The iFlame Client-Based Instantaneous Datagram Communication Substrate

by

David Michael LaMacchia

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degrees of

Master of Engineering in Electrical Engineering and Computer Science

and

Bachelor of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 1996

Author ....
Department of Electrical Engineering and Computer Science
May 17, 1996

Certified by ....
Walter Bender
Principal Research Scientist
Thesis Supervisor

Accepted by ....
Frederic R. Morgenthaler
Chairman, Departmental Committee on Graduate Students

The iFlame Client-Based Instantaneous Datagram Communication Substrate

by

David Michael LaMacchia

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis investigates a new protocol, iFlame, designed to provide highly scalable, distributed, real-time communication systems for the Internet. Scalability is achieved by using a client-oriented model rather than a more traditional server-centric one. The protocol specifies that servers maintain the locations of clients, but transactions are committed in a client-to-client fashion. iFlame is unique because it requires each client to be responsible for handling the load associated with the transactions it initiates. The claims of the protocol are demonstrated in an implementation called the iFlame Message System which combines the ability to send audio, video, text, and other MIME-encapulated data with dynamic client configuration via the Tcl/Tk scripting language. The system achieved several goals: scalability, efficiency, portability, and concurrency.

Thesis Supervisor: Walter Bender
Title: Principal Research Scientist

The iFlame Client-Based Instantaneous Datagram Communication Substrate

by

David Michael LaMacchia

Submitted to the Department of Electrical Engineering and Computer Science
on May 17, 1996, in partial fulfillment of the
requirements for the degrees of
Master of Engineering in Electrical Engineering and Computer Science
and
Bachelor of Science in Computer Science and Engineering

Abstract

This thesis investigates a new protocol, iFLAME, designed to provide highly scalable, distributed, real-time communication systems for the Internet. Scalability is achieved by using a client-oriented model rather than a more traditional server-centric one. The protocol specifies that servers maintain the locations of clients, but transactions are committed in a client-to-client fashion. iFLAME is unique because it requires each client to be responsible for handling the load associated with the transactions it initiates. The claims of the protocol are demonstrated in an implementation called the iFlame Message System which combines the ability to send audio, video, text, and other MIME-encapulated data with dynamic client configuration via the Tcl/Tk scripting language. The system achieved several goals: scalability, efficiency, portability, and concurrency.

Thesis Supervisor: Walter Bender
Title: Principal Research Scientist

To my family

Acknowledgments

This thesis could not have been completed without the support of a great many people. I wish to take this opportunity to express my appreciation for their help throughout this project. In particular:

Walter Bender, my thesis advisor, for taking a chance and making this all possible.

Gerald J. Sussman and *Harold Abelson* for their advice and assistance in all things.

Brian D. Carlstrom, with whom I developed the original iFlame project, for many hours of support, comments, and more support. This project would not have existed without him.

Brian A. LaMacchia, for doing more for me than any brother has ever been expected to do, and for coming through for me time and time again. He taught me honor, character, and brought strength to me when no one else could.

Pascal Chenais, for his wise advice and for giving me a chance when no one else would.

My parents Robert and Sherry LaMacchia, for being damn fine parents who were completely understanding.

Harvey Silverglate, *David Duncan*, *Andy Good*, and *Sharon Beckman*, for being willing to fight and for teaching me a few things.

Paul Butkiewicz, for always being the first one to hear everything and not complaining, and also for all the coffee.

Ryan Smith, for being a great friend, for sharing my sense of humor, and for all the road trips, Stuckey's, camping expeditions, and cephalopods.

Larry Goldfinger, Wendy Tuggle, and Jonathan Frome, for staying with me and understanding that I was with them in heart, if not in body.

Michael Blair, *Philip Greenspun*, *Bill Rozas*, *Kleanthes Koniaris*, and *Jason Wilson*, who made work at the AI Lab special.

Anca Mosiou, for her spirit and urging to get this thesis completed.

Joe W. Barco, for understanding what I was going through, and for the Cheez-its.

Chris Pezzee, for putting up with me as a roommate.

Bahman Rabii, *Jordan Share*, *Dan Hurda*, *Mike Stephens*, and *Weip Chen*, for friendship and all the games that kept me sane.

Hani Sallum, for his sense of humor.

Stephanie Spavaro, for being wise beyond her years, and for not being vapid nor blonde.

"Big" Jimmy Roberts, for knowing nothing, seeing nothing, and being a friend.

Trey Anastasio, *Mike Gordon*, *Page McConnell*, and *Jon Fishman*, for musical inspiration during the long hours writing code.

and to *Helen*, who was with me during two difficult years. I'll miss you.

Contents

List of Figures

Chapter 1

Introduction

Things fall apart; the center cannot hold;

Mere anarchy is loosed upon the world.

— W.B. Yeats

Yeats fussed about things falling apart and the center not being able to hold.

What really happened was that the center ceased to exist all together.

— John Barnes

1.1 Evolution of the Delocalized Community

As the Internet continues to invade the popular press and gain acceptance among people as a viable commodity, communities continue to be both created and enhanced by the network. Online communities are inherently *delocalized*; they are not limited by geography. The Internet holds the promise of allowing large numbers of delocalized people to communicate in ways that have been previously impossible. Unfortunately, technologies available on the network do not scale well, hindering the development of many of these new forms of communication. Solving this scalability problem is critical if we wish new and more varied communities to exist online. In order to do so, however, we must first understand how communities will survive and take advantage of a networked world.

It is no surprise that there is much confusion about how communities form in a networked environment: the notion of community is thousands of years old. In ancient times, a community often consisted of a single village in which each person might have had contact with at most one hundred others during his entire life. In the period following the agricultural revolution, one person may have had contact with as many as one thousand other people. By the early part of the twentieth century, a technically-minded person would had the chance to contact several thousand people on a moment's notice by simply taking a train to the next town over and stopping at one of many local pubs. And in the "Age of Information?" Millions of disembodied voices contacted through use of electronic mail[5], Usenet news[9], the World Wide Web[2], and hundreds of other options. Is this the modern community?

Some futurists might like to think so. In the past, communities were often formed in part by geography: people were thrown together in a particular suburb and suddenly they gained the classification of "neighbors." In the electronic world, without the constraints of geography, common interests drive the formation of communities. Communities may be composed of one's high school friends, the people seen on the subway every day riding to work, the people who watch the television show *Babylon 5*, or countless other groups of associations made every day. Any of these associations can exist offline, and any of these associations could be supported online. In the networked world, the difference is that geographically distant people can converse and publish information for each other in means that would be infeasible in the real world.[1]

A *delocalized community* is any grouping of people with a common interest who may not be geographically near each other. Delocalized communities can exist offline, such as in telephone chat lines, but frequently they are formed via a method of communication involving the Internet. The Internet is the direct descendent of the ARPAnet, which was developed 25 years ago and was used exclusively by scientists

[1] Just imagine if electronic mail, like a telephone, was associated with a monthly long distance bill!

for scientific research. These scientists created their own communities to better share information regarding their work. Now, a quarter-century later, access to the Internet is almost a commodity and includes people who decide that access is worth a monthly charge. They, too, find friends in the online world and arrange themselves into special interest groups, just as they do now in the real world.

Delocalized communities are defined in part by the method of communication upon which they are based. In the online world, this means that these communities are severely limited by the available technology. Most large groups of Internet users with shared interests associate via *store-and-forward* applications, which store the contents of a message authored by one user for retrieval by others at some later time. Prime examples of store-and-forward applications are electronic mail and Usenet news. Unfortunately, the Internet currently ignores another important form of communication: the ability to hold conversations in real-time.

Communication is, of course, the heart of the Internet, and it is therefore extremely surprising to discover that the evolution of online communication has been notably lopsided. Two forms of association exist on the network: *asynchronous* and *synchronous*. Asynchronous communication includes those methods where a party provides information to another party without knowledge of when that data will be received, as in the store-and-forward applications mentioned above. Synchronous communication revolves around real-time data transactions between two or more parties. Considering that the Internet's communities try their best to mimic real world associations, it is startling that almost all development has centered on improving asynchronous communication.

Prof. William Mitchell, Dean of the School of Architecture and Planning at MIT, has been examining the difference between communities online and offline for some time.[13] Mitchell's claim is that over time, communication has migrated away from synchronous conversations that occur face-to-face to other forms. These other forms of communication, he claims, are enhancements to the way people have discussions rather than replacements for the older methods. Considering the (usually intuitive) economics of time and space, people are able to choose which method of communication best suits them at any particular time. He demonstrates this via a grid comparing popular forms of synchronous and asynchronous communication with *presence* (localization) and *telepresence* (delocalization). An example of his grid is depicted in Figure 1-1.

	Synchronous	Asynchronous
Presence	Face	Note/letter
Telepresence	Phone, shared virtual environment	Voice message, electronic mail

Figure 1-1: The Communication Grid

The iFLAME protocol is an attempt to create a synchronous form of Internet communication that can be used to create communities as varied as those in the real world. Most online communities are currently based on asynchronous communication; in the real world the analogous situation would be if all conversations took place over telephone answering machines, a store-and-forward technology. To best simulate conversations and form true communities online, users desire to converse in real time.

1.2 Asynchronous Communication

Historically, the Internet has focused on providing efficient asynchronous communication services. Asynchronous applications allow a corporation with fast machines and a good network connection to serve data easily to clients. In the past it has been necessary for slow clients to allow a remote server to do intensive work for them. This type of client-server system is a *server-centralized* one, because clients depend on one or more servers to process and provide data.

1.2.1 The World Wide Web

The best example of asynchronous communication is the development of the Hypertext Transfer Protocol (HTTP) and, along side it, the World Wide Web. The Web itself is a conglomerate of old and new technologies, including the File Transfer Protocol[15] (FTP, arguably the oldest form of moving files over the network), Gopher[12] (developed in 1992 as a better means of serving files over the Internet), Usenet news, and HTTP, as well as many others. In this medium the user has the option of being publisher, consumer, or both. Unlike other asynchronous mediums like newsprint or television, it is extremely easy and inexpensive for individuals to provide useful services on the network.[2] The development of asynchronous Internet communication technology has been explosive; there is an increasing number of products such as Netscape plug-ins and positions for artists and computer programmers every day.

1.2.2 Electronic Mail

Electronic mail (email) is a venerable form of asynchronous communication with which almost all online users have experience. Email also often defines delocalized communities in ways the Web cannot. While more Web browsers are enabled every day to allow users to send electronic mail from within them, electronic mail is generally not considered to be part of the Web. This is because most mail messages are not available to a general unknown audience in the same way that other information might be provided by anonymous FTP or HTTP. On the ARPAnet, scientists used mailing lists to discuss current research projects. Not much has changed since the 1970s for email, except perhaps for the growth of the general user base and the variety of mail agents. One might argue that mailing lists each represent a single community, literally categorizing users as members of a group with a particular interest.

[2] Some might argue that an entire industry has developed around individuals hoping to make money by selling their popular small services to big companies, such as Webcrawler's alliance with America Online.

