Discuss: An Electronic Conferencing System for a Distributed Computing Environment[1]

Ken Raeburn
Project Athena
Massachusetts Institute of Technology
Cambridge, MA 02139
raeburn@athena.mit.edu

Jon Rochlis
Telecommunications Systems
Massachusetts Institute of Technology
Cambridge, MA 02139
jon@bitsy.mit.edu

William Sommerfeld
Apollo Computer, Inc.
Chelmsford, MA 01824
wesommer@athena.mit.edu

Stan Zanarotti
Laboratory for Computer Science[2]
Massachusetts Institute of Technology
Cambridge, MA 02139
srz@lcs.mit.edu

Abstract

As computers and computer networks become commonplace, electronic communication is rising in importance and utility. The challenge is to take a large, distributed computing environment and build a system which allows its users to communicate effectively and efficiently with each other. This paper compares and contrasts several common types of electronic communication, focusing on electronic conferencing. We describe the implementation of such a system, *Discuss*, for the computing environment found at MIT. Issues covered include the basic model of an electronic meeting, the currently implemented user interfaces, separation of user interface from underlying operations, splitting the workload between client and server, communications issues created by the heterogeneous environment, authentication and authorization, notification, and unification of the numerous subordinate UNIX[3] libraries into a coherent whole. The paper next summarizes the *Discuss* system's current usage, then closes with a discussion of its possible future development.

---

[1] To be presented at the Winter 1989 USENIX.

[2] This project was supported in part by the Defense Advanced Research Projects Agency of the Department of Defense and monitored by the Office of Naval Research under contract number N00014-83-K-012.

[3] UNIX is a trademark of AT&T.

Introduction

One of the most useful purposes computers and computer networks serve is helping people communicate. More and more experience is being gained in this area of computer use. While such communication has some problems (and certainly differences) when compared with traditional oral and written communication, it is not premature to claim success. At the same time, it is important to look at different types of electronic communication and the needs they meet (or fail to meet) with an eye toward possible improvements.

This paper presents the design choices made and implementation lessons learned during the development and initial deployment of *Discuss*, a computer conferencing system developed by MIT's Student Information Processing Board (SIPB)[4]. *Discuss* was designed for a heterogeneous distributed computer environment, such as that of MIT's Project Athena, and we pay particular attention to those issues which are important in such an environment.

Discuss has been in operation since 1986 and is currently used by more than 220 people at MIT. More than 100 *Discuss meetings* exist on eight server machines. Users and meetings exist in two separate administrative domains, Project Athena and the Laboratory for Computer Science (LCS).

This paper is organized as follows. We begin in Section 1 by describing the motivation for the *Discuss* project. This includes a description of the various types of electronic communication, an explanation of how conferencing differs from electronic mail and bulletin boards, and an overview of the MIT computing environment for which our system was developed. In Section 2 we define the model of a *meeting* which is the foundation of *Discuss*. Section 3 describes the currently implemented interfaces which allow a user to access meetings. The client/server model is explored in Section 4, and the communication issues caused by this split in duties are covered in Section 5. Following that, Section 6 describes the topic of authentication and authorization, showing how *Discuss* uses *Kerberos*[17] to provide authentication, but develops its own application level authorization scheme which we believe is required by a distributed conferencing system. User notification of new transactions via the *Zephyr*[4] notification system is covered in Section 7. We talk in Section 8 about the lessons learned, and conclude with some discussion of the future of *Discuss* in Section 9.

[4]The SIPB is a volunteer organization dedicated to improving the student computing environment at MIT.

1 Motivation

Excluding telephones, fax machines, real-time video, voice conferencing systems and the like, the most common forms of electronic communication are electronic mail, interactive messages, bulletin boards, and conferencing systems. All are useful for decreasing the cost of face-to-face (or voice-to-voice) communication, but each has different strengths. In this section we will compare these forms of electronic communication, focusing on the advantages of conferencing. We will also describe the distributed computing environment at MIT into which any conferencing system must fit.

1.1 Types of Electronic Communication

Electronic mail is the most widely known and accepted method of electronic communication. Using the well understood metaphor of the postal service, even novice users can pick up the basic concepts without much effort. E-mail (as it is known) is generally batch oriented, and as in the postal system mail flows from a user to one or more recipients. Each user has some mechanism for retrieving his new mail (checking his mailbox) and for posting his mail for later delivery (dropping it in the outgoing mailbox). Mailing lists exist and facilitate communication with large numbers of people interested in the same topic. Each member of the list receives a separate copy of every message sent to that list. Due to variations in queuing delays, replies sent to recipients on different network nodes may sometimes arrive several days before or after the original message, causing much confusion.

