Ubiquitous Computing:
Extending Access To Mobile Data

A Thesis
Presented to
The Academic Faculty

by

Michael David Pinkerton

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Computer Science

Georgia Institute of Technology
May 1997

Thesis Approval Page

DEDICATIONS

The most important dedication: to Nicole, my wife-to-be, for sticking by me and being so understanding when I was frustrated and grumpy.

I'd really like to thank my advisor, Gregory Abowd, for giving such a wonderful opportunity (and a chance to play with cool toys) even though I was only a Masters student. Gregory never once laughed at my ideas, and stuck up for me when I needed it. Of course, if it were up to him, this project would have been called "Newton's A-Go-Go." We'll try to ignore that.

I'd also like to the entire Future Computing Environments group (I can't name you all) for providing such a fertile playground in which to develop such new and fascinating ideas. My work would have been useless without applications to existing projects. I'm glad I could do my part in furthering their research in ubiquitous computing.

An important, but often forgotten, group are the tireless tech support folks. Without the help of the Newton DTS group, this project never would have been completed. Thank you all!

Finally, I must thank Jimi Hendrix, The Black Crowes, and The Who for providing hours of listening enjoyment as I hacked away into the early morning hours.

TABLE OF CONTENTS

LIST OF FIGURES

SUMMARY

We live in a world where fully featured mobile devices (PDA's, etc.) are gaining wider acceptance and usage. As more and more information is collected and stored on these devices, there becomes a greater need for both users and developers to be able to easily access and work with this information, either from desktop machines or from mobile devices. Current solutions to this problem are either cumbersome for users or restrictive to developers.

There are two goals of this work:

1) Provide an infrastructure and real-world applications for integrating mobile information into a desktop environment. This integration should be seamless, requiring minimal deviation from how users currently interact with their desktop machines.

2) Allow the mobile devices themselves to collaborate and share information with others. Each device will be a first-class client in the system, not just a passive information repository.

CHAPTER I

INTRODUCTION

1.1 Thesis Statement

We live in a world where fully-featured mobile devices (PDA's, pagers, cell phones, etc.) are gaining wider acceptance and usage. As more and more information is collected and stored on these devices, there becomes a greater need for both users and developers to be able to easily access and work with this information. Current solutions to this problem are either cumbersome for users or restrictive to developers.

This thesis attempts to solve some of these problems through a combination of desktop-based applications built upon a new infrastructure which supports easy information exchange with mobile devices. The specific infrastructure we provide as part of this thesis is the LlamaShare environment, the goal of which is to enable the rapid development of desktop-based applications which take advantage of mobile information, whether to provide new interfaces to users that streamline the complex interaction models required to utilize mobile information in common tasks, or to extend the ability to existing applications to mix mobile information with local and remote information.

It does not take hours of expensive user testing to notice the problems that exist with using information from the more popular PDA's on a desktop-based computer. The PDA users in our group (both Newton and Pilot) constantly vocalize their desire to have access to phone numbers or notes located on their PDA when using their desktop machine. The most prevalent solution, copying the entire contents of the PDA to the workstation, greatly restricts a user's ability to make changes at either location. Furthermore, this method pushes the burden onto the user, who is the least suited to handle such a complicated process. For mobile devices to gain a wider acceptance, the interface to mobile information must be seamless, as average users have little patience for intricate steps and convoluted interactions.

Our own projects in the Future Computing Environments (FCE) group illustrate the problems inherent in developing applications which utilize mobile information. Most (if not all) have been severely limited by the difficulties of trying to access and use the information collected on a variety of mobile platforms. For example, our PDA-based CyberGuide project [1, 2, 3] allows users to access information about their environment, but the information is static and there is no way for users to communicate contextual information such as positioning. Even retrieving the information collected on the devices proved time consuming and complicated, requiring custom client/server software to be written on both the desktop and the PDA. As there was no infrastructure available to build upon, each project made its own attempt with very little success in each case, leaving us with a variety of custom solutions which could not be applied to existing or future research projects, and a crowd of frustrated developers.

LlamaShare provides a general architecture to address both of these problems, but first its probably best to more closely examine the problems of users and developers in detail.

1.2 Problems With Current Solutions

From a user's perspective, the problems with mobile devices go far beyond just having **access** to information. While getting the information to a user's desktop machine has already been elegantly solved, actually **integrating** that information into user tasks (such as inserting mobile information into a document) requires a series of long and complicated steps which have little to do with the task for which the user needs the information. Serious effort has gone into the "synchronization" approach to accessing mobile information from the desktop world. When the user wants to access any data on their mobile device, they must explicitly go to a special "docking" program which downloads the information from the mobile unit to the computer, hopefully in a format which desktop applications can understand. This is the approach taken by the PalmComputing Pilot [4] and the Newton Connection Utilities [5, 6] from Apple Computer.

Synchronization is perfectly acceptable when the user's task only requires access to mobile information, such as synchronizing a desktop calendar with the calendar on the mobile unit. However, synchronization falls short when the task becomes more complicated. For example, it does not begin to address how a user would integrate a note stored on a mobile device into a letter they are writing. Even this simple task requires a much richer infrastructure which goes beyond simply providing access to mobile information.

To demonstrate, here is the sequence of steps required to integrate mobile information into a common task such as writing a letter:

1. The user realizes they want to use some mobile data to aid in creating their document in application "A"
2. User mentally locates the data, and realizes it is on a mobile device
3. Thinks about what program is needed to access the data (the docking application, "B")
4. Searches out and locates that application (this may take several steps by itself)
5. Goes through the steps of synchronizing the mobile device with the desktop (there could be many)

6. Thinks about what application is needed to view the uploaded data on the desktop, application “C” (almost certainly not the same the original application, “A”).
7. Searches out and locates that application (again, multiple steps)
8. Scans through all of the uploaded data to find the correct entry
9. Thinks about how to integrate this data into the document
10. Integrates it into application “A”, if and only if the data is in a format that “A” accepts.

This long and complicated process distracts the user from their current task by leading them on a wild goose chase through three different applications and many more tedious steps. Moreover, once they reach the final step, they may not even be able to integrate the data because it is not in a format the target application understands. Finally, the user now has two copies of their data. If they change either one, they must re-synchronize or risk encountering out-of-date information on either their desktop computer or the PDA.

This thesis seeks out a user interface which streamlines this interaction model of integrating mobile information to as few steps as possible:

1. The user realizes they want to use some data to aid in creating their document
2. Identifies the desired piece of data (either by issuing a query or by creating a physical marker on the desktop)
3. Drags a representation of the information into the document and drops it at the desired location

Developers run into similar roadblocks when trying to write client/server style applications which access and manipulate mobile information. The three main problem areas are:

1. Language/platform restrictions

   Libraries exist for each mobile device to handle transferring information between the PDA and the desktop, but they may only be available for certain platforms (usually Mac and Windows) and certain languages (C or C++). This seriously restricts writing applications in, say, Java on a UNIX machine.

2. Limited connectivity options

   Applications on the desktop are normally limited to communicating with devices that are either directly cabled to the same machine (serial) or on some private local network (AppleTalk). This restricts which machines have access to information on mobile devices to where the physical device is connected, and almost totally rules out connectivity from the Internet.

3. No infrastructure for mobile groupware apps

   PDA's are fertile ground for groupware applications (sharing positioning information is just one simple example), but there is almost no infrastructure available to allow the sharing of information among multiple mobile devices. Developers are forced to write their own from scratch each time.

When trying to develop an application which manipulates mobile information, these limitations can range from minor annoyances to show-stoppers. Our own efforts in the FCE group have been hampered by all three of these at one time or another. The harder it is for developers to write applications for mobile devices, the fewer applications will exist, which is a shame considering the sheer number of useful ideas which arise after even simple brainstorming. LlamaShare was developed to address these issues as well.

1.3 Contributions

In light of all of these problems, this thesis provides solutions not present in other systems. Here are the major points that LlamaShare addresses:

• Develop in any language, on any desktop platform, on any machine on the Internet

Applications on the desktop which access and manipulate mobile information can be written on any platform in any language. In addition, the use of TCP/IP as the communication layer opens up much more than the platform. Desktop-based client applications can now run on any machine on the Internet and be able to access mobile devices half a world away as if they were on the same local network.

• Access to multiple mobile devices

By using a desktop-based server as a contact point, clients have access to not just a single mobile device, but to any mobile device connected to that server. Clients can now query multiple devices for information without the requirement that any of them are in the same physical location.

• Database for collaborative use by desktop and mobile devices

The LlamaShare architecture provides a database which can be used to collaborate among multiple mobile devices, as well as with desktop applications. These data stores, called “Global Soups,” provide a single location where devices can collect and aggregate information collected individually. It also provides a single location where information intended for all devices in the environment can be deposited, either by a mobile device or a desktop application.

- Users should work with mobile information the same way they work with other information.

  No one knows yet the “right” way to serve mobile information to a user, so LlamaShare presents two different approaches which streamline the user interface for integrating mobile information into user tasks. One leverages Apple’s Cyberdog [7] technology to allow users to directly organize and manipulate physical representations of mobile data on the desktop intermixed with their Internet information. The other leverages CyberDesk [8], a research project from our own FCE group, which allows users access to a nebulous pool of information comprised of local, Internet, and mobile data. In both cases, mobile data is directly manipulated using the same metaphors with which users already are comfortable.

1.4 Overview of Remainder of Paper

Chapter 2 presents related work in the fields of mobile and ubiquitous computing and demonstrates how each project addresses particular aspects of the mobile computing. Furthermore, it delineates where LlamaShare differs from past work when concepts presented by this thesis overlap with prior research.

Chapter 3 discusses several applications which utilize the LlamaShare infrastructure and begin to address some of the user interface issues presented earlier in this chapter. There are four different applications which have been developed, two for desktop workstations, and two collaborative applications based on the Apple MessagePad.

Chapter 4 delves into details about the underlying infrastructure, describing each of the components, as well as the protocols between them.

Chapter 5 discusses major design decisions and why they were made, as well as limitations of the infrastructure and applications built on top of it. It also goes into detail about many of the technical hurdles overcome during the development of the LlamaShare infrastructure.

Chapter 6 comments on future directions and improvements for both the LlamaShare infrastructure and applications.

CHAPTER II

RELATED WORK

The key areas in which the work presented in this thesis stands apart from currently existing research and products are:

- Connectivity requirements
- Platform assumptions
- Application-specific effort
- Data integration

The following sections describe existing projects in each of these three areas and how LlamaShare differs.

2.1 Connectivity

Mobile devices are inherently mobile, thus access to information becomes complicated by the fact that these devices can be disconnected from the environment for long periods of time. As a result, there has been a large body of research focused on information access in a disconnected, or loosely connected, environment. This thesis takes a different approach, assuming that mobile devices will be constantly connected. Judging by the growth of the wireless community, in five years constant connectivity may be the rule instead of the exception.

Coda [9, 10, 11], from CMU, is a UNIX file system for mobile workstations based on the Andrew File System (AFS). Assuming that mobile workstations will be disconnected for periods of time, Coda transparently handles the consistency and replication issues, hiding them from the applications and the user. Of course, constant connectivity is desired when it can be achieved, so Coda provides an application-transparent adaptation model for maintaining continuous connectivity while migrating from one networking environment to another. For example, a laptop user might move from a connected environment in their office to a cellular connection on the road without any interruptions in service. In this way, Coda-based systems can function in both connected and disconnected environments.

Bayou [12, 13], from Xerox PARC, attempts to address several of the issues in a disconnected environment as this thesis does in a connected one by providing a mobile computing environment that includes portable machines with less than ideal connectivity. Both Bayou and LlamaShare provide an infrastructure for mobile databases on which to build a collaborative infrastructure. Bayou creates “databases” of information which can be shared easily and handle the consistency problems inherent with disconnected computing, using techniques such as replication, propagation of updates, and conflict resolution. The “database” concept is very similar to LlamaShare’s “Global Soup” and the lightweight server is analogous to the server which runs on the mobile device to serve its information to desktop workstations.

Wit II [14, 15], from the University of Washington, also addresses the fragility of mobile connections by focusing on the two most restrictive resources in the mobile environment: network bandwidth and local storage. By using techniques such as caching and prefetching, Wit II trades off increased local storage against reduced bandwidth requirements and user-perceived latency. Uniquely, Wit II provides an application framework in which applications can supply information such as object relationships and data types. As a result, the system can increase the effectiveness of its optimization techniques by prefetching related information to enable further operations after the device has been disconnected from the environment.

Unlike the previous systems which focused on connectivity limitations of mobile devices, ParcTab [16, 17, 18] from Xerox PARC ignores many of the problems of disconnected and unreliable communication by assuming a consistently available IR network. ParcTab uses custom hardware for both the mobile devices and the IR transceivers to provide a reliable and uninterrupted service to each tab. Similarly, LlamaShare uses high frequency RF via Ethernet transceivers and PCMCIA cards to establish its fully connected environment.

2.2 Platform Assumptions

This thesis presents an infrastructure which lays the groundwork for access to information on heterogeneous mobile devices that are not based on common desktop operating systems. Due to hardware constraints, mostly the lack of networking capability in commercially available devices, LlamaShare currently can only access information from the Apple MessagePad. However, once devices such as the Pilot can more easily exist on a network, the infrastructure can be expanded handle other devices. The Future Work chapter discusses this in detail.

Coda (and Odyssey, described below) are only available for devices running UNIX [11], which translates directly into laptops. Coda is typical of the research in disconnected file sharing to come out of the OS community in that it does not address other mobile devices, such as the entire PDA market, which are more portable and have longer battery life. Other projects such as FACE [19] from Princeton, the work done by Tait and Duchamp from Columbia [20], and Ficus [21] from UCLA all focus exclusively on laptops and UNIX-based machines.

Trying bridge the gap to a wider variety of mobile devices, Bayou allows information to be stored at locations other than the typical UNIX-based server. Any device, even a PDA, may carry a "lightweight" server which makes information available to any other device with which it has a connection [12]. While promising, very little of the work has actually been completed. Some simple client libraries exist, but only for Solaris, and nothing runs on any commercially available PDA's [13].

Both ParcTab and Wit II address the domain of non-UNIX mobile devices, but do so by using proprietary hardware and communications infrastructure built at Xerox PARC. In contrast, this thesis provides an infrastructure using off-the-shelf devices and components (such as the Apple MessagePad and NetWave Ethernet Access Points). Moreover, LlamaShare's devices perform most of their own computations, in contrast to the ParcTab infrastructure which emphasizes communication over local processing [18]. As a result, each tab is driven by applications running on a remote workstation and stores no information locally.

2.3 Application-Specific Effort

Different systems have different philosophies about the extent to which applications should be customized to take advantage of the features of the infrastructure. As described in Chapter 5 (Design Decisions), our infrastructure requires that applications be explicitly written to access and manipulate mobile information. Most of the burden of accessing mobile information is hidden from developers within stubs, but the use of such stubs is required. As a result, off-the-shelf applications not written to communicate with our infrastructure cannot participate.

While Coda makes no explicit demands on a particular application, Odyssey [10] explores cases where the application is designed to be aware of the loosely connected nature of the information being accessed. As a result, the application can adapt to rapidly varying connectivity parameters such as bandwidth and network quality. For example, a video application can react to dips in bandwidth by displaying a lower fidelity version of the movie, possibly without color or sound. Furthermore, the application can disclose the relationships between groups of files to the Odyssey infrastructure. Such a disclosure can then be utilized to provide strong hints about future accesses which can be exploited for prefetching.

Wit II uses a very similar technique to improve application performance in a constantly changing network environment. However, while Odyssey uses a mechanism based on groups of files, Wit II uses application-provided information such as object relationships and data types. Given this information, the system can create “object graphs” which links related objects together (for prefetching), similar to intermediate representations used in compilers [15]. As a result, the system can increase the effectiveness of its optimization techniques by utilizing such application information.

Both of these systems demonstrate that application-specific effort can be beneficial to the overall effectiveness of the system and the usefulness of the applications.

2.4 Data Integration

In the beginning, PDA-style mobile devices could do little more than store calendar and contact information and had no connectivity options whatsoever. These early devices were frustrating because users could not share information between their desktop applications and the PDA. As a result, users were forced into entering information twice: once on their computer, and again on their handheld.

Eventually, these devices gained the ability to connect to heterogeneous devices (such as desktop computers), allowing users to share information between their two main information repositories. Devices like the HP OmniGo [22] could only import and export tab or comma delimited text files and DBase III databases. To accomplish something like coordinating calendar information on both system required that the desktop calendar application be able to import and export text files. The user was then forced to manually perform the extra step of importing or exporting the information.

More recently, mobile devices such as the PalmComputing Pilot, the Psion Series 3c [23], and the Apple MessagePad have alleviated this step by automatically reading from and writing to the native data formats of many popular desktop applications. The connection utilities are not much more than automated translators with the ability to merge data from two sources into one which contains the most up to date information in each (see Figure 1). This works quite well for a highly specific task such as synchronizing a calendar, but integrating generic information still requires the user to manually locate, scan, parse, and integrate tab-delimited data. Furthermore, if information changes on the desktop, the devices must be manually resynchronized, else the PDA will not contain a consistent view of the information.

