PROJECT ATHENA TECHNICAL PLAN

Section E.4.1

Zephyr Notification Service

by C. Anthony DellaFera,
John T. Kohl,
Mark W. Eichin,
Robert S. French,
David C. Jedlinsky,
William E. Sommerfeld

Zephyr is a notice transport and delivery system under development at MIT Project Athena. Zephyr was developed for use with 4.3BSD UNIX[1] but should be portable to other operating systems. Zephyr was developed for use by network based services and applications with a need for immediate, reliable and rapid UID based communication with their clients. Zephyr meets the high throughput, high fan-out communications requirements of large-scale workstation environments. None of the communications packages currently availble for use in a workstation environment adequately meet these requirements. Electronic mail provides guaranteed delivery but is too slow and cumbersome. Broadcast messages are fast but too transient and don't scale well.

Zephyr is designed as a suite of services based on a reliable, authenticated, UID to UID notice protocol. Multiple, redundant Zephyr servers provide basic location, routing, queueing and dispatching services to Zephyr clients that communicate via the Zephyr Client Library. Other services can be built upon this base.

Note: Comments on this document may either be sent via electronic mail to zephyr-comments@athena.MIT.EDU or entered into the local Athena "zephyr-comments" *discuss* meeting. ***Areas delineated by change bars are under development and subject to change without notice!***

Other documents of interest

Besides this document, there are other documents describing parts of the Zephyr system:

- *The Zephyr Programmer's Manual* gives a description of the Zephyr Client Library, including examples of use.
- *The Zephyr User's Manual* gives a user-level description of the operation of the Zephyr Notification Service, including examples of typical use.

[1]UNIX is a trademark of AT&T Bell Laboratories.

- *The Zephyr Installation and Operation Guide* describes typical installation, operation and maintenance of the Zephyr system.
- *Zephyr Implementation Notes* discuss the current implementation of the Zephyr system.
- UNIX manual pages briefly describe the programs in the Zephyr suite.

Organization of this Document

This document presents the concept of a notification service, including design considerations, assumptions and constraints of Zephyr, and details the design, constraints, and decision rationales for each of the pieces of Zephyr.

Definitions

IP	Internet Protocol.
UDP	User Datagram Protocol.
Notice	The fundamental unit of data transmission in the Zephyr system. A notice consists of routing information in its *header* plus client-determined *data*.
ZID	A "Zephyr ID" which uniquely identifies an entity using Zephyr. In the Athena environment, a ZID is usually the Kerberos [4] Principal identifier of the entity, such as "jtkohl.root@ATHENA.MIT.EDU."
ASCII	American Standard Code for Information Interchange.
ACL	Access Control List. An ACL lists in some form the entities permitted to perform some action.
Class	A part of the notice header specifying a general class of notice type. Often considered an attribute of a notice.
Instance	A part of the notice header specifying a subclass of the notice class. Sometimes referred to as "Class instance."
Recipient	A part of the notice header specifying the destination ZID of the notice. The recipient may be the null string ("\0") to indicate that the notice should be multicast.
Multicast	To transmit to a subset of all clients.
Zephyr Client	Any program or user agent which can transmit or receive notices.
Subscription	A tuple of {class,instance,recipient} maintained by a Zephyr server indicating a client's desire to receive notices with the specified class, instance, and recipient fields in its header.
Wildcard	A string specification that matches all strings for comparison purposes.
Zephyr Realm	The set of hosts supported by a particular set of Zephyr servers. This will typically be the same as a Kerberos realm.

May also refer to the name given this realm.

1. An Introduction To The Design Of Zephyr

This chapter is an introduction to the concept of a notification service in general and to the design of the Athena Notification Service, Zephyr. The sections which follow address the following issues:

- What role does a notification service play in modern network based workstation environments?
- What are the motivating factors behind the development of a notification service?
- Under what design constraints must the service be developed?
- What environmental assumptions have been made in the design and implementation of Zephyr?
- What level and types of services should users expect from a notification service?
- What are some of the unsolved problems and topics for future development?

While much of the information in this chapter is oriented around Zephyr, Project Athena's Notification Service, it is our belief that the concepts presented here can be generalized to fit a broad range of notification services and systems.

1.1. Why a notification service?

As Project Athena deployed large numbers of workstations and servers, it became clear that current UNIX communications systems were inadequate and incapable of dealing with many of the needs of the environment being put in place at M.I.T. The advantages of a notification service as a solution to a broad spectrum of communications needs in a workstation environment became clear as we began to examine these inadequacies.

When services designed for use in a time-sharing environment are reused for a very large system of networked workstations, certain communication services begin to fail[2]. Their failure is predominantly due to their inability to cope with increasing network size (i.e., an increase in the number of both workstations and local area networks). In examining how certain of these services communicate with their clients, we have identified two primary failure modes: the inability of a service to cope with rapidly increasing numbers of client nodes, and the inability of clients to deal with the replacement of a local service with a remote service. Zephyr is the result of what was initially begun as the development of a solution to these two failure modes. Zephyr has grown into a more powerful tool than was originally anticipated; what began as the development of a "desirable" service soon turned into the development of a "required" service.

The following lists some of the existing workstation communication requirements that have been helped to scale properly by relying on the existence of a notification service.

[2]Services, in general, begin to fail, but the scope of this discussion is primarily the realm of communication services.

File Service	Users need to be informed of unexpected changes in file server status. Utilizing the notification service, a file server can send notices to the users and hosts that it knows would be affected by a change in file server status. If a user registers an appropriate subscription with the subscription service, he will receive any such notices. A third party client would be able to notify subscribers about operational issues associated with that file server.
Post Office Service	Remote post offices can notify users about the arrival of new mail.
Electronic Conferencing Systems	Electronic conferencing systems do not necessarily keep a readership list (conferences are often are open to the “general public”), so the system cannot identify a community of interest for event notification. However, users interested in the status of certain meetings can use the subscription service to subscribe to conference update notices.
Print Service	Print servers (and queuing services in general) can utilize the notification service to communicate status information back to the user who submitted a job, whether or not the user has changed location in the interim. Typical replies might be “Print job complete.” and “Paper tray 2 empty on LPS40-1.”.
MOTD Service	Message-of-the-day information (system wide, service specific or local) can be sent to users via the notification service when they begin using a particular service. Notification of changes to that information can be provided via the subscription service.
On-Line Consulting	The notification service can be used as the underpinning of a dynamic on-line consulting service. The subscription service can be used to provide topic based information routing, user location, and consultant to client rendezvous.
Host Status Service	Broadcast based host status systems (such as ruptime) do not scale to a large workstation environment; disk usage grows linearly with network size, total packet compute time grows geometrically, and broadcast facilities are available only on the local network (broadcasts do not propagate through network gateways). The notification service can provide asynchronous and immediate host status and error log notification on selected hosts or servers. Notification can be initiated either by the server using the notification service to reach predetermined recipients, or to self-selected recipients through the subscription service. This is desirable for service management purposes.
User Location Service	Broadcast based user location systems (such as rwho) also do not scale to a large workstation environment for the same reasons noted above under Host Status Service. The notification service can provide asynchronous and immediate user location and state change (login/logout) notification on selected users. This can facilitate communication among users. For ex-

	ample, within the limits of user permission and access control (described elsewhere), students could watch for their friends, Teaching Assistants, or colleagues.
Talk or Phone Service	Current systems that do not have access to network-wide user location information require their users know the exact address of someone is in order to establish a two-way communication channel. In addition, such systems typically do not provide any access control or method of rejecting an annoying “talker” beyond turning off the service altogether. Using the notification service, a “talk” facility can be constructed that locates the party being called, transmits a connection request notice to that party and, if permission is granted, automatically (via rendezvous information included in the request notice) establish a connection.
Emergency Notification	There is a requirement to provide a simple, asynchronous and secure means of sending urgent notices to all users on a workstation or in a particular group of workstations. Mail is slow and synchronous (in that the user must initiate the reading process). Broadcast methods are not useful due to scale and network boundary considerations. In addition, the workstation user must trust the broadcasting host and the person issuing the message. Finally, broadcast does not answer the question of how to notify multiple users on a workstation, should that be the case. Using the notification service, emergency notices can be sent directly to all users on any specified host.
Message Service	Current one-way communication services suffer the same problems noted above under Talk or Phone Services. Using the notification service the implementation of a “write” utility becomes extremely easy; almost all the work is subsumed by the notification service. Notices can go to individual users or to multicast subscription groups.
Other Service Events	These are only a few of the interesting services. The notification service can be used to reliably notify users of a whole range of asynchronous service events that occur in current distributed workstation environments.

1.2. Design Requirements And Constraints

A notification service is intended to provide network based services with immediate, reliable and rapid entity-based communication for small quantities of time-sensitive information. Such a service must be able to efficiently provide these capabilities, with the highest possible fan-out (i.e., client to server ratio) but without adversely affecting network load or server host performance.

As the notification service is a distributed service intended to serve a large number of workstations, it is important that all aspects of service maintenance and operation be easily controlled and manipulated remotely.