Interactive messages have proven quite useful in a different way. Unlike e-mail, they are used only when the sender and recipient are both logged in. Instead of being queued for delivery whenever the mail system gets around to it, interactive mesages are usually delivered at once and displayed immediately on the receiver's screen. Interactive messages are useful for short, real-time queries ("When do you want to leave for the movie?" "Do you know where the source for discuss is?").

Electronic bulletin boards (or "bboards") allow users to "post" messages in a place which

other users may peruse at a later date. Just as with real-world bulletin boards, bboards generally are restricted to specific topics and have large numbers of readers, frequently in the hundreds for dial-in microcomputer based systems, or thousands for the UUCP-based *netnews*[10]. The readers of an electronic bulletin board may be scattered throughout a state, country, or even the world, since, once posted, a message may be transmitted to other systems which store local copies for their users. These systems may also choose to forward the message to yet another computer. Messages generally expire after a few days or weeks, depending upon the disk space resources of the system or systems involved.

Electronic conferencing is subtly different from bboards; it is designed for a smaller number of users, frequently with privacy concerns, and with long term archival requirements. Not only do conferencing systems allow users to communicate with others interested in particular topics, but they prevent unauthorized users from reading such communication. For example, the professors and teaching assistants running a course might want to discuss possible quiz questions, but be confident that students would not be able to read their conversations (at least until after the quiz!). Also, there is frequently a requirement to retain the discussion for a long period of time so that people joining the organization at a later date may come up to speed by reading "what has gone before." While conferencing systems may have bboard-like facilities to periodically purge older entries, they are easily used for archival purposes and have more advanced mechanisms for sorting through previous entries than are usually found in bboard or e-mail systems.

In contrast to e-mail, the entries in a conferencing system are stored in one place so that the user sending a message need not know all the people interested in his message. Those interested will go to the message, rather than waiting for the message to come to them. Most importantly, this fulfills the goal of providing easily accessible archives; people can later decide they are interested in the message and access it as they would any other message in the conferencing system, rather than having to hunt around for possibly non-existent saved mail archives.

Since new messages in a conferencing system are immediately and globally visible, such a system can actually support running conversations with many participants. This is quite useful for problem-resolution meetings, in which many people may be able to fix a broken piece of equipment or answer a user's question in real-time. A dialog may be required and an archival record of the converation is needed to ensure proper resolution or simply to bring late-arriving helpers up to date on the current status of the problem.

Conferencing systems also help users avoid clutter in their personal electronic mailboxes. This is a natural result of conferences being based on specific topics. Of course, vague or overly general meeting topics may reduce this gain, underscoring the importance of the role of a moderator, or *chairman*, to ensure that conversations stay on track, perhaps by suggesting that further discussion take place in a different meeting or by creating a new meeting if one with suitable topic does not already exist.

1.2 The Distributed Services Environment

MIT's Project Athena[1][20] is an example of a computer utility based on the distributed services model of computation. Athena has approximately 850 4.3BSD UNIX workstations (of several different hardware types) scattered around campus, in locations ranging from public terminal rooms to private offices to living groups. Workstations and servers are connected by a high speed network. Each workstation has a moderate-sized hard disk used for storing essential software and for swap space. Most system software and user files are obtained from remote file servers. The location of both the system software and a user's filesystem are determined by a network name service called *Hesiod*[6]. The network authentication service, *Kerberos*, is essential because the network is unsecured and a public workstation is under total control of its current user. (The root password is even published![5])

Electronic mail service is provided by a central mailhub, which is responsible for re-writing the headers in order to present a unified address-space and for distributing messages for internal users to a post office server. Users can retrieve their mail from their post office server, located using *Hesiod*, via the Post Office Protocol modified to use *Kerberos* authentication. The *Zephyr* notification system provides delivery of interactive messages from users to other users or from service-providing processes to users requesting notification. *Zephyr* also makes use of *Hesiod* to

[5] For the curious, the root password is "mrroot."

find its servers and *Kerberos* to provide authentication of messages.

1.3 Enter *Discuss*

The SIPB undertook the task of developing an electronic conferencing system for this environment in 1986. Much of *Discuss* came out of the SIPB's experience with the Multics conferencing system *forum*[9] and a desire to see it extended beyond a single machine environment[13]. The part-time efforts of Ken Raeburn, and Bill Sommerfeld, and Stan Zanarotti have resulted in a solid production-level system, which has been adopted for use by Athena's System Development and User Services Groups, and portions of MIT's Laboratory for Computer Science (LCS). SIPB, Athena, and LCS run *Discuss* as a network service for themselves and the MIT community.

