48

Integration of Handheld Computers into Distributed Database Systems

by

Alexandre J. Lee

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

Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology
May 15, 1998
[June 1998]

Author ____
Department of Electrical Engineering and Computer Science
May 15, 1998

Certified by ____
Frans Kaashoek, Thesis Supervisor
MIT EECS Associate Professor

Certified by ____
Quinton Zondervan, Thesis Supervisor
Software Architect, Lotus Development Corporation

Accepted by ____
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Integration of Handheld Computers into Distributed Database Systems

by
Alexandre Lee

Submitted to the
Department of Electrical Engineering and Computer Science

Abstract

This thesis investigates the integration of handheld computers in a distributed database system by examining the issues in the context of a Lotus Notes network. For the purpose of this thesis, a handheld computer consists of any electronic device small enough to fit in the palm of a hand and with minimal computing power, data storage capability, and ability to connect to other computers. This thesis examines the issues in designing a system which allows existing handheld devices to remotely access and edit data stored in Lotus Notes databases, without introducing conflicts in the existing database replication mechanism and without compromising Notes' security model. One of the motivations for this research comes from the desire to establish a framework which will enable the various existing handheld devices with different architectures to communicate with central distributed database systems such as Lotus Notes. In order to evaluate the framework proposed by this thesis, a prototype was built which allows Lotus Notes users to access and manipulate their electronic mail with 3Com's PalmPilot. Although only parts of the proposed framework were implemented in the prototype, feedback from users support the importance of the synchronization issues addressed by this thesis.

Thesis Supervisor: Frans Kaashoek
Title: Associate Professor, MIT Department of Electrical Engineering and Computer Science

Thesis Supervisor: Quinton Zondervan
Title: Software Architect, Lotus Development Corporation

Acknowledgments

This thesis has been the result of not only my work, but also the help of a number of people who have supported me in various ways throughout the process.

I want to thank both of my thesis supervisors, Professor Frans Kaashoek and Quinton Zondervan. I thank Professor Kaashoek for agreeing to supervise this thesis despite the late notice, and for helping me in making sure that this thesis had all the required elements. I thank Quinton Zondervan for the countless hours he spent while helping me in the design, implementation and thesis revision process. His patience, guidance and support have contributed greatly to this thesis.

Most of the work in this thesis was done with the Workgroup Technologies and the Organizer Trident teams at Lotus. I thank everyone in both teams for the truly enjoyable work experience I had, even during the long hours before project deadlines. My special thanks goes to Candy Sidner for taking care of all the VI-A administrative details, to Daniel Gruen for getting me involved with the project in the beginning, to Steve Wells for being a major driving force behind the commercialization of this thesis' prototype, and to Rodolfo Fernandez for carrying on the work with EasySync.

I thank all my dear friends for helping me keep my sanity while working on this thesis. Finally, I cannot find words capable of expressing my gratitude to God, my parents and siblings, whose continuous love, support and understanding have given me the courage and strength to endure even the hardest of times.

Contents

List of Tables and Figures

1. Introduction

Since the introduction of the very first computers, such as the ENIAC which occupied the space of an entire room in the 1940's[1], technology has continuously brought us smaller and faster computers over the years. From the ENIAC, to the mainframes of the 1960's, to the microcomputers of the 1970's, to the personal computers of the 1980's, to the notebook computers of the 1990's, we have reached a stage where popular devices with respectable computing power can now fit in the palm of our hands. There are many interpretations for the definition for such handheld devices, and we will choose to define them as follows: a handheld device is a machine with enough computing power to run different applications and process information using a form factor which must be small enough to fit in users' hands and versatile enough to allow users to view and enter data. Currently, many devices fall under this category, including PDAs (Personal Digital Assistants), HPCs (Handheld Personal Computers), two-way pagers, and smart cellular phones.

1.1. The rising popularity of handheld devices

Since the first handheld devices were introduced in the 1980's, such as Apple's Newton, the demand for them remained relatively stagnant and small until the past few years. The market for handheld devices is expected to grow tremendously over the next several years. A recent study by International Data Corporation estimates that the number of handheld devices sold will grow by over sixty percent in the next year, from 5.1 million units in 1997 to 8.2 million in 1998[2] (Figure 1.1). Such growth comes partly from significant improvement in the technology, which has brought smaller electronic devices with increasing amounts of memory, faster processing power, better connectivity, and better means of entering and viewing data. These improvements have greatly expanded the functionality of handheld devices, enabling users to accomplish from simple tasks such as calendar management to more involved tasks such as manipulating corporate inventory databases.

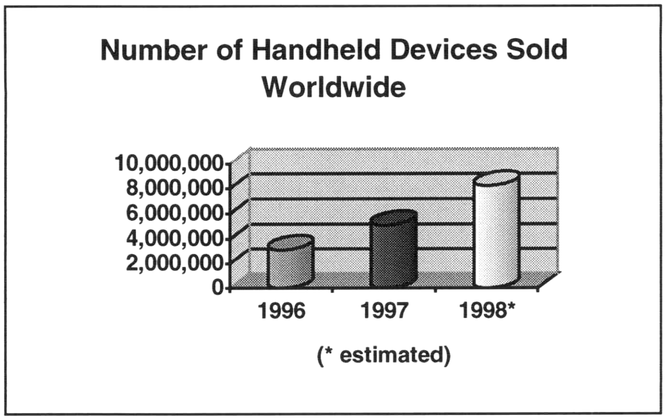


Figure 1.1 Handheld Market

Furthermore, the handheld market has been shifting away from the traditional vertical application paradigm, where each device type was designed for a specific function, be it medical patient tracking or simple personal organization. Recent devices have been built on platforms which are open to the development of any type of applications. Apple's Newton pioneered this approach in the 1980's, and Microsoft's Windows CE and 3Com's PalmPilot Palm OS have since followed. This trend towards open systems has fueled the development of a wide variety of handheld applications, and with the recent addition of network support in these platforms, handheld devices have become an effective tool for sharing and manipulating data which may reside in the network or in other handheld devices.

These improvements in handheld technology have generated an increasing demand for access to corporate information from these devices. Many companies are beginning to exploit such portable connectivity by allowing their mobile workers to directly manipulate data in central databases while they are out on the road. For example, Noblestar Systems Corporation, in Falls Church, Va., has custom developed solutions for companies such as Wal-Mart Stores Inc. and Levi Strauss & Co. which will allow the employees at these companies to dial in to corporate servers and access information using their 3Com PalmPilots[3]. Furthermore, a growing number of Lotus Notes users have indicated their need to integrate their handheld devices with the existing network of information residing in Lotus Notes servers, including PIM, email, inventory and project management data.

1.2. Problem Statement and Thesis Solution

The improvements in handheld technology and the increase in demand have created a new challenge of enabling heterogeneous handheld devices to access and manipulate information residing in central servers. Currently, the technology to be constantly connected to the network from handheld devices, such as 3Com's PalmPilot, still remains limited and costly. Wireless technologies such as CDPD modems and radio packet networks provide constant connection, but not without reliability problems and high cost. In the meantime, while wireless technologies mature, one potentially low cost solution available today for manipulating central server data from handheld devices is to download portions of the actual data set directly into the handheld. The latter then becomes a small portable cache of data that mobile workers can manipulate from anywhere while temporarily disconnected from the network.

The model of allowing handheld devices to obtain and manipulate a local cache of data has the advantages of increasing data availability and reliability for the user. However, such advantages come with the complexities of synchronizing data between a handheld device and a central server, both of which will most likely have completely different operating systems and data storage models. Some of the critical synchronization issues which must be addressed include:

- Selecting the appropriate data set to be transferred to the handheld from the server
- Converting the data between the handheld and server data formats
- Maintaining an updated mapping between the records in the handheld and their corresponding documents in the server
- Preventing data loss from simultaneous updates to the same document on the handheld and server
- Providing secure access to the server from handheld device

The goal of this project was to develop a framework which addresses the synchronization issues mentioned above while allowing any handheld application to securely access and manipulate data in a distributed database system, despite handheld memory limitations, intermittent connectivity and platform differences. Furthermore, such a framework should work without the need to rewrite the applications residing in the handheld device. Avoiding the redesigning of existing handheld applications becomes important as an increasing number of heterogeneous handheld devices enter the market. Part of our goal is to provide a framework that supports the synchronization of a wide range of handheld devices with a distributed database system. It is more efficient to utilize a general synchronization framework that easily extends to support existing applications in many new handheld devices than it is to rewrite every application in these different devices.

While the framework presented in this thesis has not been entirely implemented, parts of it were used to build a prototype which allows Lotus Notes users to manipulate their electronic mail from a PalmPilot. This prototype also became the foundation for a commercial version which is currently in the market.

The rest of this thesis is structured as follows. Chapter 2 presents some previous related work. Chapter 3 describes the development environment used for prototyping and design purposes in this project. Chapter 4 describes the requirements for a synchronization framework between a handheld and a distributed database system in the context of synchronizing a 3Com PalmPilot with Lotus Notes. Chapter 5 addresses some of the security issues of accessing a secure system like Lotus Notes from a handheld device. Note that this thesis project only aims to explain the need for improving the security of handheld access to secure databases. While a simple and more secure authentication mechanism was implemented for the prototype, the rigorous analysis of the validity of better authentication and encryption mechanisms goes beyond the scope of this project. Chapter 6 shows the application of the concepts explained in prior chapters by describing the implementation of a system that allows Lotus Notes users to access and manipulate their email from a PalmPilot. Chapter 7 presents some of the results and lessons learned from the prototypes, and possible future directions. Chapter 8 concludes with an assessment of the project.

2. Related Work

The problem of accessing and manipulating data in a distributed system from a mobile environment has been extensively explored in the research community. The Coda[4] project was one of the earlier works which addressed the issue of manipulating distributed data from disconnected clients. One of the design requirements of the Coda project consisted of providing the highest possible availability of data to disconnected clients. This requirement led them to choose an optimistic concurrency control approach, which provides higher data availability by allowing reads and writes everywhere while dealing with the potential conflicts by detecting and resolving them after their occurrence. Furthermore, Coda also improves data availability by pre-caching data while the mobile client is connected to the network. The challenge consists of caching a set of data that the user will actually find useful while disconnected, and Coda accomplishes this via implicit information such as recent reference history, and explicit information such as files containing a list of useful objects that the user identified.

Like Coda, this thesis' approach to handheld data synchronization adopts an optimistic concurrency control for disconnected data manipulation, while relying on user assistance for cache management during disconnected operation. This approach works well since the user will probably always be the best judge for what objects are needed while disconnected. However, Coda is designed to allow disconnected operation between mobile clients and permanent servers that share the same type of objects, mainly files. Such files are stored and manipulated in the same manner in both client and server, and therefore there is no need for data conversion. The nature of the problem addressed by this thesis is different, since the stated goal consists of enabling data sharing between mobile clients and distributed database systems with completely heterogeneous data storage models. Therefore, the proposed synchronization framework presented in this thesis must solve additional problems such as data conversion and data correspondence between heterogeneous systems.

The Ficus[5] project, conducted at UCLA, is also very similar to Coda, since both aim to provide data availability to disconnected clients in a replicated file system. Both systems adopt an optimistic concurrency control to provide the highest availability possible, and Ficus deals with potential conflicts through conflict resolver programs, which are triggered according to the type of the conflicting file. These resolver programs give the user total control over how conflicts are resolved. Like Coda, Ficus does not address data conversion issues, since both assume that the clients and servers are all using a homogeneous file system.

The Bayou[6] project conducted at Xerox Palo Alto Research Center, is closely related to the work presented in this thesis since it also focuses on providing data availability for mobile clients with expensive connection time and frequent disconnection time. Unlike previous projects like Coda and Ficus which provided transparent replicated data support for disconnected clients, Bayou takes a mobile-aware approach, where applications running on the mobile clients take an active role in detecting and resolving potential conflicts. In order to support application-specific conflict detection and resolution, each Bayou application includes a custom dependency check and merge procedure for every write operation, and such write operations are eventually propagated to every server via pair-wise replication. Another related work, the Rover Toolkit[7] done at MIT, also aims to provide a more general mobile-aware application programming interface, which addresses the issues of mobile communication and concurrency through relocatable dynamic objects and queued remote procedure call logs.

Much like Bayou, the proposed framework in this thesis also aims to provide application-specific conflict detection and resolution, since the wide range of different applications warrant different ways to handle conflicts. However, Bayou's mobile-aware approach requires the client application to be modified to support a new form of write operation that includes dependency checks and merge procedures. Also, the server replication mechanism must be modified to support the anti-entropy sessions used to propagate and commit write operations. This thesis' approach for designing a mobile synchronization framework differs from Bayou's because it is particularly concerned with allowing the applications in the handheld to remain unmodified. Ideally, the framework should allow any existing handheld application to integrate into an existing distributed database system.

Unlike these previous systems, the proposed framework does not attempt to solve the general problem of mobile computing. Instead, this thesis project focuses mainly on mobile data synchronization with the following specific constraints:

- The mobile clients involved are small handheld devices with limited network connectivity, memory and processing power
- The mobile clients only connect to the network to exchange data, instead of working directly with the data residing in servers
- The mobile clients have a different and typically more limited storage model than the network servers
- The native applications in the clients should not have to be modified

3. Development Environment

This chapter describes the development platforms and tools chosen for this thesis project. In order to better understand the practical issues of accomplishing such a task, the system was implemented for a particular device and database system. However, care has been taken to avoid design decisions which are based on features that are unique to the chosen systems, so that the lessons drawn from this research could be applicable to any handheld device and database systems with certain general features.

For the handheld device, these features include:

- relatively low cost ($1000 or less), allowing for deployment to a large user base
- presence of an open operating system with available tools to develop new applications for the handheld device
- connectivity to desktop computers via serial connection or LAN connection
- support for storing and organizing databases and corresponding records
- presence of a mechanism to uniquely identify each of the records in a database with some form of an unique id number

On the database system side, some of the desired features include:

- ability to replicate databases between servers
- availability of tools to access the database content directly from other applications
- reliable security options for user authentication.

3.1. Handheld Device: 3Com's PalmPilot

The handheld of choice for this project was 3Com's PalmPilot, first released in 1996. Note that the PalmPilot was originally developed by Palm Computing, which in 1995 was acquired by U.S. Robotics, which in turn was acquired by 3Com in 1997. The PalmPilot has a fairly extensible architecture and it connects well to desktop computers. The PalmPilot measures 4.7 x 3.2 x .7 inches with a 160 x 160 pixel backlit pressure sensitive

LCD screen. At the heart of the PalmPilot is a Motorola 68328 "DragonBall" CPU, running at 16 Mhz, with 2Mb, 1Mb, or 512Kb of RAM, and 512Kb of ROM, depending on the model. The model used in this project, the PalmPilot Professional, ran with 1 Mb of RAM. The PalmPilot's operating system consists of a proprietary system called PalmOS, whose functions include handling pen input and preserving information as the user switches between applications. The file system used by the PalmOS organizes both data and applications in a database structure consisting of a collection of records. The PalmPilot comes with some basic applications, such as Calendar, ToDo, Memo and Address Book applications, and the Professional model also comes with a basic email reader. New PalmPilot applications can be developed in either Microsoft Windows, Macintosh OS or UNIX systems using tools and libraries provided by 3Com, GNU, and Metrowerks. Such new applications are compiled into .prc files, which are then downloaded onto the PalmPilot.

The PalmPilot has a standard RS-232 serial connector, used either to connect to a desktop computer directly, or to attach a modem which can then be used to connect to a desktop computer. This serial connection can be used to download new applications into the PalmPilot from the desktop, or to exchange and synchronize data between PalmPilot applications and their equivalent desktop applications. The PalmPilot is capable of connecting to Microsoft Windows, Macintosh OS and Unix based desktop computers. For this thesis, the desktop computer used consisted of a Pentium based computer running Microsoft Windows NT.