Since the notification service is still evolving, it is important that it be designed such that there is some method of gracefully handling protocol compatibility from one version of the service to the next. This means that servers and clients must be able to interoperate with at least the previous and next versions.

The notification service is a communication system and depends upon an underlying network transport mechanism, so it must accept certain design constraints imposed by that mechanism, and should make attempts to remedy deficiencies present in that mechanism.

In order to provide entity-based communications, the notification service must maintain a database that maps entity names to the current location(s) in the network of the corresponding entity. The only requirement on the entity name used in addressing is that it be unique across the entire network.

1.3. Environmental Assumptions

The following items are assumed in the design of Zephyr:

Networked environment	Zephyr is designed for use in a networked environment of (marginally) cooperating hosts. For a stand-alone single-user or timesharing system, it offers very little advantage over standard UNIX utilities such as *write* and *talk*.
High-speed network	Zephyr assumes the availability of a high-speed network for communication between hosts within a given Zephyr realm. If a high-speed network is not available, many of the implicit assumptions are no longer correct and the system may not perform adequately. There is *no* requirement for broadcast capability.
Authentication	Zephyr assumes the presence of an authentication system which is secure on the network over which it operates. Because of its flexibility, Zephyr needs a way to reliably determine the sender of any given notice. Without an authentication system in place, the reliability of this identification is highly suspect, and the ability to spoof Zephyr becomes easy. [If Zephyr is being operated in a captive, friendly, cooperative environment, this may not be a cause for concern.]
Administration	The Zephyr server is designed to run indefinitely without intervention.

1.4. Fitting The Tool To The Job

To satisfy the above requirements, Zephyr is composed of three principal elements:

- A notice transport mechanism.
- A set of “canned” applications which use the transport mechanism.
- A library of functions through which new applications may use the transport mechanism (the “Zephyr Client Library”).

The core of the notice transport mechanism is a system of dynamically updated, locally authoritative servers that provide centralized routing, queuing and dispatching. Clients communicate with these servers via the Zephyr Client Library interface. The Zephyr Client Library implements the Zephyr Protocol, a reliable, authenticated, ZID to ZID notice transmission protocol.

A Zephyr server requires no maintenance for daily operations; changing access control lists or default subscriptions can be accommodated by remote logins to the server host or by automatic database-fed updates (such as those provided by the Athena Service Management System [2]).

The Zephyr Protocol includes a version number so that multiple versions may be supported simultaneously.

The Zephyr Protocol is UDP/IP based. We chose UDP/IP for several reasons. The campus network at M.I.T. uses IP almost exclusively, so we were constrained to the IP protocol suite. We also desired a “lightweight” transport mechanism with little overhead. TCP/IP has far too much overhead involved in initiating connections. UDP/IP is much more appropriate, since it is connectionless and thus has much less overhead. The 4.3BSD implementation of UDP/IP allows rapid switching of destinations, a necessity if we are to use a central server to do notice dispatching.

Since Zephyr is based on UDP/IP, it is constrained to operate within the capabilities of UDP/IP. However, there is nothing in the design of the protocol that would prevent it from using other network transport mechanisms (such as a remote procedure call system).

The five primary UDP/IP imposed constraints are listed below, along with a brief description of how they may or may not be visible to Zephyr application programmers and end users.

Duplicate Notices	UDP/IP does not provide any suppression of duplicate packets. This means that it is possible that Zephyr clients may under certain adverse network conditions receive duplicate Zephyr notices. Zephyr applications must be capable of dealing with this possibility even though Zephyr does provide some filtering to block duplicate notices.
Missequenced Notices	UDP/IP does not provide packet sequencing. While Zephyr notices do contain timestamps it is up to the application to check the timestamp and be willing to deal with notices received out of sequence[3].
Flow Control	UDP/IP does not provide any flow control capability. The application must be capable of dealing with notices at whatever rate they arrive or be willing to lose notices.
Unreliable Delivery	UDP/IP does not provide a reliable delivery mechanism. While Zephyr does provide retransmission and several levels of acknowledgment processing, it is up to the application to decide

[3]There is still the problem of synchronization of clocks on the (possibly many) sending hosts, so even the timestamps cannot be relied upon to determine the order in which notices were transmitted.

	how much overhead it is willing to incur in order to guarantee notice delivery.
Packet Size	UDP/IP packets have a relatively small, fixed size. Considering the amount of routing data and other information that Zephyr must store in each packet, there are some strict constraints on how much user data may be included with each packet. To help remedy this, Zephyr provides automatic packet fragmentation and reassembly, but it is not as efficient as other means of transmitting large quantities of data. Applications requiring large data transfers should use Zephyr as a rendezvous service, and establish their own separate data transmission channels.

How visible these constraints are to the end user is up to the Zephyr application programmer. For example, the "zwrite" application (which allows workstation users to exchange "write"-like messages) only guarantees that the message was sent, not that it will actually arrive or how many copies will arrive. This is because constraints one and four above (Duplicate Notices and Unreliable Delivery) were by design not completely eradicated.

Zephyr manages a ZID to location translation database, known as the "user location database." This database maps ZID's to tuples of location information: hostname, IP port number, and display device (among other things). This database is made available to Zephyr clients via the "user locator" service. The only ZID's normally found in the database are those designating users of the system; by convention network services do not announce themselves to the user location database, as the database is intended for use by users, not services. Zephyr maintains a separate location database (containing only ZID, IP address and port number) for all clients wishing to receive notifications. It is this database which is used to determine the proper routing of notices.

The reliability of the information stored in the user location database imposes some constraints upon applications that rely on Zephyr and the users of those applications.

- User location information stored in the database ***can*** be assumed to be a reasonably accurate report of a user login. This can be asserted because all user logins reported to Zephyr must be Kerberos authenticated.
- User location information stored in the database ***can not*** be assumed to accurately indicate that a user ***has not*** logged out. This must be asserted because there is no way to guarantee an orderly authenticated user logout (*e.g.* a workstation may hang, crash, or reboot due to problems beyond Zephyr's control).
- User location information ***not*** stored in the database ***can not*** be used to assume a user ***did not*** login. This must be asserted because a user may choose to not make his or her login information available.

As designed, Zephyr performs a reasonable amount of housekeeping to prevent transient data stored in the user location database from persisting when it is no longer valid. If a workstation crashes, the user login sessions on that workstation terminate without sending logout notices to Zephyr. This permits invalid information to remain in the database (Note: the initial logins were not invalid; that the information remains in the database is.). In order to cope with this, a Zephyr client runs when a workstation reboots, telling Zephyr to flush any previous state information associated with the rebooted workstation.

If Zephyr application programmers and Zephyr application users understand the above issues and maintain realistic expectations then the few difficulties should arise.

Zephyr clients determine what level of service they require from the notification service by choosing which type of routing they use. For example, certain client services have a concise knowledge of who their clients are and only need the notification service to route information to those clients, using their ZID (the most basic service). A file server that knows which users it is serving needs only the ability to reliably notify those users about service state changes. On the other hand, client services that cannot identify their clients (or may simply not know who is interested in such state information) may wish to notify "interested parties" about service state changes. A workstation event logger would fall into this category. This type of service would make use of the subscription service with multicasting based on who has declared themselves "interested." Such a service layer provides the ability to store communication state information for client services external to those services. This adds to the notification service the ability to provide to its clients status and availability information about other client services even when those client services are disabled and, as such, can't communicate with their own clients directly.

We do not specify how this system of services should be implemented. Our current implementation of Zephyr makes both the user locator service and the subscription service functions of a single base notification server. While this implementation may not be optimal due to too tight a coupling between the individual services, it proved easiest to implement within a tight schedule. It is our intention, at some future date, to further separate the individual services, thereby gaining the flexibility of a system of modular and abstract services.

In Zephyr, notices are understood to consist of two parts, a routing header and client data. It is the job of the Zephyr servers to route notices from Zephyr clients to other Zephyr clients and servers based upon attributes specified in the notice's routing header. Servers should never expect to be able to examine a notice's client data; it is entirely possible that that data is encrypted or otherwise uninterpretable. By examining the attributes in a notice's routing header, it is possible for any Zephyr server to compute a finite list of recipients to whom the notice should be sent. The most basic routing attribute that may be specified is a tuple of <class, instance, recipient ZID>. Additional and more complex routing attributes might need to be specified for layered services that use more complicated routing methods. For example, an experimental rule based routing service might need specialized keywords.

We refer to the process of determining multiple notice recipients based upon routing header attributes as "multicasting." Multicasting is a passive routing technique; attributes not recognized by a routing service are simply ignored. This allows layered services to implement different notice routing methods that peacefully coexist while utilizing the same base notification service. It is not our intention that scores of notice routing methods be immediately available, but rather that the notification service fulfill an immediate need without blocking future development. Multicasting differs from other common message routing techniques. It is more efficient and less subject to failure due to diseconomies of scale than broadcast techniques because the complete set of recipients for any notice can always be determined, and uninterested recipients need not process a notice only to discover it should be ignored. It requires less maintenance and incurs less administrative overhead than traditional list based message transmission techniques (such as electronic mailing lists) because additional resources, routing methods and recipients (within limits) may be dynamically added by almost any user. Multicasting is a notice routing technique that is well suited for use as the core of a notification service.

