed through the primary server, the database speed would be one-half of the single server arrangement. Because a significant portion of the request time lies in the network transmission, minimizing transmissions is very important.

The proposed solution is to open a direct connection between the client and the database server to accommodate both control and data transfer. The initial location of the remote database will be established by the primary server, but then all access will be performed directly with the database server. While this technique will minimize the differential performance between a single-layer and dual-layer server structure, it still yields a vast performance penalty when compared to a local database.

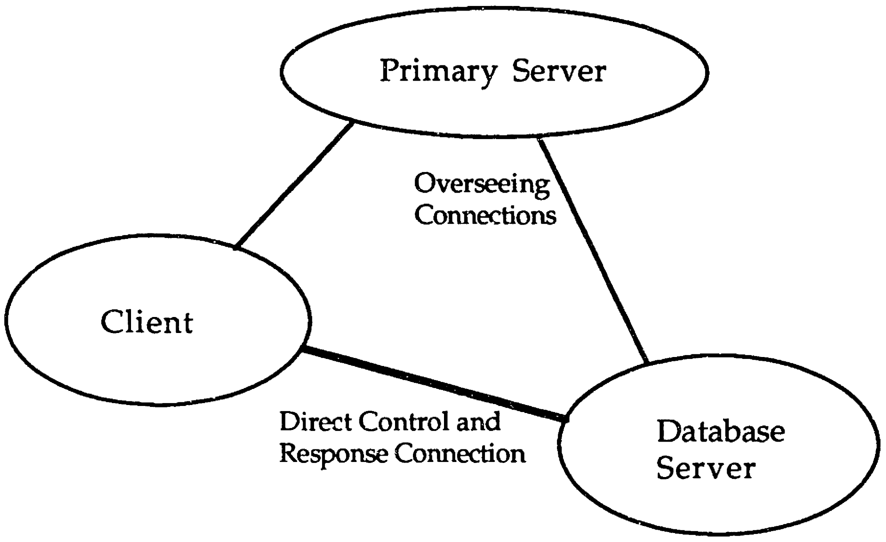


Figure 15: Data Transmission in a MUD

Protection

Security and protection become important in a multi-user environment where one user may want to restrict the availability of his information. Security is a particularly difficult concept within Project Athena's workstation-based computing environment. VEdit databases support three layers of protection: file system, file, and internal.

File System Protection

A file system forms the first layer of protection. It is possible to create file systems which can only be "attached" or accessed by certain groups of users.[9]

[9]Currently, Project Athena has two classes of remote file service: AFS and NFS. NFS, the Network File System, is the more widely used and highly distributed service. The NFS servers are reasonably secure because they each have a local credentials files to authenticate the kerberos tickets presented by the remote user. Unfortunately, the group concept is not conveniently supported by NFS. In addition, access to the server's files are on a per-filesystem basis and not per-locker (a single file system may contain several lockers). AFS, the Andrew File System, is a more recent addition. It has a highly centralized architecture allowing a few system administrators to control

Placing data in such a "fascist" locker will restrict that data to the certain specific list of users permitted to attach that file system. Unfortunately, the list of users is on a per-machine basis (not per-file system). This means that access can be restricted to specific groups of file systems but cannot be restricted to specific file systems within that group. While this solution is effective, it is also less than optimal if the database administrator does not administrator the file services. If the administrative control is separated, updating the lists of users is quite difficult.

File Protection

Secondary protection is on a per-file basis. The database server matches the user's identification and group access against the UNIX file system protection modes. This is the most effective manner to create read-only databases, or databases that can be modified by a certain select group. But beyond global access restrictions, this protection layer is strongly limited. Per-file control, however, is very effective to restrict access to detailed information contained in flat files because the database can be world readable but the flat files may be restricted to a certain group of users.[10]

Internal Protection

The primary layer of protection resides inside the sub-database itself and closely mimics the UNIX file protection scheme. The internal protection has the advantage in that it uses VEdit's own, internal access control lists (ACL) as opposed to the external, Athena-wide lists.[Rosenstein, 1989] The internal lists can be modified from within the editing system using the administration tools. Any changes to internal ACLs take effect immediately as opposed to overnight with the Athena-wide version. Furthermore, membership in a given ACL can be

the entire system. AFS has improved access control over NFS because it is on a per-directory basis and also allows greater group control. Furthermore, the AFS protections can be easily changed by any user who has administration access for that directory.

[10]One possible use of this disctinction would be for a "bugs" database. Here, the general user could retrieve all of the index information and topic heading from the database but only those users in specific groups could access the detailed descriptions and fixes.

given on a per-database level because the ACLs are associated with a given database as opposed to all of the databases. The internal protection is the more flexible, easier to use, and less prone to error than the other two forms.

Chapter 5: Using a MUD: VEdit

The multi-user database service was originally developed specifically for the Film/Video video editing system, VEdit. MUD has grown throughout the project and now represents a general network database service. It has several key features: error recovery, client specified database configuration, client security, and centralized maintenance. Let us explore MUD's functionality using VEdit as a sample client.

Database Structure

The VEdit environment is broken into three major sections: movie, project, and personal. The movie contains information which pertains directly to the source material: logging, continuity information, transcripts, and description of both the audio and video. The project may combine information from several movie and project databases into a homogeneous environment. The final element in the database hierarchy is the personal database which is intended for user-specific information and, in addition, is the only truly dynamic database.

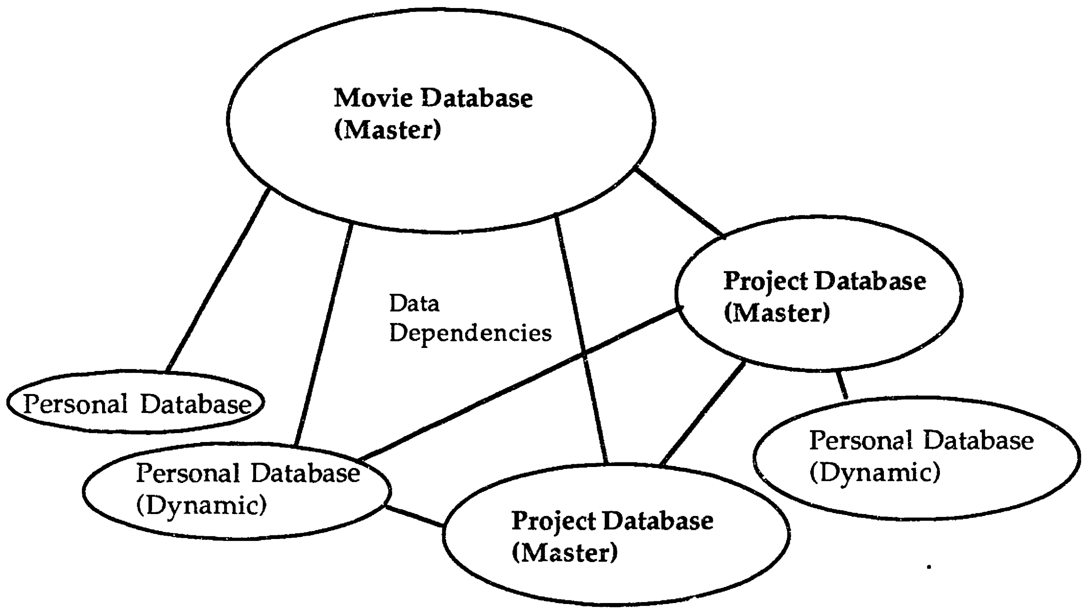

Movie Database (Master)
Data Dependencies
Project Database (Master)
Personal Database
Personal Database (Dynamic)
Project Database (Master)
Personal Database (Dynamic)

Figure 16: VEdit Usage of the MUD Database Service

Movie Databases

Within the editing environment are several movie databases. A movie database is a complete collection of information about a given piece of source material and may encompass several separate projects, interests, topics, and users. It includes transcripts, logging information, character backgrounds, annotations, comments, and anything else the filmmaker and other users decided was important.

Specifically, the movie database appears to contain all of the movie-specific information. The information ranges from descriptive information (scene lists, character backgrounds, dates, and locations among others) through edit list management information (shot breakdown of the source material and transcripts) and linked information (information linked to someplace else). While the internal arrangement of this information will vary, it will always appear as a single-level relational database While the user may recognize the four paragraph description of the JAX brewery as descriptive, the database does not treat it specially.

Project Databases

Project databases should be able not only to sub-divide a movie database but also to incorporate several separate movies together if desired. The separation between the movie and project databases is intended to isolate the production information contained in the movie database from the application and editing present in the project databases. In an academic environment, the same footage may be used by different groups of people with different goals in mind.

Let us pursue a film course project comparing the editing style used in two different films. This particular project was chosen for the example because it incorporates both functions of this database class: division and combination. The combination lies in the integration of two separate movie databases (let us take "Die Hard" and "Blade Runner" for an example). Division comes into play because the student studying editing technique will not usually be interested in character histories and someone else's editing but will be interested in the EDL information, logging, annotation and outtakes.

The project layer of the environment is intended to contain all of these project-oriented customizations while the movie layer contains the raw information about the source material. The project layer becomes more useful by permitting the project to contain information from several movies or project databases. The data combination allows critical comparison between two separate movies or projects in a homogeneous environment.

Personal Databases

The final database class used for VEdit is the personal database. These databases are intended for rapidly changing and narrowly targeted information such as individual edit decision lists (EDL), annotations, and additions and, as such, are typically isolated on a per-user basis. The degree of detail in the audit trail is set according to the user's needs: an editor may want a full audit trace while a user who is exploring a database may need only minimal tracing.

Editing Environment

VEdit's environment allows the user to do everything from exploring to editing to creating new information, references, and edits. This flexibility and functionality, while present in the user interface, is a direct outgrowth of the flexibility in the underlying database service, MUD. The database is used to store information ranging from the current editor's editing environment (a personal cutting room) to camera and continuity information from the various source material.

Exploring the Databases

The first step to using a hierarchical database architecture is being able to explore the hierarchy. This database architecture can be illustrated through two different representations, the hierarchy and the list. Both representations include whether the database is writable by the user and if defined video resources are available. Neither representation is better in all situations depending on the user's goals and the database architecture's complexity and size. In a large, disjoint environment, the hierarchical display may be more efficient while in an integrated environment the single level approach may be superior.

The list display shows the databases as a single-level format, not differentiating between a movie and a personal database. The two major advantages of the list format are that all of the databases are listed in alphabetical order and that each database is listed only once. The format permits the user to quickly enter a specific database or to find a database without traversing the hierarchical structure. For example, opening a personal database does not require opening first a movie database and possibly several group databases beforehand.

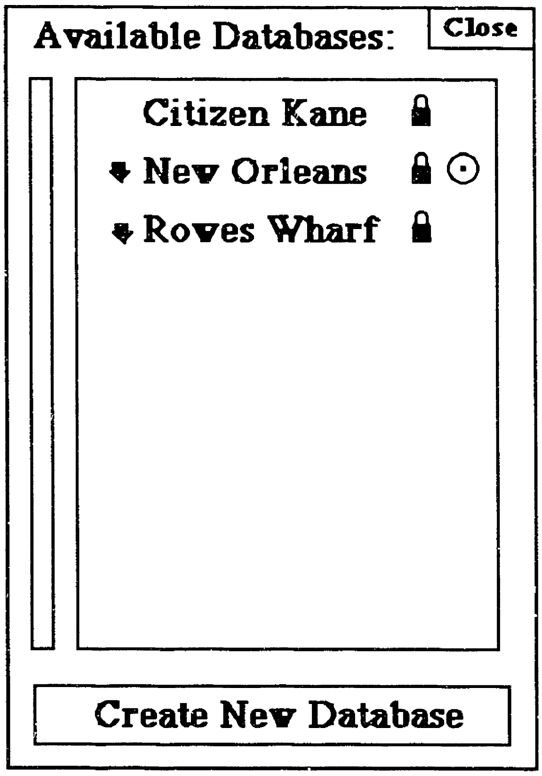


Figure 17: Databases in List Form

The hierarchical display portrays the layered database structure. The major advantage lies in the forced organization. First, it is obvious how one database relates to another and also makes it easy to find all related databases. Second, the searching routines can narrow the search to the opened branches in the case that the personal database, or one of its ancestors, has multiple parents.

