Activity Server: A Model for Everyday Office Activities

by

Sanjay Manandhar

B.S., Electrical Engineering and Computer Science
Massachusetts Institute of Technology
(1989)

Submitted to the Media Arts and Sciences Section
School of Architecture and Planning
in Partial Fulfillment of the Requirements for the Degree of

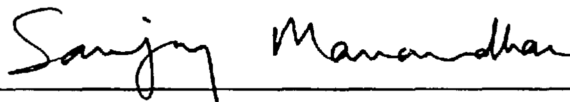
Master of Science

at the

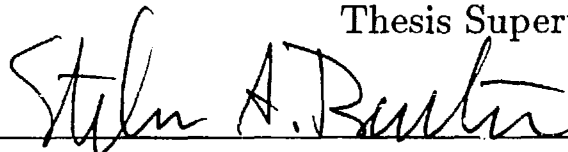
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 1991

© Massachusetts Institute of Technology 1991
All Rights Reserved

Signature of Author 
Media Arts and Sciences Section
May 10, 1991

Certified by
Christopher M. Schmandt
Principal Research Scientist
Thesis Supervisor

Accepted by 
Stephen A. Benton
Chairman, Departmental Committee on Graduate Students

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

ARCHIVES JUL 23 1991

LIBRARIES

Activity Server: A Model for Everyday Office Activities

by

Sanjay Manandhar

Submitted to the Media Arts and Sciences Section
School of Architecture and Planning
on May 10, 1991, in partial fulfillment of the
requirements for the Degree of
Master of Science

Abstract

The activity server is a program that produces periodic reports of people's locations and activities. It combines three sources of information. A finger server provides information on users' activities on computers within a local network; a phone server tells whether phones are in use; and a location server abstracts physical location of users who wear "active badges." Details of the design and implementation of the finger server (the one built expressly for the activity server) will also be discussed.

The main attributes of the activity server are: 1) it maintains history of all the participating users, 2) it coordinates among the many, possibly conflicting, pieces of information from three sources, 3) it distinguishes interdependence of activities among users, and 4) it provides a high-level abstraction about users' activities. This thesis also describes two client applications that allow means of retrieving the knowledge embedded in the activity server; the clients use a graphical and a telephonic user-interface, respectively.

Thesis Supervisor: Christopher M. Schmandt
Title: Principal Research Scientist

Contents

1 Introduction **7**

1.1 An Example 8

1.2 Background 10

2 Overview of the system **11**

2.1 The Listeners 11

2.1.1 The finger server 13

2.1.2 The phone server 14

2.1.3 The location server 15

2.2 Data Representation 16

2.2.1 The generalized linked list 16

2.2.2 Why a linked list? 17

2.3 The Socket Manager 18

3 The Finger Server **20**

3.1 Architecture 20

3.2 Mechanism 21

3.2.1 Configuration files 21

3.2.2 Host data 21

3.2.3 Polling 23

3.2.4 Callbacks 24

3.3 Parsing Finger Information 26
3.4 Maintaining Finger Information 28
3.5 Updating Finger Information 29
3.6 Alerts 30
3.7 Protocols 31
3.8 Client Interaction 32
3.9 Shortcomings 33
3.10 Optimizations 34

4 The Activity Server **36**

4.1 Architecture 36
4.2 Mechanism 37
4.2.1 Startup 37
4.2.2 Event handling 39
4.3 Data Representation 39
4.3.1 Server object 40
4.3.2 Person object 40
4.3.3 Place object 41
4.3.4 Machine object 41
4.3.5 Phone object 42
4.4 Updating Data Structures 42
4.4.1 Finger events 43
4.4.2 Phone events 43
4.4.3 Location events 45
4.5 History Mechanism 45
4.6 Client Interaction 46

5 Rules **48**

5.1 Scenario 1: Coming into the Office 49

5.2 Scenario 2: Leaving the Office 50
5.3 Scenario 3: Visiting Another Office 51
5.4 Scenario 4: Visitor in Office 51
5.5 Scenario 5: On the Phone 52
5.6 Scenario 6: Logged in on Another Machine 53
5.7 Scenario 7: Dialed In 54
5.8 Scenario 8: Remotely Logged In 54
5.9 Final States 55
5.10 Conflict Resolution 56

6 Clients **58**

6.1 The Directory Client 58
6.1.1 Implementation 58
6.1.2 User Interaction 59
6.2 The Watcher Client 60
6.2.1 Implementation 60

7 Discussion **62**

7.1 Performance 62
7.2 Future Work 63

8 Summary **66**

A Finger Server Protocol **70**

A.1 Introduction 70
A.2 Byte-stream Protocol 70
A.3 Output Formats 71
A.4 Asynchronous Output 71
A.5 Messages 71
A.6 Programmer Interface 73

B Activity Server Protocol **77**
B.1 Introduction 77
B.2 Byte-stream Protocol 77
B.3 Output formats 78
B.4 Current Activity 78
B.5 Messages 78
B.6 History of Activity 80

C Configuration Files **81**
C.1 Finger Configuration Files 81
C.1.1 Hosts Configuration Files 81
C.1.2 Users Configuration Files 82
C.2 Activity Server Configuration Files 82
C.2.1 Places Configuration Files 82
C.2.2 Users Configuration Files 83

Chapter 1

Introduction

The activity server is a program intended to improve interaction of co-workers, given that richer forms of communications like meeting in person, talking over the telephone and electronic mail, etc., are not always possible. It is intended to answer questions like,"Can I have a meeting now with colleagues A, B and C?" or "Where is colleague C?" or "Who came by my office while I was away?" Such everyday questions constantly arise in an office environment and require active participation from the solicitor of the information. It is highly desirable to have instant answers or at least intelligent inferences to such questions. By gathering and maintaining information from three sources that are indicative of office activity, the activity server helps answer many routine questions of the work place.

There are some very well-defined, routine activities in the office. Logging on to a computer to check electronic mail, placing phone calls, visiting another colleague in another office or meeting in a conference room are some routine office activities. If these routine office activities could be collected and characterized in some fashion, applying some rules and heuristics, one could devise a tool that could augment coordination and query of user activities. This thesis describes such a system, and some client applications which use it.

The inputs to the activity server are reports of routine office activities. Its output

is assertions about the state of the activity. The reports of these office activities are asynchronous and are called events. These events are gathered from input channels by three servers: the finger server provides input from the local area network of computers; the phone server recovers phone activity from the telephone network and the location server monitors special badges that relay their positions relative to a separate network of sensors.

The abstracted outputs of the activity server are made available to client applications via a well-defined set of protocols. It is hoped that the outputs and the clients that use these outputs can benefit small trusting groups of users in the office environment.

At this point, some recurring terms ought to be described. Their full description will ensue in due course. For clarity, the three data gathering sources, the finger server, the phone server and the location server, are called Listeners and unless explicit, the "server" shall refer to the activity server whereas the "clients" shall refer to the client applications of the activity server. An "active badge" is a badge, which periodically emits a coded infra-red signal. It can be tracked by a network of sensors.

1.1 An Example

Let's take a simple example from above: Where is colleague P? To answer this question, the activity server queries its three Listeners. The following are possible answers that it receives:

Phone server: The phone in P's office is not in use.

Location server: Badge P is in P's office; it has been there for the last 10 hours.

Finger Server: P is logged in to the machine in her office, idle time is zero minutes (connected to terminal d0).

The activity server draws conclusions about the location of colleague P from each Listener response and assigns a confidence level to each. From the phone server alone, it is not clear that P is in her office. From the response of the location server, P is, in fact, in her office but has not moved for 10 hours. The latter piece of information undermines the plausible conclusion that she might be in her office. These are conflicting conclusions. But the finger server responds that the idle time on her machine is nil, i.e., she is active. However, she is not active on the console (located in her office), but rather on a pseudo-terminal d0. Since all dX (X is an alpha-numeric) pseudo-terminals are open for dialin access, P has dialed in. Hence, the activity server concludes that P is not around, she has dialed in remotely.

The example above illustrates some of the key attributes of the activity server:

- History: The server needs past as well as present data. This is important in order to be able to infer the user's activities from his past actions.

- Multiple sources: The server will draw upon its Listeners to get multiple, possibly conflicting, views on real world activities.

- Multi-party: The activity server monitors many users simultaneously. Every member of the group may affect the state of other members of the group. For instance, a person visiting another colleague not only changes his own, but also the state of his colleague.

- High-level abstraction: The activity server can assemble many pieces of discrete information, which by themselves are of little value but cumulatively allow abstraction of intelligent, high-level inferences of user activities.

1.2 Background

Work in traditional artificial intelligence (AI) has concentrated on plan recognition where it is assumed that there is a precise plan that a user activity will follow [6, 12]. Many researchers working in interdisciplinary fields have proposed models varying in scope, complexity, and computational formality. For instance, in a technical paper from Xerox PARC, many dynamic models at micro and macro levels were provided [5]. Some newer work argues against planned models since it is very difficult to allow for unpredictable and unanticipated circumstances [1, 2, 15]. Although not a user modeling effort, the activity server uses a dynamic model that can infer the state of users and adapt continuously. Most of the user modeling work in AI and cognitive science research, including dialog systems [9], expert systems and student modeling have focused on systems that monitor user activities, and react to them. The activity server only monitors and builds an activity model; however, clients of the activity server that benefit from the user model can execute (react) accordingly. In addition, unlike many user modeling efforts that attempt to enhance user-system interaction or understanding, the activity server focuses on augmenting user-user interaction with the help and coordination of many discrete systems.

Many user modeling research efforts depend on a single source of data. There has been much work that rely on a single source of information, such as user-input at a computer, eye tracker information, etc. The activity server, however, benefits from three different sources of information. This redundancy is used effectively to increase reliability of the final activities assertion made by the server.

Chapter 2

Overview of the system

The activity server adopts a client-server model based on underlying TCP-IP socket communication [3]. The activity server itself is a client to its three Listeners (the finger server, the location server and the phone server). Events from these Listeners are funneled through a common event handler which timestamps each event, updates a per user database and triggers some rules (see Figure 2-1). High level inferences made by the rules are saved and are made available to clients.

2.1 The Listeners

Listeners are the indispensable information-gathering servers that the activity server relies on. They are independent and service their own pool of clients; the activity server is a special client because it connects to all the Listeners simultaneously. Each Listener is self-sufficient and is oblivious of other Listeners; only the activity server, which collects data from all the Listeners, has the global view which allows it to make more complex inferences. The rest of the section provides an overview of all the Listeners.