1.2.3 Usenet News

Like electronic mail, Usenet news ("netnews") is an interactive yet asynchronous form of Internet communication. Unlike email, netnews has the distinct advantage that the recipients of a message need not be specified; it is possible to "lurk" on a newsgroup, to read messages and be effected by them, without disclosing one's presence to anyone else. Users post messages in hierarchically-organized groups; an attempt is made (though often not successfully) to limit discussion in particular groups to particular topics. Each group is a community in itself, defined by its topic of discussion, led by prominent personalities (frequent posters), adhering to guildlines of behavior (rules of netiquette), and documenting a history (Frequently Asked Questions lists).

1.3 Synchronous Communication

Synchronous communication has undergone few changes over the past ten years. Network aware versions of the `write`[16] command and various `talk` programs are the oldest forms of synchronous network communication and are still widely used today to allow two (or sometimes more) users to establish a virtual pipeline that can send and receive text. There is little technological difference between these programs and, say, Internet phone programs that allow two people to have long distance "telephone" conversations using the Internet as the transport mechanism. Communications systems like the Internet Relay Chat (IRC)[14], MIT's Zephyr Notification Service (Zephyr)[8], and the Multicast Backbone (MBONE)[11] allow users to talk to other users of the system, either individually or in groups, in real-time. IRC and Zephyr are inherently limited because they revolve around routing all data through a central server (or network of central servers) which means that they are not scalable and can only support a finite number of users [3] Zephyr, because of its dependence on three central servers, is only feasible within a single Kerberos[10] realm. At MIT, Zephyr has a difficult

[3] For IRC this number, according to recent studies, is about 12,000 users. In fact, IRC, because of various political battles over how the web of servers should be connected, has divided itself into two main networks, the EFNet, and the Undernet, which together comprise the majority of IRC servers, although some "island" servers exist that are not connected to either.

time supporting the student body, which is about 10,000 students.[4] Similar to IRC, the MBONE uses a multicast network[6] with a specific topology to route information. Unfortunately, MBONE's multicast protocol is extremely unportable and not viable on many systems.

Communication on the Internet is certainly more varied than suggested by this simplistic attempt to divide data transfer into two groups; delocalized communities revolve around hybrids of asynchronous and synchronous technologies as well. For example, in 1993 the MIT Media Laboratory developed a system for an electronic personalized newspaper, originally dubbed the Freshman Fishwrap[5]. The Fishwrap[3] (which was later made available to the entire MIT student body) claimed to make each student reader, editor, and contributor; not only did the user input the general format of newspaper he desired and select the news he would receive, but he could also submit articles (and comments on articles) that other people in the Fishwrap community would see and could comment on as well. By involving the entire readership in the creation of the newspaper, a shared association that revolved around these submissions evolved into a community of students.

1.4 iFlame

1.4.1 Motivations for the iFlame Protocol

Every day there is a greater demand for the creation of larger delocalized communities able to commit faster and more efficient transactions. Network-aware games, from popular action games such as id Software's *Quake* to Origin's *Ultima Online* exemplify the growing desire for synchronous communication substrates that can scale to an unlimited number of people. If bandwidth-intensive applications such as virtual reality simulations and video conferencing are to ever be developed, synchronous

[4] This number is misleading. Considering the number of Athena workstations and private computers that support Zephyr, the system rarely handles above 1,500 users at any one time. Zephyr, in reality, has much difficulty servicing more users than this.

[5] As the saying in the newspaper industry goes, "Yesterday's news wraps today's fish."

Internet communications must undergo great changes.

The "Internet Flame," or iFlame, protocol is the next logical step in the evolution of synchronous communication. iFlame supports a scalable network that emphasizes the creation of communities of people located anywhere on the Internet. Because it is scalable, these communities are not restricted either by physical or virtual locality, or, most importantly, by the number of iFlame users online at any given time. People using the system send "flames"[6] to each other; each flame consists of packets of data containing anything from simple text to video. Each user controls a local client; clients use a network of servers to locate other clients, but client-client communication does not in general involve the servers. We encourage long-term users of iFlame to establish a permanent address for themselves on a server so that, as with email addresses, a user can be consistently found by others.[7] As in IRC, discussion groups can be created with access control to allow users to better shape their community. Because the iFlame protocol is primarily client-client communication, the load sharing necessary to create efficient applications from network games to video conferencing is possible.

1.4.2 Clients and Scalability: the iFlame Mantra

The basic tenet of iFlame is simple: while servers must exist to keep track of clients' locations and maintain authentication information, the client itself should do the work of sending messages to other clients. If a user decides to send a copy of a home movie he made of his child learning to walk to a group of 200 other users, his client directly bears the burden of sending a large amount of data to a large number of people, not a centralized server[8]. In addition, if a person wants to have an extensive

[6]Historically a flame is, according The Hacker's Dictionary, an instance of one who "speak[s] incessantly and/or rabidly on some relatively uninteresting subject or with a patently ridiculous attitude."

[7]As we'll see later, an iFlame address appears much like an email address, as in `dml@iflame.mit.edu`.

[8]If a centralized server did have to send out 200 copies of that movie, other users would find themselves penalized, as their messages would not be handled while the server completed this load-intensive task

private conversation with someone across the country (or across the world!) there is no reason the activities of other users should hamper their discussion.

iFLAME is able create a client-based network by locally caching the locations of remote clients with whom a user is communicating. If a remote client is added to or removed from a *forum* to which a user has subscribed, then the server is able to update this subscription information in the client's local cache in the background while that client continues to communicate with other clients. In a typical transaction, the majority of the work performed by the server is done the first time a user decides to write to a forum, since the server must initially tell that client the location of every subscriber to that forum. After the client has been initialized, the server need only update the writing client as other clients subscribe to or desubscribe from that forum.

Because of the simplicity of its design, the iFLAME protocol easily supports higher-level networked applications. While the first half of this paper describes the protocol itself, the feasibility of the iFLAME protocol is investigated in an implementation described in the second half. The system consists of a server, *iflamed*, a client, *iflamec*, a user interface to the client, *iflame*, as well as display handlers (for the X Window System) and various helper applications (for such tasks as server-side user database manipulation and interpretation of client-side user-definable Tcl scripts to control data presentation). The result is a substrate that allows for an extremely scalable system.

To best illustrate our findings, the remainder of this paper is divided into three sections. Chapter 2 describes the iFLAME protocol in great detail as well as the design decisions that led to its final form. Chapter 3 introduces the iFLAME Message System, a chat network built using iFLAME. This chapter also discusses the practical issues involved in designing other similar systems. Chapter 4 concludes the thesis by investigating whether or not the communication system succeeded as a scalable system. Suggestions for future development and other synchronous communication systems are also included.

Chapter 2

The iFlame Communication Protocol

The goal of the iFlame protocol is scalability. Unlike systems such as IRC or Zephyr, where a static group of servers can handle a finite number of clients, iFlame allows for a dynamic number of servers to exist at any time and thus puts no constraints on the number of simultaneous users. Because iFlame is proposed as a solution to a scalable integrated message system, this tenet defines the protocol. Although particular services built on top of iFlame will vary greatly, there are some fundamental claims concerning the network environment we may assume are true. First and foremost, each user potentially wishes to commit transactions with other users who may be an arbitrary distance away on the network. Second, the user is willing to take responsibility for his actions but should not be penalized for the actions of others; that is, a user's client should handle the bulk of the load generated by a resource-intense transaction. Third, the user should be willing to have his client interact with at least one server he trusts to know about his location.

The heart of iFlame resides in *forums* and *users*; a *forum* is a place were communication takes place between *users*. Forums are both single clients and groups of clients. Once a client subscribes to read the contents of a forum (by informing the server where that forum resides), the client will continue to receive the messages remote clients direct to that forum *directly from other clients* until it unsubscribes from the forum. The client does not have to worry about how remote clients will locate it; this task is handled by the server upon which the forum is hosted. Passive readers of iFlame forums do little work outside of receiving connections from other clients and displaying the received data in an appropriate fashion. Active clients that wish to send data, called *writers*, must keep track of the locations of all clients subscribed to forums to which it sends messages and also receive updates from servers when new locations are added or old ones are deleted.

This chapter describes the details of the iFlame protocol, taking into account the assumptions we have made above concerning the conditions under which the client and server operate. Arguments for and against each design decision are presented and the reasoning behind choosing a particular implementation are described. Section 2.1, *The Basics: Forums, Locations, and Users*, examines issues surrounding the fundamental concepts of the iFlame protocol. Section 2.2, *A Simple* iFlame *Interaction* which presents the basic transactions that might occur between a client and a server as well as multiple clients. A sample implementation of the protocol in the form of a chat system and the design decisions inherent to its construction are described in the Chapter 3.

2.1 The Basics: Forums, Locations, and Users

2.1.1 Forums

Forums are the key to iFlame. A forum is a similar concept to an IRC "channel" or Zephyr "class," in that it is the protocol's means of subgrouping clients. Unlike channels or classes, however, iFlame forums are hosted by a particular server much like an email mailing list. Thus, the `babylon-5` forum on `iflame.mit.edu` is distinct from the one located on `iflame.foo.org`.[1] Similarly, for a forum associated with a particular username (called a *personal forum* though they differ in no way from other forums), `dml` at `iflame.mit.edu` represents a different forum then the `dml` forum

[1] Assuming, of course, these are different machines!

on `iflame.foo.org` A forum consists of three distinct parts: a collection of *Access Control Lists*, a list of forum *readers*, and a list of forum *writers*. The forum structure is shown in Figure 2-1.

```
typedef struct forum
{
  char *name;
  tree_t *readers;
  tree_t *writers;
  tree_t *acl[NUM_ACL_TYPES];
  int refcount;                  /* reference count for GC */
} forum_t;
```

Figure 2-1: The iFlame Forum Structure

There are several reasons iFlame is organized around the forum structure; the best way to exemplify these is to distinguish the protocol from a relay system like IRC. IRC uses the channel structure to group users; a channel is a globally recognized set of information that contains a list of users reading the channel and a list of the access controls for that channel. This model works well for IRC since the system does not try to conserve bandwidth or try to push any load off of servers onto clients. In iFlame, however, readers are distinguished from writers because only forum writers require cache updates. Furthermore, since a forum resides on exactly one server, no inter-server communication is required and cache updates depend only on the forum's host server.

iFlame forums are the result of lessons learned from the mistakes of other systems. For instance, IRC's generally recognized problems with global name resolution do not exist in the iFlame model. Similarly, the race conditions that exist because a relay network can have servers receive updates at different times[2] are not present a

[2]Also, updates are arriving from a multitude of servers at once

system built using iFlame because servers do not have to talk to each other.

Access Control Lists

A forum's Access Control Lists (ACLs) constrain who can perform various operations with respect to that particular forum. There are six distinct ACLs per forum: `read`, `write`, `admin`, `deny read`, `deny write`, and `deny admin`. These ACLs allow an administrator of a forum (a member of the forum's `admin` ACL) unlimited customization of that forum. For example, the administrator could make a forum available to everyone but, say, a user who for some reason needed to be denied access (say, if the group was planning a surprise birthday party, or if a user had become unruly).