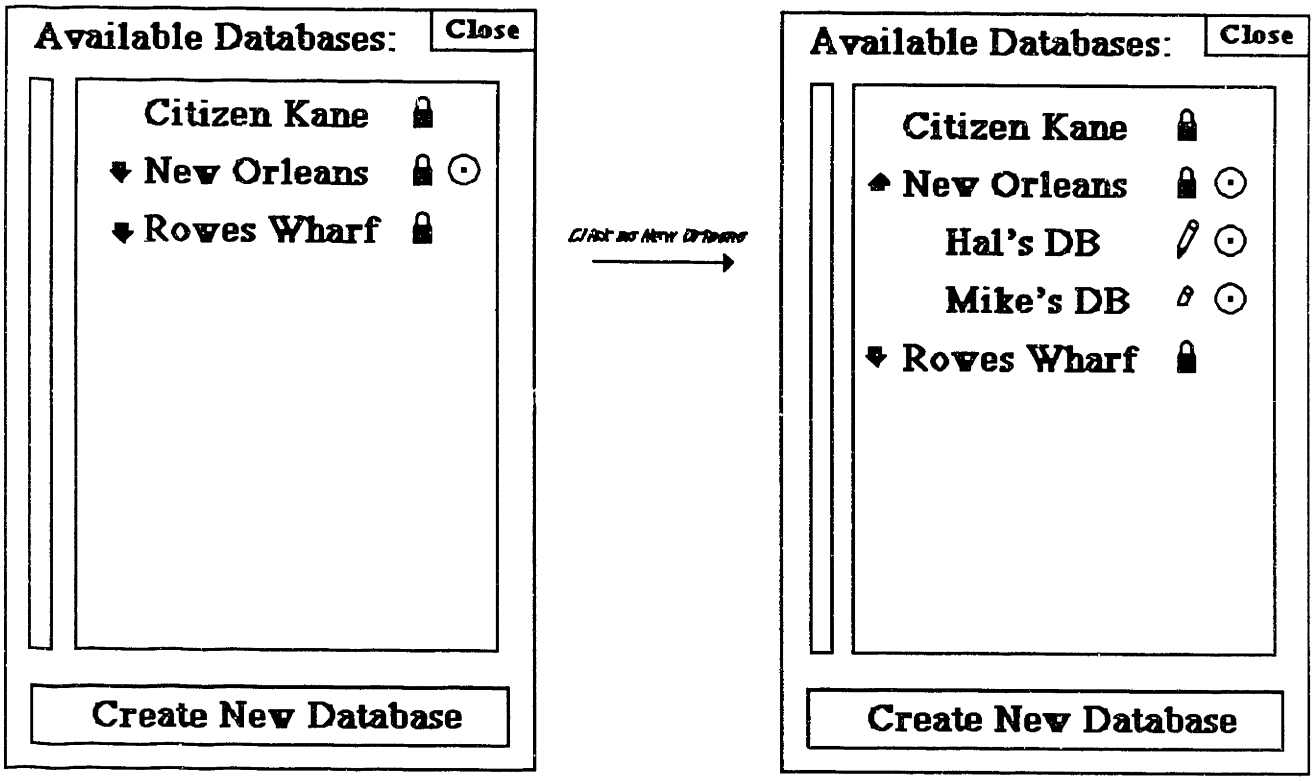


Figure 18: Databases in Hierarchical Form

Searching within the Databases

After exploring the database hierarchy, the user will want to retrieve specific information from it. VEdit handles searches by first defining the scope of the search then the query parameters and finally the presentation style. The goal is to present an efficient but flexible interface to allow the user to restrict his searches to specific parts of the editing environment.

The first important factor in a search is the scope: how much of the editing environment will be scanned for matches. Database searches seem to fall into two broad categories: those limited by the database hierarchy and those limited by a user-defined criteria. The first case, database limitation, means that the query is restrained to a particular branch of the database. For example, looking for all of the shots in Blade Runner containing Darryl Hannah. The second case, user constrained search field, means that the only a specific set of data will be search.

For example, looking for all of the shots in the palette named "Darryl Hannah" (generated by the previous request) that contain the word "quux."

Perhaps the most comprehensive but simplest definition technique is a "shelf." A shelf would be a special area of the workspace where the user would place all of the objects to search across. For example, it could contain three palette icons, one database icon, and several edit icons. In this case, the elements of the palettes and the database along with the edits would form the search set. However, the user must also be able to control the depth of the search for each element. The search depth is controlled through the three icons to the left of the icon. The uppermost icon is a closed book which means that the search is not to open that element at all. The center icon is a partially open book meaning that the object will only be searched one layer deep. The bottommost icon is a microscope which means that the objects will be recursively opened during the search.

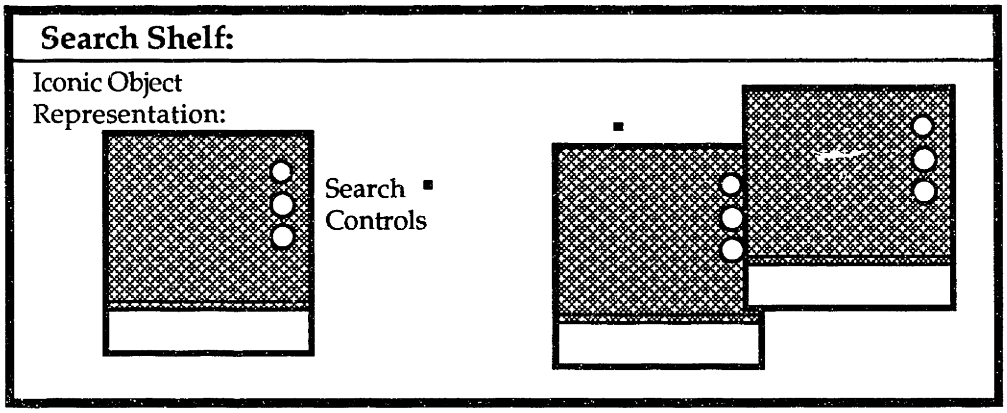
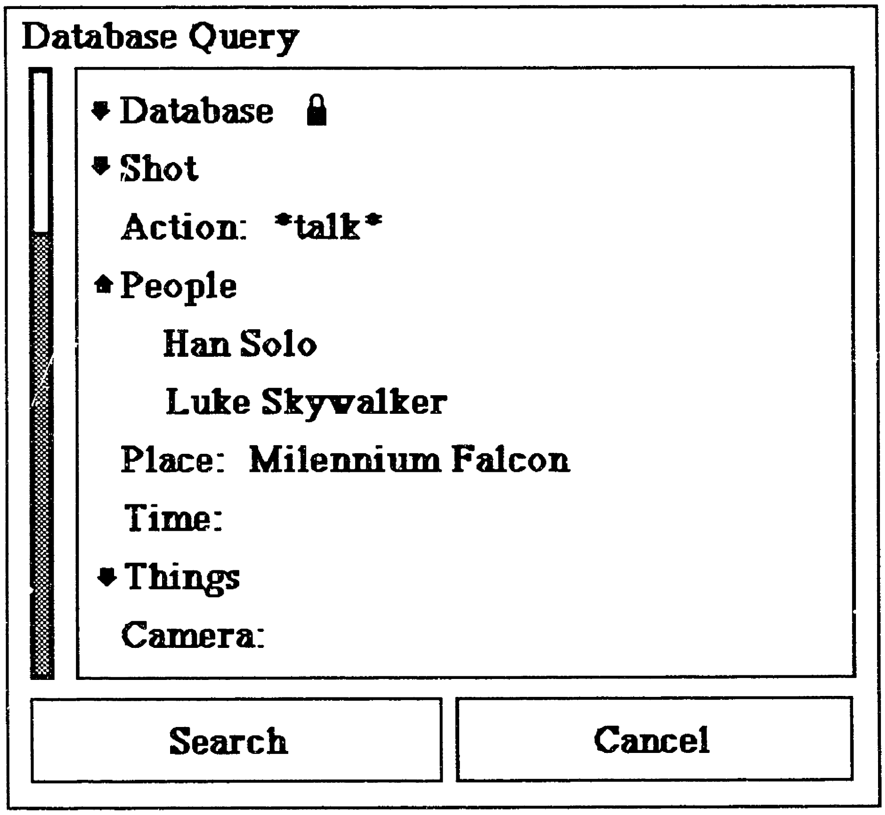


Figure 19: The Search Shelf

Searching across multiple object types poses some significant problems. The first problem lies searching across the individual object types because the field names and classes may vary. Not all of the objects will contain editing information like in point or out point. The current solution is to match those fields in the search set which are defined in the object. In addition, at least one of the fields must match the object because a disjoint set between the object and search set should not yield a match. The other problem lies in conducting the entire search. The solution for a multi-object search is to break the search down into each unique object type and then union all of the resulting response sets. These two steps, matching like fields and taking the union of the results, allows the user to search across any set of objects.

The second element in the search is the search criteria: the parameters for the matching set. Defining the search parameters is perhaps the most difficult aspect of the VEdit interface because the object types vary widely and the fields within them may differ throughout the hierarchy. The strategy for a multiple parameter search closely mirrors the hierarchical structure exploration. The user is presented with a series of objects into which he can enter the search criteria (blank fields are ignored).



Database Query

- Database
- Shot
  - Action: *talk*
- People
  - Han Solo
  - Luke Skywalker
  - Place: Milennium Falcon
  - Time:
- Things
  - Camera:

Figure 20: Establishing the Search Limitations [Kobb, 1989]

The final element of the search is result presentation. The results from a search are displayed in a separate palette named after the search request. By displaying the results in a new palette, the palette's formatting capabilities can be exploited. The two major classes of palettes are icon each object is represented by an icon and list where each object is represented by its name and class name.

Building a Personalized Database

Integrated with the editing system is a database construction kit. The advanced user may even want to add additional fields or entire templates to his own databases. VEdit provides the student with the ability to create personalized databases from creating new objects through incorporating original data into the new database.

Designing a personalized database begins by choosing the objects contained in it. VEdit comes with a collection of stock objects, as shown in Appendix B, from which the user may choose. As shown below, the user can add any existing object type to the new database. In addition, the user can define a new object type. This permits the user to incorporate only the segments of the project database which are of interest and allows the user more useful information in a limited space.