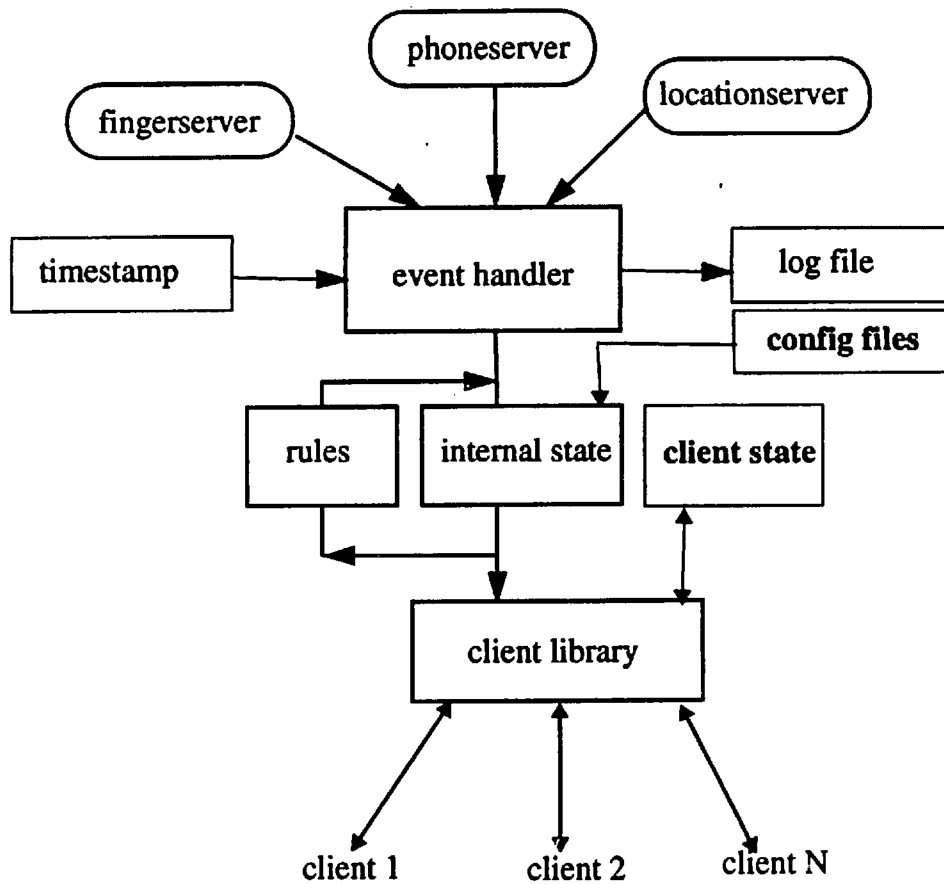


Figure 2-1: Cverall system diagram of the activity server

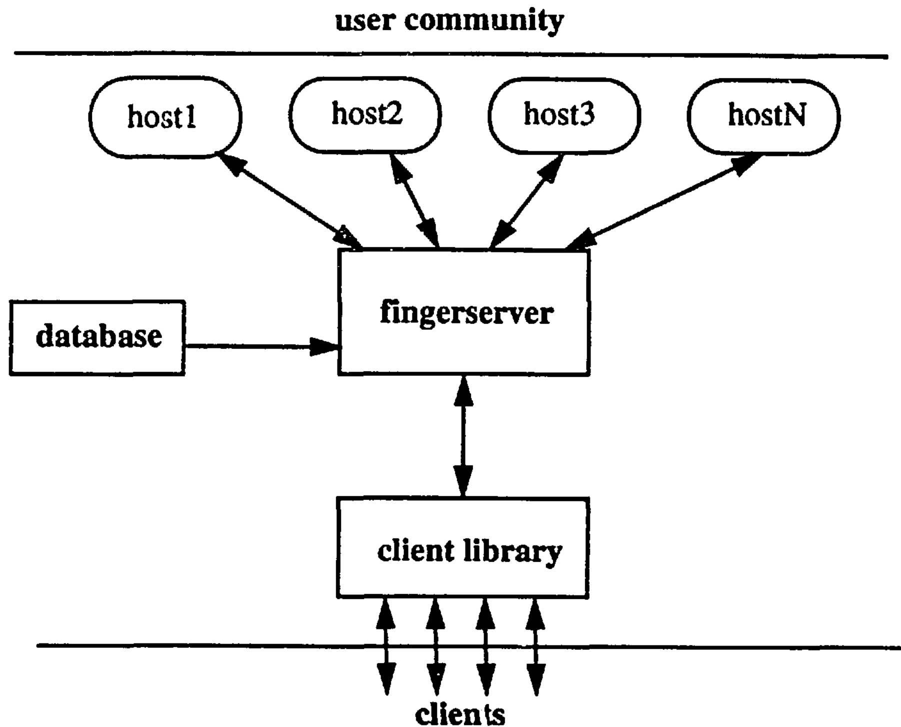


Figure 2-2: Architecture of the finger server.

2.1.1 The finger server

The finger server gathers idle times of users on a number of machines; it is a metric of activity of users on these machines. This server depends on the finger daemon [8] which keeps state (idle times in minutes) of all users on a given machine. The finger server compares information across the entire group of machines it is monitoring (Figure 2-2). For instance, if a user is logged in on more than one machine, the server can tell where the user is most active, whether he is logged in remotely and from where he is logged in. In addition, the finger server can send asynchronous events (called alerts) to clients that express interest in them. Since this server was implemented expressly for the activity server, it will be described thoroughly in the next chapter.

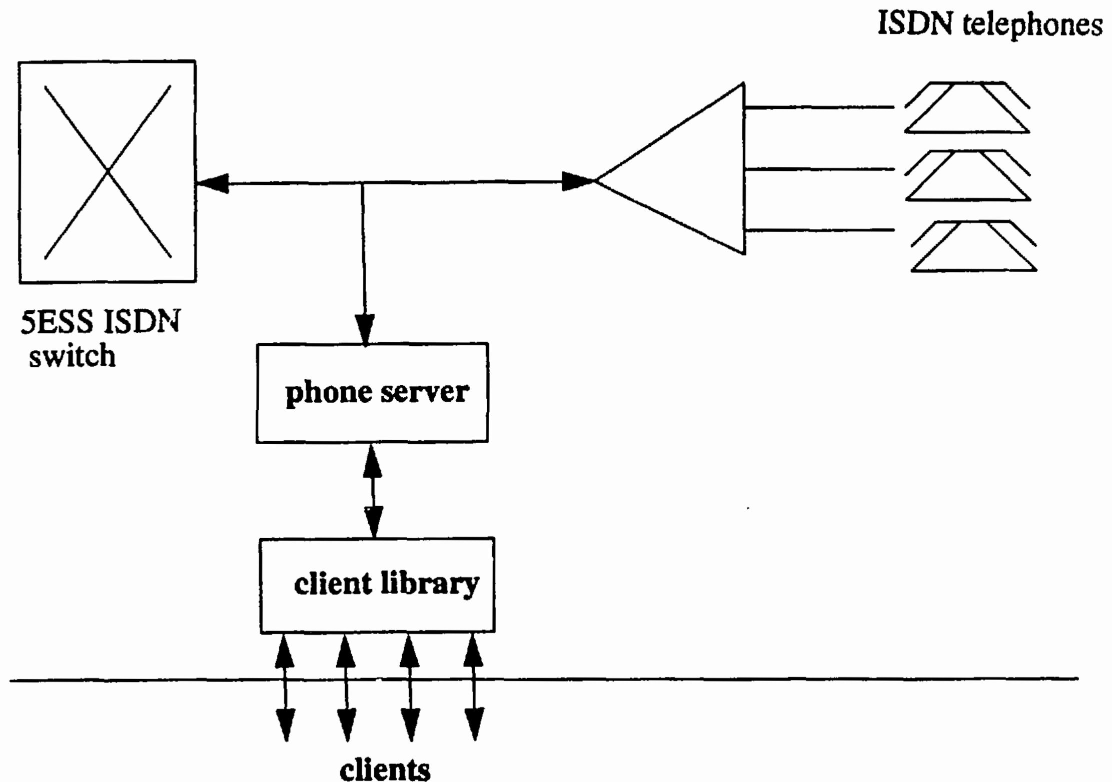


Figure 2-3: Architecture of the phone server.

2.1.2 The phone server

The phone server monitors the ISDN protocol that is exchanged between the 5ESS telephone switch and telephone sets. It gets the events of all telephone sets within its jurisdiction (see Figure 2-3). Phone state information such as idle, active, incoming call, dialing, held, etc. can be received from the phone server. For the purposes of the activity server, however, only idle, active and dialing events are critical. Inferences based solely on these three events will be discussed in a separate chapter dealing with the activity server. A crucial assumption that is made is that most users in the user community have their personal phones in their personal offices. For a more complete treatment of the phone server refer to [18].

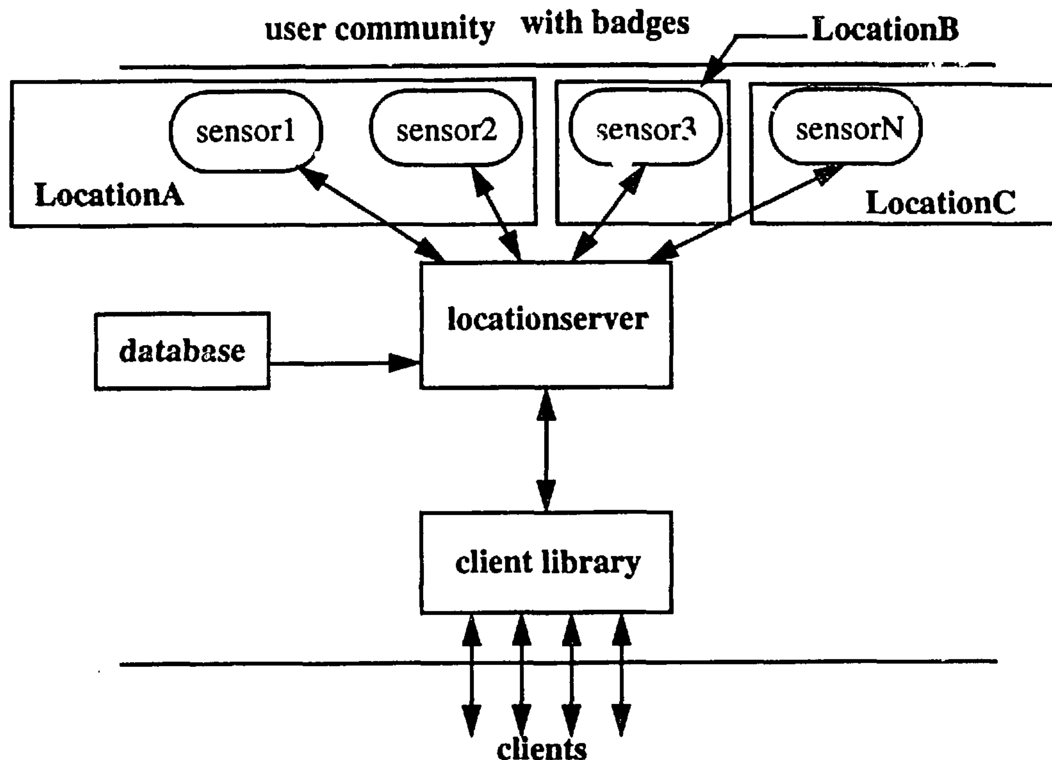


Figure 2-4: Architecture of the location server

2.1.3 The location server

The location server provides data about locations of special badges which emit infra-red signals. It has a network of infra-red sensors that are placed around the building, in hallways, offices, lab areas and conference rooms. Participating members of the user community may choose to wear a badge, called the active badge. Since each active badge has a unique code, this code and the owner of the badge can be mapped in a database. When a user wearing his badge (designed and built by Olivetti Research Laboratory in Cambridge, England), moves around in the building, and the badge is sighted by sensors, the sensors register the unique ID of the badge and report it to the location server. The location server translates the identity of the reporting sensor to its location and translates the ID of the badge to its owner, thus ascertaining the location of the user (see Figure 2-4). The sensors are polled every 5 seconds to check badge sightings; the badges themselves fire every 15 seconds.

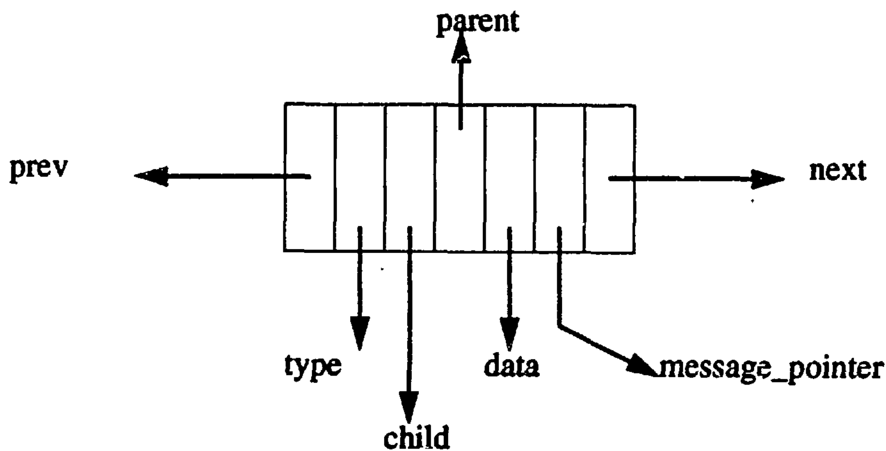


Figure 2-5: The link-node: a building block for the generalized linked list.

2.2 Data Representation

Representation of all dynamic data on the finger server as well as the activity server is via objects that are appended to linked lists. One of the design decisions was to pick a data structure that would act as the "common denominator" or a very general mechanism for storing any type of dynamic data. Hence, a generalized linked list was conceived; it provides the "chain" on which any data could be appended. This linked list is described below.

2.2.1 The generalized linked list

The fundamental building block of this generalized linked list is the link-node. It provides facilities for a quadruply linked list with message passing capabilities. Since it is not associated with any data object, it can be used to create linked list of any object contributing to generality and reusability of this model (and its corresponding code). This model represents a slightly general version of a pair of C library calls of "insque" and "remque."

The link-node consists of the type identifier to uniquely identify the data object that is appended to the link (see Figure 2-5). There are parent and child pointers for vertical connectivity and previous and next pointers for lateral connectivity; thus, in the most complex case one can have a quadruply linked list. A message handler points to a table of handlers. Depending on the message type, the appropriate operator for the data object at hand is triggered. Finally, the data part is a pointer to the data object. Whenever the internals of the data object are accessed or changed, the data field of the link must be properly type-casted; but this is a small price to pay for flexibility that the link-node model affords. For instance, the routines that add, delete, select and collect (select all) links (and by extension, their data objects) are the same no matter what kind of data is "carried" by the linked list. Note that the link-node model is not responsible for creation of the actual data objects or the message handler if there is one. It does, however, manage all the link-nodes and destroys any appended any data objects when the link-node itself is destroyed.

2.2.2 Why a linked list?

There are some very compelling reasons for opting the linked list data structure rather than other data structures. These reasons are as follows:

- Isolation: The linked list operations do not access the data objects. In usual implementations, the object itself is responsible for linked list operations (e.g. add, delete, move a link, etc.). Under the current scheme, however, the data objects are oblivious of other object (previous, next, parent or child) around them; only the link list nodes need worry about other nodes.

- Reusability: Most operations are common regardless of the data type. The same add, delete, select, collect operations can be reused.

- Flexibility: Linked lists in general are appropriate for managing dynamic data. Both the activity server and the fingerserver (the two systems built for this

project) use this modified linked list.

- Access time: Since the number of data objects, hence, the number of links, in the linked list is small, the access time for walking down the linked list is negligible. For larger numbers of discrete objects, data structures optimized for larger object domains (e.g. hash tables) would be more appropriate, however.

2.3 The Socket Manager

Like the general nature of the linked list described above, the socket manager[1] provides a general, powerful interface to perform TCP-IP socket communications. This interface makes the intricacies of socket communication manageable and comprehensible. The socket manager provides enough features that the temptation to create yet another flavor of socket connection that may duplicate functionalities and effort is slim. Needless to say, the activity server and the finger server use the the socket manager interface whenever the opportunity arises. Some of the attributes that contribute to a powerful socket manager are worth mentioning.