ACLs permit some interesting results, such as "personal" forums. A personal forum is one upon which a user wants to receive private messages. Such forums may be implemented by allowing anyone to write to the forum but only the user to read from the forum. A *login* forum, announcing the fact that a particular user has logged in, might allow only one writer yet multiple readers (the writer being the person logging in; the readers would be those interested in knowing this information). Of course, if a user does not wish to announce when he was logging in or out, he could set the ACLs appropriately. Forums with multiple readers and writers are, of course, suitable for group communication.

Forum Readers and Writers

Forums also contain lists of clients currently reading from or writing to the forum. Each list is generally updated only after receiving a cache update from a server. Note that a user who has subscribed to a forum as a reader, but not as a writer, does not receive lists of writers because they are only receiving data from other clients.

2.1.2 Locations

When a user joins a forum, his location is added to the list of locations of active readers both on the server and his own client's end. A location uniquely identifies the user's client within the iFlame system. It is the job of a server to maintain the locations of the clients authenticated to it.

A location is simply an Internet address and set of two ports; one to receive client connections and one to receive server connections.[3] An example of a client location is shown in Figure 2-2.

IP address	Client Port	Server Port
iflame.media.mit.edu	36356	36612

Figure 2-2: A typical location structure

Using this scheme, it is possible for a user to join a forum from various locations without conflict. Further, multiple users on a single machine may have their own iFlame clients, as these port numbers are chosen dynamically by the client at runtime. It is not necessary for any part of the client to run with special privileges (e.g. as "root") in order to dynamically establish these connections.

The server needs to maintain a cache of the locations of all clients reading forums that server manages. This cache includes both clients authenticated to the server as well as unauthentic clients. Clients, however, need only maintain locations of other clients to which it needs to send data. Initially, a client's local cache of these locations is empty, although each server managing forums that client is writing to has the ability to dynamically update the client as other client's add and delete themselves from forum reader lists.

The server also has an additional task besides just keeping track of which clients are reading and writing forums it manages. The server must also keep a database containing information about each forum it hosts. This database necessarily includes the ACLs for each hosted forum.

Locations are kept simple for pragmatic reasons. When a server sends cache updates to clients, it sends a series of locations to either add to or delete from a

[3] As we'll see in Chapter 3, it is important for security reasons that a client not accept untrusted server connections on the server port, otherwise that client's caches could be maliciously updated.

particular cache. As the size of a forum grows, logically more people join and leave that forum; when new users join that forum as writers they are sent more locations as the cache is brought up to date. Since a location sent to clients is just an IP address and a client connection port, relatively few bytes need be sent per location update, thus reducing the bandwidth required to update a client's forum cache.

2.1.3 Users

An iFlame *user* is simply the identification a client presents to a server to maintain authentication data. iFlame users forum the basis for the persistent identity a person maintains over time and shares with others. Each authentic location is associated with a user. When a forum request is made from an authentic location to a server, that location's associated user is compared to that forum's ACLs to determine if the request should be allowed to denied. No particular form of authentication is required by the protocol; the choice of whether to use authentication at all, and the particular method of authentication to use if desired, is left to the discretion of applications built on top of iFlame.[4]

Addresses

One of the goals of iFlame is persistence of users. Like a long-term email address, we hope that individual people will want to maintain a consistent identity on a particular server so their friends and coworkers can find them from session to session. Persistence of identity is accomplished via iFlame users. A user may have no persistent client but instead have a "home server" where others can attempt to find him. For example, `dml@iflame.mit.edu`, a proper iFlame address, designates that user `dml` is located on the server `iflame.mit.edu`. Of course, the only difference between a personal forum and a public forum (or any other type) is the set of ACLs associated with it. Thus, the address for a public forum, such as a discussion of one's garden, will have a similar format (such as `gardening@iflame.mit.edu`.

[4] Chapter 3 describes some of these in use in the iFlame Message System.

One of the considerations when designing the iFlame protocol was whether or not servers should have a global view of the entire system. That is, should there be a single `gardening` forum or should `gardening@iflame.mit.edu` be distinct from `gardening@iflame.microsoft.com`?[5] It was eventually decided that if iFlame was going to support a dynamic system then forums should reside on a particular server and no other server should necessarily know about its existence. This model also supports a decentralized system and does not reduce scalability since servers have no dependence on each other.

User authentication

Since users authenticate to a particular server, it makes sense for the home server to maintain authentication data.[6] If a user establishes any authentication method for a session besides `UNAUTHENTIC`, the server must access data about the user (submitted via some outside mechanism) each time a client opens a connection to a server. Under the `PASSWORD` scheme this initialization transaction might involve just a simple transfer of a username and/or password, while `PGP` authentication could require a more complex encryption key exchange and authentication protocol. A sample of the authentication protocol is described in Section 2.2, below.

2.2 A Simple iFlame Interaction

While forums, locations, and users are central to iFlame's protocol and support the scalability of its clients and servers, the benefits of the system begin with the protocol used for message passing. Because transactions are committed in a client-to-client fashion and servers are only used for tracking location information, transactions encumber servers with much less load than in a centralized-server environment. In this

[5] As an example, IRC uses global name resolution so every IRC channel is required to have a unique name in the system.

[6] When discussing authentication types, designations from the implementation described in Chapter Three are used. There are four methods described there: `UNAUTHENTIC` or unauthenticated transactions, `PASSWORD`, or simple password validation, `KERBEROS`, designating the MIT Kerberos authentication scheme, and `PGP`, verification of users using Pretty Good Privacy (PGP) key exchange.

section we present a step-by-step examination of the message passing protocol.

The main delivery path transfers a message first from a user to an iFlame client, then between iFlame clients, and finally from the destination client to its user, as illustrated in Figure 2-3. The server is not involved at all in these transactions so long as the topology of the system is maintained. One of three situations may require the server to connect to the client or vice-versa. First, the client could become a new *reader* on a forum; in this case, the server must be notified so that it can update its cache. Second, the client could become a new *writer* on a forum, in which case the server must notify the new client of all current readers on that forum so the new writer knows whom to connect to when writing to the forum. Third, if someone joins or leaves a forum[7] the server will have been notified and will subsequently relay that information to every client that is a writer to the forum.

[7] This includes a client crash.

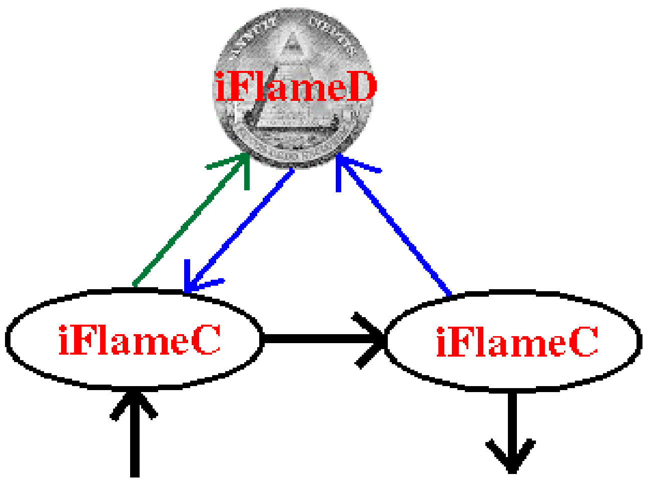

Figure 2-3: The iFlame Datapath

Client-issued commands

User clients may issue commands on behalf of the user to remote servers and other clients. These commands are described briefly in Figure 2-5. Each command is accompanied by a description of its functionality as well as the behavior associated with the various return codes with which the server could respond. In general, the iFlame protocol's return codes are defined in a manner similar to FTP. Return code are detailed in Figure 2-4.[8]

Response Code	Definition
100	unspecified action; left for expansion
200	OK; action completed; action acknowledged
300	more data required
400	local data error
500	failure or command incomplete

Figure 2-4: Generally defined client/server response codes

Client-issued commands are more verbose than what a user might directly instruct the client to send. For example, a user should never need to specifically tell a client to send a `LOC` message as this should automatically be done each time it is necessary. Similarly, a `+/- w` would never be sent from the user to the client; this command would be sent to a server the first time a user sent data to a forum to which he was not previously writing (as determined by his local writer cache).

Server-issued commands

The iFlame protocol also specifies a set of server-issued commands; in general, they allow the server to perform cache updates on the client side.[9] These commands are

[8]As is evidenced by Figure 2-4, we have left some room for further development of the protocol. Possible directions for future work are described in Chapter Four.

[9]Chapter Four addresses the security concern regarding malicious servers modifying other clients' caches.

directly outlined in Figure 2-6. Server-issued commands are always sent in response to client activity.

2.2.1 A Typical Session

An example of a typical iFLAME session is illustrated below. The session represents the client-side interaction with a server and another client. The user in this session, `dml`, initializes his client and subscribes to a sample forum, `iflame_help`, located on the default server.[10] Once subscribed, the client then sends a text message to the forum.

1. Client initialization

```
dml@Slip-Dml% iflamec
Using unix socket: /tmp/iflamec.Slip-Dml17213.0
iFlameC client initialized
```

At this point the client has been started. Initialization includes reading data from local configuration files, environment variables, establishing sockets, defining aliases, and other similar activities. The details regarding this procedure are described in Chapter Three.

2. The client is informed to begin reading a forum

```
Reply   : 200 I flame, you flame, we all flame for iFlame.
Command : +r iflame_help
Reply   : 200 +r complete
GCing connection
```

The above interaction demonstrates how a connection is accepted from our user-level client-side application. This user-level abstraction is used to gather commands

[10] The default server for this example, as designated in the session when we start to receive cache updates from it, is `alphaville.media.mit.edu`.

from the user and then send them to the client. In the implementation described in Chapter Three, this program is also used to send data to remote clients.[11]

Once the connection has been accepted, the standard `200` response code is returned via the unix socket with the welcoming banner, `I flame, you flame, we all flame for iFlame`. Again, this is a model similar to that of FTP. The command `+r iflame_help` is received by the client and the user-level program is told that the command was received. Afterwards, the unix socket is garbage collected, as designated by the debugging output, `GCing connection`.

3. The client connects to a server

```
Command : 200 I flame, you flame, we all flame for iFlame.
Request : LOC 36356 36612
Command : 200 Location set.
Request: AUTH PASSWORD
Command : 200 Authentication type set.
Request: USER dml
Command : 200 User set.
Request: PASS mypass
Command : 200 Password accepted.
Request : +r iflame_help
Command : 200 Now reading forum.
GCing connection
```

As the client wants to begin reading forum `iflame_help`, it needs to subscribe by sending a `+r` command to the server. This is done by first opening a connection to the server and then sending the command and awaiting a response.

We follow the standard procedure for opening a connection to the server. The client connects and waits for the opening server banner, `200 I flame, you flame, we all flame for iFlame.`[12]. Because this is a new server connection, the client needs to identify itself to the server by sending its location via the `LOC` command.