To address this problem, the Revelar Connection Utility (RCU) [24] reads and writes information directly to and from a connected Newton as it is requested or changed by the user. As a result, both devices always have a constant view of the information as it is being examined by the user. However, RCU is a general purpose information tool. It does not make any assumptions about the meaning of the information it is displaying and presents a very low-fidelity view of the mobile information by revealing the bare structure of field names and data types. As a result, it is very difficult for users to use the

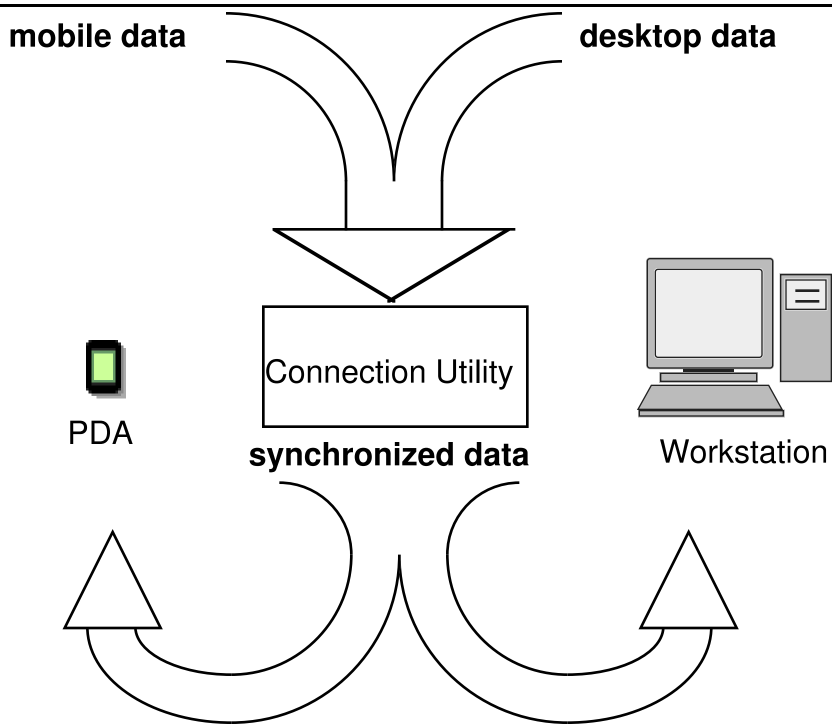

Figure 1 - Information flow in synchronization

information retrieved from the Newton as it is in a structure which they are not used to seeing and can be almost meaningless depending on the format of the information. For example, an outline on the Newton (which can have multiple levels of indentation) is presented as a single list of topics (see Figure 2). The only clue to the indentation of any particular bullet is the field named “level,” which novice users are not going to know to look for. There is no semantic connection between what the user sees through RCU and what they created on the Newton.

The applications demonstrated by this thesis take mobile information integration one step further, by allowing users to integrate information via a user interface which displays the information in a way that conveys its semantic meaning and does not require a multitude of complicated steps.

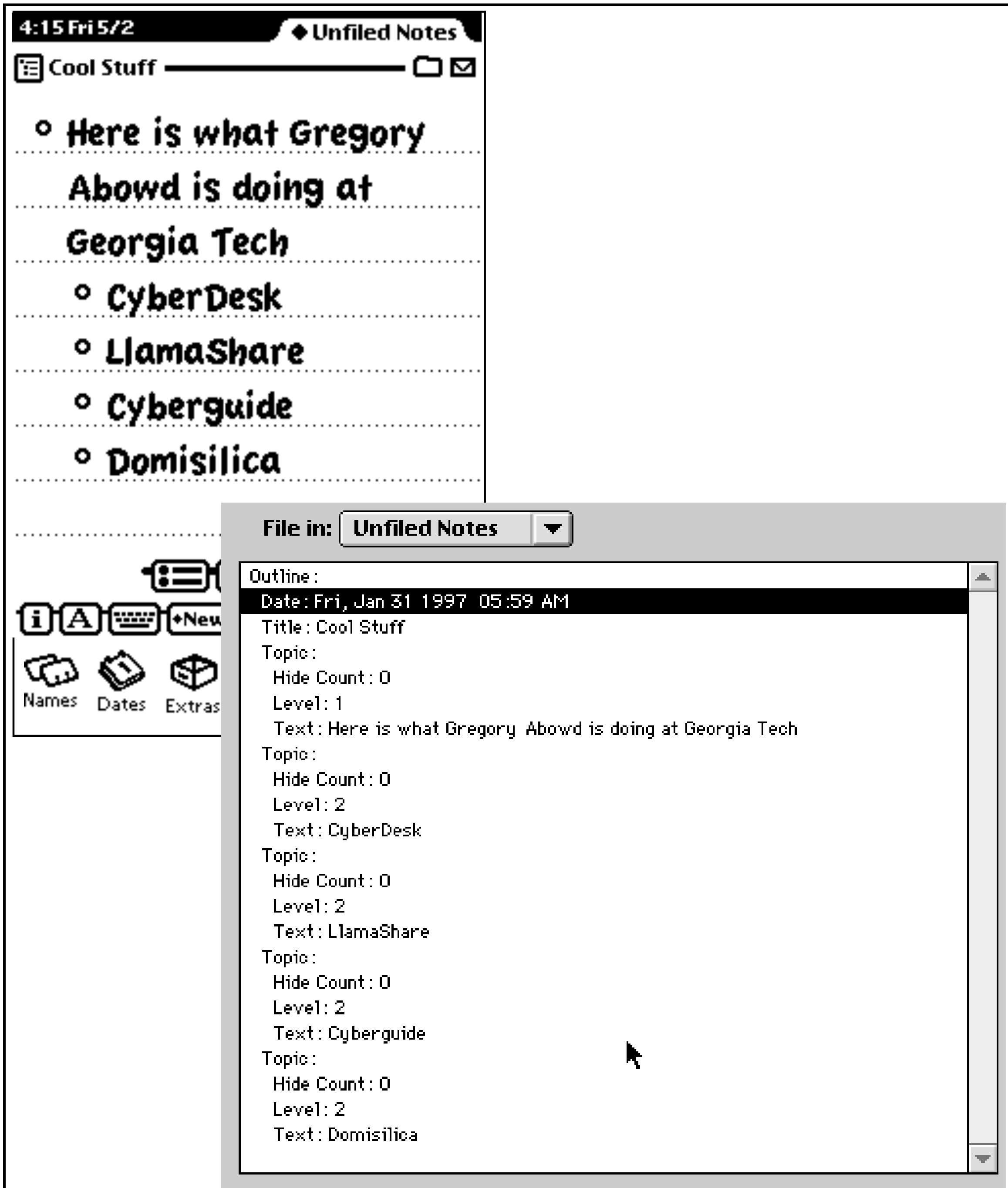

Figure 2 - RCU and the actual note on the Newton

2.5 Summary

This thesis presents an infrastructure which differs from other systems in several key ways:

- Assumes fully connected environment
- Works with truly mobile, commercially available devices (MessagePads)
- Provides applications which go beyond just providing access to mobile information (via synchronization) and address how users will actually use such information.

CHAPTER III

APPLICATIONS

This chapter discusses several of the applications which have been built on top of the LlamaShare infrastructure. The applications can be grouped into two categories: Desktop-based information clients and PDA-based groupware applications. Despite our best efforts in the past, these kinds of applications proved far too difficult without an infrastructure designed to support them. As a consequence, the applications presented in this chapter demonstrate the enabling power of the infrastructure, but more importantly, they demonstrate the ease which developers can rapidly produce applications which can take advantage of mobile information and group collaboration.

One important point should be made before continuing. While this thesis does not take credit for any of the core ideas present in the systems which we leverage, the extension and application to mobile information is a driving factor behind the LlamaShare research effort. Neither system (Cyberdog or CyberDesk) was conceived with applications to mobile devices in mind, and such applications are purely the work of this thesis.

3.1 Integrating Mobile Information into Desktop Applications

When people use their mobile device as a Personal Information Manager (PIM), they keep names, addresses, notes and to-do lists on their mobile unit as if it were a pencil and paper-based organizer like a DayPlanner™. Today's commercially available mobile devices (from PDA's to cell phones) are becoming more and more "thick," in that they store information in their own right and are not simply temporary repositories for information destined for desktop workstations. Under this model, users have a wealth of useful information on their mobile device to which they probably want access while working at their desktop. One common solution, synchronization, only complicates the issue by forcing users to explicitly make redundant copies of information just to obtain access to it. This is not only a hassle for users, but pushes the work onto the candidate least able to handle the complicated task -- the unsuspecting end user. For this reason, LlamaShare assumes information is accessed directly from the mobile device and provides user interfaces which shield the user from the retrieval of the data.

An important question is how users will address and organize their mobile data once they have access to it. This section presents two desktop applications which take very different approaches in their solutions, both of which are equally enabled by the infrastructure. The overriding theme with both approaches is that information on mobile devices should be integrated into a user's workspace or task in the same way as other remote information, such as from the Internet. Users should not have to work any differently with mobile information than they do with local or remote information.

The first application, called CyberLlama, extends Cyberdog [7] from Apple Computer. Cyberdog is a collection of OpenDoc [25, 26, 27] components for accessing remote/Internet data by providing "CyberItems" which visually represent Internet information. Since Cyberdog is based on OpenDoc, any Cyberdog component can be embedded in any OpenDoc container, a word processor for example, making the integration of web content as simple as drag and drop. In addition to Cyberdog's existing components to access and organize Internet information, CyberLlama adds components to integrate information stored on a Newton with local and remote content.

The second application extends CyberDesk [8], a separate FCE research project for allowing users to access information through simple "agents" which search multiple information spaces (the Internet, local data, etc.) based on context. The CyberDesk infrastructure simplifies the automatic integration of applications by providing services (network or local) utilized by applications in the CyberDesk environment without the application itself having to be explicitly aware of them. We have added several services to CyberDesk which locate and present mobile information (such as that on a Newton) to the user, accessible by clicking a single button.

3.1.1 CyberLlama OpenDoc Part

When it comes to integrating arbitrary content into documents, component architectures like OpenDoc or OLE almost have an unfair advantage. Components, called "parts" in OpenDoc, conform to a general API which allows them to be embedded in special parts called "containers." Containers make no special assumptions about their embedded components, beyond the general API, which allows them to literally contain anything. To developers, this yields the obvious advantage that they do not have to crowd their application with support for every conceivable content type (sounds, movies, tables, equations, etc.), but only support generic embedding through which they automatically gain the ability to contain every content type, even ones not yet invented. To users, components allow arbitrary mixing and matching of content, without any worries to the current application supporting the content in question. To integrate mobile information into desktop tasks, CyberLlama leverages the openness of the component architecture and the freedom for users to freely mix and match content.

Cyberdog, a collection of OpenDoc parts for accessing and organizing Internet content, allows users to store and categorize icons representing Internet URLs (called CyberItems) in notebooks as well as in buttons that load the appropriate content when clicked on. Additionally, CyberItem organization is left entirely up to the user. There are no rules or restrictions based on protocols (for example, you can have email addresses next to web pages, which you cannot do in either of the major browsers). Information can be organized by semantic content, not by the protocol used to access it.

In addition to supporting the standard web content types (HTML, FTP, gopher, telnet, news, mail, etc.), OpenDoc's flexible architecture makes it simple to add support for new types. To this end, we have added support for information not on the Internet, but on a Newton (notice the "My Xmas List" icon in the notebook in Figure 3). Access to mobile information is now provided through the same mechanism -- the CyberItem -- as Internet data.

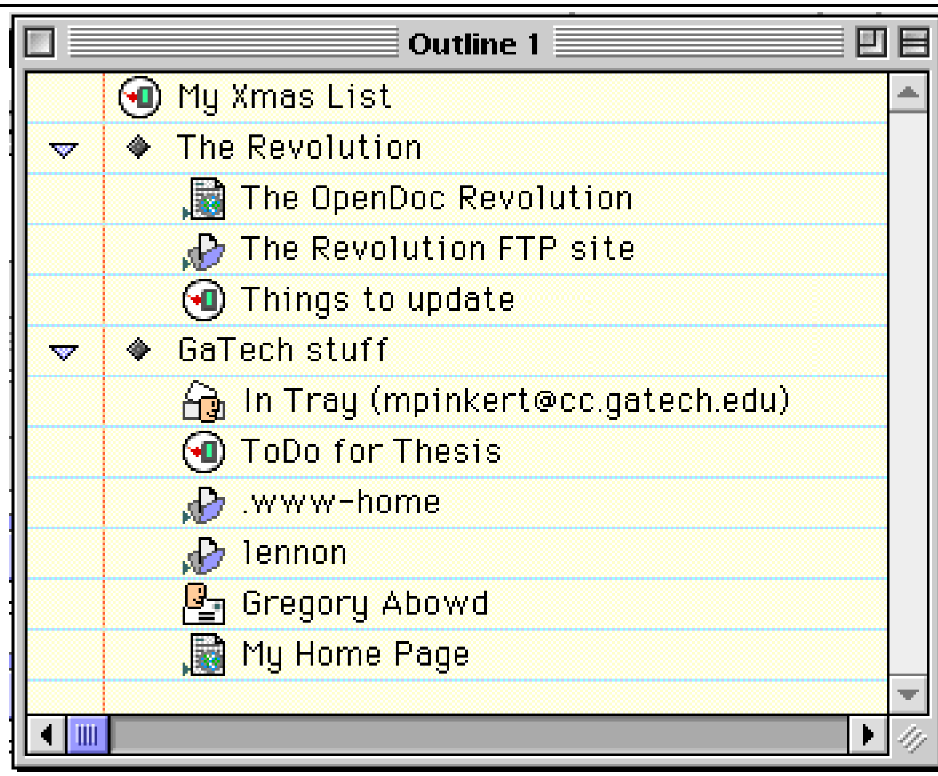

Figure 3 - A Cyberdog Notebook. "My Xmas List," "Things to Update," and "ToDo For Thesis" are Newton CyberItems

From this, we gain two important abilities. First, since all CyberItems, regardless of their type, can be organized into notebooks, Cyberdog's notebook metaphor (illustrated in Figure 3) allows the integration of mobile data into a location where users are already accustomed to looking for remote data. Second, CyberItems can be dragged into OpenDoc

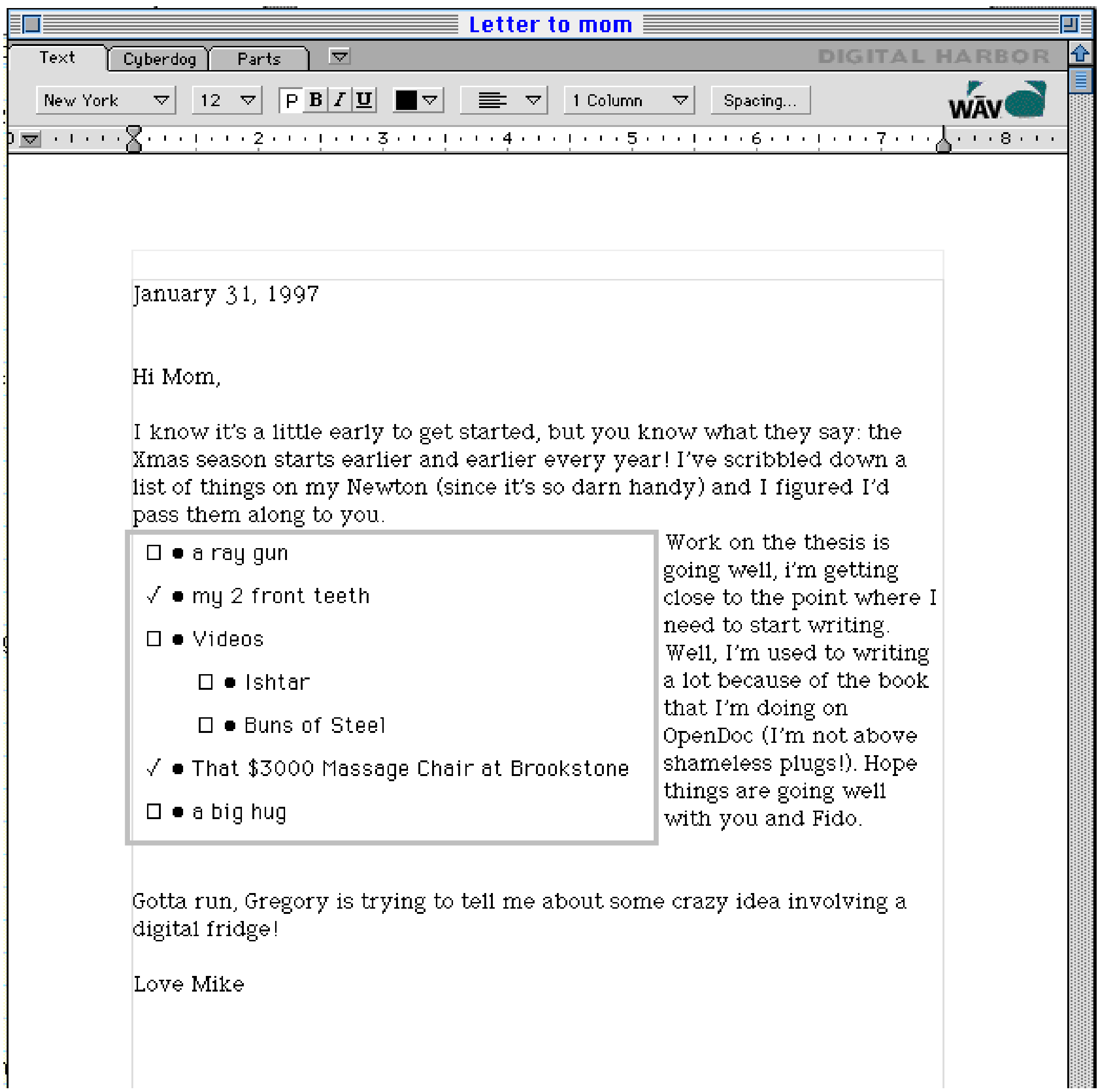

Figure 4 - CyberLlama part embedded in OpenDoc container. Gray rectangle shows embedded Newton part.

containers to display their content embedded within the document (see Figure 4). This allows drag and drop integration of mobile content into desktop content without synchronization or complicated steps.