The process of synchronizing data is handled via conduits. For synchronization with Microsoft Windows systems, such conduits consist of dynamic linked libraries on the desktop computer that take care of converting and organizing the data from PalmPilot to desktop and vice-versa. The synchronization process is controlled by the Hotsync Manager, a program which sits in the desktop computer and which runs whenever the PalmPilot initiates a synchronization session. When the PalmPilot “hotsyncs” with a computer, the Hotsync Manager executes the registered conduits. Generally, there is a conduit for each of the PalmPilot applications whose data need to be synchronized with its desktop equivalent. Figure 3.1 illustrates the PalmPilot synchronization mechanism. 3Com has also provided the APIs necessary to develop new conduits. For this project, the conduit and PalmPilot application development were done under Microsoft Windows NT 4.0 with Microsoft Visual C++ 4.0, Metrowerks' CodeWarrior, and 3Com's PalmPilot Conduit SDK.

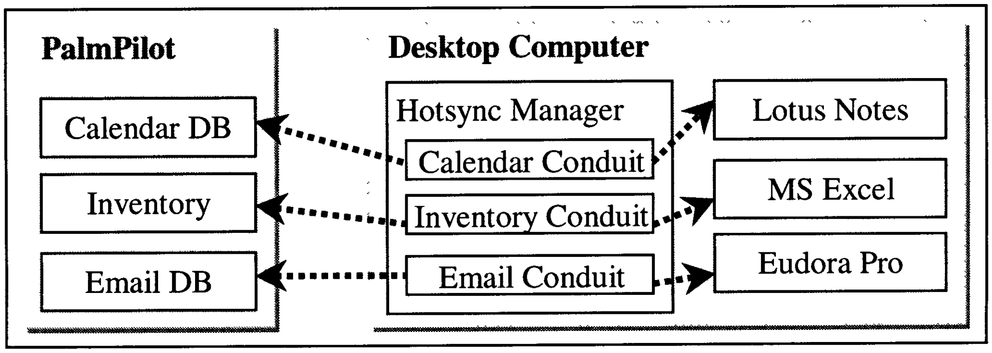


Figure 3.1 Overview of the PalmPilot synchronization mechanism.

In its current implementation, the PalmPilot's data synchronization model was primarily designed to connect the PalmPilot to the user's desktop machine. However, the model can also be extended to support synchronization with another desktop machine or a central server, which may hold data for multiple PalmPilot users. Through Network Hotsync, 3Com has provided the mechanism necessary to allow the PalmPilot to find a particular computer in a TCP/IP network. By connecting to any computer in the LAN, either directly through a serial connection or remotely with a modem, PalmPilot users can specify the name or the IP address of another computer to synchronize with. Note that each desktop computer can only synchronize with one PalmPilot at a time, and this may become a serious limitation if many PalmPilot users are expected to synchronize with one central server.

Currently, authentication between a PalmPilot and a target computer during remote synchronization relies solely on the verification of the PalmPilot user name stored in the handheld device. This simple authentication mechanism raises serious security issues as PalmPilots are used to access sensitive information residing in Lotus Notes servers.

Therefore, there needs to be a better and secure user authentication mechanism in the PalmPilot. Chapter 5 of this thesis describes one simple alternative to increase the security in the PalmPilot's current authentication mechanism.

There are other handheld devices which share many of the PalmPilot features described above, and these include HP's 620LX Palmtop PC, based on Microsoft's Windows CE 2.0, and Psion's Series 5 Handheld Computer. Even though the PalmPilot does not have some of the features in these handhelds, such as color screens and keyboards, it does have the advantages of being extremely portable as well as popular, providing a larger user base to obtain feedback from. Regardless, the design considerations presented in this thesis can be easily adapted to a wide variety of handheld devices which share the basic features described earlier. Table 3.1 lists some of the popular current handheld devices, which could have been used for this project.

Name	*Est. Price*	*Weight*	*Screen Size*	*Processor*	*RAM/ ROM*	*Oper. System*	*Input/ Output*
3COM Palm III	$399	5.7oz	2.5"x 3.3"	Motorola Dragon Ball 68320A 16Mhz	2/2 Mb	PalmOS 3.0	Touch Screen, Serial Port, IR
3COM PalmPilot Professional	$299	5.7oz	2.5"x3.3"	Motorola Dragon Ball 68320A 16Mhz	1/1 Mb	PalmOS 2.0	Touch Screen, Serial Port, IR
Hewlett-Packard 620LX Palmtop PC	$899	20.8ox	6.2"x2.4" color display	Hitachi SH3 75Mhz	16/10 Mb	Windows CE 2.0	Keyboard, Audio, Serial Port, IR
Psion Series 5	$599	12.5oz	5.3"x2"	ARM 7100 18.432MHz	8/6 Mb	EPOC32	Keyboard, Audio, Serial Port, IR
Casio Cassiopeia A-20	$599	15.2oz	6.2"x2.4"	Hitachi SH3 80 Mhz	8/8 Mb	Windows CE 2.0	Keyboard, Serial Port, IR
Texas Instrument Avigo 10	$299	7oz	2"x3"	Custom Z-80	1/1 Mb	Custom	Touch Screen, Serial Port, IR
Sharp Zaurus ZR-3500X	$399	7oz	3.5"x2.5"	Sharp Proprietary	1/1 Mb	Custom	Keyboard, Serial Port, IR
Phillips Velo 500 Handheld PC	$749	15oz	6.5"x3.5"	Philips PR31700 75Mhz	24/16 Mb	Windows CE 2.0	Keyboard, Serial Port, IR

Table 3.1 List of possible handheld devices for the proposed framework.

3.2. Distributed Database System: Lotus Notes

The distributed database system used for this project was Lotus Notes, the leading groupware system on the market. Lotus Notes is a client-server database system, where each database is a collection of documents containing plain text, graphics, World Wide

Web pages, or other multimedia content such as video and sound files. The documents themselves can be organized under different views, which offer the users the ability to organize their data in different ways according to arbitrary queries. For example, views may sort the documents according to a particular field, according to the documents' creation date, or according to the last time the documents were modified. Aside from views, documents may also be organized by placing them in different folders, which are simple containers of documents. Views and folder are different since whereas a document may exist in multiple views, a document may only exist in one folder.

Lotus Notes also supports a database replication mechanism which allows the creation of multiple replicas of a single database in different machines. Such replicas may reside on different servers or desktop computers. At any time, two replicas can be synchronized to maintain consistency between them. Replication can be initiated by the user or system administrator, and it can also be scheduled to occur automatically at regular intervals. Lotus Notes' replication mechanism is utilized by mobile users to do useful work while disconnected from the server, as well as by database administrators to distribute the load on frequently accessed databases, and to keep back-up copies of databases for fault tolerance[8]. For each document, Notes also maintains modification timestamps used to detect conflicts when replicas of the same document have been modified in between replication sessions. Note that the current Notes replication scheme is limited to Notes installations only. Part of the goal of this thesis , in the context of Notes, is to extend the replication mechanism to handheld devices, such as the PalmPilot, that could not support Notes replication.

Lotus has provided the Notes C and C++ APIs, which allow other programs to access and manipulate Notes databases. All interaction between the PalmPilot conduits and Notes was implemented via the Notes C and C++ APIs.

4. Data Synchronization Framework

As described earlier, the core of the synchronization process between a PalmPilot and Lotus Notes resides in the conduit executed by the Hotsync Manger during synchronization. The conduit handles all the complexity of manipulating, converting and updating data between the handheld device and the Lotus Notes database. Usually, each pair of handheld application and Notes database will have its own conduit, given the different synchronization needs depending on the type of data and application. The PalmPilot conduit performs much like a data conversion program, and other handheld platforms, such as Windows CE, also use such conversion programs to synchronize data between the handheld and desktop computer. The data synchronization framework described in this chapter was designed to serve as the basis for conversion programs like the PalmPilot conduit. A complete implementation of the framework should abstract away all the complexities of synchronizing a handheld device with a distributed database system by including a generic synchronization engine that takes care of tasks like data conversion and conflict resolution.

In general, when synchronizing between a PalmPilot and Lotus Notes, there are four important issues related to the transfer of data between the two platforms which need to be addressed:

- how will the user select the data set which will be synchronized
- how will the conduit convert the rich content of Notes documents to the limited format of a PalmPilot record
- how will the conduit keep track of the correspondence between PalmPilot records and Notes documents
- how will the conduit resolve conflicts when the same document has been modified in both Notes and PalmPilot.

The following sections describes how the proposed synchronization framework addresses the issues above.

4.1. Selection of synchronization data set

A Lotus Notes database can potentially be quite large, sometimes ranging anywhere from 150K to over 300Mb in size. Given that a PalmPilot Professional's total RAM is 1Mb, the issue of selecting a small subset of data to be synchronized becomes important. The PalmPilot should be viewed as a small temporary local cache of data stored on the desktop computer. Since the desired data on the PalmPilot will vary from database to database, it only makes sense for each conduit to implement its own mechanism for selecting the appropriate data set. Thus, when synchronizing a mail database, the conduit may just work with today's emails, or when synchronizing a discussion database, the conduit may just select those documents that have been recently added to the database.

However, a conduit with a predetermined algorithm for choosing a particular data set may prove to be too limiting for the user. A better solution is to provide a conduit configuration option, set at the desktop, which allows the user to input a formula for selecting the appropriate data set in the database. For example, in the case of Notes, if the formula is written in the Lotus Notes macro format, the conduit will be able to simply query Notes using the following formula to select all the documents older than 21 days:

```
SELECT (@Created > @Adjust(@Now; 0; 0; -21; 0; 0; 0))
```

Aside from accepting a formula, the conduit could also accept a particular view name, in which case the conduit would synchronize all the documents under a particular view in the database. This option is more powerful than the use of macro formulas because the user may specify much more complex algorithms for document selection in a Notes view using LotusScript, a proprietary scripting language used within Lotus Notes. Another possible mechanism for selecting documents consists of making the conduit always synchronize the documents that reside in a particular view or folder named "PalmPilot", or whichever other name that may be appropriate. This way, when user Ben Bitdiddle is working with a particular database from a Notes client, and he wishes to synchronize a particular document with his PalmPilot, all he needs to do is place that document in a folder named “PalmPilot”, and synchronize. This mechanism creates the interesting analogy where placing a document in the “PalmPilot” folder becomes equivalent to placing the document in the actual PalmPilot. Microsoft Windows 95 also has a similar mechanism where users can place files in a “Suitcase” folder, whose contents are transferred to the user’s notebook computer when the latter is connected to the desktop computer.

The possible data selection mechanisms described above assume that the user is physically in front of the desktop computer during synchronization, with access to the Notes client and the conduit configuration menu options. This is not the case when the user decides to synchronize remotely using a modem which connects directly to the desktop computer. For some of the mechanisms described above to work, there also needs to be a way for the user to enter the macro selection formula or the view name from the PalmPilot itself, and propagate that information to the conduit so that it can fetch the appropriate set of data. This can be done by providing the user with an equivalent conduit configuration program in the PalmPilot, which stores the user’s preferences in a special configuration database in the PalmPilot. During synchronization, the conduit first fetches the information from the configuration database in the PalmPilot, and then proceeds with the rest of the data exchange.

The choice of data selection mechanism will depend heavily on the nature of the database being synchronized. In some cases, a combination of the above mechanisms may actually work best, and such was the approach taken for the Lotus Notes Mail conduit described in Chapter 6. For that conduit, the user has the option of selecting messages based on common selection algorithms, such as all messages modified since the last synchronization session, or all messages received since a particular date. For those users with more specific needs, the conduit also offers the option to select messages that reside in a particular view or folder. In many other database applications aside from Notes Mail, there are also certain selection algorithms which are commonly used. For example, for discussion databases, users will often want all the new discussion threads added since the last synchronization session, and for project management databases, users often want all the documents associated with a particular project. Whenever possible, a conduit should always provide users with such database-specific common selection algorithms, while still providing other more flexible algorithms for the power users.

4.2. Data Conversion

As mentioned earlier, a Notes document is a rich repository of data which may contain text, World Wide Web pages, graphics, attachments, document links and other multimedia data types. On the other hand, records in the PalmPilot are limited to simple text and graphics. Therefore, when converting a Notes document to a PalmPilot record, attachments, document links, and other multimedia types must be handled in a special way, while the remaining text and graphics content is converted.

The conversion of graphic images from Notes to PalmPilot actually goes beyond simply translating image pixels. Images in Notes can be of any size in high resolution with up to 24 bits of color. First of all, such Notes image must be converted to a 2-bit image, which is the only format supported by the PalmPilot. Then, the image must be scaled in such way that it remains proportional to the overall layout of the page while still fitting inside the 2.5"x2.5" LCD screen of the PalmPilot. In many cases, this process will reduce the original Notes image to a barely recognizable thumbnail sketch in the PalmPilot. Ideally, the PalmPilot users should be given the option to tap such thumbnail sketches and inform the system to show an enlarged and more detailed image of the sketch in full screen. A very good implementation of such mechanism of handling image conversion can be found in Top Gun WingMan developed at U.C. Berkeley[9], which is a web browser application for the PalmPilot. WingMan fetches web pages via a web proxy, which pre-processes the requested pages by taking care of the document conversion and image processing. Images are converted to 2 bit grayscale thumbnail sketches, and when the PalmPilot user taps on a sketch, WingMan sends a request to the proxy requesting a larger and more detailed version of the picture.

Even though attachments and documents links do not mean much inside the PalmPilot, they contain information which may prove to be useful for the user when reading the document from the PalmPilot. The filename of attachments and the server, database, and document id of document links are examples of such information that could be included in the converted PalmPilot record. In certain cases, it may be just enough to include a message in the converted PalmPilot record indicating that the original Notes document contains attachments or document links.

When dealing with large Notes documents, there will be times when even the text content of the document by itself will exceed the memory limits of the PalmPilot. Sometimes, a few large converted records may be enough to fill the memory. Therefore, when synchronizing documents that may contain a lot of text, such as mail messages and discussion database documents, there should be an option to truncate the original text during conversion. Each conduit should allow the user to enable or disable the option to truncate a document if it contains more than a specified number of bytes. However, care must be taken to prevent a truncated PalmPilot record from overwriting its Notes document equivalent during subsequent synchronization session. Aside from truncation, large Notes documents can also be compressed to fit the memory requirements of the PalmPilot. The compression algorithm could be as simple as vowel removal or some other word abbreviation filtering process. In some cases, it may also be useful to include a mechanism that allows the user to tell the synchronization process to download the complete copy of a particular document into the PalmPilot. This feature would be helpful if the user receives a truncated or compressed document in the PalmPilot, and realizes that he/she needs to have the rest of the information.

Another critical issue regarding data conversion consists of mapping the fields in a Notes document to their appropriate equivalent fields in a PalmPilot record. Each PalmPilot record keeps its information in an unformatted data buffer, which may contain practically anything. Each PalmPilot application organizes this data buffer in a different way. For example, the Memo application in the PalmPilot places the title of each memo in the beginning of the data buffer, followed by a null character, then followed by the text of the memo. The Mail application places date information in the first four bytes of the data buffer, followed by recipient, sender, subject, CC, and BCC information, each separated by a null character. Because of the arbitrary nature of the PalmPilot record's data buffer, each conduit handles the field mapping in different ways.

4.3. Preserving data coherence between PalmPilot and Notes during synchronization with multiple replicas

When synchronizing a Notes database with its PalmPilot equivalent, each PalmPilot record must be mapped to its corresponding Notes document. During synchronization, if a modified document is found in Notes, the conduit must first check if there is a corresponding record in the PalmPilot. If so, the conduit must have some way of accessing that record in order to propagate the document changes to it. Otherwise, the conduit must create a new record corresponding to the modified Notes document. The same scheme applies when a modified PalmPilot record is found. If the record-document mapping information becomes corrupted, duplicate copies of records and documents may arise, and modifications may be propagated to the wrong places during synchronization. This illustrates the importance of keeping an accurate record-document mapping after every synchronization session.