[11]There is no reason this user-level program, `iflame` could not have been combined with our client application, `iflamec`, as we show in Chapter Three. This two-tier model was chosen specifically to resemble one MIT students would be familiar with, that of the MIT Zephyr system.

[12]As in our user-level banner example, the text doesn't matter, just the response code

Receiving a response that the location has been accepted and noted, the client designates the authentication protocol PASSWORD for this session with the server. As this protocol requires a username and password to be sent to the server, USER and PASS commands follow with respective responses. Finally, the +r command is sent to the server.

4. Data is input for sending to a forum

```
Reply   : 200 I flame, you flame, we all flame for iFlame.
Command :  iflame_help
CACHE:
```

The client receives the command iflame_help from the user-level application. This designates that the user wishes to send some sort of data to the iflame_help forum. Because we're operating in debugging mode, we get additional data output to the display; the CACHE: line is a result of this, and displays the contents of the forum cache local to the client, which is at this point empty.

Although not displayed here, at this point the user is prompted for data to send to the forum. This is binary data and thus can be of any particular format. The implementation described in Chapter Three uses MIME encoding for messages and we recommend MIME as a standard format for iFlame datagrams.

5. A new writer is added to a forum

```
Command : 200 I flame, you flame, we all flame for iFlame.
Request : LOC 36356 36612
Command : 200 Location set.
Request: AUTH PASSWORD
Command : 200 Authentication type set.
Request: USER dml
Command : 200 User set.
Request: PASS mypass
Command : 200 Password accepted.
```

```
Request : +w iflame_help
Command : 200 Now writing forum.
```

The above process is similar to a client sending a `+r` command to the server. The client has determined that it is not currently writing to the `iflame_help` forum so it needs to subscribe as a writer; if successful, the server will add this client to the forum's writers cache and then update the caches of other writers for that forum. Notice that the server does *not* need to update the caches of forum readers, since they do not send any data but only accept connections from other clients. When the client receives `200 Now writing forum` the writer subscription was completed successfully and the client waits for a cache update in order to send data to `iflame_help`.

6. The server updates the new writers cache

```
iflamec: Connection from : alphaville.media.mit.edu 37391
Reply   : 200 I flame, you flame, we all flame for iFlame.
Command : +F iflame_help 18.23.1.128 36356
CACHE:
```

This section demonstrates how a client receives a cache update. A connection is established from another machine, `alphaville.media.mit.edu` on remote port `37391`. In our implementation, the client checks to see if it should be accepting connections from this machine on its server port by examining a cache of allowed servers. Finding `alphaville.media.mit.edu` in this cache, the client allows the connection to succeed and receives the command `+F iflame_help 18.23.1.128 36356`. Again, we display the local cache which, since this update hasn't been handled yet, is still empty.

7. The client accepts the cache update

```
CACHE:
```

```
cache key: iflame_help
cache data: 18.23.1.128 36356

Reply   : 200 Cache update successful
Command : QUIT
Reply   : 200 Quit: command complete, closing connection
```

This transaction illustrates a client updating its local cache. In our implementation, we use simple red-black trees as the cache structure; for the client's forums, the key is simply the forum name and the associated data are the IP addresses and connect ports of the clients reading the forums. Since the client only received one update the cache only contains one location (which happens to be itself, since it is the only reader).

When the cache update has succeeded, the server sends a `QUIT` command. Had there been other readers, the client would have received additional `+F` commands designating other cache updates.

8. Sending data to a forum

```
Sending location info...
GCing connection
GCing connection
iflamec: Connection from : slip-dml.lcs.mit.edu 37892
GCing connection
To: iflame_help
From: dml
```

Having extracted the reader clients for `iflame_help` from the client's local cache, the client connects to each location in turn and sends the data. Once a connection from another client is received,[13] the data is sent from the local user-level program to the remote machine. The `To:` and `From:` fields are here displayed here only to show that this data is included in the header of the data that is sent. If the client was

[13]In this case, however, it is our own user-level program

being run in terminal mode these would appear at this point along with the actual text of the message. In this case, however, we are using the implementation's display code, so the remote client forwards the received data to that subsystem.

9. The displayer for the X Window System is initialized

```
---Executing: idisplay
Error: couldn't open unix socket
Attempting to start displayer....
iwindow: Using unix socket: /tmp/iwindow.Slip-Dml17213.0
iwindow client initialized
Reply   : 200 I flame, you flame, we all flame for iFlame.
Using unix socket: /tmp/iwindow.Slip-Dml17213.0
idisplay client initialized
```

Since the client is using the X display code described in Chapter Three, it passes the incoming data to the `idisplay` program which uses `metamail` to decode the MIME-encoded message; the `iwindow` display program is also executed since one is not already running and the transmitted text data is displayed in a small Tk window on the screen. Again, this is described in detail in the following chapter.

10. Data is displayed on the X terminal

```
Sending location info...
GCing connection
iflamec: Connection from : slip-dml.lcs.mit.edu 38148
GCing connection
To: iflame_help
From: dml

---Executing: idisplay
Reply   : 200 I flame, you flame, we all flame for iFlame.
Using unix socket: /tmp/iwindow.Slip-Dml17213.0
idisplay client initialized
GCing connection
```

This last segment demonstrates how the client is able to remove load from the server in the common case. The server is loaded only when the state of the system changes; otherwise, the clients can operate independently from it. The user has decided to send another message to the `iflame_help` forum, but since no other users have joined or left the forum, the local reader cache has not been updated (thus the reason no other server connections are evident). We take the short client-to-client path by simply sending the remote locations to the user-level application which then sends the data to the remote clients, as described above.

2.2.2 Analysis and Advantages

A protocol like iFlame has numerous inherent advantages over server-centric systems. In the common case, the datapath does not involve any servers so the clients bear the entirety of the load. Even when changes need to propagate from a server to its clients, the overhead of sending such messages is minimal in comparison to the work a server would have to do to not only maintain this information but to route messages as well.

The overhead of the iFlame protocol resides in cache updates. When a forum is composed of a small number of locations, the assumption is not only that most of the readers are also writers but also that there are few new readers joining and old readers leaving. Even if the forum is fairly volatile, the number of clients that must be contacted for cache updates is still small so the overhead generated by these updates is almost non-existent. In larger forums that tend to be more volatile by nature, the overhead of cache updates is still acceptable since the amount of data that must be managed and sent to clients is relatively small in comparison to the data that would otherwise have to be routed through a network of relayed servers.

Command Syntax	Description	Return Code
LOC <client connect port> <server connect port>	Location; identifies the connecting client to the server.	200 OK, 400 memory error or repeated command, 500 argument failure or bad port.
USER <user string>	User; identifies username.	200 OK, 400 memory or database failure, 500 argument failure.
AUTH <authentication string>	Authentication; identifies authentication scheme.	200 OK, 400 type already set, 500 unknown type or argument failure.
PASS <password>	Password; sends cleartext password.	200 OK or not using PASSWORD type, 500 no user set or argument failure.
+/− **r** <forum string>	Add/Delete reader	200 OK, 400 memory error, 500 location not set, access denied, or argument failure.
+/− **w** <forum string>	Add/Delete writer	200 OK, 400 memory error, 500 location not set, access denied, or argument failure.
+/− **a**[**r**,**w**,**a**] <forum string> <user string>	Add/Delete `READ`, `WRITE`, `ADMIN` ACL	200 OK, 400 memory error, 500 access denied or argument failure.
+/− **an**[**r**,**w**,**a**] <forum string> <user string>	Add/Delete `NO_READ`, `NO_WRITE`, `NO_ADMIN` ACL	200 OK, 400 memory error, 500 access denied or argument failure.
QUIT	Quit, end data transmission	200 OK, 300 request connection, 500 argument failure.

Figure 2-5: Outgoing client → server messages

Command Syntax	Description	Return Code
+/− **F** <forum name> <ip address> <client connect port>	Add/Delete forum from client cache	200 OK, 500 argument failure.
QUIT	Quit connection	200 OK, 500 argument failure

Figure 2-6: Outgoing server $\rightarrow$ client messages

Chapter 3

The iFlame Message System

The iFlame Message System is an implementation of a communication system built using the iFLAME protocol. It illustrates many of the advantages of a client-based system while combining the flexibility of various net-aware applications. The result is a chat mechanism that allows users to exchange audio, text, video, and other types of data in real-time.

3.1 Client Implementation

The iFlame Message System client corresponds to the client described in Chapter 2; that is, it is the half of the system with which the user interacts that sends data to other clients and receives cache updates from servers. The client consists of a networking kernel and two display components. The kernel is the portion of the client that talks to remote clients and servers; it itself is of a low-level network piece (`iflamec`) that maintains cache information and talks to servers as well as a higher-level application (`iflame`) that communicates with the user and delivers messages to other remote `iflamec`'s. In addition to the basic client the Message System adds two display components: `idisplay` and `iwindow`. The display programs handle incoming data, parse it, and display it graphically. The implementations of each of these subsystems are described below.

3.1.1 `iflamec` and `iflame`

`iflamec` is the system's cache maintainer, display handler, and authentication overseer, whereas `iflame` is the user interface to `iflamec`. Users are able to type commands to `iflame` which it interprets and uses to join or leave forums or to send flames to other users. We first describe the initialization phase of each system and then detail how the two programs work together to implement the iFLAME protocol. C code (in which the system is written), included in Appendix A, is referenced to illustrate how particular portions of the protocol are implemented.[1] In Section 3.1.4 we describe the display programs `idisplay` and `iwindow`. Coordinating with the display programs `iflamec` and `iflame` create the entire iFlame Message System.

Data Structures

State within `iflamec` keeps track of the forums to which a user is subscribed (either as reader, writer, or both) and the locations associated with those forums. Forums are cached locally as collections of the locations subscribed to them using red-black trees.[4]

`iflamec` Initialization

The initialization of the client is a three-step process: clear the caches, initialize sockets, and then configure the local user environment.

We tend to search caches in the iFlame Message System far more often than we insert or delete into or from them, thus we want to implement caches using a data structure that is optimized for searching. For our purposes red-black trees are ideal; Our red-black tree structure is shown in Figure 3-1.

The client maintains four caches, described in Figure 3-2. Each is initialized to be empty.

[1] C was chosen as the implementation language because portability is extremely important if we wish to have as many people using the system as possible. C was chosen as the implementation language because there are implementations of C for virtually every platform we wish to support that allow us the versatility we desire.

```
typedef struct tree
{
  void *key;
  void *data;
  struct tree *left;
  struct tree *right;
  struct tree *parent;
  int color;
} tree_t;

void inorder_tree_walk(tree_t *x, void (*f)(tree_t *));
tree_t *tree_search(tree_t *x,
                    void *k,
                    int (*compare)(const void *,const void *));
void rb_insert(tree_t **T,
               tree_t *x,
               int (*compare)(const void *,const void *));
tree_t *rb_delete(tree_t **T,
                  tree_t *z,
                  int (*compare)(const void *,const void *));
```

Figure 3-1: Implementation of red-black trees in iFlame

`iflamec` is connected to `iflame` via a Unix socket. Shared memory was considered as a possible implementation mechanism for this connection, but Unix sockets were chosen for the client end to preserve portability and minimize complexity. `iflamec` also has a number of TCP sockets: one for incoming connection from clients, one for incoming connections from servers and one for outgoing connections to servers. Socket initialization is fairly straightforward, and uses standard network interface code; in Appendix A contains detailed code listings.