3.1.1.1 Creating a Mobile CyberItem

When a user decides they want to access remote information (whether mobile or from the Internet), the first step is to create the CyberItem representing that information on the desktop. Cyberdog has a panel-based interface for creating CyberItems within a single dialog, called the “Connect To...” dialog. Each different kind of CyberItem has its own panel with information particular to identifying that content type. For example, creating a CyberItem for a web page requires typing in the URL and creating a CyberItem for a file on an FTP site requires the server, the path to the file, and a username/password for non-anonymous login.

To create a CyberItem representing mobile information, the user goes to the “Llama” panel (see Figure 5 below) and enters the appropriate information. But how should the user tell LlamaShare which piece of information they are interested in? Answering this question is not quite as easy as it first appears. Most accesses fall into two situations: 1) the user knows exactly what they want and where it is on the device; 2) the user knows the information is there somewhere, but not quite where. Unfortunately, each of these scenarios calls for radically different user interfaces.

The first scenario suggests building a “browser” similar to Microsoft’s Explorer or the MacOS Finder which allows the user to drill down through multiple levels, navigating directly to the desired piece of information. But what if they don’t know exactly where it is, or even if it’s there at all? The second scenario requires a “find” mechanism which returns a list of all entries which match some specific criteria (such as “contains the word ‘FCE’”).

This is great for finding information entered long ago and forgotten about, but cumbersome when the user knows exactly what they are looking for. For instance, they might know where the information is, but not the exact text within it that would match a search. Text-matching queries alone might force the user to try several different search strings before they find one that matches. Some combination of the two would certainly be the most desirable.

However, given the time constraints placed on this project, there was no way that such an interface could have been developed (basically we would be rewriting Explorer or the Finder from scratch). For this reason, either some subset of functionality needed to be developed or a choice had to be made on which scenario would be most useful in the short term. A combination approach was decided upon which provides the ability to perform both text-matching searches and limited browsing (per device only). The result is illustrated below in Figure 5[1].

The user first chooses which mobile device the information is on and then the "soup" which contains the data. (A soup is similar to a folder which holds pieces of related information, but usually for a particular application such as the Note Pad or the Calendar.) The user can then choose to get an overview of an entire soup, or search that soup for entries matching a text-search. The overview shows a quick summary of each entry in the soup, such as the title of title of a note and the first line or two of text. This should be enough context if the user knows exactly what they are looking for. The text-search also returns an overview of each entry that matches the criteria. When the user finds the desired entry, they select it and click OK. The CyberItem for that piece of information is then created.

---

[1] This panel has not been fully implemented. All work has been done with a substitute panel with a more "developer oriented" interface to allow quick testing of the remaining (and more interesting) functionality.

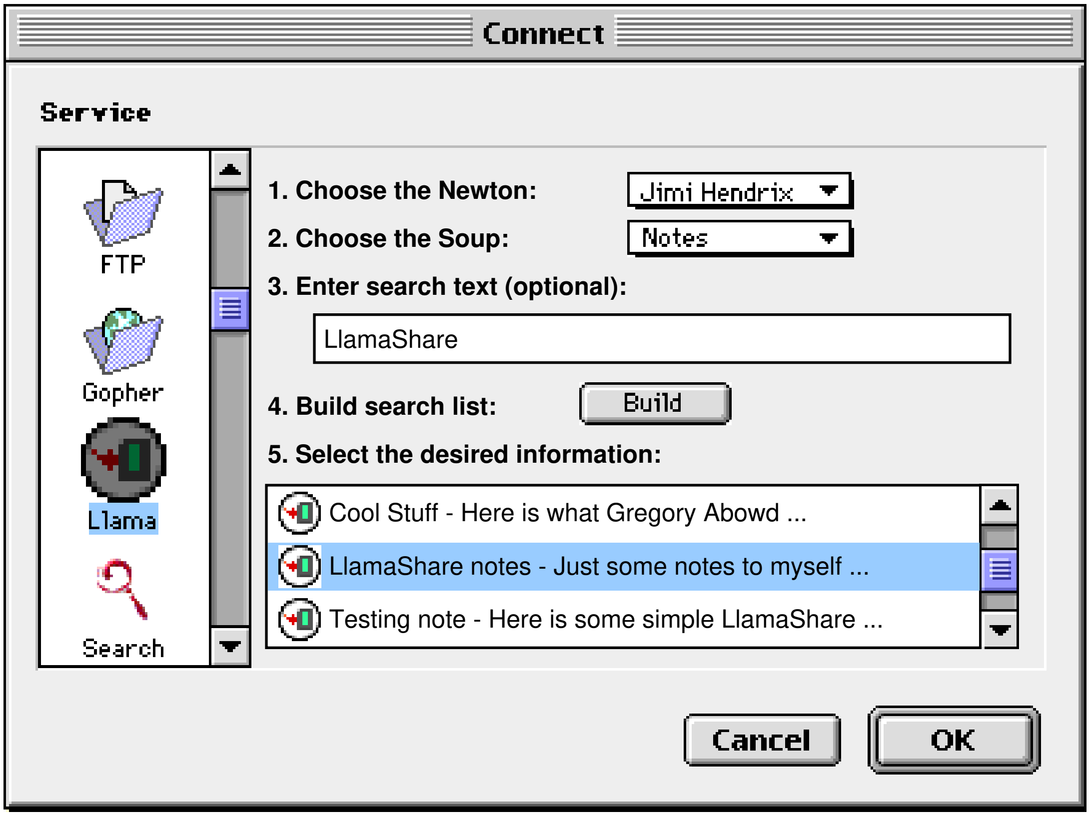

Figure 5 - The Llama ConnectTo panel

3.1.1.2 Integrated organization

Once a mobile CyberItem is created, it behaves exactly like any of the built-in CyberItem types in that it can be placed into a notebook, dropped onto a CyberButton, or dragged to the Finder. All three of these methods provide the user the flexibility to organize CyberItems any way they choose.

The most common form of organizing CyberItems, placing them in notebooks, is also the most interesting. CyberItems (and hence mobile CyberItems) can be organized in a Cyberdog notebook in any way the user wants, illustrated by Figure 3 above. There are no limitations or restrictions based on protocols, as exist in Netscape Communicator and Microsoft Internet Explorer. If a user wants to create a notebook with newsgroups, URLs, email addresses, and notes from a Newton, they are perfectly free. Another difference from the traditional “bookmark list,” which Cyberdog’s notebooks are traditionally compared to, is that a user can have multiple notebooks. This allows a single notebook to have all the items related to a particular topic without being overcrowded with all the URLs from other, non-related topics (a typical problem with bookmark lists in browsers). For example, a user can have a notebook containing everything related to a meeting, including the notes taken on the PDA, the email addresses of everyone in the group, a web page containing the project being discussed, and a newsgroup for group collaboration after the meeting.

Along the same lines, CyberItems can also be placed into CyberButtons with drag and drop. These visually programmed buttons open the specified CyberItem when they are

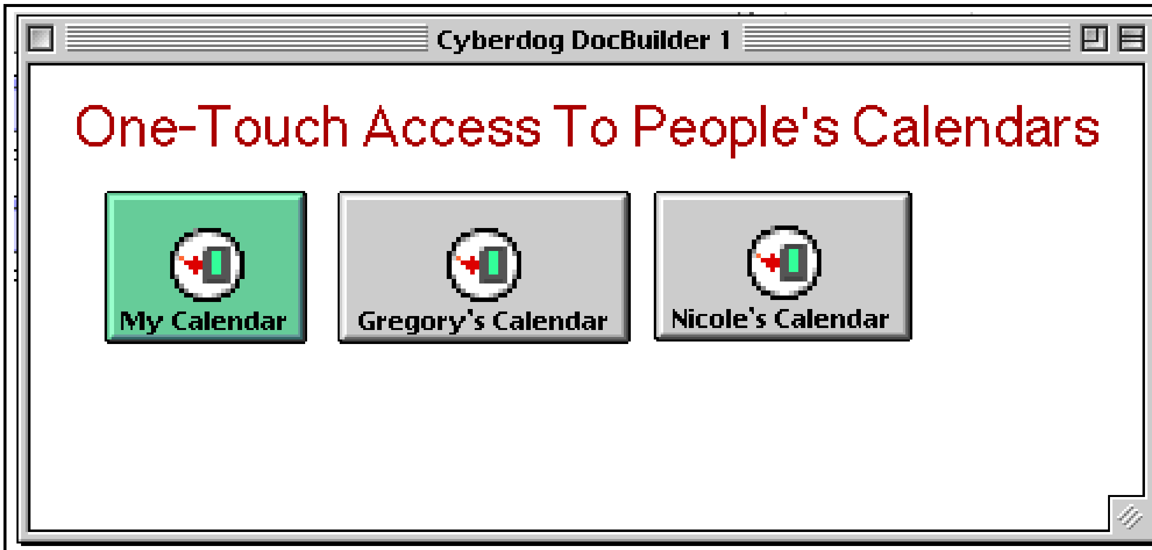

Figure 6 - A document with CyberButtons. This is just an example of what is possible

clicked. CyberButtons allow users to organize single references to mobile information directly into documents, where an entire notebook would be inappropriate. Figure 6 shows a document which allows one-click access to calendar information stored on multiple MessagePads.

Finally, CyberItems can be dragged into the Finder and are saved as files in the folder in which they are dropped, illustrated in Figure 7. It is important to note that only the pointer to the information is saved on the disk, not the information itself. This allows users to organize mobile information as files alongside other files on their desktop, which may fit

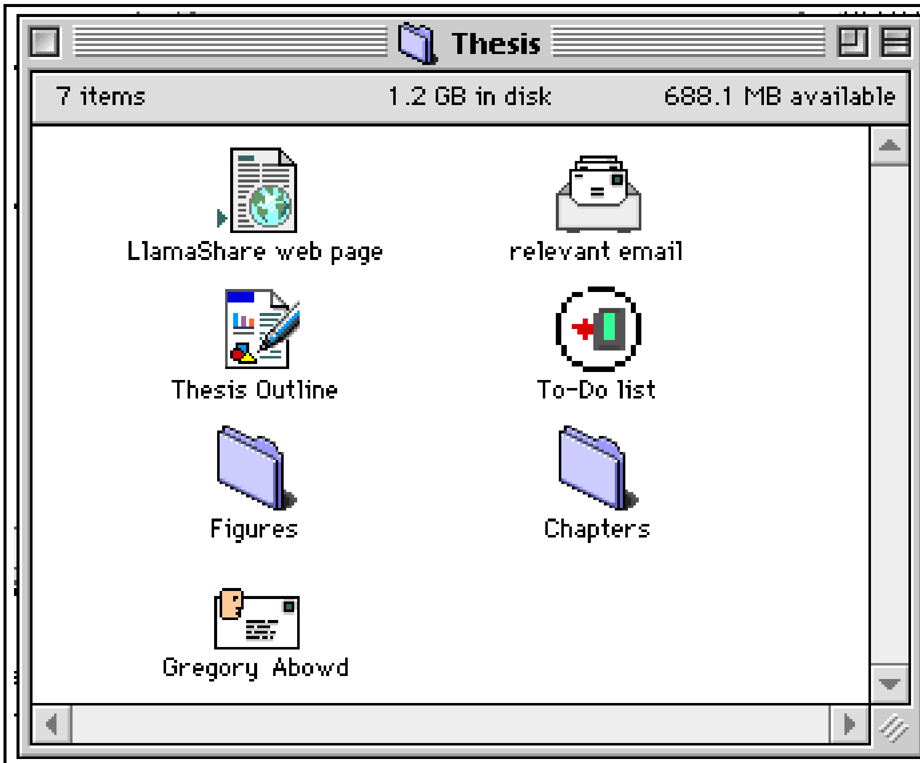

Figure 7 - Organizing CyberItems on the desktop

more easily into a user's perception of how their information is distributed. Making no assumptions as to if that statement is true, the opportunity is certainly presented, allowing the user to organize information however they choose and not the single way presented by the system.

3.1.1.3 Integrating Mobile Information into documents

We still have yet to go beyond "accessing" information, which the synchronization model can handle, even though it does not help with organizing the information. The next stage in integrating mobile information into the desktop is to physically integrate it into desktop content, such as a word processing document. Fortunately, since we are working within an OpenDoc environment, this is simple for users to accomplish.

CyberItems can be embedded directly within OpenDoc containers (such as a word processor) just by dragging the CyberItem from a notebook or the desktop and dropping it onto a document. The CyberItem is opened and the content is displayed directly in the document at the mouse location, and the result is shown in Figure 4. Notice that the user does not have to worry at all about whether or not the content is in an acceptable format because the container can accept anything. As long as the appropriate viewer part is available to display the information, the user does not have to bother with translating the mobile data to a form which can be understood and manipulated by the application.

In addition, mobile data can be edited in place, and changes are stored directly on the mobile device. There is no need to repeat the synchronization process to return the updated information to the mobile device. Along the same lines, since the display part also keeps track of the CyberItem used to display the information, each time the document is opened (or when the user explicitly hits "reload"), the document loads the updated information directly from the mobile device. As a result, the user always interacts with the most up to date version of their information, again without having to synchronize. If the mobile device is not available (i.e., not connected to the network), a read-only snapshot of the information is presented to avoid getting out-of-sync, but still allowing the user to view the most recent information.

Essentially, a user is now free to work with mobile information using the same interactions as if it is locally stored. Since the information is edited and viewed in-place in the document, the user may not even be aware that the information resides on a mobile device. This interface demonstrates that there is no need to force the complicated steps of synchronization upon users.

3.1.2 The CyberDesk Environment

CyberDesk is a Java-based project developed by Wood, Dey, and Abowd at Georgia Tech [8]. One of the goals is to provide an architecture for application integration in which applications can automatically take advantage of the services provided by other applications in the environment. These services can make use of both local information, such as a calendar or a contact manager, or Internet information, such as searching for a phone number with SwitchBoard or finding all web pages containing a text string with AltaVista.

Unlike Cyberdog, which takes the approach of giving each piece of information a physical representation on the desktop, CyberDesk allows user to search a nebulous information pool of local and Internet information in response to queries based on the text selected in a CyberDesk-aware application, such as an email application. For example, when the user selects a person's name, they are provided the option to look up that person's phone number on SwitchBoard, find their email address on WhoWhatWhere, or perform a generic search for their name on AltaVista. The actions which a user may take are

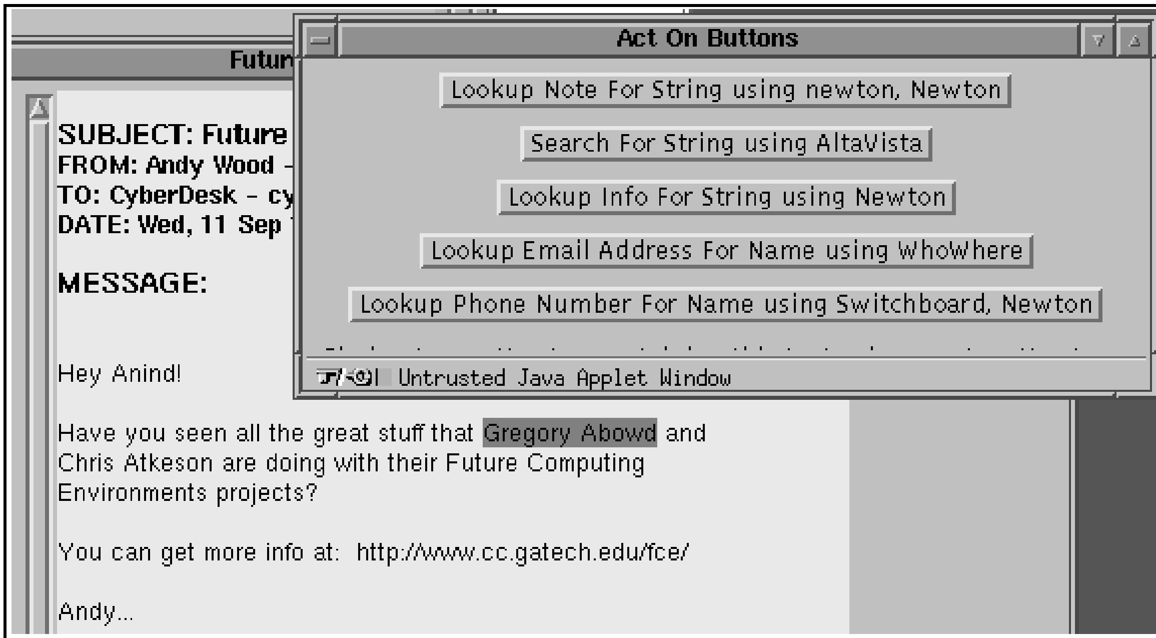

Figure 8 - CyberDesk ActOn window

dependent on the type of data selected (name, URL, email address, etc.) and appear unobtrusively in a separate window called the “Act On” window, shown in Figure 8. In addition to searching the Internet, the user is given the option to display contact information from the locally-stored contact manager if a CyberDesk aware contact manager is also present. In this way, CyberDesk blurs the distinction between local and Internet information services.

Since CyberDesk was such an open architecture, it was simple (an afternoon) to add a service which additionally searches a mobile device for the selected information. Two applets, one for displaying notes and another for displaying names, were written to allow users to browse information residing on a Newton (see Figure 9).