2 *Discuss* Meetings

The *meeting model* is the central concept in *Discuss*, as is the case with Multics *forum*. Users "attend" a set of electronic meetings, just as you might attend a regular set of face-to-face meetings. For each meeting attended you listen to what other participants have to say, and you can contribute as you wish. However, unlike face to face meetings, not all the participants in a *Discuss* meeting need be present at the same time. Participants need not be geographically close, so long as there is a network connection between the user and the machine on which the meeting is stored. Removing the time and space constraints present in face-to-face meetings is one of the major advantages of electronic conferencing.

A meeting is a set of *transactions*. Each transaction is labeled with the author (unless the author wishes to remain anonymous), a subject, the time of entry, and a unique transaction number. The contents of a transaction are stored as uninterpreted eight bit data, though the existing client user interfaces are only prepared to receive ASCII text. *Discuss* keeps track of which transactions a user has seen, and has the ability to only display transactions which are new to the user.

Transactions about the same subject are grouped in subconversations called *chains*, and tools are provided for moving through such chains. When a user replies to a given transaction, the subject of the reply is created by prepending "Re:" to the subject of the original transaction; but, more importantly, a pointer is associated with the new transaction which indicates which transaction is the original. The original transaction also has a forward pointer to the reply. In this manner a chain of transactions can be created and easily walked through. If one replies to a transaction in the middle of a chain, *discuss* simplifies the pointers by pretending that you have replied to the transaction at the end of the chain. Each transaction in a chain has a previous reference (`pref`) and a next reference (`nref`), except for the first (`fref`) and last (`lref`) transactions in the chain, which only have an `nref` or `pref`, respectively.

Transaction numbers are assigned in a strict order, unlike the *notes*[7] conferencing system, and unlike the *netnews* bboard system they are consistent for all readers of the meeting. Replies are simply given the next transaction number and threaded to produce a chain as described above. If transactions are deleted, the transaction number namespace is left unchanged. Users can refer to a transaction by number with the confidence that the transaction number will not change in the future.

Figure 1 shows an example listing of transactions. Transaction 4 has been deleted. There are two chains, one starting at transaction 2, and the other at transaction 5. The "error handling" chain has an `fref` of transaction 2 and an `lref` of transaction 6. All the transactions in the chain have `pref` and `nref` pointers as shown in Figure 2.

Entering a message in more than one meeting while maintaining the ability of the user to only see the message once is not supported. Unlike other conferencing systems, like Multics *forum*, you cannot tell who attends a meeting (unless they speak up, of course) nor who has read a given transaction.

3 User Interface

The original interface was selected for its simplicity and, for some of us, familiarity. It is a terminal-oriented, shell-like interface which loops through a prompt-input-execute sequence.

The user is presented with the idea of a meeting being attended, and a "current" transaction within that meeting; he may go to a different meeting with the `goto` command, or run one of many commands within a meeting. The com-

```
[0001]*  (2) 10/29/87 12:43 raeburn         Reason for this meeting
[0002]  (33) 10/29/87 17:06 raeburn         error handling
[0003]   (2) 10/29/87 17:22 srz@LCS.MIT.EDU Re: error handling
[0005]  (16) 10/30/87 11:06 srz@LCS.MIT.EDU send_to_kdc and use of resolver
[0006]  (20) 10/30/87 11:41 jon             Re: error handling
[0007]   (5) 11/01/87 23:10 jis             Re: send_to_kdc and use of resolver
```

Figure 1: Sample listing of transactions in a *Discuss* meeting

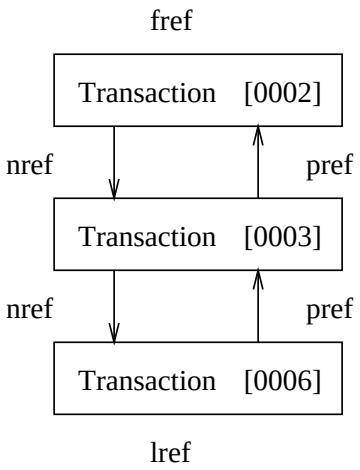

Figure 2: Transactions form chains.

mands available for examining the contents of a meeting include `list` and `print`; each command takes one or more transaction numbers as arguments. Transaction numbers may be specified as integers or ranges of integers, or symbolically, such as `current`, `pref`, or `first`.

Naturally, some method for entering transactions is needed. We provide the `talk` command for entering transactions to start new chains, or to stand by themselves, and `reply` to enter a response to an existing transaction. The `talk` command requests a subject, then allows the user to enter the text of his message. It uses a simple line-input interface, terminating the message on input of a line containing only a '.' or escaping to run an editor program specified by the environment variable "EDITOR" on `~e`. Although the protocol does permit new subjects for any transaction, the `reply` command currently assumes a subject derived from the message being replied to, prefixed with `Re:` if it is not already present. A `delete` command is also provided.