On the `iflamec` side, an open socket is managed as a *connection*. A connection is simply a structure that contains information concerning the socket's file descriptor, its active state in the dispatch loop, and storage space for incoming and outgoing data. The connection structure is illustrated in Figure 3-3.

The local user environment is read from configuration files located in a direc-

Cache Name	Description
readers	Local forums client is reading
writers	Forums client is writing and their members
aliases	Forum aliases read from configuration files
servers	Hosts client allows server commands from

Figure 3-2: `iflamec` caches

tory of the user's choice. `iflamec` uses one of these files, `.iflamec`, which contains information about the client's default server, username, possible authentication information, and forum aliases. The format of all variables except for aliases consists of the variable name and the data separated by an "=" character. If not specified in the `.iflamec` file, these values can also be established as environment variables. An example `.iflamec` file is shown in Figure 3-4.

Main Dispatch Loop

The iFlame dispatch loop is a large finite state machine that uses multiplexed TCP/IP to handle multiple connections from different remote hosts.[2] Once a connection has been accepted, arbitration of work is decided by a standard `select()` system call. The `iflamec`-specific setup for this call is depicted in Appendix A.

Concurrency

A large part of engineering the client was designing and building the I/O system. The client maintains a table for all file descriptors associating each open file descriptor with a connection structure. This system appears complex when compared to the interface to a threads package but it was felt that in this case portability was more important than code simplicity. In retrospect, we made the right choice, as the system was easier to build and debug than had been expected. Time was also saved by not

[2]The server has precedence over remote clients and the Unix socket.

```
typedef struct connection
{
  int state;                      /* state we left connection in */

  time_t connection_time;         /* connection start time */
  time_t command_time;            /* current command time */

  char in_buf[CHAR_BUF_SIZE];     /* input line buffer */
  int in_cnt;                     /* index into input buffer */
  char out_buf[CHAR_BUF_SIZE];    /* output line buffer */
  int out_cnt;                    /* index into output buffer */
  int extra;                      /* checked extra input flag */
  int old_fd;                     /* last fd */

  queue_t queue;                  /* client read buffer queue */

  } connection_t;
```

Figure 3-3: The `iflamec` connection structure

having to seek out, evaluate, and comprehend the various issues in current user-level thread systems.

States within the finite state machine are divided into two subsets: *read* states and *write* states. For the most part, read states filter into write states and vice-versa. A general diagram of the state machine can be found in [FIGURE NAME HERE]; the I/O code used to control the machine is demonstrated in Appendix A. Use of a concurrent state machine was required by the dynamic nature of the remote clients. Since clients and servers could go up or down at any time, a single-threaded model would halt other work while timing out on crashed systems. This was unacceptable behavior for the iFlame Message System.

```
iflame_server=iflame.media.mit.edu
iflame_username=dml
iflame_signature=David M. LaMacchia
iflame_password=mypass
iflame_authentication=PASSWORD
alias bdc bdc@iflame.mit.edu
```

Figure 3-4: Sample `.iflamec` initialization file

3.1.2 `iflame` Architecture

The `iflame` program acts as a user interface to `iflamec` and is responsible for sending flames to remote clients. The interface is easy to use and allows users to join or leave forums, set ACLs for forums, and send flames.

`iflame` Initialization

Initialization of `iflame` differs from `iflamec` in that we have only one socket to bind, the Unix socket. The `.iflamec` file is used here to determine certain variable definitions but we also read an `.ianyone` file (if it exists) that allows users to locate other people online. Otherwise initialization only consists of parsing the command line for various switches and options.

User commands

Figure 3-5 shows the command-line options `iflame`. The user types "`iflame command-name` $\texttt{arg}_1$ `...` $\texttt{arg}_n$." For the most part, the `iflame`'s behavior in handling the commands listed in Figure 3-5 is not interesting; adding the client as a reader and making ACL modifications, for example, require that `iflame` simply send the appropriate command for each forum (or user) listed to the client, one at a time. The client then sends these commands to the server and makes all appropriate cache updates.

User Command Syntax	Description
$+/-$ **r** $< forum_1 > ... < forum_n >$	Add/Delete client as reader to/from forums 1 through n.
$+/-$ **ar** $< forum >< user_1 > ... < user_n >$	Give/Remove `read` ACL for users 1 through n for named forum.
$+/-$ **aw** $< forum >< user_1 > ... < user_n >$	Give/Remove `write` ACL for users 1 through n for named forum.
$+/-$ **aa** $< forum >< user_1 > ... < user_n >$	Give/Remove `admin` ACL for users 1 through n for named forum.
$+/-$ **anr** $< forum >< user_1 > ... < user_n >$	Give/Remove `deny read` ACL for users 1 through n for named forum.
$+/-$ **anw** $< forum >< user_1 > ... < user_n >$	Give/Remove `deny write` ACL for users 1 through n for named forum.
$+/-$ **ana** $< forum >< user_1 > ... < user_n >$	Give/Remove `deny admin` ACL for users 1 through n for named forum.
$+/-$ **l**	locate members of forums listed in `.ianyone` file
$< forum_1 > ... < forum_n >$ [$-$**f**$filename$]	Send data to forums 1 through n. If `filename` is specified, read data from there, otherwise from the console.

Figure 3-5: `iflame` user commands

Locating forums with users online

Often a user may with to know whether one of his or her friends are currently online, or perhaps the user wishes to know who is subscribed to a particular forum.[3] In the iFlame Message System, these tasks are accomplished through use of a file that simply contains names of forums in which the user is interested. When the user gives the `+/- l` command to `iflame`, the `.ianyone` file is examined and a list of subscribers to each forum is given to the user. Of course, the user can find out this information only if he is able to access those forums according to their particular ACLs. When a client tries to locate who is reading a forum, `iflamec` sends a `+w` command for each forum to the server each forum is located on, subscribing the client as a writer

[3] This service exists for MIT Athena's Zephyr system and is extremely popular.

without sending any actual data. The server, if the client is allowed to add itself as a writer, will automatically send cache updates to the client just as if a new writer had been added. If the client is already a writer, then `iflamec` can simply report the current contents of the writer cache for that forum, as it contains who is currently marked as a reader.

3.1.3 Flamage

Flamage, the sending of messages between clients, is the central purpose of the iFlame client and its implementation is straightforward. While all messages are sent from a user's `iflame` program directly to remote users' `iflamec` clients in a point-to-point fashion, there is also significant interaction between the local `iflame` and `iflamec`.

`iflamec`'s main dispatch loop handles incoming client and server connections. When a user desires to send a flame to a forum, `iflame` checks with `iflamec` to see if that forum exists within its local cache. If so, `iflamec` enqueues the list of locations of readers of a forum in a single package and transfers this back to `iflame` which, as locations are dequeued, sends a copy of the message finger to each remote client. If the forum is not present in the local cache then the client is not yet a writer and so has to subscribe and receive cache updates from the server. The forum name is parsed (in case the remote server is not the default one) and a connection is opened to the server. In terms of the implementation, we enter a special set of states that commit these transactions.

Once locations have been received by `iflame`, the data must be enqueued and sent to each receiving client. Data to be sent is first encoded using the MIME standard; in this implementation `iflame` forms a valid RFC-822 header scheme and adds a `Content-type:` header (for textual data we define a new MIME type, `text/iflame`). The data is then sent in 2048-byte (POSIX-2 standard) packets to each receiving client.

On the receiving end of a flame, the remote `iflamec` accepts an incoming TCP connection from `iflame`. The local `iflamec` can continue to receive flames while `iflame` is sending other messages. Well-formed messages are passed to the display manager.

Once data is received, `iflamec` calls `metamail` (this time on the receiving end) to decode the incoming data. `metamail` uses a `.mailcap` file, usually located in the user's home directory, to determine how to display the particular type of data on the local machine. If the `Content-type` header specifies type `text/iflame`, typically the data is sent to the `idisplay` program.

3.1.4 `idisplay` and `iwindow`

`idisplay` is a fairly simple program; it establishes a Unix socket for communicating with `iwindow` and starts an `iwindow` program if one is not already running. Of course, data is only sent to `idisplay` if, as mentioned above, it is of MIME content type `text/iflame`.

`iwindow` uses Tcl/Tk as a scripting language for its display procedures; users can write their own display routines in Tcl/Tk should they so desire. On initialization `iwindow` starts a Tcl/Tk interpreter and reads the user's `.iflamec.tcl` file. A sample `.iflamec.tcl` file is displayed in Appendix A.

`iflamec` supports the ability for users to extend its basic functionality via this embedded scripting language, much in the same manner as other message programs like Zephyr. Because many people already use Tcl and there are many good examples of Tcl applications, we decided using Tcl as our extension language would be optimal. Furthermore, we use Tk as the default language for graphical displays to provide the same sort of flexibility. iFlame also supports a terminal mode for non-graphical displays.[4]

The use of Tcl/Tk, while having many advantages, has some potential drawbacks. Tcl allows a friendly configuration environment, but Tk may be slow in comparison to direct access to the X Window System toolkit. While this performance issue will be studied in the future, the extensibility of Tk is a great advantage in allowing others to write customized iFlame clients.

[4] `iwindow` currently uses tcl7.3/tk3.6 and will be upgraded to tcl7.4/tk4.0 in the future.

Once the initialization is complete, if `iflamec` was not run with the `-ttymode` argument (specifying terminal mode), then `iwindow` forks off a child process and establishes a bidirectional pipe to send data down to the on-screen windows. The result `iwindow` achieves is a look and feel similar to Athena Zephyr; the decision to do this stemmed from the fact that Zephyr's "windowgrams" are not as distracting to the user as a `talk` or IRC connection is. An iFlame session does not require the user to continually concentrate on the various forums to which he is subscribed. Windows with short messages can appear anywhere on the screen[5], although the default is to place new messages in some out of the way location, such as the top left or right corner of the screen. As the default, we also have windows go away when the user clicks on them with the mouse, again to mimic the look and feel of Zephyr.

In order to accomplish management of lots of window in an efficient manner, `iwindow` needs to have the child process manage all the windows while the parent process receives data from `idisplay` and sends it down the pipe. Each window has its own Tcl interpreter to determine when the ButtonDown X event is sent to it so the window can be destroyed[6]. We use a user-defined signal to tell the parent process when the child process is ready to display new data. Future versions of `iwindow` will probably not use Unix signals as they are not reliable.