Users now only need to learn one interaction model (select text, click ActOn button) to access information stored locally, on the Internet, or on a mobile device. This blurring of the boundaries, emphasized by the lack of physical representations which draw attention to

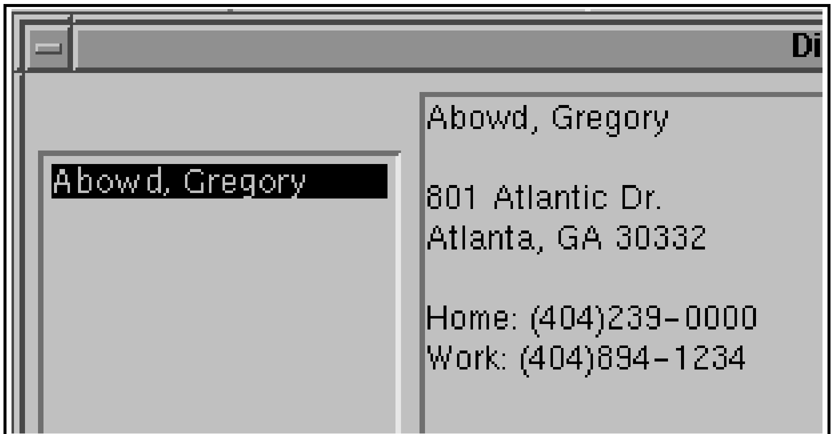

Figure 9 - A CyberDesk viewer for Newton names

the actual location of the data, simplifies the number of steps necessary to access information and makes the actual sources more ubiquitous to the user. Furthermore, since that information is retrieved based on the user's context during the completion of a task, it is seamlessly integrated into the task itself.

In order to completely hide the location of the information from the user, we have proposed a change to the way CyberDesk creates the ActOn buttons (which is currently being implemented). In the existing system, CyberDesk creates one button in the ActOn window for every applicable service, even when different services produce similar results such as looking up a phone number. As a result, the user must manually choose between the actions based on where they think the information is located (for example, a local contact manager, the Internet service SwitchBoard, or the Newton). To achieve the goal of concealing the information's true location from the user, we propose to combine all action buttons which provide a similar service into a single button which, when clicked, searches all information spaces simultaneously. While the user may get multiple responses (another issue to be resolved in the future), they no longer need to be consciously aware of the location of the information in order to access it.

What do the above applications demonstrate? Most importantly, they demonstrate how powerful removing the barrier between mobile information and the rest of a user's information can be. Once a user's computing environment is opened up to include all the information to which they have physical access, they are empowered to use all the tools and devices available to complete their task, whether it is making a phone call to a colleague or writing a summary of their last meeting. Nobody doubts the overall gains which integrating Internet information into our desktops has provided, and mobile devices, which hold important information not available on the Internet, are no different.

3.2 Collaboration among mobile devices

The previous applications focused solely on the mobile device acting as a passive server, answering queries for information already stored on it and returning the information to some client somewhere on the Internet. This is only half the story, as a mobile device should be able to act a client in its own right, requesting information from the environment to present to the device's user. Mobile, multi-user, collaborative applications have not become prevalent because the infrastructure was never available. Now that there is one, we can begin to explore the possibilities.

This section provides two such applications which allow users with Apple MessagePads to collaborate with other MessagePad users as well as one-to-one with desktop users.

3.2.1 CyberGuide

Our first mobile groupware application is based on CyberGuide, another project here in FCE which uses PDAs as mobile tour guides in an unconnected environment. The user can use the device to access information on particular items of interest in the environment, such as demos, exhibits, or the neighborhood pub (see Figure 10). In addition, each PDA has access to some external positioning system (GPS for outdoor operation, IR beacons for indoor use) and can report the user's position in relation to objects in the environment on a map. Since the device knows where it is and other information such as the current time, it can answer queries such as "when do I have to leave for the next scheduled demo?" and "which bars are still open within 1 mile and serve Guiness on tap?"

Unfortunately, each device is totally isolated and has no way to share information,

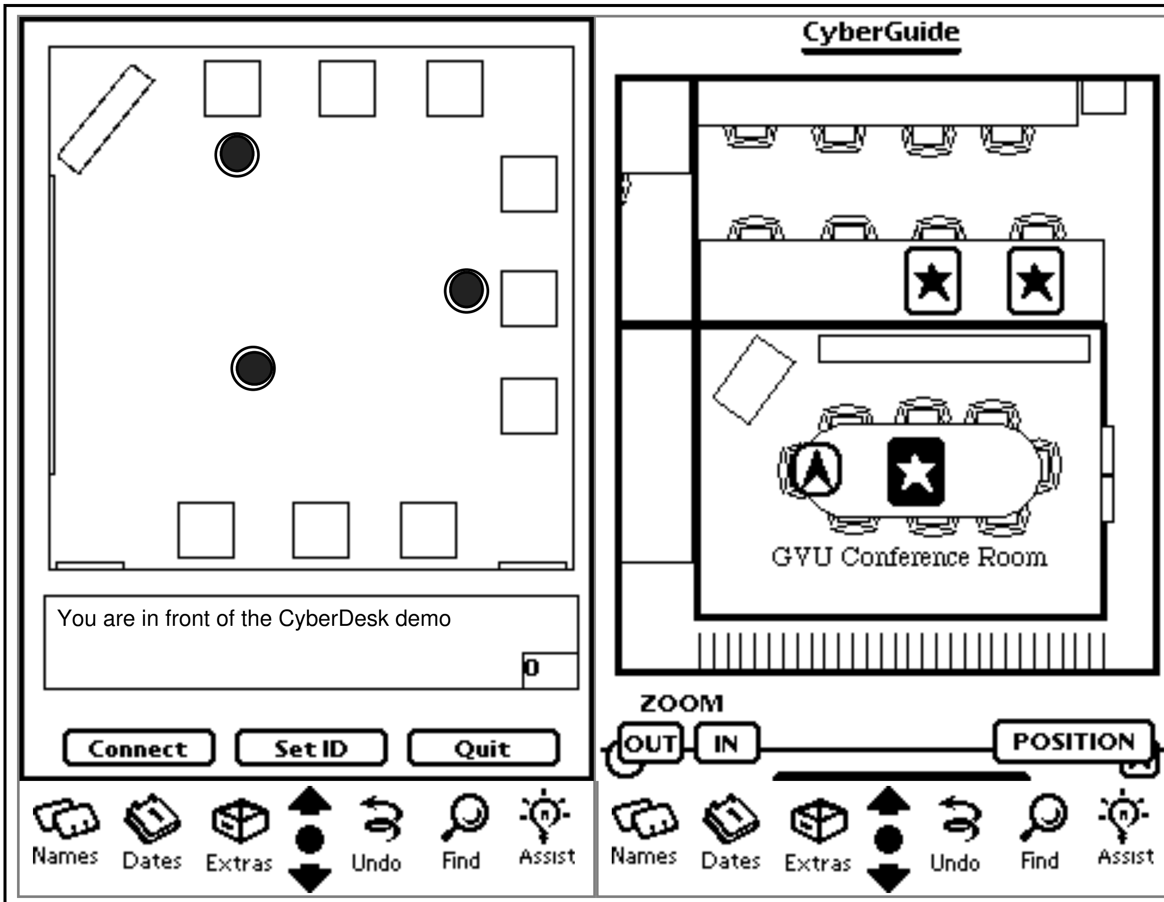

Figure 10 - CyberGuide & CyberGuide II

such as the user's location. Enter LlamaShare, which provides an infrastructure which allows mobile devices to publish and request information. The new version of CyberGuide (dubbed CyberGuide II) allows each user to see the location of every other user in the environment. For small locations, such as a single room, this application doesn't make a lot of sense, but in a larger environment, such as an entire building or an entire campus, knowing the position of your children or the tour bus would be very useful.

CyberGuide II works through a combination of desktop-based and Newton clients. Our positioning system uses several cameras to determine the location of all the IR beacons in the environment, where a UNIX-based server collects, interprets, and spits the positions out a socket to the Internet. Another client reads from that socket and writes each of the positions to a Global Soup (a LlamaShare construct which acts as a globally readable and writeable database, similar to "soups" on the Newton). These components are illustrated in Figure 11. Each Newton then, as a client, reads everyone's positioning information and displays it to the user via the map on its screen. At the same time, the Newton is responding to user taps on the map to present information about objects of interest in the environment.

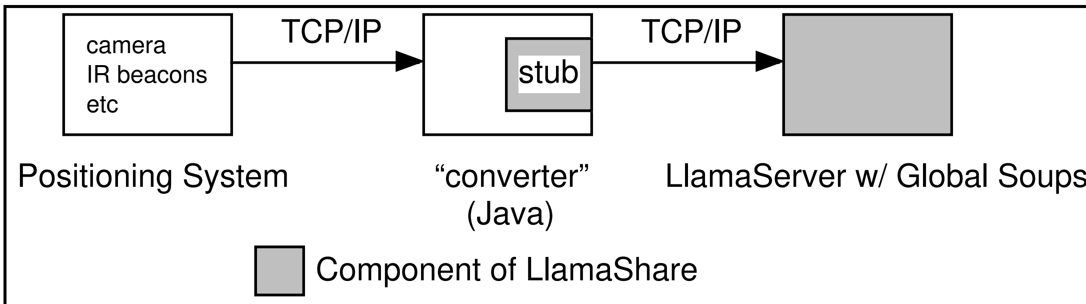

Figure 11 - The server-side components of CyberGuide II

Finally, CyberGuide II demonstrates LlamaShare's ability to share information between mobile devices in a collaborative environment. Users can enter their own comments about each demo which are saved into "Global Soups." When a user visits a demo, the comments for that demo are automatically downloaded from the network to that user's Newton and displayed. This mechanism provides a very simplistic form of group communication, and can easily be extended to include more information than just a comment, such as numerical ratings. Furthermore, the comments can be reviewed from desktop-based applications either while the demos are in progress or later, after the fact. This application truly demonstrates LlamaShare's flexibility to exchange information between clients from heterogeneous platforms.

3.2.2 CyberTALK chat

The CyberTALK project [28] is an undergraduate project designed to allow users on a mobile devices, such as a MessagePad, to communicate with other desktop users. Functionality includes paging, email, and, most interestingly, a "talk" program which allows mobile users to communicate with desktop users in a real-time conversation. Information is relayed between the two devices over the network, using "Global Soups" as message centers.

CyberTALK further demonstrates LlamaShare's ability to share information between heterogeneous clients. The desktop side of the chat program is written in Java, and can be used within any Java-enabled browser. The mobile client is written for a Newton in NewtonScript. Both clients can freely send and receive information from the other.

3.3 Summary

This chapter presented four applications, two on the desktop and two on the Newton, which demonstrate the power and flexibility of the LlamaShare infrastructure to share information between heterogeneous systems. The two desktop-based applications explore alternative user interfaces for incorporating mobile information into existing desktop or Internet content. The mobile applications demonstrate the ability of the MessagePad to act as a client in the environment, allowing the mobile user to either participate in group collaborative applications or communicate with desktop-based users in real-time. The following chapter, Architecture, describes the infrastructure on which these applications are built.

CHAPTER IV

ARCHITECTURE

This chapter covers the design and implementation of the LlamaShare infrastructure. First, an overview of the goals is presented, followed by an overall picture of the components. After that is a discussion of many of the runtime and storage details on the Newton. Next, this chapter discusses the components of the infrastructure individually, highlighting the functionality and the role each piece plays. Finally, it describes the interaction between the different components, including a detailed analysis of the communication and registration processes involved.

4.1 Goals

In order to provide a sufficient infrastructure able to support the rapid development of applications which take advantage of the sharing of mobile information, LlamaShare needed the following characteristics:

• Platform and language independence for desktop-based clients

Most of the research being done in FCE consists of Java-based applets running through browsers on a variety of platforms. However, libraries for the Newton only exist for MacOS and Win95/NT. Furthermore, these libraries could only be called from C or C++, which greatly limited our development choices. The new infrastructure should be language independent and allow clients to run on more platforms.

- Ability to access more than just the device tethered to a workstation

  To facilitate the sharing of information between users and their mobile devices, each client needed the ability to access more than just the device attached to the user's workstation by a serial cable. That way client applications could read and manipulate information, such as calendar data, from a variety of devices. Without the ability to access more than the single mobile device at the user's desk, such groupware applications would not be able to take advantage of mobile information like they do with other information.

- Mobile devices are clients which themselves request information

  As we developed the LlamaShare infrastructure, it became clear that the mobile devices should act as more than just passive repositories of information. Desktop-based groupware applications were interesting, but to really push the paradigm of ubiquitous computing required an infrastructure which allowed all devices to participate in the environment. Each mobile device should be able to share information with other devices as well as request that information for display to the user holding the device. Having such an infrastructure would allow us to develop a suite of mobile-based groupware applications with which to do research.

LlamaShare was able to address and solve all three of these goals, as demonstrated in the following section.

4.2 Overview

The LlamaShare infrastructure is comprised of three components, illustrated in Figure 12:

- a centralized server called LlamaServer
- an application stub for mobile-based clients called MobileConnect
- an application stub for desktop-based clients called DesktopConnect

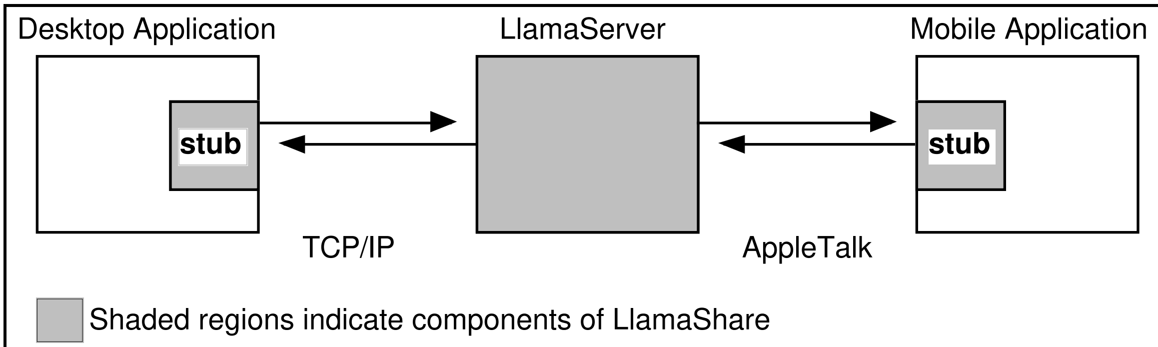

Figure 12 - Overview of LlamaShare infrastructure

The LlamaServer acts as a centralized traffic cop and oversees all communications between desktop and mobile applications. To the desktop clients, it speaks TCP/IP in order to provide language and platform independence. Using TCP-based communications also allows applications on any machine on the Internet to connect to the LlamaServer and request information from mobile devices. On the mobile side, the LlamaServer speaks ADSP (the protocol used by AppleTalk), which allows the easy connection of multiple devices to the server. The LlamaServer also contains several “Global Soups” which act as data stores in which to hold shared information. Information can be read from or written to Global Soups by both desktop and mobile applications.

Handling the communication with the LlamaServer on the mobile side is a stub called MobileConnect. This stub can act in one of two modes: server or client. As a server, this stub can receive requests for information stored on the device from the LlamaServer, locate the desired information, and then return it over the network. As a client, the stub can send or receive information between an application running on the mobile device and any of the Global Soups which reside on the LlamaServer. In both cases, the stub is embedded within an application which runs entirely on the mobile device. The application decides whether it wants the stub to operate in client mode or in server mode, but then is isolated from the details of the connection and communication.

Finally, desktop-based applications use their own stub to communicate with the LlamaServer. Named DesktopConnect, the stub provides an RPC-style calling convention to hide from the application the details of accessing information over the network. The primary job of the DesktopConnect stub is to inflate the streamed data structures read from the LlamaServer over a TCP socket into "live" data structures.

4.3 Newton Overview

At this point, some explanation about how the Newton and its storage model might be helpful. The Newton (and its application programming language NewtonScript) has some interesting features not found in most systems. First is the concept of a "frame," which is similar to a structure, record, or class in most programming languages but is dynamic in that it can grow and shrink at runtime. Frames are stored in "soups" which are similar to files except they are indexed and searchable via an API.

A "frame" is a container, like a struct in C, with typed fields called "slots" that hold the data. Frames can also be nested. However, unlike structs which have a predefined (at compile time) structure, a frame is free to gain new slots and change its overall structure dynamically, demonstrated in Figure 13. Also unlike structs, slots are not accessed by compiler-determined byte offsets but by name, which is the key factor that allows them to change structure dynamically. As a result, slots can be added to an existing frame without disrupting its usage by an application that does not know about the new slots because the application makes no assumptions about the exact structure of the frame. Finally, all type checking is done at runtime so each slot is tagged with an appropriate type, even when it is stored persistently.

Before	After

<pre>f = {
    a: "hello",
    b: 27,
    c: {
        foo: "Test",
        bar: "More"
    },
    d: 'c'
}</pre>
	<pre>f = {
	    newSlot: "This is new" ,
	    a: "hello",
	    b: 27,
	    c: {
	        foo: "Test",
	        bar: "More"
	    },
	    d: 'c'
	}</pre>

Before	**After** **f.newSlot := "This is new";**

Figure 13 - A frame, before and after adding a slot

Frames are stored in "soups," named for their ability to hold frames with heterogeneous structures (think of a soup as the "Great Data Melting Pot"). Soups are more analogous to miniature databases as each soup holds all of the frames for a particular application. For example, each note created with the Newton's NotePad application is a single frame and the entire soup, called "Notes," contains all the notes created by the user. Contrast this with the typical file system of workstations which would have a separate file for each note and no common location for data (files can be spread anywhere, making locating them difficult). Since all information for a particular application is co-located, indexing and searching are both very easy. Each soup can have indexes on any of the slots present in the frames of the soup which speeds up searching significantly, just like in a database. Furthermore, API's are provided for searching a soup using a variety of criteria ranging from any frames containing certain text to frames whose slot matches certain criteria. This functionality is nonexistent in most desktop Operating Systems and makes serving mobile information from the Newton possible.