There are also commands for dealing with several meetings. Aside from the `goto` command mentioned above, we have provided the `check_meetings` command, which checks whether any meetings the user attends have changed since the user last attended them. The command `next_meeting` can be used to step through this list meeting by meeting, allowing the user to view new transactions in all meetings he attends.

4 Client/Server Model

One of the goals of *Discuss* is that it is distributed. Unlike its timesharing-based relatives, *Discuss* allows meetings to be shared among users working on different machines. The design of *Discuss* addresses the problems of distributing its functions across different machines.

The MIT computing environment, as described in Section 1.2, influenced the design of *Discuss*. Almost all MIT computers are connected to a set of high-speed networks and gateways, allowing fast data transfer between any two machines on campus. In addition, most students use public Athena workstations, which are single-user UNIX machines with graphic displays. These workstations have spare CPU time, but retain no local state. MIT computers form a loose federation of autonomous systems. In particular, MIT has no common, network-wide file system that could store all *Discuss* meetings. This differs from the Carnegie Mellon bboard system[2] which uses the Andrew File System to distribute meetings among many machines.

Given the MIT environment, *Discuss* uses the client/server model of computation. With this model, an application is divided into two parts. One part, the server, manages a resource, while another part, the client, accesses this resource across the network by contacting the server. The client/server model is a common model for other applications in the Athena environment; it encapsulates the particular resource within the servers.

4.1 The *Discuss* Server

With *Discuss*, the particular resource is the set of meetings on a given machine. A server provides access to these meetings, allowing clients to manipulate them in a controlled fashion. The pool of *Discuss* meetings is distributed among various servers; unlike a *netnews* group, a *Discuss* meeting is located on a single machine. This allows all participants to have a consistent view of the meeting, but requires low-latency network connectivity. The transactions are numbered consistently, and users can immediately see new transactions entered by other participants.

A *Discuss* server provides a low-level interface for manipulating meetings. The operations are fairly primitive, allowing clients to get information about single transactions and single meetings. The server simply manages the meeting data, ensuring its consistency and controlling access to it. The server provides a core set of operations that can be used by any user interface. A sample set of these primitives is shown in Figure 3.

On a UNIX *Discuss* server, a meeting is stored as a directory owned by the `discuss` user on the server machine. The name of the meeting is the name of this directory; `/usr/spool/discuss/ping` is a sample *Discuss* meeting name. The meeting can be located anywhere within the server machine's filesystem. This allows users with accounts on a server machine to create *Discuss* meetings under their home directories. A *Discuss* meeting contains three files: `transactions`, `control`, and `acl`. The `transactions` file contains the text of the transactions, stored in an append-only format. The `control` file contains indexes into the `transactions` file, as well as information about transaction chains and deleted transactions. The `acl` file contains the access control list for the meeting. Access control is described later in this paper.

Discuss meetings should be placed on machines that have sufficient amount of disk space to support the meetings. Since *Discuss* meetings are archival in nature, this disk space should be backed up regularly[6]. *Discuss* server machines should also have enough computing power to support the expected number of clients.

[6] A lesson learned the hard way.

4.2 The *Discuss* Client

A user accesses *Discuss* meetings by running one of the various *Discuss* client programs. This program manages the user interface and contacts the appropriate servers to manipulate the meetings that the user attends. The client program displays the information contained in a meeting and allows the user to enter new transactions.

Since the client/server interface is low-level, it does not dictate a particular user interface; different clients can present different user interfaces, provided that they fit into the meeting model explained above. Currently, there are five different *Discuss* clients: `discuss`, `xdsc`, a `Common Ground` implementation[8], `dsmail` and an emacs[16]-based client. `discuss` is the general command interface described above. `xdsc` is an experimental client based on the X window system[15]. `dsmail` is a utility program that automatically enters mail messages into *Discuss* meetings, trimming the headers appropriately.

In the client/server interface, a *Discuss* meeting is named by its server machine name and its location on the server machine. Since these names are difficult for humans to use, the client program accepts simple mnemonic names for meetings, and converts them into this low-level form. This mapping could be provided by a name service such as *Hesiod*, but none existed at the time *Discuss* was initially designed. Instead, a user file called `.meetings`, in the user's home directory, is used. This ASCII text file records what meetings the user attends, storing its low-level name and the simple names that can be used to refer to the meeting. This allows users to have personal aliases for the meetings they attend. The `.meetings` file also contains information about what transactions the user has seen in a given meeting. This information is kept on the client side because of a desire to keep the server simple, and also because of concerns that such information shall be private.