The display of a typical iFlame Message System X session is shown in Figure 3-6

[5]As defined in the Tcl script in `.iflamec.tcl`

[6]Though, again, this behavior is just the default and can be changed in `.iflamec.tcl`

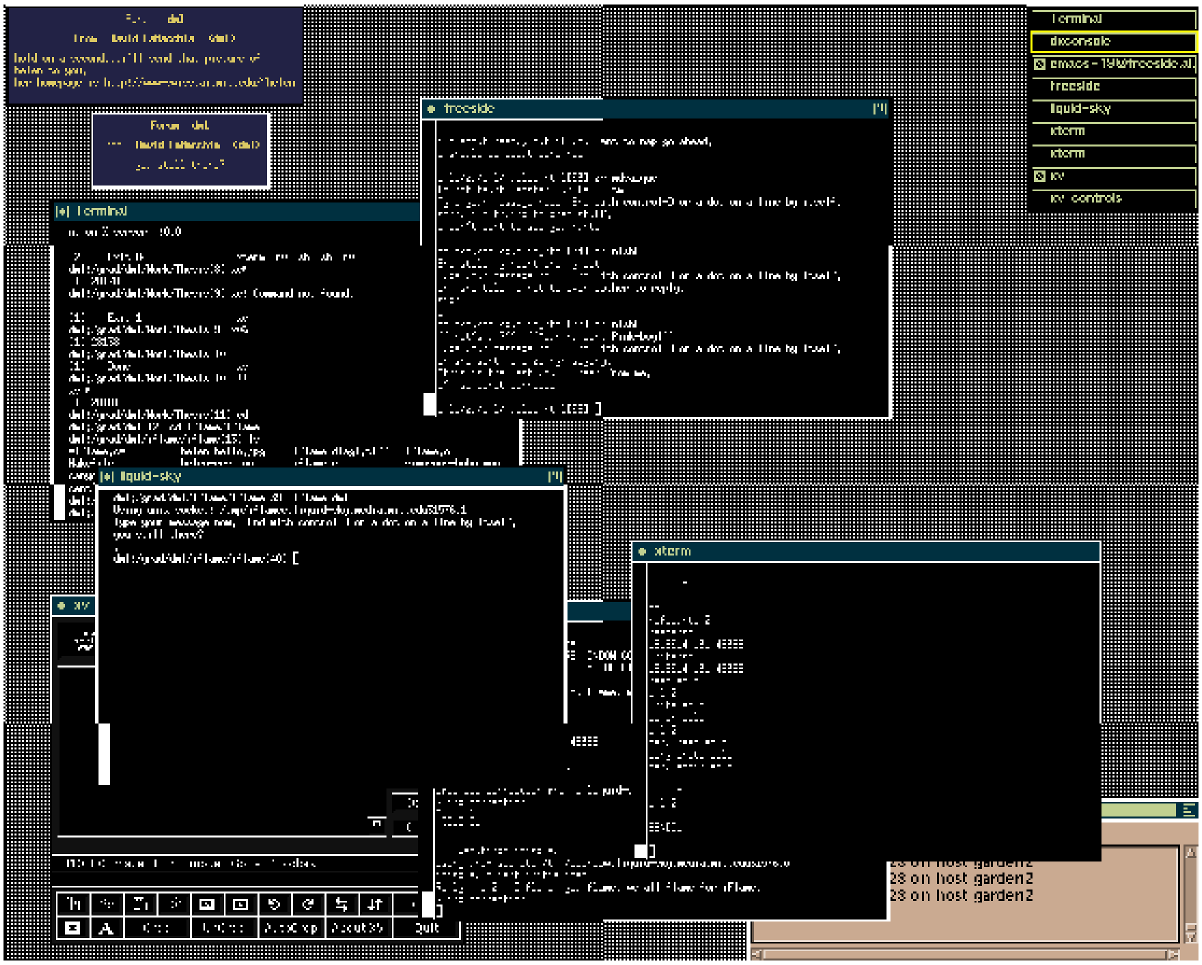

Figure 3-6: A Typical iFlame Message System Screen Dump

3.2 Server Implementation

This section describes the current implementation of the `iflamed` server. The server runs as a monolithic process that both stores the iFlame forum database and also manages concurrent connections for incoming transactions and outgoing cache updates. In addition, `iflamed` also hosts the user authentication database. Like `iflamec`, `iflamed` uses a multiplexed TCP/IP concurrent state machine design.

3.2.1 `iflamed` Initialization

`iflamed` initialization is slightly more difficult than `iflamec` initialization because the `iflamed` server needs to interact with the user authentication database in a robust, corruption-proof manner. Upon startup, `iflamed` empties its caches, establishes a socket for client connections, and then reads the database for personal forum information. Cache handling and socket creation is done in the same way as described for `iflamec`.

Authentication

At the moment, the iFlame Message System supports only two authentication schemes, `UNAUTHENTIC` and `PASSWORD`. `UNAUTHENTIC` means that no user name is required to join forums; however, this means that the client cannot alter a forum's ACLs. An unauthenticated user can still create forums, but any such forums so created are public and have no administrators. `PASSWORD` authentication, on the other hand, requires the client to have previously established a (user name, password) pair with the home server. Once a password authentication is complete the user is never required to reauthenticate or supply passwords during that particular connection. One problem with this system is that the password is currently sent from the client to the server in plaintext over the network. Thus, a packet sniffer could conceivably trap those packets, extract the (user name. password) pair, and then masquerade as another user. A better authentication scheme is needed to thwart this attack, perhaps using either MIT Kerberos or a Pretty Good Privacy (PGP) key exchange. We are currently examining the possibility of incorporating PGP into an authentication scheme for a future version of iFlame since the keys used for authenticating could also be used for a transparent encryption scheme as well.

The `iflamed` Database

For now the `iflamed` database is a simple relational database implemented using `gdbm`, the GNU database library, as a substrate. `gdbm` was chosen for our implementation because it allows more flexibility and better hashing than standard Unix `dbm` library for large numbers of users. In the future, we may move to `dbm` as it is more generally available with standard C library packages.

A small program called `iserv`, running on the server itself, allows the server operator to maintain the contents of the database, including user data, passwords, and PGP information[7], and other authentication data. By default, when a new user is added to the database a forum is created and the ACL for that forum is set to a personal forum, giving `admin` and `read` permissions to that user only. On initialization, `iflamed` loads the database and creates a new forum in its cache for each user located in the database; should the server crash during operation, personal forums residing on that server are immediately established when the server restarts. Clients using any authentication scheme (except `UNAUTHENTIC`) are required to contact the database and authenticate themselves each and time a new connection is opened to the server.

3.2.2 Data Structures

`iflamed`, like `iflamec`, uses a finite state machine to regulate connections and maintain caches of forums, locations, users, and user relationships. These caches are currently manipulated using red-black tree routines, just as was done with `iflamec`. Although the server supports simultaneous, concurrent connections, all server opera-

[7] PGP key sharing support has, however, been left for a future version of the system.

tions on the data structures are processed atomically and in sequential order.[8]. The atomic operations allow for cleaner code and reduce the complexity of the server.

The server is expected to be long lived, and thus special care was taken when writing all resource-allocation code (such as memory management routines) to prevent resource leakage. For debugging purposes, we maintain various counters that track the number of server structures supposedly accessible and compare these values to what is reachable from walking the data structures. Reference counts are used to control deallocation of structures. Although we could implement a garbage collector and walk the data structures periodically, for now routines that decrement reference counts just check locally released data for deallocation.

3.2.3 Scheduling Client Cache Updates

One final interesting server design note concerns its cache update queue. Instead of actively opening multiple outgoing connections for updating client caches, the server processes cache updates sequentially using a single, global queue. We used a sequential process herein, an attempt to minimize server load; the penalty we incur is a slight degradation in forum consistency. Since it is difficult to guarantee how quickly cache updates are distributed, this doesn't seem to be a major issue. The benefit of the queue system is that multiple updates for a single client combine themselves and thus reduce the number of required server-client connections. Also, if a client's transactions have completed and an outgoing message for that client exists on the queue, the connection can be turned around so that the client immediately receives cache updates without having to establish a new connection between client and server. As many updates can be generated when a client's cache is empty, such as when first joining a forum, this reduces the number of connections a server must make to a particular client. Similarly, when the server detects that a client has disappeared from the net while attempting a cache update, the server can free up certain allocated resources and notify other clients to stop writing to the lost client.

---

[8] The same is true of the `iflamec` client. The original model didn't support concurrent connections yet all the operations were atomic

Chapter 4

Problems, Analysis, and the Future

We have described in previous chapters a protocol that can be used to create Internet communication environments that are both extremely scalable and easy to implement. We have described how the iFlame protocol is able to provide these features, a claim that cannot be made by other recent Internet communication systems. We have also described a particular communications system, the iFlame Message System, that uses the iFlame protocol to provide users both the ability send arbitrary data (including audio, video, images, and text) and also provides fine-grain control over the messaging environment through a fully-developed and familiar scripting language. In this chapter we examine examine possible problems with both the iFlame protocol and the iFlame Message System, suggest changes for future versions of both, and speculate on how iFlame might evolve over time.

4.1 Problems

Problems with the iFlame Message System can be divided into two broad areas: general technical problems related to the underlying iFlame protocol and specific problems encountered while creating the iFlame Message System. Both areas are addressed in this section.

4.1.1 iFlame

Delivery and Fan-out

Unlike a system like Athena Zephyr, iFlame does not use UDP to deliver messages but instead relies on TCP connection. TCP, in general, has a higher overhead for each connection established between clients than UDP. In addition, fan-out is low for the client since workload grows linearly with number of readers on a forum. These constraints are acceptable given that our primary goal is to reduce server load.

4.1.2 The iFlame Message System

Firewalls

Perhaps the largest problem with the current implementation of the iFlame Message System is that it does not account for users located behind firewalls. This unfortunately denies Message System access to a vast population of corporate and government users. The problem is simple: firewalls in general restrict the ability for clients inside the firewall to talk to arbitrary clients outside the firewall. Some firewalls, for instance, do not allow clients to use FTP with outside servers because outside servers cannot open arbitrary connections to the local machine. Incorporating the iFlame Message System into a firewall-protected environment is an active area of research; clients may have to alert other clients of their presence via a common server.

The Display System

There are several problems with the current iFlame display system (`idisplay` and `iwindow`). First of all, the system relies on Unix signals to coordinate the parent and child processes when new data is received; in general, these signals are unreliable. Furthermore, the display system requires that the user have access to an external MIME decoder; while the decoder used in the implementation, `metamail`, is available for many platforms, it would be more efficient to include the encoding/decoding routines in the client to reduce the number of external system calls.

Support and Availability

The implementation of the iFlame Message System discussed in Chapter 3 was written to be highly portable; such portability was a primary design consideration.[1] However, at this time it is unclear if the implementation was entirely successful in meeting this goal. We hope that beta testing with hundreds of users on many different platforms, instead of the relatively few used to develop the system, will illuminate portability problems in the current implementation.[2] We recognize that the iFlame Message System must run on popular platforms like Microsoft Windows and the Apple Macintosh if it is to gain widespread acceptance.