The rest of this section is organized as follows. Section 4.3.1 describes some of the trade offs in terms of choosing the best method for maintaining the mapping information described above. The chosen method will determine the overall data synchronization algorithm, which is also described in detail in Section 4.3.1. Section 4.3.2 expands on the framework from the previous section by detailing the additional requirements needed to address the potential conflicts that arise when the same PalmPilot synchronizes with multiple replicas of the same database.

4.3.1 Synchronization with a Single Database

A common user configuration in Notes consists of a database which resides on a Notes server, along with a replica of the same database residing on the user's notebook computer. The former will be hereafter referred as the server replica, while the notebook copy will be referred to as the local replica. The user, Ben Bitdiddle, regularly makes additions, modifications and deletions to documents in this database. When he has a high speed connection available to the Notes server (i.e. corporate LAN), Ben modifies the server replica directly through his Notes client. When he is disconnected (e.g. on the airplane), he works on his local replica, and replicates the two replicas when he gets back to the office. Now, let's suppose that Ben acquires a PalmPilot, which allows him more flexibility while traveling. He then decides to synchronize his PalmPilot Memo application database with the server replica of his Notes discussion database, allowing him to make changes to the database on the road with just his PalmPilot.

4.3.1.1 Mapping PalmPilot Records to Notes Documents with an Id Mapping Table (IMT)

When Ben synchronizes the PalmPilot with the server replica, each record in the PalmPilot's version of the database will correspond to a document in Notes, excluding new records which were added since the last synchronization session. Synchronization consists of finding the corresponding document and record pairs (or creating the one that doesn't exist yet), and updating the text content of both so that they match. In order to correctly determine the correspondence of a document and record pair, it is necessary to maintain the mapping between PalmPilot record id's and Notes document id's. The PalmPilot uniquely identifies each database record with a record ID consisting of 4 bytes, while Notes does the same for each document with a Universal Note Identification (UNID) consisting of 16 bytes. The UNID also identifies a document uniquely across any replica of the database.

One option for maintaining the document-record mapping consists of storing the corresponding UNID in each PalmPilot record. However, modifying the information stored in each PalmPilot record violates one of the original goals of this project, stated earlier, which was to allow handheld synchronization with Notes without modifying the existing handheld applications. Storing the corresponding PalmPilot record id in the Notes document also poses a problem, since if a Notes documents gets deleted, the record id information will also be deleted, making it impossible for the conduit to delete the appropriate record in the PalmPilot.

The alternative option consists of keeping a mapping table which indicates how each of the records in the PalmPilot corresponds to a particular document in Notes. This table will be referred to as the Id Mapping Table (IMT). Given that the UNID consists of 16 bytes of data, each entry in the mapping table would consist of at least 20 bytes (16 bytes for the UNID and 4 bytes for the PalmPilot record id).

4.3.1.2 Storing the IMT

Given the need for an IMT, the question becomes where to store such a table. One option is to store the IMT in the PalmPilot itself. However, since the size of the mapping table may grow substantially, this could be a significant strain on the already limited resources of the device. Consider the case if Ben decides to synchronize both his company's Notes Address Book and his personal Notes Calendar databases with the corresponding applications in the PalmPilot. Suppose that Ben, being a busy businessman, has about 20 scheduled events per week, giving him roughly 1000 scheduled events per year, and therefore about 1000 entries in his PalmPilot DateBook after one year. Furthermore, suppose that his company has about 5000 people, giving him a PalmPilot Address Book with about 5000 entries. On average, each calendar and address book entry takes up about 50 and 100 bytes respectively. Therefore, given that each id mapping needs at least 20 bytes, the overhead in storing such mappings would be anywhere from 20 to 40 percent in additional storage space. With the above example, the id mapping for all those entries in Ben's PalmPilot would occupy at least 120K bytes, over 10% of the total storage space in a current PalmPilot Profession model. The high relative cost of storing the mapping table in the PalmPilot becomes less of an issue when dealing with databases with larger documents, such as an email database, where a typical message size (text only) can easily exceed 1K bytes. An extra overhead of 20 bytes per message would be less significant in that case. However, databases with many small documents, such as the ones described above, are prevalent enough to consider better alternatives for storing the mapping table.

To conserve space on the device, we propose to store the IMT as a special document in the Notes database being synchronized. During synchronization, the appropriate conduit reads the IMT into memory from this document, and once all the updates have been propagated to both handheld and desktop, the conduit will store the updated IMT back into Notes. Note that we could have conserved space in the PalmPilot by storing the IMT as a file residing anywhere in the desktop computer. For single replica synchronization, the choice of storing the IMT in Notes or in the file system does not make a difference. However, in order to support synchronization with multiple replicas, the most updated IMT must be available in the machine where each replica resides. In this case, storing the IMT as a document in the Notes database being synchronized allows us to take advantage of the native Notes replication mechanism, which can propagate new updates to the IMT in a particular replica to all the other existing replicas. Section 3.3.2 describes the details of the algorithm for synchronizing against multiple replicas of the same database.

4.3.1.3 Algorithm for Single Database Synchronization

With the use of an IMT to keep the id mapping information, the general algorithm for single database synchronization would consist of the following steps:

1. Find the IMT in Notes, and load it into memory
2. For all new Notes documents since last synchronization session, create new corresponding records in PalmPilot, and update IMT accordingly
3. Propagate all document modifications and deletions from Notes to PalmPilot, and update IMT accordingly
4. For all new PalmPilot records since last synchronization session, create new corresponding documents in Notes, and update IMT accordingly

5. Propagate all record modifications and deletions from PalmPilot to Notes, and update IMT accordingly
6. Update the IMT document in Notes

In steps 3 and 5, the propagation of modifications between the two platforms may encounter the case when the same document and record has been modified in both Notes and PalmPilot. Another possible conflicting case could occur when the document has been modified in Notes but it has also been deleted in the PalmPilot, or vice-versa. Section 4.4 will discuss at length the conflict resolution mechanisms needed to address these cases of conflicting modifications.

Suppose that Ben synchronizes his PalmPilot with the server replica for the first time, and that documents x and y have been recently created and selected for transfer to the Pilot database, which has no records prior to synchronization (Figure 4.1a). Figure 4.2 shows the pseudocode for the algorithm being used during synchronization. The first step for the conduit will be to look for the IMT in the server replica. Since this is the first time Ben synchronizes his PalmPilot with the server replica, there will be no IMT, and the conduit will therefore create a new IMT. Then, the conduit will look through the server replica, looking for any new documents, updates or deletions that need to be propagated to the PalmPilot (Figure 4.2 lines 10-120). Here the conduit finds documents x and y, and it looks into the IMT to check if these documents exist in the PalmPilot. Since the IMT is empty, the conduit concludes that these documents do not exist in the PalmPilot, it creates the corresponding records X and Y in the PalmPilot database, and it adds the appropriate id mappings in the IMT for the newly created records. Next, the conduit looks through the PalmPilot database, looking for any new records, modifications or deletions which need to be propagated to the server replica (Figure 4.2 lines 130-240). Since the PalmPilot database started out empty, the conduit does nothing during this step. Finally, the conduit writes the IMT from memory into a document in the server replica (Figure 4.1b).

Suppose that after the first synchronization, Ben takes his PalmPilot home, and in the train, he modifies record X and adds a new record Z in the PalmPilot. The next day, at work, he deletes document y from the server replica, and then he decides to synchronize with the server replica (Figure 4.1c). This time, the conduit will find and load the IMT that was created during the last synchronization session. The conduit then examines the modifications done in the Notes database. It finds that document y was deleted, it looks up in the IMT to find that document y corresponds to record Y, it deletes record Y, and finally it deletes the appropriate entry from the IMT. Next, as the conduit searches the PalmPilot for new, modified and deleted records, it finds the new record Z, creates the corresponding Notes document z, and adds the appropriate entries in the IMT. Then, the conduit detects that record X was modified, it looks up in the IMT for the UNID of the corresponding document x, and it uses the UNID to open the document and apply the appropriate changes that were made in the PalmPilot (Figure 4.1d).

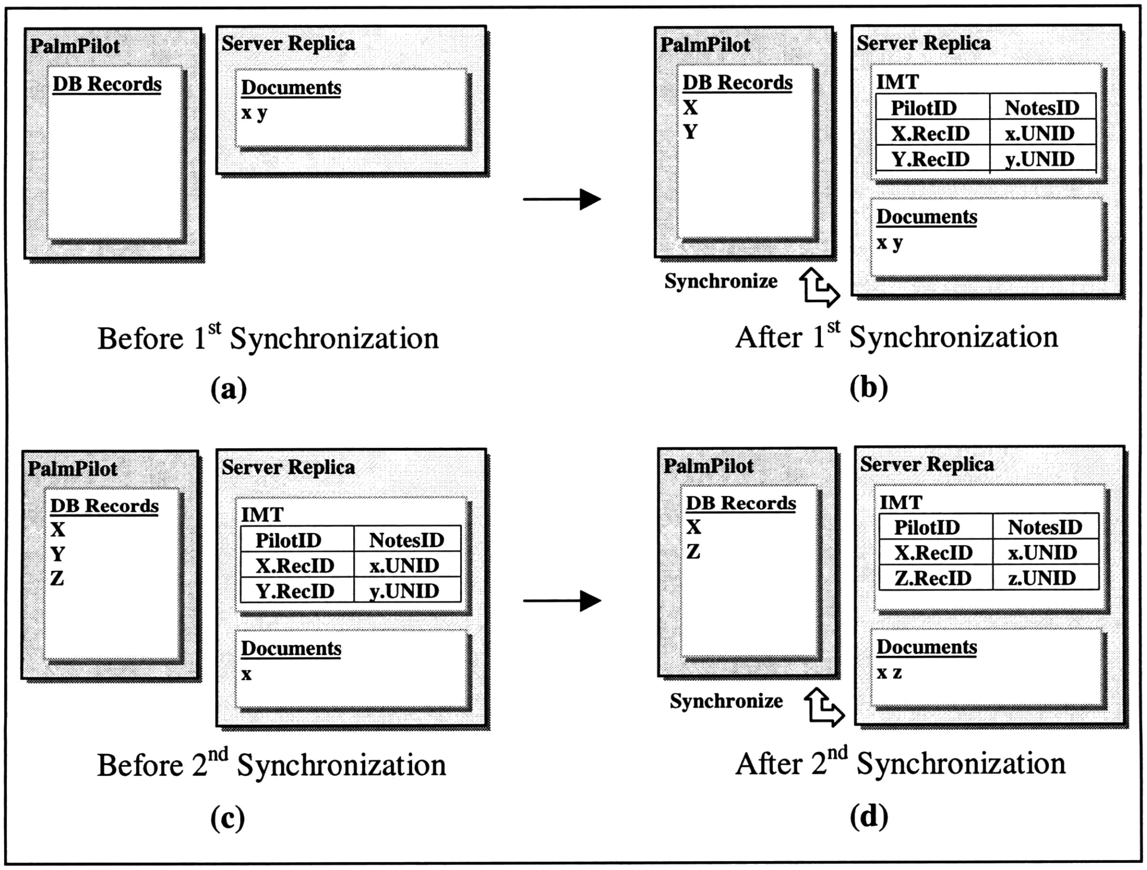


Figure 4.1 Sample scenario illustrating single replica synchronization.

Note that, for the actual implementation of this algorithm in Figure 4.2, the calls to UpdatePilotRecord and UpdateNotesDocument are responsible for determining and propagating modifications and deletions appropriately, while resolving any conflicting modifications as described in Section 4.4.

```
Synchronize()
  {
    WHILE (notesDoc = next Notes document to be synched) DO
        PilotId = LookupIMT(notesDoc.UNID)
        IF (PilotId) THEN
            pilotDoc = GetPilotDocument(PilotId)
            IF (pilotDoc is valid) THEN
                UpdatePilotRecord (notesDoc, pilotDoc)
            END
        ELSE
            pilotDoc = CreateNewPilotRecord(notesDoc)
            AddIMTEntry (pilotDoc.Id, notesDoc.UNID)
        END
    END
    WHILE (pilotDoc = next PalmPilot record to be synched) DO
        NotesUNID = LookupIMT(pilotDoc.Id)
        IF (NotesUNID)
            notesDoc = GetNotesDocument(NotesUNID)
            IF (notesDoc) THEN
                UpdateNotesDocument (pilotDoc, notesDoc)
            END
        ELSE
            notesDoc = CreateNewDocument(pilotDoc)
            AddIMTEntry (notesDoc.UNID, pilotDoc.ID)
        END
    END
  }
```

Figure 4.2 Pseudocode for single replica synchronization.

4.3.2 Synchronization with Multiple Database Replicas

Continuing with the previous scenario, suppose that after synchronizing his PalmPilot with the server replica, Ben brings his notebook local replica up to date by replicating with the server replica. At this point, both replicas will have the same documents as well as the most updated IMT available, as illustrated in Figure 4.3.

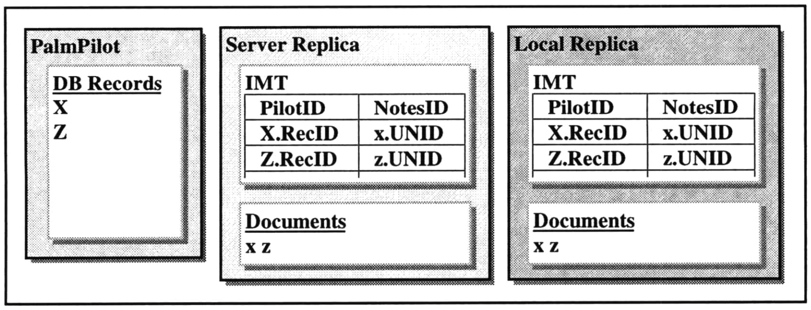


Figure 4.3 Initial state for multiple synchronization scenario.

4.3.2.1 Synchronizing with multiple identical replicas with updated IMTs

Ben then leaves work, and on the train ride home, he makes some changes to the PalmPilot version of the database by modifying record X. Once at home, Ben decides that he wants to have those changes in the local replica on his notebook so that he can take advantage of the richer editing environment available on his notebook computer. He therefore synchronizes the PalmPilot with his local replica. In this case, the simple algorithm presented in the previous section will work, even though Ben is synchronizing his Pilot with a different replica. The crucial part is the fact that the local replica has the most recent IMT, which allows the conduit to figure out which document the modified record X corresponds to so that it can propagate the changes appropriately.

However, if Ben tries to synchronize with a replica which does not have the most recent IMT, duplicate copies of the same document may be created. The following scenario and Figure 4.4 illustrates this problem. Suppose that Ben creates a new documents k in the local replica in his notebook computer (Figure 4.4a). Before going back to work, Ben synchronizes his PalmPilot with the local replica, and in the process the conduit will propagate the new additions in the local replica by creating record K in the PalmPilot (Figure 4.4b). On his way to work, Ben makes some modifications to record K (Figure 4.4c). Once at work, Ben wants those changes in his server replica, so he synchronizes his PalmPilot with the server replica, which at this point contains an outdated IMT. During the synchronization process, the conduit will find the modified record K, and it will try to find a corresponding Notes document in the IMT. Since the IMT in the server replica is outdated, it does not have any entries with record K's id, and it will therefore assume that it needs to create a new corresponding document k* for record K (4.4d). After synchronizing, Ben decides to replicate his server and local replicas, causing problems. First, documents k and k* will be created in both replicas, even though they are identical. Furthermore, since both IMTs were modified since the last replication, they will cause a replication conflict, and there needs to be some mechanism to combine the changes made to both IMTs into one single IMT.