LlamaShare extends the idea of soups with "Global Soups" (see Section 4.4 for more about Global Soups). Like their Newton namesakes, Global Soups store frames, but instead of residing on a particular device, a Global Soup resides in the LlamaServer. As a result, the Global Soup is universally accessible to all devices in the environment, both mobile and desktop.

The libraries used to communicate between the Newton and the LlamaServer are also worth explaining. The Desktop Integration Libraries (DILs) [29] are C libraries written for MacOS and Win95/NT which are divided into two layers (shown in Figure 14). At the lowest layer are the Communication DILs (CDILs) which are analogous to send and receive calls, used to send messages back and forth between the Newton and a desktop machine. The LlamaServer uses the CDILs to send small messages to the Newton such as the request command and to acknowledge the receipt of frames. Built on top of the CDILs are the High-level Frame DILs (HLFDILs, or just FDILs). As the name suggests, this layer is a higher level protocol which encapsulates the sending of an entire Newton frame between the desktop and the Newton. Additionally the FDILs hide the process of flattening the frame for transmission and reinflating it at the other end.

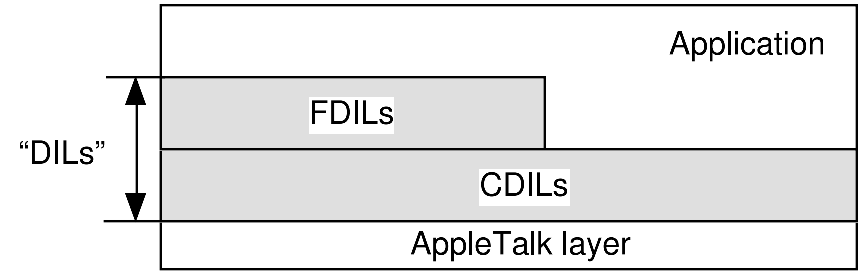

Figure 14 - Layers of the DILs

The FDILs use a process called "binding" to establish a one-to-one correspondence between a C data structure and a slot in the received frame. Before transmission, each slot in a frame is bound to a pre-allocated buffer space for the slot. This is fine when the application knows exactly which slots are part of the incoming frame, but frames are allowed to have additional slots which the application might not be aware of. To handle this unexpected information, named "unbound data" because it has not been explicitly bound by the application, the FDILs create a multi-way tree where each branch is an unbound slot. Whether bound or unbound, the type information associated with each slot as well as the slot names stay with the frame even after it crosses into the desktop world.

4.4 LlamaServer

The LlamaServer has two main roles in the LlamaShare infrastructure. The first is to provide a connection between desktop-based applications and a collection of mobile devices. The second is to act as a storage location, allowing mobile devices and desktop applications to collaborate through shared information repositories. Both of these roles enable the development of content-rich applications, hosted on both workstations and PDA's, which can leverage a combination of local and mobile information.

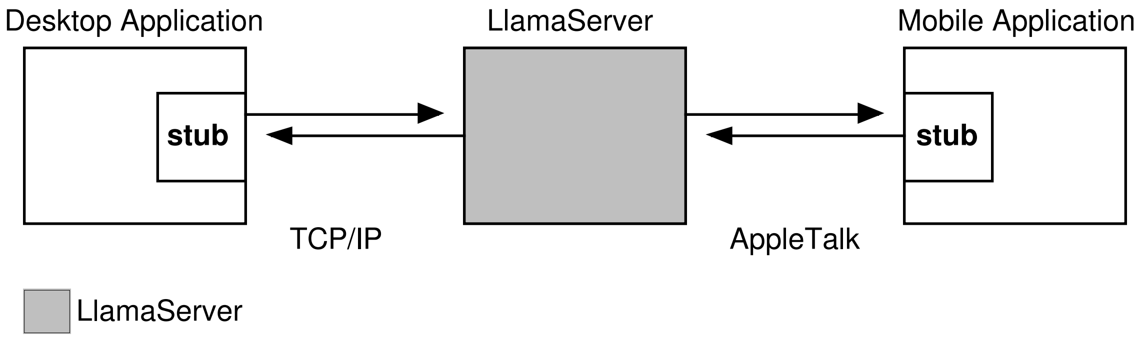

Figure 15 - LlamaServer's position in the infrastructure

Primarily, the LlamaServer is a bridge. Most desktop-based applications use TCP/IP for inter-application communication. TCP/IP allows each participant to be written in different languages and execute on heterogeneous platforms yet still be able to communicate. If desktop-based applications were given access to information stored on mobile devices, TCP/IP should continue to be the protocol utilized to access that information for the sake of ease of programming and consistency. None of the mobile devices we worked with, however, could handle TCP/IP easily which required that other protocols be adopted in order to get them on the network. The easiest protocol for our purposes was ADSP (AppleTalk) since our existing network already supported it, the MessagePad supported it without any additional hardware, and wireless support was easy to achieve. Without a bridge between the two protocols, desktop applications would lose the language and platform neutrality provided by TCP/IP, which was not acceptable.

As a result, the LlamaServer speaks both protocols and manages the communications between the two separate worlds. When a command comes in from a desktop client, the LlamaServer checks its internal list of currently connected mobile devices and then sends the appropriate command to the requested Newton using the CDILs (see Appendix A for a full listing of commands). The response from the Newton, in most cases, is a frame which is then read by the FDILs over ADSP. Since the server is reading a frame of unknown origin and content, it cannot make any assumptions about the slots and must therefore read the entire frame as unbound data. The LlamaServer then flattens the unbound frame, turning the recursive tree structure into a byte stream capable of being transmitted over TCP/IP where the byte stream is read in by the DesktopConnect stub and reinflated into a proper data structure. This exact interaction will be covered in Section 4.7, Intercomponent Communication.

The process of flattening the unbound data deserves some explanation. The FDILs handle the protocol of flattening the frame and subsequently inflating it on the desktop side internally, which means there is no API to easily do it. Even if there was, the DesktopConnect client stub would then be dependent on the FDILs to inflate the data at the desktop application, which would once again limit the language and the platform. As a result, the LlamaServer uses its own flattening protocol to send the unbound data over the Internet.

Flattening the unbound data uses a recursive protocol (see Figure 16) which makes heavy use of tags (character strings) as markers to indicate the structure of the frame within the byte stream. Each frame begins with the '[ENTR]' tag indicating the beginning of a new frame and ends with '[ENDF]'. In between comes each slot sent in order (though the ordering is arbitrary) marked by a '[SLOT]' tag and ending with a '[ENDS]' tag. Between these two tags comes the definition of the slot, consisting of an integer representing the data type, the name of the slot, and the slot data (if any). If the slot contains only a simple type (such as an integer or a character), the tag '[NCHILD]' is inserted, indicating the base case of the recursion to follow, and the next slot is sent in turn. If the slot is a frame, the tag '[YCHILD]' is used to indicate that this slot contains a nested frame. At this point, the nested frame is sent using the same format as described previously. The nesting continues until all slots are leaves (which must happen at some point) at which point the final [ENDF]

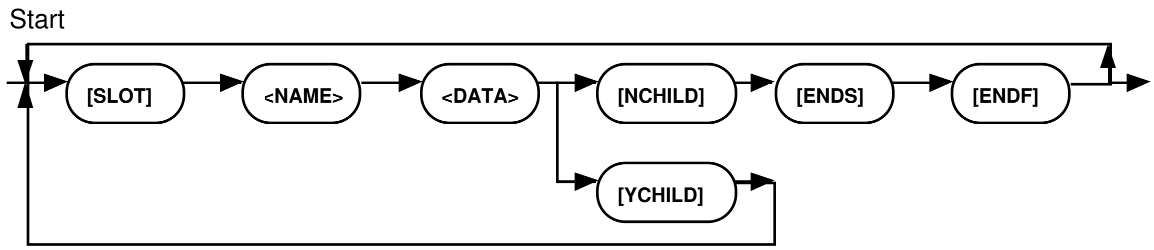

Figure 16 - Diagram of flattening protocol

for the nested frame is inserted. The next top-level slot is flattened in similar fashion. Finally, when all top-level slots are flattened, the top-level [ENDF] is inserted into the byte stream indicating the end of the definition of the frame. An example of the byte stream created from flattening a slot is shown in Figure 17.

Just as frames are flattened for transmission to the DesktopConnect stub, frames intended for the Newton (originating from the desktop application) must be inflated and bound into a DIL object in order to use the FDILs. The DesktopConnect stub flattens the frame using the same protocol described above, which makes the byte stream easy to parse. The difficult task is then building a bound DIL object from the incoming stream. Again, a recursive process is used. For each slot in the incoming frame, the algorithm adds another level to the recursion if it is a nested frame, stopping when it reaches a simple data type (the base case of the recursion). The frame is then bound from the bottom up, one slot at a time. As each slot (nested or not) is built from the byte stream, it is bound into its parent using FDIL routines. When all slots at the top level are finished, the DIL object is complete and ready to be sent to the Newton or stored in a Global Soup.

```
f = {
        a: “hello”,
        b: 27,
        c: {
                foo: “Test”,
                bar: “More”
        },
        d: ‘hi’
}
```

```
[SLOT]
        a hello [NCHILD]
[ENDS]
[SLOT]
        b 27 [NCHILD]
[ENDS]
[SLOT]
        c nil [YCHILD]
        [SLOT]
                foo Test [NCHILD]
        [ENDS]
        [SLOT]
                bar More [NCHILD]
        [ENDS]
[ENDS]
[SLOT]
        d hi [NCHILD]
[ENDS]
[ENDF]
```

Figure 17 - Byte stream produced from flattening

The other major function of the LlamaServer is to provide a central repository for shared information to facilitate the creation of collaborative applications between mobile devices. Just as soups are local repositories for Newton frames, LlamaShare’s “Global Soups” are globally accessible repositories accessible from both desktop applications and mobile applications. The server can hold an arbitrary number of Global Soups, but each must have its own unique name as that is how the soup is referenced. Any device may add to or read from a Global Soup, which enables applications to share information by publishing it to a common location. Furthermore, information in a Global Soup persists, even after a device has disconnected from the environment and its local soups are inaccessible.

Each Global Soup consists of an unordered collection of frames stored in multiple representations. In addition to the data, each entry is also tagged with the name of the Newton from which it came (the tag is empty if the frame originated from the desktop). This allows queries to return all the frames originating from one particular device, and might be used as the basis for a simple caching system. Currently, each frame is stored in two ways due to limitations and bugs within the FDILs: a bound DIL object which can be transmitted to the Newton using the FDILs, and a recursive data structure which can be flattened for transmission over TCP/IP to a desktop-based client. However, memory is not an issue within the server so this is not really a problem.

Supported commands on the Global Soup are currently very limited. Both desktop and mobile clients can add frames to a Global Soup, clear the entire Global Soup, read the entire Global Soup, and atomically read and clear the soup. The last command is useful for making sure a device gets all the contents of a soup before clearing it (indicating that it has seen everything there) and that no new entries sneak in between the “read” and the “clear” operations. Currently, there is no way to replace or update individual entries in the soup or do arbitrary queries such as what is possible with real Newton soups. The next version of the Global Soups will add introspection capabilities which facilitate indexing and updating based on information stored within individual frames. Regardless, the currently supported commands are enough to build simple collaborative applications and allow us to do research on groupware applications.

A good example of mobile collaboration is extending CyberGuide [3] to allow users to share their comments about particular exhibits or demos. The comments for each exhibit would be stored in a Global Soup. When a user decides to request more detail about a particular exhibit, CyberGuide reads the comments from the Global Soup and displays the ones relevant to the current exhibit to the user. The user would be free to add their own comments to the pool for the benefit of others in the group.

The LlamaServer is written in C++ using PowerPlant [30], a MacOS-based application framework which provides not only user interface components but also thread and networking classes. The server is highly threaded, allowing it to process a substantial number of requests simultaneously. Every connection, whether to a mobile device or a desktop application has its own thread which allows request processing to overlap, improving overall performance when the load gets heavy. However, the limits of the server have yet to be strained by our simple applications and the server has not been optimized for speed. Future work will gather statistics on throughput with varying loads and varying scenarios (wired or wireless) .

4.5 MobileConnect

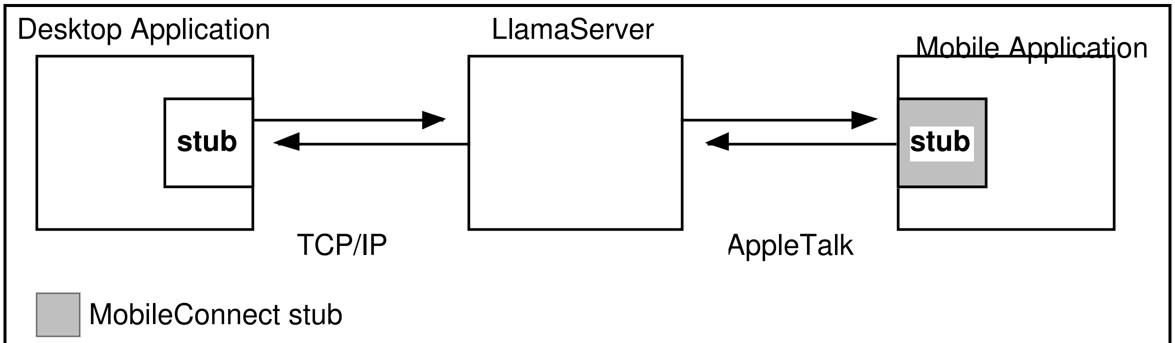

Figure 18 - MobileConnect's position in the infrastructure

The Newton can participate in the infrastructure in one of two ways, either as a client or a server. The software which handles the communications and protocols is a stub called MobileConnect contained within the currently running application (see Figure 18). The stub module has a simple external API and implements both client and server behavior, albeit in mutual exclusion. To add the capability to participate in the LlamaShare infrastructure to a mobile-based application, a developer simply adds the MobileConnect module to the application and makes a handful of function calls. Once the stub has been initialized, the application can then either serve information stored on the mobile device or read and write information to Global Soups stored on the LlamaServer.

As a client, the application running on the mobile device has access to any of the Global Soups and the information contained within them. To send information to the Global Soup, the application calls a function on the MobileConnect stub, passing a frame consisting of which soup to access, the list of frames to store, and a callback function to be called when the transfer is complete. This frame is illustrated in Figure 19. The stub then transparently handles the communication with the LlamaServer, sending the objects to the specified Global Soup. When all the frames have been transmitted, the mobile application is notified through the callback function. Retrieving frames from a global soup is roughly analogous, except that instead of passing an array of frames, the “frames” slot will be the destination for the incoming frames . In both cases, the application is unaware of exactly how the frames are transmitted (which is covered below in Section 4.7).

```
local clientSpec := {
        soupName: "Test Soup",
        frames: [ {a: "a", z:55},
                {c: "c", qq:{foo:"hello", zebra:"zzz"}},
                {e: "e", ee:"ee", eee:"eee"} ],
        onCompletion:
            func ( ) begin
                print ("****Client is finished****");
                GetRoot():SysBeep();
            end,
};

fFSM:StoreFramesToGlobalSoup ( clientSpec );
```

Figure 19 - A clientSpec frame for sending frames to a Global Soup

The application running on the mobile device can also configure the stub to accept and respond to incoming requests for information stored on the device. After the application puts the stub into server mode, no more interaction is needed between the application and the stub, which reads from the network in the background and processes its own events while the application's user interface is idle.

When acting as a server, MobileConnect handles several different access methods for mobile information on the local device. The most flexible is a text-based search, which searches a given soup for any frame containing the provided text. All frames with the matching text are returned. The next method, which works well in conjunction with the text searching, are "overviews" which are one or two line descriptions of all frames in a specified set. This set can either be an entire soup or the result of the previously mentioned find query. The overview is quite useful for allowing users to quickly scan a large volume of information in context, without having to transmit and display all of it over the network. Once the user decides which frame they are interested in, it can be loaded explicitly using the final access method. A particular entry on a mobile device can be requested by using its "Resource Id," an id unique to each piece of information on a device[2].

Resource id's are assigned to each frame in a soup by the MobileConnect stub at certain well-defined times, the most common being when the frame is transmitted from the Newton to the desktop. Resource id's are also added when frames are received from a desktop-based application and are to be stored in a soup. Finally, frames are tagged with their id when an overview is requested in order to allow applications to request an overview first, then the desired frame after the use has picked the one they are interested in, saving both time and bandwidth. At the time the frame is tagged, the frame is written back to the soup and is, as a result, permanently modified.

---

[2] this is different from the Newton's "_uniqueid" slot in that it is actually unique across all stores, both internal and card, where the system uniqueid is not.

Due to the flexibility of frames and soups, modifying a frame to add the resource id slot does not affect any other application which uses the frame. Applications are not aware of any slots which they do not explicitly use in the program, so new slots may be added at any time without causing a disturbance. Furthermore, since applications written to use soups don't rely on byte offsets and fixed-size records when writing and reading a file as done most traditional desktop-based applications, changing the frame in the soup does not wreck havoc by disturbing the carefully predetermined alignment of records.

4.6 DesktopConnect

LlamaShare enables the rapid development of desktop-based client applications which take advantage of mobile information by providing libraries which abstract the process of requesting and receiving mobile information to a function call. The DesktopConnect libraries, shown in Figure 20, are written for a particular language or OS and encapsulate all of the logic required to communicate with the LlamaServer over a TCP/IP connection and package the resulting mobile information into a structure suitable for the current development language.