4.2 Analysis

As it turns out, the iFlame Message System is at least as efficient as a server-centralized system like IRC or Zephyr. Four tests were run to compare the performance of iFlame and Zephyr, the system that iFlame most closely resembles in appearance and functionality. We now detail each of the performed experiments and discuss the results. Each test was run on an HP 735 running HP/UX.

4.2.1 Small message, continuous conversation

In the first test, a small two-line message, *"This is a test of the American broadcasting system"*, was sent 200 times from a local client to a remote client. The test was performed 10 times and the average result calculated. This test was designed to exercise iFlame's strengths in being a client-client protocol, since once the local client was established as a writer there would be no further interaction with the server. Recall that like iFlame, Zephyr was designed for sending small, real-time messages. The results of this test are given in Figure 4-1.

[1] We often sacrificed ease of implementation for portability, as in our decision not to use a threads package.

[2] At the moment, the versions of the Message System operate under HP/UX 9.01, Digital Unix (Alpha OSF/3), NetBSD, and Solaris.

iFlame performed much better than Zephyr in this test, which is somewhat surprising given Zephyr's slated design goals. There are possible explanations for the discrepancy in performance. For example, the Zephyr server we were using while conducting the tests could have experiencing an uncommon load, while the iFlame server was not. This could have also been true of the client machines. It is difficult to keep many variables from severely affecting tests in a dynamic networked environment. It should also be noted that in this trial both Zephyr and iFlame were running in terminal mode, so the X server's own latency would not affect the results.

Protocol	System time to send 200 short messages
iFlame Message System	3:17 minutes
Zephyr Notification Service	5:45

Figure 4-1: Test 1: 200 short messages, no subscriptions, average of 10 trials

4.2.2 Large message, continuous conversation

In our second test, we increased the size of the message being sent to observe whether performance is highly dependent on message length. Zephyr supports up to about 10 lines of 80 characters each[8] so a 700-character message was sent in a manner similar to the first trial. Again, we averaged our results over 10 trials. The results of this experiment are depicted in Figure 4-2; they do not differ significantly from the "short message, continuous conversation" test results.

Protocol	Time to send 200 long messages
iFlame Message System	4:04 minutes
Zephyr Notification Service	5:55

Figure 4-2: Test 2: 200 long messages, no subscriptions, average of 10 trials

4.2.3 Small message, subscription messages sent

Our third experiment emulated a conversation of 200 messages between two people within a single iFlame forum/Zephyr class. After every ten messages the sending client would unsubscribe and then resubscribe to the forum, thus emulating subscription traffic. The results are presented in Figure 4-3.

Protocol	Time to send 200 short messages
iFlame Message System	4:40 minutes
Zephyr Notification Service	6:22

Figure 4-3: Test 3: 200 short messages, plus subscriptions, average of 10 trials

4.2.4 Large message, subscription messages sent

Running experiment three with large messages did not appreciably change response time, as show in Figure 4-4.

Protocol	Time to send 200 long messages
iFlame Message System	4:54 minutes
Zephyr Notification Service	6:34

Figure 4-4: Test 4: 200 long messages, plus subscriptions, average of 10 trials

4.2.5 A Comparison Between iFlame, IRC, and Zephyr

An overview of the similarities and differences among IRC, Zephyr, and iFlame is shown in Figure 4-5. We intentionally duplicate the format of a similar comparison chart showing similarities and differences between Zephyr and email taken from *The Zephyr Notification Service*[8].

4.3 The Future of iFlame

In this thesis we have described the iFlame protocol and implemented a demonstration communication system that used the protocol to provide a scalable and portable service. There are several directions in which that future work on iFlame could take. We discuss three possible directions below.

1. Improvements to the iFlame Message System
2. Migration of iFlame to an open system
3. Other implementations of the protocol

4.3.1 Improvements to the iFlame Message System

As noted throughout this thesis, there are several areas where the iFlame Message System could be improved. First and foremost, we need to support clients located behind firewalls; firewalls have become too prevelent in recent years, protecting too many Internet users to be ignored. In order to support communication through firewalls, we would most likely need to develop a proxy server for the firewall that could route iFlame messages. Unfortunately, introducing such proxy servers hinders our clients from being able to contact each other directly, thus muting the system's ability to deliver efficient communication. Without some sort of proxy service, though, iFlame communication cannot cross firewalls.[3]

Another problem with our implementation is that currently-supported authentication schemes are neither sufficiently secure nor varied. iFlame should move towards other authentication protocols, perhaps using a high-grade digital signature, a Diffie-Hellman encryption key exchange, or even a Kerberos mechanism (each server could maintain its own Kerberos realm). Use of a system like Pretty Good Privacy (PGP) for authentication would be optimal since such support in the client could be easily leveraged into providing encryption for messages if the user desires. There are

[3]Notice, however, that within the "intranet" behind a firewall the iFlame Message System operates normally.

presently hooks for handling PGP keys in the server's database code in anticipation of this evolution.

4.3.2 Migration of iFlame to an open system

The iFlame Message System is, at the present time, a closed system in that users can only communicate through it to other users of the Message System. While it would not be difficult to add gateways between iFlame and electronic mail or usenet news[4], there is little reason to support such asynchronous communication methods in the iFlame synchronous environment. Eventually, however, we would like to see interfaces to systems like IRC and (more importantly) the World Wide Web. Not only do we want the iFlame Message System to perform the simple task of retrieving and parsing hypertext, but we also want iFlame to interface with network agents that perform tasks for the user. For example, suppose an iFlame user wishes to find the top ten documents listed in AltaVista[1] concerning the iFlame Message System. That user should be able to send the query via iFlame to a local robot perform the requested query and, when completed, send the results via iFlame to the user.

4.3.3 Other implementations of the protocol

Of course, chat systems are only one class of services that can be implemented on top of the iFlame protocol; we chose the Message System as our demonstration/proof-of-concept because it allows us to both determine if the protocol was scalable and also compare it to other common real-time communications systems fairly easily. There are several other directions, however, in which iFlame could be extended in the future. For example, real-time networked games currently use client/server architectures to share state; the server maintains all sorts of information about all the clients and each client only receives data it needs. A protocol like iFlame may be a more efficient means of communicating information; once a game is established

[4] Notice that these services are not instantaneous and would not make use of iFlame's ability to carry out real-time conversations.

among clients, the clients directly contact each other to send data. If another client wishes to dynamically enter the game, it contacts the server being used for location information; the server then updates the caches on clients already in the game thus allowing the new client to join. Efficient networked games are but one application that could benefit from iFlame; it is easy to see how iFlame could be used to improve a wide variety of applications including simulations, banking, sales/retail transactions, audio/video conferencing, and many others.

4.4 Conclusion

The iFlame protocol was designed as an evolutionary step in real-time communication. For many years synchronous transactions have fallen by the wayside as more efficient asynchronous traffic has taken the industry's favor. Recently, however, users have demonstrated a desire for better support for and emulation of real-time conversations, a task for which traditional store-and-forward systems are unsuitable. In order to foster delocalized communities founded on real-time conversation, a protocol with a high level of scalability and efficiency is required. To date, none of the synchronous communication tools available on the Internet have the necessary scalability, efficiency, accessibility, and concurrency to make them viable options for serious delocalized, real-tme communication.

The iFlame protocol described in this thesis provides a substrate for scalable systems on the Internet. iFlame accomplishes this task by being inherently client-oriented, building a distributed system by shifting as much work as possible from centralized servers to local clients. Transactions are, in general, directed in a one-to-many, client-to-client fashion. The specific implementation of the protocol that we detail several tools to construct a robust, efficient, portable communication system on top of the iFlame protocol.

Until recently, the small number of users on the Internet, the general lack of available bandwidth, and the low power of networked workstations together forced online communication to evolve into a form that was centralized around a few powerful machines. In the future, the trends that now allow more people to gain online access, with more powerful computers and greater bandwidth, will make better forms of communication possible. The demand for better protocols to support real-time communication among arbitrary groups and users is already here and iFlame is a first step towards making distributed real-time communication ubiquitous.

Metric	iFlame	Zephyr	IRC
Addressing	Explicit. Addressing for forums meant to be long-lived is static and is handled by a particular server. Addressing is one-to-many in that the user doesn't have to name each recipient.	Implicit. Addressing is one-to-many and handled by the server. A specific user can be messaged through use of a long-lived ID.	Implicit. It is not necessary for a user to know the location of another user. Channels allow users to send messages in a one-to-many fashion.
Delivery	Messages are delivered by a client directly to other clients. A user is responsible for doing the work required to send each message. A user always knows who messages are being sent to.	Messages are sent by servers. A user sends a single copy to a server which duplicates it. A user sending to a class can inadvertently send data unawares to unannounced users.	Messages are sent by relayed servers. A user sends a single copy which is duplicated by these servers. A user always knows exactly who a message is being sent to.
Messages	Messages are MIME-encoded binary data; audio, video, text, images, and other formats of unlimited size can be transferred as long as the client is willing to send it.	Short, fixed, text messages of about 10 lines, each line containing about 80 characters.	Text-only, messages are in general a few lines, with a maximum of 510 characters allowed for commands and parameters.
Message Fan-out	Low. Sending to large lists is inefficient for the client. Each client generates a copy of a message for each remote client it connects to.	High for client, low for server. Client only needs to generate one copy of a message. Replicated servers each generate a message for each client they send to.	High for client, low for servers. For each client receiving a message, a server somewhere in the network must generate a copy of that message. Every server must receive a message, even if no clients will receive it from that server.
Group Persistence	High. Forums can exist over long periods of time with a set of established ACLs and administrators.	Low. Users are persistent because authentication information resides on a Kerberos server. Classes and instances have no persistence.	None. Unless users are subscribed to a channel, it doesn't exist. Users have no persistence from session to session, unless outside "nick servers" are being used.
Configurability	High. Users have access to high-level Tcl scripting and can dynamically change the environment.	High. Zephyr's built-in scripting language, while not widely used outside Zephyr, is powerful.	Low. While users cannot change their environment, they can create complex scripts to perform tasks and create robots to carry these tasks out.
Maintenance	Medium. The server must keep track of authentication information in its database. Both server and client can dynamically recover lost resources.	Medium. While the server has little information to keep track of, the Kerberos server must handle authentication.	Low. Since there are no persistent subscriptions, the server has little to maintain.

