A System Architecture for Context-Aware Mobile Computing

William Noah Schilit

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

1995

ABSTRACT

A System Architecture for Context-Aware Mobile Computing

William Noah Schilit

Computer applications traditionally expect a static execution environment. However, this precondition is generally not possible for mobile systems, where the world around an application is constantly changing. This thesis explores how to support and also exploit the dynamic configurations and social settings characteristic of mobile systems. More specifically, it advances the following goals: (1) enabling seamless interaction across devices; (2) creating physical spaces that are responsive to users; and (3) and building applications that are aware of the context of their use. Examples of these goals are: continuing in your office a program started at home; using a PDA to control someone else's windowing UI; automatically canceling phone forwarding upon return to your office; having an airport overhead-display highlight the flight information viewers are likely to be interested in; easily locating and using the nearest printer or fax machine; and automatically turning off a PDA's audible e-mail notification when in a meeting.

The contribution of this thesis is an architecture to support context-aware computing; that is, application adaptation triggered by such things as the location of use, the collection of nearby people, the presence of accessible devices and other kinds of objects, as well as changes to all these things over time. Three key issues are addressed: (1) the information needs of applications, (2) where applications get various pieces of information and (3) how information can be efficiently distributed.

A *dynamic environment* communication model is introduced as a general mechanism for quickly and efficiently learning about changes occurring in the environment in a fault tolerant manner. For purposes of scalability, multiple dynamic environment servers store user, device, and, for each geographic region, context information. In order to efficiently disseminate information from these components to applications, a dynamic collection of multicast groups is employed. The thesis also describes a demonstration system based on the Xerox PARCTAB, a wireless palmtop computer.

Contents

List of Figures

List of Tables

Acknowledgments

Marvin Theimer has advised, encouraged, and inspired me through the arduous doctoral process. I feel extremely lucky to have worked under his guidance. He always had faith, and he always believed in me and for that I owe him my eternal gratitude.

I also owe a large debt to Mark Weiser, who's vision of ubiquitous computing got me excited once again about computer science. As the manager of the Computer Science Lab at Xerox PARC, Mark created a unique environment for research and invited me to become a part of it.

My advisor Dan Duchamp deserves special thanks for his hard work and support. He unselfishly gave me the extraordinary opportunity to work on a "long-distance" dissertation.

I'd also like to thank the members of the PARCTAB project including Roy Want, Norman Adams, John Ellis, Karin Petersen, and David Goldberg for all they taught me. Roy was a great project leader and an even greater friend. Doug Terry, Brent Welch, and everyone else at PARC helped make my time especially fruitful and enjoyable.

I gratefully acknowledge the financial support of Xerox, and of the U.S. Advanced Research Projects Agency under contract DABT63-91-C-0027.

Finally, I'd like to recognize my family. My parents, Henny and Barry, bestowed on each of their children a love for learning, and my sisters, Lisa and Rebecca, showed me the way.

This dissertation is dedicated to my best friend Kerny McLaughlin. She pulled me through this experience in one piece, and stood by me during all those years.

Chapter 1

Introduction

Over the last decade advances in digital electronics have made computers smaller, cheaper, and faster. This trend, along with other industry advances, has promoted the development and rapid market growth of small computers that can be carried from place to place. It has also created a revolution in the consumer marketplace where computers are now commonly embedded in everything from household appliances to automobiles. Whereas once it was necessary to visit special climate controlled buildings housing computer centers in order to interact with mainframe computer systems, it is now possible to carry computers with us and to communicate on the spur of the moment with computers embedded in the world around us.

The challenge of interacting with ubiquitous computer technology is the motivation for this research. This thesis focuses on an increasingly common form of computing in which users employ many different mobile, stationary and embedded computers over the course of the day. In this model computation does not occur at a single location in a single context, as in desktop computing, but rather spans a multitude of situations and locations including the office, meeting room, home, airport, hotel, classroom, market, bus, and so on. Users might access their computing applications from wireless portables, via stationary embedded devices, or from traditional workstations connected to a local area network.

Such a collection of mobile and stationary computing devices that are communicating and interoperating on the user's behalf is called a *mobile distributed computing system*. This form of computing is broader than mobile computing because it concerns mobile *people* not just mobile *computers*. These systems aim to provide people with ubiquitous access to information, communication, and computation.

One significant aspect of this form of computing is the constantly changing execution platform. The processors available for a task, the devices accessible for user input and display, the network capacity, connectivity, and costs all change. In short, the hardware configuration seen by users is continually changing. Similarly, because a person can move from one location to another, joining and leaving groups of people, the social and environmental situation in which a computer is employed is also continually changing. Knowledge of these dynamic conditions is important if software is to support a lunch room discussion among peers as well as a corporate board meeting. These distinct contexts of use affect how intrusive a computer should be and can also be an indicator of the kinds of information and actions that will be accessed. For example, an electronic whiteboard application used during meetings might automatically display the meeting agenda and notes from previous meetings.

The collection of things surrounding an individual's use of computers constitutes a mobile distributed computing environment. The computing environment changes from place to place and over time. It includes hardware and software configurations, user customizations, as well as the location, people, situation, and resources at the point of use.

The dynamics present in mobile distributed computing presents two challenges, one old and one new. The first challenge is to permit continued operation across changing circumstances in a seamless manner. That is, to enable transparent distributed computing. In some cases such as when switching from being connected to disconnected the seams will inevitably show through. This poses a different aspect of the old problem of supporting transparency where the challenge is to present and enable the most sensible transitions. The second, novel, challenge is to exploit the changing environment with a new class of applications that are aware of the context in which they are run. Such *context-aware software* adapts according to the location of use, the collection of nearby people and objects, the accessible devices, as well as changes to those objects over time. A system with these capabilities surveys the computing environment and reacts to changes to that environment.

The advantages of meeting these challenges can be illustrate by some examples:

- *Management of time-varying resources.* Software should be able to tune itself so it doesn't access the network on demand but rather takes into account the changing bandwidth and monetary costs balanced with the priority of the task at hand. High-speed video might throttle back on resolution if necessary, and news readers might download and cache many news groups, even if they may not get read, before disconnecting from a high bandwidth network. In general, scheduling algorithms managing time-varying resources should adapt current work queues in response to changes in the underlying resources.

- *Configuring dynamic systems.* The particular physical devices accessible to a mobile user depends on proximity. To simplify the user's choices a user interface (UI) might use knowledge about location and accessibility to assist in the selection and presentation of printers, displays, and other devices. For example, when asking a system for a printer it should be possible to show the closest printer at the top of the selection list. Another illustrative example of configuring dynamic systems is an application that adapts the size and content of windows when moving between display types.

- *Supporting opportunistic interaction.* When system components come and go it may be necessary for applications to postpone actions until certain prerequisites are met. It is desirable to let users act as if they are fully connected–to the extent possible–even when they are not. For example, in the case of a mobile hosts, a network file system should try to postpone updates until the activity will be unobtrusive, and mail programs should try to batch messages for transmission until a low cost network connection is made. The novelty is in making these kinds of actions automatic.

- *Contextual customization.* Mobile systems enable computer interaction in non-traditional settings, such as kitchens, hallways, meeting rooms, and airports. Software customization should not only take into account a user and host, but also the broader context of use. For example, the type of meeting a person is attending, and the other people present, should help decide whether e-mail messages or telephone pages need to be immediately delivered.

A key distinction that the reader should take away from this section is the emphasis on the mobile *person* who interacts with a changing multitude of people, computers, devices, and environments. This is in contrast to more traditional models that focus on individual computers, mobile or otherwise.

1.1 Thesis Statement

This thesis contends that traditional software approaches are ill-suited for building mobile distributed computing software. System dynamics result in software that must be manually reconfigured when features change, and software that is constantly mis-tuned for the current computing environment.

This thesis proposes an architecture for notifying applications of the changing computing environment that is simple, fault tolerant, efficient, scalable, and feasible to implement across a range of computing platforms. Several new techniques are presented in support of this goal:

New Technique 1: Most software is designed with the assumption that the system it operates in will not change significantly over time. Consequently the standard approach to application customization is to employ a one time initialization at program startup. The approach taken here is a regular architecture for communicating execution context changes so applications can re-customize at any time during program execution. This thesis also describes how one can automate the mechanical aspects of connection maintenance, information propagation, and failure recovery. However, unlike the standard approach of masking failures, it is argued that failures need to be reflected, in a controlled manner, to higher software levels. That is, failure information is treated as just another type of context change.

New Technique 2: The conventional approach for customization is to limit, either by process hierarchy or login session, the sharing of customization information among processes. Because users interact with a multitude of stationary and mobile systems through the course of a day, mobile systems need to maintain and share a context that is not limited to the boundary of process hierarchy, window manager, or host. This thesis promotes the use of a *user agent* to support a persistent dynamic execution context for all of a user's applications.

New Technique 3: Much of the dynamic information in a mobile distributed system can be organized by location. Doing so permits uniformity by allowing many different kinds of dynamic information to be obtained using one primary search mechanism. This thesis proposes a dynamic environment service that melds cartographic information with dynamic location information of entities in the environment. These *active maps* describe the location and characteristics of entities, including people, devices, and location-based service registrations, that may move over time within some region. The contribution is a unifying means of dealing with a large part of the dynamic information in a system.

New Technique 4: Disseminating information among processes in a distributed system may involve unicast, broadcast, or multicast messaging. The conventional approach for a system with a sender and multiple clients is to use multicast and expect clients to filter unwanted information. However, among wireless hosts the cost of client filtering is substantial due to bandwidth and processor limitations. A new approach addresses how to multicast selected information to many bandwidth-limited clients.

The system architecture is based on a *dynamic environment interface* which has four important characteristics making it well suited to support context-aware computing. First, it provides a uniform and efficient mechanism for communicating system dynamics between software components. Second, it incorporates a flexible way to express the characteristics of entities in the computing environment. Third, it handles server failures in a uniform and transparent way. And finally it provides applications with feedback on the timeliness of the information on which they are basing their context-based adaptions. There are many useful services that can be built based on the dynamic environment interface. Three of these specializations are of particular interest in the architecture presented: user agents, device agents, and active maps. Together these components provide a framework for context-aware computing.

1.2 Thesis Overview

The key issues that this thesis resolves are:

Information needs of applications. Motivated by categories and examples of context-aware applications this thesis examines the information needs of applications. This includes how to provide dynamic information to applications without overloading their communication links or computational resources.

System Structure. This thesis presents the recommended distribution of system functions into components and the design of protocols connecting them. Also, it describes how to organize information so applications can easily discover and inspect their computing context.

Efficient information dissemination. Two aspects of information dissemination are examined. First, how to manage and disseminate information in the face of bandwidth and connectivity characteristics of mobile computers. And second, how dense concentrations of context-aware applications can achieve reasonable performance.

Chapter 2 explores aspects of mobile distributed computing systems and introduces the issues that this thesis resolves. A system model presents the physical properties that must be managed including issues specific to mobile computing: limited bandwidth, connectivity, limited resources, and partitions. An exploration of the types and range of system dynamics gives an idea of both the limits the system must operate under as well as the kinds of information it must manage and can exploit.

Chapter 3 presents some background for this work. The related areas presented here include mobile computing, ubiquitous computing, location-based computing, and information dissemination.

Chapter 4 describes how information is disseminated using a common dynamic environment interface that manages fault tolerance issues. This include an exposition of how to efficiently dissemination information to clients.

Chapter 5 describes the key components in the architecture that employ the interface, namely the user agent, device agent and active map server.

Chapters 6 and 7 describe how the architecture is deployed in a particular mobile distributed computing system, along with the implementation and performance considerations.

Finally, Chapter 8 contains a summary and conclusion, along with some possible directions for future research.

Chapter 2

Aspects of Mobile Distributed Systems

This chapter lays the groundwork for the thesis by examining dynamic aspects of mobile distributed systems. It starts by defining a model for mobile distributed systems based on physical system characteristics. Three physical traits stand out as distinctive: the dynamics of communication, context, and location. For each dynamic quality the range of states and the consequences in terms of software systems is presented. The chapter then explores how software systems can manage and exploit these characteristics, and concludes with a description of architectural considerations and assumptions necessary for building context-aware software. Throughout this chapter specific problems that need to be addressed are described.

2.1 System Model

The mobile distributed model presented here is an extension of a standard distributed computing system model. We first describe a representative model based on [25, 27], and then the extensions. Other distributed computing models are also suitable since the basics are fundamentally the same. These basic physical characteristics include:

- A number of processors (hosts) connected to one another by a communications network.

- Each processor may have a number of user processes running on it.

- Hosts communicate through messages transmitted over the network.
- Message transmission times are variable and messages can be lost or corrupted during transmission.
- High level protocols can be built on top of datagrams to provide reliable, sequenced, point-to-point communication.
- Networks may partition, hosts may crash, and not all hosts have stable storage.
- A multicast layer allows sending a message, unreliably, to a number of processes, possibly on different hosts.

These characteristics, except perhaps for the last, are commonly found in architectures of LANs and WANs. Multicast, though less common, is available on the Internet [11], and its deployment is quickly spreading as new application domains are developed to exploit it [13]. We assume that multicast communication costs, on the average, are unit message cost for sending, and reception costs are comparable to hardware filtering of broadcast messages. Some implementations are more efficient while others are less. Multicast is included in this model because, as we will show later, it is a powerful tool for information dissemination in a mobile distributed system.

In addition to the characteristics above, the mobile system incorporates these additional physical properties:

- The network may consist of hosts linked by any and all of the following: a permanent cable, a temporary connector such as a “tether” or “dock”, or a wireless communication media such as radio or infrared.
- Some hosts are mobile and may be moved from one location to another.
- Hosts may use different communication mechanisms at different times.
- Host connectivity is variable, ranging between permanently connected, frequently connected, and rarely connected.

- Some hosts are limited in computation power, memory capacity, and network bandwidth resources.
- Users are mobile and may move between and interact with different sets of computers and devices.

The above properties show the important fact that there are three styles of mobile computing captured by this model. First, there is mobile computing that occurs when portable computers are integrated with wireless networks. Mobile computing can also describe portable computers that are used in conjunction with temporary network connections such as telephone modems. Third, mobile computing occurs when mobile users interact with traditional workstations and networks.

This model presupposes that components can be reached, at least intermittently, by some form of packet-based communications, such as Internet-IP, without knowing the destination's current location. This requires that applications running on mobile hosts either be able to use some form of "mobile-IP" communications protocol, for example [18], or that an agent architecture be in place in which an agent process on a stationary host takes the responsibility of rerouting packets destined for a mobile host in an appropriate manner. An example of the latter architecture is described in [34].

Although vehicular systems may be a desirable topic of study, this thesis focuses on a person-centric world view. As such, the system design is concerned with human-inhabited spaces, pedestrian distances, and human time-scales. The design is concerned with people and their interactions in a world of terminals, printers, hosts, devices, and other people. It is not intended for micron-sized and nanosecond-timed interactions. Since mobile people and computers are central to the model, it is also important to recognize that movement between administrative domains is one characteristic of the system model.

We've chosen to aim our system at an environment with the following characteristics:

- The system scales to a size that encompasses hosts, routers and networks, managed by multiple administrative authorities.

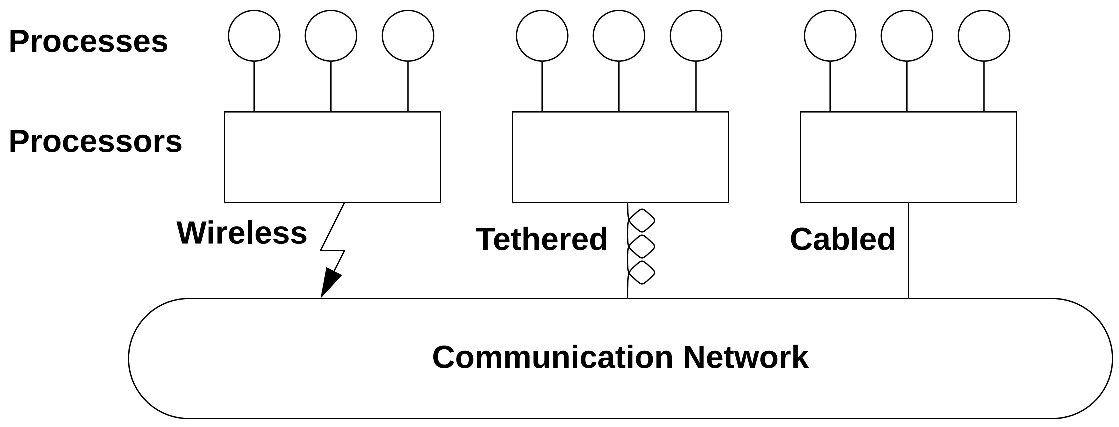


Figure 2.1: System Model

- There exists a source of location information of sufficiently small granularity to identify the particular room a mobile user or host is situated.
- The system is used in an office setting, where people carry none, one, or a few devices, and interact with a few to a few dozen devices.

In order to meet the requirement of location information, there are a variety of technologies available for tracking the locations of people and mobile computers. Active badges [44] can be attached to objects and monitored by sensors in the ceilings of rooms and corridors. Wireless nano-cellular communications [2, 12, 20, 34] can be used to communicate with palmtop and notebook computers and provide room-sized cell location information. Radio-based triangulation systems, most notably the Global Positioning System (GPS), is another location sensing technology [19, 37]. With less technology, input activity at keyboards and mice can be monitored to detect the presence of logged-in users in front of stationary workstations. Also, the user can simply manually enter specific location information, e.g., “set location kitchen at home”.

Figure 2.1 shows a model of a mobile distributed computing system. A communication network connects processors and processes. Three host to network connection types are shown: permanently cabled, temporarily tethered, and wireless.

Communication Media	Bandwidth
cellular telephone	9.6 Kbps
modem	9.6-19.2 Kbps
infrared	19.2 Kbps–4 Mbps
ISDN	128 Kbps
radio	300 Kbps–10 Mbps
Ethernet	10 Mbps
FDDI	100 Mbps
ATM	155 Mbps (point-point)

Table 2.1: Representative Communication Bandwidths

2.2 System Dynamics

This section describes the dynamics present in mobile distributed computing systems. The types of system dynamics can be broken down into three categories: communication dynamics, environmental dynamics, and location dynamics. It is these dynamics that contribute to the changing context for an executing application. Characteristics of each of these categories is presented below, along with a description of possible ways applications might adapt their behavior to different system changes.

2.2.1 Communication Dynamics

In our system model, communication may be provided by wireless transmission or physically cabled connections. Wireless hosts may be temporarily docked to high speed networks and may become disconnected altogether. This section explores the range of characteristics exhibited by communication media and the consequences on the software that employs and switches between these media.

Table 2.1 shows representative bandwidths for different media: there are huge variations in data rates among the different media. Bandwidth, however, is only one characteristic that varies considerably from one media to another. Cellular modems, for example, have substantial connection setup time and usage costs, and radio in general tends to have

bandwidth
error rate
connection setup time
usage costs
security requirements
contention
disconnection rate
round-trip delay

Table 2.2: Representative Communication Dynamics

communication drop outs caused by lack of coverage and interference. Error rates are highly variable even over a single media. Machinery, walls, buildings, lights (and other environmental factors) may all interfere with wireless signal reception.

One ramification of mobility in wireless communication is that the number of users in a wireless cell varies dynamically and therefore large concentrations may overload network capacity. A large concentration may occur, for example, when members of an organization all attend a seminar. Weiser [47] points out that network bandwidth is divided among all users of a radio cell. Since this value depends on the size and distribution of a user population, he suggests measuring wireless network capacity by bandwidth per cubic meter.

Table 2.2 summarizes some of the dynamic characteristics that may effect software. These include variations in bandwidth, error rates, connection setup time, transmission charges, the need for data security, and the density of users in a cell. The important consequences of this range of characteristics is that (1) different protocols may need to be employed; (2) change in application behavior may be desirable; and (3) change in application functionality may occur. These consequences are described in more detail below.

Protocol Changes

Network protocols are often tuned for operation under certain static conditions. The Sun RPC protocol [9], for example, has message retransmission tuned for an Ethernet. This and other early decisions of engineering trade-offs becomes a problem when the underlying assumptions change. Specifically, when communication characteristics are dynamic, protocols must quickly and accurately adapt to new configurations. For example, if protocols are to adapt to communication characteristics, then two techniques they might employ are batching and compression. These have benefits over slow link media, however, they also have drawbacks which may make them less desirable over fast channels: batching may produce transmission delays, and compression requires extra processing at the sender and receiver.

The use of different protocols may also be desirable in the area of data security. Many system administrators rely on ownership of the cables carrying, for example, Ethernet signals from one host to another. Radio transmission, however, is less easily guarded, and may more easily result in eavesdropping or the stealing of billing codes, an increasing problem for the cellular telephone industry. Encryption is therefore more important for wireless communication than for a network over which one has control of the physical transport. The consequence is that when moving among different media, different levels of security may be necessary. Always providing the maximum level is not practical because, in general, the more the security, the higher the cost. This cost is born by processor and communication resources which must perform security protocols and encryption, and also by the user who is inconvenienced by having to prove their identity.

Finally, the heterogeneity of different networks and protocols poses additional complexity for mobile hosts. Hosts may need to switch between one network and another, for example from cellular modem to ethernet. One particular problem that arises is the accurate control of when this switch occurs. Because of the disparity of communication usage fees, making the wrong decision can be expensive.

Application Behavior Changes.

Applications can benefit substantially from adapting to the huge changes in network bandwidth and from understanding the sometimes-disconnected nature of mobile computing. A convincing example is the fact that hosts are not always connected and therefore certain network functions are simply unavailable. Disconnection may be voluntary such as removing a machine from a network dock or terminating a phone call. In other cases disconnection and reconnection are unpredictable, such as when moving between wireless communication cells. In real time, applications need to adapt their behavior in order to operate under both high-bandwidth and low-bandwidth conditions. The goal of these behavior modifications is to make interactions with applications more predictable and make applications more efficient under varying communication conditions. The following discussion explains different ways applications might adapt their external behavior according to the changing network state.

One approach towards this end is to have applications adapt user interfaces. A button that requires network resources can be shown in a disabled gray form when the network is offline. In this way intelligent user interfaces can provide feedback on which user operations are unavailable [42]. Similarly the confirmation behavior of user interfaces might change for time consuming operations, or the expected time to complete various network intensive functions might be displayed near the buttons that invoke those functions.

A second approach is to adapt the behavior of output procedures. An application can present variable degrees of detail and require that the the user explicitly request the fine grained information. For example a graphic image would be transmitted in low resolution, and a document might only include an abstract and section headings. In the case of a network window system, dragging a window around might only show an outline until it is placed, instead of showing the entire window being dragged.

Finally, the feedback behavior of an application can be adapted. For example increasing the interval between checks for new messages in a mail reader application can trade off less timely notification for less network traffic. Similarly, applications can change the update rate of the status of an ongoing print job. More noticeable adaptions may include changing the method by which user feedback is presented: user interface widgets can change from sliders, which because of their continuous redrawing require a large amount of bandwidth when used across a network, to buttons, which requires less bandwidth.

2.2.2 Environmental Dynamics

The second dynamic aspect of mobile distributed systems are those changes to the phenomenological world in which computer interaction occurs. The material world people

people present
social situation
physical conditions

Table 2.3: Representative Environmental Dynamics

sense includes the environmental conditions such as light and noise levels, people around us, and social situation. These are the situations in which mobile distributed computer interaction occurs so we call this aspect of mobile distributed computing *environmental dynamics*.

Environmental dynamics has always, to some degree, been a factor in computing. However, the important difference is that for the most part people have always computed in an environment under their control, for example in an office, and have computed in a mostly unchanging environment. Mobile distributed systems enable computer interaction in non-traditional settings, such as kitchens, hallways, meeting rooms, and airports. When computers are used in these diverse settings the environmental dynamics are more variable, less predictable, and consequently more significant.

Three categories of environmental dynamics are: (1) the differences that occur when alone or accompanied by other individuals; (2) the changing social situation; and (3) the changing physical environment conditions. As with communication dynamics, the important consequences of these dynamics is that different application behavior is desirable in different situations.

People entering and departing from the vicinity of a mobile user create a dynamic situation that can be employed by context-aware software. One application for mobile hosts is to act as a smart remote control, customizing the environment a person finds himself in. For example, changing the lighting or air-conditioning level. However, the presence of other people creates complications for software that exercises control over an environment. When a work or living space is shared, other people's preferences might need to be taken into account and a person's preferences may change depending on whether they are alone or with other people.

When people gather into groups a social situation results and as groups change so does the social situation. Such changes may lead to desired adaptions in software controlling phones, pagers and other communication devices. For example, in an office setting as people move from one activity to another, it is common to vary the degree to which interruptions are acceptable. When attending meetings people generally do not want to be interrupted unless it is important, however when alone or during informal meetings it might be desired. This type of decision making takes into account not only whether people are present, but also their relationship to you and the activity being performed, i.e., the social situation.

Finally, the environmental conditions including the light and sound levels change as people move from one place to another. Light levels can be used as an indicator of activity: dark rooms often have little human activity. Variation in light level therefore can be used to signal adaption in response to low activity. The Xerox PARCTAB, for example, adapts it's beacon rate according to the surrounding light level, slowing down when in dark areas and thereby conserving battery power [45]. Similarly readability might be improved by adapting the brightness of displays depending on the ambient light level. Sound is another environmental condition that context-aware software may exploit. Detecting noise levels above a threshold can be used to turn off sound production so as not to participate in a cacophony. Alternately, an application may increase an alarm's volume so it can be heard above background noise. Finally, both light and sound levels provide information for applications to deduce whether computer output actually reaches users.

2.2.3 Location Dynamics

The third kind of dynamics in a mobile distributed system is location. Users may move from one host to another and change the location in which they compute. Similarly people might carry mobile hosts or devices and change both their location and the device's locations. Another aspect of location dynamics is that when people move, their relationship to stationary objects change. In addition, a person's relationship to other mobile people and devices is constantly changing.

Room-level location information is a scale that suffices to demonstrate the ideas in this thesis. Because the system model is geared towards a person-scale environment, this level of granularity supports the common architectural divisions people work and live in. Larger granularities, such as cities, will miss out on an important group of context-aware software that differentiate objects that are close enough to interact with from those that are not. Although higher resolution is desirable, current technology for fine-grained reporting over large regions is not generally available. As other technologies are introduced sub-room level location information is likely to become more desirable, but this level of location granularity is not within the scope of this thesis.

The following sections describe the issues pertaining to location information and the consequences of host and user mobility.

Location Information

Location information is particularly important because it provides an especially useful index to retrieve other information, such as social situation and physical conditions. When location is known, much of the additional information needed to support context-aware computing may either be inferred or looked up.