5 Communications

To communicate between clients and servers through the network, *Discuss* defines a protocol that allows clients to perform different *Discuss* operations on servers. Like other systems, *Discuss* uses several layers of network protocols. These protocols are shown in Figure 4. We give a brief description of each protocol and why it was chosen. We start with the highest-level protocol,

```
get_mtg_info (mtg_name) returns (mtg_info, err_code)
get_trn_info (mtg_name, trn_num) returns (trn_info, err_code)
get_trn (mtg_name, trn_num) returns (trn_text, err_code)
add_trn (mtg_name, trn_text, subject, reply_trn_num)
        returns(new_trn_num, err_code)
```

Figure 3: Sample set of *Discuss* server operations

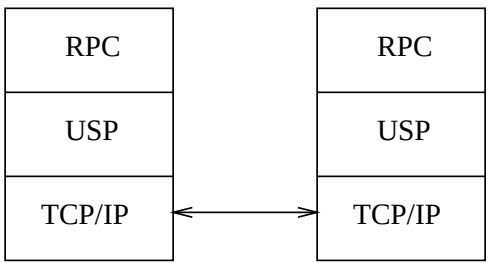

Figure 4: Protocol layering in *Discuss*

and proceed down through the lower levels.

RPC A *Discuss* server provides a set of operations that clients can perform on meetings. These operations are presented within a Remote Procedure Call (RPC) mechanism. RPC is a well-understood programming paradigm which allows the client-server interface to look like a programming interface. The exact RPC mechanism that *Discuss* uses is home-brew, because the recent RPC mechanisms, such as Apollo NCS[5] and Sun RPC[19], were not available when *Discuss* was first built. One nice feature of the *Discuss* RPC mechanism is that it is designed so that the client and the server can be linked together into a single program. This feature is useful when debugging or writing utilities that need to run quickly.

USP The Remote Procedure Call mechanism uses Unified Stream Protocol (USP) [3] to transmit data values for the RPC. USP was an early protocol to transmit integers, strings, and other information in a machine-independent way. USP solves the problems of byte-ordering and how strings should be transmitted over a stream. These days, newer protocols, such as Sun XDR[18] or ISO ASN.1[11], would be used to transmit these values.

TCP/IP USP is built on top of TCP/IP, the transport layer of the Internet protocol suite. TCP/IP provides a high-speed, reliable stream between any two hosts connected on the Internet. TCP/IP is a popular protocol, with implementations for machines ranging from mainframes to microcomputers. This allows *Discuss* to be used on different operating systems, such as UNIX and Multics.

Although the current implementation of *Discuss* uses these protocols to communicate between clients and servers, *Discuss* could use any other protocol that provides the same functionality. *Discuss* clients and servers only deal with a set of meeting operations, which are available as ordinary procedure calls. *Discuss* could be implemented on other RPC mechanisms, such as Sun RPC, Apollo NCS, or the Mercury RPC[12], without much of a problem. Alternatively, the operations could be encoded in ASCII on a standard TCP stream, as SMTP[14] is.

6 Authentication and Authorization

Many existing networked conferencing systems are explicitly insecure—they make no guarantees about the integrity, confidentiality, or authenticity of messages. They also lack a flexible authorization system to control access to messages. Messages can easily be "forged" — appearing to come from a user who did not send them — or sent anonymously. There is rarely any effective control over who can enter a message in a conference, or any way to keep messages private. *Discuss* answers these issues by using an external authentication system (*Kerberos*) in combination with its own authorization mechanisms.

6.1 Authentication

In an open network system, a network service cannot trust the integrity of its clients; they can lie about the identity of their users. Through the use of an encryption-based third-party authentication system, such as *Kerberos*, it is possible

to allow servers to verify the identities of their clients. Thus, a *Discuss* server can use *Kerberos* to be sure that a supposed author really did originate a given transaction, and can label the transaction with his authentication name.

If the meeting is located on the same machine as the client, *Discuss* does not use *Kerberos*; instead, it relies entirely on the security mechanisms inherent in UNIX. The *Discuss* library spawns a `setuid` back-end program to handle the requests. The back-end knows the real user ID of the front-end, and uses the associated user name as the authenticated name of its client. The mail-to-*Discuss* gateway uses this system; mail messages are entered from `daemon` (the username under which `sendmail` runs filter processes), with the original `From:` line of the mail message appearing in the body of the transaction.

The result of the authentication process, whether through *Kerberos* or through UNIX mechanisms, is an authenticated name which is used as the basis for authorization.

6.2 Authorization

Once it has an authenticated name, the *Discuss* server must then make authorization decisions. One of our early design goals was to provide fine-grained access controls; that is, it should be possible to control the access to a meeting down to the accuracy of a single user.