- Abstract operations: The intricate socket-level calls have been subsumed by more general calls that are understandable and easy to use.

- Asynchronous: Unlike procedural routines, the socket manager provides callbacks (for reading from sockets, carrying out housekeeping when the socket is destroyed and reporting errors) which are fully asynchronous. Asynchronous notification allows the server to do other duties when not servicing a request. The client, too, need not wait for a reply; it gets notified when the reply arrives.

- Symmetrical: Operations that exist on both the server and the client sides (e.g. opening, closing socket, registering callbacks, etc.) are the same. Such a symmetry makes the software semantically consistent as well.

[1]The Socket Manager code was by Ayisi Makatiani and Barry Arons.

- Reusability: As alluded to earlier, the yield of the module because of reusability is immeasurable. In the finger server and the activity server, the socket manager is used in two levels of server-client hierarchy.

Chapter 3

The Finger Server

Idle times provide a very simple and effective means of finding out the activities of users within a network (also known as fingering). Many machines provide a finger daemon that can be queried to retrieve history in idle times of all users on their respective machines. The daemon provides this service at a well-known port (79)[1], which the finger server uses to gets its data.

3.1 Architecture

The architecture of the finger server is based on two levels of client-server paradigm: the finger server is a client to the finger daemons while it is the server to its own clients. Communication between the finger server and the finger daemons of each of the machines, and between the finger server and its clients is via sockets. The polling module within the finger server starts polling all the machines soon after startup. Data is received asynchronously; then the parsing module recovers important pieces of information such as usernames and idle times and updates internal data structures. The internal data structure consists of three linked lists storing information about users, hosts and clients. The interaction of these data structures among themselves, with

[1]The Unix program "finger" is also implemented by querying the finger daemon of machines.

client programs and with incoming finger information is basic to the architecture of the finger server. Upon each update, a consistency check is done, the side effect which are asynchronous events called alerts.

3.2 Mechanism

The finger server amasses finger information from many, possibly non-homogeneous, machines and stores the data in a manner that can be readily available upon request from clients. This mechanism of amassing finger information is done by polling the machines; the results of the pollings are saved for three principal entities: hosts, users and clients. Their state is maintained in doubly linked lists; queries from clients are serviced by accessing data from these linked lists. The following subsections present some design choices and implementation details of the finger server.

3.2.1 Configuration files

When the server starts up it reads two configuration files that maintain the lists of users and hosts. These text files are *.finger_config.users* and *.finger_config.hosts*, respectively. The syntax and examples of these files are given in Appendix C. While the user configuration file lists only users (by username), the host configuration file maintains not only the hostnames (or their aliases), but also other optional fields such as polling frequency (in seconds), host type (Unix, Ultrix, Genera, etc.) and options for the finger daemon. Once the server has started, users and hosts may be added dynamically using the client protocols.

3.2.2 Host data

Host configuration parameters and other dynamic host-specific data such as socket identifier, time of last poll is stored in a HostConfig data object (see Figure 3-1). The HostConfig data object maintains polling-specific information - it is not responsible

```
typedef struct _HostConfig {
    char real_name[MAXCHARS];/* official name of host (not aliases)*/
    char options  [MINCHARS];/* options for the finger daemon      */
    int  type;               /* type of host (for parsing purposes)*/
    int  freq;               /* frequency of polling (in seconds)  */
    int  sock;               /* socket number during a poll        */
    long last_poll;          /* time when last poll data was read  */
} HostConfig, *HostConfig_ptr;
```

Figure 3-1: The HostConfig data object stores polling specific data of each host.

for maintaining the results of the pollings (these are stored in a separate object to be described later).

As shown in Figure 3-1, the first four members of the HostConfig data structure are read from the configuration file; except the hostname, the remaining three parameters have defaults so they are optional. The hostname given in the configuration file may be an alias of a host; the finger server obtains the official name from system databases using *gethostbyname* call before it adds it to the *real_name* field in the HostConfig structure. The *options* field provides special directives for the finger daemon. Since finger data that is sent back is dependent on the option, this field is also used as a token that differentiates the various parsing mechanisms. The *type* field maintains the type of the host; this information, too, is crucial to be able to parse finger information correctly. The *freq* field is for polling frequency, in seconds. The default frequency is 90 seconds and the minimum frequency is 60 seconds; some hosts may be polled more or less often than the default if the frequency parameter is given in the configuration file. Some remote, slow or heavily loaded machines ought to be polled infrequently.

The remaining two members of the HostConfig object are dynamic polling data. Whenever a socket is opened successfully, the socket identifier is saved in the *sock* field. When finger information is read, the sock field is reset to -1. Hence, this field can be examined to determine the state of a polling session. The *last_poll* field holds

the system time when the last poll was read successfully; combination of this field with the frequency field allows the polling module to determine if the host is ready to be polled again.

3.2.3 Polling

Every machine is polled at a certain interval to collect idle time state. The default interval is 90 seconds, whereas the minimum is 60 seconds. Although it is possible to get finger information in fine granularity of minutes and seconds, it is not possible to get other useful information in the same poll (e.g. where the user is logged in); this is a trade-off. In addition, polling frequently enough to recover the granularity in seconds is neither practical nor useful. Some machines may be polled less frequently than the default value, however. For instance, machines with many users or machines at a remote site may trade up-to-the-minute updating for better system performance.

There are two important considerations when polling in an asynchronous environment. First of all, each poll must be contingent upon the success of the previous poll. It is unnecessary to query for finger information from a particular host a second time if the preceding poll has not returned. If the state of the polling session is not maintained, machines that are slow in responding will have a flood of requests from the finger server over a period of time. Since each, request corresponds to a socket, there may be many sockets open to the same machine which is not only wasteful of system resources but can reduce overall system performance.

Secondly, coordination of pollings among the machines is important. Even though the replies arrive asynchronously, for similar machines within the same network, those replies can arrive almost simultaneously. Therefore, if all the requests are sent out at the same time, the finger server will be bogged down when all the replies arrive. During this time, client requests and other internal functions (such as updating the data structures) may be backlogged. Hence, the finger server staggers the requests uniformly over the entire polling period. For instance, to poll 25 hosts over 90 seconds,

triggering a request every three seconds is appropriate. Any incoming client request can be handled within the three second window; the maximum delay will be the time it takes to update the data from a single machine. Figure 3-2 shows the entire polling mechanism. The stagger time between polls is achieved by using a timer.

For various reasons it may not be possible to reach the finger daemon of a machine (e.g. the machine is down, the network connection is faulty or the finger daemon has been disabled). In such situations, a simple backoff mechanism is applied to subsequent polls of that machine. After two consecutive failed connections to the daemon, the polling interval of the machine is multiplied by a programmer-defined value (6 in this implementation). If connection still fails at this new interval for 6 times, the interval is yet again multiplied by 6. The rationale is to try connecting every ten minutes or so in the first hour, then every hour or so in the first day, then stabilize at once per day. Note that should the connection succeed, or should the user "add" the machine after it is healthy again, the initial interval is reinstated.

3.2.4 Callbacks

At the time of polling, a socket to a machine is opened and a read and a destroy callback each are registered with the Socket Manager. These callbacks, which are the same for all the pollings, call the routine ReadCallback when the socket has something to be read and DieCallback when the socket is closed. The polling then moves on to the next machine; the server does not wait and block to read from a socket. Hence, the receipt of data is fully asynchronous. When data is available at a socket, its descriptor disambiguates among the many machines that may have outstanding polls (note that HostConfig structure of each machine saves the socket descriptor upon each poll). ReadCallback then reads the data and the finger daemon closes the socket. Although it is a streams connection, the finger daemon keeps the socket open only as long as is necessary; that is, it is not possible to reserve a socket for future pollings. The destroy callback is useful when some number of tasks need

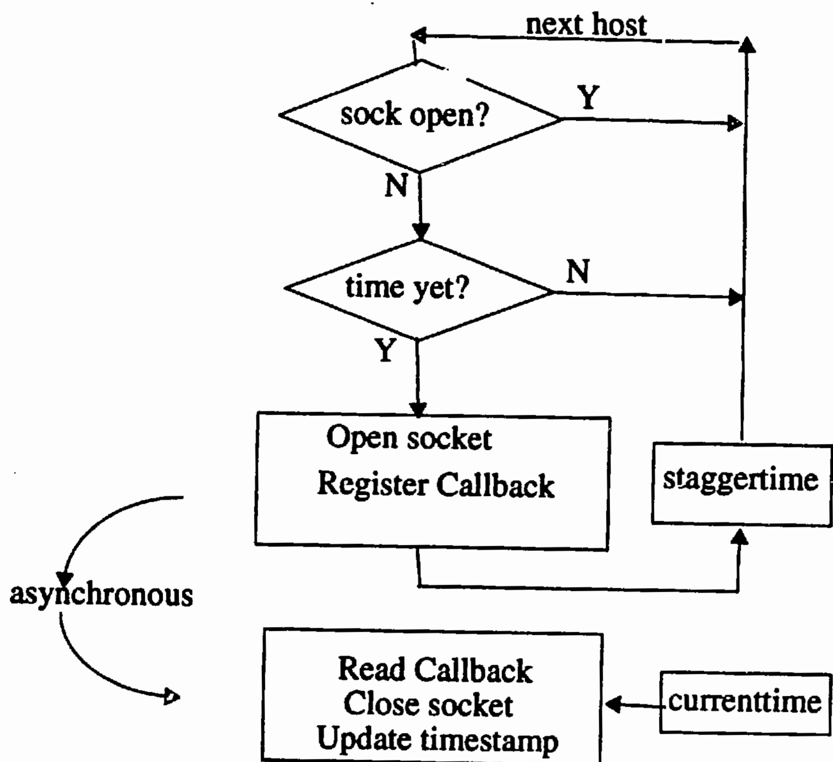


Figure 3-2: Flowchart of the polling module.

to be done whenever a socket is closed.

The asynchronous nature of the callbacks allow the finger server to start a query on another finger daemon or even service a client request while waiting for a reply from a machine. The client requests are serviced using callbacks (FsRequestHandler and FsKillHandler) as well.

3.3 Parsing Finger Information

Information returned by the finger daemon is machine-specific. So far, machines that use Unix, Ultrix or Genera (an operating system for Symbolics Lisp machines) are supported. Figure 3-3.A shows finger output from a Lisp machine running Genera; Figure 3-3.B shows output from a machine running Unix and Figure 3-3.C shows output from a machine running Ultrix. Typically the information received will have at least the username, hostname and the idle time; most Unix systems also report the terminal. While idle times indicate the user's activity or lack thereof, the terminal name is useful in locating the user's physical location. If the user is connected to the console ("co") terminal of a machine and has a small or zero idle time, it is evident that no matter where else the user is logged on, she is physically located where the console of that machine is. On the other hand, if the user is on "dX" (X is an alphanumeric) terminal, it is clear that the user has dialed in.

Sometimes multiple rounds of polling is necessary when not all the information is provided by a single poll. If a machine has many users logged in and the finger server is interested in only a small subset of such users, it uses a fast poll that gives a brief list with usernames; then it conducts a poll on only those users who are logged in and are active. It is wasteful to make individual polls on users who may or may not be logged in, especially on slow, heavily loaded or remote machines. Hence, a fast poll (e.g. using the -i option) is used (see Figure 3-3.D). From that information, individualized polls (see Figure 3-3.E) can recover other information such as the host

```
A. Finger information from a machine running Genera
[obvious]
cahn     Janet Cahn   OBVIOUS 39:30 Obvious: Sound Room x0316...

B. Finger information from two machines running SunOS
[chips]
Login       Name                TTY Idle    When     Where
barons   Barry Arons             p0  19: Tue 16:46  leggett
geek     Chris Schmandt          p1   1d Thu 15:32  moosilauke

[hydrox.media.mit.edu]
No one logged on

C. Finger information from a machine running Ultrix
[cecelia.media.mit.edu]
Login       Name             TTY Idle    When    Office
bv       Barry Vercoe      p0   27 Fri 18:39
tod      Tod Machover      p1    1 Sat 13:04  E15-411 MIT 3-0394
sanjay   Sanjay Manandhar  p2      Sat 13:07  E15-355     253-0312

D. Finger information with ''-i'' option a machine Ultrix
Login      TTY            When            Idle
geek       ttyp8    Fri Apr 26 18:24   1 day 15 hours
sanjay     ttyq6    Sun May  5 10:31

E. Finger information with the username option an Ultrix machine
Login name: sanjay                      In real life: Sanjay Manandhar
Office: E15-355, 253-8076               Home phone: 617-661-0432
Directory: /u2/sanjay                   Shell: /bin/tcsh
On since May  5 10:31:27 on ttyq6 from toll.media.mit.e
Plan:
```

Figure 3-3: Examples of typical finger information.

the person is logged in from.

3.4 Maintaining Finger Information

The result from the polls update the user and host data structures. After each host reports, its corresponding host data object along with its internal linked list of users is updated and any state transitions are immediately reported to the clients that requested asynchronous notification. In figure 3-4, the *name* member corresponds to the official name of the host and the *timestamp* is the time the host was last updated. All users logged in to the host are maintained in an internal linked list, the head and tail pointers of which are also members of the HostInfo object.