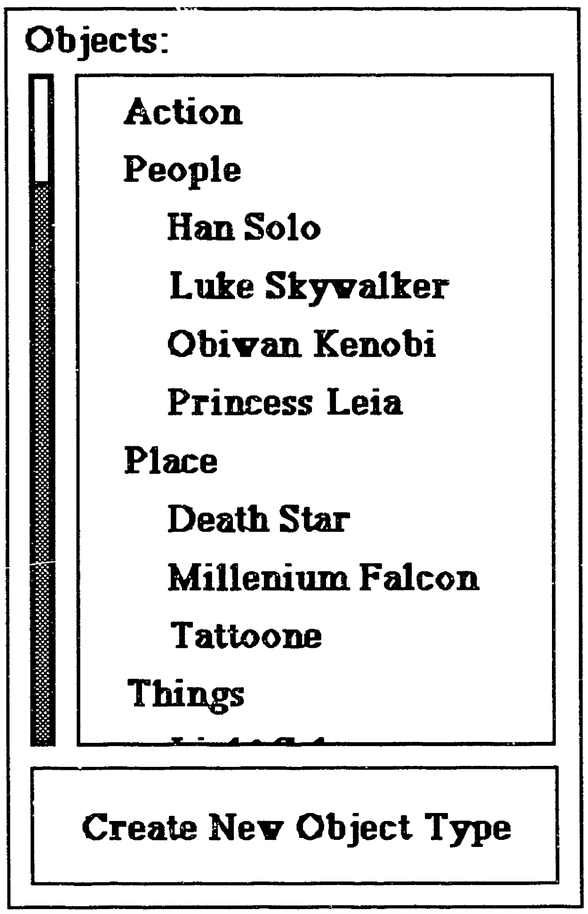
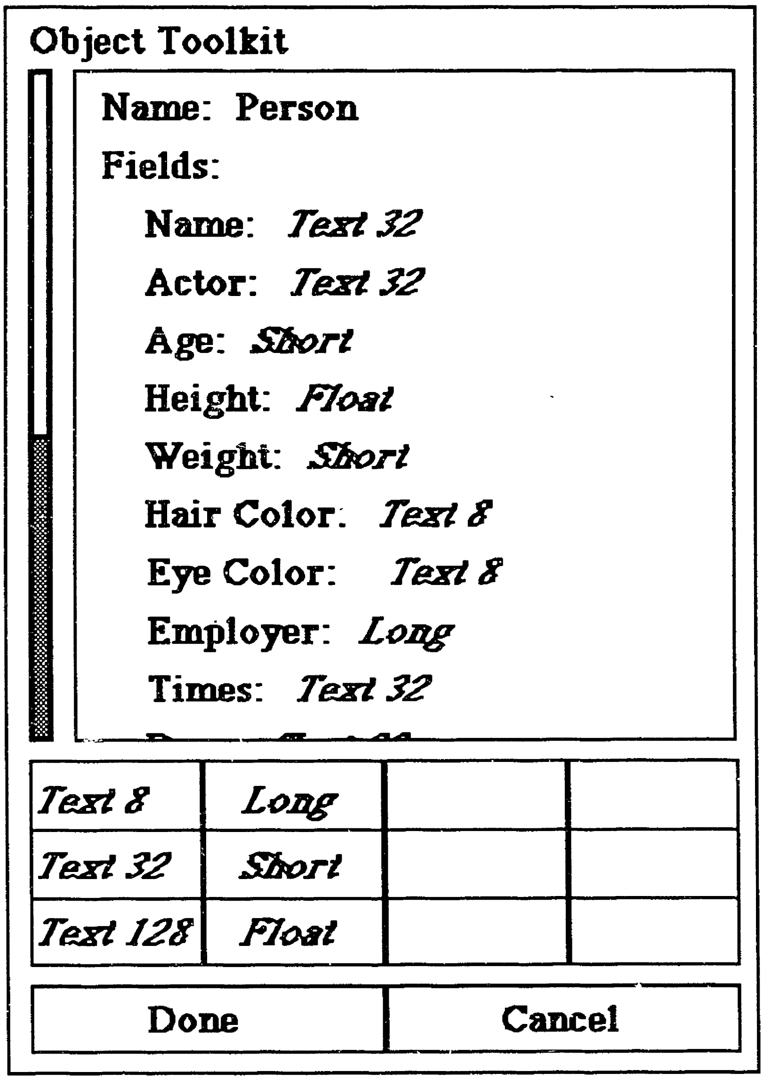


Figure 21: Database Construction Kit [Kobb, 1989]

The second step is defining any new object classes. Here, the object construction kit replaces the database construction kit from before. Once again, the user can select any existing entry type from the menu at the bottom of the construction kit and add it to the current template. Entries can also be removed by the user by first selecting and then pressing the delete button. In addition, the user could incorporate an existing template adding its fields to the ones currently defined. Finally, the user can set the mode for the entry: editable, readable, invisible, etc.


Object Toolkit

Name: Person

Fields:

Name: *Text 32*

Actor: *Text 32*

Age: *Short*

Height: *Float*

Weight: *Short*

Hair Color: *Text 8*

Eye Color: *Text 8*

Employer: *Long*

Times: *Text 32*

Text 8	*Long*		
Text 32	*Short*		
Text 128	*Float*		

Figure 22: Object Construction Kit [Kobb, 1989]

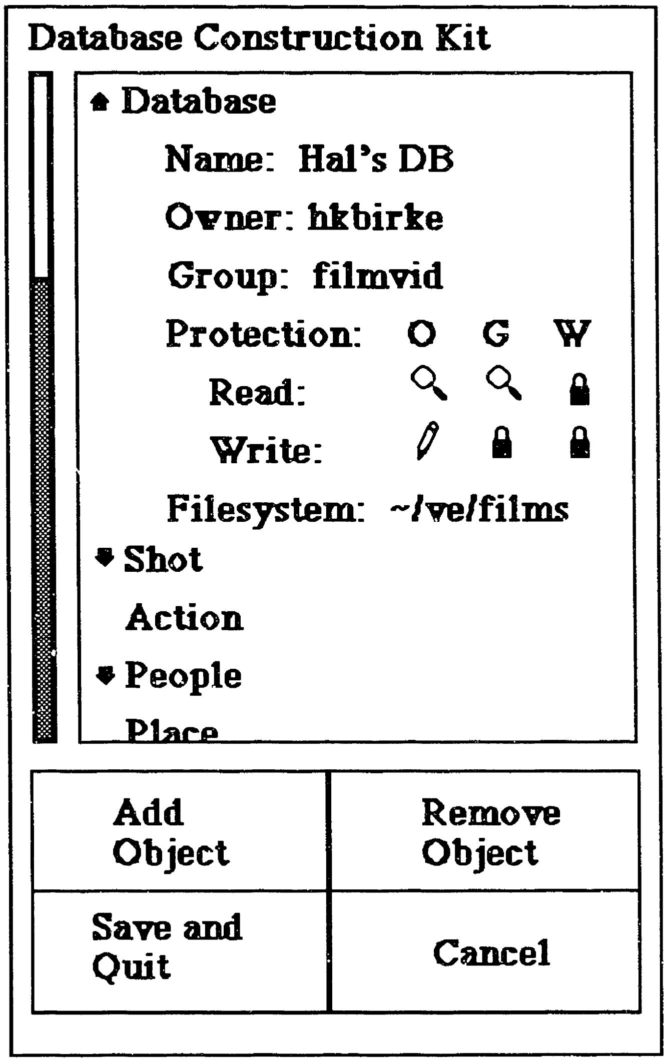
The final step in personalizing a database is incorporating information from the original databases. The incorporation seemingly breaks into two pieces, reference and replacement, which differ only in the time that the conversion takes place. The reference case is converted at run-time, and the replacement case is converted at creation. In either case, fields not present in the current template will not be accessible even if they are present in the preceding template.

Maintaining the Databases

Perhaps the most intriguing capability of the MUD is the built-in maintenance capabilities. In a true multi-user environment, there appears to be a need for administration at the database level to parallel the administrative control at the UNIX level. In response to this apparent need, VEdit has some built-in maintenance tools for environment control, layout, and protection to ease the maintenance of the VEdit editing environment.

One of the problems in a distributed database lies in the physical organization. The first VEdit maintenance tools address the environment hierarchy and layout: how the databases are physically linked and where they are located. These tools were placed in the client for security reasons: the client should control what groups have access to administrate the database. While these tools are part of the client application, they can easily be incorporated into a different client.

Another important issue in a multi-user database is protection control. The second batch of tools are intended to control database access. Once again, the tools themselves are part of the client but could easily be incorporated into a different client. These tools encompass everything from adding internal VEdit groups to changing the modes of various databases. One example is the database protection menu below which allows the user to change the internal protection mode of a database.



Database Construction Kit

▲ Database

Name: Hal's DB

Owner: hkbirke

Group: filmvid

Protection: O G W

Read: 🔍 🔍 🔒

Write: ✎ 🔒 🔒

Filesystem: ~/ve/films

▼ Shot

Action

▼ People

Place

Figure 23: Database Protection Menu [Kobb, 1989]

The third batch of tools form the previously discussed object construction set which allow the user to build new database objects from the standard set of building blocks. Again a part of the client, these tools can be rapidly included in a different client or packaged into a stand-alone construction kit.

Chapter 6: Implementing MUD

Implementation of any server-client system can be broken down into three parts: client interface, network transmission, and server implementation. The server implementation covers the flexible database implementation atop the database package. The network support primarily deals with layering a multiple connection server atop a single user database, making the multiple remote users appear to be the same local user to the database package. The final segment is the client interface which is essentially simulates a local database interface over a network.

Scope of Implementation

Unfortunately, the implementation can be considered minimal at best. Due to severe time constraints, the discussed system has yet to be implemented. In fact, this thesis is a detailed design discussion and specification for a multi-user networked database service and not an implementation critique of one. While the service has not been implemented, the VEdit test case, the motivation for the entire project, has been under constant development for the past ten months.

Networking the Database

The first element in a MUD implementation is the network support and interface. The network interface begins with the database interface and extends through network support into user authentication and advanced error handling protocols. The initial specification for the inter-process communication is detailed in the first section. It is followed by a specification for secure authentication of both the servers and the clients. The last element in the network interface is error recovery.

Network Communications

The first phase of a network implementation is to implement a TCP/IP based client-server and server-server protocol. The protocol defines three ports:

MUD server, primary server, and remote server. The first port, the MUD server port, is used by the primary server to ensure connectivity between all of the servers and the clients. The primary server port carries control communications to and from the primary server. The remote server port carries data between the database servers and the clients.

The MUD server port is used to by the primary server to keep track of the entire database. Clients open connections to the primary server using this port and the primary server listens to this port for sever status information. Database servers use this port for similar reasons. Traffic on this port includes control commands, authentication requests, and connection information.

The second port is the server data connection which is used for the actual client-server communication. This port is used for raw data communication and not control information. As opposed to the MUD server port, traffic on this port does not pass through the primary server to eliminate an excess transmission of the raw data and increase the apparent database speed. To further increase the apparent database speed, the client is not required to re-authenticate to the server. Instead they exchange unique session keys which had been generated previously by the primary server.

The remaining port is the remote server connection which carries the server-server communication. While the communication between the servers may appear to be very similar to the client-server communication, the command set is slightly expanded. In addition, the authentication procedure is different because the servers must re-authenticate with each connection. The additional security is required because the servers all trust one another.

Multiple, Simultaneous Network Connections

After building a network abstraction, the next stage of the implementation is the network server itself. The server implementation is heavily based on Galatea, a network based video device server. Essentially, the servers listen on all open connections and handle requests in a round-robin fashion.

Remote Authentication

One of the important, and often overlooked, elements of a secure networked database is remote user authentication. If the database was used from a single workstation, each user could have a secure, local account. If the connection is from a remote machine, however, verifying the remote user can become very difficult without a remote authentication service.[11] Fortunately, Project Athena provides a such a secure authentication service, Kerberos, precisely for this purpose.

Client/Primary Server

Authentication between a primary server and a client is a four step process. First, the client requests a ticket for the database service located at the primary server. Kerberos returns the MUD ticket and shared MUD key encrypted in the client's session key. Second, the client decrypts the MUD ticket and shared key and transmits the MUD ticket and client session key to the MUD server encrypted in the shared MUD key. Third, the primary server uses its own, local copy of the shared key to decrypt the transmission,[12] extracts the ticket and decrypt that. The primary server then sends a response encrypted in the client's session key back to the client. At this point, the client and server have mutually authenticated.[13]

Primary Server/Database Server

Authenticating between a primary server and a database server is a simpler process. First, the primary server requests a MUD ticket for the database

---

[11]Project Athena workstations are essentially public machines, complete with a published super-user password. The only method to verify the validity of a user is through a remote authentication scheme such as Kerberos.[Miller, 1988]

[12]The local copy of the shared key is stored in /etc/srvtab or another local file located on each server machine

[13]A much clearer and illustrated example can be found in [Transarc, 1989] in section 2.4: a more complex mutual authentication procedure.

server's host. The primary server then sends a generated session key to the database server encrypted in the database server's shared key. The database server decrypts the primary server's request. After obtaining a MUD ticket for the primary server's host, the database server returns the session key encrypted this time in the primary server's key. At this point the two servers have mutually authenticated.

Database Server/Client

The most difficult authentication lies between the database server and the client. The problem lies not in the authentication, but in propagating the client's tickets to the database server. The current authentication scheme requires that the database server be authenticated with the primary server.

Authenticating a client and database server is rather straightforward. First, the primary server (which is already authenticated with the database server) sends the database server's shared key to the client encrypted in the client's session key. Second, the client requests a MUD ticket for the database server from Kerberos as in the client, primary server authentication. The client then sends the new session key encrypted in the database server's key to the database server. The last step is the database server returning its shared key encrypted in the clients session key. Once again, the two processes have mutually authenticated.