A good understanding of Zephyr in perspective can best be acquired by comparing the Zephyr Notification Service and a more traditional method of workstation message delivery, electronic mail. The following table compares some of the critical metrics of notice/message delivery that are applicable to both Zephyr and electronic mail delivery by a typical UNIX mail delivery system, *sendmail*.

A Comparison Between Zephyr And Mail

Metric	Zephyr	Electronic Mail (Sendmail)
Addressing	Implicit/Dynamic: All addressing is dynamically determined; an explicit "address" is not required. One-to-one addressing is supported via explicit specification of recipient ZID. In addition, subscription based notification allows the recipient to be determined by notice attribute information.	Explicit/Static: Sender must know and explicitly provide the name and address (except for "local" mail) of each recipient or list of recipients being sent to. Static mailing list support is provided. Only recipients explicitly named will be delivered to.
Delivery Method	Notices are delivered via dynamically routed datagram. No connections need be established or maintained. Multiple levels of notice acknowledgment are supported, to provide reliability as needed.	Mail is delivered via point to point SMTP connection. Acknowledgments are not supported per-se, but return receipts may be requested (but may not work in all SMTP implementations).
Delivery Action	Asynchronous/Active: Notices arrive and are displayed without user intervention.	Synchronous/Passive: Mail is read by user action. Mail, in general, is delivered to one particular place (a "post office" or "mail drop") for each user; he or she must then actively retrieve it.
Message Length	Usually short notice length, with support for fragmentation of larger messages.	Long, typically unfixed, message length. Mail messages may be and often are extremely large, on the order of many pages of text. When message length is fixed it is usually done so by unpredictable rules that vary from site to site.
Message Persistence	Notices are considered time sensitive, no queuing is provided. If the destination client is not on the network then the notice is discarded. If a user didn't see it, he probably didn't need to.	Long time-to-live. Messages typically remain in your mail drop until you retrieve and delete them. Two weeks' worth of junk mail remains two weeks' worth of junk mail.

A Comparison Between Zephyr And Mail (continued)		
Metric	Zephyr	Electronic Mail (Sendmail)
Message Fan-out	High fan-out: Sending to large lists is efficient. When multicasting, each client sends only one copy of a notice regardless of the number of recipients. Each server receives only one copy of a notice being routed regardless of the number of recipients. Client-determined lists require separate notices for each recipient. No notice queuing is provided by the server. Clients determine whether or not to retain notices.	Low fan-out: Sending to large lists can be ***very*** resource consuming. If a message is sent to many users many copies are generated by the mail processing system, each of which is retained until deleted by its recipient. Queuing due to inability to contact destination hosts can consume large amounts of file storage space.
Traffic Performance	High volume/High throughput: Notices may be transmitted in large numbers due to the low overhead of dynamically routed datagrams.	Medium volume/Low throughput: Multiple large mail messages can send a reasonable volume of data but slowly as connections need to be established and routes determined.
System Configurability	Dynamically reconfigurable: Dynamic resource allocation and configuration within the base notification services allows automatic and simple user level reconfiguration of other services.	Statically reconfigurable: Reconfiguration of *Sendmail* is wizard level work and has significant global impact. No utilities are provided for dynamic system modification or reconfiguration. All changes must be made centrally and atomically.
System Maintenance	Low maintenance: Zephyr servers dynamically recover unused resources through time-outs and reference counting. <class, instance, ZID> tuples, ZID location information and other subscription service resources have a well defined time-to-live.	High maintenance: Mail requires a post office staff to maintain post office boxes (mail drops), mailing lists, the routing system and to manually reroute undeliverable "dead letters."

1.5. Future Directions And Unsolved Problems

Once the basic notification service is in place it becomes a simple matter to provide many other layered services based upon it. The "talk" service mentioned above is a good example of a service that utilizes multiple Zephyr services: the user locator service and the notice routing service.

We envision Zephyr as a transport service that can incorporate new notice routing methods as they are developed. Because of the dynamic configurability of the subscription service, Zephyr allows communication development efforts to occur side-by-side with running production systems that utilize Zephyr. For example, we are currently working with researchers at MIT's Sloan School of Management who are looking at using Zephyr as the transport service layer for a rule based communication system [3]. Such a system under development could use the subscription service to gather and process notices, and when complete could coexist with the subscription service as an alternate routing method.

Following is a list of some of the areas that the authors have slated for future development. They are either unsolved problems or areas that we feel need further investigation.

- Make the Zephyr Protocol and retransmission algorithms robust enough for use across long-haul and/or lossy networks.
- Modify the Zephyr server to interact sensibly with other Kerberos authentication realms, including problems of user registration across realms and notice forwarding across realms.
- Develop a more formal interface definition for use between the Zephyr notice transport layer and Zephyr routing services.
- Develop a more advanced user interface for the Zephyr WindowGram Client.

2. An overview of Zephyr's Components

2.1. Notice transport

Zephyr's notice transport mechanism involves two major components, the Server and the HostManager.

The server provides the user location service and the subscription service. All notices delivered via Zephyr are routed through at least one Zephyr server. There are typically a small number of Zephyr servers for each site using Zephyr.

The HostManager provides a reliable contact point for clients. Each host supporting Zephyr runs a HostManager, which locates a server and routes any notices sent by clients to the server. If a server becomes unreachable, the HostManager will attempt to contact a new server. By placing this functionality in one place, the server's job of detecting host failures is made easier, since it has a known contact point. In addition, the clients become less complicated as they need not find and establish communication with a server.

2.2. Clients

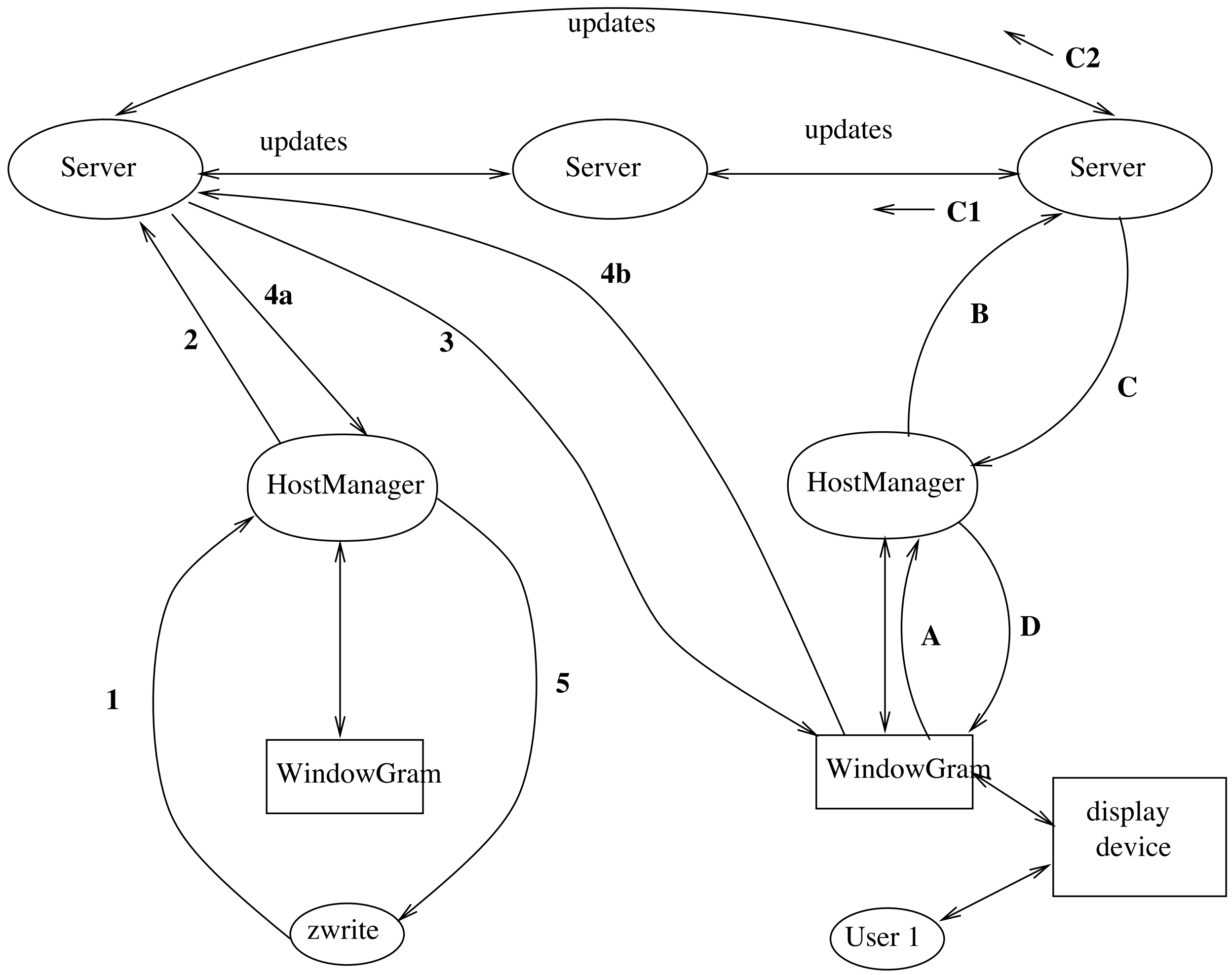
Client programs use the notice transport mechanism to deliver and receive notices across the network. Clients send notices to a Zephyr server (via the HostManager) to establish a subscription. Other clients send notices to servers for redistribution to clients. The recipients of the redistributed notice are determined by the subscription service.