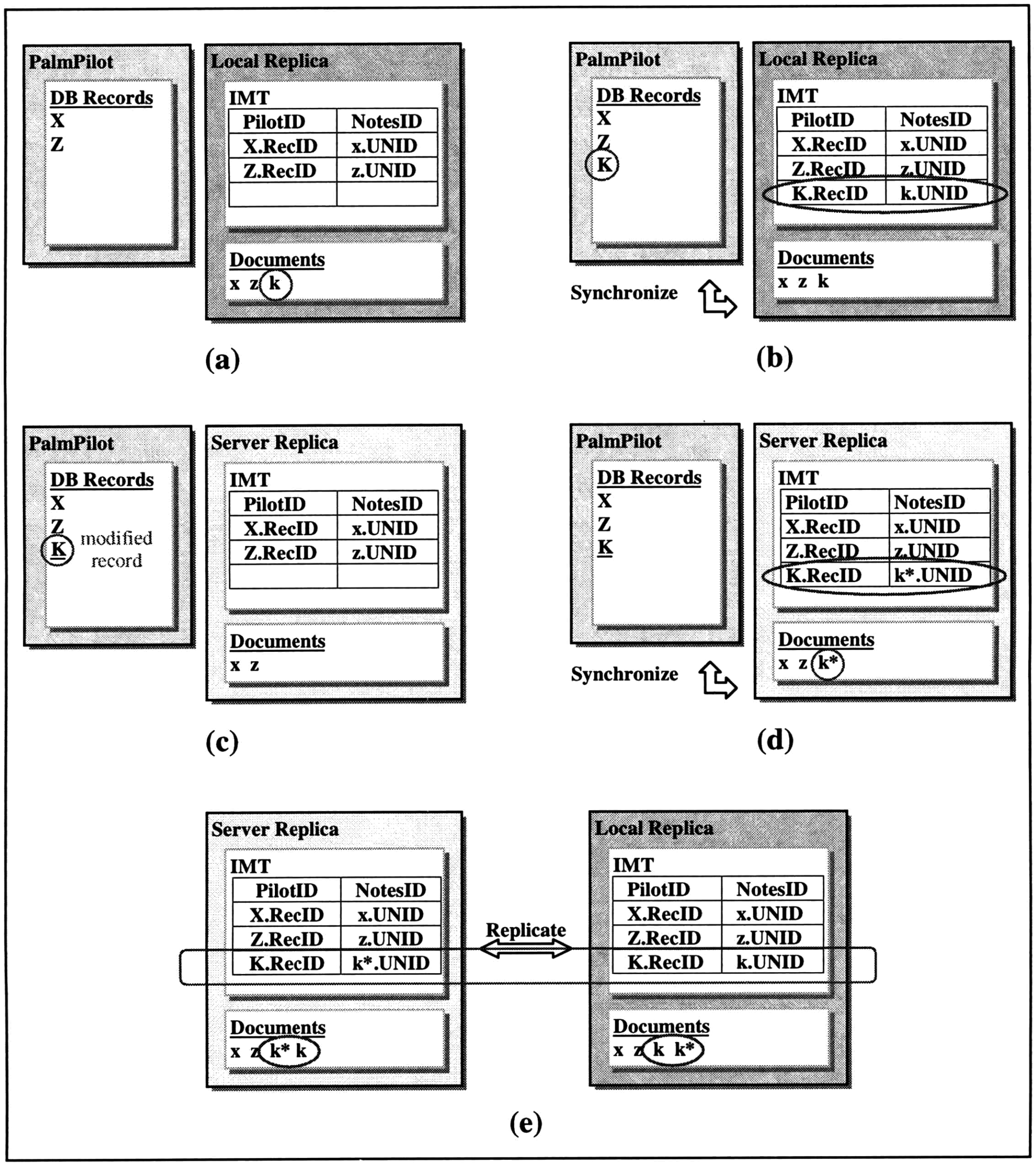


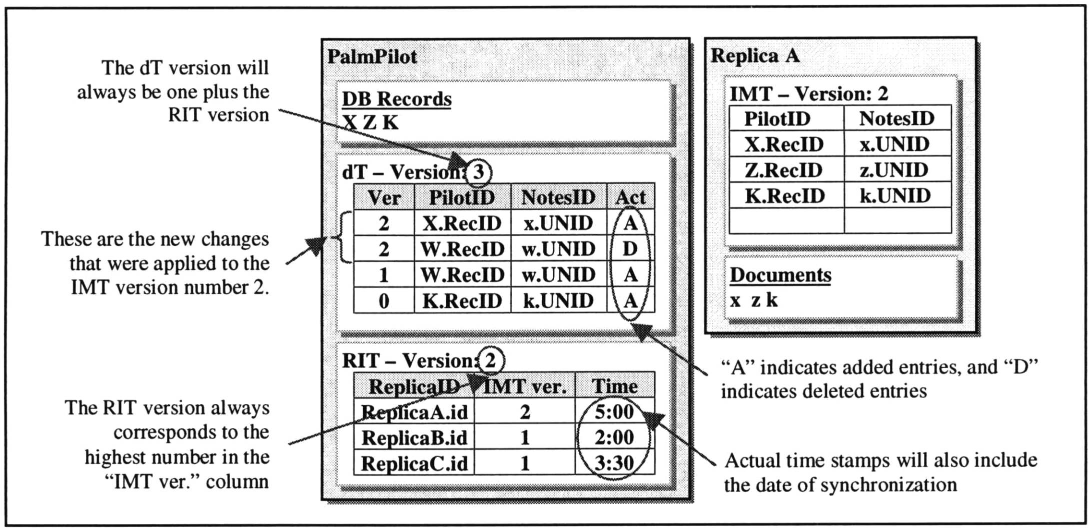
Figure 4.4 Scenario illustrating the potential problems of multiple replica synchronization. Note the creation of duplicates at stage (e).

The scenario described above indicates that synchronization with multiple database replicas with the algorithm presented in Figure 4.2 can be supported only when the user always synchronizes with an updated replica which contains the most recent changes to the database as well as the most recent modified IMT. Therefore, before synchronization, the user always has to replicate in order to prevent the duplicate problems described above.

4.3.2.2 Using the Delta Table (dT) to relax replication requirement

There will be times when users will not be able to replicate before performing successive synchronization with different replicas. If Ben goes home with his notebook without first replicating the server and local replicas, and he finds out that he cannot connect to the company network from home because his network service is down, then he may not be able to synchronize his PalmPilot with the local replica without causing the kind of problems described above. In cases like this, we would still like to allow the user to synchronize without the potential conflicts created by using an outdated IMT.

The underlying problem with synchronization with multiple replicas is that the user may sometimes synchronize using an outdated IMT. To address this problem, the synchronization algorithm described above is extended with the addition of a Delta Table (dT), which is stored in the PalmPilot version of the database (Figure 4.5). Each entry in this dT consists of a PalmPilot record id to Notes document id mapping, along with an action flag reflecting the type of update to the IMT that this entry corresponds to (addition or deletion). During a synchronization session, whenever an id mapping needs to be added or deleted, an entry is first added to the dT with the appropriate mapping and action flag. The dT essentially keeps a log of all the mapping changes that have been made during the synchronization with multiple replicas. When synchronizing with a replica which does not contain the most recent IMT, the combination of the log information from the dT and the available outdated IMT is enough to construct the equivalent of an updated IMT.



Ver	PilotID	NotesID	Act
2	X.RecID	x.UNID	A
2	W.RecID	w.UNID	D
1	W.RecID	w.UNID	A
0	K.RecID	k.UNID	A

ReplicaID	IMT ver.	Time
ReplicaA.id	2	5:00
ReplicaB.id	1	2:00
ReplicaC.id	1	3:30

PilotID	NotesID
X.RecID	x.UNID
Z.RecID	z.UNID
K.RecID	k.UNID

The dT version will always be one plus the RIT version

These are the new changes that were applied to the IMT version number 2.

The RIT version always corresponds to the highest number in the "IMT ver." column

"A" indicates added entries, and "D" indicates deleted entries

Actual time stamps will also include the date of synchronization

Figure 4.5 Components of the complete framework supporting multiple replica synchronization

4.3.2.3 Using the Replica ID Table (RIT) together with the dT to prune the dT

While the dT eliminates the need to always synchronize with an updated IMT, it potentially violates the size constraints of the device, as the dT may grow arbitrarily large over time. To avoid this problem, there needs to be a mechanism to permanently commit the recorded changes in the dT to an IMT while pruning entries in the dT that are no longer needed. To implement such mechanism, additional components aside from the dT must be added to the framework presented in the previous section.

First, each replica's IMT must have an IMT version number, which increases whenever new changes from the dT are committed to a particular IMT. To keep track of which IMT version each of the replicas have, a Replica ID Table (RIT) must be stored in the PalmPilot database. The RIT contains an entry for each replica that the PalmPilot has synchronized with so far, and each entry has the replica's ID, the replica's most recent IMT version number found by the system, as well as the date and time indicating when the last synchronization session with the replica occurred. The RIT version number consists of the highest IMT version found in the RIT.

The dT itself also needs to have a version number, which can be interpreted as the current version number of the global IMT, consisting of all the information in the most up to date local IMT plus any additional changes in the dT. The dT version will always be one plus the version of the latest local IMT, indicating that any new additions to the IMT recorded in the dT will belong to a newer version of the global IMT. When the changes in the dT are committed to a local IMT, the latter's version number is updated to match the dT's version number, which then increments by one. The entries in the dT are categorized by the dT version number at the time they were added. Therefore, from the dT and the RIT, one can determine exactly which entries are missing in any given outdated local IMT. Furthermore, one can also determine which entries in the dT are already present in all of the local IMTs.

During synchronization, whenever the IMT needs to be modified, an entry is first added to the dT, indicating whether a particular mapping needs to be added or deleted. This entry is added under the category corresponding to the current dT version number. Therefore, as in Figure 4.5, if the dT version number is 3, then any new future entries to the dT will be added under version 3.

The basic algorithm for committing the dT changes consists of only committing changes to the most recent version of an IMT. The assumption is that the Notes replication mechanism will eventually update all of the other older versions of the IMT in various replicas. The conduit can determine whether an IMT is the most recent or not by checking the RIT version number. During synchronization, if the RIT and IMT version numbers match, then the conduit should commit any new changes in the dT to the IMT upon completion. By only modifying the most recent version of the IMT, one avoids the potential conflicts that could occur if two replicas with modified IMTs were to replicate with each other.

The system does not need to keep those dT entries that have already been propagated to all the replicas. Therefore, the dT can be pruned by eliminating all the entries categorized by an IMT version number that is lower than the lowest IMT version indicated in the RIT. For example, in the scenario illustrated by Figure 4.5, the RIT indicates that all the replicas have an IMT of version 1 or later. Therefore, any changes made in version 0 can be discarded in the dT, which means that the last entry in the dT can be deleted.

However, if the user were to stop synchronizing with a particular replica which has an entry in the RIT, the dT could again grow arbitrarily large, as the IMT version number of that replica's IMT would never increase in the RIT, preventing the dT from being pruned. This problem can be addressed by using the timestamps in each of the RIT entries. During each synchronization session, if there is an RIT entry which has not changed for a long period of time, that entry is removed from the RIT, allowing the dT to be pruned of entries which were maintained only because of this outdated replica. The length of time for which to maintain an RIT entry could be set by the user, but should probably be constrained to less than 1 year.

4.3.2.4 Algorithm for Synchronization with Multiple Database Replicas

The pseudocode in Figure 4.6 describes the algorithm for synchronization with multiple database replicas by adding the steps necessary to maintain the dT, RIT and IMT version numbers. In the case where a replica does not contain an IMT yet, and the RIT indicates that there are other replicas that do have a version of the IMT, then the user is requested to first replicate with the other replicas in order to obtain a reasonably current version of the IMT. Also, new changes in the id mappings are registered in the dT first, and eventually they are committed to the IMT.

```
Synchronize(ReplicaId)
{
    IF (IMT does not exist) AND (RIT does not exist) THEN
        IMT = New IMT
        IMT.Version = 0
        RIT = New RIT
        RIT.Add(ReplicaId, CurrentDateTime, IMT.Version)
        RIT.Version = IMT.Version
        dT = New dT
        dT.Version = IMT.Version + 1
    END
    IF  (IMT does not exist) AND (RIT is not empty)  THEN
        DialogBox "Please replicate this database prior to first synchronization."
        RETURN
    END
    RIT.UpdateEntry(ReplicaID, CurrentTime, IMT.Version)
    WHILE (notesDoc = next document to be synched) DO
        PilotId = dT.Lookup(notesDoc.UNID)                    ;; checking the dT
        IF (PilotId is NOT valid) THEN
            PilotId = IMT.Lookup(notesDoc.UNID)        ;; checking the IMT
        END
        IF (PilotId) THEN
            pilotDoc = GetPilotDocument(PilotId)
            IF (pilotDoc is valid) THEN
                UpdatePilotRecord (notesDoc, pilotDoc)
            END
        ELSE
            pilotDoc = CreateNewRecord(notesDoc)
            dT.Add(pilotDoc.Id, notesDoc.UNID)
        END
    END
    WHILE (pilotDoc = next record to be synched) DO
        NotesUNID = dT.Lookup(pilotDoc.id)
        IF (NotesUNID is NOT valid) THEN
            NotesUNID = IMT.Lookup(pilotDoc.Id)
        END
        IF (NotesUNID) THEN
            notesDoc = database.GetDocument(notesDocUNID)
            IF (notesDoc is valid) THEN
                UpdateNotesDocument (notesDoc, pilotDoc)
            ELSE
                // Do nothing. Document doesn't exist yet in this replica!
            END
        ELSE
            notesDoc = CreateNewDocument(pilotDoc)
            dT.Add(pilotDoc.Id, notesDoc.UNID)
        END
    END
```

Figure 4.6 Pseudocode for multiple replica synchronization (continued in following page)

```
530     IF (IMT.Version = RIT.Version) AND (dT.Modified?) THEN
540         dT.Flush(IMT)
545         IMT.Version = dT.Version
550         RIT.Version = IMT.Version
560         dT.Version = dT.Version + 1
570    END
580     RIT.UpdateEntry(ReplicaID, CurrentTime, IMT.Version)
590     RIT.RemoveOldEntries()
600     dT.RemoveOldEntries()
}
```

Figure 4.6 (continued from previous page)

The code in lines 10 through 90 takes care of the boundary condition of a first time ever synchronization between this database and the device. New IMT, dT and RIT tables are created and initialized. These steps should be implemented atomically to ensure a consistent state. Lines 100 to 130 address the case of a first time ever synchronization between this replica of the database and the device, where the database has not recently been replicated with other replicas, and thus has no IMT document stored for this device. This is an unlikely case, and the algorithm cannot make progress, since it cannot construct a complete IMT from the dT alone. Thus, the algorithm has to abort, with a request to the user that they replicate the database first before attempting to synchronize. In line 140, the algorithm retrieves the IMT from the database, and updates the RIT entry for this replica, since the replica may now contain a more up to date version of the IMT due to having replicated with other replicas in the system.

Lines 150 through 320 take care of synchronizing any new or modified Notes documents to the Pilot. This is very similar to the code in figure 1, except for the presence of the dT. In line 160, the algorithm looks for the mapping first in the dT, and consults the IMT only if the entry is not found in the dT. Lines 330 through 520 synchronize any new or modified Pilot records to Notes. The only notable difference is in the ELSE clause in line 420, which handles the case where a document that was present in a different replica of the database, is not yet present in this replica. In this case, the algorithm does not create a new version of the document, but rather keeps the version in the Pilot. During subsequent synchronization, the algorithm will again try to save the changes to Notes, until it finds the document in one of the replicas. The fact that the document occurs in the dT/IMT indicates that it should exist in at least one replica of the database.

Lines 530 through 600 bring all the version numbers up to date, and remove any old data from the dT and RIT. Line 530 checks whether the IMT in the current replica is in fact the same as the most up to date IMT anywhere in the system, by comparing the IMT version number to the RIT version number. If the IMT is in fact older, then no further action is required. All the modifications are kept in the dT until an up to date replica is found. If the IMT is up to date, and the dT has been modified since the last flushing, any new additions to the dT are flushed to the IMT, and the IMT and RIT version numbers are incremented to match the current dT version. Finally, the dT version itself is incremented, indicating that any new future additions to the dT will belong to a new IMT version. Lines 530 through 570 should be implemented atomically, to ensure a consistent state. Line 580 updates the RIT entry for this replica, by time stamping it with the current time, and by updating the version number (which may have been incremented in line 550). Line 590 invokes a routine that iterates over the RIT, and removes any entries who's time stamps are older than the limit for how long to maintain information about a replica. Line 600 invokes a similar routine which scans the dT for any entries that pertain to versions of the IMT that no longer occur in the RIT, and removes them.