Linked list of users within the hosts consists of UserInfo data object (see Figure 3-5). The *name* corresponds to the username of the user, the *terminal* to the terminal name on which the user is logged in and *idle* to the idle time of the user at the time of the polling. The *idle_old* and *idle* members are sufficient to describe the four state transitions (login, logout, active and dormant) of users within a host (described in the next section). These state transitions are recorded in the *alert* field. The *hostname* and *host_old* fields of the UserInfo objects are superfluous within the HostInfo structure. Users that login to the host are added to the list of users within the host, while users that log out are removed. Hence, the HostInfo data object describes the state of any particular host completely. Any query of host information by any client is looked up in this data structure.

The most recent activity of users is maintained in a separate linked list of UserInfo objects. This list provides overall state of any particular user. Hence, a user is not removed from the list when he logs out. When the UserInfo data object is used in this list, the *hostname* and *host_old* fields are relevant. In fact, the combination of these two fields along with *idle* and *idle_old* can characterize another very important asynchronous event, called *moved*. All the other fields are relevant and used with

```
typedef struct _FS_HostInfo {
    char name[MAXCHARS];        /* hostname */
    long timestamp;             /* time of last update */
    Link *userHead;             /* list of users logged on the host */
    Link *userTail;
} FS_HostInfo, *FS_HostInfo_ptr;
```

Figure 3-4: Structure that saves the state of each host.

```
typedef struct _FS_UserInfo {
    char name     [MINCHARS];   /* username                      */
    char hostname[MAXCHARS];    /* host on which logged in       */
    char host_old[MAXCHARS];    /* last host on which logged in */
    char terminal[MINCHARS];    /* current terminal              */
    long timestamp;             /* unix_time                     */
    int  idle;                  /* current idle time             */
    int  idle_old;              /* idle time on previous pass    */
    int  alert;                 /* async notification:alerts     */
} FS_UserInfo, *FS_UserInfo_ptr;
```

Figure 3-5: Structure that saves the state on users.

similar semantics as within a HostInfo object.

3.5 Updating Finger Information

Whenever a host successfully returns from a poll, the corresponding host object will be updated. At the very least, the timestamp of the host will be updated. If there are users logged in, their idle times and timestamps are also updated. Immediately after such updating a consistency check is done, as a result of which the data structure may be modified and asynchronous alerts may be issued.

Reporting changes to the user data structure is a little more complex, however. There is no one-to-one correspondence between polling of a host and the user object. The user data structure reflects the most recent activity across the entire gamut of

1.	IF	timestamp	> user_timestamp
	AND	terminal	= "co"
	AND	idle	≤ user_idle
2.	IF	timestamp	> user_timestamp
	AND	idle	< user_idle
3.	IF	timestamp	> user_timestamp + POLLFREQ

Figure 3-6: Precedence rule for updating UserInfo object.

hosts. Since not all hosts respond at the same time, and some hosts may respond after other hosts may have been polled more than once, a precedence rule is used to ensure legitimate but slightly old user data is not overwritten. The precedence rule, going from high to low precedence is given in Figure 3-6.

In the rule above, POLLFREQ is normally the default polling frequency. However, if a user is active on a machine which has frequency larger than the default frequency, that larger frequency will have to be chosen. Note that if a user is remotely logged in from machineA to machineB to machineC and is active on machineC, all three machines will register zero idle time but only the first will have zero idle time on the console; hence, only the information from machineA is saved in the user data structure. This is also the machine at which the user is physically located. This paradigm is consistent with the requirements of the activity server, since it is interested in locating the users (and determining their activities) through their machine activity.

3.6 Alerts

Alerts signify transition of user activities from one state to another. Four kinds of asynchronous alerts are provided. These are *login, logout, active* and *dormant.* The first three of these can be inferred directly from finger information. The fourth, dormant, is an artificial transition from active to dormant state of activity. IDLE_THRESHOLD (5 minutes in this implementation) is the ceiling of idle time above which a

State	Transition conditions		
LOGOUT:	if	user_timestamp	≥ host_timestamp
LOGIN:	if	user_idle_old	= -1
	and	user_idle	≥ -1
DORMANT:	if	user_idle_old	≥ IDLE_THRESHOLD
	and	user_idle	≥ IDLE_THRESHOLD
ACTIVE:	if	user_idle_old	≥ IDLE_THRESHOLD
	and	user_idle	≥ IDLE_THRESHOLD
DIALIN:	if	tty	= dX
	and	tty_old	!= dX
	and	idle	≤ IDLE_THRESHOLD
MOVED:	if	hostname	!= host_old
	and	idle	≤ IDLE_THRESHOLD

Figure 3-7: Algorithms for alerts.

user is considered not active at the machine. By monitoring these alerts, a client application can construct a dynamic model of user activity. In addition to these four, the user status uses two more, *dialin* and *moved*. Dialin is a special case of login or active alert signifying fresh activity at a dialin port of a machine. On the other hand, *moved* is a special case of active in which the user moved (physically) from one machine to another. The algorithms that mark state transitions are given in Figure 3-7.

3.7 Protocols

There are two levels of finger server protocol. The lower level protocol is at the byte stream level; the second, a C interface, based on the byte stream level, is also available. Once a socket connection to the server is established, simple, case-insensitive, ASCII commands may be sent via the connection. The server uses the same connection to send its replies. This human-readable interface allows for understanding of commands and responses with a minimum experience with the system. A brief mode can be set so that non-human clients may get the important data easily. Appendix A describes

```
Connected to toll. Escape character is '^]'.

locate barons
[PERSON barons][HOST leggett, TTY co][IDLE 3 min] 03:21:02 PM

set-mode short
OKAY

get-host-info leggett
barons leggett co 6 03:23:21 PM

bye
Goodbye.
Connection closed by foreign host.
```

Figure 3-8: A sample interaction with the finger server.

the byte-stream and C interface in greater detail.

Figure 3-8 shows a sample session with the finger server using telnet. (User commands are shown in italics).

3.8 Client Interaction

Clients can determine the form and format of outputs. The cumulative state of all the client requirements are saved in a SockInfo structure shown in Figure 3-9. The user may set these output modes and requirements using the server-client procotol (Appendix A). The SockInfo structure is created for each client when it connects to the server and is destroyed when it disconnects. The first member of the SockInfo is the *name* (converted from an integer value to a character string so that searching will be facile and consistent with that of other data objects) of the socket on which the client is communicating. The *out_mode* holds the output mode that the client requested (default is long mode); see Figure 3-8 for examples of short and long mode of output. The *sync_mode* field, if asserted will withhold

```
typedef struct _FS_SockInfo {
    char name    [MINCHARS];  /* socket number              */
    int  out_mode;            /* output mode (short|long)   */
    int  sync_mode;           /* flag for sync output mode */
    int  all_host_async;      /* xmit all hosts async data */
    int  all_user_async;      /* xmit all user async data  */
    Link *u_asyncHead;        /* list of users to track    */
    Link *u_asyncTail;
    Link *h_asyncHead;        /* list of hosts to track    */
    Link *h_asyncTail;
} FS_SockInfo, *FS_SockInfo_ptr;
```

Figure 3-9: Structure that saves the wishes of each client.

all asynchronous reporting requests (default is asynchronous). The *all_host_async*, if asserted, is an indication to the server to send all asynchronous alerts encountered by the hosts; likewise, *all_user_async* indicates that all asynchronous alerts encountered by the users should be reported. Should the client be interested in only a small subset of users or hosts, these are saved as linked list objects within the SockInfo object.

3.9 Shortcomings

There are some shortcomings which are inherent in the finger daemon while others were introduced by the finger server itself.

- It is possible not to "see" quick logins and logouts on a machine. Since the polling cycle executes every few minutes, a login-logout pair on a machine by a particular user may not be reported by the finger daemon. Likewise, if a user exceeds the IDLE_THRESHOLD for a few seconds but then hits a key, this DORMANT state of up to 59 seconds is not reported. The granularity of the finger daemon is in minutes up to 59 minutes, then in hours and minutes up to 9 hours and 59 minutes, then in hours up to 23 hours and then in days. On the other hand more precise granularity may not be truly useful.

- Activity in some programs is not noticeable to the finger daemon. For instance, activity solely in the gnuemacs editor program will increment the idle time as if the user was away from the terminal.

- The finger daemon reports only the idle time, it does not report what program may be running. There are other Unix programs and daemons that monitor execution of programs at any particular time but their services were not used for a number of reasons. Not all machines run these daemons (rwho, w, rusers, etc.) but almost all machines keep the finger daemons running. (Some sites disable even the finger daemon for security reasons). Hence, finger server can remain very general and modest in its requirements. Many of the other daemons run by mutual broadcasts and receives. This can put a severe burden on network and machine performance. Idle time history alone can be fairly useful.

- One of the shortcomings of the finger server is that there is a latency in asynchronous reporting. Should there be a race condition between two hosts reporting back to the finger server, the updates of the second host on the queue will be delayed by the time it takes the first host to dump all its data and for the finger server to update all its data structures. Typically the latency is in the order of a few seconds.

3.10 Optimizations

Some optimizations were made so that the finger server could service all its clients with minimum latency. Querying the finger daemon to find a limited number of users' idle times is a very costly operation. It hurts network and machine performance. The finger daemon checks some system files such as /etc/hosts file and figures out real names, telephone numbers, office number, etc. Most of this information is extraneous for the purposes of the finger server. Hence, sending appropriate options to the finger daemon will return only the desired information. Non-default options have been used

in two instances for a slow and heavily loaded machine. Unfortunately, these options are not available on all machine types.

Less frequent polling may be used to reduce performance losses. Hence, a separate polling frequency parameter can be set for each machine in the configuration file itself. Infrequent polling can alleviate system latency and also reduce the burden on the responding machines. A design feature, for future implementation could include dynamic adjustment of polling frequency. The system could record the response time and poll machines with poor response times less frequently. Similarly, it is wasteful to poll machines with no one logged on; these too, can be polled infrequently with a proviso that the original frequency would be reinstated once someone logs in.

Abnormal conditions on machines can cause delay in their responses. Hence, these conditions are noted and a simple backoff mechanism has been used. In future extensions a more complex backoff mechanism could be used.

Chapter 4

The Activity Server

The activity server, the main object of this thesis, provides a mechanism to glean high-level abstraction of the activities of the user community. The activity server gathers data from the Listeners and looks for corroborating, conflicting or complete lack of data. With the help of IF/THEN rules, the activity server attempts its best estimate of user activities; the next chapter is devoted to the rules that the activity server employs. The activity server also saves some history on data from all Listeners so that the rules can benefit from past data. The Listeners provide asynchronous events; the activity server performs further filtering on these events and cross-comparison among Listener events. Querying clients will receive replies generated from this final state of activities. This chapter deals with the design and implementation issues of the activity server while the subsequent chapters will deal with the rules and clients.

4.1 Architecture

The activity server embodies an architecture similar to the one described for the finger server for inter-process communication. There are two levels of client-server socket communications - between the activity server and the Listeners and between

the activity server and its own clients. Much of the flow of control within the activity server is triggered by incoming asynchronous events. The events initially pass through the event handler which does database lookups and determines if the event is valid. A valid event proceeds to the module that updates the current state of the data structures; invalid events are discarded. The type of the event will trigger a number of rules in the rule-set; the output of the rule-set is saved in the final state of the system, which is made available to the clients.

4.2 Mechanism

How do the events from the Listeners affect the state of the users? The entire mechanism from system startup to the update of internal data structures will be provided in this section. To this end, many implementation decisions and their rationale will be discussed.

4.2.1 Startup

A number of initializations are done when the activity server starts up. First, the activity server must open a socket in the advertised port (4502) to add the service; clients may then connect to the server via this port. For diagnostic purposes, a log file saves transcripts (along with timestamps) of interactions with the Listeners as well as with the clients.

Secondly, routine information is cached in memory. For instance, the help file, *.mati_help*, that will be sent to clients upon request, is read from disk and saved. Similarly, configuration files that specify the users and places to monitor, *.mati_config.users* and *.mati_config.places*, respectively, are read and the contents are used to create data structures (the format and examples of these files are given in Appendix C). Frequently needed yet static information such as office phone numbers and room

numbers, is also looked up in the database[1] and added to the data structures. An interrupt handler is registered so that when an interrupt signal is received, the server can close sockets, update the log file and make a graceful exit.

Next, connection to each Listener is attempted. If the connection is successful, initial state information is solicited. The finger server returns the least idle time and hosts of all users; the location server returns the location of all users and the phone server returns the current state of office phones of all users. The initial requests to all three Listeners are all synchronous. It is assumed that the domain of users that the activity server is interested in is a subset of the domain of users of each of the Listeners. Should this not be true, it will be evident at startup time when the activity server is collecting initial state information. The activity server will try to issue an "add-person" command to the Listener if the user is not one the users monitored by that Listener but if this operation fails, that user will be struck from the list of users being monitored by the activity server.

Once the initial state information is received, each Listener is requested to report asynchronous events. The activity server selects requests to the Listeners that best fit its need. This is the first cut to managing and filtering what could potentially be a deluge of events. For instance, the phone server allows its clients to express interest in a subset of all its telephony events. The activity server expresses interest in only the DIALING, ACTIVE and IDLE events. The rationale for choosing just these three and filtering out the rest is discussed in a later section. Likewise, the activity server requests only changes in physical location – not all user-host activities – from the fingerserver. From the location server, it solicits tracking information of only the users it is interested in, not all users that may own a badge.