Each meeting has an access control list (ACL), listing pairs of authenticated names and the associated access rights they have to the meeting. There are seven distinct permissions:

answer allows the user to reply to an existing transaction,

chairman allows the user to change the access control list.

delete allows the user to delete any transaction in a meeting.

read allows the user to read any transaction in a meeting.

owner allows the user to read, delete, and retrieve transactions that he entered; usually, chairmen are willing to allow a user to "take back" what he said, but some might not want to allow that.

status allows the user to find out summary infomation (last modified time, number of transactions, etc.) about the meeting.

write allows the user to start a new chain.

It should be noted that no one permission directly implies another. For example, chairman access does not imply read access.

A user name of "*" in an access control list gives the rights for users who are not explicitly mentioned on the ACL; in addition, a user name of "???" matches unauthenticated users. A meeting chairman can prevent anonymous transactions by denying answer or write permission to unauthenticated users.

In addition to providing control over who can access a meeting, *Discuss* also provides control over creation of meetings; specifically, an access control list file named `acl` must exist in the directory in which the meeting is created, and that ACL must grant 'a' access to the user who is attempting to create the meeting.

In the great UNIX tradition, there is currently no quota system limiting the amount of disk space that an individual meeting can use. A system administrator may impose a disk quota for the `discuss` pseudo-user; the total will apply to all meetings. *Discuss* handles disk-full or over-quota conditions gracefully.

7 Notifications

Since a meeting has a single location, other users can immediately see any new transactions added to a meeting. This adds a real-time flavor to *Discuss*; users can reply to transactions immediately, and create an on-going conversation among several participants. To facilitate this, *Discuss* allows users to be notified when a new transaction is added to a meeting.

Discuss uses the *Zephyr* NotificationService to inform interested users of new transactions in meetings. *Zephyr* isa notice transport system that delivers interactive messages in a networked workstation environment. Users register which classes of messages they are interested in; *Zephyr* will deliver messages to the appropriate users.

When a *Discuss* server adds a new transaction to a meeting, it sends out a *Zephyr* notice. Since this notice contains information such as the subject and author of the transaction, the transmission of the notice should not violate the privacy of the meeting. The server does not send out public notifications of new transactions in private meetings. If the meeting is publically readable (read access to "*"), it sends out the *Zephyr* message to recipient "*." Anybody can subscribe

to these messages. If the meeting is private, then the *Zephyr* message is only sent to those users who have read access to the meeting.

8 Client Tools

When writing the first *Discuss* client program, we came across a number of problems that had to be solved. Instead of solving these problems in a limited way, we attempted to build general tools that could find use in different situations. These tools have found their way into other *Discuss* clients, and other software projects.

This section describes the tools that were built to support *Discuss*. These tools are the *Discuss* library, which contains general routines that can be used by any *Discuss* client, the subsystem (*ss*) library, which supports command-line interfaces, and the Common Error-handling (*com_err*) package, which allows multiple packages to report errors in a coherent way.

8.1 The *Discuss* library

When we wrote our first *Discuss* client program, we were aware that there would be those who would not like the style of interface that we chose, and who would want some other interface to use. We wanted to be able to build clients with different interfaces without resorting to "sharing via text editor."

To meet this goal, we designed the *dsc* library. It performs all interaction with *Discuss* servers (via RPC calls) as well parsing and updating a user's `.meetings` file. Thus the user interface needs only know how to call these routines and need not know any of the implementation details.

Using this library, a prototype X-based client `xdsc` has been written. Although we found one or two very minor problems in the actual implementation, we believe the modularity imposed by the *dsc* library has greatly reduced the effort required to develop new user interfaces.

8.2 A Simple User Interface

Another package we wrote for this project was the original shell-like user interface. While the routines implementing the specific commands are very particular to *Discuss*, the driver that prompts for them is not. The package we use, known as *ss*, provides the input and dispatch routines for not only our own original *Discuss* client program, but also a control program used by *Zephyr*, one of the *Kerberos* administrative utilities, and various other programs which need a simple interactive shell-like interface. This section gives a brief overview of the use of this package.

This subsystem package consists of a command table translator and a run-time library. The command table translator converts a descriptive list of commands (including several command names, a short one-line description, and a subroutine name for each) into a C source file which is compiled and linked into the the application, along with the run-time library. The *ss* library consists primarily of a dispatch routine which prompts for input, reads and parses a command line, and calls one of the subroutines listed in the command table. A user of this system calls a routine to create a subsystem object, and then invokes a dispatch routine with that subsystem object as a parameter. The routines are called with "argc, argv" arguments similar to the C function `main`, along with a third parameter indicating which subsystem object the command was invoked from.

This subsystem package includes as built-in commands a "?" command, which lists the commands available (and their descriptions), and some simple commands like "quit". It manages input handling, primitive shell-like argument parsing (breaking up a string into an array of words), and a help utility that makes use of programmer-supplied directories of help files (to be viewed with "more").