When managing room-scale location information, it is useful to realize that people tend to move along certain well-defined paths, down stairwells, through hallways, and along roads. Information about a person's location therefore does not have to follow an abstract geometric model, such as a coordinate-based Euclidean-space, but rather can be represented by a more intuitive human-space model.

In order to navigate and interact with a world of computational objects it is necessary to have a notion of geography and location. The kinds of location information that are commonly used in our human-spaces are proximity, containment, adjacency, and how to move from one place to another. The problems that must be resolved are what geographic organization of information supports queries of "nearby" objects.

The importance of "place" for a broad class of objects leads to the definition of the term *located-object* to refer to entities that have an associated physical location. Examples of located-objects include persons, printers, terminals, and workstations, as well as location-dependent services, such as information agents associated with particular parts of a store or building.

Example Consequences of Location Dynamics

There are both system and social consequences of location dynamics. From a system viewpoint, when mobile hosts can move from one place to another, they may also move out of range of their current wireless cell and into another. This raises the problem of maintaining continued network connectivity, a problem known as Mobile-IP [18].

Another system-level consequence of user mobility is authentication. Currently once a user has logged into a machine the expectation is that they remain present at that machine. However, for mobile users interacting with stationary devices, logging out and logging in again are continual actions. Of course authentication is not necessary if the functions to be accessed are public. However, that means that personal customizations are not applied, and private data can not be accessed, and therefore the system will not appear seamless.

Another consequence of user mobility is that it changes the ability to use nearby devices. That is, certain devices, especially user interface devices, must be nearby in order to use them. For example, a display must be within sight, a speaker within hearing distance, and a keyboard or mouse within reach.

A social implication of location dynamics is that certain actions that may be desirable in one location are not in another. This is true with other technologies: a cellular phone ringing in a concert hall is socially unacceptable. In general, changing ones location may also change the desirability of operations and information. For example, when searching for a place to eat it is usually nearby restaurants that are most important.

2.3 Context-Aware Computing

This section explores the problems and issues related to context-aware systems. Some illustrative examples are given. The context-aware computing cycle, consisting of discovery (dynamic data), selection (location-based information), and use (application adaption), is described next. Following this is an exposition on the types of information for use by mobile applications along with how that information can be used. This is followed by the issues and problems in the areas of system modeling, information dissemination, and application adaption. This section concludes with architectural considerations for a context-aware computing system. These are the issues to be addressed by the design presented in chapter 4.

2.3.1 Some Examples

Context-aware computing is the ability of a mobile user's applications to discover and react to changes in the context in which they are situated. We list a few concrete examples to illustrate:

- To provide time-of-use assignment of engineering trade-offs such as protocol constants and layers.
- To help navigate the computerized world by providing a display of "interesting" located-objects, both nearby and far away.
- To keep a record of located-objects and persons one has encountered, for use by applications such as "activity-based information retrieval," which uses the context at the time the data was stored to assist in retrieval [23].
- To detect location-specific information, for example, electronic messages left for the user or for public perusal.
- To keep a look out for nearby devices that can be used opportunistically by applications, such as additional display terminals in a room.
- To detect nearby people, located-objects, or services that are relevant to reminders or actions set to be triggered by their presence.
- Tracking a particular located-object as it moves around a region. Examples include tracking a co-worker you wish to talk to and tracking the office coffee cart in order to be made aware when either is nearby.
- Tracking located-objects with a specified set of attributes in a particular region. An example is tracking all members of a workgroup.

- Finding the located-object nearest to a specified location (usually their own) that meets a specified set of constraints. Examples include finding the nearest printer and finding the nearest system administrator.

- Monitoring the activity at a particular location. A typical example is keeping track of available terminals at one's current location.

These examples serve to reinforce some of the ideas presented elsewhere in this chapter. First that applications may deal with a wide variety of information stemming from communication, environmental, and location dynamics. This information includes parameters and user customizations. Also, an important class of information in a context-aware system is the located-object. It is natural that located-objects have some manner of geographic organization. That querying, comparing, or ordering this information involves some knowledge about geographic relationships. Additionally, information about proximate objects–things that can be seen, heard, and touched–is of particular interest. Finally, learning about changes over time as opposed to simply querying is desirable.

2.3.2 Context-Aware Cycle

The actions carried out by context-aware software can be viewed as a three step process.

- Discovery – learning about entities and their characteristics.

  When the computing-context changes, a system must find out what new resources are available and what their characteristics and capabilities are. This context information must have a representation, be capable of inspection, and propagate to applications when changes occur.

- Selection – deciding which resources to use.

  A system should be able to select entities based on the surrounding context. For example, where the user is located, and what co-located objects are present. How selection decisions are expressed, by type, location, etc., is a key concern given the enormous amount of information potentially available in the environment.

Category	Examples
Physical Objects	people, computers, displays
Conditions & State	available, in-use
Spatial Relations	with, near
Phenomena	dimly lit, noisy
Ways & Means	instructions, network address
Customizations	hold calls

Table 2.4: Summary of Information Types

- Use – employing the resource.

  Employing resources may or may not require application involvement.

Discovery, selection, and use of contextual information requires an underlying well-defined collection of information representing the entities in the environment and their characteristics.

2.3.3 Types of Information

A key point about the dynamics present in mobile distributed systems is that a wide variety of information about the world is important. This information is summarized in table 2.4 and described in the paragraphs below.

Physical objects ranging from people to controllable devices to furniture must be modeled. People, printers, sensors are examples that have dynamic state whereas others, such as furniture, and hallways do not. The state of a device might be on, off, running, or idle. For a person it might be “alone,” “busy,” or “sleeping.” As mentioned, many objects also have a location that needs to be described. Location can be represented as coordinates in a system or presence in a containing space, such as a room.

Information about ways to achieve a task also need to be represented. Mobile users interacting with unfamiliar equipment need information about ownership and direction and other assistances for use. For example, a user might want to access instructions on how to operate a video recorder, or how to replace toner in a printer. In addition, software dealing with novel devices needs information on capabilities.

Finally, information about customizations need to be stored. This includes personal preferences as well as customization for a group, owner or other authority. For example, a user's directive "when in meetings record audio," a constraint "no messages during meetings," or setting the window size for different devices. Group customizations may provide information on how to resolve conflicting personal customizations and owner-authority customization may specify usage policies.

2.3.4 Modeling a System

In order to define a data model for a context-aware computing system a data format capable of representing the information outlined in the previous section is required. There are a number of possible representations for this information. They range from primitive data types to full types and objects. Primitive types are flexible but, depending on the system requirements, they may not be efficient or concise enough for representing the desired information.

Whereas production-quality systems may choose a complex representation, this thesis employs a fairly simple data model. This should not be considered a drawback since a "simple" format is easier to make universal and there isn't a significant performance penalty for such an approach. The cost of a simple data representation, such as text, is mostly the expense of encoding and decoding. In this thesis the dominating factor is the time for network transmission, so the performance improvement of a rich type space is not critical.

In a context-aware system, shared-information is primarily used for the descriptions of located-objects. In order to access this information, users employ generated queries from applications. These queries need a search structure on objects, in particular the definitions of keys over which the query search is defined. If only queries accessed located-objects then perhaps no further structuring would be necessary, except perhaps to satisfy the desire to define a common interchange format. Applications that issue queries may very well forgo the external description of data objects (such as a schema), since this information can be an implicit part of an application's design.

However, accessing information from exploratory browsers is distinct from query-based information access because whereas an application may have an implicit understanding of data formats, browsers generally do not. Browsing therefore introduces a need for structural information that may be satisfied by either schemas or self-describing data. A problem with schemas and other meta-descriptions of data formats is that they add management overhead, especially when distributed across multiple severs. One way to avoid this complication is to include the structural description of objects along with their contents. For the purposes of this thesis, self-describing data is sufficient to illustrate the points. However, one drawback of this kind of self-describing data is that it requires additional storage for each individual data structure.

2.3.5 Information Dissemination

A context-aware system must determine how information is propagated among system components. The design must accommodate slow links, partially connected hosts, and variable communication speeds. Slow links may not be able to transmit all the context changes an application desires to see. Partially connected hosts must manage information that may or may not be out of date. Finally, wireless hosts may be moved into high concentrations where the slow wireless communication bandwidth is further reduced by the need to share the medium.

Applications that probe for context changes trade off the overhead of polling versus the timeliness of the information they receive. That is, to reduce the interval from when information changes at a server to when an application sees the change at a client requires reducing the interval between successive polls. Another drawback of polling is that it generates a constant network overhead even when information changes infrequently. The alternative approach is to employ a publish-subscribe paradigm that limits the extraneous network traffic by using callbacks from servers to clients. The price is increased complexity for maintaining callbacks in the face of faults. However, by automating the callback-maintenance process–essentially automating re-binding and re-subscription–this disadvantage is not a problem.

Adaptive applications need to be exposed to system dynamics in such a way that they are not overwhelmed with information or error conditions such as network failure. There are two issues. First, the inefficiency of overloading communication links and processors with unnecessary information; second, the inconvenience of applications having to learn about all dynamic information updates occurring in the computing context when only a small subset might apply to them.

A standard approach for managing communication links is for low-level network software to produce periodic ping messages. When these messages have not been detected for some period, the network software informs applications that connectivity is lost. However, this approach is less than ideal since false disconnection reports are likely for wireless computers, due to intermittent loss of communication signal. In addition, switching between communication media with different characteristics, such as wireless and cabled, makes fixed-length timeouts impractical.

A different approach is for application software to set limits on the timeliness requirements of high-level pieces of information. When these requirements are broken, applications are notified by the communication software. Generally it is a bad idea for application to specify timeouts since they have no notion about the underlying communication costs, however, in this case it is appropriate because the values reflect human time-scale requirements rather than a guess of an appropriate network round trip delay.

2.3.6 Application Adaptation

In addition to the information describing objects in a computing environment a context-aware system needs to define how applications adapt to the changes in the computing environment. Specifically, whether notification of environment dynamics should be made directly to applications or whether an abstraction, such as device interfaces, can be used to hide the actual system dynamics. In software systems, transparency is enabled by a virtual or generic interface, for example Unix Vnodes [22]. This approach aims to provide an abstract interface that hides low-level characteristics of physical devices. The advantage of such an approach is that it simplifies the application or system interaction. Because this approach has found popularity in systems design it is worthwhile to explore whether it is appropriate here.

In order to apply this abstraction technique to the domain of mobile distributed systems we need to first determine the details a virtual layer can hide. Defining such a layer is more difficult than for other domains since the interface must manage not only a device's characteristics, but also their absence and reappearance over time. For certain classes of services, building this abstraction is feasible and perhaps desirable. For example, a printing service can mask the absence of a printer by using job queuing.

Although adaption employing a virtual layer using queueing appears feasible for the printer example, there are a number of problems in the general case. First, consider context-aware interactions with devices and computers. One difficulty with providing a virtual interface across different kinds of devices is that the underlying device characteristics change over time. A device coming online may have a different resolution or printer page description language. For this reason it is necessary to specifying device functions in terms of high level goals instead of low-level functions, or to delay the conversion to device-specific formats until just before use. For example, the interface might re-invoke the application, or use conversion procedures defined as part of the virtual device abstraction.

Although the printer example above may be suited for a virtual interface because it can queue up work, many kinds of interactions are not suitable because they require immediate action. Since mobility causes devices to come and go, the changing availability of the interface would itself defeat attempts at transparency. Sending an urgent message is an example. The problem then is that applications have no control over when operations work or when they block. Applications have no opportunity to either compute or prompt for alternatives. The NFS interface is an example of this, it hangs when it is not able to reach a server.

More important, however, is the fact that context-aware computing does not just involve transparent access to devices. There are many domains, such as UI interactors, where adaption according to the context is desirable. These domains are not necessarily system services and so it is hard to imagine how a single system abstraction masking dynamics can apply. Indeed there might need to be dozens of different specialized interfaces. In addition, the concept of context-aware applications is relatively new and there is not yet sufficient experience to formalize what these abstraction layers might be without hindering future development.

An alternative to a virtual system interface is to expose applications to changing context information. One such approach is to employ an "open" system composed of dynamic collections of asynchronous, parallel, communicating agents [7]. The difference is that instead of defining individual task-specific interfaces for operations such as printing, a common data and communication mechanism is employed that provides the basis for a number of tasks. Whereas virtual device interfaces may be seen as a "high" level interface because they specify much application specific information, the coordination language is "low" level because it manages more general interaction details. The division of work between application and agents can be defined on a task by task basis. This is the general approach taken by the system design presented in this thesis. The advantage of this approach is also its disadvantage: the functionality is under-specified and left to higher level abstractions. However, if the data and communication model is oriented towards the common needs of higher level software then it provides a reasonable point for building interfaces.

2.4 Architectural Considerations

The architectural considerations include the following:

- Fault tolerance and an application model for handling the disconnection inherent in mobile distributed systems.
- Interoperability on a number of hardware platforms from very small handheld and embedded devices to large mainframes.
- Scalability to large multi-building sites and possibly all devices connected by the Internet. Context-aware mobile computing should work with large as well as small changes in location.

Fault tolerance and gracefully degraded operation when disconnected are desirable because network disconnection may be common on wireless hosts. One possibility is to expose network connectivity to applications the same as many RPC systems do [9]. Applications would then be responsible for connection recovery. Alternatively system software could timeout and automatically re-connect to servers without involving applications. However, if servers are restarted then this would require recreating server state. In addition, this degree of transparency is not desirable if it makes applications behave as if no context changes are occurring, since this may not be the case. We propose exposing only changing state pre-specified by applications, thereby providing the right amount of information without overwhelming applications under conditions of intermittent connectivity.

Dynamic environment servers are back-end processes and as such their design can assume an execution platform having workstation-level resources. Dynamic environment clients, however, must be much less demanding of hardware resources. Clients may execute on palmtop, laptop and other small devices. This observation constrains the design-space for clients because certain approaches become unrealistic. For example, requiring that clients manage a sophisticated database is not feasible.

The number of clients for different changing data items places limits on information producers, such as the number of updates they can send per unit time. A one-server approach has scale, accessibility, and privacy problems. A completely distributed solution where information producers each disseminate information must be carefully designed or it will be prone to communication overload since clients must filter out unwanted information. These are the considerations addressed in Chapters 4 and 5.

2.5 Summary

The basic problem considered in this dissertation is the design of a framework for applications to help them manage the dynamics present in mobile distributed computing systems. This chapter has presented the important aspects of mobile distributed systems as they effect this goal.

The chapter began by presenting a system model for mobile distributed computing. A new model is necessary because people use computers in new ways: they perform traditional networked desktop computing intermixed with mobile computing. Mobile distributed computing is broadly defined to include use of wireless hosts, portable hosts temporarily connected to networks, and mobile users interacting with stationary and communication capable embedded devices.

The range of dynamics in mobile distributed systems was covered next. System changes are broken down into three categories: communication dynamics, environmental dynamics, and location dynamics. An understanding of the range and type of this information aids in design decisions concerning their computer representation.

The chapter continued with a more detailed presentation on the problems and issues associated with context-aware systems. This includes the context-aware cycle of discovery, selection, and use or adaption. The issue of information representation is covered next. Dynamic information includes simple values for things like communication parameters, user customizations, as well as more sophisticated information about mobile hosts and other located-objects. The section also compared polling versus callback as methods for data dissemination and examines the practicality of application transparency to system dynamics.

This chapter concluded with architectural considerations. A context-aware architecture must provide fault tolerance and a way to handle frequent disconnection inherent in mobile computing. It should provide interoperability on a number of hardware platforms: small handheld computers, embedded devices, and workstations. It should also provide scalability to large multi-building sites.

Chapter 3

Related Work

The design of a framework that allows applications to exploit a rich collection of information about their context of use draws from several areas of research. First, similar to mobile computing, it attempts to manage variable communication and hardware characteristics. Second it attempts to augment a user's computing experience in the vision of "ubiquitous computing." Third it is related to location-based computing since location is a large part of a user's context. Last it is related to other information dissemination mechanisms like Linda and Field. This chapter describes work in these areas and compares their approaches to the problems of this thesis.

3.1 Mobile Computing

Connectivity, communication characteristics and the configuration of peripherals are examples of things that change much more frequently in mobile than in desktop systems. There are a number of proposals for managing these mobile system dynamics in particular problem domains. In this section we briefly describe solutions in the areas of mobile internetworking, remote paging, and user interfaces.

One goal that has motivated much of the work in mobile computing is the idea that applications run on mobile hosts should be the same as those run on desktop systems. In this view, mobility and its consequences should be made transparent to applications. For example, the Coda file system has been designed to support disconnected operation transparently using local caching [21].

Another example of how systems can support application transparency under mobile conditions is Columbia's mobile internetworking work. Mobile internetworking addresses the problem of providing network access to hosts whose physical location changes over time. Since a route to a mobile host can not be directly deduced from the network address, traditional network routing is insufficient. The solution provided in [18] is to first create a "virtual mobile network," and second, to add special routers that gateway between the virtual network and the rest of the Internet. The new class of routers, called Mobile Support Routers (MSRs), form a distributed database for translating between virtual and physical addresses. This system exploits locality of host mobility to efficiently manage tracking and routing information. In addition it employs on-demand acquisition of mobile host location information to aide in scalability.

Virtual interfaces like those described above are a particularly good mechanism for hiding system details from applications. In cases it is also possible for services to automatically and transparently reconfigure in order to maintain a high level of service. One example is an algorithm for paging from a mobile computer into the memory of the "closest" paging servers [36]. One important aspect of this work is that client-server matchups are made dynamically and vary over time. The adaption is controlled by the client measuring communication latency for multiple paging servers and gradually shifting to employ the most efficient ones.

The previous examples illustrate the popularity of using a virtual system paradigm for mobile computing problems. However, these single-domain solutions are only partially related to our work. The reason is that since they can take advantage of key features of a problem domain they may be able to maintain transparency. However, a general application framework must be as application-independent as possible. In general, there is no reason to believe that the trade-offs made designing existing interfaces for a mostly static desktop environment still apply for a dynamic mobile computing environment. Indeed, a major point of this thesis is that a class of new, uniquely-mobile context-aware application are possible if this barrier is ignored.

As the ideas in this thesis evolved, other people at PARC also recognized the advantage of exposing contextual information. The *intelligently autonomous* programming model by Tso and Goldberg [15, 42] is an approach to designing mobile computer applications that are customized for an intermittently connected environment. Commonly, when unplanned network disconnections occur the user is hung because a required file is not accessible. To accomplish a more user-friendly interaction the model gives users control over a local file cache. As a demonstration, a mail application was modified to include new calls that interact with the cache: there are calls to find out what is in the cache; to add a file to the cache; and to register a callback when a file is removed. The mail program provided new interfaces such as using lightly-grayed menu items for files not in the cache, and darkly-grayed items when the network is down. When the user asks for a new file to be added to the cache, the program used a pop-up dialog box asking for suggestions on which files to remove.

The intelligently autonomous model can be considered a sub-domain of context-aware computing involving information about communication connectivity and files. One revealing difference is that Tso's model is limited to information about files while the framework presented in chapter 4 manages "objects." Another major difference is that the architecture presented in chapter 5 supports communication to software agents executing beyond the mobile host, in order, for example, to find out about nearby people and devices. This external communication raises a number of problems that are addressed such as fault tolerant and efficient dissemination of updates. The main point in common between these two systems is the notion that enabling applications to exploit the dynamics in a mobile distributed system is desirable.

3.2 Ubiquitous Computing

Ubiquitous computing is the idea that invisible computation everywhere can enhance life in the real world [46]. The Computer Science Laboratory (CSL) at Xerox PARC has established a number of research projects to explore this vision. This section briefly surveys some of these projects.

The PARCTAB system [34, 45] is based on palm-sized wireless PARCTAB computers (known generically as "tabs") and an infrared communication system that links them to each other and to desktop computers through a local area network. The tab infrared network [2] consists of cells defined by the walls of a room surrounding an IR transceiver. These small cells, referred to as nanocells, enable the system to pin down a user's location to the resolution of a room. PARCTAB applications can only access the tab's location while they are actively being used. This is because location identifiers (i.e., room names) are a part of each user interaction event.

The PARCTAB's simple form of location information enabled a number of location-based applications. However, the system had a number of shortcomings that made it impossible to add more sophisticated context-aware capabilities to applications. First, since locations were associated with user interactions, only one application at a time got informed of location changes. Second, there was no easy way to find out about other people, PARCTABs, or devices nearby. Finally, the information about locations is limited. There is no easy way to find the relationship or even distance between two locations. This thesis complements the work of the PARCTAB system by providing solutions to these problems.

Spreitzer and Theimer describe a location-based application called "ubiquitous message delivery" [39]. This application delivers a message to a mobile user by whatever means are possible. It may employ nearby displays, wireless palmtops, workstations, etc. The decision on how and when the message should be presented to the user may depend on contextual information. The program may consider the location of the recipient, identity of sender, importance of the message, contents of the message, nearby display terminals, others in the vicinity.

The ubiquitous message delivery program is the type of specialized location-based application that the architecture in this thesis aims to support. Their work is mostly concerned with privacy and security issues of location information and as such is complementary to our work. This thesis explores a number of new areas. First, the communication mechanism developed for this thesis is scalable and fault tolerant. In particular, the update dissemination technique presented in chapter 4 supports applications where dense populations of users share limited bandwidth. Also, as described in chapter 4, applications manage high level "timeliness" constraints on objects which greatly simplifies the view applications on intermittently connected hosts see. Finally, the use of an active map service for managing context information presented in Chapter 5 provides a set of operations including discovering geographic relations of containment and path distances.

3.3 Location-based Computing

Early work on location-based applications was undertaken by Olivetti Research Lab (ORL) [44]. This research focused for the most part on the hardware design and implementation of infrared beaconing badges (called *active badges*) worn by individuals, and networks of infrared receivers. Unique badge identifiers sent to the stationary receivers provide location information to a software system. The main software application is an "aid for a telephone receptionist" showing a table of names alongside a dynamically updating location and telephone extension. The system also provides a limited set of commands such as showing which badge wearers are in the same room. Staff wearing badges can have telephone calls directed to their current location. The original ORL system did not take context into account. Badge wearers expressed a desire to control call forwarding using context information: who they are with, where they are, and the time of day. "Personal control scripts" were added to a later version of the system to address this problem [43].

Similarly, work by the author explored the use of Active Badges for triggering the execution of Unix commands according to the location of badge wearers, and whether they were entering, leaving, or settled into a particular location. Commands were invoked with knowledge about the nearest workstation display and the badge sighting location [35]. A similar program was built at the Rank Xerox Research Centre. These programs can be viewed as investigating a specific application domain of the more general architecture presented in this thesis.

There has been some more general work on location-based systems, including [16, 44]. The system concurrently developed by Harter at ORL [16] uses a subscription based location service similar to that described in Chapter 4. The early Olivetti research provided inspiration for our current line of research. However, the architecture presented in this thesis is driven by more general configurations and contextual rather than location-specific applications classes. So, for example, the design presented supports multiple location sources, intermittently connected mobile hosts, and containment and path information that describe relationships between objects. Our communication mechanism handles scaling and overload conditions for systems covering "medium-sized" regions, such as buildings and small campuses, whereas Harter's system uses simple unicast remote procedure call.

3.4 Information Dissemination

Linda [5, 6, 7] is a communication mechanism developed by David Gelernter and Nicholas Carriero at Yale University. Linda defines a process creation and coordination abstraction in which processes communicate through a shared data space called a *tuple space*. In Linda, information is exchanged in the form of persistent tuples as opposed to ephemeral messages. A tuple space is a form of associative memory where tuples are identified by matching on a key.

Gelernter refers to Linda as an example of a *coordination language*. The term is given to communication mechanisms that tie together distributed applications. A distributed programming language is said to consist of a computation language plus a coordination language. This separation fosters portability, heterogeneity, and economy in that the same mechanism can be used for interaction among multi-processors for fine grain parallelism and among distributed-processes for course grain parallelism. Linda has been implemented for various languages, including C, Lisp, Postscript, Modula-2, and even Fortran [7].

A tuple space is a single shared communication channel that does not scale well. The approach proposed in [26] is to segment a tuple space into multiple, recursive, tuple spaces where each is used for communication between coordinating processes. However this partitioning adds some complexity requiring a somewhat different approach to building small and large systems.

Linda and other coordination languages provided a starting point for the communication design presented here. However our problem domain is somewhat more restrictive in that we are concerned with updating multiple clients and can therefore more easily take advantage of various multicast dissemination mechanisms. Linda-based systems also are not designed to manage hosts over slow links, and are not concerned with the problems of fault tolerance in a partially connected network.

An alternative approach to subscription-callback is that of an “information bus” where all changes are broadcast to all applications who then filter information they are interested in [29, 32, 41]. This may seem more desirable because it requires no subscription specification by the client and hence a simpler interface. However, it is unsuitable for systems with applications running on mobile hosts because it causes unnecessary network traffic.

Chapter 4

Dynamic Environments

In order for applications to adapt preferences and configurations according to their context, they must first have a way to access pertinent contextual information. This chapter introduces the Dynamic Environment (DE) model, the common base mechanism used for triggering customizations and configuration changes in context-aware applications. The DE model defines an efficient communication mechanism and an application interface. The DE interface is employed by a collection of services described in chapter 5.

4.1 Overview

Often, information about the execution context, user preferences, and system configuration are expressed at *initialization time* when a program starts running. For example, Unix software employs a few different facilities for customization:

- command line arguments
- environment variables
- “rc” initialization files
- databases like Xrdb [1]

[1] Xrdb, the X11 resource manager, is the part of the X11 window system that manages user preferences for colors, fonts, etc. Each X11 screen has an associated Xrdb server process.

Recent computing history can be viewed as consisting of three eras: the period of mainframes and dumb terminals; window-based workstations; and multi-host or mobile distributed systems. Each era poses different requirements on customization mechanisms. Customization by environment variables dates to the era of dumb terminals. In this situation a user has one screen and runs one process at a time. Unix environment variables are a representative customization mechanism. Environment variables get inherited from parent process to child, which suffices because all work is carried out in a sub-process of an interactive shell process. This mechanism has survived into the windowing-workstation era, even though it has shortcomings. One commonly encountered problem is that an environment variable set in one shell window is not propagated to other windows. So, for example, changing the PRINTER[2] variable in one window can lead to confusion when the print command is issued in a different window.

The second computing era brought forth networked desktop workstations running windowing software such as X11 or Andrew. In these systems programs may be executing on different hosts. This practice makes environment variables inappropriate since there is no inheritance of variables across hosts, and so a database associated with each display is used.

The third era of computing is characterized by users interacting with a multitude of stationary and mobile systems through the course of a day. A person might use their office workstation, then have a meeting where they use a whiteboard-sized display, work on a wireless PDA or laptop during lunch, and then return to a workstation at their office or home. Mobile distributed systems of this sort need to maintain and share a context that is not limited to the boundary defined by a process hierarchy, display manager, or host.