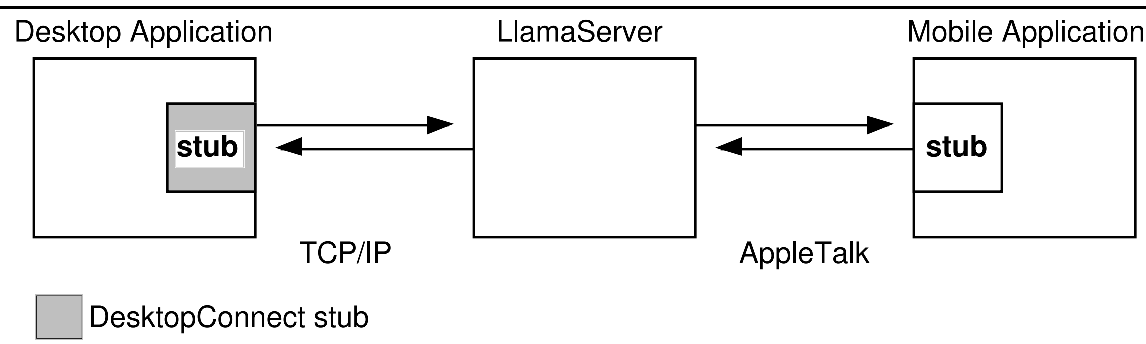

Figure 20 - DesktopConnect's position in the infrastructure

A single generic library which could be used regardless of the language or platform is not feasible because there is no consistent networking API which covers all platforms. Even within a single platform, different languages can vary greatly in their requirements on the runtime architecture and meeting all of those needs from one code base would be impossible. However, we have already built stubs to handle several of the more common languages, including C++ and Java. We certainly get the most mileage out of the Java library, which allows true cross-platform development and finally allows application developers interested in taking advantage of mobile information to use UNIX.

Accessing information from any mobile device in the environment is as easy as making a function call. The API provides a synchronous, RPC-style calling method which hides the complexities of writing client-server code, such as asynchronous callbacks, from the application. In the Java case, the function calls return a Java Vector, which is a list of objects, one for each piece of information retrieved from the mobile device. The application only has to iterate over this list to access the information.

Internally, the library communicates with the LlamaServer over TCP/IP. It sends and receives information using the protocol detailed in the next section. The most important role of the library is to inflate the frames as the come across the network. Recall that when the LlamaServer sends frames, it flattens them for transmission over TCP/IP using a recursive protocol which utilizes tags to indicate different attributes of the frame (see Section 4.5 for a complete description of the format of the outgoing byte stream). The DesktopConnect library reads the incoming byte stream, interprets the tags, and creates data structures through a process which is effectively the mirror of the flattening process.

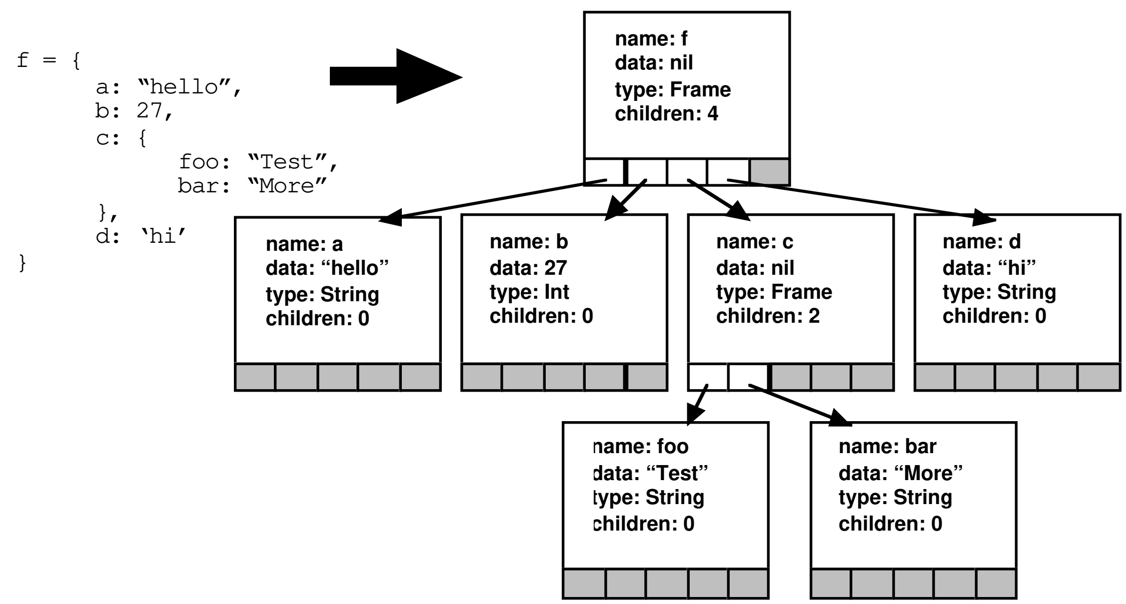

Figure 21 - Desktop representation of a Newton frame

The difficulty lies representing the frames, which are hierarchical data structures of unknown format, size, and content, in a format easily accessible by programmers in more static languages such as C++ or Java which do not support dynamic structures. In Java, for example, each Frame class consists of a Vector (Java's linked list container type) of other frames, which represent the information stored in the slots of the parent frame, as shown in Figure 21. In other words, a Frame has a list of children Frames, which themselves can have their own list of children Frames, ad infinitum. The recursive structure finally terminates when each frame consists of only leaf data types, such as strings or integers. These leaves are represented as Frame classes with no children, and instead contain the actual data. Frame classes which actually represent internal frames contain no actual data besides the children representing each slot (and thus have a nil data slot).

As with any information an application expects to interpret, it must know something about the information before it can proceed. Static languages such as C++ and Java rely on knowing the structure of the information at compile time in order to use byte offsets to impose a structure on an arbitrary stream of bytes. Frames on the Newton behave much differently since their structure can change arbitrarily, without the knowledge of the application. Consequently, an application interpreting Newton frames on the desktop cannot rely on byte offsets to access certain attributes within the frame, and must rely on a different approach: knowing the names of the desired slots. The LlamaShare library provides routines to inspect Frame objects by locating and returning the Frame class representing the requested slot (see Figure 22 for an example).

```
import llama.NewtFrame;
import llama.MobileConnect;

// assume we have the frame in variable f
try {
      NewtFrame c = f.GetSlot ( "c" );
      NewtFrame foo = c.GetSlot ( "foo" );
      System.out.println ( foo.GetData() );
}
catch ( SlotNotFoundException e ) {
      System.out.println ( "Slot not found" );
}
```

Figure 22 - Java code to access slot f.c.foo in the frame from Figure 21

4.7 Intercomponent Communication

This section covers the protocols between the different components described in the previous sections. A typical request for mobile information goes like this:

1. A desktop-based application makes a function call in the DesktopConnect library. The library packages and sends over TCP/IP the appropriate 4-letter command (detailed in Appendix A) and its parameters to the LlamaServer. The client application is suspended until all of the results are returned, as with a typical RPC.

2. The LlamaServer reads the information off of the network and figures out which mobile device is the recipient of the command (this is part of the parameters). Next, it sends a command over ADSP (AppleTalk) to the appropriate Newton to the MobileConnect stub, acting as a server, running on that Newton.

3. The MobileConnect stub collects the requested frames and sends them back to the LlamaServer (flattening is handled transparently by Apple's FDIL libraries).

4. The LlamaServer then flattens the frames into a byte stream and sends it back to the DesktopConnect library where it is inflated back into a data structure. The frames are put into a list (such as a Vector in Java) and returned as part of the normal return mechanism of the initial function call.

4.7.1 The Registration Process

There are several problems which need to be addressed when implementing the communications protocol between the server and the desktop and mobile components. The following issues motivated the final design:

- There is only a single logical connection to each mobile device that everyone must share. Unlike desktop systems, mobile devices do not generally expose OS threads to user applications. As a result, handling multiple simultaneous connections is unwieldy.

- Information read from the connection to the mobile device must be shared between the thread communicating with the desktop client and the thread communicating with the mobile device.

- The mobile device can be a client in its own right. The server must be able to differentiate between a response to a query and a request for information originating from the mobile device over the same connection.

To this end, the LlamaServer implements a registration mechanism in order to maintain control over the multitude of threads and messages involved during any transaction.

Inside the server, each connection, whether to a desktop client or to a mobile device, is represented by a thread. A thread associated with the desktop client, called a Desktop Thread, is created when a client application connects over a well-known TCP port.

Its main duty is to read and write information to and from a mobile device or a global soup over the TCP connection. A thread associated with a mobile device, called a Mobile Thread, is created when a mobile device connects to the LlamaServer over ADSP. Unlike the more general Desktop Thread, its duty is to sit on the CDIL pipe and watch for incoming data.

The registration process is helpful when these two varieties of threads must interact. When a Desktop Thread wants to communicate with a mobile device (to request a set of frames, for example) it first locates the Mobile Thread associated with that device using a global lookup table, and then registers its interest by sending a “register” message to that thread. The Mobile Thread returns a unique identifier (called a “cookie” after identifiers used in web servers) which acts to correlate incoming responses from the mobile device

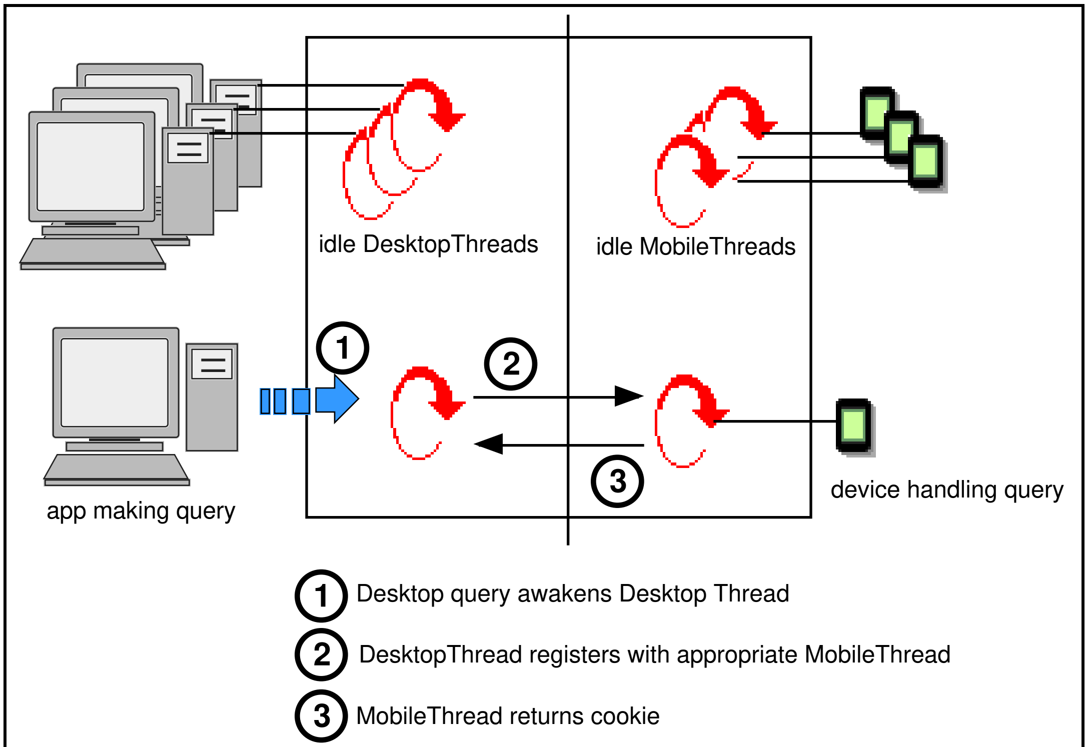

Figure 23 - DesktopThread/MobileThread registration process

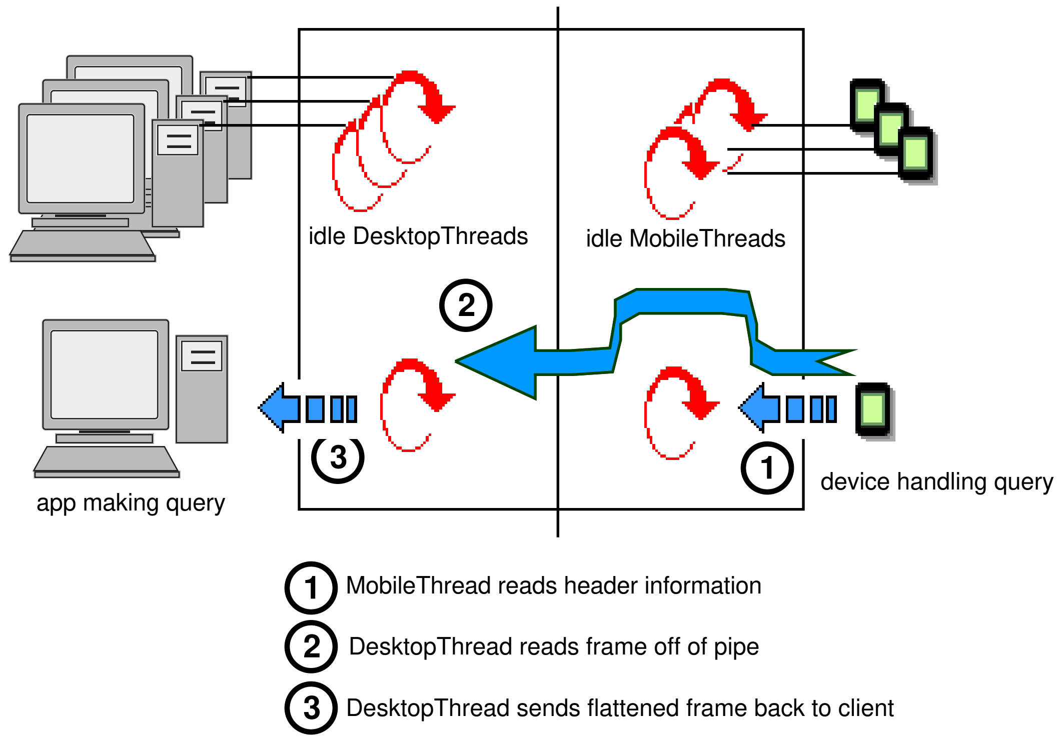

Figure 24 - Handling of response from mobile device

with the appropriate Desktop Thread. The Desktop Thread next sends this cookie to the appropriate mobile device along with the command requesting (or sending) the desired information and then goes to sleep, awaiting the response. This process is illustrated in Figure 23.

When the mobile device responds, it prefaces the frame with a header block consisting of a tag and the cookie sent to it by the Desktop Thread who made the request. The tag, described in more detail below, indicates that this message is a response to a query from a desktop client and the cookie identifies to which thread this information is intended. At the server, the Mobile Thread, busy watching the CDIL pipe, notices the incoming response and reads in the header information (and only the header information). It determines that the message is a response (by the tag) and scans its internal list of registered threads to determine which Desktop Thread has this cookie, and signals that Desktop Thread to wake up.

The two threads are now finished interacting, which greatly simplifies the programming model. Once active, the Desktop Thread can now directly read the incoming data off the pipe, freeing the Mobile Thread to return to monitoring the pipe (see Figure 24). The Mobile Thread no longer needs to participate in the process because it does not require access to the information requested by the Desktop Thread. This simplifies the programming model because only one thread has access to the data and avoids the necessity for using complicated and error prone synchronization methods, such as semaphores, to coordinate access when passing shared information between threads.

The process then repeats itself until the communication is complete. If any more information needs to be retrieved from the mobile device (such as the next frame in the requested set), the Desktop Thread then goes back to sleep and the process repeats when the next message with the same cookie comes along. When the Desktop Thread is done reading information from the device, it sends an "unregister" message to the same Mobile Thread where it registered before and then disconnects (or makes a new request). The same cookie is never reused, thus new requests require reregistration.

The above process handles the case where a desktop client makes a request for information from a mobile device (i.e., MobileConnect is in server mode), but ignores the situation where a mobile device wants to be the client. Recall that there is only one connection to any mobile device so any client requests must go through the same pipe as responses to queries. Going by the above scenario, when an incoming message arrives at the LlamaServer, some thread must be awakened to handle the information since this is not part of the Mobile Thread's job -- only there is no Desktop Thread to wake up. As a result, several small enhancements need to be made.

As mentioned briefly before, we need a special tag to distinguish a request for information originating from a mobile device from a response to a query issued by a Desktop Thread. To accomplish this, the header of every message sent from the mobile device has, in addition to the cookie, a special 4-letter tag that indicates its function: 'RSVP' for a response and 'RQST' for a request. When the Mobile Thread sees the 'RQST' tag, it must take another course of action. Instead of trying to wake up a registered Desktop Thread, the Mobile Thread forks a new thread, called a Mobile Client Thread, which serves a similar function to the Desktop Thread. Once created, this new thread registers itself with the Mobile Thread who created it and then sends the new cookie back to the mobile device. Any further communications from the mobile device now use this cookie to associate themselves with the Mobile Client Thread.

As a result, the infrastructure can utilize the registration process for coordinating the access to shared information between threads regardless of which side instantiates the transaction. Not only does it simplify the thread interaction, it promotes code reuse in both the LlamaServer and the MobileConnect components since the behaviors and communication protocols are consistent.

One final note. Even though the MobileConnect component on the Newton can only process a single request at a time, the infrastructure is present to handle multiple, simultaneous requests over a single connection. This is simply a limitation of the Newton applications not having access to multiple threads, not of the LlamaShare architecture. In fact, most commercial mobile devices would also have this problem.

4.7.2 Protocol In Detail: Desktop to LlamaServer

The following is a detailed description of the protocol between the DesktopConnect library on the desktop side and the LlamaServer when the client requests mobile information. The scenario is very similar when information is being sent to a mobile device.

1. A client application running on a desktop-based workstation instantiates a “connection” object defined by the DesktopConnect library. The job of this object is to handle all communications between the application and the LlamaServer. When created, the connection object opens a TCP connection to the LlamaServer on port 5000 (a pre-agreed upon port number).