Should the attempt to connect to the Listeners fail, a timer[2] is set to attempt reconnect later. Since the Socket Manager allows only one timer, if more than one

[1] A separate database module can be invoked to create an in-memory representation of the ASCII database.

[2] This value is 3 minutes in the current implementation.

Listener needs to reconnect, they use the same timer callback.

4.2.2 Event handling

All events, synchronous and asynchronous, pass through the event handler. The event handler is able to distinguish which Listener is reporting by inspecting the socket the information is received on; it unbundles the data accordingly and updates relevant data structures.

Considerable filtering of events is done by the event handler. Since the events trigger almost all the modules of the activity server, it is advantageous to identify and discard spurious or unwanted events early on. First of all, it discards events for users the activity server is not interested in and if spurious or duplicate data packets arrive, these are also duly ignored.

Next, it ensures that each packet has a consistent username. The phone server sends calling/called telephone numbers and the location server sends real names; these are converted to usernames by doing database lookups.

Events that are not arrested by the filtering mechanism are considered relevant and are timestamped even though all the Listeners return their data with a timestamp. This is necessary because system times vary; a single point of timestamping on a single machine will provide a standard reference point to compare temporal information of events.

4.3 Data Representation

The activity server needs to keep the state of a number of entities including servers (Listeners), clients, places and people. These entities are modeled as data objects which are described below. The following section will describe how the incoming events modify these data objects.

```
typedef struct _ServerInfo {
    int type;                  /* type of server */
    int upstate;               /* health of the server */
    long timestamp;            /* time of last event or downtime */
} ServerInfo, *ServerInfo_ptr;
```

Figure 4-1: The ServerInfo object saves the state of Listeners.

4.3.1 Server object

The ServerInfo object maintains the name, the server state (up or down) and the last time an event arrived or the time it went down (see Figure 4-1). If a server goes down, the the assertions made by the rules are affected since most of the rules are context-dependent; when a server is down for an appreciable length of time (more than 15 minutes in this implementation) a different set of rules are triggered to compensate for the loss.

4.3.2 Person object

The PersonInfo object includes a person's username, which is used for addressing, and office location, office phone, and the type and time of the last event. In addition, events from each Listener are stored in its corresponding data structure within the person object. Each of the object representing the Listeners maintain data that is relevant to their corresponding Listeners. As shown in Figure 4-2, the place object is a pointer rather than a real data object. This is because a phone number and a username are unique to a person; hence, use of the phone or the username (on machines) will not affect the activity of any other user directly. However, many people can share the same physical space (office, conference room, etc.). One person's entry or exit may affect the activities of others. If all occupants point to the same place object, dynamic information such as people's entry/exit need be updated in only one place, the place object; all changes will then be visible to all occupants.

```
typedef struct _Person {
    char name        [MAXCHARS]; /* username of the person */
    AS_PlaceInfo *place;         /* current locations */
    AS_PhoneInfo phone;          /* current and past phone states */
    AS_MachineInfo host;         /* current and past finger alerts */
    char home_phone [MINCHARS]; /* office phone number (static data)*/
    char home_office[MINCHARS]; /* office number(static data) */
    int place_unknown;           /* flag if badge is not visible */
    int last_event;              /* source of last event */
    long timestamp;              /* time of last event */
} AS_PersonInfo, *AS_PersonInfo_ptr;
```

Figure 4-2: The PersonInfo object saves the state of all persons.

```
typedef struct _PlaceInfo {
    char name[MINCHARS];        /* name/room number of the place */
    int  nfolks;                /* number of people in the place */
    char folks[MAXCHARS];       /* name of all current occupants */
    Link *current_uHead, *current_uTail; /* current occupants */
} AS_PlaceInfo, *AS_PlaceInfo_ptr;
```

Figure 4-3: The PlaceInfo object saves the state of all physical spaces.

4.3.3 Place object

A separate linked list of all PlaceInfo objects is maintained. The place object consists of the name, for addressing and informational purposes, and the lists of all occupants (and their entry times) currently in the room. Hence, it is possible to retrieve not only the number of occupants in a room, but also their names and the duration they have been there. Figure 4-3 shows the PlaceInfo data structures.

4.3.4 Machine object

The MachineInfo object consists of all the information sent by the finger server.

```
typedef struct _MachineInfo {
    char name[2*MINCHARS];      /* official name of the machine */
    char loc[MINCHARS];         /* physical location of the machine */
    char tty[MINCHARS];         /* terminal */
    char alert[MINCHARS];       /* async event */
    int idle;                   /* idle time (in minutes) */
    long timestamp;               /* system time event was registered */
    Link *hostHead, *hostTail; /* history of past finger alerts */
} AS_MachineInfo, *AS_MachineInfo_ptr;
```

Figure 4-4: The MachineInfo object saves the user state on machines.

```
typedef struct _PhoneInfo {
    char name[MINCHARS];        /* phone number as character string */
    char state[MINCHARS];       /* state of the phone */
    long in_time;               /* time event arrived */
    long out_time;              /* time symmetrical event arrived */
    Link *phoneHead, *phoneTail;/* linked list of past phone events */
} AS_PhoneInfo, *AS_PhoneInfo_ptr;
```

Figure 4-5: The PhoneInfo object saves the state of user phones.

In addition, the physical location of the machine is looked up in the database and stored in the *loc* field, see Figure 4-4.

4.3.5 Phone object

The PhoneInfo object consists of the phone number, saved in *name*, the *state* of the phone, the time the phone was active, *in_time*, and the time it became idle, *out_time*.

4.4 Updating Data Structures

Since events from different Listeners affect and modify different parts of the data structure, each type of event will be considered separately.

```
barons  leggett  co 0 Moved  05:25:27 PM
barons  leggett  co 0 Active  06:38:21 PM
hindus  hood  co 6 Dormant  06:39:58 PM
warlord toxicwaste  p0 0 Login  07:16:16 PM
root  toxicwaste  p1 0 Logout  11:20:23 PM
```

Figure 4-6: Format of short form of output from the finger server.

4.4.1 Finger events

The finger server was designed with the requirements of the activity server in mind. Hence, the six alerts that the activity server receives, together with the username and the hostname, can be used directly to make useful inferences. Details of the inferences made by the finger server were given in the previous chapter. A sample of the short form of output for "track-location" command, similar to the request of the activity server is provided in Figure 4-6.

4.4.2 Phone events

Although the phone server can provide a wide array of events for each of the phone line it is monitoring, only three suffice for the purposes of collecting phone activities. These three events are DIALING, IDLE and ACTIVE[3]. Other events such as incoming call, held, pending, local hold, rejected, etc. are not only extraneous to the activity server, but also add to the complexity unnecessarily.

The three-state transition diagram shown in Figure 4-7 is simple yet sufficient to characterize all the scenarios in which a user's phone may indicate presence or absence of activity. The five scenarios that are possible are given as follows:

1. IDLE - DIALING - IDLE (no answer - outgoing call)

[3]The terminology here is consistent with that used by the phone server itself. DIALING is the state of taking the phone off-hook and actually keying in the numbers. If a call connects after dialing or after taking the phone off-hook (for incoming calls), it is in an ACTIVE state. If the phone reverts to the dormant state of polling for incoming calls or dialing instructions, it is in an IDLE state.

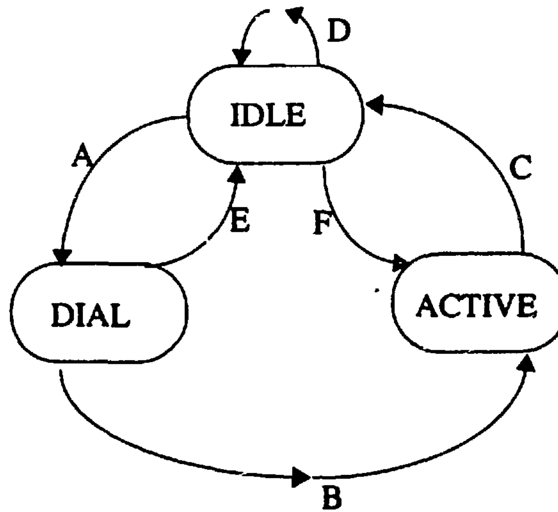


Figure 4-7: Five scenarios of the phone server that are of interest to the activity server.

2. IDLE - DIALING - ACTIVE - IDLE	(call complete - outgoing call)
3. IDLE - ACTIVE - IDLE (less than 2s)	(transferred, e.g. voice mail)
4. IDLE - ACTIVE - IDLE (more than 2s)	(call complete - incoming call)
5. IDLE - IDLE	(no answer - incoming call)

In scenarios 3, phone becomes active for a fraction of a second, when it is being transferred. Assuming that no conversation is less than 2 seconds long (it is a safe assumption), this scenario can be discarded. Hence, only scenario 4 is a genuine call. From scenarios 1), 2) and 4) one can safely infer that a user is in his office (assuming that user's personal phones are not answered by others, out of common courtesy or otherwise), and the remaining scenarios can be used to infer that the user is not in his office.

From the argument above, it can be concluded that only a few of the events need be saved to be able to recover useful activity in the past. Although the current state of the phone is saved until the next phone event arrives, only the ACTIVE - IDLE pair

of events in scenarios 2) and 4) will be saved along with their *in_time* and *out_time* signatures. Hence, inspecting past histories is especially apt for the phone events since only a combination of events comprise a sensible duration of activity.

4.4.3 Location events

The location events update the data structures pertaining to the user and place object. When a user wearing a badge wanders beyond the periphery of the sensor network or when the user takes off the badge, the place becomes "Unknown." Instead of discarding this event, it is saved until the next event arrives. Then the activity server may be able to guess the location of the "Unknown" place.

Every place object maintains the current list of all occupants with their corresponding time of entry. Maintenance of current occupants is important because each occupant may affect the activity of the others. If one of the two occupants of a room leaves, he not only affects his own state, but also the state of the person who remains in the room. The obverse effect is true when a person joins another person who is alone in an office. The events triggered by comings and goings when there are more than two people are less important although the number of occupants at any point in time or the entry time of any resident must be known should a client inquire.

4.5 History Mechanism

The activity server retains history on crucial information that suggest state changes. For instance, times when a person enters a room or makes a call are noted so that inquiring clients can be notified of the duration of the time in the office, conference room, another office, or time on the phone. This information helps people make deductions about others' activities. For instance, a person who has been in the conference room for one minute will most likely be there for some time; however, if the person has been there for over an hour, the likelihood of his leaving the room is higher.

The same argument can be made of phone conversations or time spent in the office. In addition, timestamps of when a person comes back to the work area, the time he leaves, the number of times he remotely logs in or dials in are relevant to his activity. If the activity server reports that a person has not been to his office in several days, the fact that he is dialed in or has recently logged in remotely is very significant - inquiring members of his group can be assured that he can be reached by electronic means.

Timestamps of events from each Listener are also saved. These timestamps are used by the rules to deduce possible change in state. For instance, if a badge has been seen in the same place for more than two hours, it is indicative of a badge that was left inadvertently within the line of sight of a sensor (the usual practice is to leave the badge turned over so that it does not transmit signals, which not only trigger false events, but also reduce the life of batteries).

4.6 Client Interaction

Fundamentally there are two kinds of outputs from the activity server. Clients can get the current state of any user's activity or get the history of user activities over a period of time up to the present. The former is the *locate* command while the latter is the *activity* command (for more detailed information on the server-client protocols consult Appendix B). The *locate* command can be synchronous or asynchronous; in the asynchronous mode it is called *track*. The *locate* command can be in long mode or short mode (see Figure 4-8). The long mode allows for human-readability while the short mode is more suited for machine-readability should a client program attempt to parse the output from the activity server. While *locate* takes a username as an argument, it is possible to get all users' current activity by using *locate-all* (*track-all* in the asynchronous mode). As shown in Figure 4-8 the locate command,

```
Connected to toll.

locate sanjay
[PERSON sanjay][PLACE 355][STATE active on machine]
[PHONE 253-8076][HOST toll][TIME 4:59:04 AM]

set-mode short
OKAY

locate sanjay
sanjay 355 10 253-8076 toll 05:03:53 AM

activity hindus
hindus  was on machine hood, in own office, 352, about 2 hours ago.

bye
Goodbye.
Connection closed by foreign host.
```

Figure 4-8: A sample interaction with the activity server.

provides not only the user's current state of activity, but also the place, nearest phone, machine-name (if it is relevant) corresponding to the state.

The *activity* command provides English text as output to a user's activity. An important feature of this command is that it not only reports the current state of the activity, but also mentions past history of user activity. The text generator used for this command decides which of the history data is relevant, phrases it in English and sends it to the client. There is no short or asynchronous mode for this command. Additionally, the *activity* takes a username as a command; there is no facility to get everybody's information simultaneously.

Chapter 5

Rules

The rules that are triggered by incoming events are central to the activity server. These rules are necessary to draw higher level conclusions and to resolve conflicting data from the various Listeners or accentuate data that is in agreement. Each event triggers some number of rules. For some events, there are rules that check the final state and decide if the incoming event will add any new information to what the activity server already knows. Other events require cross-checking with other Listeners. In essence, the rules do the arbitrations and tradeoffs, apply consistency checks and make the final decisions for the activity server.