The problem lies in receiving a ticket granting ticket (TGT) on the database server but with the clients ticket. Using the new version of the Kerberos library, it is possible to receive a TGT on the remote machine. An example of this procedure is available in [Berkenbilt, 1989].

Network Error Handling

An essential part of any network based server is error correction. Network errors seem to fall into three major categories: momentary, lasting, and filesystem failures. Each type requires different recovery techniques and timetables, but neither should incapacitate the entire MUD.[14]

---

[14]Unfortunately, if the primary server goes off-line for any reason, the entire MUD not only becomes unusable but has to undergo a complete state reset. The database may be unavailable for

Momentary Network Error

A momentary error is essentially a transmission glitch: one or two packets damaged while traveling between the server and the client or a packet repeated, dropped or relocated during transmission. By layering the communication atop TCP/IP, most of these errors are automatically handled before reaching the client application. The TCP/IP protocol itself will recover from momentary packet corruption, omission, relocation or repetition.[15] This leaves the client to handle the lasting failures.

Lasting Network Error

A lasting failure is a network error which causes TCP/IP to break communication and close the socket. These problems are typically caused by one of the machines crashing or the network link between the two machines being isolated. In any case, the remote machine will be unavailable from several seconds to several days.

Galatea uses an appropriate, effective recovery technique. First, there is a server-server query protocol where one server can send an acknowledgement request to another. This query will determine if the remote machine is reachable and operating as a server. If the remote machine is unavailable, the acknowledgement request will time out and the remote machine will be marked as down. In addition, a machine will be marked down if a connection to it is dropped. Finally, if a request to the remote machine times out without being dropped, an acknowledgement is sent. Only a successful response to the acknowledgement will keep the remote machine labelled as active.

Unfortunately, the dual server arrangement in MUD can make error recovery more difficult. The primary server maintains the available server tables

---

many minutes and all queries in process will have to be restarted. The severity of a primary server crash can be mitigated by supporting a multiple primary server configuration similar to the Zephyr arrangement at Project Athena. [DellaFerra, 1989]

[15]A complete description of UNIX networking can be found in [Rochkind, 1985]. Using TCP/IP within a network service is discussed at length in [Secrest, 1986] and [Lefler, 1986].

and serves as a clearing house to the remote database servers for this information. It is theoretically possible for two database servers to each be able to communicate with the primary server but yet be unable to communicate with each other. The difficulty arises in which server to mark down. In fact, this isolation situation could be much more widespread where a large group of servers can communicate with each other but not with any of the servers in the other group. The author has yet to determine an effective recovery technique for the group isolation problem. One possibility is to mark down which ever group is smaller. Another is to mark down the less used group.

On the other hand, the dual server arrangement can greatly increase the robustness of the system. The primary server could assign a different remote server that has access to that portion of the database as a replacement. While this requires maintaining a more complicated database server arrangement, it can be a major advantage in an unstable distributed environment. The replacement permits a relatively invisible operation to replace the newly unavailable database segment.

Once a server is marked down, there must be some method of recognizing that the server has returned. There seem to be two separate modes of recovery depending on the state of the remote server. First, the remote server has just started. In this case, it simply announces itself to the primary server and becomes registered as a database server. The second case is more typical of a network failure: the remote database server does not realize that it has ever been marked as unavailable by the primary server. To recover from this, the simplest method is for the primary server to query the supposedly unavailable database server every few minutes to see if it has reawakened. Incidently, this is Galatea's method of recovery from network failures. In this case, the remote database may be unavailable for some small period of time, typically less than five minutes, after its server could have first been contacted.

Filesystem Error

The remaining error mode occurs with remote filesystems. Due to the distributed architecture of a MUD, not all storage may be local to the database server. Therefore, the data may become unavailable at any time during server operation. It is important that the loss of a file locker does not interrupt database service for more than a few moments.[16] The length of the filesystem timeout can be set when the locker is attached. But once the file locker has become unavailable, it must be reattached when the filesystem returns to operation. In this case, the database server must issue attach requests for that remote filesystem periodically on approximately the same frequency as the primary server polls defunct database servers.

Furthermore, the unavailability or reacquisition of some portion of the database must be transmitted to the primary server. This technique has two important advantages. First, it can permit rapid notification of the clients of the new database status. Second, it permits the primary server to assign a new server/filesystem for that portion of the database. While this requires duplication of the data and a method to resynchronize the separate filesystems when they both become available, it allows the database to remain available in the event of a file server crash.

Primary Server Structure

The primary server is designed to control the database as a whole. This includes maintaining all of the remote connections, dividing the database amongst the database servers, and ensuring that the proper authentications have been made.

Maintaining Remote Connections

The most important role of the primary server is to maintain all of the remote connections. Given the volatile network environment, machines will frequently become inaccessible for extended periods. Managing all of the remote connections can be roughly divided into opening and closing them and ensuring that all servers are operational.

[16]Loss of a remote filesystem will generate operating system-level blocks until either the remote locker becomes available again or the request times out after anywhere from sixty seconds to ten minutes. While sixty seconds is marginally acceptable, ten minutes is not an acceptable timeout.

Opening and Closing Sockets

In the current specification, the primary server has an open connection to each client and database server in operation. The TCP/IP stream connection is created when the remote service becomes available and is closed when it is no longer in use.

Local Cache of Database Servers

Maintaining a local cache of the database server information is crucial for dependable operation. First, a local file containing server information will survive a server or machine crash and enable the primary server to restore state rapidly when restarted. Second, because the cache contains all of the possible database servers, the primary server can poll all of the machines listed there to ensure that they are operational.

Using UDP to Locate Remote Servers

Keeping track of many remote machines can become an intricate demanding task and has no obvious solution. The chosen technique is to send a small UDP packet (a ping) to each remote service, client or server, every few minutes. If the ping is not returned, the connection is closed and the remote service is declared inactive. If the remote service is a server as opposed to a client, then the primary server will continue to ping it every minute until it comes back on-line.

<u>Structuring the Database</u>

The second important task for the primary server is to maintain the database, dividing the database up amongst all of the working servers. In addition to the forementioned database server cache, there is a database segment cache which contains information about each element in the database and which server is its default server.

The database is restructured every time a database server either goes off-line for comes back on-line. In the case of a server going off-line, the server's active segments are distributed amongst the remaining servers wherever possible. If it is impossible to attach a segment to an operational database server, then the segment is marked down and removed from the database. In the case of a server coming back on-line, all of the server's default segments are returned to it.

Maintaining a Primary Server

A primary server can be maintained in two separate methods.The first, and easily implemented, technique is to edit the configuration file and restart the server. The second, and more convenient, is to allow users with a specific authentication, say "database administrator," to change the configuration files over a network configuration. The first technique, restarting the server, has its advantages in a single-user environment because of its simplicity and straightforward implementation. In a multi-user environment, shutting down the primary server (and therefore the whole database) is not a viable option. The remaining choice is to permit database management over the network.

It seems that the easiest technique for implementing the cache files is to run a database server on the same machine as the primary server. This particular database server will only control the database structure and availability information and none of the real data contained in the database. The primary server can then communicate with the local database server using the standard client library to retrieve current database layout and availability information. The only problems with the additional server is if the file system that the data is stored on becomes unavailable or the database server crashes. While not too much can be done about the second failure, the first problem is overcome by placing the layout and availability files on local filesystems. Local storage of the database information should not be a major problem because it only numbers a few hundred bytes per host.

Journals and Logs

In addition to maintaining MUD state information, the primary server must record this information to be used later. While the information could be stored in a local disk file, using MUD as storage seems to be advantageous. The MUD storage could be searched like any other database and could be replicated, backed up, and maintained along with the rest of the MUD.

The simplest technique to permit interaction with the logs through MUD is an unlogged database. By storing the information in a database, it will be able to be formatted and searched rapidly. In a large scale system, filtering the small amount of signal out of the great noise is very important.[17] The database will not be logged otherwise every entry into the log files will create a log entry because of the modification which will in turn be logged. It is evident that this will result in an infinite loop of logging that something was logged.

Log:

key:	unsigned long
type:	unsigned long
Client:	Tuple
Logging Host:	Tuple
offset:	unsigned long
Message:	char 256

Figure 24: Log Record Structure

Database Server Structure

The low-level server implementation can be further subdivided into n portions. The first element is developing an interface for an existing relational database package. The second step is determining a standard relational database layout. The next step seems to be optimizing the relational databases based on the file layout scheme. Finally, the database server needs to propogate or recover from database level errors.

C-Tree

The database itself is implemented using C-Tree, a commercially available relational database product for a wide range of platforms. A commercial software package was chosen because it should operate correctly and any programming

[17]Project Athena has shown that the filtration can be very important. Currently, all workstations on campus centrally log certain system messages. Searching across many thousand messages per day for important ones can be very time consuming and difficult.

errors will be rapidly fixed. Furthermore, the support originates from an outside vendor which further reduces the internal maintenance effort. C-Tree, when compared with other readily available relational database packages, also seems to be a significant win. The package seems reasonably bug free, flexible, and stores information in network byte order so it can be read on all machines.[18]

File Layout

MUD divides the database into a set of relational databases with each relational database containing an object type. A database is divided into several smaller ones based on object type because relational databases typically require that all of the records in a database file have the same format. Remembering that a database may be broken up into more than ten relational databases at the C-Tree level, file management becomes an important issue. The maximum number of simultaneously open files is typically either thirty-two or sixty-four. While sixty-four files may seem to be quite a large number of files, they are quickly filled with database connections and queries.

The easiest method of avoiding this pitfall is to never leave relational databases open unless they are actively being searched. In other words, open the database file and the appropriate relational databases, conduct the search and then close them all. This technique minimizes the number of files open at one time by keeping the files open for the minimum amount of time. Unfortunately, repeatedly opening and closing large numbers of files is not very efficient.

An alternate method to avoid too many simultaneously open files is to split up the MUD database servers among several machines. While this greatly adds to the implementation complexity and interserver message traffic, it will reduce the number of open files by reducing the number of databases used by a given server. For an initial implementation, the first method's simplicity more than offsets the second method's speed advantage.

---

[18]While network byte order may seem insignificant, it permits the database server to be run on any hardware platform. If the data was stored in machine byte order, databases created with the server running on an RT (big-endian architecture) will not be usable when the server runs on a VAX (little-endian architecture).

Using Indices

Indexing, pre-sorting the database by certain fields, can greatly improve the access time and search time across the databases. Indexing replaces the run-time search that is O(1) with a precomputed indexed lookup that is O(log n). This technique has its greatest advantage in large, static databases. The major problem with indexing lies in the pre-computation because the index file must be recomputed every time a value in the indexed field changes or a record is added or deleted. But in an essentially static database, indexing on commonly searched records can result in a major performance advantage.

C-Tree directly supports indexing by storing the tabulations in separate index files. It also contains a complete set of routines to create, search across, and maintain these index files. The problem with indexing is that the fields must be specified. The database could maintain a histogram for each database segment like Ingres, but the complexity and computation involved seem to outweigh the automation. MUD's solution is to permit the user to specify which fields should be indexed.

Record Locking

Record locking is very important in a multi-user environment. The results of multiple users modifying the same record simultaneously could be catastrophic. With MUD, the first element in the locking scheme is to ensure that a given database is able to be written to by only one server. MUD accomplishes locking a database to a server by opening the C-Tree files in exclusive-only mode, meaning that no other process can open the file. Once the database is locked to a server, the individual records must be lockable by a client.

C-Tree, however, does not directly support record locking under 4.3BSD because the operating system does not support it either. Per-record locking can be simulated easily by maintaining a lock index for each database.[FairCom, 1988] Because only one server will use a specific database at one time, it will simply maintain the state state of each record using the lock index file, simply one bit for each record in the database. Maintaining the lock information in a file is advantageous because it will survive server crashes and database reassignments.

Caching Query Results

If the database server were to cache the results of expensive database operations, server performance may improve. The most frequently used, computationally expensive command is the query because it typically traverses a large segment of the database to find the matches. Caching results of recent queries can be advantageous because applications will often make repeated, similar requests in a short time period. The caching also is advantageous in a multi-stage query for precisely this reason.