In addition, applications must separate customization from initialization if they intend to support customization changes during program execution. Historically most applications have run from start to stop in a short time frame. Today, however, people run applications, such as mail and emacs, that persist over very long time periods. People evolve a context of windows, editor buffers, command history, etc., which is expensive to recreate. As a consequence, mobile distributed computing requires quick instantiation of a context

[2]This determines the printer to which output is sent.

as well as persistence across sessions. For example, a context that follows the user is useful for migrating application windows, where windows move from one display to another following a mobile user.

Unix applications provide an approach to separating initialization from configuration. Non-interactive servers, by convention, re-read configuration files when they receive the HUP signal; X11 window managers (like twm) support the same facility through a menu command. In both cases, reconfiguration requires manual intervention and is often heavy-handed, with the entire internal state being destroyed and recreated. There are two problems with this approach. First, there is no facility for incremental changes and therefore re-initialization resets all state where perhaps only some should change, and second, it is based on configuration files that need to be changed, and hence are constantly monitored by interested parties.

The X11 windowing system provides a server-based resource manager, Xrdb [33], for communicating user preferences to applications. The advantage of Xrdb is that clients accessing a central server do not need a duplicate (or common) collection of initialization files on all hosts. Although the database-server approach has the advantage that it allows changing parameters without editing files on disk, it does not notify existing applications of those changes. One could imagine extending the notion of X-events or Xrdb to solve these problems. A new type of X-event, for example, could indicate what environment information has changed. However, this solution implies that applications would require X11 servers on any hosts on which they are run.

4.1.1 Approach

Rather than focus on a limited solution, we present a general purpose approach. Essentially our design employs an extension and elaboration of the abstraction known as callback or "upcall" messages in which asynchronous messages or remote procedure calls are sent by servers to clients [8]. A key point is that this interface provides a uniform way to "see" the environment changes occurring in a mobile distributed system. It also provides a uniform fault model. It can be used between applications and remote services, or user and system software.

The Dynamic Environment Interface communicates environment changes using a data-oriented model. Clients of the interface update information about objects and/or submit queries to obtain information about other published objects. Queries match against the attribute keys and values of objects. Standing queries, called *subscriptions,* can also be specified; the interface has a method to send changed information that matches a subscription as it becomes available.

The Dynamic Environment Interface represents a feature of the mobile computing environment, such as a person, printer, or network interface, as a *dynamic environment object* or simply an *object*. Objects are grouped into *dynamic environment sets* managed by server processes. Clients learn about their execution environment by reading objects from servers. When changes to the execution environment occur, server processes monitoring the environment reflect these changes in the objects they publish. Clients interested in this information notice these changes by either re-reading environment objects or by subscribing and receiving asynchronous change notifications.

Dynamic environments have a number of properties that make them desirable for context-aware computing:

- Dynamic environment objects are persistent. Since the entities they model exist over time, it is natural that objects themselves persist.

- Changes to data objects along with change-notification are actively propagated so programs observe and react to environment changes more quickly than in a polling approach.

- Connectivity information associated with objects tells programs whether the information is up to date and accessible. This information, for example, lets applications provide suitable user interface feedback for changes in connectivity.

- The application interface automatically handles server restarts by noticing when servers go away, and reconnecting and initializing them with subscriptions when they return online.

It should be noted that other design choices exist for communicating environment information to applications, most notably those based on a procedural model, e.g., Remote

Procedure Call (RPC). The advantage of a declarative data-oriented approach is that it naturally models the computing environment – entities in the real world can be represented as data objects in the model. Furthermore, the data-centric approach provides straight-forward caching that permits a degree of support for frequently disconnected systems. In addition, the use of data objects allows persistence so that reading values and writing values do not need to occur at the same time. In contrast, in the case of RPC, the entities must both be active and in communication at a common point in time.

Although the declarative approach has been sufficient for purposes explored in this thesis, procedural models that incorporate transactions are also a design choice. Fortunately, we haven't found a need for supporting several distributed data objects that must change state together atomically. So it suffices that the delivery of notifications from a single dynamic environment set are serialized with no general ordering guarantees among objects in different sets. If necessary, special case ordering can be achieved by applications. For example, when objects are produced by the same source, e.g., a location sensor, they can be time-stamped. This restriction reflects our belief that subscriptions generally refer to individual environments and when multiple subscriptions are made, for example, sets representing multiple buildings, there is only a rare causality among events.

It is worth pointing out that the proposed approach cannot support, for example, physical world sensors at the same location managed by different environment servers. In such a case applications may briefly show contradictory information. For those cases where atomicity is important, it is assumed that the synchronized data can be managed by a single server, whether or not the back end is distributed. In particular special cases should be handled by processes incorporating domain-specific knowledge that stabilizes and provides a hysteresis on multi-source inputs before producing a single computed output. In chapter 6 is a description of how combining multiple, possibly contradictory, sources of a person's location is solved in a similar manner. These assumptions have allowed us to ignore transactions and other similar models.

4.1.2 Design Issues

The issues this design addresses are scaling, fault tolerance, application and communication overload.

In order to handle the problem of scaling we partition the information into individual *dynamic environment servers*. Each server manages a set of objects (an *environment*) and delivers updates to clients that have previously shown interest by *subscribing* to the server. This tradeoff, compared to systems like Linda [6], which has a single global data space, requires that clients know about the “structure” of the dynamic environment universe. That is, clients must know which servers contain the information they are interested in.

Any number of servers are used to organize the information in a system based on dynamic environments. For example, in a context-aware architecture it is useful to have one environment for each user, plus environments for rooms, work groups, etc. Clients use network messages to get objects from servers, and servers use messages to provide callbacks to clients. The key feature of an environment server is that environment changes are quickly seen by multiple processes through the use of network callback messages. As such the mechanism can be considered an efficient information dissemination technique. The application programming interface does not define how servers obtain this information, for example by reading sensors, or collecting data from the network, but rather that is left to the individual derivative servers that employ dynamic environments.

Because distributed servers are involved the system must address the challenges of fault tolerance in the face of network partition. In order to handle this, two techniques are used. First, if connections to servers go away, a clients library automatically attempts to recreate connections and regenerate the shared state that exists between the two. Second, because hosts may be intermittently connected, clients specify how far out of date objects may become and are then informed when these guarantees cannot be maintained.

Network information overload is handled by three techniques. First, clients can access information using a query interface synchronously, and obtain a specified amount of information only when they ask for it. Second, clients are able to apply filters on updates limiting the kind of information they will receive from a dynamic environment. Third, clients can specify how much information they are willing to accept in a unit time. This is effectively a bandwidth limit.

The final issue addressed is how to efficiently disseminate update information without overloading servers, clients, or the intervening networks. Dynamic environments use multicast communication to transmit information to clients. Employing a number of multicast groups gives the server flexibility in assigning the distribution of load among server, client and the communication network.

The remainder of this chapter expands on the dynamic environment communication model. First, the data representation used by environment servers is described. Following this is a description of the application programmer's interface. The chapter concludes with a technique for efficiently disseminating subscription-based information.

4.2 Data Model

We use the term *dynamic environment object* (or simply *object*) to mean a container that stores information. Objects may store information about entities in the real world: people, computers, printers, rooms, etc. An object might also be used to represent non-physical things such as network bandwidth, temperature, light-level, user preferences, and service registrations.

The term "object" should not be confused with the concept of an object in an object oriented programming language. A dynamic environment object does not, for example, include a notion of methods and inheritance. The term "object" is used simply as a non-specified collection of data. Dynamic objects differ from "records" and other data containers because, as described below, their contents are self-describing, and also because there is associated "meta" information that helps clients to predict the timeliness of an object's content.

4.2.1 Data Types

The data model employed by dynamic environments is based on a minimal collection of primitive types. To represent objects there is one basic type and one structuring type. The basic type is the universally understood text, and the structure of the data is defined by

```
{Name Badge:802}
{PublicKey "R0lGODdhDAMdAc}Yd"}
{With {schilit spreitzer theimer}}
```

Figure 4.1: Key-Value Pairs

lists of text as well as the list type itself. Note that any information can be represented in this textual form with some suitable encoding. In the print-form of Figure 4.1, braces are used to show the list structure and text literals do not need to be quoted unless they include spaces or braces.

4.2.2 Self-Describing Representation

Using these types, objects are defined to be self-describing data-structures. That is, the structure as well as the value are always included in a object's content. There are two advantages to this approach. First it avoids the complexity of separate schemas and of maintaining these across multiple distributed servers. Second, it permits meta-tools, such as browsers, to manipulate the database without knowledge of the specific contents.

The format for the self-describing objects is based on keys and values, collectively called an *attribute*. This is shown in Figure 4.1. Keys are simple text types but values can be either text or lists. Figure 4.2 show some example objects.

4.2.3 Attribute Conventions

Although the use of self-describing information simplifies certain design aspects, it also creates problems. Because the data format employs an implicit, integrated schema rather than one that is explicit and separate, the usefulness of this representation depends on the extent that dynamic environments and their clients agree upon and follow conventions in the use of attributes. Some conventions, for example, the required attributes for a printer, depend on the high-level use of a printer object and are defined by software above

```
{{Name User:ragnap}
 {username "Ryle R. Ragnap"}
 {office LID:35-2-1-56}
 {projects {tab context-aware}}
 {Agent sunrpc_2_0x30000cad_5|tcp_13.2.116.17_3785}}

{{Name Printer:snoball}
 {Location LID:35-2-1-06}
 {format {ip ps text}}
 {features {duplex staple highlight}}}

{{Name Host:fermius}
 {Location LID:35-2-1-01}
 {addr 13.2.116.17}
 {Agent sunrpc_2_0x30000cad_5|tcp_13.2.116.17_378}
 {availability idle}}

{{Name Camera:2230a}
 {Location LID:35-2-2-30}
 {availability inuse}
 {Agent sunrpc_2_0x30000cad_5|tcp_13.2.116.17_378}}
```

Figure 4.2: Dynamic Environment Objects

the base-level DE servers. However, there are a number of issues at the level of dynamic environments: names, attribute conflicts, and object references.

Dynamic Environments place no restriction on objects except that they have a unique Name attribute. By convention, object names consist of a naming domain (a “type”), followed by a colon and an instance name. For example Tab:0.0.1 and User:schilit. For our system there are a small number of domains and we use a central registry, e.g., the way the NIC administers Internet domains, to avoid conflicts in assigning names. Different software components are responsible for specifying the keys and values of attributes within a naming domain.

Another convention to help applications manage objects is that certain keys within objects are *global attributes* and have the same meaning across multiple objects. For example, the key for the object’s current location, Location, is an attribute of both device and people objects. Global attributes, like naming domains, are recorded to avoid conflict. In addition, since new global attributes may be created and inadvertently override existing attributes, keys starting with an upper case letter are reserved for globals attributes and lower case must be used for all other keys.

Finally, there is a convention for referencing one object inside another. The value is simply the full object name:

```
{Location LID:35-2-2-00}
{key-operators {User:dino User:ron}}
```

4.3 Application Programming Interface

A collection of dynamic environment objects is called a *dynamic environment set* or simply an *environment*. This is the grouping of information which is exported over the dynamic environment interface by servers and imported by clients. Partitioning information in this way satisfies a need for a natural grouping as well as scaling of the system since each group can be maintained by separate server processes.

Within a dynamic environment two objects with the same name are identical and there is no particular ordering among objects. The Name is the only attribute that must be de_discover(description, environment)

Search for a dynamic environment matching the description and associate it with the environment handle. All future calls refer to the discovered environment.

Table 4.1: Binding Interface

present in an object. Aside from this the interface places no restrictions on the content of objects in an environment.

The rest of this section describes the dynamic environment programming interface in more detail. This interface can be used by components within the same address space, in different address spaces on the same machine, and across address spaces on different machines. This application interface defines what the computing environment looks like and how clients are kept aware of changes; however, it does not say how it would be implemented.

4.3.1 Binding

In general, dynamic environment servers are located by a naming service called Discover. Discover registers servers with a description of the objects they export and the binding information needed to create a network connection.

The Dynamic Environment Interface provides a call to allow clients to lookup registered environments (see Table 4.1). Discovering a dynamic environment is similar to binding to an RPC server. Servers export environments and clients bind to them before they can be used. The purpose of environment descriptions is to mask from clients, as best as possible, the fact that the source of environment objects are potentially faulty server processes that may move from one processor to another to increase availability. Clients must know about dynamic objects, however, they do not need to keep track of other issues such as network connectivity and host status. Section 4.3.4 describes the use of environment descriptions.

Note that although the contents of a dynamic environment server may change fairly de_query(environment, pattern, maximum, options) returns(result)

> *Queries the environment set and returns the collection of objects, up to the maximum specified, matching the pattern. The* options *field is used to specify how the server should interpret the query and return results. The base level server supports the* update_limit *option, which is a data-rate threshold, in bytes/second, on the objects returned.*

de_subscribe(environment, pattern, options, fcn, arg)

> *Queries the environment set using the filter pattern and calls user-supplied procedure* fcn *to report the current objects and future changes.*

de_unsubscribe(environment, pattern, fcn, arg)

> *Delete an existing subscription.*

fcn(environment, pattern, options, arg, backlog, object)

> *The format for a user function invoked as a callback during subscription updates. The first four parameters come from the* de_subscribe *call. The* backlog *indicates the number of updates that have not yet been processed, and* object *is the object.*

Table 4.2: Query and Subscribe Interface

frequently, the registrations for the servers do not. This means that a name lookup for a server registration is not a difficult problem and can use a number of existing naming services such as DNS [40] or NBP [4].

4.3.2 Queries

The dynamic environment interface provides two methods for retrieving objects from an environment: query and subscribe. These are shown in Table 4.2. Query synchronously returns objects and subscribe initiates a standing query that causes asynchronous delivery of objects. Both query and subscribe select a subset of objects in an environment through the use of a *query pattern*.

Query patterns permit simple matching against the attributes of a dynamic envi-

`*`	Match any (zero or more) characters.
`?`	Match any single character.
`[...]`	Match any single character in the enclosed list(s) or ranges(s). A list is a string of characters. A range is two characters separated by a minus sign (-), and includes all the characters in between in the ASCII collating sequence.
`{ str, ... }`	Match any single string (which may be a pattern) in the comma-separated list.

Table 4.3: Query/Subscription Pattern Matching Language

ronment object. The query language can select objects having any number of specific attributes, and it can select objects whose attributes have particular values. However, it cannot select objects according to complicated expressions. In this case client processing is necessary. For a class of interesting applications this query language is suitable. Future work might examine more powerful query languages.

Patterns are applied to sets of environment objects to select a matching subset. In a similar manner as self-describing objects, a query is composed of text and list types. For patterns, the value part of the key-value pairs can be an expression that matches values using wild cards. This matching form is based on the Unix shell matching operation and is selected over other regular expression matchers for it's ease of use and execution speed. The format for the value expression in a pattern is summarized Table 4.3.

Example query patterns are shown in Figure 4.3. The first example returns objects having a "room" value starting with the prefix "LID:35-2-" (i.e., building 35, on the second floor) and an audio control attribute. The next returns any objects having a "badge" attribute with value "802" or "804"; the last returns any objects having a "printer" attribute that also have a "model" equal to "lw."

Although query patterns can be implemented at either the client or server, one of their benefits comes when they are used to reduce network traffic emanating from the server, and the related filtering occurring at the client. An implementation that balances the load such that maximal server and minimal client filtering occurs will be to the best advantage

```
{{Room LID:35-2-*} {AudioControl *}}
{{ActiveBadge "{802,804}"}}
{{Printer *} {model lw}}
```

Figure 4.3: Example Queries

of mobile hosts having limited bandwidth networks.

4.3.3 Subscriptions

In order to avoid the problem of overloading clients and communication links with repeated polling queries, applications may subscribe to the information they are interested in and the implementation attempts to transfer only the desired information to the application host. This means that applications or application libraries must be able to accept asynchronous messages. There are a number of approaches to handle this even in applications that are solely single threaded – and we describe these in the implementation chapter.

The general idea behind a subscription is that the clients specify ahead of time the information they are interested in. Subscriptions are then treated as a standing query. The query matches objects stored in an environment set and as the contents of the environment set changes, the new information is propagated to the subscribers. At any time clients may cancel existing subscriptions and/or invoke new ones.

For clients connected to low-bandwidth communication links or for hosts with limited computational resources available for “ancillary” applications the question of how to limit update traffic is extremely important. Furthermore, if a client waits until a communication channel is flooded with updates before taking any action, then it may be difficult or impossible to send a message adjusting to a “smaller” subscription. Unfortunately, it is not easy to infer the amount of traffic that any given DE subscription query will generate since circumstances can dynamically change.

Consequently, the DE servers lets clients specify their desired bandwidth limits as part of their subscriptions. The DE server guarantees never to send more traffic to a client than that client's bandwidth limit specifies, even if it means not sending all the update information that the client's subscription query has matched. In order to alert clients when they are missing update information because of bandwidth limitations an additional count field within the update message is used to indicate how many updates are pending. Clients receiving an update message with a non-zero count field know that they have received the latest available location information for any object that is included in the update message, but they may be unaware of the latest changes to any object *not* mentioned in the update message. It is up to each client to decide how to deal with partial update information.

4.3.4 Fault Tolerance

By their nature, mobile systems operate in an intermittently connected, faulty computing environment. Providing fault tolerance means something different for a context-aware computing system than the traditional effort to mask failures. This is because an application responding to changes may require exposure to all changes occuring in the computing environment, including how well communication channels are functioning. The approach taken is to selectively hide aspects of the dynamic environment: the mechanical aspects of failure detection and subsequent reconnection are hidden but the failure events are exposed. Fault tolerance at the dynamic environment level involves two mechanisms: server reconnection and latency feedback. These are described below.

Servers that crash, networks that partition, and other communication errors will cause clients to poll for live servers allowing them to eventually reestablish connectivity. Once the client reconnects it also automatically recreates the lost server state which consist of the client's standing queries. Aside from survivability after component failure, this mechanism permits "lazy" server relocation to a new host. This is similar to "automatic binding" in DCE [38]. Server reconnection is facilitated by two aspects of the architecture. First, binding specifications for environment servers are descriptions of their contents rather than their physical locations, and so can be used to locate servers wherever they are run. Second, the data-oriented interface makes it possible to easily recreate shared state between clients and servers. The state consists of subscriptions provided by clients and environment objects from servers.

Specifically, before applications can access an environment they present a pattern to the discovery procedure. This procedure searches the local host and network for a server exporting a dynamic environment matching the pattern and associates it with the client's dynamic environment handle. When a server is restarted, the discovery process is repeated and any standing subscriptions associated with the environment are re-registered.

The second mechanism providing fault tolerance is to expose the connectivity information in a meaningful manner. Since wireless hosts are commonly subject to short term disconnections it is not generally useful to notify applications of the frequent low level communication problems.

For managing the intermittently connected network that dynamic environment clients might encounter, *latency feedback* is used. This approach avoids a strict distinction between online and offline in favor of a time value representing how potentially far out of date some environment object may have become. It lets applications specify the high-level requirements related to the timeliness of specific pieces of information. When the communications layer is receiving messages from servers this value is low, and when no messages are being exchanged, the value increases.

Latency feedback operates as follows: each object has a "meta" state. This state consists of the number of seconds an object may have become out of date as computed by the communications layer, and also an application requirement on how up to date the object must be. The communication layer monitors servers and notes when the latency goes above the application settable threshold, e.g., the network becomes unavailable, and provides this information to applications in a similar manner as value changes.

This technique has the additional benefit that it lets clients efficiently monitor connectivity. The communication layer can schedule when to send liveness checks to servers based on information about all specified latencies. Note the fact that clients associate latency callbacks with server objects does not mean that this information is sent to the individual environment servers. Rather, the client-side communication library maintains the information necessary to invoke application callback functions according to an environment's communication latency. The interface is shown in Table 4.4. The procedure allows setting a callback to occur when the latency crosses the user threshold. So, for example, if a communication de_set_latency(environment,object_name,latency,fcn,arg)

> *When the object* name *from the specified environment might be more than* latency *seconds out of date, then invoke the callback function.*

de_get_latency(environment,object_name) returns(latency,fcn)

> *Return the previously set latency and callback function for an object.*

fcn(environment,object_name,latency,arg,offline)

> *This is the form of the callback function. The parameters are as specified in* de_set_latency *except for* offline*, which is a number less than or greater than 0 according to whether the object latency time is greater than or has returned to less than the pre-specified limit.*

Table 4.4: Value Latency Interface

black-out occurs, the user function is called once latency gets larger and once when again after connectivity is established and the latency goes back to normal.

4.4 Information Dissemination

Dynamic environment communication consists of client initiated queries and subscriptions and server initiated update messages. This section describes how servers disseminate subscription-update messages to clients. This design considers the problem of balancing server, client and network loads. We first explore a unicast-based approach that tends to load the server and network. Broadcast messaging offloads the server, but increases loads on client's communication links. Finally, we propose an approach based on multiple multicast groups that gives the server control to balance the load as necessary.

The simplest implementation of a dynamic environment server involves using reliable unicast communications, such as remote procedure call (RPC), between the DE server and its clients. In most circumstances, RPC is a satisfactory communications paradigm to use. The reason is that, for the most part, changes to physical objects happen infrequently compared to the timescales appropriate for computer communication. This is especially true if clients and DE servers are either directly connected to high-speed networks or are not sharing their slow-speed communication link with many others. As an example, consider location information for people. There are not many object updates because people are stationary most of the time. Furthermore, most of the time people occupy separate offices or cubicles and so any wirelessly connected computers they have are likely to occupy different communication cells in isolation. Of course, there are significant numbers of workplaces for which these assumptions of mostly stationary, mostly separated, people do not hold. Similarly, not all wireless technologies employ room-sized cells for their communications.

In a unicast design, brief periods of increased load can be handled by simply queuing DE update messages for later transmission and by batching together multiple updates. Two benefits are gained this way: (1) the packet overhead of an update message can be amortized over more update items, and (2) if the interval between updates spans several sightings of the same moving object then fewer update items can be sent by only including the latest sighting for the object.

The disadvantages of this approach are that subscribers obtain their information in a less timely fashion and that they obtain less accurate information about the activity around them. This is a problem with location information, for example, because a client may find out too late or not at all that something the client wanted to know about has passed nearby. Discarding updates also requires some additional message processing by the DE server. In any case, this approach to controlling overload works only if the overload lasts for a brief period of time. We describe more suitable alternatives in the remainder of this section.

4.4.1 A Reliable Multicast Design

Much of the DE server load generated during overload situations is due to sending the same update message, over and over again, to many different subscribers. This duplication can be eliminated by employing multicast to distribute the update message to all interested subscribers at the same time.

The data structure needed inside the DE server to provide reliable communications to a particular multicast group is called a *multicast channel*. Multicast channels employ a standard negative-acknowledgment protocol. To support this, each multicast message contains a sequence number. Receivers check to make sure they see each sequence number in turn. If a missing message is discovered, then the receiver sends a "heal" request specifying the missing sequence number. The sender then retransmits the requested message in a heal response. In order to limit the time it takes receivers to detect a missed message, whenever there is no normal message traffic for longer than a "synchronization interval," the sender transmits a "synchronization message." The synchronization interval is a settable server parameter. Under high load situations waiting for synchronization messages to detect misses rarely occurs since data messages perform the same function.

4.4.2 Broadcast Channel

The DE server sends out asynchronous updates, or callback messages, that contain information about the changing context. Using a single broadcast channel for all callback messages minimizes the DE server load. For example, the information about a located-object's change-of-state can be sent once only instead of once for each client whose subscription query matches the event. For network segments having multiple clients the load would also be minimized, since identical messages occur only once.

However, this reduction in server and network load comes in exchange for additional load placed on all clients and on some communication links. This is because all clients—and the communication links joining them to the DE server—now receive all updates the DE server sends out. For clients that are power conscious and/or attached to slow communications links (e.g., a palmtop connected over a 19200 bps infrared link) this solution is unacceptable.

4.4.3 Multiple Multicast Channels

Instead of indiscriminately sending update information to all clients multiple multicast channels can be used to send out information to selective groups of clients. This requires that the DE server be able to figure out how to assign some or all updates to one or more multicast channels to be used for dissemination.

Receiving updates in response to subscriptions is the common way clients interact with the DE server. To characterize this form of interaction more precisely, we define the DE server as a set of "publications": $P = \langle p_1, p_2, \ldots, p_m \rangle$. This is the set of environment objects for which the DE server maintains information. Any client, i, can specify interest in a dynamically changing subset of P's publications by submitting a subscription query, S_i. The set of all such subscription queries is denoted by S. All members of S are run against P whenever the state of any member of P changes. We assume that at any given time, t, at most one member of P will change its state[3]. If a member of P changes at time t then the set of subscription queries matching the object is denoted the *update query set*, $u_q(t)$. The associated set of clients who should receive notification of the change is denoted the *update client set*, $u_c(t)$.

Figure 4.4 shows an example of how servers manage the group membership at the client. Clients first send a subscription request to a dynamic environment server which responds with the matching objects and communication configuration commands. Communication configuration commands tell the client to join or leave specific multicast groups. After the subscription is made, objects are received from the server over the client's unicast address or one of the multicast groups it has been directed to join. Along with updates the DE server may send additional configuration commands indicating that application should join or leave a multicast group. In practice command message overhead is reduced by merging them with update messages. The server uses information about all subscribers to generate the assignment of groups for clients.

We define the general *multicast-channel assignment problem* as follows. Let $G = \langle g_1, g_2, \ldots, g_n \rangle$ be a collection of multicast channels. We desire to define a mapping, $\mathcal{M}_c(t) : u_c(t) \rightarrow G$, of $u_c(t)$ onto G, such that if the clients specified by $u_c(t)$ join the multicast channels specified by $\mathcal{M}_c(t)$ at time t and the DE server sends out the update information intended for $u_c(t)$ to the multicast channels specified by $\mathcal{M}_c(t)$ at time t then the following will hold:

- Every subscription client i will receive the update information it has specified with its query, S_i, and

[3]This assumption holds in practice because the DE server receives information about changing objects via RPC and processes each RPC in turn, but it is not true in general.