2. The LlamaServer has a thread whose job it is to watch TCP port 5000 and create a new Desktop Thread when a new connection is requested from this port. The newly created Desktop Thread is now in charge of handling this connection and the main thread returns to watching the connection port.

3. The client application makes a method call of the connection object (or just a function call in a non-object oriented language) to access information on a mobile device. The client is suspended until the requested information has been totally gathered, just like an RPC.

4. The DesktopConnect stub sends the appropriate 4-letter command to the server, followed by any applicable parameters supplied by the client application when making the function call. The DesktopConnect stub blocks, waiting for a response to the query.

5. On the server, the Desktop Thread in charge of the connection to the client application reads the command and the parameters from the network. It then

determines which server routine to execute based on the command and executes it. In most cases, this involves communicating with a mobile device to get or put information (accessing Global Soups are the exception). If so, one of the parameters will include the name of the mobile device to access.

6. The server maintains a global list of connected devices which contains, for each device, a pointer to the Mobile Thread in control of that device and a semaphore guaranteeing only one request may be processed at a time. The Desktop Thread scans this list searching for the requested device. When found, it locks the semaphore and then registers with the appropriate Mobile Thread (the registration mechanism is described in the previous section). The semaphore is useful because MobileConnect can only handle one connection at a time due to limitations in the Newton OS. Once devices provide better support for threading, this restriction can be lifted and is not an inherent limitation in the server.

7. Once registered, the Desktop Thread continues executing the requested command by sending the appropriate 4-letter command to the mobile device along with the cookie using the CDILs. The thread then goes to sleep, awaiting the response (shown in Figure 25).

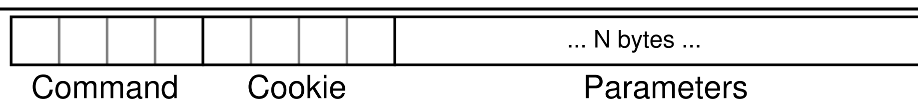

Figure 25 - Message sent from DesktopThread to mobile device

8. When the response arrives, the Mobile Thread will wake up the Desktop Thread. The Desktop Thread then reads the data off the pipe using the FDILs. The frame is read in as unbound data (see Section 4.3 Newton Overview for a description of the FDILs and unbound data).

9. The Desktop Thread next flattens the frame (see Section 4.4 LlamaServer for a discussion of flattening) and sends the byte stream to the client.

10. When the client receives the message, the DesktopConnect library inflates the frame into a data structure and stores it in a list. It then blocks, waiting for the next message.

11. After sending the frame to the client, the Desktop Thread tells the Newton to send the next frame, and goes back to sleep, awaiting the next frame from the Newton.

12. The process repeats until all the frames have been read, at which point the Desktop Thread unregisters itself. On the application side, the DesktopConnect library returns the list of frames it has been building using the standard return mechanism for functions.

4.7.3 Protocol In Detail: LlamaServer and MobileConnect (server mode)

The following is a detailed description of the protocol between the LlamaServer and the MobileConnect stub when acting in server mode. These events are prompted by a desktop client requesting information from a mobile device, described above.

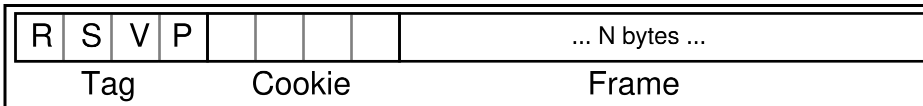

Figure 26 - “Response” message received by LlamaServer

1. MobileConnect, when acting as a server, simply waits for commands over the pipe between the LlamaServer and the mobile device. When one comes in, MobileConnect reads the command, the cookie, and the parameters.

2. MobileConnect then executes the particular command, accumulating the resulting frames into an array. These frames are then sent, one by one, using the following protocol: if there is a frame to send, send the string ‘OK’ (with the appropriate header information). Otherwise, send the string ‘NO.’ When this message is received by the LlamaServer, the Mobile Thread will wake up the appropriate Desktop Thread to read OK or NO. If NO, the Desktop Thread cleans up and finishes the command. If OK, the thread goes back to sleep, awaiting the frame itself.

3. The frame is then transmitted from the Newton to the LlamaServer where the header information is again read by the Mobile Thread and the appropriate Desktop Thread is awakened. The Desktop Thread then reads the frame off the network using the FDILs, then tells the Newton to continue by sending an acknowledgment string via the CDILs. This message is shown in Figure 26.

4. MobileConnect then reads the ACK string and continues until there are no more frames to send. At that point, it goes back to waiting for the next command.

4.7.4 Protocol In Detail: LlamaServer to MobileConnect (client mode)

When the Newton wants to act as a client, things work differently. Since the implementation of MobileConnect's internals use a finite state machine based on the Newton's event model, control must be returned to the main event loop in order for anything to happen. For this reason, synchronous RPC-style calling conventions cannot be used as they are on the desktop side. On other mobile devices, the MobileConnect might be implemented differently, allowing the implementation of synchronous calls.

The following is a detailed description of the protocol between the LlamaServer and the MobileConnect stub when acting in client mode.

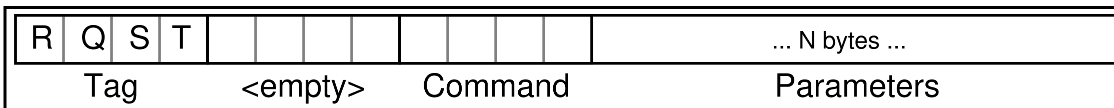

Figure 27 - "Request" message received by LlamaServer

1. The client application running on the Newton makes a call to the MobileConnect stub passing a frame, called a client spec, containing the information to send or receive and a completion routine to be executed when the communication is complete.

2. MobileConnect then sends a message, formatted with a header like all the others, to the LlamaServer where the header is read by the Mobile Thread associated with that device. However, in this header, the tag reads 'RQST' instead of 'RSVP' to indicate that this is a new request for information, and not a passive response to a query from the desktop world (see Figure 27). The Mobile Thread forks a new Mobile Client Thread to handle the interaction with the Newton and participate in the registration process.

3. The first order of business of the Mobile Client Thread is to send the cookie (obtained by registering above) to the Newton so that any subsequent messages from the Newton will be handled by waking up this thread.

4. The process of receiving frames is the same as above, except with the server prefixing each frame with "OK" or "NO" to indicate if there are any more frames to follow. In fact, the same code can is used.

5. When the request is complete, MobileConnect stores the requested frames in a slot provided within the client spec parameter calls the completion routine.

4.8 Summary

This chapter went into detail about the major components of the LlamaShare infrastructure (server, mobile stub, and desktop stub) and how they interoperate. The next chapter discusses why we made certain design decisions regarding the architecture and many of our assumptions about how it will be used.

CHAPTER V

DESIGN DECISIONS, HURDLES, AND LIMITATIONS

This chapter explains the rationale behind many of the design decisions and shows how the devices themselves contributed to many of the problems encountered. Some of our decisions might seem narrow in the overall picture, but considering the limitations of the devices we were dealing with, they make more sense. Many of the decisions were made because of rather strict time constraints, which forced a single course of action. Better solutions to these issues will be discussed in the “Future Work” chapter. This discussion will also expose many limitations of the LlamaShare infrastructure. Again, proposed solutions will be addressed in the “Future Work” chapter.

There are 5 major design decisions, each introducing its own hurdles:

- Some information is kept permanently on the mobile device
- Mobile devices always connected
- A centralized server (LlamaServer) instead of point to point communications
- Client applications explicitly written to handle mobile data
- Assumes a homogeneous mobile environment

5.1 Some information is kept permanently on the mobile device

There are two ends of the spectrum in terms of how mobile devices store information: thick clients and thin clients. A thick client, equivalent to a non-networked desktop machine, holds all of its information permanently and doesn't rely on any outside source to constantly feed it data or keep it up to date. Thick clients are totally self-contained units. A thin client, on the other hand, can be likened to a Network Computer (NC) which relies on some other source for its programs and data. A thin client is little more than a window onto information stored remotely.

Instead of trying to go with either extreme, LlamaShare assumes that some information will be stored on the mobile device, but other information is required from the environment in order to augment its view of the world. Thick clients, which have no need to share and accept new information, are of limited usefulness in a world where people work with multiple devices. In a fully connected world, thin mobile devices such as ParcTab [18] would be the best solution, but retail hardware (mostly PDAs) are not designed to be used as such. Achieving such a system with Newtons or Pilots would require substantial development on top of the existing systems, and ignores the built-in capabilities which already exist in these devices.

With the assumption that some data will permanently reside on the mobile device comes the requirement to be able to uniquely address information on these devices from the desktop. This prompted the "Resource Id" solution described in Chapter 4, which assigned to each piece of information on the Newton a tag uniquely identifying it on the device. It is apparent that the NewtonOS engineers did not have remote information addressing in mind when they developed the Newton's storage system. While the Newton does have a "uniqueId" slot in each frame stored in a soup, it is unique only within the physical store (internal or card) in which the frame resides. The result is that two pieces of information can actually have the same unique id on the same device, one residing internally the other on a card. This was unacceptable for our purposes, as MobileConnect would be unable to determine which piece of information the user wanted.

Our solution of using resource id's is not trouble free. Since the id's are added without the knowledge of the NewtonOS, they cannot be totally relied upon. One major problem is that the resource id for any given frame is only unique with respect to other frames on a single device. Just beaming a frame from one MessagePad to another will cause major problems if the beamed frame already has a resource id that exists on the destination device. Not having access into the internals of the OS, there is no way to guarantee uniqueness when the user transfers information using a mechanism other than LlamaShare.

Solving this problem requires one of two solutions. The first would be to tag each piece of information with more than just an id number, such as appending the name of the unit to the id. This would allow the information to travel from device to device, while still retaining its uniqueness. However, this situation breaks down quickly when two people have the same name (the name of the user is the name of the unit). Instead, we could append a serial number (unique to each MessagePad), but only the newest MessagePads have such a feature and this would not be an adequate solution for any other devices such as the Pilot. The second solution would be to patch the storage system of each device, which is neither portable nor simple. Lobbying for PDA designers to support this kind of tagging might be the best long-term solution.

Finally, frames need to be given a resource id before they can be accessed individually. This "chicken before the egg" situation is caused, again, by the fact that our assignment of resource id's are based on using LlamaConnect, not the OS. One way around this shortcoming is to request the entire soup (or an overview) beforehand which will assign id's to those frames which do not already have them. However, this requires sending everything once, which takes time. An alternative solution would be to have an application on the MessagePad which scans every soup and updates the resource id's, but this would require user intervention. Again, obtaining OS support for the data identification infrastructure LlamaShare needs is a must.

5.2 Mobile devices always connected

Obviously, a mobile infrastructure which assumes that the devices will always be connected has inherent limitations -- today. As we look towards the future, connectivity is getting cheaper and easier. Wireless communication is booming. Take laptops for example. With digital cellular modems, users can go out into a field and still be on their company's Ethernet. Instead of spending time on research where communication is fragile and transient, LlamaShare looks shortly into the future where these concerns are not a problem in order to address other issues regarding mobile computing. To simulate this, LlamaShare makes use of wireless technologies which allow devices to freely roam within a limited area (50 meter radius).

The decision to assume that mobile devices are always connected to a shared network was made for several reasons. Primarily, it assures that programmers writing applications to take advantage of mobile data have immediate and uninterrupted access to this information, which simplifies programming and facilitates the rapid development of these kinds of applications. Before developers can get a feel for what applications are possible, they need an infrastructure which makes using mobile information as simple as possible. Otherwise, they too easily get bogged down in the details and never get to the interesting part -- the application.

Unlike tethering devices to desktop workstations with a serial cable, getting the devices on the network proved a much greater challenge than anticipated. Hooking up tablet-sized devices to the network was no problem, since many of them were Windows-based and could take advantage of the same drivers/cards available for their laptop cousins. However, getting the smaller PDA’s on the network was more difficult and was one factor as to why the MessagePad was chosen over the Pilot. Neither device could be placed on our Ethernet, even with add-on cards, and neither the MessagePad nor the Pilot could easily handle TCP/IP (at most only via a PPP connection). The MessagePad, however, could do AppleTalk which allowed us to quickly connect multiple devices to the LlamaServer over our existing network. Unfortunately, this ties LlamaShare almost exclusively to the MessagePad, as no other devices use AppleTalk. Finally, the DILs do not yet work with TCP/IP (while they do with AppleTalk), allowing us to take advantage of existing communication libraries to get the communication layer of the LlamaServer completed quickly.

Using AppleTalk also allowed LlamaShare to quickly move from wired to wireless solutions. The connectivity was initially developed using wired connections over our local EtherTalk (AppleTalk over Ethernet) network, but we recently purchased NetWave wireless access points [31] and Dayna PCMCIA cards [32] for the Newtons. The access points broadcast Ethernet over a very high frequency RF signal which the wireless PCMCIA cards can access. The card’s driver reads the AppleTalk bundled inside the Ethernet and provides the Newton with the illusion that it is connected. While we would prefer that the Newton did plain Ethernet, using AppleTalk gave us the opportunity to work in a wireless mode much more easily than serial solutions. Once the Newtons were untethered, the restriction to being fully connected didn’t seem so impractical, as users could move around within a 50m radius inside the building and still be connected. Chaining multiple access points together would yield an even larger roaming range.

Two recently announced developments make us more hopeful about being able to move away from AppleTalk. Apple is adding Ethernet support in it’s 2.0 release of the

Newton Internet Enabler (NIE) [33]. This will allow the MessagePad to speak TCP/IP over an existing Ethernet network without having to resort to a dial-in PPP connection as per the current solution. Also, PalmComputing has announced that its next version of the Pilot will also allow synchronization over TCP/IP. Exactly how the Pilot gets on the network has yet to be disclosed. In any respect, as mobile devices get more and more network-savvy, the problem of being restricted to the MessagePad and AppleTalk vanishes.

5.3 Centralized server instead of point to point communications

LlamaShare utilizes a centralized server as the contact point between desktop systems and mobile devices, mainly to bridge the gap between the different networking protocols used by the desktop and mobile environments. TCP/IP has almost become a ubiquitous standard for inter-application communication over both inter- and intranets. Most developers with experience in writing client/server applications for accessing remote information are familiar with TCP/IP. Even Java has included TCP/IP-based networking classes along with the core classes that comprise the language.

However, the LlamaShare server does much more than act as a simple TCP to AppleTalk router, which is important as the infrastructure moves away from AppleTalk. The LlamaServer is responsible for maintaining the Global Soups which allow multiple devices to share information in a collaborative environment. Without a central contact point, this would not be possible. Additionally, the LlamaServer provides a single contact point for multiple mobile devices which may move around, appear, and disappear from the environment. Having a fixed IP address for desktop clients to connect to frees the client from having to worry about where the mobile device is when making requests for mobile information. The next phase of development (discussed in the next chapter), allows linking multiple LlamaShare servers together over the Internet, providing seamless access to mobile devices across the world by only having to connect to a single, relatively static, IP address. With direct point to point communication, achieving this would be difficult.

The problem with any centralized system is that the point of convergence becomes a bottleneck. In the current architecture, the LlamaServer is an obvious bottleneck since every request and acknowledgment must pass through it. Despite the performance penalties, we believe the extra functionality that a centralized server provides outweighs any performance problems. Furthermore, the proposed extensions to link multiple servers over the network relieves the burden of each individual server while maintaining the illusion of a centralized server to applications.

5.4 Client applications explicitly written to handle mobile data

Unlike the synchronization model, which provides built-in translators from the mobile device's native data format to those used by popular applications, no such translation scheme is part of LlamaShare. That means that users cannot just hook up LlamaShare and expect to be able to read their calendar using Now UpToDate. Applications must be specifically written to know to request the information from the LlamaServer and then understand the data format used by the individual mobile devices. We believe that having direct access, without forcing the user to pre-download all mobile information prior to using it, allows developers to write much more flexible and powerful applications which take advantage of mobile information (CyberDesk [8] is one example).

There are two options for presenting such information to users: a generic interface which can handle all possible information formats, or a custom application built to specifically handle each different data format. The generic approach, taken by programs such as Revelar Connection Utility [24], allows access to all Newton information in one application, with a consistent interface. However, this interface gives no semantic meaning to the information and just displays it in its purest form - a skeleton of frames and slot names. While all the information may be on the user's screen, making any sense of it is a totally different story. Being so generic, it cannot present the information in a meaningful form. Custom applications, while more difficult to build and more costly in terms of development time, can present information to users in a rich form which users can understand.

The two desktop applications presented by this thesis (see Chapter 3) are both explicitly written to understand Newton information and to access this information through stubs which speak to a LlamaServer. While most (if not all) of the communication with the infrastructure is handled via stubs, developers are still aware that they are connecting to a LlamaServer and that the data the stub returns is Newton information which requires special processing. We feel that this allows developers to write applications which makes manipulating mobile information easier for users by hiding the burdens associated with using such information within the application. Both CyberLlama and our extensions to CyberDesk illustrate how clients written to understand mobile information provide a more streamlined user experience than RCU and synchronization.

Finally, having an understanding of the mobile information allows the application to provide more input to the infrastructure about information that goes together. Systems like Wit II [15] and Odyssey [10] are investigating application-side extensions which allow the underlying infrastructure to make better decisions about caching and prefetching. Such optimizations would be impossible to do well (if at all) for applications which were not specifically written to understand the nature of the mobile information they were manipulating.

5.5 Assumes a homogeneous mobile environment

Several of the previous design decisions pointed LlamaShare to using the MessagePad as the primary mobile device accessible by the infrastructure. A combination of easy networking (via AppleTalk) and dynamic data storage (via the soups) made the MessagePad a perfect choice to prove the feasibility of the infrastructure and the usefulness of the applications.