This algorithm does not describe how to deal with failure conditions. In general, it is a good idea for any implementation to maintain backup copies of the dT and RIT in case the device suffers a memory reset. This includes backing up data that was not written to Notes because the document to which it pertains does not yet exist in the current replica.

4.3.2.5 Sample Scenario for synchronization with multiple replicas

Returning to the example with Ben, suppose that he has not synchronized his PalmPilot with neither his server or local replicas, both of which are currently in sync. Figure 4.7 shows the initial state of the PalmPilot and the replicas.

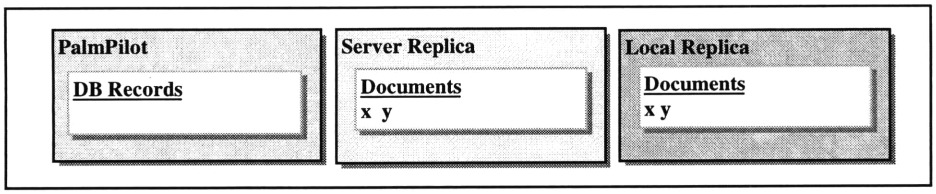
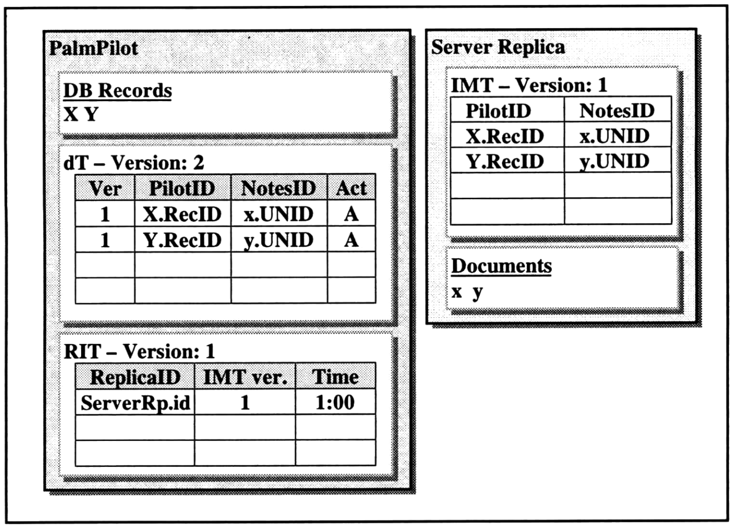


Figure 4.7 Initial state of multiple replica synchronization sample scenario.

Ben first synchronizes his PalmPilot with the server replica. Figure 4.8 illustrates the state of the PalmPilot and server replica after this synchronization. Since the server replica does not have an IMT yet, and the PalmPilot does not have a RIT either, the system will initialize all the necessary tables. A new IMT with version 0 will be created in the server replica, and a new RIT and dT (initialized to versions 0 and 1 respectively) will be added to the PalmPilot. The conduit will also add an entry to the new RIT, indicating that the server replica currently has the IMT version 0. After these initialization steps, the conduit will go through the new and modified documents in the server replica, and for each of those documents, it will try to find a corresponding record in the dT and IMT. Since both are currently empty, the conduit will assume that the documents do not exist in the PalmPilot, it will create the corresponding records, and it will add the appropriate id mappings in the dT. Next, the conduit will go through the new and modified records in the PalmPilot, but since the latter was initially empty, nothing is done in this step. Finally, since the RIT and IMT versions are equal and the dT has been modified since the last time it was committed to an IMT, the conduit will conclude that it is dealing with an up to date IMT, and it will therefore flush all the changes in the dT to the IMT, increase both the IMT and RIT versions to 1, bump the dT version to 2, and update the RIT entries accordingly. Note that we can determine if a dT has changed since the last time it was committed to an IMT by checking whether there are any entries in the dT categorized under the dT version number.



PalmPilot

DB Records

X Y

dT – Version: 2

Ver	PilotID	NotesID	Act
1	X.RecID	x.UNID	A
1	Y.RecID	y.UNID	A

RIT – Version: 1

ReplicaID	IMT ver.	Time
ServerRp.id	1	1:00

Server Replica

IMT – Version: 1

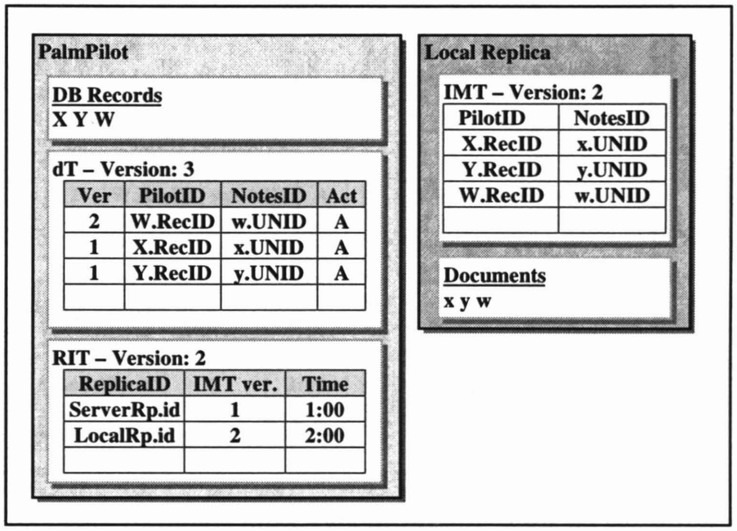
PilotID	NotesID
X.RecID	x.UNID
Y.RecID	y.UNID

Documents

x y

Figure 4.8 State of tables after synchronizing with server replica

Ben then goes home and tries to synchronize his PalmPilot with his local replica. Note that he has not replicated the server and local replicas at this point following his last synchronization. The conduit will notice that the local replica does not have an IMT, but since it knows that there is an IMT for this database from the RIT, it tells the user to replicate with a replica that contains an IMT. Ben therefore replicates his server and local replicas, he modifies document x, and he adds document w. He then tries to synchronize with his local replica. Figure 4.9 shows the state of the tables after this synchronization. When the conduit goes through the Notes documents, it will first try to propagate the changes made to document x. It looks up in the dT, it finds the record that corresponds to document x in the PalmPilot, and it applies the appropriate changes. Then, for the new document w, the conduit will fail to find a mapping in both the dT and IMT, indicating that a new record must be created in the PalmPilot and a new mapping added to the dT under version 2. Finally, since the IMT and RIT versions are both the same (1) at this point and the dT has been changed since it was last committed to an IMT, the conduit will flush the new dT changes to the local replica's IMT, and it will set both IMT and RIT versions to 2, and increase the dT version to 3.



PalmPilot

DB Records
X Y W

dT – Version: 3

Ver	PilotID	NotesID	Act
2	W.RecID	w.UNID	A
1	X.RecID	x.UNID	A
1	Y.RecID	y.UNID	A

RIT – Version: 2

ReplicaID	IMT ver.	Time
ServerRp.id	1	1:00
LocalRp.id	2	2:00

Local Replica

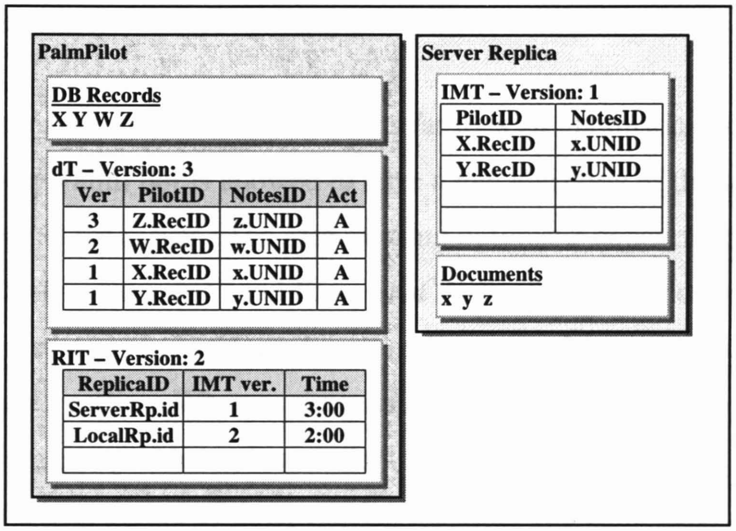
IMT – Version: 2

PilotID	NotesID
X.RecID	x.UNID
Y.RecID	y.UNID
W.RecID	w.UNID

Documents
x y w

Figure 4.9 State of tables after synchronizing with local replica.

Ben then goes back to work, and on the way, he makes modification to record W in the PalmPilot. Once at his office, Ben also creates a new document z in his server replica, and he tries to synchronize. Figure 4.10 shows the state of the system after this synchronization. At this point, he has not replicated his server and local replicas, and therefore his server replica does not yet contain document w which was created earlier. When the conduit goes through the PalmPilot records, it will notice that record W was modified, and it will try to propagate these changes to Notes. It looks up the dT, and it does find the id of the corresponding Notes document, but when it tries to access the document in the server replica, it will fail since the latter has not been propagated from the local replica. In this case, the conduit will just leave record W marked modified, so that the next synchronization session will try propagating those changes again. This shows how the proposed framework avoids the creation of duplicates that could occur during synchronization with multiple outdated replicas. Concluding the synchronization session, the conduit will add a new mapping in the dT for document z, and it will not flush the changes in the dT to the server replica IMT, since it knows from the RIT that there is another replica with a more recent IMT.



PalmPilot

DB Records
X Y W Z

dT – Version: 3

Ver	PilotID	NotesID	Act
3	Z.RecID	z.UNID	A
2	W.RecID	w.UNID	A
1	X.RecID	x.UNID	A
1	Y.RecID	y.UNID	A

RIT – Version: 2

ReplicaID	IMT ver.	Time
ServerRp.id	1	3:00
LocalRp.id	2	2:00

Server Replica

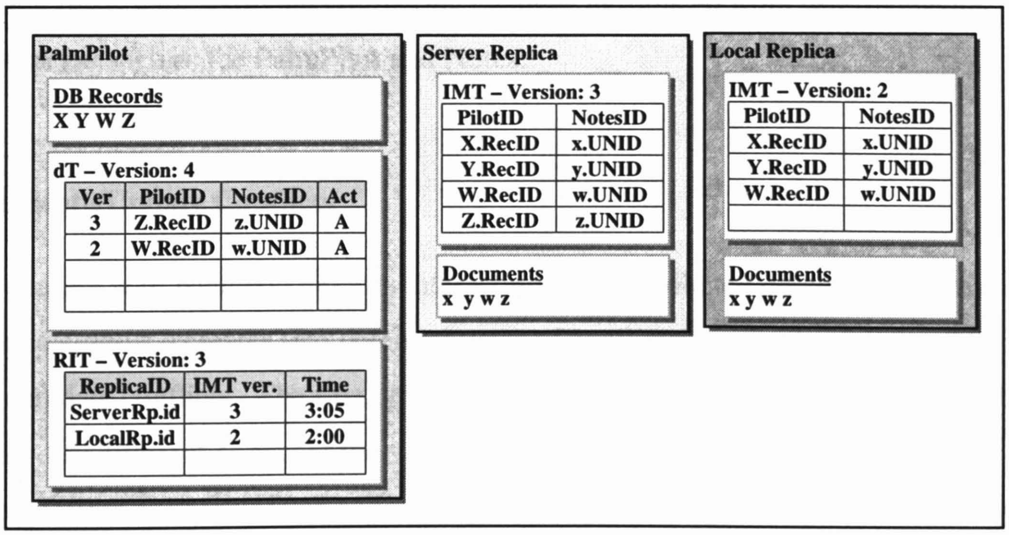
IMT – Version: 1

PilotID	NotesID
X.RecID	x.UNID
Y.RecID	y.UNID

Documents
x y z

Figure 4.10 State of tables after synchronizing with server replica again.

Ben then replicates his server and local replicas, which propagates the newer IMT from the local replica to the server replica. Immediately afterwards, he tries to synchronize with the server replica once again. This time, the conduit will realize it is synchronizing with the most recent version of a local IMT, and since there have been changes to the dT, it will flush them to the server replica's IMT, updating its version, while incrementing the dT version to 4. Figure 4.11 shows the final state of all the components in our scenario.



PalmPilot

DB Records
X Y W Z

dT – Version: 4

Ver	PilotID	NotesID	Act
3	Z.RecID	z.UNID	A
2	W.RecID	w.UNID	A

RIT – Version: 3

ReplicaID	IMT ver.	Time
ServerRp.id	3	3:05
LocalRp.id	2	2:00

Server Replica

IMT – Version: 3

PilotID	NotesID
X.RecID	x.UNID
Y.RecID	y.UNID
W.RecID	w.UNID
Z.RecID	z.UNID

Documents
x y w z

Local Replica

IMT – Version: 2

PilotID	NotesID
X.RecID	x.UNID
Y.RecID	y.UNID
W.RecID	w.UNID

Documents
x y w z

Figure 4.11 Final state of all tables.

4.4. Conflict Resolution

The data synchronization model described thus far uses an optimistic concurrency control strategy, where any PalmPilot is allowed to have a mutable copy of the same Notes document residing in the server. The idea of using optimistic concurrency control was first introduced by Eswaran *et al.*[10] and Kung *et al*[11] in the late 70's and early 80's. Since then, a number of systems have adopted such replica control strategy, including Coda and Bayou as mentioned earlier. Whereas the pessimistic approach avoids potential conflicts, it creates a much bigger problem when a client with shared control of an object remains disconnected for a long period of time, causing the other clients to defer any updates to the object. Under the premise that multiple clients do not often modify the same object concurrently, the optimistic approach is clearly better in terms of achieving the highest possible availability of data for all clients. However, with availability comes the price of potential conflicts, particularly when two clients actually modify the same document at roughly the same time. With optimistic concurrency control, the basic idea consists of letting the few potential conflicts occur while resolving them later. There have been a number of projects addressing the issue of resolving conflicts in a distributed system using optimistic concurrency control, including Adya *et al.*[12] and Agrawal *et al.*[13] The nature of conflict resolution addressed in this thesis differs from these past projects because it deals with resolving conflicting changes introduced from heterogeneous environments such as the PalmPilot and Notes.

4.4.1 Update/Update Conflicts

When dealing with handheld devices interacting with distributed database systems, the optimistic approach creates two potential types of conflicts: update/update conflicts and remove/update conflicts. The first type of conflict occurs when a single user or multiple users modify different copies of the same document in between replication and synchronization sessions. Such copies may reside in the handheld device and in any of the database replicas. Some examples of update/update conflicts include:

- Document x exists in both Replica A and in the PalmPilot. Ben modifies document x in both systems, and then tries to synchronize.
- Document x exists in both Replica A and B. Two different users modify document x in both replicas. During the usual scheduled time, both replicas replicate, and different versions of document x will be encountered.

Lotus Notes already provides a mechanism to resolve conflicts like the second example above, where the same document is modified in different replicas in between replication. When it detects a modification conflict, Notes will create the different modified documents in both replicas, thus ensuring that both will have all the versions of the document. However, out of the possible versions of the same document, Notes will indicate a preferred version which corresponds to the most recently modified document. All the other versions will be considered as replication conflict documents. The premise is that once replication is done, and regardless of which replica is being accessed, the user will know which version of the document was modified most recently, and this version will most likely have the changes that the user wants. The user will also see the other replication conflict documents which were created, and it is up to the user to decide which version of the document to keep. Notes essentially leaves it up to the user to resolve modification conflicts by ensuring that the user has all the conflicting versions of the document in question. This approach works well because the user will most likely be the best judge on how conflicts should be resolved, but it relies on the assumption that update/update conflicts do not occur very often. If that assumption was not the case, every Notes database would be filled with conflict documents which the user must clean up.