Figure 1 shows a typical interaction between clients, HostManagers and servers.

3. The Zephyr Protocol

This chapter describes the current Zephyr protocol, version ZEPH0.2. A description of the protocol used for database transfers between servers is given below in section 4.4.

Figure 1: A typical exchange between clients



The arrows labeled **A, B, C, C1, C2, D** depict notices involved in establishing a subscription. The WindowGram client requests a subscription by transmitting a notice to the HostManager (A), which forwards it to a server (B). The server responds with an acknowledgment (C) which is forwarded to the client (D). The server also forwards the request to the other servers (C1, C2).

The arrows labeled **1, 2, 3, 4a, 4b, 5** depict notices involved in sending a notice from one client to another. A user on one host sends a notice to another user on a different host by sending the notice to the local HostManager (1), which forwards the notice to its server (2). The server transmits the notice to the other user (3), and then acknowledges the notice to the sender (4a). The recipient acknowledges receipt of the notice (4b). The HostManager forwards the server acknowledgment to the sender (5).

3.1. Overview

The Zephyr Protocol is implemented on UDP/IP, providing authenticated subscription-based multicast and ZID to ZID message delivery. The Zephyr Protocol is used both for messages delivered to end users of the system and for control and maintenance of the message transport system.

Notices may be sent from a client program to a Zephyr server, from one Zephyr server to another, or from a Zephyr server to a client program; they are not intended to be sent directly from one client to another. Instead, a Zephyr server always acts as a relay for the message.

Each datagram may contain either part of a notice or a complete notice; separate notices must be transmitted in separate datagrams. Transmission and reception of notices is modular and properly abstracted. Zephyr clients and servers utilize a common Zephyr Client Library that provides routines for such operations as datagram port setup and shutdown, notice transmission, and notice reception.

All notices share a common header format followed by a client specific data area. All notices are routed, queued, parsed and acted upon by examining this header. In this way the client specific data area need never be examined except by the destination Zephyr client(s).

It is intended that Zephyr be usable as a "rendezvous" system; thus, higher-level protocols can be built using Zephyr as a transport mechanism. The rendezvous can be accomplished by using Zephyr to exchange internet addresses and port numbers.

3.2. Notice Datagram Format

The datagram is in two parts. The first part is a variable length header, followed by a client data area. All header fields are represented in net ASCII. Those header fields with multiple-byte integer values are translated into network byte order before conversion to net ASCII. Each field is terminated by a NUL (character code zero). Non-string values are represented in net ASCII as "`0xaabbccdd 0xeeffgghh...`", where `aa` is the hexadecimal representation of the first byte of data, `bb` is the representation of the second byte, etc.

In order, the datagram fields are:

Version String	This consists of a unique identifying string (initially "ZEPH") followed by a major version number, a period, and a minor version number. Changes in the major version number will cause the Zephyr Client Library to report that a packet was formatted using an incompatible version of the Zephyr protocol. A change in the minor version number is used to indicate that the overall format is compatible, but certain pieces of data may need to be treated differently.
Number of Header Fields	4 byte integer. This field is included so that future extensions to the Zephyr protocol may be made by appending additional fields to the header. The Zephyr library will ignore fields it is not expecting. This count includes the Version String and Number of Header Fields fields.

Notice Kind	4 byte integer. This field contains the kind of notice, using one of the named values below. UNSAFE (code 0) - The notice is simply transmitted. No acknowledgment from the local HostManager or a server is expected. UNACKED (code 1) - The notice is acknowledged by the HostManager, but the HostManager does not forward the server acknowledgment to the client. The HostManager acknowledgment will be handled internally by the Zephyr Client Library. ACKED (code 2) - The notice is acknowledged by both the HostManager and the server. The HostManager acknowledgment will be handled internally by the Zephyr Client Library. The application must handle the server acknowledgment itself. HMACK (code 3) - The notice is an acknowledgment from the HostManager to the client application. HMCTL (code 4) - The notice is a HostManager control message. SERVACK (code 5) - The notice is a server acknowledgment indicating that the notice was received and handled successfully. SERVNAK (code 6) - The notice is a server acknowledgment indicating that something went wrong during the handling of the notice. This normally indicates that there was an authentication failure. CLIENTACK (code 7) - The notice is an acknowledgment from a client. These notices are generated automatically by the Zephyr Client Library. STAT (code 8) - The notice is requesting the destination HostManager or server to return statistics about itself.
Unique packet ID	12 bytes. 4 bytes internet address, 8 bytes host-generated timestamp.
Port	2 bytes. This is the return port that should be used to respond to this notice, if such a response is necessary. The port from which the message was received (as reported by the operating system) should be *ignored*, since it will typically be the port of a Zephyr HostManager or server.
Authentication	4 byte integer. A code representing the type of authentication that was used while formatting this packet. The currently defined codes are: 0 - No authentication 1 - Kerberos authentication
Authenticator length	4 byte integer. The length in bytes of the following authenticator field.
Authenticator	Variable length character data. The authenticator used to determine the authentication of the packet.
Class	NUL-terminated ASCII string identifying a notice's class. The notice's class specifies the service class of notice's originating Zephyr client. Class is the highest level of notice classification. Examples of classes are: LOGIN, ZEPHYR_CTL, MESSAGE, FILSRV.

Class Instance	NUL-terminated ASCII string containing the particular instance of the class with which this notice deals. Examples of class instances are: rfrench@ATHENA.MIT.EDU, SUBSCRIBE, PERSONAL, HELEN.MIT.EDU:/filesystem.
Opcode	NUL-terminated ASCII string identifying the particular operation which the notice's originating Zephyr client has performed or expects the target(s) to perform. Opcodes are class specific. Examples of opcodes are: USER_LOGIN, USER_LOGOUT, SUBSCRIBE, UNSUBSCRIBE.
Sender	NUL-terminated ASCII string identifying the ZID of the sender of the notice.
Recipient	NUL-terminated ASCII string containing the ZID of the desired recipient of the notice. If the recipient is the null string then the target(s) are determined by the notice's class and class instance as outlined in the Server Chapter (Chapter 4).
Default Display Format	NUL-terminated ASCII string that is used by the WindowGram client to display the notice if it has no other rules regarding the notice's particular class and class instance.
Checksum	4 byte integer. This field is for authentication. When Kerberos is in use, this field is built by the Zephyr server using the quad_cksum routine in the DES library to cryptographically checksum the previous header fields using the DES session key. It is assumed that the byte order of the checksum generated by the DES library is invariant across all machine architectures.
Fragmentation count	NUL-terminated ASCII string. This string is in the format "part/partof", such as "707/8000", indicating the data area of this packet contains a block of data starting at byte 707, and the total data size of the unfragmented notice is 8000 bytes. If this field is empty, the packet was not fragmented during transmission. A more detailed description of packet fragmentation can be found in the Library Chapter (Chapter 6).
Unique notice ID	12 bytes. 4 bytes internet address, 8 bytes host-generated timestamp. All fragments of a fragmented notice will have identical Unique notice ID's.
Other fields	Other fields will be placed here when the protocol is expanded.

Variable length client data:

Data	Byte stream data. The format of this data is specific to the particular communicating clients. It normally consists of one or more NUL-terminated strings containing ASCII data.

Sample datagram (268 bytes long):

```
    0:   "ZEPH0.2\0"                              Version string
    8:   "0x00000011\0"                           Number of fields (17)
    19:  "0x00000002\0"                           Kind - "ACKED"
    30:  "0x1248008D 0x22FA1319 0x000AEA49\0"     Unique packet ID
    63:  "0x004b\0"                               Port
    70:  "0x00000000\0"                           Authentication (None)
    81:  "0x00000000\0"                           Authenticator length
    92:  "\0"                                     Authenticator (None)
    93:  "MESSAGE\0"                              Class
    101: "PERSONAL\0"                             Class instance
    110: "\0"                                     Opcode (None)
    111: "rfrench@ATHENA.MIT.EDU\0"               Sender
    134: "tony@ATHENA.MIT.EDU\0"                  Recipient
    154: "Message from $sender at"                Default format
         " $time:\n\n$message\0"
    195: "0x12345678\0"                           Checksum
    206: "\0"                                     Fragmentation count
    207: "0x1248008D 0x22FA1319 0x000AEA49\0"     Unique notice ID
    240: "Hello - This is an example!\0"          Message
```

3.3. Naming Conventions

Users and daemons share the same ZID namespace.

3.4. Predefined Class Operations

The following table describes the predefined classes and their effects when received by servers, HostManagers, and WindowGram clients.