All the rules are based on heuristics applied to an office situation. Some of the general heuristics can be summarized as follows:

- The common activities of two or more people bears more weight than other individual activities of any one of the people. For instance, if two people are meeting in an office, the event that reports that either person's workstation is dormant is not relevant.

- A person's office is the reference point for many of his activities. Hence, events that indicate activities (or lack thereof) in individual offices are very important. For this reason, a person's office number, phone number and machines are closely tied to the PersonInfo data structures. History of key events pertaining to

activities in the office is also important.

- The final state of any activity must follow consistently from current train of events. If there is a conflict, it is resolved using some of the strategies described in the last section.

These rules are implemented as conditional statements in C, but are described here in abstract terms. They are forward-chaining [17], or they work from a current situation to a final state. The rest of the section considers many important scenarios of activities. Within each scenario, the conditions and actions (or antecedents and consequents) will be delineated and the reasons for picking those particular antecedent/consequent pairs will be considered.

Some of the variables used in the rules below are taken from actual rules. The variable *last_in_office* is the time a person was last reported as being in his own office by one of the Listeners, *in_since* is the time the person came into the office. The counters, *rlogins* and *dialins*, denote the number of times a person remotely logged in or dialed in while away from the office. *Currenttime* is the current system time. The final states of the activities are treated in depth in the latter part of the chapter.

5.1 Scenario 1: Coming into the Office

Arriving at the office after some time is important for the activity server because it provides a starting point from which it can start keeping history. For instance, at the start of a day or, more generally, after a period that is can be construed to be more than a break, some initializations are done. By choosing a time duration of 4 hours we discount absence from the office for a casual roaming through the halls, for visiting another office or even for lunch breaks, etc.

IF	last_in_office > 4 hours	THEN	save currenttime as in_since
			reset rlogins and dialins

phone is office_phone
place is office_number
state is alone_in_office

Note that events from any of the Listeners that show that the person has come into the office, i.e. activity at the console of the machine in the office, activity on the office phone or sighting of the badge by a sensor in the office, can trigger and fire this rule. The most significant assertion from the rule above is that the current time is the time when the person came into the office. Why is this fact important? At a later point in time, an inquiring client can find the duration the person has been around; this fact can help decide how long the person may be around or even what the person may be doing. Note that the activity server does not take days of the week or times into account. If it did it could attempt to predict in the near future as do many user modeling research efforts; but that is beyond the scope of this work.

5.2 Scenario 2: Leaving the Office

Although lack of activity on the phone is a binary value, it is not so clear-cut for the machine or badge activity. Nevertheless, each one has a time threshold above which the person is inactive within the Listener's own jurisdiction. Therefore, when all the Listeners show dormancy of some sort, this rule is fired. All inquiring clients will be notified that the person is not around; other qualifying statements such as, when the person was last in the office, for how long, the number of times he remotely logged in/dialed in since he left the office will be transmitted.

IF dormancy on all Listeners THEN last_in_office is currenttime
in_duration is currenttime - in_since
phone is UNKNOWN
place is UNKNOWN
state is NOT_AROUND

5.3 Scenario 3: Visiting Another Office

Whenever a person's badge is sighted in another office or the person logs into the console of a machine in that office, this rule fires. Two separate conclusions about the state can be drawn by noting how many other people are around. Other people are significant in this scenario because someone who is alone in another office may be working at a machine there temporarily, or just looking for the original occupant of that office. However, if more than one person is in the room, we can assume that all the participants are busy in some kind of a meeting (formal or informal). A special case of the many-people scenario is if the place happens to be a designated conference room. Then, the chances of people being in a meeting is very high.

In addition, this rule can spawn a side effect rule if there was only one person in the room before the new person walked in. The side effect is that the new person not only changed her own state, but also changed the state of the original occupant of the office to a busy-in-a-meeting state. Similarly, when the visitor leaves, she changes the state of the original occupant (back to alone-in-office state) as well as her own. These side-effects manifest themselves only when there is one person in the office before or after the visit. If a second visitor walks in, there is no side effect on the other two because they were busy to begin with.

IF	visiting another officeX	THEN	phone is officeX_phone
			place is officeX_number
			duration is duration_in_officeX
IF	alone	THEN	state is alone_in_another_office
ELSE			state is busy_visiting_another_office
			visitors are person1, person2, ...

5.4 Scenario 4: Visitor in Office

The obverse effect of the previous scenario is true for the original occupant of the office. From his perspective, the only significant changes are the state, which is "busy with visitors in office" and the start of this new state. If more visitors join in, nothing changes except the number of people in the room; the state of the room is updated dynamically so should anyone inquire the correct number of visitors and their names, it will be known.

IF	visitor in office	THEN	phone is office_phone
			place is office_number
			visitors are person1, person2, ...
			in_duration is currenttime - in_since
			duration is visitor_duration_in_office
			state is busy_with_visitors_in_office

5.5 Scenario 5: On the Phone

When a person makes a successful connection on the phone, the person is busy on the phone in the office. If the person dials but cannot complete the call, we can be assured that the person is in his office but his state is not busy (ignoring the fact that he is really busy while hitting the keys on the phone and listening for rings, etc.). Note that this rule can be triggered after the "visitors in office" scenario or vice versa. Then, the state cannot be simply, busy on the phone, it will have to be busy on the phone and with visitors in the office. If visitors walk in while the person is on the phone, similar augmented state applies.

IF	active on the phone	THEN	phone is office_phone
			place is office_number

duration is currenttime - phone_active
state is busy_on_the_phone

5.6 Scenario 6: Logged in on Another Machine

The definition of "another machine" here is a machine that is within the premises but not in the person's own office. If the person is logged in from a machine (still remotely logged in from a network standpoint but not from a physical standpoint) within the premises, the place and phone will be known. Given these two pieces of information, person-to-person communication is also possible. In addition, the duration of activity at another machine and the duration of the time away from the office is also noted.

```
IF    rlogged in                THEN   phone is placeX_phone
                                       place is placeX_number
                                       host is hostname
                                       duration is currenttime - rlogin_time
                                       last in office is last_in
                                       duration_office is duration_in_office
                                       state is busy_on_another_machine
```

5.7 Scenario 7: Dialed In

An assumption made here is that dialin access is used only from outside the premises. So by default, the person is not around and the phone and place are unknown. Although the hostname will be known, the person is definitely not at the console. Querying clients and people can find out when the person was last in the office and for how long, which can be an indication to whether the person has left for the day or might possibly come back. Nevertheless, it is evident that the person is reachable via electronic means and in many circumstances, this form of communication is sufficient.

IF	dialed in	THEN	phone is UNKNOWN
			place is UNKNOWN
			host is hostname
			increment dialins
			duration is currenttime - dialin_time
			last in office is last_in
			state is dialed_in

5.8 Scenario 8: Remotely Logged In

This scenario is similar to the previous one but there is more flexibility since he is logged in to a machine at the remote end. As noted earlier the connotation of "remote" is in the physical sense, not the network sense.

IF	rlogged in	THEN	phone is UNKNOWN
			place is UNKNOWN
			host is hostname
			increment rlogins
			duration is currenttime - rlogin_time

last in office is last_in

duration_office is duration_in_office

state is busy_on_remote_machine

5.9 Final States

After the rules that apply to each scenario are applied, the final state of activity is determined. The following is an enumeration of activity states that were considered relevant. For states in which the system is unable to locate the person and get the recent activity, relevant past activity (recovered by the history mechanism) is given to the clients.

1. Busy on the phone.
2. Busy visiting another office.
3. Busy with visitors in own office.
4. Busy in a meeting in the conference room.
5. Busy on the phone and got visitors in the office.
6. Alone in the office.
7. Alone roaming the halls.
8. Alone in another office.
9. Alone and active on the workstation.
10. Not around, remotely logged in from the outside.
11. Not around, dialed in from the outside.
12. Not around, state unknown.

5.10 Conflict Resolution

Very often events from Listeners can determine the scenario of activity unambiguously. For instance, if the phone is active and the person's last state was "alone in office," the current state would be "busy on the phone." However, if the person's last state was "not around, dialed in," there is a conflict. The activity server, then, chooses a state using a conflict resolution strategy. First, recency is checked; if this fails to resolve the conflict, a precedence rule given below (from high to low) is used:

1. Dial in
2. Remote login
3. Badge sighting
4. Phone activity
5. Login at home machine
6. Activities on other machines

In the example, above if the "phone active" event followed soon after the "dialed in" event, "not around, dialed in" state stands because the the "dialed in event" is very recent. However, if the "dialed in" event had arrived several hours earlier, the "busy on the phone" state holds. This particular conflict can be resolved by the recency strategy; however, if the conflict was not resolved, the next step would have been to favor the event that is higher on the precedence list.

The precedence rule above was chosen on an ad hoc basis ordered by the stringency of constraints that would apply. For instance, "dialin access" is used only when a person is outside the premises. While it is normal to stay logged in to one's own workstation, this practice is not true for dialin access. Hence, while activity on a workstation in the office can be due to spurious mouse movements (due to vibrations, etc. or by someone else using the machine temporarily) this is less likely for dialins.

Likewise phone activity may be triggered even if the original occupant is not in the office if others around that office answer the phone (out of common courtesy) or make calls themselves. However, it is very unlikely that someone will wear another person's badge. Hence, the badge sighting event is higher on the precedence list than phone activity.

As mentioned earlier, this precedence list is arbitrary. Many factors could change the ordering. However, this ordering has worked effectively for the site where this system was built (the MIT Media Lab).

Chapter 6

Clients

There are two clients that rely on the activity server. These clients are described below.

6.1 The Directory Client

The directory client is a phone-based client that accesses the information within the activity server. This client provides information about a particular person to any inquiring member of the group. The reply consists of English text (spoken by a synthesizer) which will include the activity of the person, and qualifying remarks about that assertion. The user - client interface is the twelve-button touch tone keys of the telephone.

6.1.1 Implementation

The client runs in a loop monitoring a phone line. When a call comes through, it greets the caller and solicits username and password. It connects to the activity server and provides a menu of services. The menu includes several choices: retrieving 1) the activity of any person, 2) the list of people the activity server knows about, and 3) the current version of the activity server software. When the server returns

with the text, the client pipes the text through the Dectalk text-to-speech synthesizer which, in turn, pipes it through the telephone line to the ultimate user. After the user decides to exit, the client unlinks the connection to the server and waits for the next call.

6.1.2 User Interaction

Once logged in, easy instructions tell the user how the system works and what the keys do. The user is permitted to interrupt the client at any point and return to the top level menu or exit; the top level menu also provides the keying instructions for the client. A command can be chosen by the designated numbers, 1 for "activity", 2 for "list-people" and 3 for current "version" number. Usernames are entered by using the letters on the keypad. The system provides username completions of incomplete names when terminated by the "*" button.

The "activity" command returns the user's current activity; if the user is not around, it will say when the user was last seen in the office and for how long. If an error occurs, the text will start with the word "ERROR" (as does any error). For "list-people" command, the text string containing all current users' usernames will be returned. The "version" command returns the version of the server, e.g. 1.0. The following is an interaction with the client (Responses from the system are in special fonts).

User calls up the Directory client.

[Welcome. Enter your user name from the touch-tone key pad.]

User enters username.

[Please enter your password.]

User enters password.

Client connects to the server.

[Hello, Lisa.]

[This is the phone link to the activity server.]

[For activity, press one, to list all users, press two, for version information press three.]

User presses three.

[Current version is 1.0]

User presses one.

[Pick a user]

User presses S-A-N-J-A-Y

[sanjay is busy on the phone and has been around for 5 hours]

User presses #

[Have a nice day.]

6.2 The Watcher Client

The Watcher Client is a client with a graphical interface. It is an X Window System application that displays bitmapped images identifying users and activity fields such as the nearest phone, host logged on to, etc.; this client is an enhancement of a previous effort [16]. Figure 6-1 shows how a lineup of users can be displayed (the users get to choose their own cartoon characters or a bitmap of their own faces); the lower window shows how the user information may be displayed. This client updates the activities of participating members of the group. The display is updated whenever any user changes state on the phone, in physical location or on the network of workstations. Many tracking and messaging facilities are also available.

6.2.1 Implementation

When Watcher starts up, it makes two socket connections to the activity server and requests full asynchronous tracking capabilities on one of them; the second is for synchronous communication.