In the first example mentioned above, the same document is modified in both Notes and the PalmPilot in between synchronization. Such conflicts can also be handled in the same manner in which conflicts between replicas are handled by Notes. During synchronization, both versions of the conflicting document are created in the replica and in the PalmPilot, leaving it up to the user to decide which one should be kept. A more assertive conflict resolution approach would consist of keeping only the most recently modified version of the document in both handheld and replica database. However, determining the most recent version depends on keeping track of when documents were modified. Notes automatically records the date and time when a particular document changed. On the other hand, the PalmPilot and other handheld devices do not always store the modification time in each record. Thus, given a corresponding record and document pair, it is difficult to determine which one has the most recent modifications. Even if the handheld device did keep track of modification times, it is likely that the handheld and replica times will be out of sync. Therefore, any conflict resolution mechanism dependent on clock synchronization would be unreliable for resolving conflicts between handheld devices and distributed database systems.

There also exists many other approaches of resolving update conflicts between the PalmPilot and Notes. For some databases, it may make sense to always ignore the handheld version when a conflict is detected. Others may resolve the conflict according to the value of a particular field in the document. The nature of the database will dictate the update conflict resolution approach. For our synchronization framework and the prototypes, we have chosen to adopt the above approach of keeping both conflicting versions in both platforms, mainly because it is a conservative approach that will avoid loss of data regardless of the type of database being manipulated.

4.4.2 Update/Update Conflicts with Multiple Replica Synchronization

The types of update/update conflicts described above assumed that there was only one PalmPilot synchronizing with one database. There are actually other potential conflicts related to race conditions which occur when there is one PalmPilot synchronizing with multiple replicas of the same database. Suppose that we have one PalmPilot which synchronizes with replicas A and B, which also replicate between themselves regularly. The scenarios described below start out with all three locations with the same identical updated document x.

Scenario 1:

- Modify x in replica B
- Modify x in replica A *(document x in replica A is now the most up to date copy)*
- Synchronize PalmPilot with replica A
- Synchronize PalmPilot with replica B

Problem: The modified version in replica A is the latest version, and therefore it should be the one ultimately kept in the PalmPilot. This remains true up to step 3. However, in step 4, the synchronization process overwrites the original updated document in the PalmPilot with an older version from replica B. This occurs because both versions of document x in replica A and B have the same UNID. In the IMT, this UNID is mapped to one particular record in the PalmPilot which will receive the modifications from both replicas. Since there is no date/time information indicating which modifications a particular record contains, any modification, old or new, will be propagated from Notes.

Scenario 2

- Modify x in replica B
- Modify x in PalmPilot
- Synchronize PalmPilot with replica A
- Synchronize PalmPilot with replica B

Problem: Very similar to the problem encountered in Scenario 1, except in this case, the most updated version of document x originated from the PalmPilot. In the end, the PalmPilot is again left with an outdated version of document x that came from replica B.

In both scenarios, the source of the problem lies in the fact that there is no mechanism to determine whether a particular PalmPilot record contains the most current information. In step 4 of both scenarios, the Notes document in replica B contains old modifications which overwrite the newer modifications residing in the PalmPilot, since the synchronization process did not realize that PalmPilot had the newer version. This situation is actually temporary, since Notes replication will eventually propagate the most up to date version of document x to all replicas. In the scenarios above, when replicas A and B replicate, conflict documents will be created since a modification conflict will have occurred, given that document x was modified in both replicas. Any subsequent synchronization will bring both versions of document x to the PalmPilot, and the user can then determine which one to use. However, there is still a small gap of time in which the PalmPilot user could potentially be dealing with stale data.

The simplest solution to this problem would consist of storing a timestamp in each PalmPilot record, indicating when they were last modified. Such additional information would allow the synchronization process to determine when to ignore modifications when the PalmPilot has a newer version. However, this approach would require changing the existing PalmPilot application to support this functionality, and such approach would violate our requirement of preserving the existing handheld device applications. Another solution consists of extending each entry in the RIT to also include the time when each replica was last modified. Whenever a document is modified, Notes updates the replica's last modified date/time. With this extra information during a synchronization session with a particular replica, the handheld can determine if there are other replicas that may potentially contain newer updates. If this is the case, then the synchronization process should warn the user that the PalmPilot may contain stale data.

4.4.3 Remove/Update Conflicts

Remove/update conflicts occur when the user deletes a document in Notes and modifies the corresponding record in the PalmPilot, or vice-versa. The same could occur between Notes replicas, where the same document is deleted in one replica and modified in another one. Once again, there exists a variety of strategies to solve this type of conflict, ranging from always ignoring any deletions done in the PalmPilot, or always propagating deletions from Notes to the PalmPilot even if there are modifications to the record in the PalmPilot.

For remove/update conflicts within Notes replicas, the Notes replication mechanism always restores the document in the replica in which it was previously deleted. This approach has the advantage that modifications are never lost. For remove/update conflicts that occur between Notes and the PalmPilot, we propose to adopt the same approach that Notes uses for conflicts within replicas. If a PalmPilot record is deleted, but its corresponding document in Notes was modified, a new PalmPilot record containing the modifications is created during the next synchronization session. In the same way, if a Notes document is deleted but its PalmPilot counterpart is modified, a new Notes document with the PalmPilot modifications is created. For the purposes of a general synchronization framework, this conservative approach will work for most types of databases, as long as the conflicts do not occur too often. Developers will always have the option of extending the framework with additional conflict resolution strategies.

5. Authentication of PalmPilot during synchronization

Allowing handheld devices to synchronize with information residing in Lotus Notes databases creates a security problem by adding a potentially insecure channel of access into the Lotus Notes network. More specifically, the desire to support remote handheld synchronization leaves the Notes system vulnerable to attacks from unauthorized handheld devices trying to access the system remotely. Part of the problem is that current handheld devices do not support robust authentication mechanisms that ensure that only authorized handheld devices synchronize with a particular desktop computer. As mentioned earlier, for example, PalmPilot's Hotsync Manager authenticates a PalmPilot prior to synchronization by only checking a user name and id number stored in the handheld device. Aside from the handheld to desktop computer authentication problem, there is also the issue of authenticating a handheld user with Lotus Notes. Currently, a Notes user in a client machine authenticates himself/herself with a password and a personal ID file residing in the client. If the user tries to remotely access the Notes network using a handheld device, he/she will need some secure mechanism to present both a password and personal ID file for authentication with Notes.

The rest of the chapter is organized as follows: Section 5.1 provides some background information by briefly describing Lotus Notes' security mechanism, and Section 5.2 describes a simple alternative for solving both authentication problems mentioned above. The prototype described in Chapter 6 includes an implementation of this alternative. Note that the simple solution presented in this chapter does not aim to be robust enough to effectively solve the problem of securely accessing Notes from a handheld device. A detailed discussion on the correctness of robust security mechanisms goes beyond the scope of this thesis, since the latter only attempts to raise the issue of the need for better security mechanisms. Therefore, while the solution presented in section 5.2 serves as a possible starting point towards solving the security problem in question, the development of more robust authentication and encryption protocols is left for future work.

5.1. Lotus Notes Security

One of the strengths of Lotus Notes lies in its security model. Notes uses a validation and authentication mechanism to ensure that only authorized users access the information in the servers. Each user has a personal ID file, which contains the user name, a set of certificates, and a pair of private and public encryption keys generated using the RSA Cryptosystem. These keys are used both for authentication and data encryption. When the user tries to access a Notes server, the latter utilizes the information in the user's personal ID file to check whether the user has access to its information. More detailed information on how Notes uses the information in the personal ID file for authentication and validation can be found in the Lotus Notes Server Administration Guide[14].

The heart of Notes' validation and authentication mechanism lies in the information present in the user's ID file. To protect such information, Notes adds another layer of security by allowing users to encrypt the information in the ID file using a secret key generated from a password chosen by the user. If an ID file is protected, when the user tries to access a particular server, the Notes client with the user's ID will prompt the user for the appropriate password. Only with the right password will the client decrypt the user ID, and use its information to access the server. The Notes client tries to prevent password spoofing by increasing the delay in between the input of incorrect passwords. Users are actually not required to encrypt their user ID file with a password, but such practice is strongly recommended. For the remainder of this discussion, we will assume that the user has protected the ID file with a password.

5.2. Providing remote handheld access without compromising Notes security

The key parts of Notes' validation and authentication mechanism consist of the user ID file and its password. When synchronizing a handheld device with Notes, both the ID file and password must be available. Most of the times, the user will synchronize with a Notes client residing in the user's own desktop machine. In this case, the user's personal ID file will already be present in the computer, requiring the synchronization process to only have to prompt for the user's password to proceed with the validation and authentication process. If the Notes client is already running during synchronization, the user also has the option to allow the synchronization process to use the same password previously entered by the user from the Notes client. This option essentially lets the Notes client share the user's password with other Notes applications, such as the synchronization process.

The mechanism described above will work as long as the user always synchronizes directly with the desktop computer, making it possible to enter the Notes password directly from the desktop computer when prompted by the Notes client. However, current handheld technology allows users to also synchronize remotely with a particular machine in the network. Such remote synchronization model presents a problem if the Notes client prompts the user for a password during remote synchronization, especially since current handheld devices do not support a secure method for entering a password in the device and transmitting it securely to a desktop machine through the network. Assuming that the user always synchronizes with a machine that has the user's personal ID file, the basic problem then becomes how to securely transfer a clear text password entered in the handheld device to the desktop computer through an open unprotected network.

A simple solution for transmitting a password securely consists of first encrypting the Notes password in the desktop machine using a newly generated symmetric key. The encrypted password is then stored in the desktop machine itself, and the key used to encrypt the password is stored in the handheld device. During remote synchronization, the stored symmetric key is transmitted from the handheld device to the conduit, which uses the key to decrypt the password stored in the desktop computer. The decrypted password is then used to access the Notes network. Our first prototype of the Notes Mail PalmPilot conduit, described in Chapter 6, uses this approach for secure remote synchronization. This solution has the advantage that the password itself is never transmitted over the wire, and it also provides a better authentication mechanism since only the handheld device with the right key will be able to synchronize and access the

Notes network remotely. However, this approach is also vulnerable to various malicious attacks. For example, this approach is exposed to replay attacks, since a hacker can potentially intercept the key stored in the handheld while it is being transmitted from the handheld device to the desktop computer, and thereafter use it to illegally synchronize with the Notes network from a handheld device.

While the approach described above provides a better secure solution for remote synchronization, other alternatives with better protection against security attacks should be explored in future works.

6. Case Study: Lotus Notes Mail Synchronization Conduit

In order to better understand the practical issues of implementing a synchronization framework for the PalmPilot and Lotus Notes, a prototype conduit which allows users to read and edit their Lotus Notes mail from their PalmPilots was designed and implemented for this thesis project. At the time of development, there was no commercial product available that offered such capability, and given that there was a great demand for such conduit within Lotus, the prototype seemed to be a good starting point, especially since there was a large group of people within the company available to test it.

Lotus Notes Mail consists of a regular Notes database with special views and folders designed to organize incoming and outgoing mail. The Inbox folder contains all the incoming mail, which are marked unread until first opened by the user. The Trash folder contains all the mail that the user wishes to purge from the system at a later time. The Drafts view contains all the memo drafts which will be sent at a later time, while the Sent view contains a copy of all the outgoing mail that the user chose to save. Users may also create additional folders or views within the Mail database, and move messages from any of the default folders or views to the custom created ones. When the user deletes a particular message, it is first placed in the Trash folder. The message is only cleared from the system when the user chooses to purge the Trash folder.

While the Notes Mail databases are usually kept in the server machine, users often create a local replica of their mail database in their own computers. For notebook users, this allows them to manage their mail on the road while disconnected, and once they come back to the office, they replicate with the server copy to upload and receive any new changes. Some notebook users also directly access the server copy of the Notes Mail database through a modem connection.

On the handheld side, the PalmPilot Professional version comes with a Mail application built in. This application organizes messages in five different categories: Inbox, Outbox, Deleted, Filed and Draft. The Inbox, Outbox, Deleted and Draft categories are equivalent in function to Notes' Inbox, Send, Trash and Drafts views/folders. The Filed category is used to store those messages from the Inbox that the user wishes to keep for a while. When the user composes a new message, and chooses to send it, it is automatically placed in the Outbox category. The Outbox category can also be viewed as a queue of messages that need to be sent once the PalmPilot is connected to the network.

6.1. Design Goals

In the process of developing the Notes Mail conduit, the primary goal was to allow users to utilize the existing built in Mail application in the PalmPilot to read, compose, edit, and send messages in their Lotus Notes Mail database. Furthermore, we assumed that users could synchronize with either the server or local replicas of their mail database, depending on the availability of a network connection.

Given the limited viewing capabilities of the PalmPilot, only the plain text of mail messages was preserved during the conversion from Notes document to Pilot record. Therefore, graphics, attachments, document links and other possible multimedia content in Notes mail messages were not transferred to the PalmPilot.

Even though the different default views and folders of the Notes mail database map directly to corresponding categories in the PalmPilot's Mail application, it did not make sense to simply apply a two-way synchronization algorithm for the contents of every view and folder. More specifically, the user probably does not want to have every message ever saved in the Notes Sent view transferred to the PalmPilot's Outbox category. Therefore, the conduit only transfers messages from the PalmPilot Outbox category to Notes' Sent view, and not vice-versa. Also, given that PalmPilot users cannot edit the messages in the Inbox category, only a one-way synchronization algorithm from the Notes Inbox to the PalmPilot Inbox needs to be applied. For draft messages, the direction of synchronization will depend on the user's preference. If the user wants both the PalmPilot Draft category and Notes Draft folder to always contain the exact same draft changes, then a two-way synchronization would be required. However, the user may want to just propagate draft changes from the PalmPilot to Notes, or vice-versa. The same applies for deletions, since the user may not want to propagate deletions from Notes to the PalmPilot, or vice-versa.

Given the possible different ways that users may want to handle the synchronization of outgoing, incoming, draft and deleted messages, the conduit must allow the user to choose how to handle each of the different types of mail messages. For Inbox messages, there should be the option to enable or disable downloading inbox messages from Notes to the PalmPilot. For outbox messages, there should be the option to enable or disable sending the messages that are in the PalmPilot's Outbox category. For drafts messages, there should be the options to ignore draft changes or to propagate draft changes from PalmPilot to Notes and vice-versa. For deletions, similar options should also be provided, allowing the user to either not propagate deletions between Notes and PalmPilot, or to propagate deletions from Notes to PalmPilot, vice-versa, or both ways.

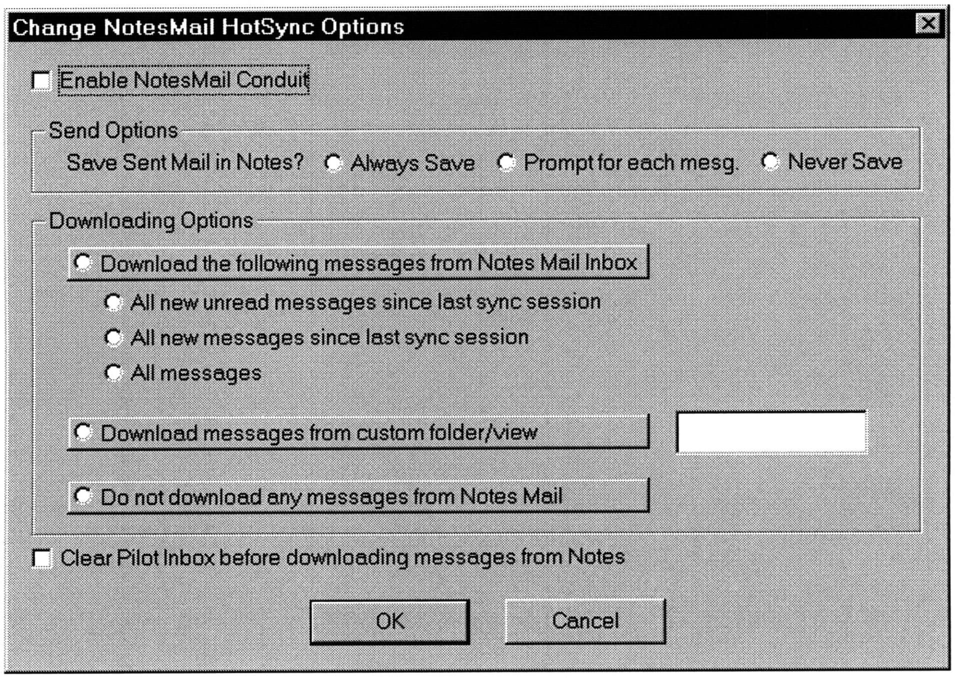
When downloading inbox messages from Notes to the PalmPilot, there should also be options to allow users to select a small subset of inbox messages, given that a Notes Mail database can be arbitrarily large. Some of the useful options for users include downloading only new messages since last synchronization sessions, downloading only new and unread messages since last synchronization sessions, downloading all messages received since a particular date, and downloading all messages in the Notes Inbox.