Reserved Class Definitions			
Class	Instance	Opcode	Effect
ZEPHYR_CTL	CLIENT	SUBSCRIBE	Client -> Server. Subscribe the client indicated by the port number in the notice to the class, class instance, and recipient triples listed in the message section of the notice. If this notice establishes the first subscriptions for the client, also subscribe the client to the server default subscriptions. This notice must be authenticated.

Reserved Class Definitions (continued)			
Class	Instance	Opcode	Effect
		SUBSCRIBE_NODEFS	Client -> Server. Subscribe the client indicated by the port number in the notice to the class, class instance, and recipient triples listed in the message section of the notice. This notice must be authenticated.
		UNSUBSCRIBE	Client -> Server. Unsubscribe the client indicated by the port number in the notice from the class, class instance, and recipient triples listed in the message section of the notice. This notice must be authenticated.
		CLEARSUB	Client -> Server. Clear all subscriptions relating to the port number specified in the notice. This notice may be authenticated. If it isn't, the server will attempt to verify that the client at the specified port has actually gone away.
		GIMME	Client -> Server. Return subscriptions associated with the specified port. This notice must be authenticated.
		GIMMEDEFS	Client -> Server. Return default system-wide subscriptions.
ZEPHYR_CTL	HM	BOOT	HostManager -> Server. Tell the server that this host has just booted and all state associated with it should be flushed.
		FLUSH	HostManager -> Server. Tell the server to flush all state associated with this host.
		DETACH	HostManager -> Server. Tell the server that this HostManager no longer considers it to be its owning server.

Reserved Class Definitions (continued)			
Class	Instance	Opcode	Effect
		ATTACH	HostManager -> Server. Tell the server that this HostManager now considers it to be its owning server.
HM_CTL	SERVER	SHUTDOWN	Server -> HostManager. Tell the HostManager that the server is going down, and that it should find another server. This packet optionally includes a suggested server in the client data area for the HostManager to transfer to. If included, the suggestion will be the IP Address of the suggested server, in Internet Address dot notation, in ASCII.
		PING	Server -> HostManager. Ask the HostManager to acknowledge. This assures that the HostManager (and thus the host) is still operating after a client fails to acknowledge notices.
	CLIENT	FLUSH	Client -> HostManager. Tell the HostManager to flush all host information by sending a FLUSH message to its server.
		NEWSERV	Client -> HostManager. Tell the HostManager to abandon its current server and choose a new server.
HM_STAT	HMST_CLIENT	GIMMESTATS	Client -> HostManager. Ask the HostManager to return statistics about itself.

Reserved Class Definitions (continued)

Class	Instance	Opcode	Effect
LOGIN	<ZID>	<EXPOSURE>	Client -> Server. Tell the server that the user named by <ZID> has logged in to the host listed in the message field. This notice must be authenticated. The opcode contains the exposure level, one of NONE, OPSTAFF, REALM-VISIBLE, REALM-ANNOUNCED, NET-VISIBLE, or NET-ANNOUNCED. The client data area contains the following NUL-terminated ASCII strings, in order: official hostname, login time, and display device.
		USER_LOGOUT	Client -> Server. Tell the server that the specified user has logged out. The client data area contains the same fields as specified above for USER_LOGIN.
		USER_FLUSH	Client -> Server. Tell the server to flush all location information for the user.
USER_LOCATE	<ZID>	LOCATE	Client -> Server. Ask the server to return all visible locations of the user named by <ZID>. This notice must be authenticated.
WG_CTL	USER	REREAD	Client -> WindowGram client. Ask the WindowGram client to re-read the description file (which specifies its action on receipt of notices).
		SHUTDOWN	Client -> WindowGram client. Ask the WindowGram client to cancel its subscriptions and to ignore subsequently delivered notices.
		STARTUP	Client -> WindowGram client. Ask the WindowGram client to reinstate subscriptions canceled by a SHUTDOWN notice and to display subsequently delivered notices.

Reserved Class Definitions (continued)			
Class	Instance	Opcode	Effect
ZEPHYR_ADMIN	“”	HELLO	Server -> Other Server. Inform the other server that this server is operating normally.
		IHEARDYOU	Server -> Other Server. Inform the other server that this server received its HELLO notice.
		GOODBYE	Server -> Other Server. Inform the other server that this server is ceasing operation.
		LOST_CLIENT	Server -> Other Server. Inform the other server that this server cannot contact a client and the other server should attempt to verify the failure of the client. The client data area contains two NUL-terminated strings: the Internet address of the client’s host, in Internet ASCII dot notation, and the port number of the client, in ASCII.
		KILL_CLIENT	Server -> Other Server. Inform the other server that this server has verified the failure of a client, and that the other server should remove the client from its database. The client data area contains information in the same format as noted above for LOST_CLIENT.
		STATUS	Client -> Server. Ask the server to respond with various statistics about its operation.
	<VERSION>	DUMP_AVAIL	Server -> Other Server. Inform the other server that a brain-dump is available. The instance is the version number identifying the brain-dump protocol. The client data area contains two NUL-terminated strings: the Internet address of the originating server, in ASCII dot notation, and the port number from which the brain-dump is to be obtained, in ASCII.

4. The Zephyr Server

The Zephyr server manages subscriptions, default subscriptions, user location information, and access control lists, and redistributes notices between clients.

4.1. Server Authority

Each server is authoritative only for those client hosts which have associated themselves with it. All other data the server holds is considered correct but may be invalidated at any time by command of the server which is authoritative. This scheme provides a distributed database between the servers. Each server may correctly redistribute any notice without consulting any other server. Incremental update is available, but without the complexity of maintaining complete consistency between all the servers, at the cost of associating with each piece of information an identification of the server which controls that information.

For efficiency, we desire that each server maintain the entire database of subscriptions and locations. But maintaining a fully distributed database with incremental update is too difficult to be worthwhile for Zephyr. So we have compromised our desire for distribution with the reality of maintenance and implementation constraints.

4.2. Server Initialization and restart

When a server is started, it either asks the name service *Hesiod* [1] or consults a file[4] to determine which other servers it should communicate with. It puts each other server into the DEAD state (see Figure 2), schedules an immediate HELLO packet transmission for each other server, and commences operation. In addition to the “normal” servers, each server maintains the state of a “limbo” server. This nonexistent server is used for storing the state of hosts which were formerly associated with a server which is now considered DEAD. If a server loses contact with another server for a significant amount of time, it considers that server DEAD, and transfers the information which was formerly controlled by the other server to “limbo.”

The server also initializes the access control lists of any registered classes in the class registry, and reads and caches the system default subscriptions from a file.

If two servers establish contact with each other, they exchange authentication information (to verify identities) and any authoritative data they hold and mark each other as UP. They may also exchange “limbo” information if one of the servers does not have any hosts in “limbo.” This allows the information associated with each host to be retained across server failures and restarts.

When exchanging information, a server may ask another server to send its idea of the server’s state and use that information as its authoritative information. This prevents loss of information if a server crashes and restarts before other servers notice its failure. If the failed and restarted server did not retrieve its state from another server, any client requests dependent on that lost state might be improperly dispatched or discarded.

The servers normally communicate using the standard Zephyr protocol. If a server receives any response from another server it previously thought DEAD, it opens a “brain-

[4]This choice is dependent on compile-time flags.

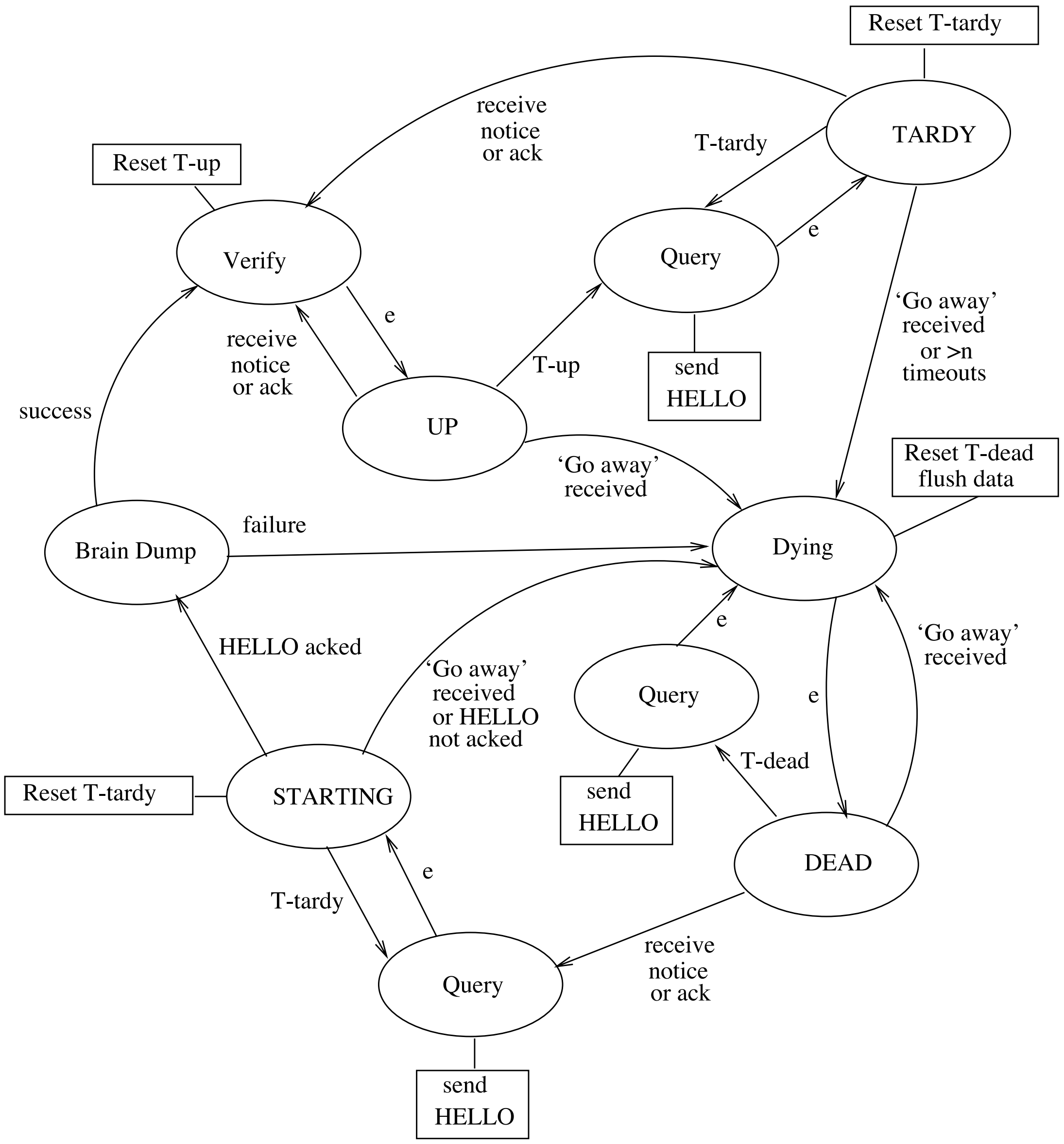
Figure 2: The Finite State Machine (FSM) of server states