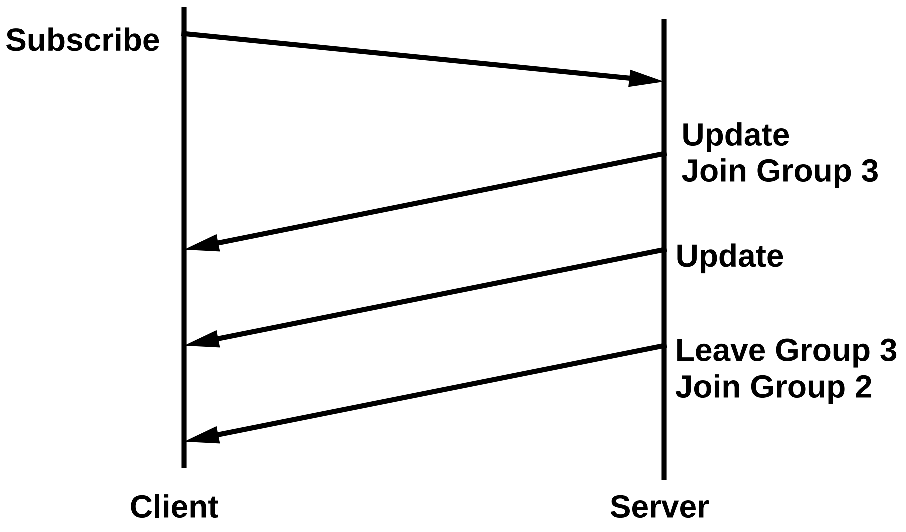


Figure 4.4: Assigning Multiple Multicast Groups

- The overall "system overhead" of disseminating update information will be minimized.

Unfortunately, when considered in its full generality, there are a variety of system overheads to consider when choosing $\mathcal{M}_c(t)$. Furthermore, these overheads are borne by different parts of the system and are only partly comparable to each other:

- The DE server bears the cost of computing $\mathcal{M}_c(t)$, sending updates about changes to P, and disseminating $\mathcal{M}_c(t)$.

- Clients bear the cost of tracking changes to $\mathcal{M}_c(t)$, as well as filtering unwanted updates.

- The DE server sending update messages to multiple channels, as defined by the mapping $\mathcal{M}_c(t)$, puts additional load on communication links and the server.

- Notification time is affected by the use of extra messages. If the DE server sends multiple messages for the same update information then that will introduce a delay for all but the first group of recipients. This is a cost borne by the clients.

Given these different cost metrics it is difficult to define what minimum system overhead means. Worse yet, even if the problem is restricted by choosing a particular cost metric, it will likely not be easily solvable in general. For example, choosing to minimize server and communication costs by requiring that every update be sent using exactly one message. That is, $\mathcal{M}_c(t)$ would be restricted so that every update client set, $u_c(t)$, is mapped to exactly one member of G. In this case each update message is sent out just once, so server and network load are constant. The work of minimizing system overhead would involve choosing $\mathcal{M}_c(t)$, so that the message filtering overhead that client subscribers experience due to receiving unwanted update messages is minimized. Unfortunately, this problem contains a solution space that is exponential in the number of publications, subscribers, or multicast channels involved. For example, assigning clients into groups, so that for any possible update there exists a group with no extraneous clients, requires a quantity of channels exponential in the number of subscribers. Similarly just searching through the potential mappings is exponential.

4.4.4 Detecting Multiply-Subscribed Queries

Because the general multicast-channel assignment problem is too difficult we must resort to a special-case approach for solving a limited form of the problem. Towards this end we make a number of assumptions about how a DE server is used. These assumptions are derived from experience with location-based applications but they may apply to other domains as well.

First we note that the location-oriented DE server load is determined largely by queries that are issued by large numbers of clients as well as by queries monitoring locations with frequently changing contents, for example, meeting rooms and hallways. In contrast to these “hot spots” of activity, monitoring a single located-object as it moves from one place to another in a human time-frame produces only light message traffic. Our second observation is that, in most cases, many clients will specify the same or very similar subscription queries for a particular meeting or for their regional and object-specific subscription queries. We hypothesize that this is true in general.

The DE server can exploit these situations by recognizing when multiple clients are specifying the same subscription query and employing a multicast channel to service the update traffic for that query. This approach imposes no additional filtering overhead on clients and reduces server and communication system load to the extent that clients can be encouraged to employ the same subscription queries. In particular, if many clients specify the same query for their subscription to a particular location then meetings can be handled efficiently because the DE server can assign a single multicast channel to that query and can use it to send a single update message to all the clients who submitted that query. Similarly, if many clients specify the same regional subscription query then the DE server can service those subscribers via a single multicast channel that is dedicated to them.

A more detailed description of how this approach works is given next. First observe that a slightly different mapping than the one already defined is required. Specifically, we define $\mathcal{M}_q(t) : u_q(t) \to G$, which maps update query sets to multicast channels (instead of the associated update client sets). This is so that clients of multiply-subscribed queries can be "combined" into one multicast channel. We define $\mathcal{M}_q(t)$ by defining an equality relationship for subscription queries, such as string equality for the (ASCII) source representation of queries, and then defining $\mathcal{M}_q$ such that at all times, t, equal subscriptions map to the same member of G and unequal subscriptions map to different members of G. The size of G is assumed to be sufficiently large to allow this.

The DE server computes $u_q(t)$ by maintaining a count of the number of current subscribers who have submitted any given query and assigns multicast channels to those with more than some pre-specified minimum number q, of subscribers[4]. In particular, the DE server maintains a record for each different query corresponding to a currently submitted client subscription[5]. This record contains a count field, a multicast channel identifier, and a list of all clients that have submitted this query. A unique unicast callback address is also maintained for each client. The count corresponds to the number of clients who are currently interested in that particular subscription query. If the count for a query is equal to or above q then the multicast channel identifier is used to send update information to all the relevant clients. If the count is less than q then update information is disseminated using the

[4]This number is small and depends on the implementation costs of switching to multicast channels versus continuing to employ unicast messaging.

[5]Clients may withdraw subscriptions. Subscriptions are also garbage collected when their submitting clients are detected to have gone away.

list of client callback addresses attached to the record. When the DE server receives a new subscription request that contains the q-th instance of a query it assigns a multicast channel to the query, records the channel's identifier, and notifies all clients subscribing to the query to join the new multicast channel for update information. It does so using the list of unicast callback addresses attached to the query record.

Detecting multiply-subscribed queries will only work well if clients actually specify the same queries. To encourage this applications can define a "standard" set of query templates that are parameterizable by a standard set of attribute types. When two clients substitute the same parameters, for example because they are at the same location, then identical queries will result. This provides a fairly flexible set of queries that can still be easily recognized and mapped to appropriate multicast update channels by the DE server.

4.4.5 Detecting Recurring Update Query Sets

The previous section described how updates to clients subscribing to the same queries can be disseminated with a common multicast channel. This section presents the idea that when different queries result in updates to the same set of clients, they can also share a multicast channel. For this to happen, we must recognize recurring update query sets, $u_q(t)$; that is, sets of non-identical subscription queries that repeat across a number of update events. This will occur in situations where different groups of clients want different, but possibly overlapping, "cuts" of a large amount of location information that is available for a particular location or region. As an example, consider a large multi-organization conference that is taking place. Many members of each organization may want to see only the location information pertaining to their own organization and, perhaps, a few "sister" organizations, rather than information about all attendees of the conference. This will be especially true for clients sitting on small hosts connected via slow communications links. Regional queries can be similarly specialized.

A side benefit of recognizing recurring update sets is that it allows the DE server to use a simpler form of query equality for recognizing multiply-subscribed individual queries. Two queries whose source form does not match, but which specify the same result, will always appear together at the level of update sets. For example a query $\{\{$`type person`$\}\}$

and a second query having $\{\{$`type person`$\}$ $\{$`department legal`$\}\}$ are not identical but match the same located-objects when all people at the query location are from the "legal" department.

To detect recurring update query sets the DE server must keep a history of the update query sets that it has computed in the past. When a new update event occurs (at time t), the DE server computes its update query set, $u_q(t)$, and then searches its history of update query sets to see if the same query set has occurred before. If it has occurred several times before then it is a good candidate for having a multicast channel assigned to it so that updates to that set can be handled with a single update message. More precisely, the DE server keeps a list, H, of maximum size m, of previously computed update sets. Each list element contains the representation for a previously computed query set, a count of the number of times the DE server has encountered this particular query set, and a multicast channel identifier. Just as with individual query records, if the count for a query set is equal to or above the pre-specified value, q, then the multicast channel identifier will designate a multicast channel that can be used to disseminate update information. If the count is less than q then the multicast channel identifier will be `nil`.

When no multicast channel has been assigned to an update query set, update information must be disseminated separately to the subscribers of each query in the query set. This is done as follows: for each query in the query set send out update information using either the multicast channel or the list of channels that is associated with that query. We shall refer to this means of update as the *multiple-messages-per-query algorithm* in the rest of this section.

The exact steps the DE server performs for an update event occurring at time t are given below:

1. Compute $u_q(t)$.

2. Search for an exact match of $u_q(t)$ in the history list H. Each element in the history list, denoted H_i, is a prior update query set. An exact match occurs when each query in $u_q(t)$ is equal to exactly one query in a candidate query set, H_i. In otherwords, given the set of queries that satisfy the current update, see if the exact same set of queries occurred for a past update. Matching can be done efficiently by using a hash scheme

to reduce the number of potentially equal update sets in H to one or a few candidates. Equality matching of update sets is done using representations that have been sorted into a canonical order.

3. If no exact match is found then $u_q(t)$ is added to the front of H with a count of 1 and a `nil` multicast channel identifier. If this causes the size of H to exceed m then the least recently used query set is discarded from the list. We maintain a doubly linked list of update sets in which an item is moved to the front of the list on each use. This makes the update query sets sorted in the order of their last use, and the item at the rear of the list is the one we discard. Update information is sent to clients using the multiple-messages-per-query algorithm.

4. If an exact match is found—suppose it is element H_i—then the following steps are performed:

   a) Increment the count of H_i.

   b) If H_i's multicast channel identifier is not `nil` then use the indicated multicast channel to send out update information.

   c) If H_i's multicast channel identifier is `nil` and the count field is less than q then send out update information using the multiple-messages-per-query algorithm.

   d) If H_i's multicast identifier is `nil` and the count field is exactly q then perform the following steps:

      – Assign an unused multicast channel to the query set and assign the channel's identifier to the multicast channel field of the query set's record.

      – For each query in the query set send out both the update information as well as the identifier of the newly assigned multicast group using the multiple-messages-per-query algorithm. Clients are expected to henceforth join the designated multicast channel for all future update information[6].

[6]Note that because we use reliable multicast, clients are guaranteed to find out about changes in the set of multicast channels they should listen to, even if they miss the relevant multicast message. However, care must be taken to ensure that when a client detects a missed multicast message that contained such information the client asks for all relevant messages that might have been sent out already via the new multicast channel.

Given the ability to detect both multiply-subscribed queries and recurring update query sets one might wonder how important each scheme is to load reduction. We observe that in any system that is large enough to have a diverse clientele most of the multicast traffic is likely to go over channels belonging to the recurring query set scheme. However, tracking multiply-subscribed queries allows the query set scheme to avoid sending large numbers of unicast update messages when a query set does not end up using a multicast channel for update dissemination. Indeed, because of this, if only one scheme could be used, then tracking multiply-subscribed queries would likely be preferable to tracking recurring update query sets.

4.4.6 Multicast Switch Over Costs

The DE server assigns a multicast channel to queries with more than a pre-specified minimum number, q, of subscribers. This section looks in more detail at what value q should be set to. This depends on the implementation costs of switching to multicast channels versus continuing to employ unicast messaging.

The cost of notifying existing subscribers of the new channel are actually quite small. This is because when the server finds that q has been reached, it does not immediately send a channel command, but rather waits until the next update is required and piggybacks the channel command with useful data. A channel command consists of a multicast address, a sequence number, and an add/drop flag. This accounts for an additional 24 bytes to the data message. From this we might conclude that the cost is so low, it warrants setting q to 2, since sending one more message would save bandwidth when multiple clients are in the same wireless cell.

However, there are hidden costs. The IP multicast implementation uses maintenance messages in order to manage group membership [10]. So a mobile host invoking a system call to join a multicast group would also generate an extra message. In addition, depending on the implementation, other multicast management messages might occur over time. One problem is that algorithms for wireless multicast are still under development so there is no standard cost. However, some of the evolving wireless multicast protocols, such as [1], use reliable management messages over the wireless medium and employ a base station proxy to handle other traffic. In this case, keeping q at very small number is still reasonable as long as the update messages are of comparable or larger size than the multicast maintenance messages.

4.4.7 Limiting Bandwidth

Limiting the rate of updates to clients is controllable through the application programmer's interface (section 4.3.3). The effect of bandwidth-limited subscription on our multicast update schemes is as follows: instead of using a single multicast channel for each multiply-subscribed query or recurring update query set we now use $b + 1$ multicast channels, where b is the number of different bandwidth limits that have been specified for the relevant individual query or any of the queries in the relevant recurring query set. Each of the $b+1$ multicast channels is used to send a different "band" of update traffic for a query or query set. For example, if bandwidth limits b_1 and b_2 have been specified for a query then we employ three multicast channels, m_1, m_2, and m_3. The first b_1 bytes of update traffic in any given second will be sent via m_1. If any traffic is still available for sending during that second then $b_2 - b_1$ bytes of it are sent via m_2. Any remaining data is sent via m_3. Clients desiring bandwidth limit b_1 are told to join only m_1. Clients desiring limit b_2 are told to join both m_1 and m_2. Clients with no bandwidth limit are told to join all three multicast channels.

By using different multicast channels in the fashion described, the DE server only has to send out an update message once instead of b times. The price paid for this approach is that clients must be told more often of new multicast addresses to join; in particular, whenever a new client shows up with a different bandwidth limit than has already been seen for their subscription query. Since information about all but the first multicast channel to assign to a query or query set can be disseminated using an already assigned multicast channel, this overhead is not a problem.

4.5 Summary

This chapter presented the dynamic environment communication model. Dynamic environments follow a publish-subscribe metaphor, where a process publishing an object causes asynchronous updates to processes that have previously subscribed. For ease of use, information in dynamic environments is self-describing, based on keys and values.

The important characteristics of dynamic environments described in this chapter are:

- *Decentralized Service.* Instead of employing a global service the interface accesses communities of servers brought together at the client using a naming service.
- *Query patterns & low bandwidth communication.* The query pattern effectively permits the clients to filter and receive only the information they desire.
- *Latency Feedback.* The communication layer has the advantage of knowing the application's requirements with respect to information timeliness. When no other traffic is present, this information makes verifying connectivity more efficient.
- *Flexible data distribution implementations.* The asynchronous data model as opposed to an RPC approach lets us trade off use of multicast and unicast message distribution in the implementation. This is particularly desirable when a single dynamic environment set is the subject of multiple subscriptions.

Chapter 5

Context-Aware Computing Architecture

This chapter describes a system architecture for context-aware computing. The active components in this architecture use dynamic environments as a common data and communication interface.

The kinds of information that our system manages can be broadly categorized as related to devices, people, and their context. This chapter starts with a description of how this information is divided into components and how these components interact. Following this is a more detailed presentation of the dynamic environment servers that compose the architecture: *device agent*, *active map*, and *user agent*.

5.1 System Organization

Chapter 2 presented the various kinds of dynamic information that might be usefully employed by context-aware systems. The minimal collection of information required for such a system is people, devices, and context. Information about people is used to determine personal preferences and customizations as well as to permit adaptions according to who is nearby. Without information about people, software cannot take into account the notion that individuals have similar goals that software can support and divergent goals that software can mediate. People are a major part of the dynamics of work environments. Information about devices is also necessary. Devices are the tools that can be used to get the task at hand done. As stated earlier, “device” is used in a generic sense for anything that can interact by messages, such as printers, displays, mobile hosts, thermostats, phones, etc. Without device information users cannot automate their interaction with the environment and users are burdened by having to know about frequently changing devices. Finally, context, consisting of users' relationships to other objects in space, is necessary to tell not only what can be accessed but also whether access is desirable.

An architecture providing these basic information categories can be organized in a variety of ways. A logically centralized design has many nice features such as the ease of locating the information, and ease of adding new policies that reference multiple kinds of data. However, a centralized approach fails to scale, involves a trusted central authority, and therefore has privacy concerns, and does not handle the partially connected nature of mobile computing.

The architecture presented here divides the world of information into three groups corresponding to the three types of information above.

User information is private and individually customized, so it is encapsulated, at least conceptually, in a “user agent,” one for each user. This gives people control over their own information, including whether they want to make it public or not. Dynamic device information should be stored close to a device making it is less likely for applications to have device connectivity but be partitioned from the information required to access them. Therefore, device information is encapsulated into a “device agent,” one per device. Finally, the need to make queries over location and context information makes it useful to cluster this data into an “active map service.” This is for efficiency so applications don't need to query many places at once. These are the logical components of our system and are illustrated in Figure 5.1. These components satisfy the needs for a basic level of context-aware functionality. However, other derived dynamic environment servers may also be useful, for example, site-wide customization variables, inventories of equipment, etc.

5.2 Component Interactions

This section provides a description of how the various system components work together. The reader should refer to Figure 5.1 and the labeled components which are ref-

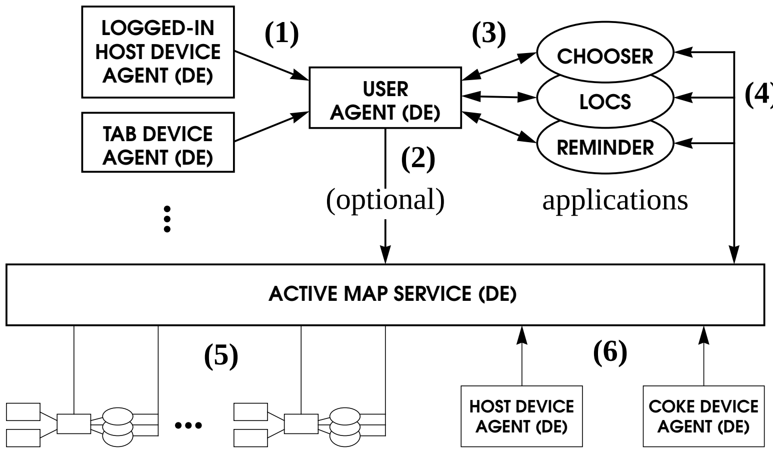


Figure 5.1: Components and their interactions

erenced in the text. In the figure, the rectangular boxes are all specializations of dynamic environment servers; the ovals are applications; the single arrows represent the publish operation and the double arrows represent the subscribe operation.

A number of processes cooperate on behalf of each user, including one or more device agents, a user agent and applications. These components are shown above the “active map service” in Figure 5.1.

The device agents are responsible for monitoring or managing a device, for example a workstation or palmtop. When the location or other state for that device changes, the device agent updates the device-object. Device agents for devices owned (or logged-in) by a user will publish objects to the user agent (1) and “public” device agents with no associated user publishes to the active map (6).

The user agent serves as an arbiter over user-specific information. For example, a person’s location information is synthesized by the user agent from various device agent sources. The user agent also publishes location information to the active map, subject to any privacy constraints the user has set (2). Publishing information in the active map permits sharing with other user agents who then note these changes.

A user’s applications subscribe to the user’s user agent dynamic environment which provides user global customizations and information (3). Using this environment, applications learn about changes to a user's preferences as well as the changing set of device objects that are currently "owned" by the user. Applications also subscribe to the active map service for context information, such as nearby people and public devices (4).

The final component of the system is the active map service. The active map stores public information about a geographic region and allows queries and subscriptions of that information. The active map is kept up to date by the various device agents and user agents. The active map functions similarly to a traditional name service in that clients go to this service to find each other as well as relevant located objects. Figure 5.1 shows that a number of users access the map concurrently, treating it like a meeting place where they go for sharing contextual information (5).

This section presented a description of how system components interact and how location and context information flows between them. However, because the system is based on a common dynamic environment interface and is structured as an open system, other interactions are also possible. For example, applications can directly interact with device agents through their dynamic environment interface. Additionally, device agents may provide binding information to device-specific interfaces in the device object. So an application might very well interact directly with devices or other processes using domain specific interfaces. This is the case in the PARCTAB system described in Chapter 6.

5.3 Device Agent

Device agents are dynamic environment servers specialized to monitor and provide information for workstations, printers, palmtops, or other particular devices. The agent records device-specific information in a "device-object." When the information about the device changes, for example from in-use to not-in-use or from one location to another, an update message is sent. For public devices the agent updates the active map and for private devices the agent updates the user agent. The reasons for this distinction are described in section 5.5.

In a context-aware system there may be many kinds of device agents, for example,

Name	The device name, e.g., Badge:38
Location	The location of the device.
Agent	Connection information for the functional network interface of the device if one exists.
DE	Connection information for the Dynamic Environment network interface for the device.

Table 5.1: Device Agent Global Attributes

PDAs, hosts, and displays. The information contained in a device-object varies from agent to agent, but serves a common purpose: it is a source of access information for the associated device. Towards this end, a device object may contain status, capabilities, costs, and binding handles to the device's network interfaces, among other information.

The set of attributes common to device agents is shown in Table 5.1. The device-object includes a name identifying the type and instance. A location identifier tells where the device is located, which for mobile location sensing devices may be updated over time. The device object may also contain a handle to the device's functional interface. For example, a display device agent may provide an address for an X-window display manager. These common attributes are augmented by device-specific information. A display object might include a dimension, and a PDA might include an idle-state.

For applications, the primary source of device objects is indirect through the user agent or active map since their job is to provide an organized view of device information. Device agents, however, also export a dynamic environment containing the device-object along with other state. When this state changes, subscribers of the dynamic environment automatically receive updates. This secondary means of disseminating device information serves a number of functions. For applications employing the device, this interface can provide finer grained information about the internal device state without necessarily distributing this information to all casual active map subscribers. The dynamic environment is also a convenient debugging and logging mechanism for device usage. Finally, the environment provides a connection point for deploying other experimental architectures.

(1) What is co-located with Bob?	(4) Are Bob and Phone-3 on the same floor?
(2) Is Bob in the building?	(5) How far is it from here to Printer-10?
(3) Is PDA-3 in room 2101?	(6) How do I get from room 6A to there?

Table 5.2: Types of Questions for an Active Map

The device agent's dynamic environment interface can be found in a number of ways. An agent can be looked up using the discover protocol described in 4.3.1 or it may be looked up by location in the Active Map Server described in the next section. The interface may be found using a reference from another object. For example, a User object may include a reference to a Host. Finally, a reference to the device agent can be passed to applications on their command lines when they are started. Applications for the palmtop PARCTAB computer, described in the next chapter, use this last method to learn the agent network address.

5.4 Active Map Service

The *Active Map Server* (AMS) publishes dynamic environment objects having a location as well as other geographic information. Each AMS supports a set of locations called a *region*. For a particular region, the active map is the meeting place where clients can find each other as well as relevant located objects. The AMS acts as a location-oriented directory service: location serves as the key that clients use as a first step in determining most of a user's context. The kinds of information that the active map can provide are summarized in Table 5.2.

The remainder of this section describes the key concepts in the design of an active map service. We start with the notion of "location" as a spatial container and, later, as a distance along a path. We present located-objects, the entities that populate an active map service. We then describe the located-object operations: query, subscribe and move.

5.4.1 Service Organization

The design of an AMS faces a tradeoff. On the one hand clustering of location information is advantageous so applications don't have to query many places at once. Ideally the information cluster should be as big as possible without performance of individual queries suffering because the server becomes overloaded. The approach here is to support regions that cover the geography over which context-aware applications naturally perform queries, e.g., a building or campus.

The active map design consists of a collection of regional servers. Regions cover localized administrative or geographic domains, such as buildings or campuses. Although not strictly necessary, regions should be selected such that they reflect the locality of user movement for reasons discussed below. One drawback of this approach is that clients are made responsible for shifting from one region to another, and in the case of a client interested in information on the border of two regions, managing the interactions with multiple regions.

We believe that shifting between multiple regions is not a frequent occurrence if regions are suitably defined. This is based on the observation that a user's access pattern is largely influenced by a "locality of reference." This locality is similar to the idea in computer memory systems: once an access is made to an object, future accesses tend to reference nearby rather than distant objects. In terms of our system, the most useful objects for context-aware computing are close at hand, either co-located or requiring a short time to get to. Context-aware applications are more likely to monitor the contents of the room and building they are situated in than the contents of a building in the next town. In other words, located-information used by context-aware applications will have a high locality of reference based on physical proximity. Moreover, a person's preferences for learning about and interacting with nearby things in conjunction with the architectural spaces people inhabit leads to the conclusion that most AMS subscriptions will be within a campus, shopping center, work site, or other group of buildings, and probably within a single building. People tend to not care about adjacent regions very often, that is, until they actually make the transition into the neighboring space. Therefore organizing information into regions will generally not cause extra communication overhead in managing multiple regions as long as

```
{{Name User:ragnap}
 {Location LID:PARC-35-2-1-01}
 {personName "Ryle R. Ragnap"}
 {unixName ragnap}
 {office LID:PARC-35-2-1-01}
 {Agent sunrpc_2_0x31000041_1|tcp_13.2.116.64_3577}
 {DE tcp_13.2.116.64_3579}}
```

Figure 5.2: Example Located-Object

regions are big enough.

Therefore clients are required to accept the burden of interacting with any and all AMS regions that constitute larger domains of interest, such as cities and states. An important consequence of this approach is that the implementation of an AMS service must be able to scale to cover regions the size of buildings and small campuses so that "regional" queries can be implemented in an efficient fashion. This is supported by measurements presented in the chapter 7.

There are a number of advantages to the regional approach. First, the service can be partitioned among servers each managing one or more regions. This means that scaling the system to cover larger geographic domains is just a matter of adding more servers to manage those domains. Second, regions provide a natural administrative barrier. The information pertaining to a domain can be maintained and access controlled according to the domain preferences. That is, the method provides administrative autonomy.

5.4.2 Located Objects

The active map manages dynamic environment objects called *located-objects*. Any information that a client wishes to make publicly available can be entered into the active map with a located-object registration; however, every located-object must include a `Location` attribute that describes its current physical location. Examples of located-objects include persons, printers, terminals, telephones, PDAs, pagers, and workstations, as well as

measure	example
in (containment)	in building 35
at (exact location)	at location 35-2-1-01
with (co-location)	in the building with John
path distance	A to B is 10 feet
path	from A goto B, distance 3

Table 5.3: Active Map Geographic Information

location-dependent services, such as information agents associated with particular parts of a store or building.

The value of the Location attribute is an identifier pre-defined to be an AMS location. This value is called a *location identifier* or LID for short. For example, LIDs might be PARC-35-2-2-00 and commons. Location identifiers represent real world places such as a room, hallway, building floor, building wing, building, street, town, and so on.

The definition and meaning of other attributes associated with located-objects are defined by the specific application clients using the active map. As in dynamic environment objects, each key is a literal text and each value may be either a text or list. These attributes permit queries on subsets of located-objects and enable applications such as browsers. Figure 5.2 shows a located-object representing a computer user.