Even with the options provided above, there will probably still be some users who would want to download a different set of Notes mail messages into their PalmPilots. For such users, there should be an additional option for downloading messages from an arbitrary Notes view or folder into one of the PalmPilot mail categories. Ideally, the conduit should allow the user to specify the name of the Notes view or folder to be synchronized. This option would work well under the following two scenarios. Suppose Ben Bitdiddle wants to have all the messages he has received from Alissa P. Hacker in his PalmPilot. To do so, he can create a view named "Foobar" which only selects those messages from Alissa, and then tell the mail conduit to download the messages in the "Foobar" view into the Filed category in the PalmPilot. On the other hand, suppose Ben finds one particular message which he wants to have in his PalmPilot. To do so, he creates a folder named "Pilot" in Notes, and he places that message in the new folder. He then configures the mail conduit to place the contents of the "Pilot" folder into the Inbox category in the PalmPilot, and synchronizes.

6.2. Early Prototype

The first prototype of the Notes Mail conduit was developed in the beginning of the summer of 1997. Since it was originally intended as an experiment to explore the general implementation issues of transferring Lotus Notes data to PalmPilot, this first prototype did not have many of the features described in the design consideration given above. It supported transferring outgoing messages from the PalmPilot to Notes, and downloading Inbox messages from Notes to the PalmPilot's Inbox category. In the conduit's configuration, the user could choose to download all new messages since the last synchronization session, all new unread new messages since the last sync session, or all the messages in the Notes Inbox. The conduit also allowed users to download all messages in a particular Notes view/folder into the Filed category in the PalmPilot.

Draft synchronization and propagation of deletions were not supported in this prototype, since there was no mechanism keeping track of how documents in Notes mapped to records in the PalmPilot. Therefore, with this prototype, the PalmPilot was practically a cache where users could store messages to be read on the road, and where users could store new composed messages to be sent at a later time when network connection was available. With this model, it was not necessary to keep a mapping of Notes document to PalmPilot record id, since propagation of changes and deletions was not yet supported. Therefore, many of multiple replica synchronization issues described in this thesis did not apply for this prototype. The configuration dialog box for the conduit in this first prototype is shown in Figure 6.1, which summarizes the features that were supported in this version. The code used to implement this first prototype version is included in Appendix A.



Change NotesMail HotSync Options

□ Enable NotesMail Conduit

Send Options

Save Sent Mail in Notes? ○ Always Save ○ Prompt for each mesg. ○ Never Save

Downloading Options

○ Download the following messages from Notes Mail Inbox

○ All new unread messages since last sync session

○ All new messages since last sync session

○ All messages

○ Download messages from custom folder/view

○ Do not download any messages from Notes Mail

□ Clear Pilot Inbox before downloading messages from Notes

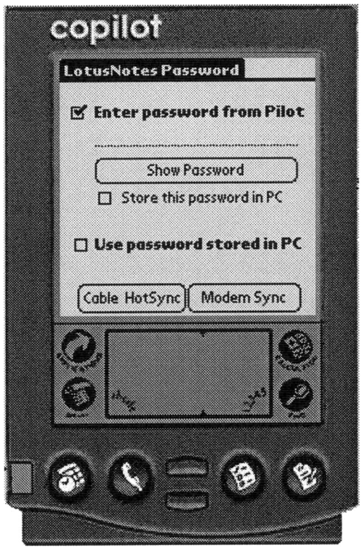
Figure 6.1 Main configuration setup window for Notes Mail Conduit.

Notes Mail messages sometimes may contain over 1 Mb of text, and synchronizing such messages would quickly overflow the memory constraints of the PalmPilot. To address this potential problem, this first prototype truncated any mail message that contained over 20K of text during the conversion of Notes document to PalmPilot record. For those messages which were truncated, a warning message was appended to the top, indicating to the user that the contents of the message being read is incomplete.

With this first prototype, different ways to access the Notes network from the PalmPilot remotely were also explored. As explained earlier, when synchronizing with a client computer remotely using a modem, there needs to be some mechanism to enter the Notes password from the PalmPilot and transfer that information securely over the network.

This is not an issue when directly synchronizing with the desktop computer since the user can authenticate himself by entering the password directly in the desktop computer. For this prototype, authentication problem was addressed by first encrypting the user's password, then storing the encrypted password in the desktop computer and the key to decrypt the password in the PalmPilot. This way, in the beginning of a remote synchronization session, the conduit would first probe the PalmPilot for a valid key, which it would then use to decrypt the stored encrypted password in the desktop. If successful, the conduit will then have the password necessary to access the Notes network. This method has the added benefit that only a valid PalmPilot, with the right key, can synchronize with the user's desktop and access his Notes information.

The process of creating a key, encrypting the password, and storing the key in the PalmPilot was added to the setup of the conduit. Whenever the user changed his password, he/she would have to run the conduit's setup configuration in order to create a new key and store a new encrypted password. For the key generation, Microsoft's Crypto API was used, and it provided all the calls necessary to generate a 40-bit key used to encrypt and decrypt a string of text. For those users who still did not have a valid password key stored in the PalmPilot, there is the option to enter the password in the PalmPilot itself prior to a remote synchronization session. In this case, the password is stored in the PalmPilot temporarily, and during synchronization, the conduit reads the password from the PalmPilot. The risk with this approach is that the password is transferred over the network in plain text, leaving it target for potential hacker attacks. We left the option under the assumption that the user would only use it within a secure network link. Also, if the password is entered from the PalmPilot, the user has the option to permanently store the password in the desktop, in which case the conduit would encrypt the password before storing it, and store the key to the encrypted password in the PalmPilot. In future remote synchronization, the user can use such key to log in instead of entering a password. Figure 6.2 shows the login screen in the PalmPilot which was developed for password input prior to synchronization. The code used to develop this PalmPilot application is included in Appendix B.



LotusNotes Password

☑ Enter password from Pilot

□ Store this password in PC

□ Use password stored in PC

Figure 6.2 PalmPilot application used to enter Notes password.

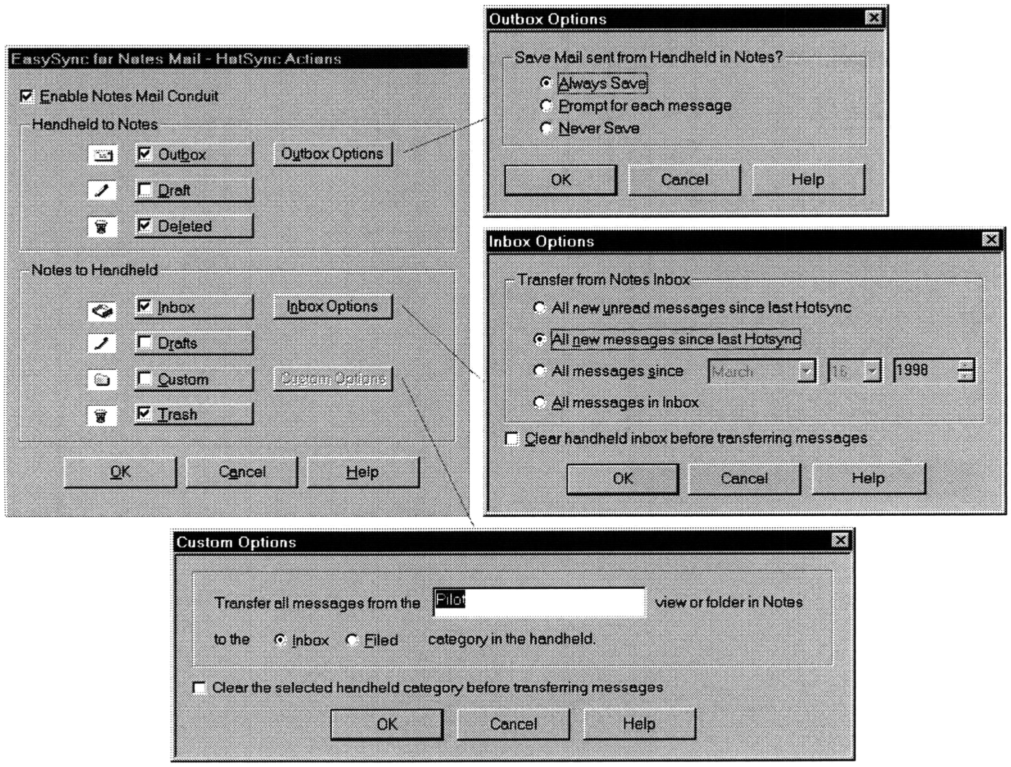
This first prototype was distributed to people within Lotus and IBM, and it was also made available to the public through the Lotus Research web page. Some of the early informal feedback obtained from users indicated some interesting behavior in terms of managing Notes mail from a PalmPilot. It seemed that a popular option was the one which allowed users to download messages in a particular custom Notes view/folder into the PalmPilot's Inbox. Users utilized this option to download specific messages into the PalmPilot, such as messages containing meeting schedules or driving directions which the users would need while on the road. Not many users who synchronized with the desktop seemed to use any of the options to download Notes Inbox messages into the PalmPilot Inbox. However, users who synchronized remotely with a modem often took advantage of such features to read any new messages while on the road. Some even mentioned that they no longer carried their notebook computers on certain trips since they could just use their PalmPilots to check for any important messages.

6.3. Transitioning from Research Prototype to Commercial Product

When the prototype was first released in July of 1997, Lotus did not have an official version of a Notes Mail conduit available for customers, despite the high demand for such a conduit. Given that this prototype seemed to have many of the basic functionality needed for a commercial version of the conduit, Lotus decided to use the prototype as the base code for a commercial version.

Two critical features were missing in the first prototype which needed to be added for the commercial product: the ability to propagate changes to mail drafts, and the ability to propagate deletions. In order to implement these features, a table mapping Notes document id's to PalmPilot record id's was needed, since the conduit needed some way to find the corresponding Notes document of a modified PalmPilot record, or vice-versa, in order to apply the changes accordingly. The framework presented earlier in Section 4.3 would had been ideal to solve this id mapping problem. However, there was also the need to integrate the Notes Mail conduit into a suite of existing conduits known as EasySync 1.0 for Notes, which synchronized the Notes Address Book, Calendar and To Do databases with the PalmPilot. Therefore, in an effort to simplify the integration process, the approach of the other existing conduits to solve the id mapping problem was used. This approach consisted of keeping the IMT in Notes, but in order to address the multiple replica synchronization issues discussed earlier, this approach prevents the user from synchronizing with a different replica until the most updated IMT has propagated to the replica that the user is currently synchronizing with. Therefore, suppose Ben has his main server replica of his mail database, along with a local replica in his notebook. If Ben makes changes directly to his server copy of the mail database, synchronizes with it, and goes home with his notebook and PalmPilot, he will not be able to synchronize his PalmPilot with the local replica of his mail database in his notebook until he replicates that local replica with the server copy.

With the IMT handling approach described above, it was possible to implement the ability to propagate draft changes and deletions to the Notes Mail conduit. For this commercial version, the propagation of read/unread marks was also included. Therefore, when a message is marked unread in Notes, once it is synchronized to the PalmPilot, it is also marked unread in the handheld. If the user reads the same message from the PalmPilot and marks it read, then during the next synchronization session, the same message will be also marked read in Notes. Another additional feature consisted of detecting when a Notes messages had a file attachment, and indicating in the beginning of the body of the translated PalmPilot record that the original message in Notes contains an attachment. The synchronization options of this commercial version are shown in Figure 6.3, which displays the dialog boxes for configuring the conduit. For the final release version, shipped in January 1998 as part of the Lotus EasySync 2.0 for Notes, the Notes Mail conduit supported all the features which were outlined in the design goals described in Section 6.1.



EasySync for Notes Mail - HotSync Actions

☑ Enable Notes Mail Conduit

Handheld to Notes

☑ Outbox

□ Draft

☑ Deleted

Notes to Handheld

☑ Inbox

□ Drafts

□ Custom

☑ Trash

Outbox Options

Save Mail sent from Handheld in Notes?

● Always Save

○ Prompt for each message

○ Never Save

Inbox Options

Transfer from Notes Inbox

○ All new unread messages since last Hotsync

● All new messages since last Hotsync

○ All messages since March 16 1998

○ All messages in Inbox

□ Clear handheld inbox before transferring messages

Custom Options

Transfer all messages from the Pilot view or folder in Notes

to the ● Inbox ○ Filed category in the handheld.

□ Clear the selected handheld category before transferring messages

Figure 6.3 EasySync 2.0 Mail Conduit options

6.4. Additional Features for Future Versions of Mail Conduit

There are a some additional changes which we would like to see implemented in future versions of the Notes Mail conduit.

1. The algorithm for handling multiple replica synchronization described in Section 4.3 should be integrated into the conduit, eliminating the current constrain which forces users to replicate whenever they wish to synchronize with a different replica.
2. Currently users can only change their synchronization options from the desktop through the menus shown in Figure 6.3. There will be times when the user will want to change such settings for a particular synchronization session initiated remotely. Therefore, the same configuration options shown above should be made available in the PalmPilot, allowing users to choose their settings of choice prior to a remote synchronization session.
3. Note that for the commercial version, the remote log in and password transfer features which we had experimented with in the prototype version were not included due to shipping date constraints. In order to ensure the security of the Notes network during remote synchronization, future versions of the conduit should include a secure password transfer mechanism.

The remote operation options described above will become critical with the server synchronization model in which many users have expressed interest. Under this model, the handheld device synchronizes directly with the Notes server, bypassing the desktop machine with the Notes client. Handheld users with modems will be able to dial in to their network, select the Notes server with the information they need, and synchronize directly with the desired server. With server synchronization, the handheld device becomes a thin client in a networked environment.

However, providing handheld devices with direct access to server raises certain security issues. One of these include the issue of where to store the Notes user IDs, which usually reside on the same machine as the Notes client. Storing the Notes ID in all the servers the user may access is cumbersome, and storing the Notes ID in the PalmPilot raises the issue of whether the handheld device should be considered a safe place to keep such important information, given how easy such small devices can be stolen. Keeping the Notes ID in the server side can be simplified by setting up a designated synchronization server, which contains the Notes IDs of all users, and which serves as a proxy through which handheld users can access other servers. If the Notes ID is kept in the server, then there also needs to be a secure mechanism to transfer the password, used to decrypt the information in the Notes ID, from the handheld device to the server. This illustrates the importance of the remote login mechanisms described earlier. This thesis only touches upon some of security issues that may arise with remote synchronization. Future implementation of the server synchronization model will require a more detailed analysis of the interaction of Notes security and handheld devices.

7. Evaluation

7.1. Notes Mail Conduit Usability Test Results

In order to get a better idea of how users felt about the released version of the Notes Mail conduit, a limited survey study was conducted with a selected number of users who have been involved with the project since its beta testing stage. The goals of this study consisted of learning how the Mail conduit was being used, how users felt about additional features such as support for multiple replica synchronization and direct server synchronization, and how well the conduit's features and performance fulfill users' needs. A copy of the survey sent to the selected users is included in Appendix C.