Each server maintains a separate FSM for each other known server. There are four major states: UP, TARDY, DEAD, and STARTING.

Transitions labeled with ‘e’ are empty transitions which always occur, requiring no events or input.

Boxes attached to states indicate actions performed upon entering that state.

Timers are named T-xxx where xxx is a name representing the timer’s use. Resetting a timer means resetting the expiration of the timer to the full value. T-xxx labeling a transition means that transition is taken when the timer expires.



dump" socket and requests the other server to connect and initiate the "brain-dump" protocol. When communicating via the "brain-dump" protocol, servers use the Zephyr library routines for packing and unpacking notices, but instead of using the standard datagram transport they use a TCP/IP connection for reliable transport for the "brain-dump." If this "brain-dump" succeeds, each server schedules an immediate HELLO packet transmission to its peer, which when acknowledged will result in a state transition to UP.

4.3. Server Operation

The Zephyr server operates in a dispatch loop. Through its datagram socket it receives notices from clients and servers. These notices are dispatched via appropriate server routines. The server also processes previously set timeouts, such as a retransmission timer or a server contact timer.

4.3.1. Timers

The Zephyr server associates a timer with each other server. The timeout of this timer and the action taken when it expires varies depending on the state of the other server in the current server's FSM. Figure 2 shows the FSM and the actions taken.

Each notice which has been distributed to a client but not acknowledged has a timer which controls retransmission attempts.

Each inter-server update notice which has been forwarded to the other servers but not acknowledged has a timer which controls retransmissions.

4.3.2. HostManager interactions

Any notices which have been transmitted to clients but not acknowledged are queued and retransmitted several times with a short interval between retransmission attempts. If the notice remains unacknowledged, the server attempts to establish contact with the HostManager on the client's host. If the server is authoritative for that host, it sends a HostManager Ping packet to the hostmanager. If the server is not authoritative, it requests the authoritative server to send the ping. If the server believes the host to be in "limbo," the recovery attempt is aborted.

If the ping is acknowledged, the state associated with the unresponsive client is flushed. If the ping is not acknowledged after a short interval, the state associated with the entire host is flushed, under the assumption that the machine has crashed[5].

All notices originating from a HostManager are acknowledged. The acknowledgment may take one of several forms, depending on the disposition of the notice. If the notice was unauthentic or unauthorized and attempted a function requiring authentication and/or authorization, the acknowledgment is type SERVNAK with message LOST. If the packet assumed some internal server state and that state was not present, the acknowledgment is type SERVNAK with message FAIL. Otherwise the type is SERVACK, and the message is SENT if the notice was redistributed or processed without error or LOST if no clients were subscribed to that notice.

[5]Note that if only the HostManager was dead, and not the whole machine, the clients on that host would still have no way to transmit information via Zephyr, and the flush is still appropriate. See Chapter 5 for more details on the HostManager.

4.3.3. Notice handling

When datagram notices are received, the server invokes the appropriate dispatch routine to handle the notice, based on its origin and the contents of its header. The following table details the dispatch rules.

Dispatch rules

Notice origin	Notice class	Dispatch routine
A known server	(any class)	`server_dispatch`
Some other origin	HM_CTL	`hostm_dispatch`
	ZEPHYR_CTL	`control_dispatch`
	LOGIN	`ulogin_dispatch`
	USER_LOCATE	`ulocate_dispatch`
	ZEPHYR_ADMIN	`server_adispatch`
	<anything else>	`sendit`

When the notice originated from another server known to this server, **`server_dispatch`** performs more dispatch processing:

- If the notice is an acknowledgment of some previously transmitted notice (i.e. the notice has z_kind == SERVACK), the packet it acknowledges is removed from the retransmit queue and the server is marked as having responded (this may modify the state of the other server and cause a notice transmission, according to Figure 2).
- If the notice is of class ZEPHYR_CTL, LOGIN, or USER_LOCATE, it is passed on to the corresponding dispatch routine, after determining the true origin of the notice. This case occurs when a server has acted on a client request, and asks the other servers to update their databases to maintain synchronization by transmitting the notice it received from the client to the other servers.
- If the notice is of class ZEPHYR_ADMIN, it is passed on to **`admin_dispatch`**, which dispatches on opcode:
  - If the notice has opcode HELLO, the notice is a hello packet, and the FSM for the originating server is updated appropriately.
  - If the notice has opcode IHEARDYOU, the notice is an acknowledgment of a hello packet, and the FSM for the originating server is updated appropriately.
  - If the notice has opcode GOODBYE, the notice is a shutdown packet, indicating impending shutdown of the peer server. All the state associated with that server's hosts is put into the "limbo" state until those hosts contact a new server, and the server is marked as DEAD in the FSM.
  - If the notice has opcode DUMP_AVAIL, the server initiates a "brain-dump" connection to the peer server (see Section 4.4).
  - If the notice has opcode LOST_CLIENT, the server initiates the HostManager recovery protocol with the host of the client named in the body of the message. The body contains two strings, each NUL-terminated. The first is an ASCII representation of the address of the client in Internet ASCII dot-notation. The second is an ASCII representation of the port number of the client.

- If the notice has opcode KILL_CLIENT, the server flushes the state associated with the client named in the body of the message. The body is in the same format as the LOST_CLIENT notice.
- If the notice has any other opcode, its reception is logged and it is ignored.

hostm_dispatch does further dispatch work, based on the opcode of the notice:

- If the notice has opcode BOOT, the server flushes any state associated with clients on the originating host, and adds (or replaces) the host on its list of "owned" hosts for which it is authoritative, informing other servers that it has assumed "ownership" of the host. The other servers mark the host as owned by the server which has claimed ownership.
- If the notice has opcode FLUSH, the server flushes any state associated with the originating host, but maintains the owning server of that host.
- If the notice has opcode ATTACH, the server claims "ownership" of the host and informs other servers that it has done so. The other servers mark the host as owned by the server which has claimed ownership.
- If the notice is an acknowledgment of a server query resulting from a failed client (see Section 4.3.2), the server flushes the information stored for the failed client, and turns off the timer set when the query was sent.

control_dispatch does further dispatch work, based on the opcode and instance of the notice:

- If the instance is HM, the notice is passed to **hostm_dispatch** for processing.
- If the opcode is GIMME, the originating client is returned a list of his subscriptions.
- If the opcode is GIMMEDEFS, the originating client is returned a list of the system default subscriptions.
- If the opcode is SUBSCRIBE, the originating client is subscribed to the <class, instance, recipient> triples specified in the message body. The message body contains null-terminated strings. Each set of three is considered a triple, starting at the beginning of the message body. If the client had no previous subscriptions, the client is also subscribed to the system default subscriptions.
- If the opcode is UNSUBSCRIBE, the originating client is unsubscribed to the triples specified in the message body. The format of the body is identical to that of the SUBSCRIBE operation.
- If the opcode is CLEARSUB, the originating client is unsubscribed to all the triples he is currently subscribed to.
- If the notice has any other opcode, its reception is logged and it is ignored.

ulogin_dispatch does further dispatch work, based on the opcode of the notice:

- If the opcode is OPSTAFF, REALM-VISIBLE, REALM-ANNOUNCED, NET-VISIBLE, or NET-ANNOUNCED, a location for the ZID of the originating client is added to the location database, with the exposure set as the opcode directs, and other fields filled in as specified in the body of the notice. The body contains three NUL-terminated strings: the machine name of the client's host, the time of the exposure change, and the name of the display device in use by the client.