5.4.3 Location Information

In addition to providing a dynamic environment of located-objects, the AMS also manages information about the relationships that exist between locations.

In people's daily lives two kinds of spatial relations are commonly used: containment and travel distance. These relations can answer the high-level questions shown in Table 5.2. Other spatial relations, discussed later, such as Euclidean distance between positions within a coordinate system, are not as well suited for human activity.

In order to answer these kinds of questions a range of knowledge is necessary. For example, that an object in a room it is also in the building containing that room and knowing

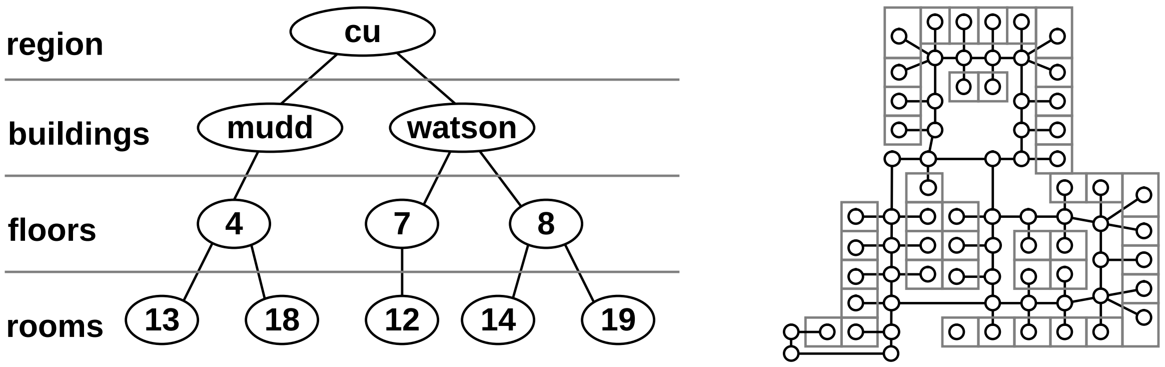


Figure 5.3: Active Map Containment Hierarchy and Graph of Locations

how far it is from one place to another as well as how to get there. Table 5.3 summarizes this knowledge. The following sections describe the information structures used to provide this information, namely the containment hierarchy and graph of locations which are shown in Figure 5.3.

Containment

All locations known to the active map have a containment relation: rooms are contained in buildings and buildings are contained in a region. The active map supports functions to describe the hierarchy, and the comparing of two locations to see if one is contained within another. In addition, queries on the AMS can be targeted at all objects within a particular container. It should be mentioned that the detailed structure and depth of any region's location containment hierarchy is region-specific, and the containers, which are located-objects themselves, are determined by a system administrator.

A location is defined to be a spatial container – something having physical dimensions encompassing a domain. To be *in* a location means to be within a location's domain; to be *at* a location means to be in no smaller location's domain; and two objects *with* each other share a common location. For example, a person *at* location 2101, or *in* building 35. Additionally, containment tells us whether two objects share some common location domain, in which case they are "with" each other. This property can be used for questions such as "is

```
procedure in(lid1,lid2)
begin
    return substring(lid1,lid2);
end
```

Figure 5.4: Computing "in"

John in the same building as Doug?"

Location identifiers are formalized to use the container names from the root separated by dashes, similar to Unix path names. For example:

```
PARC-35-2-1-01
CSB-2-10
```

The first shows the PARC region, building 35, floor 2, pod 1, room 01. The dashes make the container structure explicit. The meaning of each level in the hierarchy, however, may change from region to region. The system administrator defines hierarchy *labels*, e.g, building, floor, which are made available to clients.

The LID format is useful for figuring relations between two locations. So, to see if one object is "in" a location a procedure checks if the location is a prefix of the object's location (see Figure 5.4). To be "at" a location, which is defined to mean situated at exactly the same location, is simply an equality test. Questions of the form "in the same room as" and "in the same building as" can also be answered. The algorithm for this computation takes as input the locations of two objects, e.g., PARC-35-1-02 and PARC-35-2-02, along with the region's hierarchy containers labels (e.g., region, building, floor, pod, and room), and the container label for the query, e.g., building. This is shown in Figure 5.5. Steps 1 and 2 use text operations to determine the index for the least common ancestor of the two locations. For the example locations above, this would be two because the common prefix PARC-35 has two components. Step 3 and 4 determine if the requested label, e.g., "floor," is a container that is part of the common prefix.

```
   procedure with(lid1,lid2,region-hierarchy,label) returns(bool)
   begin
1.     differ = the character position where lid1 and lid2 differ;
2.     ncontainers = the number of dashes in lid1[1..differ-1];
       ncontainers is now the number of labels in common
3.     lposition = the position of label in region-hierarchy;
       lposition is the level of the requested label
4.     if lposition ≤ ncontainers then
5.         return true
6.     else
7.         return false
   end
```

Figure 5.5: Computing “with”

Travel Distance

Geographic containment information does not, however, serve well for questions about distances between locations and how to get from one place to another. In order to see if two locations are “within 15 ft. ,” of each other it is useful to store distance information for adjacent objects. The active map also represents the world in terms of a graph-of-nodes describing the pathways that can be taken between locations encompassed by the graph. In this graph, each location is defined as a vertex and each pathway between two locations is an edge.

A graph-of-nodes representation captures the essential nature of distance in this system model because people generally move between two places by a single, shortest path. There are few desirable paths between locations because corridors and other architectural elements constrain how people move. At larger scales, elevators and stairs bridge floors, and on a campus, walkways connect buildings. Paths between distant places can be computed by combining information from these multiple scales.

When the system administrator creates the AMS location-hierarchy they also define a graph of locations and the distances between them. A program takes this graph as activemap_hierarchy(region) returns(labels)

> *Return the labels for each level in the region's hierarchy. For example "region-building-floor-room" as a text string.*

activemap_distance(src, dst) returns(distance, step)

> *Return in* distance *the number of feet between* src *and* dst. *Return in* step *the first step along the shortest path between* src *and* dst.

activemap_path(src, dst) returns(path)

> *Return all the steps between* src *and* dst *in a text list in* path. *The list consists of triplets* {distance left next}*, where* distance *is the number of feet for the current edge,* left *is the number of feet remaining for the path, and* next *is a location identifier for the next step.*

Table 5.4: Active Map Path API

input and produces a path table, which is loaded into the AMS during program initialization.

For each location pair the table stores a <distance, step> record which is a standard technique to concisely represent paths in a graph. The distance is a number for the length of the path from the source to the destination. The step is a location identifier for the first location along the path. Clients reference the path information using the interface shown in Figure 5.4. The remainder of this section describes how path information is used.

Two uses for path information are to generate directions on how to move from one place to another, and to find objects within a certain distance. Generating complete path information between two locations can be used, for example, in an application that provides directions to users. This procedure is relatively easy because the step field of the path record contains the next location along the shortest path. Generating the path is then shown in Figure 5.6.

The second use of path information is to find objects within some distance, D, of a particular location. This is done by selecting all the matching objects in a region, say

```
procedure path(source,destination)
begin
    activemap_distance(source,destination,distance,step);
    while destination $\neq$ step
        output step;
        activemap_distance(step,destination,distance,step);
    end
end
```

Figure 5.6: Computing "path"

printers, and then verifying that the path distance from the source falls within the required limits. The same approach can be made more efficient by pruning the search to only include certain geographic containers.

The path information that is loaded into the AMS at initialization time is computed using the all pairs shortest path algorithm [3]. This computation takes time $O(n^3)$ and space $O(n^2)$ for a graph of n vertices. For our AMS configuration there are 127 vertices (locations) and 136 edges. Generating the table takes 120 seconds on a SPARCstation 2 and results in a 500 Kbyte initialization file.

For large AMS regions the preprocessing time and space become limiting factors. The method used computes all paths at once; however, for larger regions it is possible to trade off online and offline processing by having the graph-of-nodes repeated at different levels (or scales). In this case the hierarchy includes edges, for example, of roads between building domains. To manage this the AMS region is partitioned into multiple components each with shortest path information along with shortest path information between components. Components follow the divisions in the containment hierarchy, for example, each building is a component. Finding the shortest path involves joining a number of segments: source to component exit, component exit to component entrance, and component entrance to destination. There may be as many as b^2 comparisons necessary where b is the maximum number of boundary (i.e., exit and entrance) edges. In addition, the offline processing involves additional steps to compute the shortest path between components. This is done by removing all vertices along the shortest path within a component and then running the shortest path algorithm the remaining boundary points.

Other Spatial Relations

Although the two spatial relations described above permit a good deal of expressiveness, there are other spatial relations as well. Two such relations worth mentioning are absolute Euclidean distance between two coordinates and distance measured by various phenomena.

The Euclidean distance does not account for the physical limitations that people deal with. For example, under Euclidean coordinates, the distance between two rooms is the same whether the rooms are side by side or on top of each other. Euclidean distance is no more useful than “as the crow flies” directions are to drivers that must stay on roads. Given room-sized spatial resolution, the path distance described in the previous section provides a more realistic approach to spatial relationships. If smaller distances were distinguished then a hybrid scheme incorporating coordinate distance would be desirable.

There is also a class of spatial relationships that deal with the propagation distance of phenomena: within earshot of; in sight of; out of reach of; in range of; out of earshot; out of sight of. For various reasons these relations are difficult to compute. For one thing, they depend to a large degree on the characteristics of the subject and object. In some cases a detailed map of paths and obstacles along with a route planning mechanism is needed. In other cases subjective measures, such as seeing or hearing are involved. To use distance measured by these classes of sensation phenomena literally would require detailed knowledge of subject, object and the physics of the communication media. Fortunately they are generally less common in usage. Our system does not rule out the use of these more computational-based measures, but it does not directly support them. They must be delegated to agents with the specialized knowledge necessary to answer queries in these forms. “Within sight” becomes interesting at the sub-room level if the orientation of a user’s head can be detected by the system.

activemap_query(location, pattern, maximum, options) returns(result)

> *Query the active map server for up to maximum objects matching the pattern. The location specifies the active map container to search. Returns a list of objects.*

activemap_subscribe(location, pattern, options, fcn, arg)

> *Initiate a subscription for objects matching the pattern within the container specified by location. Options may include* situated-with *plus a located-object for positioning the query relative to some object. Invokes fcn with arg for each matching object.*

activemap_unsubscribe(location, pattern, fcn, arg)

> *Delete an existing subscription.*

activemap_move(location, object-name, object)

> *Direct the active map to move an object from one location to another. The* object-name *is the concise object name. The* object *parameter is optional and used to rewrite the contents of the object.*

Table 5.5: Active Map Query API

5.4.4 Query & Subscribe Operations

The AMS query interface is an extension of the generic dynamic environments interface. The one-time query returns information about located-objects at the time of issue, whereas subscriptions return a stream of updates over time as the active map changes. The AMS permits two options for queries that assist in exploring the hierarchical structure of located-objects. Queries can have a target container and they can be “situated-with” a located-object.

When a query has a target, only located-objects within that container are returned. When a query is situated-with a particular located-object, the query follows that object as it moves within the active map. Consider the case of software on a mobile host that wishes to be kept informed of objects in the host’s vicinity. On each move operation the host could delete and reinstate at the new location the current collection of standing queries. The situated-with location option permits a potential optimization by permitting the delete-install inner loop to be implemented at the server instead of within the client.

5.4.5 Move Operation

Agent processes that manage objects are responsible for updating the object's state within the active map. For example, a user agent will update the associated user-object information. When an object's location or attributes change the agent issues an active map move operation. The move places the object at a new location in the active map and causes a post-processing stage. During this post-processing, the active map may send update messages reporting the changes to subscribers that have posed standing queries.

5.4.6 Active Map Scaling

A key issue in building active maps is that of scale. These concerns in part motivate the design of dynamic environments in Chapter 4. We review these AMS-specific requirements here. A variety of issues must be dealt with to avoid overloading the AMS, its clients, and the communication facilities that join them:

- Large meetings of mobile users may cause considerable traffic because context-aware clients at the meeting as well as remote applications may need to be informed of the frequent occupant changes. High message volume may also occur in high traffic locations such as building lobbies.

- Low bandwidth links (usually wireless) limit the amount of information that can be sent to some clients.

- Mobile hosts have significantly stricter resource budgets and hence cannot do as much work as other hosts. For example, concern for power conservation may limit the activity that battery-powered hosts are willing to perform on an ongoing basis.

As mentioned simply partitioning the AMS into many servers, each managing a small area, is only of limited help because clients frequently desire to know about information covering an entire region whose size is independent of what might be most suitable for the AMS. For example, clients may want to know about all people in a building.

The worst-case usage scenario that our architecture is designed to handle is the "meeting scenario" in which several hundred people, all using context-aware applications, converge on an auditorium over a period of several minutes. To illustrate some of the issues involved, consider how location information about this scenario might be disseminated to the people involved: the simplest—a naive unicast approach—would result in n updates going to each of n clients, which quickly gets out of hand. Waiting for quiescence over a time interval and batching the updates would reduce the message count to 1 to each of n clients, but would sacrifice timeliness. Using broadcast or multicast [11] for the updates would reduce the aggregate message traffic even further, but would likely *increase* the traffic seen by some clients, since not all clients are interested in exactly the same information. This increase may be problematic for clients residing on small hosts (such as PDAs) or those connected via low-bandwidth links.

5.4.7 Active Map Availability

Our design relies on a centralized active map server, implying that the AMS will stop functioning whenever the server crashes. This is not a problem in practice for two reasons:

- The active map server is host-independent and hence can be run on any available server machine. A distributed watch-dog facility can monitor and restart the server if it fails.

- A well-known multicast channel allows a newly restarting AMS to obtain current location information from all located-objects agents (i.e., device and user) currently in its region. Agents are expected to be listening to this channel address for a SERVER message in order to re-publish their located-objects with the AMS.

The result of this approach is that active map servers do not stay down for very long, assuming that the system makes two or more machines available as potential server hosts.

We have not addressed the issue of partitioning an AMS region. One can imagine AMS clients starting up new instances of the server in their partition when they notice its absence and having servers watch for each other (e.g., using the well-known multicast channel mentioned above) and perform re-integration whenever partitions heal. However, this represents future work.

One might be tempted to avoid the availability issues of a centralized service altogether by trying to implement the AMS as a distributed communications protocol for exchanging information between changing location-based objects and interested clients. Located-objects could multicast their update information and clients could multicast queries to find out what they need to know about the current state of some part of the system. Unfortunately such an approach will, in general, impose more filtering load on the system's located-object managers and clients because there is no centralized "traffic router" that has extensive knowledge about who is interested (and more importantly, *not* interested) in the range of information. Given the need to avoid overloading weak clients and slow communication links we consider the disadvantages of a totally decentralized design to outweigh its advantages, especially given the ability of a centralized server to quickly recover from failure.

5.5 User Agent

In a context-aware system each user has a collection of personal information that might include, among other things, their preferences, the characteristics of devices currently in-use by that user, and the time and location of the most recent interaction. In our system, each user's information is logically centralized in a user agent. This provides a well-known place for a user's own applications, as well as applications run by others, to find out about the user. This section describes the user agent from a functional viewpoint and proceeds with a presentation of the agent's distributed organization and protocols that gives applications some degree of functionality while disconnected from other system components.

5.5.1 Functional Description

The user agent performs two functions. First, it manages a dynamic environment, called the *user environment*, containing user-specific information. Second, analogous to a device agent that creates a device-object, the user agent creates a user-object giving details about the user including their last known location.

The user environment allows applications to share user-specific state across devices. In this role, it provides a common "execution context" for all of a user's applications. The environment includes personal preferences like handedness as well as contextual and location-dependent customizations the user has specified. This customization information is initialized from a file when the user agent starts and may be changed by the user anytime thereafter.

In addition to updates from the user, the environment is also updated by device agents. Specifically, when a device is "owned" (i.e., in use) by the user, the device agent sends device-object updates to the user agent. For example, once a user logs into a host, a host device agent monitors the keyboard and mouse, and upon sensing activity will forward the sightings (a device object) to the user agent. Device objects contain various attributes including the time and type of interaction, and the device location. In the case of a stationary host the location is a constant value and for a location-sensing PDA device agent, the location varies over time.

By updating the user environment, device agents keep the environment populated with a collection of device-objects. This is useful because applications supplied with information about the collection of devices in use by a user can more easily employ multiple devices. For example, when a user returns to his office and logs in to a workstation, applications running on a laptop might switch over to use the larger display.

The second function of the user agent is to manage it's own user-object. This object is updated in the user environment and provides the best estimate of the user's location. The user-object is a derived piece of information since it is computed from sightings of device-objects. The reported location is only an estimate because device agents may produce multiple streams of potentially conflicting location sightings. This might occur when a user leaves their active badge on their desk, or someone types on a workstation logged-in to another user. It is the job of the user agent to weigh the accuracy of incoming reports and to compute a best estimate of the user's current location. When the user agent computes a new location it updates the user-object in the user environment. The user agent, if so instructed, also produces an active map move operation for the user-object. In this way other active map subscribers see updates to the user's location.

5.5.2 Example of Use

Applications subscribe to the user agent's user environment in order to obtain user information. They also pose application-specific queries to the active map for public contextual information, such as nearby public displays and other people. This section provides a brief example of application usage, and explains why the computation of "co-located" objects should rely on location information from a device-object, rather than from the user-object.

As an example of how the user agent is employed, consider a mail reader program run on a mobile host. When the application starts up it locates the user agent and subscribes to the user environment. It obtains the settings of various preferences, including whether incoming mail should produce an audible trill. This value may be changed by the user, user agent, or other applications during the course of the day, and the mail reader will be kept notified. The mail reader is also programmed to subscribe to the active map service for the changing set of co-located objects. It can, for example, notice people in the same room and highlight flagged or unanswered mail messages from them. This allows the mail user to communicate the mail information directly.

The above example mentions the use of "co-located objects" by applications. Lets clarify what this means. First, a co-located object is an active map object that has the same location container, usually a room. However, it is not immediately clear whether the "location" that selects the objects should come from the user-object or the device-object on which the interaction is occuring. Note that since the user-object may be computed from multiple location reports these two are not necessarily the same. A difference occurs, for example, when a user interacts with someone else's PDA. The problem here is that a user will be surprised if the palmtop device they are interacting with says it is located across town. Fol-

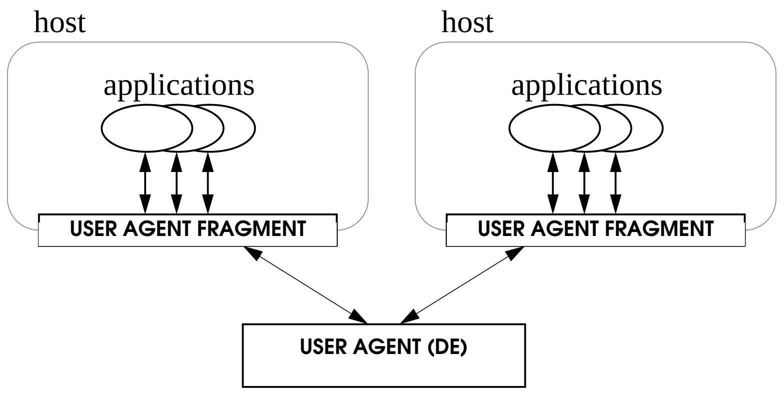


Figure 5.7: User Agent Organization

lowing the principle of least astonishment, context-aware applications written for mobile hosts should generally employ the device location for the target of a context subscription. User location only needs to be used if some interaction specifically deals with that user, for example, a display of all users.

5.5.3 User Agent Organization

The user agent is an abstraction that provides user-specific information, including the execution context for a user's applications. It can be implemented in a variety of ways, such as a logically centralized server. This approach make it easy to locate information and has only a single place to introduce new policies that use a variety of information.

Instead of a completely centralized approach our user agent design employs a back-end "user agent" process plus a front-end "user agent fragment" (UAF) process running on each host executing a user's applications. This section describes why this design is advantageous for efficiently managing environment subscriptions, and how such a division is desirable in the face of network partitions.

Under normal conditions, the front-end user agent fragment acts as a smart pass-through for communicating with dynamic environments, and in particular, the user agent and active map. The UAF sits between applications on the local host and dynamic environments on networked servers. This is shown in Figure 5.7. The UAF provides a more efficient use of network resources since it can collapse identical active map and user environment queries and reduce communication overhead when multiple applications on the same host reference these remote servers.

The second advantage of a user agent fragment residing on each host is to manage dynamic environment information during a network partition. Although complete information is not available during partitions, applications can achieve a certain degree of context-aware function. Specifically, when multiple applications exist, either on the same or different hosts, they can potentially share their global environment variables. Also applications can potentially learn information about the execution context when device agents, and other sources of context information are present.

Two kinds of partitions can occur in this system. A “wired-partition” might occur when a link between the user agent and applications goes down in the wired network. A “wireless-partition” occurs when connectivity between wireless hosts and the wired network is disrupted. Partitions of this kind might occur when a group of people using infrared networks are out of range of a base-station, but may otherwise communicate among themselves. It is this latter form of partition that our design is geared to handle.

It is useful to make a distinction between partition types because they warrant different solution approaches. Wireless-partitions consist of small groups of hosts with limited resources. Regenerating a central server is not practical in this case because there may not be a host available to run the server.

Given that regenerating a central server is not feasible, it is worth reviewing whether the lack of a centralized manager of communication resources is important. There is little advantage to a centralized server when partitions consist of a single broadcast cell. This is because environment servers manage various loads by redistributing traffic onto a number of multicast groups. Although multiple multicast groups along with intelligent network interfaces can reduce host message filtering, the real benefit of such an approach is to reduce network utilization. But when all clients share the same broadcast cell no grouping alone will change utilization, and only when the partition size is large enough to cover multiple cells does the multiple multicast group management reduce traffic. It can be concluded that centralized traffic routing is less important for partitions of a small number of cells. This

Message	Response	Description
any	SERVER	Dynamic environment server hearing any traffic redirects sender to use central server.
UPDATE		Update of a dynamic environment to peers.
QUERY	UPDATE	Client request of matching objects from peers.
CHECK	UPDATE	Client request of update of a dynamic environment.

Table 5.6: Summary of Decentralized Message Exchanges

supposes that all clients are able to filter all queries, which is the case in wireless-partitions.

An alternative to server regeneration is to decentralize the management of dynamic environment using broadcast (or multicast) to send updates and queries. Such a decentralized method has the drawback that for large partitions, steps must be taken to avoid network overloads. These steps may involve limiting the domain a message is distributed over, or limiting the overall bandwidth that dynamic environment traffic can consume. These solutions sacrifice global knowledge and timeliness. A distributed multiple group assignment method is another way to manage the overload problem. However, such an approach puts much more of a resource demand on clients since they need to manage channel information as well as environment objects.

The choice between centralized and decentralized methods for managing partitioned networks presents a design tradeoff. Large configurations, characteristic of wired-partitions, can be handled using regeneration of a centralized service that can then manage systems loads. However, since the target environment is mobile hosts, ruling out configurations of limited hosts is a serious restriction. In addition, large partitions will be less likely to occur since this means a failure of hard-links as opposed to lack of wireless service coverage. For these reasons a decentralized approach for managing partitions is more suitable, and is described in the next section.

5.5.4 Decentralized Dynamic Environments

Because the active map and the user agent are based on a dynamic environments, our solution for managing information during network partitions addresses how to manage dynamic environment databases in a distributed manner. One reason for a user agent fragment instead of a simpler dynamic environment cache is because it allows, for example, the more sophisticated behavior described here during disconnected conditions. The problems necessary for supporting a decentralized approach are how active map and user agent information is disseminated, where the user location computation occurs, how switching between centralized and decentralized situations occurs, and how overload cases are managed.

In this design, messages are sent over a dynamic environment channel based on a pre-defined multicast group. When a partition consists of a single cell, the messages are effectively broadcast. Both clients and servers of a dynamic environment listen to this channel at all times. During central server operation, however, there is no traffic.

The remainder of this section addresses the partition problem using decentralized maintenance of a dynamic environment and how overload conditions are handled. First, it presents the message types exchanged between agents, servers, and user agent fragments. A summary of messages is shown in Table 5.6. We continue with a description of how these messages are exchanged and conclude with a explanation of how network overload conditions are handled.

Message Types and Exchanges

Four kinds of messages are sent over the dynamic environment channel. The UPDATE message is broadcast by device agents and others when an environment object is modified. The QUERY message requests updates from peers for objects matching a pattern within a particular dynamic environment. The CHECK message is used to synchronize copies of an environment among multiple user agent fragments. Finally, the SERVER message is sent by a user agent or active map server to advertise it's presence.

Specialized environment servers–the active map and user agent–are registered to receive messages broadcasts over the dynamic environment channel. Normally there is no traffic on this channel. When an environment server restarts, a SERVER message is sent containing the environment and server address. This permits clients to reattatch. If an environment server notes any other traffic for their environment, it means that clients have reconnected after a partition. When this happens a server simply broadcast another SERVER message.

Under normal conditions, the user agent fragment and device agents update dynamic environment servers. An update consists of reliable delivery of a message containing an environment name and one or more environment objects. A partition is detected when the reliable delivery of one of the update message fails. In this case, clients redirect the message to a multicast channel where they unreliably send the UPDATE message. These clients continue to send updates to the multicast channel instead of directly to environment servers until they are redirected with a SERVER message.

Before describing the QUERY and CHECK messages we need to explain how objects are managed. In a decentralized approach, with no single point of coordination, it is possible to encounter objects with the same name but different contents. To support unix filesystem semantics it is desirable to distinguish the most recently written version. To manage this a *modification timestamp* is added to the name of objects. The timestamp is a <host, time> pair, where host is the place where the object was modified and time is the hosts' time when the modification occurred. Before an object is updated in an environment, it is compared with the existing object, if any. When the host part of the timestamp are the same, the object with the smaller time is overwritten. When the hosts are different, the default resolution policy is the same (i.e., Unix semantics, "aggressive" clocks always win)[1].

The user agent fragment listens for UPDATE messages for the user's environment and the active map region and updates the local copies of these dynamic environments. This allows the UAF to passively keep up to date the copies of an environment. The QUERY message is used to actively request updates on an environment. Query involves sending an environment name and object pattern in order to provoke peers to send UPDATE messages.

The CHECK message is used to efficiently converge two clients having different copies of the same environment. A client periodically sends out a CHECK message with the

[1] In practice this should not be a problem because most objects are written from a single device agent.

name of the environment and the checksum of all object names and versions. If a client hears another CHECK message with the same checksum it does not need to send one out for the next period. If, however, the checksum differs, the two clients create a reliable connection and exchange environment information.