The disadvantages of result caching lie in server memory usage and complexity. The server must be extremely to update the cached request if any of the queried databases have changed. The simplest technique is to flush the cache every time a searched database is modified. A more efficient, complicated approach is to check the cache against the modified data on a per-record basis. This technique permits the cache to be left relatively intact, only changing the information that is modified, and improves the longevity of the cache. The other major caching problem is memory usage: the server size has just been dramatically increased by storing all of this additional information. Even though the cache only consists of database tuples, there may be thousands of matches to a given request. Furthermore, the more queries that are cached, the faster the repetitive search and the slower the simple operation because the cache must be checked for each write.

Caching becomes a significant advantage in large, static databases because it will eliminate a long series of relational database accesses. In smaller databases, the performance gain is not as evident because a smaller percentage of time would be spent searching for a query with the same number of matches. In dynamic database, caching slows down the write operations as each changed record must be validated in the cache.

Avoiding Variable Length Records

Most database packages including C-Tree permit variable length records, but they often attach a large performance penalty with the varying record length. MUD avoids using a variable length record structure in most cases by either record shadowing or flat-file referencing. The preferred technique depends on the nature of the variable length record and whether the variable length portion is often searched.

File Replacement

If the record structure contains extensive descriptive information, a text description of an item, flat-file referencing can remove the body of the text from the record and place it in a separate UNIX file. Instead, a file system name and file name, both small, fixed length fields, are placed in the record structure. The resulting record structure is a fixed length record.

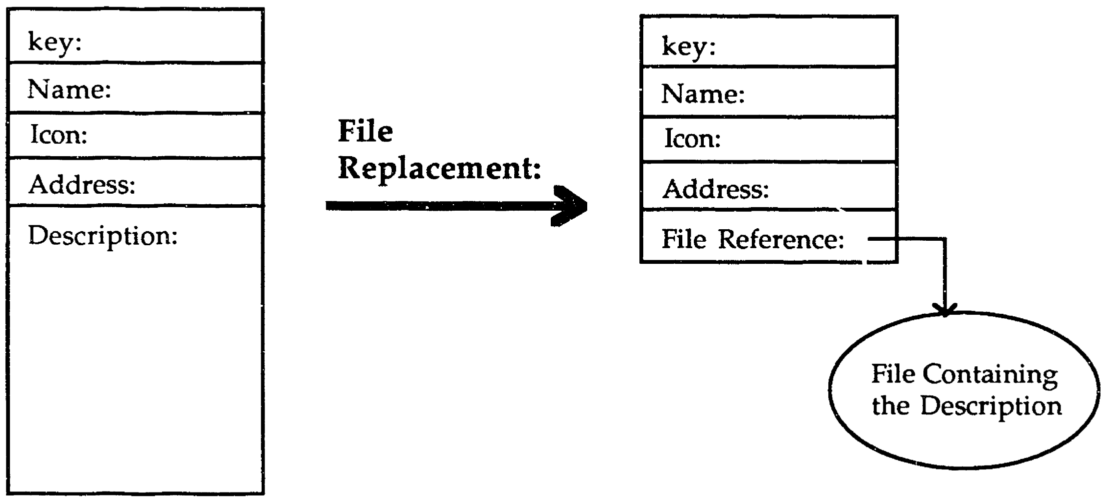


Figure 25: Results from Using File Replacement

File replacement is useful only when the replaced information is not used in searches. Two problems arise when attempting to search across this type of record. First, the data is located in a separate file which much be opened, read, and then searched. The extensive file i/o incurs a significant speed penalty. Second, and most important, C-Tree does not directly support searching on a non-C-Tree file. As long as the replaced data is not searched, however, it will greatly speed searching the rest of the record.

Record Shadowing

The other technique to translate a variable length record into a fixed length structure is record shadowing, replacing a single entry with one covering several separate records. Record shadowing is more useful for discrete data than descriptive text.

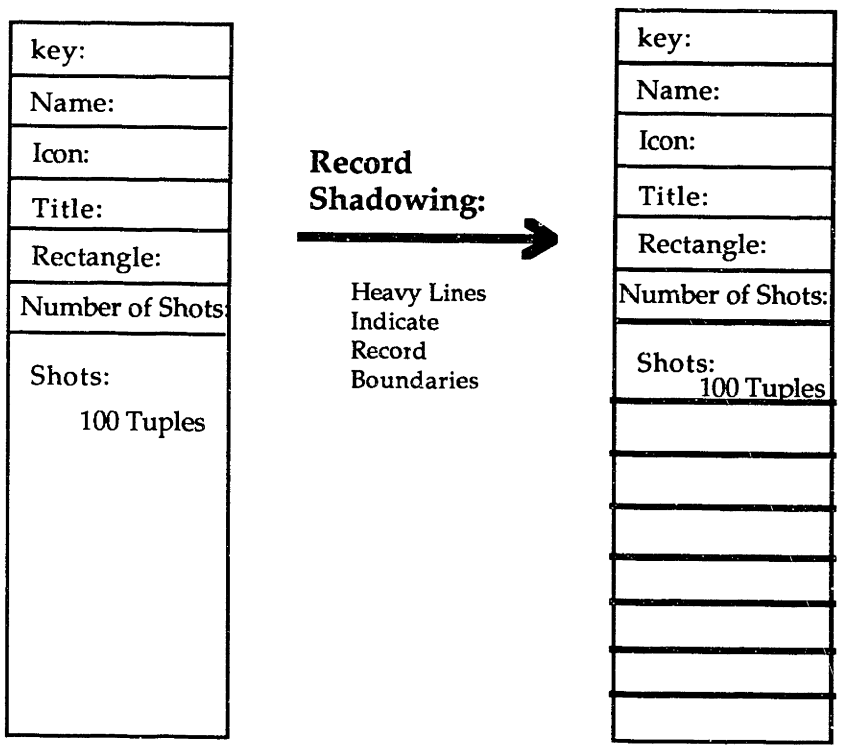


Figure 26: Record Shadowing

Shadowing has several important disadvantages compared to a simple fixed length record database which range from decreased performance and additional complexity. First, the record becomes longer because it must now include additional fields. Second, the routines to extract and record information become more complex because they must handle the multiple entry records. Finally, searching across this database will take longer than the simple fixed length database because the additional entries must be examined.

Most of the disadvantages can be compensated for by adding two fields to each entry: position and next. The entry's location within a record is denoted by the position field. The next field gives the record number of the next entry in the chain. By combining these two, it is possible to extract an entire record from the database as a O(1) operation. While the extraction will be slower than extracting a single record because the entries need not be consecutive, the extraction time does not depend on the size of the database.[19]

Record shadowing encodes the variable length record into a series of entries. Shadowing yields improved performance compared to file replacement when searching across the variable length fields and to true variable length records in all cases. However, it may also increase the complexity and size of the database more than file replacement. Shadowing seems appropriate for discrete information which is frequently searched but not for descriptive text.

Journals and Logs

The database server's logs are very similar to the primary server's. They are maintained in the same format and server the same purposes: audit trace and error recovery.

Error Handling

The most challenging segment of the implementation is error handling. Properly catching, evaluating, and propagating errors in a multi-user environment is difficult because some errors will not only affect the local server but also may affect other database servers and clients. These errors are divided into two major sets: UNIX file errors and database errors.

UNIX file errors are the errors dealing with the filesystem including improper permissions to read the file or non-existent files. A filesystem error, while it may be triggered by a C-Tree command, does not lie within the database

[19]Strictly speaking, this is not true. In a large database, it is more probable that the database will be spread across a large section of the physical disk drive which would increase the drive seek time between records.

itself but instead hinges on the availability of the database. These errors are propagated back to the primary server. Reactions to these errors range from ignoring them to retrying the command or marking that portion of the database unavailable.

Database errors are those errors internal to C-Tree not dealing with the file system. Examples of database errors include illegal search parameters, no matches, and invalid or mismatched database formats. The other important segment of database errors are authentication errors: where the user is not permitted to access or write the requested information due to internal MUD protection violations. While these errors may be less severe than a filesystem errors, they are handled similarly. Typically an error code will be returned to the client through the primary server notifying them what class the problem fell into.

Client Function Library

The final element in the initial implementation is the client-side interface. Essentially, the client library consists of four parts: initialization, database management, database interaction, and additional utility functions. Before delving into the data transfer and storage protocol, let us begin with establishing server connections and authenticating both the client and the server.

Initialization

MUD initialization is divided into two major areas: server connections and authentication. The first area, server connections, creates and closes sockets between the client and the primary server. Authentication is a set of routines based on the Kerberos remote authentication system which verify not only the client but also the primary server.

MUD Connections

The first step in using a MUD is to establish a connection to the primary MUD server. The command, MUD_OpenServer(), establishes the connection to the primary server, performs an initial client authentication, and returns a MUD structure. MUD_CloseServer() closes a connection with the specified primary

MUD server. In addition, it frees all client storage associated with that database and causes the database servers to flush all information including Kerberos tickets pertaining to the client performing the MUD_CloseServer(). A MUD_CloseServer() is performed automatically when the client exits.

```
MUD MUD_OpenServer (host, user)
      char *host;
      char *user;
```

host	Specifies the host name of the system on which the primary MUD server is running. If the host is NULL then the default MUD server will be contacted.
user	Specifies the user name to attempt authentication with the remote MUD server. The user must already have Kerberos tickets on the local host. If a NULL user is specified then a generic authentication will be performed.
return value	If NULL is returned, the server connection failed. If the authentication fails, a MUD structure will be returned, but the user fields will be empty. In either case, the error handler will be called if set.

```
int MUD_CloseServer (mud)
      MUD mud;
```

mud	Specifies the connection to be closed.
return value	TRUE is returned if the close is successful.

Authentication

MUD requires that the client authenticate with the primary server. MUD_Authenticate() has two primary uses. First, it permits the client to reauthenticate to the primary server. Given the eight hour lifetime of Kerberos tickets in the Athena environment, it is possible for a session to span multiple ticket sets. Second, it permits the client to authenticate to the server as multiple users to increase access. For example, the client could authenticate as a user and also as a database administrator to perform some maintenance activities. MUD_FlushAuthentication() causes the user's Kerberos tickets to be removed from all remote servers.

```
int MUD_Authenticate (mud, user)
      MUD mud;
      char *user;

int MUD_FlushAuthentication (mud, user)
```

```
MUD mud;
char *user;
```

mud The database connection returned by MUD_OpenServer().

user Specifies the user name to attempt authentication with the remote MUD server. The user must already have Kerberos tickets on the local host. If a NULL user is specified then a generic authentication will be performed.

Errors

MUD has an error recovery interface similar to Galatea and the X Window System. The recovery routine is called directly by the client library when an error is detected. The routine recovers from the error and the client program continues executing.

Status

Most MUD procedures can generate a wide variety of the errors detailed in Appendix A. Instead of returning the error number to the client, MUD simply returns a zero from the procedure indicating that it failed. MUD sets an element in the MUD structure, MUD_Error, to the appropriate error number. Usually, the client will not need to deal with this error number unless it installs its own error handler.

Error Handler

Like Galatea, MUD permits a client defined error handler which will be called whenever an error code is returned by the primary MUD server. The procedure is set through MUD_SetErrorHandler() and can be retrieved with MUD_GetErrorHandler(). In addition, MUD_HandleError() will execute the defined error handler. While there are large numbers of possible error codes, several types are easily handled and oft arising.

```
int MUD_GetErrorHandler (mud, handler)
      MUD mud;
      void (**handler) (mud, handler);

int MUD_SetErrorHandler (mud, handler)
      MUD mud;
      void (*handler) (mud, int);

int MUD_GetDefaultErrorHandler (mud, handler)
      MUD mud;
      void (**handler) (mud);
```

```
int MUD_SetDefaultErrorHandler (mud)
      MUD mud;

int MUD_HandleError(mud)
      MUD mud;
```

mud The database connection returned by MUD_OpenServer().

handler An error handler procedure to be called when errors are reported by the primary server. The procedure takes only one parameter, a MUD server structure.