Whenever the status of any user changes, this information is passed on to Watcher via the open socket set aside for asynchronous communication; Watcher in turn, will

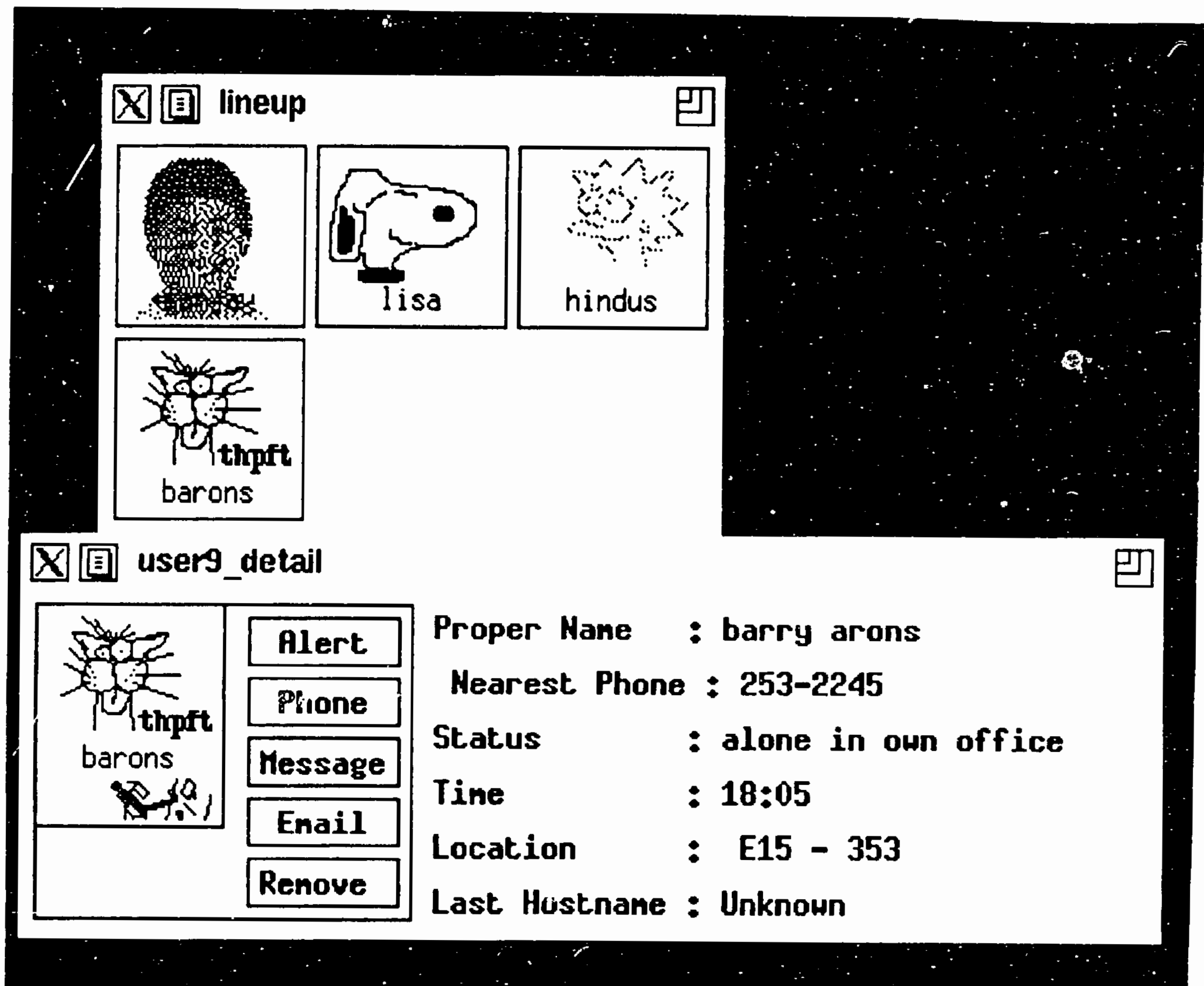


Figure 6-1: An example of Watcher

modify the display to reflect the changes.

A synchronous capability that provides more detailed description of a person's activity is also available. The activity information is the same English text provided when the "activity" command for any user is is requested. Watcher allows this text to be displayed in a pop-up window.

The synchronous information is solicited and provided over the socket meant for synchronous output. Although it is possible to send both synchronous and asynchronous output over the same socket, for simplicity in design, the two socket approach was taken.

Chapter 7

Discussion

The principal contribution of a system like the activity server is the mechanism it provides a user community to coordinate each other's office activities. This can ease scheduling problems, reduce phone tags, and, in general, induce more productive office work. In the short time that the server had been running when this document was prepared, many performance and other issues were evident. The following sections, discusses performance of the system and possible directions for future work.

7.1 Performance

The performance and reliability of the activity server may be undermined by many factors. The foremost is the varying latency in receipt of events from the Listeners. For instance, the phone server reports with minimum latency while the location server has some latency. The finger server may have variable latency depending on net traffic and the kinds of machines polled. Hence, for very fine-grained, up-to-the-second activity reports the activity server would not be suitable.

Secondly, the reliability depends, to a large extent, on external factors. For instance, whether or not the users are wearing their active badges affects the inferences made by the activity server although it will still run without badge sightings data.

For instance, if a user is at a meeting in the conference room and is not wearing her active badge, the finger server is the only Listener that is useful. If the mouse on the machine that the user is logged on is moved slightly for any reason, the finger server will report an active terminal. The inference, the wrong one in actuality, would be that the user is active at that machine. Barring these constraints, the activity server can provide enough information to make reasonable activity classifications.

Thirdly, the nature of the text output is crucial in sharing the knowledge of the activity server. Although factually correct, poorly worded or lengthy text output can be not only confusing, but also tedious. The text generation task is a difficult one, especially if the semantics and naturalness are to be preserved. When the activity server was being used by many people, their greatest problem was in recovering the exact information from the English text.

7.2 Future Work

There are several areas in which the work started by the activity server can progress. The first area is an online calendar, which although more static over the course of a day, is certainly very dynamic over the course of many days or weeks. A modified Unix calendar with granularity of an hour or half hour slots rather than a day could be used as an additional Listener. The only drawback would be that, unlike other Listeners, the calendar server would require direct involvement from the user (i.e., update the calendar regularly). Unless the user is conscientious about updating, the events that the calendar Listener can provide will be too sparse for them to be very useful. Nevertheless, for projections on user activities in the future, the activity server might benefit from an added Listener like the calendar server.

One of the handicaps of the actual implementation of the activity server is the static nature of the rules. Many assumptions have been made in the rules. Should the assumptions change, the rules will be rendered useless. Hence, the ability of the

activity server to read in a new rule-set dynamically ought to magnify the versatality of the server.

The utility of the activity server depends immensely on the history it keeps. Should the server go down, all the history is lost. A mechanism to read the last log file and re-create the state just before the server went down would be very useful.

The actual utility of the activity server can be gauged by a usability study. It would be interesting to find out whether or not people use the activity server actively. Investigating informed users' apprehensions or lack thereof towards a system that monitors unobtrusively might shed light on the actual utility of systems like the activity server.

A system like the activity server might uncover privacy issues. As emphasized earlier, a tool such as the activity server can be used very effectively in small groups of trusting individuals (e.g., project teams). However, the amount of information that the server receives should be controllable by the user himself using a customization directive alterable only by that individual. Therefore, if a user does not want others to get telephone information, for instance, the activity server should be able to shift the emphasis to other Listeners. Having the user decide the inputs into his own model is not new; some user modeling systems have allowed the user to "edit" the model to enhance the model [7]. This "editing" concept can be used to enhance the model and also preserve user privacy. In addition, in future implementations, each of the Listeners and the activity server may require a password so that the information is provided only to trusted clients; the Directory client already requires a password.

Many potential clients can benefit from the activity server. Some earlier work has concentrated on messaging services after locating the user on a local-area network [14]; the Watcher Client provides some messaging service as well but this capability can be greatly augmented.

Finally, the activity server can be said to be an idiot savant. It can report about a limited domain of activities accurately but it cannot learn from data it collects. A

serious user modeling effort can be launched if the activity server could learn from past inputs of the Listeners. Instead, of saying "UserA is not around, she was last seen one hour ago" it could say " UserA was last seen one hour ago, she is probably at lunch because she usually goes to lunch around this time."

Chapter 8

Summary

This thesis implements a system, the activity server, that provides information on activities of members within small trusted groups. It uses multiple information gathering techniques which provide redundant and possibly conflicting information. Among the information gathering sources, the phone server reports on phone activities while the location server reports about physical whereabouts of the user community. Another data gathering system, the finger server, was designed and implemented to provide the activity server with workstation activities of users.

Many important design issues and pitfalls of the activity server and its auxiliary data gathering systems have been discussed herein. It is hoped that the activity server can be used as a tool for the office environment that could augment and enrich the communication within the community of participating members.

Acknowledgments

I would like to thank Barry Arons for offering invaluable suggestions and for finding and squashing socket manager bugs. Janet Cahn, Debby Hindus and Chi Wong and the Speech Group provided helpful hints and gave much needed moral support. Jim Davis was a tireless reader and a well-informed critic. Chris Schmandt, my thesis supervisor, deserves many thanks for posing the problem and giving me many useful suggestions. Sun Microsystems, Inc. provided the support that made this project possible.

My office mates, Foof, Hakon and Nathan brought me food and gave me healthy doses of encouragements, both of which are necessary for survival. Uday, my brother, did my laundry several times and kept me in touch with reality. Debra gave me the support I needed through noisy phone lines from Venezuela. Finally, my parents, living twelve thousand miles away, had the confidence that I would complete this document.

Bibliography

[1] P. E. Agre, *The Dynamic Structure of Everyday Life* Cambridge, Mass. MIT AI Lab TR #1085. 1988.

[2] P. E. Agre and D. Chapman, *What are plans for?* Cambridge, Mass. MIT AI Lab Memo #1050. 1988.

[3] A. Birrel and B. Nelson, Implementing Remote Procedure Calls, *ACM Trans. on Computer System*, **2**, 1 (February, 1984), pp 39-59.

[4] T. S. Cormen, C. E. Leiserson and R. L. Rivest *Introductions to Algorithms.* Cambridge, Mass. MIT Press and McGraw-Hill Book Company. 1990.

[5] C. A. Ellis, *Formal and informal models of office activity.* Xerox PARC Technical document. 1984.

[6] H. Kautz, A circumscriptive theory of plan recognition. *Intentions in Communications* (Eds. Phillip R. Cohen, Jerry Morgan and Martha E. Pollack). MIT Press, Cambridge, Mass. 1990.

[7] J. Kay, *um*: A toolkit for user modelling. Second International Workshop on User Modeling. March 30-April 1, 1990.

[8] K. Harrenstien, *NAME/FINGER Protocol*, RFC 742. 1977.

[9] A. Kobsa and W. Wahlster (Eds.), *User Models in Dialog systems.* New York. Springer Verlag. 1989.

[10] D. B. Lenat and R. V. Guha, *Building large knowledge-based Systems.* Addison-Wesley Publishing Co., Inc., Reading, Mass. 1990.

[11] D. A. Norman, *The Design of Everyday Things.* Doubleday Currency, New York. 1990.

[12] M. E. Pollack, Plans as complex mental attitudes. *Intentions in Communications.* (Eds. Phillip R. Cohen, Jerry Morgan and Martha E. Pollack). MIT Press, Cambridge, Mass. 1990.

[13] C. E. Shannon and W. Weaver, *The Mathematical Theory of Communication.* University of Illinois Press, Chicago. 1963.

[14] L. F. G. Soares, S. L. Martins, T. L. P. Bastos, N. R. Ribeiro and R.C.S. Cordeiro, LAN Based Real Time Audio-Data Systems. Proceedings of ACM SIGOIS '90 Conference on Office Information Systems. 1990. pp. 152-157.

[15] L. A. Suchman, *Plans and Situated Actions* New York, NY. Cambridge University Press. 1987.

[16] S. Tufty, Watcher. MIT Bachelor's Thesis. 1990.

[17] P. H. Winston, *Artificial Intelligence.* Addison-Wesley Publishing Company, Reading, MA, 1984.

[18] C. Wong, Personal Communications. MIT Master's Thesis. 1991.

Appendix A

Finger Server Protocol

A.1 Introduction

The service is a byte-stream Internet protocol based on TCP. The protocol supported is also called fingerserver, and is on TCP port 4500. This port is unique within the Media-Lab and MIT Athena. In addition to the byte-stream protocol, a client library built on top of the byte-stream protocol exists for C programming language. First, the details of the byte-stream protocol will be described - this will lay the groundwork for the higher level C interface.

A.2 Byte-stream Protocol

A client uses *finger server* by opening a stream to the server, sending and reading lines of ASCII, and closing the stream. Client messages to the server begin with a keyword followed by arguments. Messages are divided into two classes:

- **commands** produce exactly one line in response
- **queries** produce an unlimited number of lines in response. The last line will always be blank (only a newline character).

Commands always produce exactly one line of output. If an error occurs, the output will begin with the string `ERROR`. Otherwise, if the command is a question, the line will be the answer. If the command is executed for side effect, the line will be the word `OKAY`.

Note that some commands, known as **requests** cause the server to produce asynchronous output at future times. Some commands need no arguments, others need arguments that specify which one from a given set.

The server is not case sensitive.

A.3 Output Formats

There are two output formats, long and short, for finger server replies. Acknowledgements and errors do not have a short format.

- **username** username of the person. Long format, [PERSON username].
- **hostname** name of the host. Long format, [HOST hostname].
- **terminal** name of the terminal. Long format, [TTY terminal].
- **idle-time** an integer in minutes. Long format, [IDLE idle].
- **alert** kind of asynchronous alert. Long format, [ALERT alert].
- **time** system time. Times can be printed in human-readable or machine readable format.

A.4 Asynchronous Output

In addition to responses to questions and queries, the server also produces output asynchronously when it has been instructed to track people or hosts. This output can arrive at almost any time, so the client should be checking for it. In the C library, a handler may be registered so that it can processed incoming responses asynchronously.

A.5 Messages

`HELP`
Query Returns list of all messages.

`VERSION`
Question Returns a line specifying the version of the protocol. The version at this time is 1.0.