Although this package lacks such features as intelligent argument handling and command and argument completion, it is extensible and still under development. Even without these features, we have found this subsystem management package to be useful for a number of applications.

8.3 Error handling

One of the aspects of the *Discuss* client programs is that they use different packages within the same program. These packages come from different sources, and each has its own way of reporting error conditions. Although UNIX has well-defined error handling for kernel routines, there is no standard way for libraries to return errors. In many cases, each package defines a set of error codes, and provides a separate method of converting this error code into an error message. For some procedure calls, an error may arise from several sources. In this situation, there is no way for the calling procedure to understand the error.

In response to this problem, we created the *com_err* package. It allows multiple libraries to return error codes in a coherent manner. With *com_err*, errors are represented as 32-bit unsigned integers. These integers can be returned by functions, passed between different libraries, and sent over the network. The *com_err* library supplies a routine to convert these integers into error messages.

To use *com_err*, the programmer first creates a text file describing the error table. This text file names the error table, and lists the symbolic names of the error codes and the error messages associated with them. The *error table compiler* reads this text file, and produces two files: a C source file that defines the symbolic names as C-preprocessor symbols, and an object file containing the error messages. The programmer includes the C source file to obtain definitions for the error codes, and links the object file into the application.

To avoid collisions between error codes, *com_err* uses the error table name to generate the numbers for error codes. Error tables are named using four letters. *com_err* collapses these letters into the top 24 bits of the error code, leaving 8 bits to differentiate among different errors in the same error table. This restricts error tables to at most 256 entries, which has proven more than adequate. The first 256 error codes are reserved for UNIX errors, so that UNIX errors can be treated as *com_err* codes.

We added *com_err* to *Discuss* by generating error tables for the different packages it used. For packages that have their own error codes, such as *Kerberos*, the caller of the package converts the returned value into a *com_err* codes by adding a constant. All *Discuss* routines return standard *com_err* error codes.

com_err provides a flexible approach of handling error codes, and has been adopted for other projects. For example, *Zephyr* uses *com_err* to encode and transmit error indications. *com_err* provides the glue that allows multiple packages to be integrated smoothly.

9 Status and Future Directions

Discuss has found ever-increasing popularity since its initial implementation in Fall 1986. Its user community has grown from a core set of SIPB members to more than 220 MIT students and staff. The number of meetings has grown from a few test meetings on a couple machines to 116 meetings on eight server machines.

9.1 Current *Discuss* meetings

Some of the current *Discuss* meetings archive mailing lists that exist on the Internet. For example, there are *Discuss* meetings that archive the "Info-Mac", "TeXHaX", "Risks", and "Kerberos" mailing lists. These meetings allow MIT computer users to read these lists without getting themselves added to the lists and having these messages intermixed with their regular electronic mail.

Discuss meetings support several services at Project Athena. They archive user bug reports, so that developers can review them in a readily accessible and efficient manner. *Discuss* is also used by the Athena On-line Consulting (OLC) System to store sessions between users and consultants. This way, all consultants can review interactions between users and consultants.

Several Athena development projects use *Discuss* to coordinate the activities of project members who have different schedules and different geographic locations. *Discuss* has been used to help develop *Discuss* itself, *Zephyr*, and *Kerberos*. With a *Discuss* meeting, chaining allows replies to be organized, and the archival nature of *Discuss* means that new members of the project can review past transactions.

Finally, *Discuss* meetings play a social role at MIT. There are *Discuss* meetings for random discussion, changed *plan* files, depressing thoughts and cheery thoughts, as well as private meetings for self-selected groups of people. These meetings frequently generate interesting, lively discussion.

9.2 Future of *Discuss*

At the moment, *Discuss* appears to have attained critical mass, in that the current user community is producing enough interesting information to sustain itself. Future development in *Discuss* will be aimed at improving the current system, and increasing its usage within the MIT community.

The next improvement to *Discuss* will be better user interfaces, so that *Discuss* can be used by people who are unfamiliar or uncomfortable with command-line interfaces. Currently, one of the authors is working on `xdsc`, an X-based client, so that users can read and enter *Discuss* transactions using a mouse/window interface. Minor

changes such as displaying a user's real name instead of his user name could have a large user friendliness payoff. Another way of increasing *Discuss* usage is to port *Discuss* onto other machines with distinctive user interfaces, such as the Apple Macintosh.