Database Management

The database management section of the client library is also composed of two major parts: database structure and template storage. The first section, database structure, returns information about the database hierarchy to the client. The second element of database management provides a method for uploading templates from the client to the server and also downloading them from the server to the client.

MUD uses a hierarchically arranged database where the individual segments of the database are linked together. While the complexity of the links may vary from one application to another, the links still exist and the database structure can be extracted. MUD_GetCurrentDatabase returns the current database segment. MUD_GetTopLevelDatabase yields the segment considered to be the database root by the primary server. MUD_GetParentDatabases returns all database segments directly related to the given segment. Finally, a complete list of all children of the current segment is returned by MUD_GetChildDatabases. The remaining function in this portion of the library, MUD_SetCurrentDatabase, sets a new current database segment.

```
Database MUD_GetCurrentDatabase (mud);
      MUD mud;

int MUD_SetCurrentDatabase (mud, dbase);
      MUD mud;
      Database dbase;

Database MUD_GetTopLevelDatabase (mud);
      MUD mud;

int MUD_GetParentDatabases (mud, dbase, parents, maxParents);
      MUD mud;
      Database dbase, *parents;
      int maxParents;
int MUD_GetChildDatabases (mud, dbase, children, maxChildren);
      MUD mud;
      Database dbase, *children;
      int maxChildren;

int MUD_CreateDatabase (mud, dbase)
      MUD mud;
      Database database;
```

mud — The primary server connection returned by MUD_OpenServer().

dbase — Database structure representing a given portion of the overall database environment. This is returned by MUD_GetCurrentDatabase() and MUD_GetTopLevelDatabase().

parents — An array of database structures that are parents to the given database. In other words, all of the databases that are referenced in the template segment of the given database. This array must be pre-allocated by the client.

maxParents — Maximum number of parent database structures to load. Typically this is the size of the **parents** array.

children — An array of database structures that are children of the given database. In other words, all of the databases that directly use information present in the given database. This array must also be pre-allocated by the client.

maxChildren — Maximum number of child database structure to upload. Typically this integer is the size of the **children** array.

return value — For MUD_GetCurrentDatabase() and MUD_GetTopLevelDatabase(), the return value is the corresponding database structure. MUD_SetCurrentDatabase() and MUD_CreateDatabase() use the return value as a status (see above under Errors). The MUD_GetParentDatabase() and MUD_GetChildDatabases() return the total number of matching databases.

The other half of of database management is the template control routines. These procedures control exchange templates between the client and the servers as well as control which templates are available to the database servers. Because a template is another object class, creating, storing, and loading them can be performed using the standard read and write routines specifying a template object class. The remaining routines break down into two rough groups: upload and download. To send a template either to or from the MUD server there is a single template transmission function, either MUD_GetDatabaseTemplate() and MUD_SetDatabaseTemplate(), and a multiple template transmission function, MUD_GetDatabaseTemplates() and MUD_SetDatabaseTemplates().

```
int MUD_GetDatabaseTemplate (mud, dbase, objectType, template)
      MUD mud;
```

```
        Database  dbase;
        int  objectType;
        Template  template;

int  MUD_SetDatabaseTemplate  (mud,  dbase,  objectType,  template)
        MUD  mud;
        Database  dbase;
        int  objectType;
        Template  template;

int  MUD_GetDatabaseTemplates  (mud,  dbase,
                                              templates,  maxTemplates)
        MUD  mud;
        Database  dbase;
        Template  *templates;
        int  maxTemplates;

int  MUD_SetDatabaseTemplates  (mud,  dbase,  number,  objectTypes,  templates)
        MUD  mud;
        Database  dbase;
        int  number;
        int  *objectType;
        Template  *templates;
```

mud The database connection returned by MUD_OpenServer().

dbase Database structure corresponding to the desired element of the database environment.

template The template for the given database structure for a specific object class. The template contains all of the formatting information for each object class present in the database.

templates An array of templates for a given database. This array must be pre-allocated by the client.

maxTemplates Maximum number of templates to load. Typically this will be the size of the templates array.

Database Interaction

After establishing which portion of the database to work with, the next step is to be able to load elements from and store them into the databases and then find them again. The first third of this library portion are the routines which retrieve information from and store information into the database. The second, and more complex, portion of database interaction is the searching itself. The final third of the interaction segment is record locking.

Loading and Storing

Retrieving information from and writing it into the database are rather straightforward two-step procedures. First, the remote server must verify that the user has the desired access. Second, the data must be transmitted either to the server or to the client depending on the request. MUD_StoreObject() and MUD_StoreObjects() enter either one or more than one element into a given database segment. If a segment is not specified (the tuple contains a negative key) then a new record is allocated and the key number is placed in the tuple. MUD_LoadObject() and MUD_LoadObjects() read either one or multiple records from the given database segment.

```
int MUD_StoreObject (mud, dbase, tuple, objType, object)
      MUD mud;
      Database dbase;
      Tuple tuple;
      int objType;
      Object object;

int MUD_StoreObjects (mud, dbase,
                              tuples, objType, objects, numObjects)
      MUD mud;
      Database dbase;
      Tuple *tuples;
      int objType;
      Object *objects;
      int numObjects;

int MUD_LoadObject (mud, dbase, tuple, objType, object)
      MUD mud;
      Database dbase;
      Tuple tuple;
      int objType;
      Object object;

int MUD_LoadObjects (mud, dbase,
                              tuples, objType, objects, numObjects)
      MUD mud;
      Database dbase;
      Tuple *tuples;
      int objType;
      Object *objects;
      int numObjects;
```

mud	Connection returned by MUD_OpenServer().
dbase	The database to load from or store into.

tuple If the operation is reading from the database, it specifies which database record to load. If the operation is writing to the database, it returns the database record that was written.

tuples If the operation is reading from the database, it specifies which database records to load. If the operation is writing to the database, it returns the database records that were written. This array must be pre-allocated by the client.

object Object that should be stored into or read from the database in a single object operation. The object must be pre-allocated by the client.

objects An array of objects that should be stored into or read from the database during a multiple object operation. This array must be pre-allocated by the client.

numObjects The number of objects to read or write in a multiple object operation. The size is typically the smaller of the object and tuple arrays.

Searching

The MUD library provides several classes of functions to permit a variety of different queries to be performed. These searches are divided based on database traversal and matching functions. Before delving into the actual library functions, the basic functionality needs to be discussed. There are two major control parameters for searching: search type and ALU function.

Given the hierarchical structure and large scope of the database, controlling the progression of a search is very important. The search type determines how the search proceeds from the initial database segment. The currently supported categories permit searching upwards, downwards, globally, and locally limited by the client's access permissions.

The ALU function controls whether or not a given record is considered a match. The ALU function name is directly derived from the like-named logical operator. Most databases limit the user to a simple "and" function which matches records across all fields. MUD has added several additional ALU functions for additional functionality: or, nand, nor, xor, and noop. For example, the "or" function matches records which have at least one defined field the same while the "nor" function matches records with no fields the same.

The database also limits the number of objects which can be matched and returned to the client. MaxObjects controls the number of objects, passed to the server, which may be used in a single query. MaxReturn limits the number of records which will be matched until the query aborts. This is intended to avoid accidental, large queries which could match the whole database.

The MUD servers maintain these search parameters for each client based not only on the clients authentication level but also on client requests. While the client cannot request more information or more general search categories than the default parameters based on the authentication level, the client can further limit them. MUD_SetSearchParameters() enables the client to establish a different parameter set. MUD_GetSearchParameters() retrieves the client's current search parameters. Finally, MUD_GetDefaultSearchParamaters() returns the default search parameters and MUD_SetDefaultSearchParamaters() returns the search controls to the initial server limits. These routines are all atomic: if any of the three parameters is outside the server imposed limits, the entire request will fail.

```
int MUD_SetSearchParameters (mud, searchClass, searchOperation,
                             maxObjects, maxReturn)
      MUD mud;
      int searchClass;
      int searchOperation;
      int maxObjects;
      int maxReturn;

int MUD_SetDefaultParameters (mud)
      MUD mud;

int MUD_GetSearchParameters (mud, searchClass, searchOperations,
                             maxObjects, maxReturn)
      MUD mud;
      int *searchClass;
      int *searchOperation;
      int *maxObjects;
      int *maxReturn;


int MUD_GetDefaultSearchParameters (mud,
                                    searchClass,
                                    searchOperations,
                                    maxObjects,
                                    maxReturn)
      MUD mud;
      int *searchClass;
      int *searchOperation;
      int *maxObjects;
      int *maxReturn;
```

mud	The database connection returned by MUD_OpenServer().
searchClass	Specifies the desired search pattern, one of SEARCH_TREE, SEARCH_BRANCH, SEARCH_LEAF, SEARCH_GLOBAL.
searchOperation	Specifies the ALU operation used during the search. It must be one of SEARCH_AND, SEARCH_OR, SEARCH_XOR, SEARCH_NAND, SEARCH_NOR, SEARCH_NOT, or SEARCH_NOOP.
maxObjects	Specifies the maximum number of objects allowed to be searched for at one time.

maxReturn	Specifies the maximum number of matches to return to the client at one time regardless of maximum number specified by the search command.

In addition to setting search parameters, clients can manually control search caching within the database servers. MUD_ClearSearchSet() allows the client to free a specific search set. MUD_ClearSearchSets() permits the client to flush all search sets associated with a particular segment of the database. These two routines are intended to be called after a multiple stage search has been completed. MUD_ClearAllSearchSets() enables the client to free all server-side cached search information. This routine is most effective in recovering from MUD_MEMORY_EXHAUSTED errors during search operations. In all cases, the client must be authenticated with the primary server and have sufficient permissions to execute the commands. All search set commands return TRUE only if wholly successful. While a FALSE return value may be returned, some of the search sets may have been cleared before the error condition arose. If a query is attempted using a cleared search set, the original query is performed and then the requested operation is performed.

```
int MUD_ClearSearchSet (mud, searchSet)
        MUD mud;
        int searchSet;

int MUD_SetSearchSet (mud, numObjects, Objects)
        MUD mud;
        int numObjects;
        Object *object;


int MUD_ClearSearchSets (mud, dbase)
        MUD mud;
        Database dbase;

int MUD_ClearAllSearchSets (mud)
        MUD mud;
```

mud	Primary server connection returned by MUD_OpenServer().
dbase	Database segment for which all search sets are flushed.
searchSet	Search set, as returned from a search command, to flush from server memory.

The next group of routines in the search segment of the client library are the search routines themselves. These fall into two groups: the general search routine and several convenience search routines. The general search routine is a more powerful but slightly more cumbersome than the convenience routines.

The general search routine, MUD_Search, uses the current search set, ALU function, and progression to conduct a multi-object query. The major advantage of the general search command is the ability to search against multiple objects. This reduces the number of server requests, cached search results, and network transmissions. Multiple, simultaneous searching has its advantages over a series of single queries and then intersecting or unioning the results because the database is only searched once.

```
int MUD_Search (mud, dbase, numObjects, objType, object)
        MUD mud;
        Database dbase;
        int numObjects;
        int *objType;
        Object *object;
```

mud	Primary server connection returned by MUD_OpenServer().
dbase	Database segment which acts as the root for the search.
numObjects	Number of objects included in the search list.
objType	Class of the search object. Used to reference the object template within the server. This is an array of integers.
object	The keys to search with are in this array of objects. Only fields present in the objects are used as match criteria. The object types must exist in the root database.

The convenience routines simply specify the search class, or the database traversal mode, in the function name along with the ALU function. The functions do not change any of the defined search parameters; they are defined internally to the convenience routines. In fact, the convenience routines' functionality is reduced because they do not accept a list of objects to match against. These routines may signal a variety of error conditions ranging from protection violations to illegal search parameters and too many matches.