If the exposure is sufficiently broad, a login notice is fabricated for this user (with opcode set to USER_LOGIN) and transmitted to those clients which are both permitted to receive such notices and subscribed to login messages for that ZID.

- If the opcode is NONE, the location (as specified in the notice) of the ZID of the originating client is removed from the location database.
- If the opcode is USER_LOGOUT, the ZID of the originating client is removed from the location database, and if his exposure is sufficiently broad, the notice is retransmitted to those clients which are both permitted to receive such notices and subscribed to login messages for that ZID.
- If the opcode is USER_FLUSH, all locations for the ZID of the originating client are removed from the location database.
- If the notice has any other opcode, its reception is logged and it is ignored.

`ulocate_dispatch` does further dispatch work, based on the opcode of the notice:

- If the opcode is LOCATE, any locations of the ZID named in the instance of the notice which the sender is authorized to receive are sent to the port specified in the port field of the notice. Each location is returned in the message body as a triple {machine_name, login_time, terminal} of NUL-terminated strings. Successive locations are appended to the message body.
- If the notice has any other opcode, its reception is logged and it is ignored.

`server_adispatch` does further dispatch work, based on the opcode of the notice:

- If the opcode is STATUS, a notice containing server status information in the message body is returned as an acknowledgment to the originating client.
- If the notice has any other opcode, its reception is logged and it is ignored.

When the notice is some other class, the notice is passed to **`sendit`**, which checks authorization for transmission of this class, and then transmits the notice to all clients subscribing to it. Most notices will be of this type.

4.3.4. Subscriptions

The Zephyr server maintains a list of subscriptions for each client. Each subscription contains the class, class instance (possibly a wildcard), and recipient of the notices requested. *For security considerations, the recipient may only be either the ZID of the subscribing client or a null recipient.* A null recipient is effectively a multicast address. Any interested client may subscribe to a class/instance pair with a null recipient field (subject t access control restrictions on that class), and any notice transmitted to the same class/instance pair with a null recipient field will be redistributed to all those subscribed clients.

In addition to the subscriptions explicitly requested by a client (via a ZEPHYR_CTL/SUBSCRIBE notice), all clients are automatically subscribed to a set of default subscriptions (the list of default subscriptions is stored in a file on the server host.). A client may unsubscribe to any of these defaults via the standard ZEPHYR_CTL/UNSUBSCRIBE mechanism.

4.3.5. Classes and Access Control

There are two types of classes maintained by the server, registered and unregistered classes. Any client may send or subscribe to unregistered class notices. The class registry is a file specifying which classes are registered. Each registered class is associated with four access control lists. Each list specifies the ZID's with the authorization for the function associated with the list. It is permissible to place wildcards in the access control lists. The four authorization types are:

- SUBSCRIBE authorization. If a ZID is named in this list, it may subscribe to notices in the class. Otherwise, it may not subscribe to notices in this class.
- INSTWILD authorization. If a ZID is named in this list, it may specify a wildcard instance in a subscription to notices in the class. Otherwise, it may only subscribe to non-wildcard instances of the class.
- TRANSMIT authorization. If a ZID is named in this list, it may transmit notices of the class. Otherwise, it may not transmit notices of this class.
- INSTUID authorization. If a ZID is named in this list, it may transmit notices of this class with any instance. Otherwise, it may only transmit notices of this class with instance equal to its own ZID.

If the file representing the access control list specifying an authorization type for a given registered class is missing, all authenticated ZID's are granted that authorization, but no unauthenticated ZID's are granted authorization.

4.3.6. Instances

The server does not make any restrictions on instances except those imposed by access control lists.

4.3.7. User Locations

The Zephyr server maintains a table of user locations. Each entry includes the ZID of the client at the location, machine name, time of login, display device name, and an exposure level. All elements but the ZID are stored exactly as reported by the login notice. The ZID is extracted from the authentication information.

4.4. Server-to-server Download Protocol

When two servers establish contact, they initiate the Server-to-server Download protocol (Brain-dump protocol). If at any time during the brain-dump exchange an error occurs, the entire dump is aborted.

When the brain-dump is completed, each server will have nearly identical copies of the database information (the information assigned to "limbo" and to other servers not a party to the brain-dump operation may differ). In effect, the brain-dump re-plays all the operations necessary to establish the state present in the servers.

The initiating server sends its peer an opcode "DUMP_AVAIL" notice (described below in Section 4.4.1). When a server receives such a notice from another server, it attempts to connect to the indicated TCP port. Upon success, it sends a Kerberos ticket and authenticator to the peer, and waits for a ticket and authenticator in return.

When a server which is listening to a TCP socket receives a connection request, it accepts the connection and closes the listening socket. It then reads an authenticator from the newly accepted socket, and if authentication yields the principal “zephyr.zephyr” in the server’s Kerberos realm, it forms an authenticator and sends it to the peer[6].

After authentication succeeds, the peers begin the actual data transfer. The server which bound and listened to a socket begins a send loop, and its peer begins a receive loop. When this transaction is complete, the roles are reversed.

The data transmission is accomplished by using the standard Zephyr library routines to format and decompose notices into and from a character stream. Each formatted notice is transmitted via the TCP connection to the peer, preceded by a two-byte length field (in network byte-order) specifying the length in bytes of the following packet.

The server in the receive loop (the receiver) asks the sender for data in separate “chunks.” The receiver may, at its option, ask for the “LIMBO” data and “MY_STATE” data (data on hosts assigned to “limbo” and those assigned to the receiver), in that order. It then must ask for “YOUR_STATE” (data on hosts assigned to the sender).

Servers which have no hosts in the limbo state will request “LIMBO” data. In this way, hosts which have not been heard from since a server crash-restart cycle are retained in the database. Servers which have never received any information from another server will request “MY_STATE” data. In this way, if a server restarts quickly, before a peer notices his absence, his authority is retained for those hosts which have not yet chosen a new server.

For each chunk request, the receiver enters a loop. Inside this loop it expects one of the following:

- A boot notice for a host.
- A login notice for a user. The receiver must have received a host boot notice before it will accept a login notice.
- A client registration notice (Opcode “NEXT_CLIENT”). The receiver must have received a host boot notice before it will accept a client registration notice.
- A subscription notice. The receiver must have received a client registration notice before it will accept a subscription notice.
- An end of dump notice (Opcode “DUMP_DONE”).

When an opcode “DUMP_DONE” notice is received, the loop is exited, and the next chunk is requested. If the receiver desires no more “chunks”, it is finished, and the roles reverse (if appropriate).

The server in the send loop waits for a request for a chunk. It then iterates over all hosts associated with the chunk requested, sending a host boot notice, then all the locations on that host, and then sends client registration and subscription notices for each client on that host. When finished with the set of hosts, it sends an opcode “DUMP_DONE” notice.

[6]This is a crude form of mutual authentication; in the future a more secure form of mutual authentication involving decrypting, modifying, and returning random data will be used.

4.4.1. Server brain-dump notice definitions

Following is a table describing notices used in the brain-dump protocol which are not part of the reserved class definitions (Section 3.4). All classes, instances, and opcodes used are NUL-terminated ASCII strings.

Messages used for Brain Dumps			
Class	Instance	Opcode	Effect
ZEPHYR_ADMIN	<VERSION>	DUMP_AVAIL	Inform the other server that a brain-dump is available. The instance is the version number identifying the brain-dump protocol. The sender is the hostname of the originating server. The client data area contains two NUL-terminated strings: the Internet address of the originating server, in Internet ASCII dot notation, and the TCP port number from which the brain-dump is to be obtained, in ASCII.
ZEPHYR_ADMIN	LIMBO	DUMP_AVAIL	Request the other server to send the state associated with hosts in "LIMBO."
	YOUR_STATE	DUMP_AVAIL	Request the other server to send the state associated with hosts assigned to the other server.
	MY_STATE	DUMP_AVAIL	Request the other server to send the state associated with hosts assigned to this server.
	"" (empty string)	DUMP_DONE	Indicate that this phase of the brain-dump (or the entire brain-dump) is complete.
	CBLOCK	NEXT_CLIENT	Begin a subscription transfer for a new client. The sender field is the client's ZID. The client data area contains two NUL-terminated strings: The client's port number, in ASCII, and the client's DES session key, in ASCII. The client's Internet address is assumed to be the address of the most-recently registered host.

5. The HostManager Client

5.1. Overview

Each host in the network that supports Zephyr clients runs a Zephyr HostManager Client. When a host boots it starts a HostManager which contacts a Zephyr server (designated by the Hesiod name server), sending it a <ZEPHYR_CTL,HM,BOOT> notice specifying that the server should flush all states previously associated with this host.

The HostManager is the focus for all outgoing messages from the local machine. Messages are sent from client programs via Zephyr Client Library function calls to the HostManager, which sends an acknowledgment to the library routine. It then relays the packet to the current “owning” server. The server sends an acknowledgment back to the HostManager. The HostManager uses this to make sure that the server is still responding. It then forwards the acknowledgment to the client, unless the notice type specified that the acknowledgment should not be forwarded.