The user agent, device agent and servers employ these messages to provide a similar level of functionality while partitioned as while connected to a central dynamic environment server. When a client fails to update a dynamic environment, they enter a redirect state where updates for that environment will be unreliably sent over the dynamic environment multicast channel. The UAF also starts sending a periodic CHECK message for environments that have subscriptions. This permits clients to notice updates that might have been lost, or to acquire updated objects from hosts moving into a wireless cell. The QUERY message supports client queries for environments that are not being kept synchronized by the UAF. Finally, servers receiving any messages for their dynamic environment issue a SERVER message thereby redirecting clients back to the single server.

Handling Overload

When partitions are large or encompass agents generating rapid update messages, communication overload results. Two rules are used to provide graceful degradation in the decentralized case. First, the scope of update messages is limited to a site using multicast time-to-live or other similar mechanisms. Second, the channel utilization for all shared environments is given a preset maximum percent by the system administrator. Clients generating updates monitor the environment channel and, depending on the observed traffic, may wait an interval of time before sending update messages in order to keep the channel traffic within the specified capacity. The waiting policy employs a probability that the channel is idle. When there is no measured traffic this probability is 1 and when traffic equals or exceeds the capacity limit, the value is 0. Before sending a message the client decides to to send the message or delay using an exponential backoff depending on a randomized decision based on this probability, similar to the Ethernet communication protocol. Other schemes for prioritizing information are also possible.

As mentioned earlier, context-aware software on mobile hosts should be designed to employ the host's instead of the user's location for everything except situations where the individual's location is clearly appropriate. Computed user location information is useful for some applications, however, there no longer a single place where user information is being created in the decentralized case. Because the computation of user location is fairly simple it can be carried out in the UAF process. The UAF monitors UPDATE messages for devices objects associated with the user. When a new user-object is computed, the UAF sends it in an UPDATE message only if the object differs from the one already known. This is necessary to prevent UAFs thrashing when a user-object changes only a timestamp value.

This section has described a design for managing partitions in a context aware system. A centralized server approach is desirable because it can manage system load and is similar to the non-partitioned case. However it has little benefit in small partitions and restricts the set of possible configurations to those that can support servers. Instead, we have presented a decentralized way to manage dynamic environments using multicast messages. The network overload that occurs for large partitions is avoided by using a bandwidth limit. This approach permits simple clients but in the high load case, sacrifice timeliness of updates. We think this is an acceptable trade-off since large-scale partitions occur from back-end communication failures as opposed to wireless link failure or lack of coverage.

5.6 Summary

This chapter has presented a context-aware computing architecture consisting of three logical components: device agents, active maps, and user agents. Each of these components is based on the dynamic environment publish-subscribe communications model presented in chapter 4.

Device agents manage device-objects that are published in the active map or the appropriate user agent if they are "owned." One important attribute contained in the device-object is location.

The active map allows searches over located-objects registered within a particular location. Locations are defined as part of a containment hierarchy and are therefore nested. The active map knows about a number of spatial relations: containment, exact location, co-location, path distance and paths. Active maps are a meeting place where clients can find each other as well as relevant located objects. The design of active map service relies on a clustering of location information into a number of servers. This is not a problem in practice because the dissemination method in chapter 4 allows fairly large groupings that cover geographic regions that encompass the objects people can easily interact with.

The third component in the architecture is the user agent which encapsulate user-specific information. A user agent exports a user environment and publishes a user-object. The user environment contains all of the devices in use as well as user-global parameters and customizations.

The user agent design incorporates a back end server and a number of front-end fragments run on each host executing a user's application. The purpose of the fragment is to efficiently manage subscriptions and, more importantly, to provide a method of continued operation when the application is partitioned from the backend. This chapter presented a design for sharing information when the fragment is separated from the back-end but still in contact with other devices and hosts. This situation may occur when, for example, there is no base-station but a number of other mobile hosts are present.

Chapter 6

A Context-Aware System

The design of context-aware applications depend to some extent on the structure and capabilities of the underlying mobile distributed computing system. While chapters 4 and 5 presented a general architecture for context-aware computing, this chapter gives a specific example, the PARCTAB system, along with four examples from context-aware application categories.

This chapter starts with a description of the PARCTAB's hardware, network and system design. Only a general overview is presented, further information may be found in [2, 34, 45].

The chapter continues with a description of how the PARCTAB system integrates into the context-aware architecture presented earlier, and concludes with a description of prototype applications.

6.1 Guiding Principles

The PARCTAB system consists of palm-sized mobile computers that can communicate wirelessly to workstation-based applications. The system was built to explore the capabilities and impact of mobile computers in an office setting. The goals of the project include the design of mobile hardware and system software for personal communication and computing.

The PARCTAB system is designed around a small number of basic principles and assumptions:

- *Extreme portability.* The device is designed to be carried or worn at all times, much like a pager. It's size, weight, and features are intended to promote casual, spur of the moment, computing. For example, it has no power switch and instead automatically turns itself on when a person starts interacting and off after a person has finished interacting.
- *Constant connectivity.* The system assumes the palm-top unit is always connected to the network infrastructure.
- *Location reporting.* The location of each PARCTAB is always known to system software.

The system designers specifically avoided addressing certain issues, such as intermittent connectivity and disconnected operation, that would be of concern in a product. Restricting the system capabilities in this way let us conserve time and money without effecting the project's main goal, which is to investigate location-based and casual computing. The most significant advantage of assuming constant connectivity is that the device can use the computing power on the network. That is, instead of relying entirely on local processing the device interacts with programs running on a user's desktop system. An equivalent device with a desktop-speed processor would have been heavier, larger, and more costly to development.

6.2 Hardware Design

The PARCTAB mobile hardware (tab for short) is depicted in figure 6.1. Positioned over the display is a transparent touch sensitive panel that can be operated coarsely with a finger or more accurately with a passive stylus. Both the touch panel and the display have a resolution of 128x64 pixels. The grip part of the tab incorporates three finger-operated mechanical buttons that can be used individually or in chords. Finally the unit includes a piezo-electric speaker that permits a number of different tones to be generated by applications.

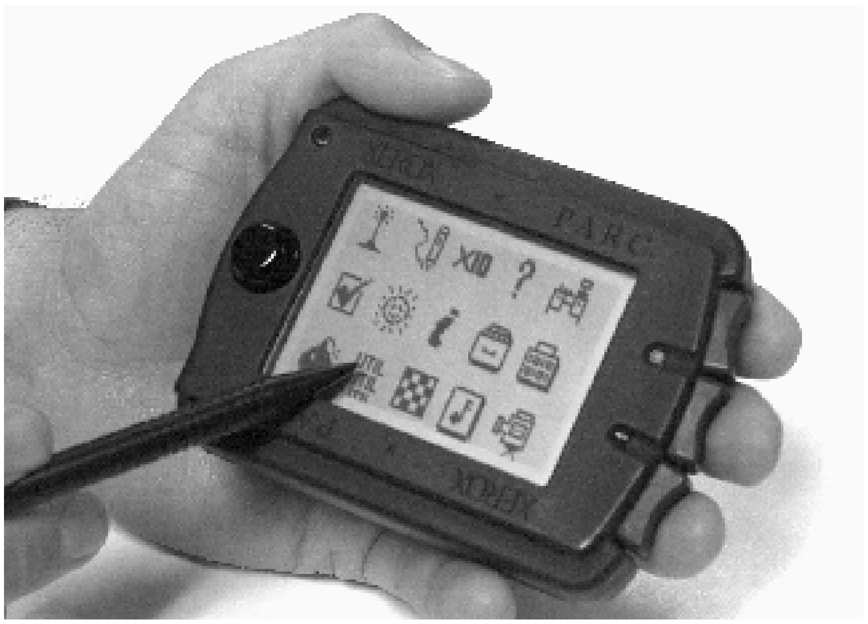


Figure 6.1: The PARCTAB Mobile Hardware

The tab communicates using infrared signals at a speed of 19.2 Kbaud. The spread of infrared emissions are contained by the walls of a room providing room-sized communication cells. The small cell size provides good aggregate bandwidth as long as few active units are in each cell. The room-sized cells employed by the communication network also generate location information used by higher level software.

Power management was an important consideration. For a compact and low power design we built the unit around a 12MHz Intel 8051 family 8-bit microcontroller with on-board EPROM, RAM and I/O ports. The tab can operate under nominal use for 10 minutes per hour, 8 hours per working day for about a week before needing to be recharged.

To foster casual use our design also paid special attention to ergonomic factors. The PARCTAB is designed into a custom, production-quality, plastic case with a removable belt clip that is about half the size of current commercial PDAs. The package is symmetric and can be used in either hand. When converting from left to right hand use, a setup command rotates the display and touch-screen coordinates by 180 degrees.

6.2.1 The Infrared Network

The tab infrared network is made up of small communication cells each being defined by the walls of a room surrounding a transceiver. Transceivers attach to nearby workstations which are then further connected to a LAN. A transceiver is a communication hub for a group of PARCTABs that are located in a particular cell. Typically its communication radius is about 20 feet, or less if limited by walls. The following tasks are carried out by the transceiver hardware:

- encoding and decoding infrared packets;
- buffering data;
- link-level protocol checks (format, checksum);
- providing a serial line interface;
- visual indication of communication status.

Line of sight between mobile tabs and a transceiver is not required for the PARCTAB network. This is because the hardware uses "diffuse infrared" signals that are emitted in many directions at once. These signals also reflect off of surfaces thereby spreading to all points in a room. The transceiver, for example, uses IR emitters placed at 15 degree intervals on a circular printed circuit board. The receiver consists of two detectors with a viewing angle of 360 degrees (figure 6.2). The transceiver is designed to be attached to a central point on the ceiling of a room, as this usually gives an unobscured communication path over the largest area.

Transceivers connect to workstations through a serial line and workstations are connected by a standard Ethernet. The approach of using an existing local area network and extending it to provide wireless nano-cellular communication is attractive because the cost of using existing LAN wiring is small. In addition, there already exists well established communication mechanisms between workstations in distributed systems. However, although most offices are equipped with at least one workstation having spare RS-232 ports, this approach requires nearby hosts for other areas less commonly equipped, such as meeting rooms.

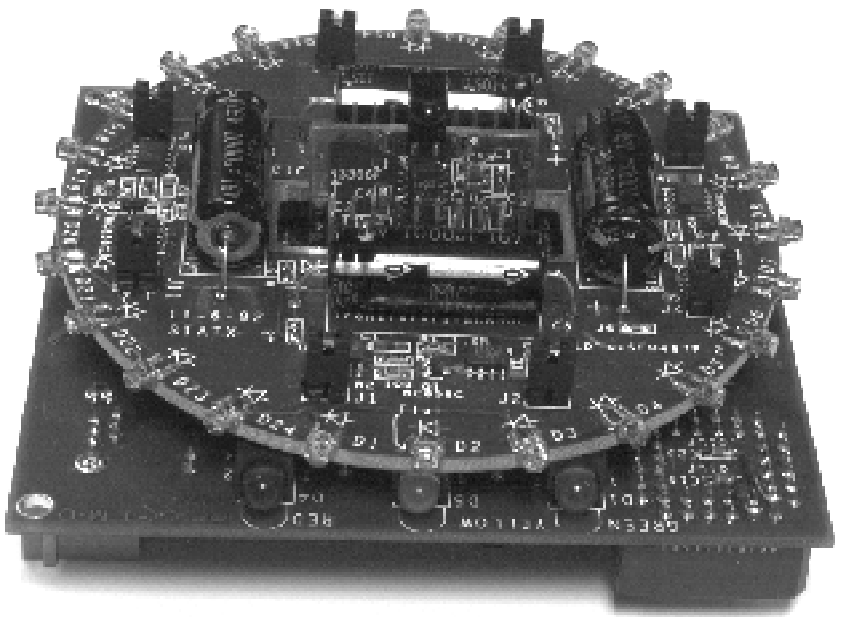


Figure 6.2: The PARCTAB Transceiver

6.3 System Architecture

The PARCTAB system design connects mobile hardware to workstation-based applications hiding the details of host mobility, message routing, and error recovery from applications. Furthermore, in order to experiment with context-aware applications, it was also a design goal to expose the PARCTAB's current location.

To meet these goals we use three types of system processes: the IR-gateway for managing infrared transceivers, the tab device agent for providing location-independent reliable communication, and the shell for starting and managing user applications. Figure 6.3 shows the relation between the PARCTAB, the transceiver hardware, the system processes, and user applications. Applications connect to a tab agent and are generally event driven much like X11 or Macintosh applications. The following sections describe each of the system components and concludes with a step by step example of the flow of information between applications and PARCTABs.

6.3.1 Infrared Gateways

The IR-gateway process controls one or more infrared transceivers connected to a workstation's serial ports. The gateway reads IR packets written by transceivers and forwards them to tab agents. In the reverse direction the IR-gateway receives packets from an agent over a LAN, encodes them for IR transmission, and writes them to the serial port where the transceiver broadcasts them over the IR medium in that cell.

When the IR-gateway receives an infrared packet it must determine the appropriate agent to forward it to. This is accomplished by using the packet's source address as a key in a name service lookup over all tab agent servers. Once the agent's communication end-point is obtained in this manner, the gateway stores it in a long-lived cache making name service accesses rare. In order to facilitate return communication from the agent back to the tab, the IR-gateway adds a return address (its own communication end-point) to all agent destined packets. The IR-gateway also adds a location identifier (LID) to each agent message. The LID, as presented in Section 5.4.2, is a short textual description for the location of the transceiver which applications use as a key into location databases and services.

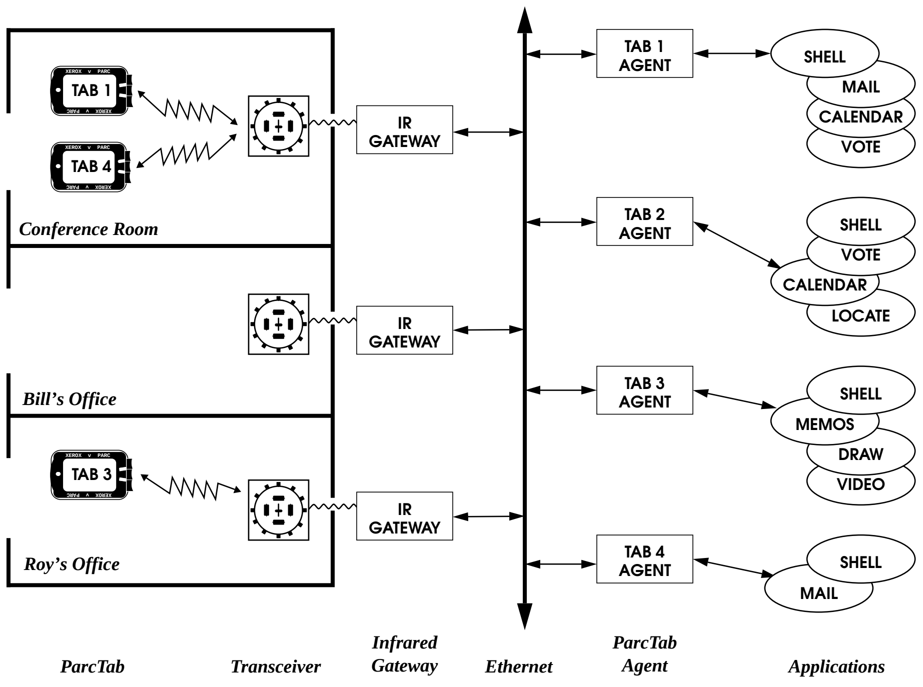


Figure 6.3: The PARCTAB System Architecture

In addition to its main function of forwarding packets between the local area and infrared networks, the IR-gateway performs configuration, error reporting, and error recovery functions. For example, if packets are sent too quickly to the transceiver, the hardware send buffer will overflow and the IR-gateway throttles back clients by increasing the request service time.

6.3.2 Tab Agent

For each PARCTAB there is exactly one agent process which provides a "switchboard" service linking applications to tabs. Agents perform three functions. They receive requests from applications to deliver packets to the mobile PARCTABs and in the reverse direction, forward messages (along with LIDs) from the tabs to applications. They also provide an authoritative source of tab information for context-aware applications, and third, they manage application communication channels. These are described in more detail below.

Messages sent to PARCTABs are either unreliable one-way datagrams providing low level location-independent access to the infrared network, or reliable two-way Remote Procedure Calls (RPC). Since the agent is an intermediary on all messages, it has the most complete information on the tab location. Even if the PARCTAB moves to a new cell, its periodic beacon identifier will ensure the location of the PARCTAB will soon be updated correctly. In the case of reliable RPC delivery, if a request or reply packet is lost or the location of a tab is temporarily unknown, the agent retries indefinitely using a backoff algorithm that takes into account whether the tab is sighted, busy executing another RPC request, or missing from the network.

In the other direction, messages from PARCTABs, mostly user interface generated events, are sent using a reliable link-layer protocol. If duplicate messages are received by the agent, either due to retransmissions or multiple cells picking up the signal, they are filtered using a sequence number contained in each message.

In addition to message delivery, the agent is the authorative source of information about the tab. In particular, the agent manages the Tab object shown in figure 6.4. The agent reports a tab's last known location and one of three states, "interactive" when it is being used and generating user events, “idle” when only beacons are being received, and “missing” when the tab is not being sighted by the agent. Since the PARCTAB is an “owned” rather than “public” device, the tab device agent sends updates of the Tab object to the user agent.

The tab agent also exports a dynamic environment containing general device state. This is used for debugging and logging purposes. Currently the additional state includes an object describing the application interacting with the tab. When the user switches from one application to another, the TabApplication object in the device agent’s environment is rewritten. In the current system a central logging server makes a connection to all agents and from then on receives a stream of location and application use updates which it writes to files for later analysis.

One early lesson that was learned is that the tab display is too small to support multiple interacting applications. So, although many applications may access the tab over time, there is only one “current application” at any one time which can send messages and receives events and associated LIDs. An application not in control of the tab is said to be *suspended*, and although it may run in the background, it may not perform user interaction operations until it is *resumed*. One exception to this rule lets suspended applications to asynchronously notify a user with an attention tune.

The third function of the agent is to manage application communication channels. To support suspending and resuming applications, the agent provides an RPC interface, called AppControl. Whereas all applications may suspend themselves, only a privileged “shell” application may control any application. An application must present the correct capability in order to perform privileged operations. The agent’s AppControl interface separates the mechanism of application management from the associated policy and user interface issues handled by shells.

Since the agent itself contains no tab-specific functions, different shells may be used to communicate with different devices. For example, we have also used the infrared network with HP100LX palmtop computers. The standard shell is described in the next section.

Communication to mobile hosts is achieved using tab agent processes. A competitive design for the PARCTAB network would be to implement Mobile IP [17]. This approach was not chosen in part because of the extra tab-resident software required, and also because the agent can be used to mediate a single communication stream, that is, the agent filters messages from non-current applications before they reach the wireless medium. In addition, our network needs to expose mobile host location, which is not currently a part of mobile IP proposals.

6.3.3 Example Communication Scenario

To explain the mechanism at work in more detail consider that a PARCTAB is located in a particular cell and has generated a button event in response to a user action. A transceiver nearby will pick up the event packet signal and transmit it over the serial connection. The IR-gateway process listening to the serial line reads the packet and uses the packet's source address to find the agent's communication end-point. The packet plus the transceiver's location identifier and the IR-gateways's network "return" address are then all sent off to the agent process.

Upon receiving the message, the agent filters duplicates, and then forwards it to the current application. In the application the event is decoded and causes a procedure call defined by the application developer which may then initiate a state change in the application.

In the reverse direction applications send requests to PARCTABs. In the following description an application has just received an event from a tab and in response it sends a request to the PARCTAB to update its display. A library procedure that an application designer links into the application is used to pack the appropriate display data into a request packet. When the application invokes this routine, in order to initiate the display update, an RPC request is made on the agent and blocks the application thread waiting for a reply. This request is then forwarded by the agent to the most recent IR-gateway that was in communication with it.

The IR-gateway encodes this packet for transmission to the transceiver over the serial link where it is broadcast over the IR medium. If the PARCTAB that has been addressed receives the packet, it will decode and execute the function code and then transmit a reply back to the IR-gateway that, in turn, will forward the packet to the correct agent using the mechanism described earlier.

When the PARCTAB user roams from one cell to another, pen and button events are picked up by a new transceiver and forwarded to the agent where they cause the last known location to be reset. The agent will then send future messages destined for the tab at the new transceiver location. For the case when the user moves to a new location yet does not access the tab, we rely on periodic beacons to inform the agent of the new location. Note that the tab beacons even when it is powered "off." The tab agent has a number of states and backoff strategies for managing the reliable delivery of RPC messages. One special situation it handles is when the agent receives a message from a new location during the middle of a retransmission delay, in which case it immediately resends its current message to the new transceiver location.

6.3.4 Shell

The tab shell provides a user interface for managing applications in much the same way as a conventional command interpreter. The shell is started by the device agent during initialization, and if the shell exits, it is automatically restarted. The main function of the shell is to accept a user's selection of an application. The shell provides this function by showing screens of bitmap icons and text buttons on the tab display. These represent programs and other screens. When a program button is pressed, the shell creates a new Unix process and uses the AppControl interface to authenticate it with the agent, instructing the agent to switch to the new application. In order to expedite these steps, applications generally suspend themselves instead of terminating the process when the user selects quit, allowing the shell to resume the application without creating a new Unix process.

When the application exits or suspends itself, the shell program is resumed. In the event an application locks up in some way, there is a special "agent escape" event that can be generated by a PARCTAB that forces the agent to suspend the current application and switch back to the shell.

The shell allows customization by the use of two types of initialization files. First, if the user wants to personalize the layout of their shell screens by augmenting the system default layout, they can specify additions and changes in a tabrc-personal file. More adventurous users may want to completely replace the screen layout in which case they create

Tabrc	→	Part*
Part	→	`(Initialize` Action*`)`
	→	`(Screens` Screen*`)`
Screen	→	`(`*label*: Widget*`)`
Widget	→	`(Text` *text x y invert*`)`
		`(TextButton` *text x y* Action`)`
		`(Bitmap` *bitmap-file x y*`)`
		`(BitmapButton` *bitmap-file x y* Action`)`
Action	→	`(Screen` *label*`)`
		`(Beep` *duration octave note ...*`)`
		`(Program` *program-args*`)`
		`(Load` *tabrc-file*`)`

Table 6.1: Tab Shell Initialization Grammar

a tabrc file. A picture of the default top-level shell screen can be seen in figure 6.1. The contents of the tabrc file define the buttons, bitmaps, text, and active areas that appear on the PARCTAB screen and make up the shell's user interface. The interface is structured around labeled screens. A screen contains text or bitmap decoration, and text or bitmap buttons. The buttons invoke built in actions: jumping to another screen, starting and resuming applications, and playing a tune over the PARCTAB speaker. The grammar for the file is shown in Table 6.1. It consists of two parts, a section that defines the screen structure seen on the tab, and a section for specifying startup actions, such as running programs. In this format, the star ("*") indicates zero or more occurances of the item.

6.3.5 Application Programming

Applications can be written in a scripting language suitable for small applications, or a high-level compiled language. The scripting approach is described in [31]. For greater flexibility over the user interface, applications are programmed using Modula-3, a relatively new language that includes a number of features that are valuable for building large systems. Existing applications programmed in some other language may be ported to access the tab through the use of the agent's RPC interface.

We implemented a class-based hierarchy of widgets, loosely modeled on the Trestle window toolkit [28], that provided routine components such as iconic and text buttons, scrollbars, bitmaps, text labels, scrollable text areas, and dialog boxes. Unlike a traditional window system, our widgets do no paint-clipping, since the very small screen generally precludes overlapping of widgets. This greatly simplified the implementation without burdening clients.

Some applications want to interact with several PARCTABs simultaneously. For example, a group of tabs could act as a shared drawing or notepad, displaying what's drawn on one tab to all the others in the group. To support such concurrent use of multiple tabs by a single application, we provided the TabGroup programming interface. A TabGroup is merely a group of tabs controlled by a single program. Tabs may join and leave the group at any time depending on the application requirements. Using TabGroup, the program can wait for all pending output to be delivered to all tabs in a group, synchronize on input events and other events, and detect tabs that have stopped responding to output for some reason. Using a single process to control a group of tabs with standard interfaces provided by the tab programming library is often easier than running a separate process for each tab and having the processes communicate by application-specific RPC interfaces.

Programmers generally construct their applications using an event-driven model and the widget library. Integrating asynchronous callbacks from other sources, such as the user agent, is thus simplified. To coordinate concurrency between asynchronous events from multiple sources (i.e., threads), we employ a UserEvent which can be sent by any thread in an application and serializes execution through a tab event thread.

6.4 Context-Aware Integration

The initial design of the PARCTAB system sent only location identifiers to applications. In this simple design the tab agent forwarded the tab's location to applications along with each user event. The widget library compared the location to the device's previous location and generated a pseudo locationChanged event before delivering the user event. This scheme was suitable for prototyping a number of applications that were con-

```
{{Name Tab:0.0.1}
 {UserName schilit}
 {state interactive}
 {Location LID:35-2-1-08}
 {when 775362216}}
```

Figure 6.4: Tab Object

cerned only with location. However, it was far from ideal. First, only the active application noticed location changes immediately, since the agent only sends events to one application at a time. This is a particular problem for programs that want to perform asynchronous notifications or actions. In addition, applications are not aware of the static or mobile objects that are co-located with them. Finally, applications only know a location identifier, they have no higher level notion of the geography they are situated in or the relation between one location and another.

The architecture presented in this thesis addresses these issues. In the PARCTAB context-aware system, the tab agent plays the role of a device agent. The system employs user agents and an active map in order to provide context-aware capabilities to applications. The remainder of this section describes how the tab device agent integrates with the other system components.

The tab agent sends location information to the current application, to the user agent, and also to a network-accessible dynamic environment. The information in a Tab object, maintained by the tab device agent, is shown in figure 6.4. The tab agent generates sightings when the tab moves to a new location or when it changes states. The tab may be in one of three states: interactive, non-interactive, and missing. An interactive state means the PARCTAB has been used by the user in the last 120 seconds, whereas a non-interactive sighting means the device is beaconing but not generating user interface events. Finally, a missing state means the device is not within range of the transceiver network. Transitions between the last two states are computed internally by the tab agent depending on the beacon rate of the device.

```
{{Name Host:fermius:schilit}
 {Location LID:35-2-1-01}
 {UserName schilit}
 {tty ttyp5}
 {login 775362216}
 {mesg y}
 {when 777578039}}
```

Figure 6.5: Host Agent Object

6.4.1 Host Agent

A second kind of device agent employed by this demonstration system is the host agent. This agent determines when interactions are occuring on a (generally stationary) host by monitoring the keyboard and mouse in a similar manner to a screen saver. When a transition between interactive and non-interactive occurs, the host agent generates a Host event which is sent to the associated user agent. The information is shown in figure 6.5. Included is the host's location, the user, when the user logged in, and when the user last interacted at the host. Additionally, the terminal device and an indication of whether the user is accepting messages on that device ("mesg") are included.