There are four convenience routines: MUD_LeafSearch(), MUD_BranchSearch(), MUD_TreeSearch(), and MUD_GlobalSearch(). The first routine searches only the given database segment. Branch searching begins with the named database segment and continues downward. Tree searching begins with the specified segment and continues upward in the database. The final, exhaustive search mode is global search which searches the entire database.

```
int MUD_LeafSearch (mud, dbase, searchSet, objType, object)
      MUD mud;
      Database dbase;
      int searchSet;
      int objType;
      Object object;

int MUD_TreeSearch (mud, dbase, searchSet, objType, object)
      MUD mud;
      Database dbase;
      int searchSet;
      int objType;
      Object object;

int MUD_BranchSearch (mud, dbase, searchSet, objType, object)
      MUD mud;
      Database dbase;
      int searchSet;
      int objType;
      Object object;
```

mud	Primary server connection returned by MUD_OpenServer().
dbase	Database segment which acts as the root for the search.
searchSet	Specifies a previously searched group as an initial search set.
startingPos	Specifies the position within the search set to begin the upload.
objType	Class of the search object. Used to reference the object template within the server.
object	The key to search with. Only fields present in the object are used as match criteria. This object must exist in the root database.

Once the application has conducted searches it may want to combine the results of those searches before loading the data over the network. MUD provides two routines, MUD_IntersectSearchSets() which generates a new search set consisting of the tuples present in both sets and MUD_UnionSearchSets() which generates a new search set consisting of the tuples contained in both sets.

```
int MUD_IntersectSearchSets (mud, set1, set2)
      MUD mud;
      int set1, set2;

int MUD_UnionSearchSets (mud, set1, set2)
      MUD mud;
      int set1,set2
```

mud	The primary server connection returned by MUD_OpenServer().

set1, set2	Key numbers to two separate search results.

After conducting the search, the client must then retrieve the data from the remote server. MUD_GetResults() permits the client to upload a specific segment from the overall results based on starting position and number of results to load. It returns the number of results actually uploaded from the database server. As before, MUD_GetResults can signal a variety of errors ranging from MUD_NO_PERMISSION to MUD_BAD_PARAMETER.

```
int MUD_GetResults (mud, searchSet, startingPos, results, maxResults)
        MUD mud;
        int searchSet;
        Tuple *results;
        int maxResults;
```

mud	Primary server connection returned by MUD_OpenServer().
searchSet	Specifies a previously searched group as an initial search set.
startingPos	Specifies the position within the search set to begin the upload.
results	An array of database tuples containing the results of a specific search. This array must be pre-allocated by the client.
maxResults	Maximum number of database tuples to return to the client. Typically, this is the size of the results array.

Locking

The final element of the client interaction library is record locking: allowing the client to lock certain records or groups of records so that they can be updated. The records are automatically unlocked when the MUD connection is closed.

```
int MUD_LockRecord (mud, database, tuple)
        MUD mud;
        Database database;
        Tuple tuple;

int MUD_LockRecords (mud, database, number, tuples)
        MUD mud;
        Database database;
        int number;
        Tuple *tuples;

int MUD_LockSearchSet (mud, searchSet)
        MUD mud;
        int searchSet;
int MUD_UnlockRecord (mud, database, tuple)
      MUD mud;
      Database database;
      Tuple tuple;

int MUD_UnlockRecords (mud, database, number, tuples)
      MUD mud;
      Database database;
      int number;
      Tuple *tuples;

int MUD_UnlockAll (mud)
      MUD mud;
```

mud	Primary server connection returned by MUD_OpenServer().
database	Specific database to operate on.
tuple	Individual record specification to lock or unlock.
number	Number of tuples to lock or unlock.
tuples	Array of record specifications to lock or unlock
searchSet	Specifies a previously searched group as an initial search set.

Utility Functions

In addition to searching, loading, and storing, the MUD Client Library permits the user to reset the connection, flush client-specific information from the server, and upload performance and usage statistics from the server. These routines are not really intended for normal use, but they are intended to aid in debugging, benchmarking, and error recovery.

MUD_ResetServer() performs a full reset of both the client-side and server-side of MUD. The only difference between a reset and closing and reopening the mud connection is that the connection with the primary server remains open during the reset. The client, however, must reauthenticate, reestablish connections to database servers and reset any internal client MUD parameters. Resetting the server does not require valid Kerberos tickets and can be performed by all clients. MUD_ResetServer() returns TRUE only if the operation is fully successful. It is useful to note, however, that the only failure mode for a reset is MUD_PRIMARY_RESET meaning that the client cannot contact the primary server.

```
int MUD_ResetServer (mud)
```

```
        MUD mud;
```

mud The database connection returned by MUD_OpenServer().

If a full reset is not desired, the client can also flush all information about itself from the database servers. MUD_FlushServer() leaves the connections intact while flushing search results and other client-specific information from the database servers. Since the connection between the primary server and the client is maintained and the primary server's local client information including authentication is retained, the client must have valid tickets and permission for a flush server operation. The function returns TRUE only if the entire operation was successful.

```
int MUD_FlushServer (mud)
        MUD mud;
```

mud Server connection returned by MUD_OpenServer().

return value Returns TRUE if a flush was successfully conducted. Returns FALSE any portion of the operation failed.

In addition to resetting the server, the client can also get information about the server. MUD_GetServerStats() uploads a portion of the current statistics from the primary server. The degree of detail will depend on the client's level of authentication (system administrators will permit global, detailed information while a standard user will only be able to get information about his connection). Once again, the return value is TRUE if successful.

```
int MUD_GetServerStats (mud, stats)
        MUD mud;
        Statistics stats;
```

mud Server connection returned by MUD_OpenServer().

stats Statistics structure returned to the client. This structure must be pre-allocated.

The next utility function MUD_Compress() which will take a given database and remove all of the deleted records, returning to the user a compressed, compact database. Unfortunately, because the MUD implementation uses record-shadowing, the compress routines become a little more complicated. The result is a modified compression routine which first determines if a given deleted record is actually being used by a previous record. The precise record length can be determined from the record size and repetition count stored in the database cache. After determining a record's length then the compression can skip to the first start after this point and begin normal processing again.

Chapter 7: Conclusions

MUD seems to solve several significant problems with available multi-user, networked database services: real-time interaction, configurable databases, distributed services, centralized administration and secure transactions.

Real-Time Interaction - Uses a network protocol layered on top of an ISAM relational database to provide rapid access to large, remote databases.

Configurable Databases - Uses templates either read from the database or downloaded by the client to describe the data organization.

Distributed Services - Permits multiple database servers to control separate segments of the database to improve speed and availability.

Centralized Administration - Stores information in standard MUD database format so it can be controlled through a standard MUD client.

Secure Transactions - Uses the Kerberos authentication service to aid in mutual authentication and provide encryption keys for internal communication.

Chapter 8: Future Work

MUD is by no means complete either in design or implementation. Multi-server support has not been thought out with enough consideration. In addition, authentication in general can use additional refinement. Most importantly, however, the implementation is essentially non-existent.

Multiple Server Support

The design for a MUD server does not directly address using and implementing multiple servers for the database storage. While the interface could permit the client to open multiple server connections, allowing the database server to make the remote server connections seems preferable. There seem to be two different approaches to the problem: using server-server or server-client protocol.

The server-client protocol communicates between servers using the same protocol as a client would use when communicating with a server. The requesting server would make its requests and receive its information through the client library functions. This technique makes it easier to implement the server-server interface, but restricts the server to the client's limited requests. More importantly, it requires the server to ask the its own client for a new set of Kerberos tickets for the remote server.

The server-server protocol establishes a different protocol for interserver communications. The major disadvantage when compared with the client-server communication lies in implementation. Fortunately, the implementation can be minimized by modifying the client-server code for the new formats. The advantages lie in authentication and survivability. By using a separate communications protocol, the local server and the remote server can authenticate each other via Kerberos without asking a client for Kerberos tickets. In addition, the protocol can be tailored to different specs. For example, servers will rarely want to execute remote storage requests but will often want to access query results.

User Authentication

In addition to implementing the server, user level authentication and protection needs to be implemented. The proposed technique is to make the MUD server a secure client using the Kerberos authentication system. The requirements for this become rather hazy at the moment because Project Athena is not utilizing the most recent version of the Kerberos remote authentication service. The problem is that a special ticket, called a ticket granting ticket or TGT, must be issued to the user and this is not currently supported. The TGT is needed so that the client can generate valid Kerberos tickets for the user to use on other machines. Without these remote tickets, the databases must all be world-readable and server-writable in order to work. Essentially, the users all map to the same database user, eliminating all but the internal database protection.

Implementation

Due to time constraints imposed upon the author, the MUD system was not actually implemented. The implementation naturally segments itself into three major categories: database server, C-Tree interface, and client library. The advantage to this particular division is that the parts can (and should) be implemented in parallel to result in a working MUD before February.

Developing a reasonably robust client library is the simpler of the three elements of the implementation. Since the client library is rather small, concise, and highly abstracted, it is essentially independent from the server-side implementation. Based on Daniel Applebaum's implementation of Galatea, the client library should be implemented in less than one month and remain essentially static throughout server development.

Implementing a secure, robust MUD server is not a simple, straightforward task. Based on Galatea, server development will be an ongoing evolution. Most of the initial effort should be placed in the load, store, and search functionality with authentication and error recovery coming later. Fortunately, the server should be usable within a few months and fleshed out in less than one year.

The most difficult portion of the server-side implementation lies in the C-Tree interface. While C-Tree is a very fast relational database, programming with it can be rather gruesome and obtuse at times. Fortunately, Jeff Johnson has spent the last year learning how to use C-Tree rapidly, effectively, and efficiently.[Johnson, 1989] Given Johnson's past work as a starting point, the relational database interface should be a difficult but doable segment of the project. It seems reasonable to allocate one month for an initial implementation with refinements occurring throughout the rest of the server development.

All in all, the server implementation should be completed in three man-terms. One programmer developing the database server, one the C-Tree interface, and one the client library and sample application (VEdit). A final implementation, however, will not be forthcoming for at least one year.

Appendix A: VEdit Database Objects

Before embarking on the description of the objects themselves, let us define a few common object fields. These structures are not actual objects themselves but are contained within the objects.

The first two, clip and switch, are used to define edits. A clip is a segment of source material while a switch describes how to go from one clip to another.[20]

Clip:

Initial Frame:	unsigned long
Source:	Tuple
InFrame:	unsigned long
OutFrame:	unsigned long
Speed:	signed long

Switch:

Frame:	unsigned long
InputClip:	unsigned long
OutputClip:	unsigned long
Duration:	unsigned long
Switch Type:	StateArray

The field sub-type is used to specify the consistes of a field name, type, and set of flags. It describes a field within a record as defined in the object's template.

Field:

Name:	char31
Record Type:	unsigned long
Flags:	StateArray

The file reference is used during file replacement to eliminate a textual description from the record and replace it with a file handle. The handle consists

[20]A complete description of the Galatea Seamless Extension including clips and swiches can be found in [Applebaum, 1989].

of a hesiod file system name that can bew resolved by the attach command and a path to look at once the remote filesystem is attached.

FileReference:

Realm:	Tuple
FileSystem:	char31
Path:	char127

The tuple is the generic database handle: a unique identifier used to find a specific record anywhere in the database. The first field, the realm, denotes what database hierarchy the record is in which allows separate hierarchies to co-exist. The database key specifies which database within the realm to look at. The object key further narrows the record's location to a specific object type. The final field, the record key, contains the specific key in the object database.

Tuple:

Realm Key:	unsigned long
Database Key:	unsigned long
Object Key:	unsigned long
Record Key:	unsigned long

Computer

The computer record contains information describing an individual machine and is used to maintain information about hosts used by the MUD server.

Computer:

key:	unsigned long
Name:	char31
Address:	unsigned long
Icon:	Tuple
Owner:	Tuple
Location:	Tuple
Description:	char63
numPrimDatabases:	unsigned long
<PrimDatabases>:	Tuple
numSecDatabases:	unsigned long
<SecDatabases>:	Tuple

Database

The database is an object which describes a database, its configuration, protection, location, and appearance. Typically there are multiple copies of this record: one within a given database and one located in the database database so that the database can be easily located.