Whenever the HostManager fails to receive an acknowledgment of a transmitted notice, it retransmits the notice several times, after which time it seeks out a new server.

The Zephyr server may occasionally send an <HM_CTL,SERVER,PING> notice to the HostManager if it doesn’t receive an acknowledgment from a client program. The HostManager simply sends back an acknowledgment notice.

Upon receipt of a SIGHUP, the HostManager sends a <ZEPHYR_CTL,HM,FLUSH> notice to the server, to indicate that all information about this host should be flushed. This can be used to insure that any stale information in the server database is cleaned up when workstations are no longer in use. The HostManager then goes into the “deactivated” state. In this state, it remains unattached to any server until a client sends a notice to be forwarded to a server, at which point the HostManager contacts a server, sends it a boot notice, and forwards the client’s notice.

5.2. HostManager Subsystems

The HostManager is driven by a blocking read function call which waits until a datagram arrives on its input port. The packet is checked for the host of origin. If the host is the local loopback address (Internet address 127.0.0.1), then the packet is sent to the transmission tower routine, otherwise the server manager is called.

5.2.1. Transmission Tower

The transmission tower routine is called whenever a client program sends a notice destined for a server. If the HostManager is in contact with a server, it forwards the notice to the server. Whether the HostManager sends the notice or not, it adds the notice to a queue of unacknowledged notices, to be used should retransmission be necessary.

If the notice is of kind HMCTL, the notice is handled internally by this routine. <HM_CTL,CLIENT,NEWSERV> notices cause the HostManager to call the New Server routine. <HM_CTL,CLIENT,FLUSH> notices cause the HostManager to send a <ZEPHYR_CTL,HM,FLUSH> notice to its server, and go into “deactivated” state.

5.2.2. Server Manager

The server manager routine is called whenever a notice arrives from a Zephyr server port on a machine other that the local host. If the originating host is not the current server machine and it is not an acknowledgment (kind SERVACK or SERVNAK) or HMCTL (kind HMCTL) notice, the receipt of the notice is logged and it is ignored.

The HostManager expects only three notice types from the server. These are HMCTL control notices and packet reception acknowledgments, which can be either SERVACK or SERVNAK types.

If an HMCTL notice is received, the notice opcode is examined. If it is “SHUTDOWN”, then the New Server routine is executed. If the opcode is “PING”, then the HostManager sends an HMACK notice back to the sender. If the HostManager was not in contact with a server before the HMCTL notice was received, the queue of unacknowledged notices is resent, unless the notice received from the server is a SERVER_SHUTDOWN message.

If the packet received is an acknowledgment of a packet sent earlier, the HostManager checks to see if the client wants the acknowledgment, and if so sends the acknowledgment packet to the client. It then removes the packet from the queue of unacknowledged notices.

5.2.3. New Server Routine

The new server routine is called when the server manager receives an announcement that the server is going down, when a packet is not acknowledged and the HostManager decides to change servers, or when a client sends a new server notice to the HostManager.

The routine first sends an unacknowledged <ZEPHYR_CTL,HM,DETACH> notice to the current server so that the server can remove its references to this HostManager provided the server has not crashed, but is just overloaded. The HostManager then sends another Zephyr server a <ZEPHYR_CTL,HM,ATTACH> notice. When the new server acknowledges the attach notice, it becomes the “owning” server. The HostManager then sends all of the unacknowledged packets to this new server, to insure that all of them get delivered.

5.2.4. Queue Routines

Whenever a client sends a packet destined for a server to the HostManager, it is placed in a queue and then forwarded to the server. The queue is a linked list, each element consisting of a timeout field, a number of retries already sent, and the unacknowledged packet. The timeout field is set to the current time (in seconds) plus the number of seconds to wait before a timeout should occur.

Whenever a timeout occurs, each entry in the queue is examined, and the number of retries is incremented in each one which has a timeout value less than the present time. If the number of retries is greater than the configured safety margin, the New Server routine is called. If the number of retries is less than the safety margin, the packet is resent to the server.

Whenever a new server has been successfully contacted, the entire queue is resent. Each entry in the queue is sent, its number of retries is reset to zero, and the timeout field is set to the proper value.

6. The Zephyr Client Library

The Zephyr Client Library, a C language function library, implements the Zephyr protocol. Specific function interfaces are described fully in the *Zephyr Programmer's Manual*.

6.1. Functionality provided

To be gleaned from the *Zephyr Programmer's Manual*.

7. The WindowGram Client and Browser

7.1. Overview

The Zephyr WindowGram client is the primary Zephyr user application. It displays incoming notices on the user's screen as directed by a description file. The WindowGram browser allows the user to save notices and review them later. The WindowGram client and browser are the only sections of the Zephyr system that will be immediately obvious to the naive user.

7.2. Client Operation

When a user logs in, a WindowGram client is automatically started for him or her by the standard initialization files. The WindowGram client registers the user with the user location database, using the last exposure set by the user (or a system default if the user has never set his own exposure), sends the user's standard subscriptions to the server, executes the user's initialization program (or the system initialization program if the user has not specified his own), reads the user's description file and the system default description file.

The WindowGram client then waits for incoming notices on an allocated port. When a notice is received, it is matched against the description file and, if no match was found, against the system default description file, and is acted upon appropriately.

In addition to notices delivered by the normal client-server path, the WindowGram client may also receive incoming packets directly from user applications programs that instruct it to perform special functions.

The WindowGram client will work whether or not the user is using an X Window System ("X") display. If the client is not using X, a simple terminal-based display (similar to the UNIX *write* utility) will be provided. The user may, if desired, select the simple terminal-based display while using an X display.

If the client is using X, the WindowGram browser can be used. The browser waits for commands from the user (via mouse or keyboard), allowing the user to scroll through the notices he has received, discard those he doesn't want, or save certain notices in files.

A similar browser will be available in the future as a separate application for clients not running X.

When the user logs out, the system delivers a SIGHUP to the WindowGram client, which then deregisters the user from the user location service, and cancels all its subscriptions.

7.3. WindowGram Subsystems

The WindowGram client is composed of the following subsystems. The client waits until any of several input channels (file descriptors) is ready, and then the appropriate subsystem is invoked to respond to the channel.

7.3.1. Notice Handler

When the WindowGram client receives a notice, its class, instance and origin are checked. If the class is “WG_CTL,” the instance is “USER” and the origin is the local host, the notice is passed on to the WINDOWGRAM_CTL handler. Otherwise, the notice is displayed by the Display Handler as directed by the description files.

7.3.2. WINDOWGRAM_CTL Handler

When the WINDOWGRAM_CTL handler is invoked, the notice opcode is examined, and the following actions are taken:

- If the opcode is “REREAD,” the user’s description file is re-read.
- If the opcode is “SHUTDOWN,” the user’s current subscriptions are retrieved and saved, and then the subscriptions are canceled. In addition, the client ignores all further notices except “WG_CTL” notices. This state is called “catatonia”
- If the opcode is “STARTUP,” the saved subscriptions are re-subscribed to and the client resumes normal operation. If the client was not in “catatonia,” there is no effect.

Because these notices are sent directly to the client from another client on the same host, there is no need to subscribe to these notices.

The WindowGram client will make sure that the packets are authentic (from the proper user) before performing any of the above operations.

7.3.3. Display Handler

When a notice arrives, the Display Handler composes a displayable form using the format specified in the description file. It then displays the notice. The actual display is based on the type of display interface and description file used. The display interfaces available are:

- X Window System Display
- Generic terminal

Any other interface that can control a file descriptor (for *select(2)*) or interrupt the program can be integrated into the existing client.

8. Acknowledgments

The authors would like to acknowledge the following people from MIT Project Athena for their help in making Zephyr a reality: Michael R. Gretzinger former Systems Programmer and David G. Grubbs former Manager of Systems Integration for their input into the initial concept of a Notification Service. Dan Geer, the Manager of Systems Development at MIT Project Athena for his undying support of our efforts.

9. References

[1] S. P. Dyer and F. S. Hsu.
Hesiod Name Service.
In J. H. Saltzer (editor), *Project Athena Technical Plan*, chapter Section E.2.3. M.I.T. Project Athena, 1987.

[2] P. Levine, M. R. Gretzinger, J. M. Diaz, B. Sommerfeld, and K. Raeburn.
Service Management System.
In J. H. Saltzer (editor), *Project Athena Technical Plan*, chapter Section E.1. M.I.T. Project Athena, 1987.

[3] W. E. MacKay.
An Educational Communication System That Integrates Electronic Mail, On-Line Consulting, Electronic Meetings and Educational Software..
Project Athena Working Paper 87-4, MIT Project Athena, 1987.
Work in progress.

[4] S. P. Miller, B. C. Neuman, J. I. Schiller, and J. H. Saltzer.
Kerberos Authentication and Authorization System.
In J. H. Saltzer (editor), *Project Athena Technical Plan*, chapter Section E.2.1. M.I.T. Project Athena, 1987.