6.5 Applications

The examples of Chapter 2 present some possible applications for mobile distributed computing. This section considers four application domains that the PARCTAB is well suited to explore. These domains are: proximate selection; automatic reconfiguration; contextual information and commands; and context-triggered actions. For each application domain, we present a general description followed by a particular example in more detail including how a prototype application is implemented using the architecture of Chapter 5.

6.5.1 Proximate Selection

Proximate selection is a user interface technique where the located-objects that are nearby are emphasized or otherwise made easier to choose. In general, proximate selection involves entering a “locus” and a “selection.” However, of particular interest are user interfaces that automatically default the locus to the user’s current location.

There are at least three kinds of located-objects that are interesting to select using this technique. The first kind is computer input and output devices that require physical interaction. This includes printers, displays, speakers, facsimiles, video cameras, thermostats, and so on. Another example is people in the same room to whom you would like to “beam” a document. The second kind is situated non-physical objects and services that are routinely accessed from particular locations; for example, bank accounts, menus, and lists of instructions or regulations. The third kind is the set of places one wants to find out about: meeting rooms, offices, restaurants, and stores, or more generically, exits and entrances. Consider an electronic “yellow pages” directory that, instead of the subject divisions of information, sorts represented businesses according to their distance from the reader.

The context-aware infrastructure presented in this thesis, creates opportunities to explore issues in proximate selection. For example, location information can be used to weight the choices of printers that are nearby. Table 6.2 shows proximate selection dialogs for printers using three columns: the name of the printer, the location, and a distance from the user. One interface issue beyond the scope of this thesis, is how to navigate dialogs that contain this additional location information. For example, should dialogs use the familiar alphabetical ordering by name or should they be ordered by location. Shown here are (a) alphabetically ordering by name; (b) ordered by proximity; (c) alphabetical with nearby printers emphasized; (d) alphabetical with selections scaled by proximity, something like a perspective view.

Another area of investigation created by this type of interface is how to manage the UI bandwidth requirements. Presenting information that changes, either due to the user moving or the contents of the dialog changing (e.g., other people moving) will cause update network traffic. One approach is to view location information with more or less precision based on the situation. The interfaces in Table 6.2 are fine-grained – the distance column

Name	Room	Distance
caps	35-2-2-00	200ft
claudia	35-2-1-08	30ft
perfector	35-2-3-01	20ft
snoball	35-2-1-03	100ft

(a)

Distance	Name	Room
20ft	perfector	35-2-3-01
30ft	claudia	35-2-1-08
100ft	snoball	35-2-1-03
200ft	caps	35-2-2-00

(b)

Name	Room	Distance
caps	35-2-2-00	200ft
claudia	**35-2-1-08**	**30ft**
perfector	**35-2-3-01**	**20ft**
snoball	35-2-1-03	100ft

(c)

Name	Room	Distance
caps	35-2-2-00	200ft
claudia	35-2-1-08	30ft
perfector	35-2-3-01	20ft
snoball	35-2-1-03	100ft

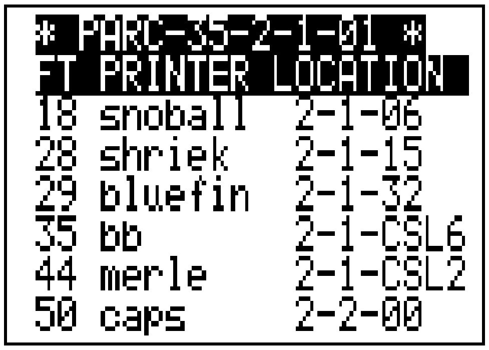
(d)

Table 6.2: UI Techniques for Proximate Selection

requires updating for each change in location of the locus. In contrast a coarser-grained view of the same information might show a zone rather than a distance. Driving around town with such a dialog would, for example, change only when the viewer, or the objects in the selection dialog, crossed the city limits.

A proximate selection dialog built for the PARCTAB shows how different parts of our system interact. Since the tab is limited in resolution the interface uses a UI similar to Figure 6.2 (b), shown in Figure 6.6. The top line gives the user their current location and the second line is a key providing column labels for the rest of the screen. The column labeled "LOCATION" uses a suffix of the printer's location in order to save screen space. The printer names are presented sorted by distance from the user, and can be selected with the pen. This dialog is created using a process described below.

The procedure shown in Table 6.3 generates the information for the dialog. It takes as input the device's current location ("locus") and queries the active map for located objects of the type "Printer." With this collection of objects it then finds the distance from the locus parameter to each printer object and returns a sorted list for display. For the duration that the dialog is active, the dialog monitors the Tab object on which it is being run,



```
* PARC-35-2-1-01 *
FT PRINTER LOCATION
18 snoball  2-1-06
28 shriek   2-1-15
29 bluefin  2-1-32
35 bb       2-1-CSL6
44 merle    2-1-CSL2
50 caps     2-2-00
```

Figure 6.6: Proximate Printer Selection Dialog

and when that changes location, the display needs to be redrawn.

The process described above can be made more efficient by presenting information for areas smaller than the entire active map region. For example, the UI could present the printers on the current floor of the building. In addition, distances can be cached to reduce active map interactions.

6.5.2 Automatic Contextual Reconfiguration

Reconfiguration is the process of adding new components, removing existing components or altering the connections between components. Typical components and connections are servers and their communication channels to clients. However, reconfigurable components may also include loadable device drivers, program modules, hardware elements, etc. In the case of context-aware systems, the interesting aspect is how context of use might bring about different system configurations and what these adaptions are.

When a group of people is in one place, the people can easily share the physical objects in that place. For example, people in a meeting room share a table that might hold scattered papers, and a whiteboard with diagrams. To promote similar sharing, we wrote a multi-user drawing program for the PARCTAB which provides a workspace for each room, a sort of virtual whiteboard. Entering a room causes an automatic binding between the mo-

```
procedure proximate_printers_dialog(region,locus) returns(dialog)
begin
    Return in dialog a table of name, location, distance. The
    input is the LID locus and region name.
    printers ← activemap_query(region,"{{Name Printer:*}}",0,"");
    foreach p in printers
        printer_loc ← keyget(p,"Location");
        distance,step ← activemap_distance(locus,printer_loc);
        dialog[p].name ← keyget(p,"Name");
        dialog[p].location ← keyget(p,"LID");
        dialog[p].distance ← distance;
    end
    return sort(dialog);
end
```

Table 6.3: Computing content of a proximate printers dialog

bile host and the room's virtual whiteboard. In this way people in the same room can easily collaborate using the virtual whiteboard. Moving to a different room brings up a different drawing surface. Automatic reconfiguration creates the illusion of accessing this virtual object as if it were physical. Reconfiguration can also be based on information in addition to location, for example, the people present in a room. If a project group is meeting then the project whiteboard is active.

The PARCTAB whiteboard is implemented as a single back-end server for each instance of a whiteboard instance. Tabs sharing a whiteboard connect to the process. In this way many tab agents are all communicating with one process so sharing of the whiteboard contents occurs using internal data structures instead of network messages. To coordinate multiple tabs within a single application we use the TabGroup data structure, which allows communicating strokes and other information from one tab thread to another without coordination problems.

The context-based reconfiguration is created using registrations and a "connector" user interface. Network registrations for room-based whiteboards are stored in the active map in the room which they are "in." Project or workgroup whiteboards simply register with the discover protocol for region-wide naming. A user then accesses these registered software processes using the connector UI. The connector monitors the co-located objects for either room-based whiteboards or multiple project members at the current location. When either situation occurs the connector adds a button for accessing the particular whiteboard.

Using context as well as location makes virtual whiteboards more powerful than their physical analogues since a virtual whiteboard can persist from meeting to meeting, and can follow participants from room to room.

6.5.3 Contextual Information and Commands

People's actions can often be predicted by their situation. There are certain things we do when in the library, kitchen, or office. Contextual information and commands aim to exploit this fact. Queries on contextual information can produce different results according to the context in which they are issued. Similarly, context can parameterize "contextual commands," for example, the print command might, by default, print to the nearest printer.

The location browser is a PARCTAB application that views a "location-based filesystem." The dialog is shown in Figure 6.7. When moving from room to room, the browser changes the displayed information to match the viewer's location. In addition you can move the browser locus manually. Our system is set up so that when in an office we see the occupant's Unix finger plan file; when in the public area of our lab we see a general description of the research carried out in that area; and when near the kitchen we see directions for making coffee and finding supplies.

This interface provides a contextual naming system. It is most useful when people either do not know the name of something, or it is a long procedure to discover the name for something. For example, when in the same room as a document scanner it can be used to to find "proximate help." Another example is a meeting room that uses an electronic calendar for bookings. Compared to searching a file system for the electronic calendar, it may be easier to find by "visiting" (either physically or virtually) the meeting room location and invoking the "reservation" button.

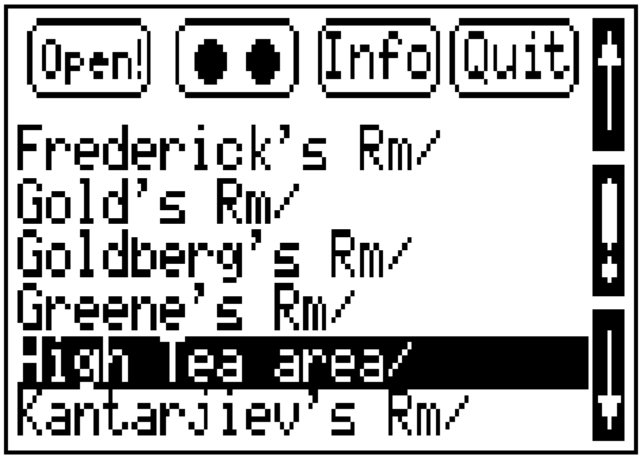
Aside from displaying data files parameterized by the viewer's location, the location browser also runs programs. Contextual commands of this kind may take two forms. First, the appearance of the command itself might change depending on context of use. For example, when in the library the button to invoke a card catalogue database might appear prominently whereas it is normally hidden. Second, a command may appear the same but produce parameterized results. For example the location browser's user interface presents a migrate button that appears identical from room to room, but that causes the user's windows to migrate to different host displays depending on the location in which it is invoked.

A simple implementation of a location-based file system uses the device's location as an index into a directory hierarchy. The application monitoring the execution context notices when the device location changes and switches to display the new directory. The problem with this approach is that it relies on a static filesystem structure and so does not capture the movement of other objects, particularly mobile hosts and people from one place to another. Because of this, objects changing rooms will not be seen. A more flexible approach is to use the active map as a point of indirection, the place where all information is found.

Since information is associated with other objects such as rooms, people, printers, etc., it is useful to define a global attribute that the location browser understands. This attribute may be a part of any active map object and serves as a pointer to textual information. Specifically it points at a Unix file, directory, or program. For a room the active map contains a pointer to a directory holding files of information for the room. Similarly a User object may contain a pointer to a file describing information about the user. The function of the browser is to monitor the co-located objects and display those objects having this browseable information.

6.5.4 Context-Triggered Actions

Context-triggered actions are simple IF-THEN rules used to specify how context-aware systems should adapt. Information about context-of-use in a condition clause triggers consequent commands; something like living in a rule-based expert system! A number of applications can be organized in this way. The category of context-aware software is similar to contextual information and commands, except that context-triggered actions are invoked



Frederick's Rm/
Gold's Rm/
Goldberg's Rm/
Greene's Rm/
High Tea area/
Kantarjiev's Rm/

Figure 6.7: Location Based File System

automatically according to previously specified rules.

One context-triggered program we have experimented with performs "contextual reminders." Contextual reminders pop-up a message on the PARCTAB screen when a particular situation occurs. The situation can include when, where, who and what is with you. For example, the "next time in the library" or the "next time I see Marvin," or "when I'm back at my desk."

In order to permit complicated conditions as reminders, the TCL scripting language [30] is used. The basic framework invokes a user's "reminder procedures" on each change to user environment and set of co-located objects. The reminder procedure has access to the current change in context as well as many other kinds of information. For example, the procedures can examine the current application context, co-located objects, or other global state. The user's reminder script can invoke other TCL procedures to query the active map or run Unix processes. The reminders framework provides support procedures shown in table 6.4

Category	Example
Date and time	`clock after April 15`
	`clock before December 31 11:59pm`
	`clock between 10 and 12noon`
	`clock dayofweek tuesday`
Location	`in-room 35-2-2-00`
Co-location	`with {{Name User:adams}}`
	`with {{Name Host:*} {features Color}}`
Path	`path 35-2-2-00 35-2-2-10`
Distance	`distance 35-2-2-10 35-2-1-01`

Table 6.4: Predicates for Context-Triggered Actions

6.6 Summary and Conclusions

This chapter presented a mobile distributed computing system based on a hand-held wireless computer. The PARCTAB hardware supplies room level location information and allows two way 19.2kb infrared communication. The original system design didn't permit general context-aware applications. This chapter describes how the PARCTAB system was integrated with the architecture presented in the previous chapters.

In this combined system, the tab agent plays the role of a device agent. Secondary location information is available from a host device agent which monitors keyboard activity on workstations. These components provide a tab-object and host-object to a user agent. The user agent in turn updates the active map with the user-object and exports the user environment. Applications interact with the active map to find out about their changing context of use.

This chapter presented four categories of applications that have been prototyped in this system. The first category is proximate selection, an example of which is a dialog that sorts a list of printers by their distance to the user. The second is contextual reconfiguration, for example a shared drawing program that shows different "virtual whiteboards" depending on the location of use, and the people nearby. The third is contextual information and commands, an example of which is a file browser that always shows information about nearby locations and objects. The last category is context-triggered actions, which was demonstrated with a contextual-reminders program. This program pops up a message on the tab device when user scripted conditions about the co-located objects and user environment are met.

Chapter 7

Performance Evaluation & Issues

This chapter describes performance evaluation and explores a number of open issues. The chapter begins with a description of a framework for performance measurement which is followed by some measurements of the active map service. The next section discusses dynamic application design and the use of shared memory to implement efficient access to asynchronous dynamic environment updates in single-threaded applications. Following this is a look at a number of issues including multicast assumptions, non-multicast wireless links, security, and other styles of efficient data dissemination.

7.1 Mobile User Simulation

One difficulty with evaluating the performance of a context-aware system is that system load depends on the activities and movement of people. Because of this, any real-world technique to induce a load is going to be difficult to reproduce. In addition, system measurements during actual activity must run for long periods in order to capture the infrequent high load conditions that are important to a performance evaluation.

These evaluation problems can be overcome by playing back "canned" activity through the system while it is offline to other inputs, and then measuring the result. In the case of the shared active map service, the play back data consists of location sightings for users. One choice for such a workload generator is to record user sightings from the PARCTAB (or badge) system over time and then play them back. The problem with this is that it requires a sizable user community already be in place. Playing back duplicates, or multi-day recordings all at once is one way to deal with this. However, another problem is that our current systems miss sightings when users walking quickly pass base-stations before the beacon interval expires. Accurate sightings are desirable in order to represent future location-reporting technology and the increased load a larger number of sightings incur. In any event, the direct recording approach still makes it difficult to examine various loads that may not occur in that particular office environment.

The approach we took is to build an artificial workload generator, `SimMob`, for producing sets of sightings that were then be played back to an active map server. `SimMob` can easily generate artificial sightings for hundreds of users as they travel around regions with hundreds of locations.

The workload generator is intended to model a PARCTAB, active badge, or similar beaconing mobile location system. In these systems periodic beacons are produced by mobile devices and received by fixed location receivers. The beacon rate of active badges is 15 seconds and for PARCTABs it is 30 seconds. More frequent beacons allow more accurate tracking because the system will report a room entrance on average after $1/2$ the beacon interval. However, the drawback of frequent beacons is that they quickly wear down batteries and also take up bandwidth which is especially a problem when multiple beacons are at the same location.

The workload generator uses a discrete event simulation of user movement consisting of multiple "people" moving from vertex to vertex on a graph representation for a building. Running a simulation consists of a number of steps. First the user "modes," and "mode transitions" are defined. Then a floor plan for the region is entered. These inputs produce a collection of sightings records over the simulation period from the `SimMob` program. Finally this information is fed into playback agents running on the network. The rest of this section describes the workload generator in more detail and describes these steps.

The probability matrix that defines the likelihood of a user moving from a particular location to another location is called a "mode." For example, a mode might specify that the probability of moving from room 35-2-1-01 to room 35-2-1-48 is 80% with all other transitions taking up the remaining 20% probability in equal portions. In this evaluation, two standard types of workloads were used, the "meeting" mode and "normal" mode. The meeting workload has people start in individual locations and at some point converge on a single location over a specified period of time, typically about 3 to 5 minutes. The normal workload has users spend most of the time in their offices and every once in a while venture out to another location.

Modes provide a good deal of flexibility in defining a person's movements. In practice, however, people change their behavior during the day. In order to model this, the workload generator understands "mode transitions." These are simply a way to specify that a new mode description should be used after a particular time during the simulation. These transitions apply to individual users, so for example, a person might start in "work mode" and switch to "lunch mode" at noon.

The final input for a simulation is data about the space in which the simulation occurs. The workload generator is given a map of all the locations in a region and the distances between adjacent locations. This is used to determine paths and distances for moves: the main simulation loop chooses a destination based on the user's mode and takes this destination and expands it to the sequence of steps (i.e., sightings) along the shortest path. These steps are then emitted at times based on the step's distance and a global travel-rate.

The main loop of the simulation is shown in Figure 7.1. This procedure is run once for each user and the resulting events are sorted and output into a single event stream. It starts by setting the initial mode and location. Then it enters a loop where it outputs sighting events. First it checks for mode transitions to see if the user is switching to a different move probability matrix. Then it randomly chooses a new destination based on the probabilities from the mode. The sightings along the path are output, and the loop continues after a Poisson distributed sleep interval is added to the simulator's clock.

7.2 Active Map Performance

The test-bed system was designed to tell us two different things: how well the active map server we built works and what kind of loads might occur under various different user movement scenarios. In particular, we were interested in characterizing the behavior of the system under "normal" load circumstances as well as when a meeting of many people

```
procedure MakeMoves(user: User; stopTime : Time.T)
begin
    (*Starting location, mode, and rate is the user's defaults *)
    loc ← user.home;
    mode ← user.initMode;
    decisionRate ← user.initDecisionRate;
    (* Start the run by randomly sleeping so not all users are lockstep. *)
    clock ← user.startTime + Random.Subrange(random,0,decisionRate);
    while clock.seconds ≤ stopTime.seconds do
        IF the next mode transition has passed then
            Copy the mode and decisionRate information.
            Reset the clock to be near the time the transition occured.
        end;
        (* Generate the move to the destination. *)
        path ← mode.random(loc, user.home, mode, decisionRate);
        if loc # path.dst then
            (* Expand the path and output steps along it. *)
            prevLoc ← path.src;
            while path.path # NIL do
                with step = List.Pop(path.path),
                        travelTime = step.d * walkRate DO
                    (*  Increment the clock according to the travel time. *)
                    INC(clock.seconds, travelTime);
                    Output the step and the time at which it occured.
                    prevLoc ← step.dest;
                end;
            end;
            prevLoc ← loc;
            loc ← path.dst;
        end;
        (*  add in user "think" time till next decision. *)
        clock ← Poisson.poidev(decisionRate,seed);
    END;
    Set user.moves and return.
END MakeMoves;
```

Figure 7.1: Mobile user workload generator main loop for a single user.

took place.

Figure 7.2 shows some benchmark numbers for the AMS test-bed. These measurements were made on a Sun SPARCstation 2 client communicating using Sun RPC over an Ethernet to a SPARCstation 2 server machine running the active map server. The line marked “query match” shows the time required to match a simple query against a collection of AMS objects with a single object being successfully matched and returned to the query client. The slope shows that each additional object matched contributed about 66 microseconds to the service time. The line marked “query return” shows the cost of returning objects whose size is around 400-500 bytes to the client. Objects this size are realistic in our system because descriptions consist of several attributes, although more compact representations are possible. The slope of the line indicates about 2.5 milliseconds are needed for each additional object returned. The line marked “subscriber update” shows how AMS service time for unicast subscriber updates depends on the number of messages that must be sent to disseminate information about an update event.

In order to understand how the system might behave under different usage scenarios we used the artificial workload generator, `SimMob`, for producing sets of sightings that can then be played back to an active map server. The AMS was tested using the “meeting” and “normal” workloads and the results are described below.

To see how large a normal work community might be handled by the AMS we ran a normal workload against a unicast version of the AMS with 1000 users in the system. As input to the workload generator we used an 80% probability of staying in the office location which resulted in 12% of the community being mobile at any given time, on average. This translated into mobile users being sighted an average of every 8.9 seconds, resulting in a computed average of 600 moves per minute overall. The generated workload was executed on 18 SPARCstation hosts each with between 55-56 client processes, along with hosts for the server and a monitor program. Running the workload produced encounters with other people occurring on about 40% of the moves. No regional queries were included in this workload; users monitored only their own locations. For the unicast design we observed an average delay of 23ms in receiving updates—about twice the unloaded case of 11ms—implying that the AMS should easily be able to handle the generated workload of a thousand users under “normal work” conditions.

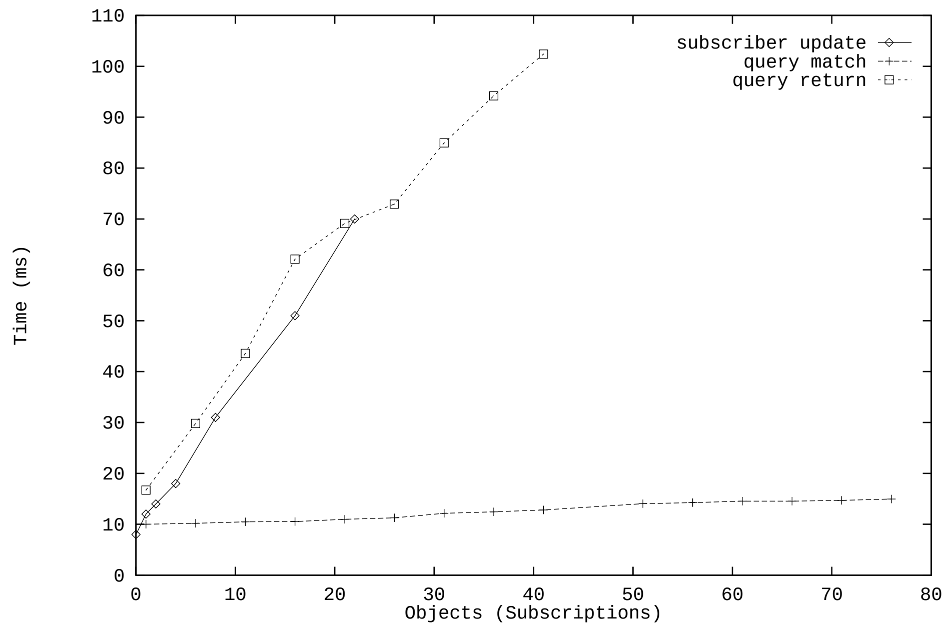


Figure 7.2: Benchmark numbers for an active map server running on a SUN SPARCstation 2.

To confirm the fact that situations like meetings will quickly overload a unicast design we also ran several meeting workloads against the unicast version of the AMS. The intent was to explore one possible “worst case” scenerio, not necessarily the average case. As above, the simulation was run with 1000 users in the system. All users started in their office, and over a five minute period converged on an “auditorium” location. The event stream was played back with each user subscribing to the objects at their location. With this subscription, when a user enters a new location, two things happen. First, a message with objects for all the users already at the location is sent to the new user. And second, an update is sent to each of the residents of the location informing them of the new user. This procedure causes $O(n^2)$ messages when n users converge on a single location.

The result observed was that after approximately 50 users enter the auditorium, the delay in reporting updates starts to increase substantially. This is shown in Figure 7.3. In the scenerio presented there are two potential points of overload: the server and the network. Some computation can tell us which resource overloaded first. The rate of people entering the location was observed to be 50-55 in a 30 second period, or 2500-3025 messages in 30 seconds. This means the server was handling updates at the rate of 80-100 msgs/second or 10-12 ms/msg. An otherwise unloaded server can sustain updates at about 11 ms/msg (see Figure 7.2) so this rate borders the point that overloads the server’s update processing. Another thing to note is that the network bandwidth during this period was about around 256-320 Kbps, which is a fraction of a 10 MB ethernet, but will overload slower, 19.2 Kbps links. In contrast, a single multicast channel version of the AMS seems able to support meetings of up to about 530 participants.

The overall performance of a version of the AMS that incorporates the multiple multicast schemes described here is determined primarily by the average number of update messages that the server ends up sending out per update event. That is, the cost of managing the data structures needed for these schemes is small compared to the cost of actually sending update messages out on the network[1]. When users employ standard queries the way that client applications try to encourage them to, the average number of messages that the

[1] If update query set sizes were to start numbering in the many tens to hundreds then the cost of sorting them into canonical order and storing them would start to be significant. However, the design is based on the assumption that many different queries will not match any given update event.

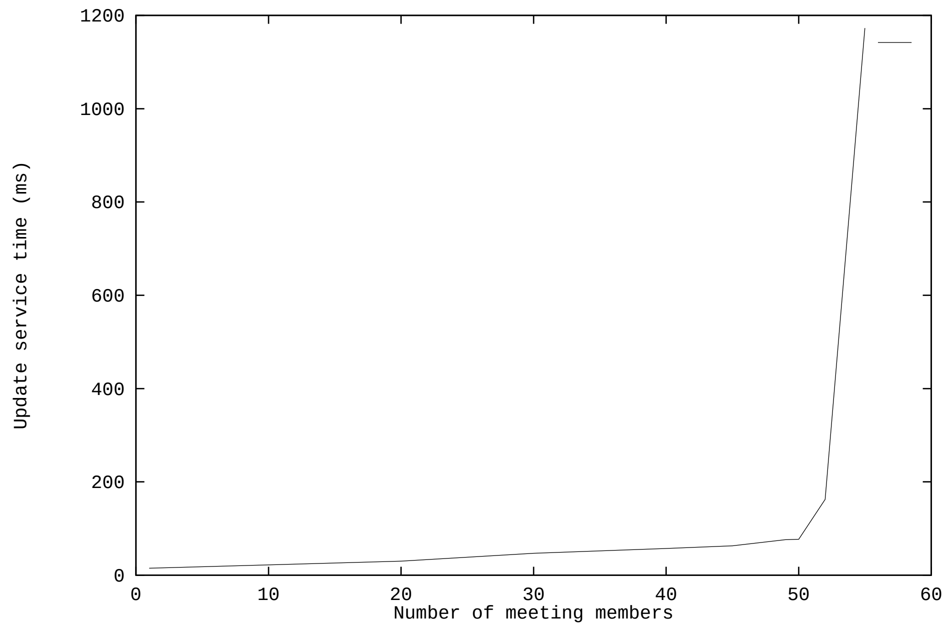


Figure 7.3: Average update time for clients monitoring a meeting as a function of meeting size.