Database:

key:	unsigned long
Name:	char127
Parent:	Tuple
Icon:	Tuple
State:	StateArray
numGroups:	unsigned long
<Groups>:	Tuple
<Access>:	StateArray
numLocations:	unsigned long
<Location>:	File Reference
<Log File>:	File Reference

Edit

The edit is the atomic list management unit used by VEdit. It contains the physical information about the shot or squence of shots enabling it to be played back. Specifically it contains the physical information (shot), ancestral information (parent), and EDL management information (clips and switches).

Edit:

key:	unsigned long
Shot:	Tuple
Parent:	Tuple
Mode:	StateArray
NumberClips:	unsigned long
<Clips>:	Clip
Number Switches:	unsigned long
<Switches>:	Switch

Employer

An employer record holds all of the necessary information pertaining to an employer. In addition to the name, address, telephone number and owner, the record contains all of the employees and the represenative icon.

Employer:

key:	unsigned long
Name:	char31
Type:	Tuple
Icon:	Tuple
Owner:	Tuple
Location:	Tuple
Description:	FileReference

Group or ACL

The group or ACL (Access Control List) record is a variable length record containing not only the name for a group but a list of all of its members. It also contains a textual description and a representative icon.

Group:

key:	unsigned long
Name:	char31
Icon:	Tuple
Description:	char127
Number Users:	unsigned long
<Users>:	Tuple

Icon

The icon record contains all of the information needed to form a video icon: location, frame, and size. It also contains the input region for the icon: the area of the frame to digitize to form the icon.

Icon:

key:	unsigned long
Icon Width:	signed long
Icon Height:	signed long
Source Media:	Tuple
Frame:	unsigned long
Video X:	unsigned long
Video Y:	unsigned long
Video Width:	unsigned long
Video Height:	unsigned long

Item

The item record is the generic record containing a name, type, icon, and decription for a physical object like a car.

Item:

key:	unsigned long
Name:	char31
Type:	Tuple
Icon:	Tuple
Description:	FileReference

Log

A log record is used to record database transactions by both the primary and database servers. It uses a modified form of file replacement to optimize storage: a specific file reference is not made, instead the offset is used in searching a journal file for the message text. The log file name is recorded in the database structure.

Log:

key:	unsigned long
type:	unsigned long
Client:	Tuple
Logging Host:	Tuple
offset:	unsigned long
Message:	char 256

Palette

A palette is a variable length record contains the information necessary to construct a palette full of icons. The first part of the record includes the palette name, icon, size, location, and description. The second half of the record contains the list of all of the objects contained in the palette and where they should be positioned.

Palette:

key:	unsigned long
Name:	char127
Icon:	Tuple
Palette X:	signed long
Palette Y:	signed long
Palette Width:	unsigned long
Palette Height:	unsigned long
State:	StateArray
Description:	FileReference
Number Elements:	unsigned long
<x>:	unsigned long
<y>:	unsigned long
<Elements>:	Tuple

Person

A person, or character, record is used to store information about a character in the movie database. It is very similar to the employer record, containing additional fields for job type and employer. Essentially, it is composed of the person's name, address, phone number, job, employer, and description along with the icon representation.

Person:

key:	unsigned long
Name:	char31
Address:	char31, char31
Zip Code:	unsigned long
Telephone:	unsigned long
Icon:	Tuple
Job Type:	Tuple
Employer:	Tuple
Description:	FileReference

Place

The place record contains important information about a physical location: namem address, phone number and description. In addition it contains a reference icon and the type, or category, of building.

Place:

key:	unsigned long
Name:	char31
Address:	char31, char31
Zip Code:	unsigned long
Telephone:	unsigned long
Type:	Tuple
Icon:	Tuple
Description:	FileReference

Realm

The realm object goes in conjunction with the user, group, and database objects. As the user object describes an individual user within a realm, the realm object describes a realm within the network. It is used to seperate the various Kerberos realms from one another because database, user, and group ids may be replicated amongst them. For example, there may be a user at Carnegie-Mellon University with the same username as a different user at MIT.

realm:

key:	unsigned long
Name:	char31
Icon:	Tuple
Net Suffix:	char31
NumberServers:	unsigned long
<ServerName>:	char31
<ServerIPAddress>:	unsigned long

Shot

A shot is a variable length record that contains all of the physical information about a given edit. The physical information ranges from date, time, camera information and location of the shot to the actors in the shot, description, and initial cut.

Shot:

key:	unsigned long
Name:	char31
Icon:	Tuple
Date:	unsigned long
Time:	unsigned long
Place:	Tuple
Number Actors:	unsigned long
<Actors>:	Tuple
Camera Info:	StateArray
Initial Edit:	Tuple
Description:	FileReference

Source

The source record contains information regarding a piece of source material. The record includes not only the name, icon and description but also the source type (audio, videodisc, videotape, film, uncontrolled) and locations that the source may be found.

Source:

key:	unsigned long
Name:	char31
Icon:	Tuple
Source Type:	unsigned long
Description:	FileReference
Number Locations:	unsigned long
<Locations>:	Tuple

Template

The template contains enough descriptive information to format a given object. It is composed of the template name, icon, and object class in addition to the descriptions for each entry within an object record.

Template:

key:	unsigned long
Name:	char31
Icon:	Tuple
Object Class:	unsigned long
Number Fields:	unsigned long
<Fields>:	Field

Type

The type record is used as a generic text lookup and is used typically as a cross reference to avoid a string match. It simply contains a string in addition to the required four byte identifier.

Type:

key:	unsigned long
Name:	char 31

User

The final database object is the user: a record containing all of the personal information associated to a user. In addition to the user's first and last names, it also contains some default group information and representative icon. The remaining two fields. username and uid, uniquely specify the user to the operating system.

User:

key:	unsigned long
Username:	char8
Last Name:	char15
First Name:	char15
uid:	unsigned short
Default UNIX group:	unsigned short
DefaultGroup:	Tuple
Icon:	Tuple

Appendix B: Glossary of Technical Terms

Access Control List: A group and a list containing users who are members of the group. This is very similar to a UNIX group.

Client: A process that requests a service from a server in the course of performing some functions of its own. Often the client is acting on behalf of a user. [Transarc, 1989]

Cluster: A group of workstations located in the same room.

Database Server: The server that oversees a particular portion of the database.

Database Tuple: A reference to a specific element in the entire database. The tuple consists of the database id, object id, and record id.

Edit: The VEdit term for a "shot." Encompasses everything from a simple butt-cut to an EDL containing several hundred separate shots.

Ethernet: The network which links computers together within a small area. For example, all machines in a cluster are on the same loop of ethernet but the clusters are not connected together with ethernet.

File System: A directory tree which can be attached to remote workstations. For example, the filesystem "hkbirke" is the author's home directory structure.

Hesiod: A name service that allows an application to retrieve associations between a name, particular service type, and information about that named service. [Dyer, 87]

ISAM: Indexed Sequential Access Method. This is an old IBM relational database term that has lost its specific meaning. Broadly, it describes a class of relational databases which can be accessed both sequentially and through an index.

Kerberos: A remote authentication service designed, implemented and used by MIT Project Athena. It permits processes to verify the identity of the remote service.

Key: The unique number given to a Kerberos principal by the Kerberos ticket granting server. This number is used to uniquely identify the client.

Mutual Authentication: The process by which two separate processes verify that the other process is who it claims to be.

Network: The pathway over which information is transmitted from one computer to amother. The physical ethernet cable, connectors, gateways, bridges, and other associated hardware associated with the network.

Palette: VEdit's general purpose cataloguing device, similar to a directory for UNIX or a bookshelf for a library.

Primary Server: The server that controls and maintains the MUD environment.

Record Shadowing: Representing a variable length record as a fixed length record by storing the remaining elements in successive deleted records.

Server: Aprocess which performs a related set of services such as fetching files, maunipulating volumes, providing authentication, and distributing database service. [Transarc, 1989]

SQL (Structured Query Language): SQL is both a description for a database interface and a class of databases. As an interface, it is a flexible, powerful, high-level interface specification. As a class, it is composed of all databases that use the SQL query language. Ingres and Sybase are SQL databases.

Template: A list of field names, types, and states for a record type.

VEdit: VEdit is the integrated video editing system written by Hal Birkeland and Mike Kobb of the Film/Video Section of the MIT Media Lab. It serves as the sample application for the database service.

Workstation: A publicly available, single-user machine similar to a personal computer.

References

Applebaum, D. "The Galatea Network Video Device Control System," short paper, MIT Media Laboratory, 1989.

Beauchamp, Donovan C. *A Database Representation of Motion Picture Material,* S.B. Thesis in Electrical Engineering and Computer Science, MIT , 1987.

Berkenbilt, E. Jay "rkinit," MIT Project Athena, 1989.

Birkeland, Halvard K. *VEdit Users Manual,,* Film/Video Section, MIT Media Laboratory, 1989.

Davenport, G. "New Orleans in Transition, 1983-1986: The Interactive Delivery of a Cinematic Case Study," Film/Video Section. MIT Media Laboratory. Paper presented to the International Congress for Design Planning and Theory, Boston, August, 1987.

Davenport, G. "Software Considerations for MultiMedia Video Projects," Film/Video Section, MIT Media Laboratory. Paper presented to the X11 Video Extension Technical Meeting at Bellcore, June 1988.

Davenport, G., Birkeland H., and Kobb, M. Conversation, 15 September 1989.

DellaFerra, C. *et al Section E.4.1: Zephyr Notification Service,* MIT Project Athena, 1989.

Dyer, S. "The Hesiod Name Server," MIT Project Athena, 1989.

Dyer, S. "Hesiod Name Service Application Programmer's Guide." MIT Project Athena, 1987.

FairCom *c-tree File Handler Programmer's Reference Guide.*, 1988.

Johnson, J. Research Progress Report, Film/Video Section, MIT Media Laboratory, 1989.

Kobb, Michael J. Research Progress Report, Film/Video Section, MIT Media Laboratory, 1989.

Kohl, John T. *A Database System for the MIT Residence/Orientation Program,* SB Thesis in Electrical Engineering and Computer Science, MIT, 1988.

Leffler, S. *et al An Advanced 4.3BSD Interprocess Communication Tutorial.* Computer Systems Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

Levine, P. *et al Section E.1: Moira, the Athena Service Management System,* MIT Project Athena, 1989.

Mackay, W. and Davenport, G. "Virtual Video Editing in Interactive Multi-Media Applications,"*Communications of the ACM,* July 1989.

Mendelsohn, N. "GDB C Library Reference Manual," MIT Project Athena, 1986a.

Mendelsohn, N. "A Guide to Using GDB," MIT Project Athena, 1986b.

Miller, S., Neuman, B., Schiller, J., and Saltzer, J. *Section E.2.1: Kerberos Authentication and Authorization System,* MIT Project Athena, 1988.

Pepe, Louis J. *A Digitial Icon Representation for Movie Editing and Analysis,* S.B. Thesis in Electrical Engineering and Computer Science, MIT, 1988.

Relational Technology Inc. *Ingres/QUEL Self-Instruction Guide,* 1985.

Rochkind, Marc J. *Advanced UNIX Programming* , Prentice-Hall, Inc. Englewood Cliff, NJ, 1985.

Rosenstein, M., Geer, D. and Levine, P. "The Athena Service Maintenance System," MIT Project Athena, 1989.

Rubin, Benjamin *Constraint-Based Cinematic Editing,* S.M. Thesis in Visual Science, MIT, 1989a.

Rubin, B. and Davenport, G. "Structured Content Modeling for Cinematic Information," SIGCHI conference proceedings, 1989b.

Sechrest, Stuart *An Introductory 4.3BSD Interprocess Communication Tutorial* Computer Science Research Group, Department of Electrical Engineering and Computer Science, University of California, Berkeley, 1986.

Steiner, J., Neuman, C. and Schiller, J. "Kerberos: An Authentication Service for Open Network Systems" in *Usenix Conference Proceedings,* Dallas, Texas, 1988a.

Steiner, J. and Geer, D. "Network Services in the Athena Environment," MIT Project Athena, 1988b.

Sybase, "1.0 Release Notes: Sybase SQL Server and DB-Lib" as included in NeXT 1.0, 1989.

Transarc Corporation *An Overview of the Andrew File System.* Pittsburgh, 1989.