Figure 4-5: A comparison of iFlame, Zephyr, and IRC

Appendix A

Code Samples

```
void initialize_sockets(void)
{

  int count = 0;
  int i;

/* Incoming Client Socket */
  addr.sin_family = AF_INET;
  addr.sin_port   = 0;
  addr.sin_addr.s_addr = INADDR_ANY;
  len = sizeof(struct sockaddr_in);
  sock_ic = socket(AF_INET, SOCK_STREAM, 0);
  bind(sock_ic, (void *) &addr, len);
  getsockname(sock_ic, (void *) &addr, &len);
#ifdef REALLY_DEBUG
  fprintf(stderr,"client connect port: %d\n",addr.sin_port);
#endif
  listen(sock_ic, 5); /* max backlog */
  if((i=fcntl(sock_ic,F_GETFL))==-1) goto lose;
  i|=O_NONBLOCK;
  if(fcntl(sock_ic,F_SETFL,i)==-1) goto lose;

/* Incoming Server Socket */
  srvr.sin_family = AF_INET;
  srvr.sin_port   = 0;
  srvr.sin_addr.s_addr = INADDR_ANY;
  len = sizeof(struct sockaddr_in);

  sock_is = socket(AF_INET, SOCK_STREAM, 0);
  bind(sock_is, (void *) &srvr, len);
  getsockname(sock_is, (void *) &srvr, &len);
#ifdef DEBUG
  fprintf(stderr,"server connect port: %d\n",srvr.sin_port);
#endif
  listen(sock_is, 5); /* max backlog */
  if((i=fcntl(sock_is,F_GETFL))==-1) goto lose;
  i|=O_NONBLOCK;
  if(fcntl(sock_is,F_SETFL,i)==-1) goto lose;
```

Figure A-1: `iflamec` Client/Server socket initialization

```
/* Incoming Unix Socket */
  clnt.sun_family = AF_UNIX;
  sprintf(clnt.sun_path, "%s%s%d.%d", PATH_STRING,
                        (char *)getenv("HOST"),(int) getuid(), count);
  ulen = sizeof(struct sockaddr_un);
  sock_iu = socket(AF_UNIX, SOCK_STREAM, 0);
  while(bind(sock_iu, (void *) &clnt, ulen)) {
    sprintf(clnt.sun_path, "%s%s%d.%d", PATH_STRING,
                          (char *)getenv("HOST"),(int) getuid(), ++count);
  }
  listen(sock_iu, 5); /* max backlog */
  if((i=fcntl(sock_iu,F_GETFL))==-1) goto lose;
  i|=O_NONBLOCK;
  if(fcntl(sock_iu,F_SETFL,i)==-1) goto lose;

/* Outgoing Server/Client Socket */
  out.sin_family = AF_INET;
  out.sin_port   = 0;

  if (sock_ic > sock_is) if (sock_ic > sock_iu)
      lowest_fd = sock_ic;
    else lowest_fd = sock_iu;
  else
    if (sock_is > sock_iu) lowest_fd = sock_is;
    else lowest_fd = sock_iu;
  lowest_fd++;
  num_fd=lowest_fd;
  min_fd=lowest_fd;
  max_fd=lowest_fd;
  for (i=0;i<NOFILE;i++)
    connection[i]=NULL;
  return;
 lose:
  perror("iflamec: init_sockets");
  exit(1);
}
```

Figure A-2: `iflamec` Unix socket initialization

```
FD_ZERO(&readfds);
FD_ZERO(&writefds);
FD_SET(sock_is,&readfds);
FD_SET(sock_ic,&readfds);
FD_SET(sock_iu,&readfds);
extra=FALSE;

for(fd=min_fd;fd<max_fd;fd++)
  if(CONNECTION_P)
    if (WRITE_STATE)
      FD_SET(fd,&writefds);
    else
      {
        FD_SET(fd,&readfds);
        if(EXTRA)
          extra=TRUE;
      }

if(select(max_fd,&readfds,&writefds,NULL,NULL)==-1)
  goto lose;
```

Figure A-3: The `iflamec` `select()` system call setup

```
int open_server(char *datbuf)
{
  int old_fd;

  strcpy(namebuf, datbuf);
  indx2=strtok(namebuf, "@");
  if ((indx2=strtok(NULL, "@")) != NULL) {
    out_tmp.sin_family = AF_INET;
    out_tmp.sin_port   = 0;
    strcpy(new_server, indx2);
    if((tmphost = gethostbyname(new_server)) == NULL) {
      fprintf(stderr, "Couldn't get host %s \n",new_server);
      return(FALSE);
    }
    /* Add new server to server list */
    add_server(new_server);
    memcpy((char *) &out_tmp.sin_addr,tmphost->h_addr,tmphost->h_length);
    len = sizeof(struct sockaddr_in);
    out_tmp.sin_port = htons(IFLAMED_PORT);
    if(!server_connect(out_tmp)) return(0);
  }
  else
    if(!server_connect(out)) return(0);
  /* Checking for 200 OK connection */
  read_input();

  /* Send location information */
  if(sprintf(OUT_BUF,"LOC %d %d\n",
      addr.sin_port, srvr.sin_port)<0) goto lose;
  STATE = LOC_WRITE;
  old_fd = fd; /* store fd */
  dispatch();
  fd = old_fd;
  server_closed=FALSE;
  return(1);
 lose:
  perror("iflamec: server_connect");
  return(0);
}
```

Figure A-4: iflamec's open_server() function

```
user_t *read_database(datum database_key, datum database_data,
  user_t *user_data)
{
  int name_size, authdata_size;
  int close_flag = FALSE;
  char *name_offset,  *authdata_offset;
  if (user_data != NULL) {
    free(user_data);
    user_data = NULL;
  }
  if ((user_data = make_user(UNAUTHENTICATED))==NULL)
    goto lose;
  if (dbf==NULL)
    dbf = gdbm_open(database_path, CHAR_BUF_SIZE, GDBM_WRCREAT,
                    (S_IXGRP|S_IROTH|S_IWOTH|S_IXOTH|S_IRGRP|S_IWGRP|
                     S_IRUSR|S_IWUSR|S_IXUSR), 0);
  else close_flag = TRUE; /* database was open prior to entering */
  database_data = gdbm_fetch(dbf, database_key);
  if (database_data.dptr == NULL) {
    /* User not in database */
    free(user_data);
    return(NULL);
  }
  name_size = *(int *)(database_data.dptr);
  name_offset = (char *)(database_data.dptr + (sizeof(int) * 4));
  if ((user_data->name = malloc(sizeof(char) * name_size + 1)) == NULL)
    goto lose;
  user_data->refcount = *(int *)(database_data.dptr + sizeof(int));
  user_data->authtype =  *(int *)(database_data.dptr + (sizeof(int) * 2));
  authdata_size =  *(int *)(database_data.dptr + (sizeof(int) * 3));
  authdata_offset = (char *)(database_data.dptr + (sizeof(int) * 4)
                    + (sizeof(char) * (name_size + 1)));
  if ((user_data->authdata = malloc(sizeof(char) * authdata_size + 1)) == NULL)
    goto lose;
  strcpy(user_data->name, name_offset);
  strcpy(user_data->authdata, authdata_offset);
  if (!close_flag) {
    gdbm_close(dbf);
    dbf = NULL;
  }
  return(user_data);
 lose:
  if (!close_flag) {
    gdbm_close(dbf);
    dbf = NULL;
  }
  perror("iflamed: read_database");
  return(user_data);
}
```

Figure A-5: `iflamed` Database Access

```
proc display {} {

    global text;
    global from;
    global forum;
    global sig;
    global width1;
    global width2;
    global ttymode;

    switch $ttymode {
        "0" {
        frame   .f1 -width $width2 ;
        frame   .f2 -width $width2 ;
        frame   .f3 -width $width2 ;
        wm geometry . +0+0 ;
        message .forum2 -relief flat -text "Forum"  -width $width2;
        message .sig2 -relief flat -text "From"  -width $width2;
        message .t -text "$text"  -width $width2;
        message .sig -relief flat -text "$sig"  -width $width2;
        message .from -relief flat -text "<$from>" -width $width2;
        message .forum -relief flat -text "$forum"  -width $width2;
        pack .f1 ;
        pack .f2 ;
        pack .f3 ;
        pack .forum -side right -in .f1 ;
        pack .forum2 -side left -in .f1 ;
        pack .sig2 .sig .from -side left -anchor s -in .f2 ;
        pack .t -side left  -anchor w -in .f3 ;
        bind . <Any-ButtonRelease> {destroy .} ;
        bind .sig <Any-ButtonRelease> {destroy .}  ;
        bind .sig2 <Any-ButtonRelease> {destroy .}  ;
        bind .from <Any-ButtonRelease> {destroy .}   ;
        bind .forum <Any-ButtonRelease> {destroy .}  ;
        bind .forum2 <Any-ButtonRelease> {destroy .}  ;
        bind .t <Any-ButtonRelease> {destroy .}   ;
        bind .f1 <Any-ButtonRelease> {destroy .}  ;
        bind .f2 <Any-ButtonRelease> {destroy .}  ;
        bind .f3 <Any-ButtonRelease> {destroy .}  ;
        return TCL_OK;
    }
    "1" {
        format "Forum: %s\nFrom: %s\n---\n%s" $forum $from $text;
    }
  }
}
```

Figure A-6: The .iflamec.tcl file

Bibliography

[1] The Alta Vista search engine, `http://www.altavista.digital.com/`.

[2] T. Berners-Lee, *RFC1630: Universal Resource Identifiers in WWW: A Unifying Syntax for the Expression of Names and Addresses of Objects on the Network as used in the World-Wide Web*, June 1994.

[3] Chesnais, P., Mucklo, M., Sheena, J., *The Fishwrap Personalized News System.* 1995.

[4] Cormer, T., Leiserson, C., and Rivest, R., *Introduction to Algorithms.* 1990.

[5] D. Crocker, *RFC822: Standard for the format of ARPA Internet text messages*, 08/13/1982.

[6] Deering, S., *RFC1112: Host Extensions for IP Multicasting.* August 1989.

[7] C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, Athena Technical Plan *Section E.4.1: Zephyr Notification Service*, M.I.T. Project Athena, Cambridge, Massachusetts (June 5, 1989).

[8] C. A. DellaFera, M. W. Eichin, R. S. French, D. C. Jedlinsky, J. T. Kohl, and W. E. Sommerfeld, *The Zephyr Notification Service*, M.I.T. Project Athena, Cambridge, Massachusetts (December 21, 1987)

[9] B. Kantor, P. Lapsley, *RFC0977: Network News Transfer Protocol: A Proposed Standard for the Stream-Based Transmission of News*, (February 1, 1986).

[10] Kohl, J., Neuman, C., *RFC1510: The Kerberos Network Authentication Service (V5).* September 1993.

[11] Macedonia, M. R., Brutzman, D. P.,"MBone Provides Audio and Video Across the Internet," IEEE Computer, Vol.27 no. 4, April 1994, pp. 30-36.

[12] McCahill, M. *The Internet Gopher: A distributed server information system*, from *ConneXions–The Interoperability Report*. 1992.

[13] Mitchell, W., *City of Bits*, available online at `http://www-mitpress.mit.edu/City_of_Bits/`, 1995.

[14] J. Oikarinen, D. Reed, *RFC1459: Internet Relay Chat Protocol* (May 26, 1993).

[15] J. Postel, J. Reynolds, *RFC959: File Transfer Protocol (FTP)*, 1985.

[16] T. Rinne, *RFC1756: Remote Write Protocol - Version 1.0* (January 19, 1995).