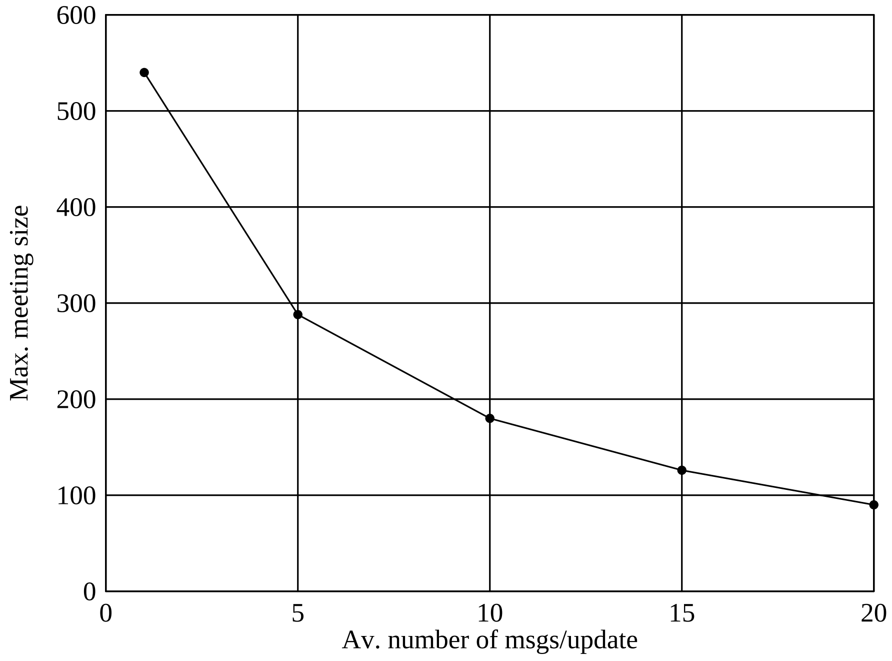


Figure 7.4: Maximum size of meeting that can be handled as a function of the average number of messages that must be sent out per meeting update event.

AMS must send out for an update event is kept quite low. Indeed, for most update events, a single message suffices to update all regional subscriptions. The same is true for subscriptions to meeting locations. Unfortunately, because the set of context-aware applications is not well explored, only production use of this system by real users who have had a chance to write new applications will tell us how many non-standard subscription queries will occur in practice.

Figure 7.4 illustrates how the maximum size of meetings handled by the AMS server depends on the average number of messages that must be sent out per update event. The values were obtained by disabling the use of multicast channels for identical clients sets and generating artificial meeting workloads with different numbers of distinct subscription queries. In this graph, the maximum meeting size is the point at which the AMS is receiving updates quicker than they can be disseminated.

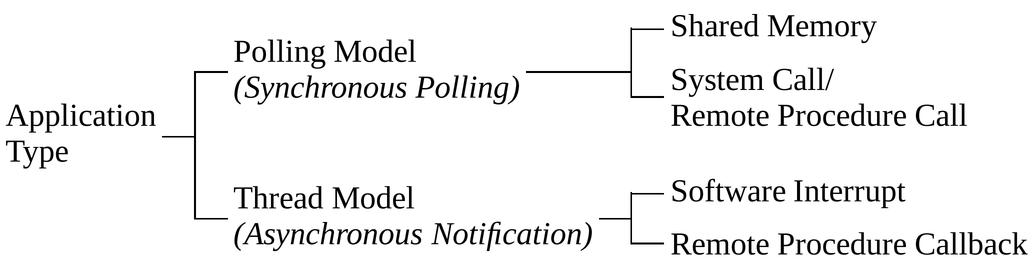


Figure 7.5: Dynamic Environment Interaction

7.3 Issues in Dynamic Application Design

The ease by which dynamic environment notifications are introduced into applications depends on the application's existing framework. For example, applications lacking an event or threaded execution model cannot easily field asynchronous notifications. To address this there is a mechanism for applications designed without concurrency in mind. The technique uses shared memory between an application and a local environment server "manager" to support low-overhead polling.

The issue of how context is introduced into applications is orthogonal to the dynamic environment interface. It is a matter of whether the interface is directly available to the application or whether a local manager process veneer stands in between. Figure 7.5 categorizes the software mechanisms available to Unix applications for inter-process communication. These reflect the design choices for supporting a Unix-based dynamic environment that relies on communicating values over the network or between processes. The figure is divided into techniques for multi-threaded applications capable of concurrency, and applications built with a polling model and not capable of concurrency. Single-threaded clients can access the dynamic environment by polling for changed values from shared memory, by invoking a system call (e.g., to read a file) or by issuing a remote procedure call. Concurrency-capable applications can be notified asynchronously by software interrupt, remote procedure callback, or similarly from input available on a non-blocking file descriptor.

generation
generation-check
data

Table 7.1: Shared Memory Data Structure

The same issues that effect the network usage of a dynamic environment interface also relate to an application level interface. Asynchronous notification has the advantage that the application is immediately informed of changed values and no cycles are wasted on polling. This is essentially a direct application of the dynamic environment interface.

Polling environment objects is another approach for integrating context information into existing software. The goal is to provide an efficient facility in the Unix development environment. The design uses a separate manager-process that is capable of asynchronous update along with shared memory [14, 48] to store the dynamic environment. This facility provides low-overhead polling since applications only reference memory instead of performing a system call to find out if the environment has changed. This technique is from Lamport [24] and allows concurrent access to data structures without requiring locks.

This technique operates as follows: an application maps a memory segment containing the dynamic environment into its address space and periodically polls the contents for changes. A separate process is used to receive dynamic environment updates from the network and write them into the shared memory. The costs of shared memory procedures on a Sun Microsystems SPARCstation 2 machine are shown in table 7.2.

In order to avoid system calls in managing the concurrency of the writer and reader process the procedure uses the data structure shown in Figure 7.1. The writer and reader coordinate by referencing a two-part generation counter and in the opposite order. The pseudo code for this is shown in Figure 7.6. This is intended for uniprocessor architectures, and may be incorrect for some multiprocessor memory models where special concurrency instructions would need to be performed. The `poll` procedure optimistically reads the data until it finds that the two counters are identical. The `put` procedure makes the values in the two-part counter identical only after the data is stored. While the multi-word data is in the

object size (bytes)	poll time (microseconds)
0	31
100	65
200	74
1000	150

Table 7.2: Shared Memory Polling Performance

process of being stored, the counters differ.

7.4 Further Issues

This section presents issues that have not been covered elsewhere in the dissertation. We start by exploring the assumption that applications can use large numbers of multicast groups, and then look at how unicast wireless links, such as are available through cellular telephony, might work. Covered next are the the issues surrounding privacy of active map information, followed by caching and low level alternatives to information dissemination.

7.4.1 Local-Scope Versus Internet Multicast Groups

The assumption that applications can use large numbers of multicast groups is currently not reasonable in a general Internet setting: dynamic global address assignment is difficult and significant numbers of Internet-wide multicast groups would overload the routing tables. Fortunately, most DE server clients are likely to reside on hosts local to the DE server's region. Therefore by limiting the range of messages sent over a multicast group—for example, to a campus—the same group can be "recycled" and used from one campus to the next.

Currently we limit the range of IP multicast datagrams by setting their "time-to-live" (TTL) parameter: multicast datagrams are forwarded from one subnet to another only

```
procedure poll_dynamic_environment() returns(data)
begin
    (* Return if there is no update since the previous call. *)
    if (generation_poll == shared->generation) then
        return NULL;
    else
        Copy the data making sure no updates occur during copy.
        repeat
            check ← shared->generation_check;
            copy ← shared->data;
        until (check == shared->generation);
        Store the generation number for next time and return data.
        generation_poll ← check;
        return copy;
    end
end


procedure put(data)
begin
    (* Store the update data using guard counters. *)
    shared->generation++;
    shared->data ← data;
    shared->generation_check++;
end
```

Figure 7.6: Lock Free Read/Write

if their TTL is above a certain threshold. A drawback with this approach is that DE server regions must correspond to the TTL regions defined by the multicast network routers that will be used (or that the network routers have their TTL region definitions be appropriately tuned).

Because some clients may reside on remote hosts, an DE server cannot rely solely on local-scope multicast groups to disseminate heavily subscribed updates. In many cases, simply unicasting the relevant updates to the one or two remote clients interested in them is sufficient (in addition to disseminating them via local-scope multicasts).

If each DE server obtains one or a few Internet multicast group addresses for itself then these can be used to alleviate load in cases where a significant number of remote clients exist and are interested in heavily subscribed updates. This works well to the extent that fewer Internet multicast groups are needed than the DE server has obtained for itself. As the demand for additional Internet multicast groups increases, the DE server can control its own load by reusing each Internet multicast group for multiple multiply-subscribed queries or recurring query sets and forcing remote clients to assume additional filtering overhead. Note that the use of Internet multicast groups can be kept completely separate from the DE server's handling of local clients via local multicast groups (since clients' "remoteness" can be readily determined). Hence the problems of dealing with remote clients need not affect the local system.

7.4.2 Dealing With Unicast Wireless Links

Many wireless communication technologies—such as current cellular telephone systems—do not offer multicast support. An agent-based implementation for routing packets to mobile hosts also makes the provision of multicast difficult since agents communicate with their mobile hosts in an essentially unicast-based fashion.

A hybrid scheme can help somewhat for systems that do not support multicast semantics across their wireless communications links: the DE server still employs multicast to reduce its own load; however, communications are directed at agent processes on LAN-connected hosts that convert the multicast traffic to unicast and forward it to the actual client applications running on mobile hosts. Unfortunately this does not alleviate the load problems that occur when the same update message gets sent separately to many clients residing in the same slow, wireless communications cell.

This problem can be partly alleviated by moving the relevant client applications partly or wholly to stationary hosts that are directly connected to a multicast-capable network. If a suitable local host is available then this may not be difficult and may even simplify matters: complicated filtering needed to cull a small, manageable amount of information from the update traffic generated during overload situations can be done on a faster stationary machine instead of on a slower, resource and power-conscious, portable machine. The final information results, which will presumably be much smaller in size than the original update traffic, can then be sent to application "front-ends" that reside on users' portable machines.

In some cases, however, a suitable local host may not be available. While a mobile user is in his "home" DE server region he may have access to a trusted workstation or server machine on which to safely run his applications. If the user visits a remote region he is unlikely to have access to a suitable local machine there and would be forced to run his applications remotely on a machine in his home region. Depending on the applications and networks involved, the additional delays this would impose may or may not be an issue. In general, we conclude that, while unicast wireless links can be partially dealt with—especially when clients have access to local trusted hosts—the availability of multicast on wireless links is far preferable as a basis for an DE system.

7.4.3 Privacy

One weakness of the design presented in this dissertation is that it uses a central active map service without addressing the resulting security and privacy issues. The user of this system has only the coarse control over whether they publish or don't publish a user-object for themselves in the active map. Some of these issues are explored in [39].

A limited amount of privacy can be obtained when the user agent process publishes the user-object in the active map. Instead of updating the user-object to be in a leaf container, such as a room, the agent can specify some higher level container, such as a building. This means that users can export some location information without having their fine grained location known.

Another way to provide privacy is for users to encrypt their user-object information. Each time the user agent changes the object location, it would be desirable for the encrypted object to change appearance so it is not recognizable. This requires a method for distributing keys between parties that are interested in sharing their location information.

Privacy can also be provided by partitioning active map servers into administrative organizations in which users are usually willing to export their location information. For example, a university department might be such an organization. This division, in conjunction with a way to deny service to users unless they have prior authorization might add an acceptable level of privacy.

Yet another technique is to support a secure active map service. The AMS would only disseminate information according to access lists or protection groups. Presumably such a technique would also use data encryption. Since multicast is used for dissemination, however, outgoing information must be decryptable by a group of people. Such a mechanism is beyond the scope of this dissertation.

7.4.4 Exploiting Constant Data

The design in this thesis did not describe caching issues. Value caching of active map geographic information is a simple addition to the user agent fragment. Caching in this case involves storing previous results to path and distance calls and returning those instead of going over the network to the active map each time.

General caching of objects however is not as easy. It would be desirable for active map servers to be able to send out the minimum information on each update and assume that the clients held the full attributes for the object in their cache. So for example, updates might only include the changed attributes, usually a short object identifier, and the location identifier. However, in a worst case, cache misses could generate more traffic than was saved by using multi-multicast dissemination in the first place. Two approaches are for the active map to track what is in a clients cache and for clients to cooperate through a multicast channel, so they don't all send cache miss messages at once. Both approaches add a fair degree of complexity to the design, and represent future work.

7.4.5 Network Dissemination

The dissemination techniques described in chapter 4 assume an infrastructure similar to the Internet. In particular, it assumes that network topology is not known and that user code may not be run in routers. If either of these is feasible other improvements describe below can be made.

First, if topology is known, that is, if it is known which hosts are on which wireless subnets, then the active map service can further optimize the assignment of multicast groups. For example, when switching unicast recipients to a multicast group the decision does not need to be made unless the threshold q is reached within a single wireless cell. Similarly, the other unicast recipients need not be immediately switched over to multicast since they do not immediately benefit.

Another advantage can be gained if user code can be run in routers, in this case the wireless base-stations. This approach is somewhat similar to the Information Bus [29]. In the case of the dynamic environment subscriptions, if the base-stations could run user queries then they could filter out all unnecessary information and only broadcast the required object updates. In this design the routers now take a large portion of the AMS workload. However, this involves changing the existing network infrastructure, and more importantly letting users run their software on routers, which raises security and resource usage issues.

7.5 Summary

This chapter described how a system based on user activity can be measured by using a workload generator that plays back user sightings obtained from a mobile user simulation. Systems measurements are necessary to show the scalability of an active map server. This is important because the decision to partition the service into individual servers was based on the reasoning that “large enough” regions could be used. It was expected that region sizes over which one naturally does queries, i.e., a building, could be supported by a single server.

The measured performance confirms that this is the case. The meeting simulator shows that the AMS supports up to 530 participants all rapidly converging on a single location, and all subscribing to changes in their location. This is a worst case scenario of a class or large meeting, and in practice much lighter loads occur because of the spread of activity around a workplace. Measurements also showed that the multi-multicast approach has a big advantage over simply using unicast, which only supports meetings of 55 or so clients under similar conditions. Note that meetings of 530 are not a strict limit of the server, but rather a point where other load management techniques, such as batching and bandwidth limiting take over.

The chapter continued by exploring a number of issues that have not yet been covered. First the cost of integrating dynamic environments into existing application that aren't oriented towards asynchronous updates. The idea presented is to use shared memory with lock-free access which allows for a very efficient implementation of polling.

Another issue was whether it is a reasonable assumption to use a large number of multicast groups. It was explained that since the scope of groups can be limited, they can be reused over wide areas. A related conclusion was that multi-point wireless access supporting multicast is preferable to point-to-point connections for our design. This is particularly relevant since wireless communication is currently dominated by point-to-point cellular telephony. The section continued with a description of how the use of multicast makes it more difficult to implement both privacy and client caching. A final issue concluded that if more control over low level networking is available, more efficient dissemination decisions would be possible at the AMS, and perhaps in other parts of the network.

Chapter 8

Conclusion

In this last chapter we summarize the contributions of the thesis and then briefly discuss future directions.

8.1 Contributions

The purpose of this research has been to show a system architecture that facilitates the creation of context-aware software. We have developed three major ideas to realize our goals.

The first idea, dynamic environments, provides a uniform data oriented communication model particularly well suited for notifying wireless clients of changing infrastructure information. In order to cope with low speed communication links, clients pre-specify their information needs and their bandwidth limits. In order to handle intermittent connectivity clients rely on latency feedback as opposed to the classic two-state notion of connectivity. This technique lets applications control connectivity notices by asking for values for how far out of date each high level piece of data may become. A side benefit is that this cumulative “advice” from applications gives the communication layer a good idea of how hard it must try to keep in touch with network servers. In this model, fault tolerance is improved by automatically reconnecting and regenerating state between clients and servers when servers are restarted. Finally, scaling is provided by simply having a number of distinct servers each handling different kinds of information.

The second idea is a context-aware system organization, specifically the kinds of dynamic environment servers that are appropriate and how they interact. One component is an active map, that helps users navigate a mobile environment, serves as a meeting place for context-enabled software, and provides a key for finding out about the surrounding world. We found that geographic information in terms of path distance and spatial containment turns out to capture many common spatial concepts. Another component is a user agent for managing user information including a shared user environment supporting efficient recustomization of applications and easy sharing of multiple devices by applications. We've found that managing the execution context and the intermittently connected nature of components is facilitated if a part of the user agent resides on each host where applications are run. The final component required for a minimal context-aware architecture are device-agents that contribute information about devices, including their location. One thing learned in this design is that the information needs for people, devices, variables, and locations can all share the same basic format and data oriented interface.

The third idea is a technique for load management using dynamic assignment of multiple multicast groups. We've found that neither simple unicast nor broadcast works well for our expected system configurations and information demands. To handle this we developed the idea of letting servers dictate to clients, as often as necessary, a different multicast channel over which updates arrive. This creates a mechanism for dynamic distribution of message load where servers use multiple multicast groups and unicast channels in order to avoid overloading clients and communication channels. A few assumptions about our system allow us to come up with a relatively inexpensive policy for load management. We've found that the general problem of minimizing overhead by assigning multicast groups to clients according to their information needs is computationally expensive. However, by examining our specific problem domain we arrived at two techniques that simplify assignment, namely to group clients using identical queries and sets of clients receiving the same message traffic. This idea pointed out that multicast is an important tool for information dissemination to wireless hosts.

8.2 Future Work

8.2.1 Applications

The numerous applications types which this work enables need to be investigated more rigorously. These application classes will drive the requirements of future context-aware systems. It is not yet clear whether the applications we have chosen to aid in the design will prove to be the most compelling.

8.2.2 Internet Integration

Since the application domains we've outlined are, for the most part, local area, we've concentrated on a local area design. To support remote access, the active map needs to be extended to enable efficient remote access of information. We've outlined one plan for doing this.

One assumption our system made was that wireless IP multicast would be generally available. This is currently not the case and future research needs to address it. Additionally, some work is necessary to influence mobile-IP standards to expose location information when available.

8.2.3 Varying Scales & Venues

A future area of research is to explore how to combine Euclidean distance with the graph of nodes distance already present in the active map. The reasons for this is that in the future sub-room location information will become available and it would be useful, for example, to know head orientation in order to determine where a person is looking. Similarly in different venues, such as campus quadrangles and big open spaces, multiple paths exist, and some kind of Euclidean measure may prove simpler.

One problem with the active map is that it requires an administrator to set up a description of the physical spaces. A future research area is to explore how to add a learning component to the system so that it can be trained to know about locations and their relations as people use them.

8.3 Conclusion

This dissertation has described an architecture for context-aware computing consisting of device agents, user agents, and an active map service. Each of these services is based on a dynamic environment model that is well suited for communicating changing values to hosts over wireless links. The architecture compliments work done on the PARCTAB system, which, along with a number of context-aware applications, is presented as a demonstration of these concepts.

References

[1] Arup Acharya and B. R. Badrinath. Delivering multicast messages in networks with mobile hosts. In *The 13th International Conference on Distributed Computing Systems*, pages 292–299, May 1993.

[2] Norman Adams, Rich Gold, Bill N. Schilit, Michael Tso, and Roy Want. An infrared network for mobile computers. In *Proceedings USENIX Symposium on Mobile & Location-independent Computing*, pages 41–52. USENIX Association, August 1993.

[3] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[4] Apple Computer. *Inside Macintosh Volume I*, September 1987.

[5] Nicholas Carriero and David Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, September 1989.

[6] Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.

[7] Nicholas Carriero and David Gelernter. Coordination languages and their significance. *Communications of the ACM*, 35(2):96–107, February 1992.

[8] David D. Clark. The Structuring of Systems Using Upcalls. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, Shark Is., WA, 1985.

[9] John R. Corbin. *The Art of Distributed Applications*. Springer-Verlag, New York, 1990.

[10] S. Deering. Host extensions for IP multicasting. Request for Comments (Standard) RFC 1112, Internet Engineering Task Force, August 1989.

[11] Stephen E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[12] Alan Demers, Scott Elrod, Christopher Kantarjiev, and Edward Richley. A nanocellular local area network using near-field RF coupling. In *Proceedings of Virginia Tech's Fourth Symposium on Wireless Personal Communications*, pages 10.1–10.16, June 1994.

[13] Ron Frederick. Experiences with real-time software video compression. In *Sixth International Workshop on Packet Video*, Portland, OR, September 26-27 1994.

[14] R. A. Gingell, J. P. Moran, and W. A. Shannon. Virtual memory architecture in SunOS. In *Proceedings Summer 1987 USENIX Technical Conference*, pages 81–94. USENIX Association, June 1987.

[15] David Goldberg and Michael Tso. How to program networked portable computers. In *Proceedings Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 30–33. IEEE, October 1993.

[16] Andy Harter and Andy Hopper. A distributed location system for the active office. *IEEE Network*, pages 62–70, January/February 1994.

[17] J. Ioannidis, D. Duchamp, and G. Q. Maguire Jr. and S. Deering. Protocols for supporting mobile IP hosts. Technical report, Mobile Hosts Working Group, June 1992.

[18] J. Ioannidis, D. Duchamp, and G.Q. Maguire Jr. IP-based protocols for mobile internetworking. In *Proceedings SIGCOMM '91*, pages 235–245. ACM, September 1991.

[19] Thomas A. Stansell Jr. Civil GPS from a future perspective. *Proceedings of the IEEE*, 71(10):1187–1192, October 1983.

[20] Christopher A. Kantarjiev, Alan Demers, Ron Frederick, Robert T. Krivacic, and Mark Weiser. Experiences with X in a wireless environment. In *Proceedings USENIX Symposium on Mobile & Location-Independent Computing*, pages 117–128. USENIX Association, August 1993.

[21] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. In *Proceedings Thirteenth ACM Symposium on Operating System Principles*, pages 213–225. ACM, October 1991.

[22] S.R. Kleiman. Vnodes: an architecture for multiple file types in Sun UNIX. In *Summer Conference Proceedings, Atlanta 1986*. USENIX Association, 1986.

[23] M.G. Lamming and W.M. Newman. Activity-based information retrieval: Technology in support of personal memory. In *Proceedings of IFIP-92*. IFIP Congress, 1992.

[24] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM*, 20(11):806–811, November 1977.

[25] B.W. Lampson, M. Paul, and H.J. Siegert, editors. *Distributed Systems*. Springer-Verlag, New York, 1988.

[26] Gilberto Matos and James Purtilo. Reconfiguration of hierarchical tuple-spaces: experiments with Linda-Polylith. Technical Report CS-TR-3153, University of Maryland Institute for Advanced Computer Studies, College Park, MD 20742, October 1993.

[27] Sape M. Mullendar, editor. *Distributed Systems*. Addison-Wesley, New York, 1989.

[28] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[29] Brian Oki, Manfred Pfluegl, Alex Siegel, and Dale Skeen. The information bus – an architecture for extensible distributed systems. In *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, pages 58–68, Asheville, NC, December 1993.

[30] John K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the 1990 Winter USENIX Conference*, pages 133–146, 1990.

[31] Karin Petersen. Tcl/tk for a personal digital assistant. In *USENIX Symposium on Very High Level Languages*, pages 41–54. USENIX Association, 1994.

[32] Steven P. Reiss. Interacting in the FIELD environment. *Software Practice and Experience*, 20(S1):89–115, June 1990.

[33] Robert W. Scheifler and James Gettys. *X Window System*. Digital Press, third edition, 1992.

[34] Bill N. Schilit, Norman Adams, Rich Gold, Michael Tso, and Roy Want. The PARCTAB mobile computing system. In *Proceedings Fourth Workshop on Workstation Operating Systems (WWOS-IV)*, pages 34–39. IEEE, October 1993.

[35] Bill N. Schilit, Norman Adams, and Roy Want. Context-aware computing applications. In *Proceedings Workshop on Mobile Computing Systems and Applications*, pages 85–90. IEEE, December 1994.

[36] Bill N. Schilit and Daniel Duchamp. Adaptive remote paging for mobile computers. Technical Report CUCS-004-91, Columbia University Computer Science Department, February 1991.

[37] Leonard Schuchman, Bryant D. Elrod, and A.J. Van Dierendonck. Applicability of an augmented GPS for navigation in the national airspace system. *Proceedings of the IEEE*, 77(11):1709–1727, November 1989.

[38] John Shirley, Wei Hu, and David Magid. *Guide to Writing DCE Applications*. O'Reilly & Associates, Inc., 1994.

[39] Mike Spreitzer and Marvin Theimer. Architectural considerations for scalable, secure, mobile computing with location information. In *Proceedings of the 14th International Conference on Distributed Computing Systems*, pages 29–38. IEEE, June 1994.

[40] D.B. Terry, M. Painter, D. Riggle, and S. Zhou. The Berkeley Internet Name Domain server. In *Proceedings of USENIX Association Summer Conference*, pages 23–31. USENIX Association, 1984.

[41] The ToolTalk service. A SunSoft White Paper. Revision 01, June 1991, SunSoft, Inc. 2550 Garcia Avenue, Mountain View, CA 94043.

[42] Michael Tso. Using property specifications to achieve graceful disconnected operation in an intermittent mobile computing environment. Technical Report CSL-93-8, Xerox Palo Alto Research Center, June 1993.

[43] Roy Want and Andy Hopper. Active badges and personal interactive computing objects. *IEEE Transactions on Consumer Electronics*, 38(1):10–20, February 1992.

[44] Roy Want, Andy Hopper, Veronica Falcao, and Jonathan Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[45] Roy Want, Bill Schilit, Norman Adams, Rich Gold, Karin Petersen, John Ellis, David Goldberg, and Mark Weiser. The PARCTAB ubiquitous computing experiment. Technical Report CSL-95-1, Xerox Palo Alto Research Center, March 1995.

[46] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, September 1991.

[47] Mark Weiser. Some computer science issues in ubiquitous computing. *Communications of the ACM*, 36(7):74–83, July 1993. In Special Issue on Computer-Augmented Environments.

[48] M. Young, A. Tevanian, R. Rashid, D. Golub, J. Eppinger, J. Chew, W. Bolosky, D. Black, and R. Baron. The duality of memory and communication in the implementation of a multiprocessor operating system. *Proceedings of the 11th Symposium on Operating Systems Principles*, pages 63–76, November 1987.