?
Synonym for `HELP`.

`QUIT`
Command Breaks the server connection. This message has not been implemented; for now, the escape sequence of the underlying communication mechanism, eg. *telnet*,

can be used.

`BYE`
Synonym for `QUIT`.

`SYNC`
Command Suppresses asynchronous output. A client uses this command if it has issued any queries (that is, it expects asynchronous output) and it wants to send a command or question. Since synchronous and asynchronous outputs are not tagged, there could be an ambiguity at the client's end if the asynchronous outputs were not temporarily disabled. Hence, a client can send the `SYNC` command. It can then send any message it wants, confident that any output from the server will be in response to that message, and no other. After sending messages, the client should then send the `ASYNC` command to enable asynchronous output.

`ASYNC`
Command Allows asynchronous output.

`LOCATE` *person*
Question Returns the host, tty and idle time of the person.

`LOCATE-ALL`
Query Returns a series of messages for all persons currently located. Note that this is a query, not a request.

`TRACK` *person*
Request Henceforth, whenever *person*'s activity on any host known to the server changes, the server will send a message.

`UNTRACK` *person*
Command Stops tracking *person*.

`TRACK-ALL`
Request Tracks all people.

`UNTRACK-ALL`
Command Cancels any previous tracking requests.

`ADD-HOST` *host*
Start keeping state on *host*.

`ADD-PERSON` *person*

Request Start keeping state on *person*.

`GET-ALL-HOST-INFO`
Query Get info on all users on all hosts.

`GET-HOST-INFO` *host*
Question Get info on all users on *host*.

`LIST-HOSTS`
Query Get the names of all hosts the server knows about.

`LIST-PEOPLE`
Query Get the names of all users the server knows about.

`RESET`
Command Revert to default state (reset modes and requests).

`SET-MODE` *mode*
Command Set output mode. Allowable modes are SHORT and LONG.

`STATUS`
Query Print out your personal requests and modes. Lists three fields: output mode (short or long), asynchronous mode (true or false) and people being tracked (all, none or usernames).

`USER-ON-HOST` *user host*
Question Find out if *user* is on *host*.

`TRACK-LOCATION`
Request Report change in all user location.

A.6 Programmer Interface

A programmer interface, built on top of the byte-stream, exists for C language. Client programs in C can use this interface and appreciably reduce the complexity and detail for interacting with the server.

The FingerInfo structure given below will be filled by the routines whenever applicable. This structure and other definitions are given in the header file finger_client_utils.h.

```
#include <finger_client_utils.h>

typedef struct _FingerInfo
{
    char user     [MAXCHARS];
    char host     [MAXCHARS];
    char terminal [MINCHARS];
    int idle;
    char alert    [MINCHARS];
    struct _FingerInfo *next;
    struct _FingerInfo *prev;
} FingerInfo, *FingerInfo_ptr;
```

```
int fs_init()
```

Connect to the fingerserver. Returns a socket identifier (greater than 0) if successful, -1 otherwise.

```
FingerInfo * fs_locate(fd, user)
    int fd;
    char *user;
```

Get the most recent activity of *user*. If not logged on, *host* field in FingerInfo will contain string "UNKNOWN" in it. Returns a pointer to the FingerInfo structure if successful, NULL otherwise.

```
int fs_locate_all(fd, result) int fd; FingerInfo **result;
```

Get most recent activities of all users the server knows about. Returns the number (0 or greater) signifying the number of users found, -1 for error. The actual finger information is given in *result* which must be freed after calling this routine.

```
int fs_get_host_info(fd, host, result)
    int fd;
    char *host;
    FingerInfo **result;
```

Get idle information on all users on *host*. Returns the number (0 or greater) signifying the number of users found, -1 for error. The actual finger information is given in result which must be freed after calling this routine.

```
int fs_user_on_host(fd, user, host)
    int fd;
    char *user;
    char *host;
```

Returns 1 if *user* is logged in on *host*, 0 if not and -1 if error.

```
int fs_list_hosts(fd, hostlist_ptr)
    int fd;
    char **hostlist_ptr;
```

List the hosts the server knows about. Returns 0 if okay, -1 otherwise. Fills *hostlist_ptr* with host names. Whitespace is the delimiter. Must free *hostlist_ptr* after use.

```
int fs_list_people(fd, userlist_ptr)
    int fd;
    char **userlist_ptr;
```

List the people the server knows about. Returns 0 if okay, -1 otherwise. Fills *userlist_ptr* with host names. Whitespace is the delimiter. Must free *userlist_ptr* after use.

```
int fs_track_all(fd, handler)
    int fd;
    int (*handler)();
```

Request changes in machine activities of all users. Returns 0 if all okay, -1 if an error occurred.

```
int handler(info)
    FingerInfo *info;
{
    /* do whatever you want with info */
    free(info);
}
```

All routines requesting asynchronous events need to register an event handler of the form given above.

```
int fs_untrack_all(fd)
    int fd;
```

Disable receiving asynchronous events about all users. Returns 0 if all okay, -1 if an error occurred.

```
int fs_track(fd, user, handler)
    int fd;
    char *user;
    int (*handler)();
```

Request changes in machine activities of *user*. Returns 0 if all okay, -1 if an error occurred.

```
int fs_untrack(fd, user)
    int fd;
    char *user;
```

Disable asynchronous events about *user*. Returns 0 if all okay, -1 if an error occurred.

```
int fs_set_mode(fd, mode, val)
int fd, mode, val;
```

Set or reset async reporting or output format modes. In mode=1, will start short output format if val=1, long format if val=0. In mode=2, will start synchronous reporting if val=1, asynchronous if val=0.

```
int fs_get_status(fd, sync_ptr, outmode_ptr, tracklist_ptr, users)
int fd;
int *sync_ptr, *outmode_ptr, *tracklist_ptr;
char *users;
```

Get the status of various setting for the client. If *sync_ptr* points to 1, the client is requesting synchronous output, if *outmode_ptr* points to 1, the client is requesting short output format and if *tracklist_ptr* points to 1, the client is requesting track_all otherwise *users* might contain the list of users (delimited by whitespace) that the client is tracking.

```
int fs_reset_status(fd)
int fd;
```

Reset settings to default values: async, long mode and track none.

```
int fs_free(ptr)
    char *ptr;
```

Used to free data pointed to by *ptr*.

Appendix B

Activity Server Protocol

B.1 Introduction

The service is a byte-stream Internet protocol based on TCP. The protocol supported is also called matiserver, and is on TCP port 4500. This port is unique within the Media-Lab and MIT Athena. By desing, some of the protocol implementations are compatible with that of the finger server protocol; however, they are repeated for completeness.

B.2 Byte-stream Protocol

A client uses *activity server* by opening a stream to the server, sending and reading lines of ASCII, and closing the stream. Client messages to the server begin with a keyword followed by arguments. Messages are divided into two classes:

- **commands** produce exactly one line in response
- **queries** produce an unlimited number of lines in response. The last line will always be blank (only a newline character).

Commands always produce exactly one line of output. If an error occurs, the output will begin with the string `ERROR`. Otherwise, if the command is a question, the line will be the answer. If the command is executed for side effect, the line will be the word `OKAY`.

Note that some commands, known as **requests** cause the server to produce asynchronous output at future times. Some commands need no arguments, others need arguments that specify which one from a given set.

The server is not case sensitive.

There are two main classes of responses. One provides the state of current activity, the other provides the history of activity leading up to the current time.

B.3 Output formats

There are two output formats, long and short, for activity server replies. Acknowledgements and errors do not have a short format. The history of activity does not have a short format either.

B.4 Current Activity

The current activity response can have short or long format, synchronous of asynchronous output mode. The following is a description of the fields in the output of current activity.

- **username** username of the person. Long format, [PERSON username].
- **place** room number of the place. Long format, [PLACE place].
- **state** current state of activity. Long format, [STATE state].
- **phone** phone number of nearest phone. Long format, [PHONE phone-number].
- **hostname** name of the host. Long format, [HOST hostname].
- **time** system time. Times can be printed in human-readable or machine readable format.

In addition to responses to questions and queries, the server also produces output asynchronously when it has been instructed to track people. This output can arrive at almost any time, so the client should be checking for it.

B.5 Messages

`HELP`
Query Returns list of all messages.

`VERSION`
Question Returns a line specifying the version of the protocol. The version at this time is 1.0.

`?`
Synonym for `HELP`.

`QUIT`
Command Breaks the server connection. This message has not been implemented;

for now, the escape sequence of the underlying communication mechanism, eg. *telnet*, can be used.

BYE
Synonym for QUIT.

SYNC
Command Suppresses asynchronous output. A client uses this command if it has issued any queries (that is, it expects asynchronous output) and it wants to send a command or question. Since synchronous and asynchronous outputs are not tagged, there could be an ambiguity at the client's end if the asynchronous outputs were not temporarily disabled. Hence, a client can send the SYNC command. It can then send any message it wants, confident that any output from the server will be in response to that message, and no other. After sending messages, the client should then send the ASYNC command to enable asynchronous output.

ASYNC
Command Allows asynchronous output.

LOCATE *person*
Question Returns the place, activity state, phone and host of the person.

LOCATE-ALL
Query Returns a series of messages for all persons. Note that this is a query, not a request.

TRACK *person*
Request Henceforth, whenever *person*'s state of activity change, the server will send a message.

UNTRACK *person*
Command Stops tracking *person*.

TRACK-ALL
Request Tracks all people.

UNTRACK-ALL
Command Cancels any previous tracking requests.

ADD-PERSON *person*
Request Start keeping state on *person*.

`LIST-PEOPLE`
Question Get the names of all users the server knows about.

`RESET`
Command Revert to default state (reset modes and requests).

`SET-MODE` *mode*
Request Set output mode. Allowable modes are SHORT and LONG.

`STATUS`
Request Print out your personal requests and modes. Lists three fields: output mode (short or long), asynchronous mode (true or false) and people being tracked (all, none or usernames).

B.6 History of Activity

The sole message for history of activity is given below. Short or asynchronous modes are not allowed (they would not be meaningful). The output is not in discrete fields but in English text. Although the output changes depending on the state of the activity, the usual format provides the state of the current activity followed by the duration of that activity. If the person is not in his office, the time when he was last in his office, the time he dialed in and other qualifying statements will be provided.
`ACTIVITY` *person*
Question Returns the activity state and its history in English text.

Appendix C

Configuration Files

Both the finger server and the activity server read a few configuration files at startup time. Without the configuration files the servers will not know what entities they ought to save information on. In fact, without these files, the server will stop execution.

The finger server needs two files, .finger_config.hosts and .finger_config.users which list the hostnames and usernames, respectively. Likewise, the activity server will need .mati_config.places and .mati_config.users. All the files take the pound sign # to mean the beginning of a comment till the end-of-line.

C.1 Finger Configuration Files

C.1.1 Hosts Configuration Files

The mandatory field of .finger_config.hosts is the hostname. Other optional fields must follows in the sequence of hosttype, polling frequency and options (to the finger daemon).

The default value for hosttype is 0, meaning a regular Unix machine; 1 signifies it is a machine running Ultrix and 2 means the machine is running a Lisp operating system (e.g. Genera). Polling frequency is in seconds and the default is 90 seconds. The minimum is 60 seconds (anything lower than the minimum will be set to 60). By default there are no options to the finger daemon. Not any option will be supported, however. The parsers within the server may not support all output formats since each option will produce a different output from the machines (finger daemon).

```
.finger_config.hosts
# Configuration file of hosts for the finger server.
# Pound sign (#) signifies the start of a comment till the end-of-line
# Host types are:
#define FS_HOST_SUN      0
```

```
#define FS_HOST_DEC      1
#define FS_HOST_LISPM    2
#
#hostname        hosttype        pollfrequency   options

media-lab        1               240             -i
toll             0
kilimanjaro      0
#obvious         2
```

C.1.2 Users Configuration Files

There is only one field that will be read in the Users configuration file, .finger_config.users. This field is the username of the person. The pound sign can be used to put comments.

```
.finger_config.users
#users

geek # Chris Schmandt
jrd    # Jim Davis
ccwong # Chi
hindus # Debby
```

C.2 Activity Server Configuration Files

C.2.1 Places Configuration Files

The only mandatory field is room number or name of the place (whatever appears in the DB). An optional field, that will make assertion clearer, is the type field; offices are 1, lab area 2, conference rooms 3 and corridors are 4. If no type is given, it is assumed to be an office except for corridors which are always known as corridors due to their second character being a 'c'.

```
.mati_config.places

#places type

327   1                 # geek's office
331     1                 # cubicles outside geek's office
343     2                 # sound studio
```

```
344     2                 # garden
345     2                 # sound room
350     1                 # lisa's office
352     1                 # chi&debby's office
353     1                 # barry's office
355     1                 # sanjay's office
356     3                 # conference room
3c2     4                 # 3rd floor corridor, Boston side
3c1     4                 # 3rd floor corridor, Garden side
3c22    4                 # 3rd floor corridor, Fenway side
```

C.2.2 Users Configuration Files

This file is similar to its finger server counterpart. There is only one field that will be read in the from the users configuration file, .mati_config.users. This field is the username of the person. The pound sign can be used to put comments.

```
.mati_config.users
#users

geek # Chris Schmandt
jrd   # Jim Davis
ccwong # Chi
hindus # Debby
warlord # Badger guy
```