One of the key features which made the MessagePad so attractive was the flexibility of the soups. Frames, being dynamic in nature, can be modified by adding slots without affecting the application to which the frame actually belongs. Adding resource id's to each frame in a soup takes full advantage of this capability, and the applications on the MessagePad which use this information are none the wiser because they can simply ignore slots which they do not understand. Additionally, soups are easily searchable programmatically, and much more flexible than a plain text-based search. Implementing the "find" command from the desktop was as simple as opening a soup and calling its find routine with varying parameters (such as only look in the "title" slot or the "last name" slot). Finally, since frames carry with them the types and slot names of the data, introspection once on the desktop is much easier, especially when the desktop doesn't know the exact format of the information it is receiving. If a client only received a chunk of binary data, it would need to be hard-coded to that data format. Any changes to the exact pattern of bits would break the application. Not so with frames, which allow the ordering of slots to change (or even go away entirely) without necessarily affecting applications.

However, we realize that there are more mobile devices out there than just MessagePads. Currently, applications have to be written assuming that the information they receive from the LlamaServer are Newton frames. This assumption completely breaks down when the mobile device is something other than a MessagePad, such as a Pilot or a cell phone. Furthermore, the internals of the LlamaServer currently understand only Newton frames. Support for any other device would have to be written, and then grafted onto the “flattening” protocol used to exchange frames between the server and the desktop client. The Future Work chapter describes some ideas we have to circumvent this problem and extend into a heterogeneous mobile environment.

5.6 Summary

This chapter presented the key design decisions which drove the development of the LlamaShare infrastructure and applications. Several of the decisions, such as the choice to use the MessagePad exclusively and the use of AppleTalk over TCP, were based on limitations of existing hardware which should disappear as these devices mature. Other decisions, such as a centralized server and custom client applications, were motivated by our desire to build a rich set of easy to use applications which would leverage our infrastructure to allow users to collaborate in mobile and desktop environments with mobile information.

The following chapter, Future Work, details how we plan to expand the infrastructure so that it can better support more platforms, more devices, and important issues such as privacy.

CHAPTER VI

FUTURE WORK

6.1 Heterogeneous Platforms and Data

As discussed in the previous chapter, the current LlamaShare infrastructure is tied very heavily to the Newton because of the flexibility provided by the soups and ability to easily inspect the contents of arbitrary frames. Provisions must be made, however, to allow for other kinds of mobile devices to connect to the LlamaServer and share information with desktop systems, or with other mobile platforms through Global Soups. Unfortunately, the communication protocols and data formats of the more popular mobile devices (Newton, Pilot, WinCE) are all different to the extent that makes data interchange very difficult.

At minimum, specific sections of the LlamaServer must be rewritten to communicate with each new platform. Currently, the thread within the server which speaks to the Newton, called a MobileThread, relies on the DILs from Apple which are Newton specific. In order to accommodate other devices, the functionality which the DILs provide (communication and data transfer protocols) must be duplicated. A generic solution, such as developing our own protocol, has the benefit that we do not need to rely on vendors for libraries, allowing us to integrate devices (such as pagers and mobile phones) which may not have such communication libraries. On the contrary, using custom libraries for each platform would allow us to take full advantage of the communication abilities of a given device as we would not be restricted to the lowest common denominator at the time the protocol was developed. In either respect, the rest of the server does not require modification. Each connected device has its own thread (similar to a MobileThread) running code to communicate directly with that device. When requests arrive at the server, these threads will take part in the registration mechanism just like the MobileThreads do now for the Newton.

While the infrastructure adapts easily to communicate with other devices, the different data formats used by each device pose a greater problem. One solution might rely on a “meta-format” which describes the internal structure of each piece of information. A possible first step would use meta-content description languages like MCF [34] from Apple Computer, the underlying layer to their HotSauce technology, to provide to desktop applications a description of the semantic content of the information transmitted. For example, a desktop application would receive a piece of data with a MCF description that tagged it as a “name” or a “note.” The next step, once the content is identified, is to add a “format description” to each piece of information. This description, similar to a legend on a road map, details how to interpret and inspect the information if the format is unknown to the application. The intelligence to parse the format description would be built into the stub which already must be used to communicate with the LlamaShare infrastructure, making it as transparent to the application as possible.

An alternative solution would store the mobile information in a common format in an object-oriented database, which provides the typing features of a meta-content language for free. Incoming information would pass through translators for the appropriate platform to translate it into a format generic to all platforms. This database would replace the Global Soup, as it would be accessible to all devices on the network (like an ordinary database). One problem with this approach is that it might not be able to represent all of the data formats of every available platform and might have to take a least common denominator approach, causing applications to lose much of the richness of the original data. Furthermore, translating the information back to the individual native formats for storage on the devices themselves would have the same problem, causing the act of storing information in the database to be lossy. Finally, unlike the MCF format which can grow dynamically as new devices add new data formats, the common format might not be able to handle the requirements of new devices.

6.2 Caching

In a perfect environment, the devices would have infinite battery life and infinite wireless range. However, especially in the short term, we have neither. Continuous connectivity drains batteries very quickly and even if an entire building can be covered with a wireless network, there are times when the user will leave the environment. The infrastructure should be able to continue to provide information about the departed device to those who ask long after it goes off-line. The easiest way to solve this is by caching the device's contents on the LlamaServer.

While not intended to handle the flexibility that regular soups provide in terms of access and searching, Global Soups could be extended to support these operations in order to allow fulfillment of queries even after the device is no longer in the environment. Using the same mechanism as the Newton's synchronization tool, the Newton Connection Utilities [5], information is pulled off the device while the device is idle and synchronized with the current contents of the Global Soup. When the user removes the device from the network, all subsequent queries would be handled by the Global Soup without the device having to be connected.

An interesting side-effect of caching the contents of the device (either in their entirety or selected portions) would be the ability to remove the "personal-ness" of the mobile device. Instead of these devices being restricted to a single user, they could be scattered around an office and any user could pick up any device which was closest to them. A setup similar to ActiveBadges [35] could be used to automatically determine which user picked up the device. The device would then download the particular user's information from the Global Soup and make it available from that mobile device. This goes more to the side of a "thin client" which acts more as a window into information stored elsewhere, but introduces the concept of a Group Digital Assistant (GDA).

In some respect, synchronization is the most basic form of caching. What we are proposing, however, goes beyond synchronization when a device is currently connected. Our caching mechanism, for example, would be able to detect that requested information had been previously requested and had not changed on the mobile device. As a result, the copy of the information cached in the Global Soup would be used instead of going back out to the device. In the case where the information on the device had changed, the infrastructure would purge the old information from the cache and request the new information from the device automatically, unlike synchronization which requires explicit re-synchronization by the user. In summary, our caching mechanism would be similar to synchronization when the device was not connected, but would gain the benefit of having direct access to new information when it was connected.

6.3 Security/Privacy

The privacy model currently in place can be best described as a "participatory" model. If the user wants access to the information of others, they must provide access to their own information as well. For the most part, this makes sense, as users who are interested in group collaboration already do things like share calendars on-line and post notes as to their location on their door or screen saver [36]. Spreitzer and Theimer [37] claim such "friendly environments" exist where there is mutual trust between participants. However, all information should not necessarily be freely accessible, and they go on to define privacy to mean that information about a person remains known only to that person unless they explicitly hand it out to someone else.

In response to this, the key area of future work should limit what data can be accessed on the device while it is connected to the environment. There are two granularities in which this can occur. First, users should be able to specify which soups are either shareable or off-limits. For example, office co-workers would probably want to share their "Dates" soup containing their calendar information but not their "Pocket Quicken" soup which contains how much money they have in their respective accounts. At a finer granularity, users should also be able to specify which items within a particular soup can be accessed. This gives users the ability to publish notes taken at meetings, but not notes of a more personal nature. Either of these methods could be easily implemented at the MobileConnect stub, customizable with the appropriate user interface. MobileConnect can be easily extended to reject requests for information in soups designated off limits or even reject requests for particular resource id's which are not public. Additionally, the user could be given this choice, allowing a case-by-case veto as each request is made.

Security and authentication pose more of a problem in the infrastructure. It is easy to spoof an AppleTalk name on a network so unsuspecting users might connect to the wrong program, written by a spy. Once connected, the spy could ask the Newton for any information it wanted (as long as it was not designated as off-limits by the privacy mechanism). Mechanisms such as Kerberos [38] or public-key cryptography systems could be employed to help authenticate users who request information.

6.4 Multiple LlamaServers

Mobile devices, by nature, move from place to place as users travel. It would be useful when away from the office to still allow a particular device to be in the environment and be accessible as if it were connected in a local LlamaServer's network. Unfortunately, patching into an AppleTalk network from the outside is difficult at best. A better solution involves multiple LlamaServers, each running at a different location within their respective networks. Distributing the devices among multiple servers also has the effect of reducing the bottlenecks mentioned in the previous chapter.

A separate registration service, similar to the Domain Name Service (DNS), could be employed to centrally gather the IP address of the LlamaServer to which each device is connected. The DesktopConnect stub would first contact the name service to find out the IP address of the current LlamaServer and then once that information is provided, connect to the appropriate server for the device. This solution would not create much additional work for the individual servers and would probably even reduce the individual loads since the servers would no longer be as much of a bottleneck. The only additional work would be to keep the name service up to date which would require transmitting updates when a device connected or disconnected. In addition, multiple name servers could be used in order to reduce the bottleneck at each name server, similar to the way DNS works.

As a result, client applications would have access to any mobile device, regardless of its physical location, through a single contact point. Each application needs only two relatively static pieces of information to access information from that device: the IP address of the closest name server and the name of the Newton. This allows individual mobile devices to transparently roam from server to server without disrupting applications or CyberItems which may be hard-coded to talk to a particular name service.

6.5 Criteria for UI evaluation

To date, we have not performed any user testing to support the claims we make about the usefulness of having transparent access to mobile information. Some questions which we would like to answer by getting real users to use applications built on top of LlamaShare are:

- Now that information can reside anywhere, where do users store their information?
- How often do users integrate mobile information into desktop tasks?
- What are the most common kinds of mobile information accessed from the desktop?
- Do users organize their mobile information in the same places in which they organize their desktop or Internet information?
- Do users create information on the desktop and want it on their mobile device for later access?
- Does sharing mobile information aid in group communication?

6.6 Extensions to applications

The applications presented in Chapter 3 never advanced much beyond the "proof of concept" stage because of our desire to demonstrate as many different kinds of applications as possible in the short time frame. As a result, they are functional, but far from polished. Additionally, there are features which we would like to add to each application to round out its functionality and make it even more useful.

In both the CyberLlama and CyberDesk applications, the components used to view the mobile information do only that -- view. The LlamaShare infrastructure supports writing changes back to the mobile device, but the applications themselves do not support it. We would like to add the following:

- Extending the components to allow modifying mobile information and then storing it back to the Newton.

- Allow users to generate content from scratch on the desktop side and save it on the Newton.

Currently, the selection of viewers for mobile information on the desktop is rather limited. CyberLlama implements only a “Notepad” viewer, and the CyberDesk applets can only display “Notepad” and “Names” information. The Newton, however, has many more data formats which users would benefit from being able to access and modify on the desktop:

- A “Calendar” component which would allow users to view and modify the calendar information from a given Newton. CyberDesk could be expanded to show the calendar of the user whose name is selected in an email window by directly accessing the calendar information off of that Newton.

- An application to read the calendar information off of several Newtons and display an aggregation of everyone’s calendar info in one place. This would make scheduling group meetings much easier.

- CyberDesk has many built-in types which we would like to support, such as dates, phone numbers, and email addresses. Writing services for CyberDesk which pull the appropriate information from the Newton would make the Newton services as complete as the Internet services.

6.7 Conclusion

This thesis presented an infrastructure which allowed the rapid prototyping and development of applications which take advantage of mobile information. Additionally, it presented several applications which streamline the process for users to access their information off of their mobile devices and seamlessly integrate mobile information into desktop tasks and applications. Finally, it described an infrastructure which allows group collaboration applications to be built on mobile devices.

APPENDIX A

TABLE OF COMMANDS

Command	Name	Description
NEWT	Get Newton List	Returns a list of the connected MessagePads
LIST	Get Soup List	Returns a list of all the soups on a given MessagePad
SOUP	Get Soup	Returns all frames in the specified soup on a given MessagePad
OVER	Get Soup Overview	Returns an overview (2 line summary) of every frame in the specified soup on the given MessagePad
ENTR	Get One Entry	Returns one frame from a specified soup on the given MessagePad
FIND	Find Text	Returns all frames that contain the specified text on a given MessagePad
FOVR	Find Text (overview)	Returns an overview of all frames that contain the specified text.
STOR	Store Frames To Newton	Stores the incoming frames in a specified soup on a given MessagePad
GLOB	Store Frames To Global Soup	Stores the incoming frames into the specified Global Soup (does not handle replacement, only appends to existing data).
GCLR	Clear A Global Soup	Clears the specified Global Soup
GDEL	Delete A Global Soup	Deletes the specified Global Soup
GGET	Read From A Global Soup	Returns all frames from the specified Global Soup.
GGCL	Read and Clear A Global Soup	A combination of GCLR and GGET. Reads all frames from the specified Global Soup and then, in one atomic action, clears the soup.

REFERENCES

1 Long, S., Aust, D., Abowd, G., Atkeson, C. Cyberguide: Prototyping Context Aware Mobile Applications. CHI'96 Short paper. April, 1996.

2 Sue Long, Rob Kooper, Gregory D. Abowd, and Christopher G. Atkeson. Rapid Prototyping of Mobile Context-Aware Applications: The Cyberguide Case Study. In the *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom'96)*, November 1996. To appear.

3 Abowd, G., Atkeson, C., Hong, J., Long, S., Kooper, R., Pinkerton, M. Cyberguide: A Mobile Context-Aware Tour Guide. To Appear in *ACM Wireless Networks*, 1997.

4 PalmComputing Pilot web page (http://www.usr.com/palm/index.html)

5 Newton Connection Utilities web page (http://www.newton.apple.com/product_info/SW/ncu.html)

6 Apple MessagePad web page (http://www.newton.apple.com)

7 Apple Computer. *Cyberdog Programmer's Kit*. Addison-Wesley, 1996.

8 Wood, A., Dey, A., Abowd, G. CyberDesk: Automated Integration of Desktop and Network Services. Technical Note In *Proceedings of CHI' 97* (Atlanta, GA, March 1997), ACM Press

9 Kistler, J. *Disconnected Operation In a Distributed File System*. PhD Thesis, Carnegie Melon University. May 1993.

10 Satyanarayanan, M. Mobile Information Access. In *IEEE Personal Communications* (Feb 1996).

11 Coda web page at CMU (http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html(

12 Demers, A., Pertersen, K., Spreitzer, M., Terry, D., Theimer, M., Welch, B. The Bayou Architecture: Support for Data Sharing Among Mobile Users. *Mobile Computing Workshop* (1994).

13 Bayou web page at Xerox PARC (http://www.cs.cmu.edu/afs/cs/project/coda/Web/coda.html)

14 Watson, T. Wit: An Infrastructure for Wireless Palmtop Computing. Qualifier Presentation, slides (Oct 1994).

15 Watson, T. Wit II - Overview (from web page, http://snapple.cs.washington.edu/wit/witII/).

16 Schilit, W. *A System Architecture For Context-Aware Mobile Computing*. PhD Thesis, Columbia University, 1995.

17 Want, R., Schilit, B., Adams, N., Gold, R., Petersen, K., Goldberg, D., Ellis, J., Weiser, M. *The ParcTab Ubiquitous Computing Experiment.*

18 Schilit, B., Adams, N., Gold, R., Tso, M., Want, R. The ParcTab Mobile Computing System. In *Proceedings Fourth Workshop on Workstation Operating Systems* (Oc 1993).

19 Cova, L. Resource Management In Federated Computing Environments. PhD Thesis, Princeton University, October 1990.

20 Tait, C. *A File System For Mobile Computing*. PhD Thesis. Columbia University, 1993.

21 Guy, R. Ficus: A Very Large Scale Reliable Distributed File System. PhD Thesis, University of California, Los Angeles, June 1991.

22 HP Omni-go web page (http://www.hp.com:80/handheld/communicators/communicators.html)

23 Psion web page (http://www.psion.com/)

24 Revelar Connection Utilities web page (http://www.revelar.com/rcu.html)

25 OpenDoc web page at Apple Computer (http://www.opendoc.apple.com)

26 The OpenDoc Revolution web page (http://opendoc.macintosh.net)

27 Apple Computer. *The OpenDoc Programmer's Guide*, Addison Wesley, 1995.

28 CyberTalk web page (http://www.cc.gatech.edu/classes/cs3302_97_spring/projects/team7/notebook.html)

29 Apple Computer. *Newton Desktop Integration Libraries*. 1997.

30 Metrowerks, Inc. *The PowerPlant Programmer's Guide*.

31 Netwave's web page (http://www.netwave.com/)

32 Dayna's web page (http://www.dayna.com/)

33 Newton Internet Enabler web page (http://www.newton.apple.com/product_info/SW/nie/nie.html)

34 Apple's HotSauce home page (http://hotsauce.apple.com/)

35 Want, R., Hopper, A. *Personal Interactive Computing Objects*.

36 Bellotti, V., Sara Bly Consulting. *Walking Away From The Desktop Computer: Distributed Collaboration and Mobility in a Product Design Team*. 1996

37 Spreitzer, M., Theimer, M. Architectural Considerations for Scalable, Secure, Mobile Computing With Location Information. In *14th International Conference on Distributed Computing Systems*. June 1994.

38 Steiner, J.G., Newman, C, Schiller, J.I. Kerberos, an authentication service for open network systems. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*. Dec 1993.