Along with different interfaces, we are considering adding new features to *Discuss*. With an X-based interface, it is useful to consider multimedia transactions, where *Discuss* transactions could mix pictures and text. *Discuss* could also be better integrated with mail, so that users could reply to *Discuss* mail archives directly. Another natural extension is the idea of voting, so polling can be conducted in an authenticated and convenient manner. *Notes* and *Forum* have shown that a "chairman message", shown to all users of a meeting or even just users about to enter a new transaction is useful; adding such a feature to *Discuss* would not be difficult. Per-transaction or per-chain "attributes" have been requested for meetings which track tasks, so tasks represented by transactions or chains could be marked as "resolved" or "urgent". This would be more difficult to do in a general way, but would probably be worth the effort.

Integrating a nameservice like *Hesiod* into the meeting location process would simplify finding new meetings and would allow meetings to be easily moved from one server to another. At the moment this information is scattered among all the `.meetings` files. Since we are not sure that the benefits from this are worth the cost of centralized registration of meetings, this is still a topic for further thought.

Another open issue is that of server-side state. Currently the *Discuss* server retains no per-user state. This greatly simplifies its implementation, but it makes the `.meetings` file more important and does not allow one to discover who else is attending a meeting or which transactions they have read. Some people feel having this information available would be useful, others think privacy concerns preclude releasing such information.

10 Acknowledgments

We would like to thank Robert French, Ted Ts'o, and Brian LaMacchia for their work on the implementation of *Discuss*; Nancy Gilman for her efforts to document *Discuss* for mortal users; Brian LaMacchia, Andrew Berlin, Dan Geer, Judith Provost, Ron Hoffmann and others for reviewing early drafts of this this paper; the fellow members of the SIPB[7] for fostering the kind of supportive environment that leads to work such as *Discuss*; and most of all, the users, who have made *Discuss* a success.

References

[1] E. Balkovich, S. R. Lerman, and R. P. Parmelle. Computing in Higher Education: The Athena Experience. *Communications of the ACM*, 28(11):1214–1224, November 1985.

[2] Nathaniel Borentstein et al. A Multi-media Message System for Andrew. In *USENIX Association Winter Conference 1988 Proceedings*, pages 37–42, February 1988.

[3] David D. Clark. Unified Stream Protocol. Technical Report CSR RFC 272, M.I.T. Laboratory for Computer Science, February 1985. Internal Working Paper.

[4] C. Anthony DellaFera et al. The *Zephyr* Notification Service. In *USENIX Association Winter Conference 1988 Proceedings*, pages 213–220, February 1988.

[5] Terence H. Dineen et al. The network computing architecture and system: an environment for developing distributed applications. In *USENIX Association Summer Conference 1987 Proceedings*, 1987.

[6] Stephen P. Dyer. The *Hesiod* Name Server. In *USENIX Association Winter Conference 1988 Proceedings*, pages 183–190, February 1988.

[7] Raymond B. Essick IV and Rob Kolstad. Notesfile Reference Manual. *UNIX User's Supplementary Documents*, February 1983.

[8] Chris Hancock. Common Ground. *Byte*, 10(13):239–246, December 1985.

[9] Honeywell Information Systems. *Multics Forum Interactive Meeting System User's Guide*, CY74-00 edition, July 1982.

[10] Mark R. Horton. How to Read the Network News. *UNIX User's Supplementary Documents*, April 1986.

[7] All of the authors are members of the SIPB. Ken, Jon, and Stan have served as Chairman of the SIPB.

[11] Information processing systems — Open Systems Interconnection. Specification of Abstract Syntax Notation One (ASN.1). Technical Report ISO 8824, International Organization for Standardization., December 1987.

[12] B. Liskov et al. Communication in the Mercury System. In *Proceedings of the 21st Annual Hawaii International Conference on System Sciences*, pages 178–187, January 1988.

[13] Jay Pattin. A Design for a Networked Computer Conferencing System. Technical report, Massachusetts Institute of Technology, May 1983. Bachelor's Thesis.

[14] J. B. Postel. Simple Mail Transfer Protocol. Technical Report NIC/RFC 821, Network Working Group, USC ISI, August 1982.

[15] R. W. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1987.

[16] Richard Stallman. *GNU Emacs Manual*, sixth edition, March 1987.

[17] Jennifer Steiner, Clifford Neuman, and Jeffrey Schiller. Kerberos: An Authentication Service for Open Network Systems. In *USENIX Association Winter Conference 1988 Proceedings*, pages 191–202, February 1988.

[18] Sun Microsystems, Inc. XDR: External Data Representation Standard. Technical Report NIC/RFC 1014, Network Working Group, USC ISI, June 1987.

[19] Sun Microsystems, Inc. RPC: Remote Procedure Call Protocol Specification Version 2. Technical Report NIC/RFC 1057, Network Working Group, USC ISI, June 1988.

[20] G. Winfield Treese. Berkeley UNIX on 1000 Workstations: Athena Changes to 4.3BSD. In *USENIX Association Winter Conference 1988 Proceedings*, pages 175–182, February 1988.