Fifty surveys were sent out, and the preliminary feedback results presented here reflect the information in twenty returned surveys. Due to time constraints, a more extensive user study involving a larger population was not conducted. Furthermore, most of the returned surveys came from users who worked within Lotus and IBM. A more representative survey should also include feedback from users with different backgrounds and work environments.

According to the study, users generally dealt with Mail messages smaller than 10 Kbytes (Figure 7.1), which meant users rarely had messages truncated by the conduit's 20 Kb limit imposed for messages transferred to the PalmPilot. Part of the reason why these messages were relatively small in size is the fact that only the text part of the messages were extracted during synchronization. Most users felt it was not necessary to convert the rest of the rich text of the original message, including graphics, attachments, and editable fields/buttons. Given the small and limited screen of handheld devices, it seems that users are not too concerned with information beyond the text content of messages.

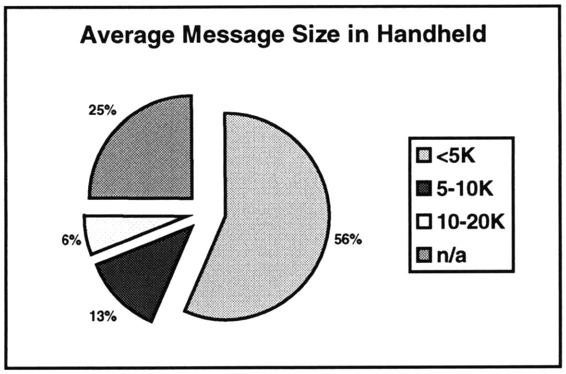


Figure 7.1 Average size of messages in PalmPilot

A majority of surveys seemed to indicate that most people used the PalmPilot as a mail reader, since very few users regularly sent out messages composed in the PalmPilot itself. Figure 7.2 illustrates the large number of messages transferred to the PalmPilot from Notes compared to the small number of messages sent out from the PalmPilot. Users usually complain that the PalmPilot's input and editing capabilities are still too cumbersome to compose long messages. This will probably change as handheld technology advances, and speech and handwriting recognition become more prevalent.

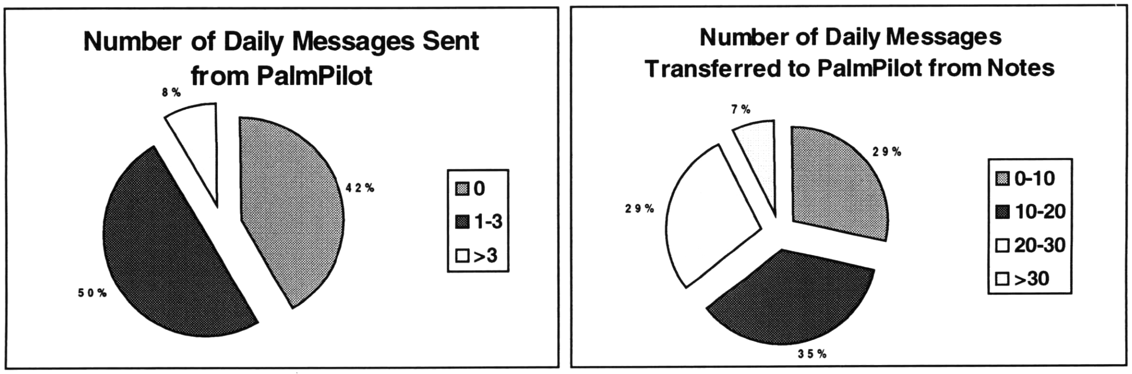


Figure 7.2 Average number of messages transferred to and sent from the PalmPilot

In terms of the data selection/manipulation options, the most popular ones amongst users were the options to synchronize messages in Notes Inbox and propagate deletions from Notes to PalmPilot (Table 7.1). This actually proved to be contrary to the earlier feedback we had obtained with the first prototype of the conduit, where most users had indicated that they preferred to synchronize messages in a custom view/folder. This could be partly because the option to only download new and unread messages from the

Notes Inbox did not work properly in the earlier versions of the prototype. This problem was fixed with the released product version. In general, it seems that the data selection options provided by the conduit fulfilled the needs of most users.

Synchronization Feature	***Usage %***
Propagate Drafts from PalmPilot to Notes	37
Propagate Drafts from Notes to PalmPilot	18
Propagate Deletions from PalmPilot to Notes	44
Propagate Deletions from Notes to PalmPilot	63
Synchronize Messages in Notes Inbox	63
Synchronize Messages in Custom View/Folder	37

Table 7.1 Percentage of users who used each of the synchronization options.

In terms of the Inbox synchronization options, it was actually surprising that a fair number of users chose to synchronize their entire Notes inbox (Table 7.2). We had originally assumed that users would have hundreds of messages in their Notes inbox, and therefore they would probably just want to synchronize a small subset of those messages. However, it seems that many users keep their Inbox fairly small by constantly deleting and filing messages appropriately. One common request from users was to add another option to synchronize only those Notes inbox messages which were a certain number of days old.

Inbox Selection Option	***Usage %***
All new unread messages since last Hotsync	39
All new messages since last Hotsync	7
All messages since a particular date	15
All messages in Inbox	39

Table 7.2 Preferred Inbox selection method for users who synchronized their Notes Inbox

Regarding the performance of the Notes Mail conduit, most users indicated that synchronization time usually took less than a couple minutes. For those users who transferred over 20 messages between PalmPilot and Notes at a time, synchronization did take longer, but it never went beyond five minutes. During the development of the conduit, we concluded that the biggest bottleneck for the conduit's performance was in the network connection between the Notes client and server, and the serial connection between the PalmPilot and the Notes client. The actual data conversion and synchronization mechanism did not contribute significantly to the synchronization time. Likewise, with the server synchronization model, the performance bottleneck will probably be in the network connection between the PalmPilot and the Notes server.

Currently, the release version of the Notes Mail conduit does not support synchronization with multiple replicas which may be out of sync with each other. Many users felt this was a serious limitation, since they often synchronize with replicas of their mail database residing in both their notebook and desktop computers. This again illustrates the need to implement the multiple replica synchronization framework outlined in section 4.3 in future versions. A majority of users also indicated that they would want to have direct remote server synchronization, but some of them did express concern about how such synchronization model would affect the security of their Notes system. Therefore, future work on server synchronization should expand on the security issues that chapter 5 of this thesis presented.

7.2. Assessment of the overall synchronization framework and future work

Both the prototype and release version of the Notes Mail conduit have allowed us to test parts of the more general synchronization framework that we have outlined in this thesis. The data selection and conflict resolutions mechanisms described in Chapter 4 are currently being used by the Notes Mail conduit, and they seem to be meeting the needs of most users, as evidenced by the results of our survey. For the Mail conduit, we have followed the proposed framework's conservative approach for resolving update/update and update/remove conflicts, where information is never lost through the creation of duplicates. However, synchronization of certain other types of databases can be made more efficient through more aggressive conflict resolution strategies, especially if there is a high rate of conflict occurrence. Both the data selection and conflict resolution strategies can be heavily dependent on the nature of the database, and therefore our proposed synchronization framework has only attempted to establish general guidelines, as opposed to set specific algorithms which would have probably become too application specific.

In terms of the data conversion process, this thesis project only experimented with extracting the plain text from Notes documents to PalmPilot records. Possible next steps include the conversion of graphics and doclinks. In Notes, a doclink consists of an active link which allows users to jump from one document to another by clicking on the link. To a certain extent, doclinks in Notes are analogous to hypertext in the World Wide Web. When synchronizing documents with doclinks to a PalmPilot, it would be useful for the conduit to also automatically synchronize the documents that the doclinks refer to, and to add the appropriate links in the PalmPilot version of the original document. However, one must be careful about trying to convert every single element in a Notes document to its handheld equivalent, since sometimes, only the plain text information is already sufficient in the limited environment of a handheld device with a small screen and limited memory. As evidenced by the user feedback for the Notes Mail conduit, most users felt that extracting just the plain text from mail messages was enough. Therefore, much like data selection and conflict resolution, the process of determining a data conversion strategy should be coupled closely with the type of database being synchronized.

The multiple replica synchronization algorithm presented in section 4.3 has not been implemented due to time constraints. A good next step for this project would consist of the development of a general purpose API which would abstract the complexities of maintaining the different tables during synchronization. Ideally, this API would enable a software developer to quickly build conduit-like applications that would synchronize any range of handheld devices with multiple Notes replicas. A good test of the versatility of such API would be to use it to rewrite the Notes Mail conduit and develop the equivalent conduit for a Microsoft Windows CE device.

One of the goals of this thesis was to address the issue of providing secure access to the database system from handheld devices. Chapter 5 described the nature of this security problem, and it presented a possible simple solution to the problem. As stated earlier, the development of a more robust security mechanism for remote handheld access went beyond the scope of this thesis. Nevertheless, future work on this area should follow up on the issues presented in Chapter 5 by developing better authentication mechanisms to ensure that only authorized users access the information in the network from their handheld devices.

Many of the design decisions in this project were directly related to one of the original requirements of preserving the existing handheld applications. As a matter of fact, much of the complexity of correctly maintaining an IMT was necessary because we chose to not modify the existing handheld applications. The virtue of such requirement becomes apparent as we apply the proposed framework to the wide range of different types of handheld devices currently available. We have not extended our work beyond the PalmPilot, but the development of the general synchronization API mentioned above should be the first steps towards showing that the high cost of re-writing the same application across different handheld devices overshadows the added complexity of developing a framework which does not require the re-development of existing applications.

8. Conclusion

As the demand for handheld-based solutions continues to grow, it is clear that there needs to be better ways to integrate handheld devices with existing information networks. Improvements in handheld technology and the growing user base of such devices have made it possible to develop and test reliable systems where computer networks and handhelds are capable of working together. This thesis project aims to provide a better understanding of the issues behind transforming handheld devices such as the PalmPilot into a seamless extension of existing distributed database system like Lotus Notes. More specifically, this thesis proposes a general framework that addresses the critical synchronization issues outlined earlier in this thesis. These issues included appropriate mechanisms for selection and conversion of data, conflict resolution, multiple replica synchronization, and secure remote synchronization. The proposed framework was shaped by the premises that it should not require the re-development of existing applications residing in the handheld device, and it should work despite handheld memory limitations, intermittent connectivity and platform differences.

Even though most of the work done in this project has involved the PalmPilot and Lotus Notes, much effort was taken to avoid design decisions which were specific to either platform choices. However, our experience has been limited to the implementation of a synchronization process involving Notes and the PalmPilot, and therefore there could still be more critical general synchronization issues that arise when allowing others types of handheld devices to connect with other types of distributed databases systems. Thus, the work in this project should be complimented by additional experiments with other systems such as Microsoft Windows CE-based handhelds and Oracle database system. Over time, the framework presented by this project will become more thorough and robust as more systems are explored.

The Notes Mail conduit described in Chapter 6 is just one example out of many other possible applications where synchronization with handheld devices can add versatility and convenience. Inventory management from the road, field data collection, and remote project management are some examples of other areas which highlight the potential markets for the integration of handheld devices into distributed database systems. The combination of the continuous improvement of handheld technology and the synchronization framework in this thesis can potentially bring handheld devices one step closer to being truly useful tools in a networked world.

Appendix A - Source code for Notes Mail Conduit prototype

```
/*
      File:       MailConduitMonitorClass.cpp
*/

#include "stdafx.h"
#include "string.h"
#include "SyncMgr.h"
#include "Condutil.h"
#include "hslog.h"
#include "MailConduitMonitorClass.h"
#include "MailRecordClass.h"
#include "resource.h"

#define YEAR_OFFSET 1904
#define NOTES_YEAR_OFFSET 1900
#define SIZE_OF_PILOT_RECORD 10000
#define SIZE_OF_TOTAL_BYTES 10000
#define MAX_BUFFER_SIZE 10000
#define INBOX_CAT_ID 0
#define OUTBOX_CAT_ID 1
#define DELETEBOX_CAT_ID 2
#define INFORMATION_CAT_ID 10
#define INFO_REC_SIZE 4
#define MAX_VIEWFOLDER_SIZE 100


#define DELETE_BIT 0X80
#define DIRTY_BIT 0X40

CMailConduitMonitor::CMailConduitMonitor (PROGRESSFN pFn,
                                          CSyncProperties& rProps):
CBaseConduitMonitor(pFn, rProps)
{
      logfile.open("mailcond.log", ios::out, 0);
      logfile<<"Beginning to log the sync process"<<endl<<endl;
      logfile<<"CSyncProperties:"<<endl;
      logfile<<"m_PathName: "<<m_rSyncProperties.m_PathName<< endl;
      logfile<<"m_LocalName: "<<m_rSyncProperties.m_LocalName<< endl;
      logfile<<"m_UserName: "<<m_rSyncProperties.m_UserName<< endl;
      logfile<<"m_RemoteName: "<<m_rSyncProperties.m_RemoteName<< endl;
      logfile<<"m_nRemoteCount: "<<m_rSyncProperties.m_nRemoteCount<<
endl << endl;

      HKEY software,usrobotics,pilotdesktop;
      RegOpenKeyEx(HKEY_CURRENT_USER,"Software",0,KEY_ALL_ACCESS,&softw
are);
      RegOpenKeyEx(software,"U.S.
Robotics",0,KEY_ALL_ACCESS,&usrobotics);
```

```
      RegOpenKeyEx(usrobotics,"Pilot
Desktop",0,KEY_ALL_ACCESS,&pilotdesktop);
      RegOpenKeyEx(pilotdesktop,"ApplicationMail",0,KEY_ALL_ACCESS,&app
licationmail);
}

CMailConduitMonitor::~CMailConduitMonitor()
{
}

long CMailConduitMonitor::StartSync()
{
      long retval;

      CONDHANDLE conduitHandle = (CONDHANDLE)0;
      Activity syncFinishCode = slSyncFinished;

      logfile << "-> Registering the Conduit"  << endl;

      // Registering the conduit w/ SyncMgr.DLL for communication w/ HH
      if (retval = SyncRegisterConduit(conduitHandle))
            return(retval);

      LogAddEntry("",slSyncStarted,FALSE);

      logfile << "-> Obtaining Remote Table"     << endl;
      retval= ObtainRemoteTable();
      if (retval != CONDERR_NONE)
            return retval;

      logfile << "-> Starting Fast Sync process" << endl;
      retval = DoMailSync();

      if (!IsCommsError(retval))
      {
            SyncResetSyncFlags(m_RemHandle);
            SyncCloseDB(m_RemHandle);
      }

      if (retval)
            syncFinishCode = slSyncAborted;

      LogAddEntry("NotesMail",syncFinishCode,FALSE);

      if(!IsCommsError(retval))
            SyncUnRegisterConduit(conduitHandle);

      return (retval);
}


void CMailConduitMonitor::LogApplicationName(char* appName, WORD len)
{
      // Load string from the resource file.
      ::LoadString(m_DllInstance, IDS_COND_NAME, appName, len);
}
```

```
long CMailConduitMonitor::ObtainRemoteTable(void)
{
      long retval = CONDERR_BAD_REMOTE_TABLES;

      // Opening the Mail Database on the Pilot side, and putting a
handle
      // to the Mail DB into the m_hRemoteDB variable.

      retval = SyncOpenDB ( m_rSyncProperties.m_RemoteName[0], 0,
m_RemHandle,eDbWrite|eDbRead);
      if (retval == SYNCERR_NONE)
      {
            logfile << "--> Successfully opened remote database" <<
endl;
            logfile << "--> Error Code Returned:  " << retval << endl
<< endl;
      }
      return (retval);
}


long CMailConduitMonitor::DoMailSync()
{
      long err = SYNCERR_NONE;
      long lRet = SYNCERR_NONE;
      int temp_result;

      CRawRecordInfo pilotRawRec;
      CRawRecordInfo InfoRec;
      CMailRecord pilotMailRec;

      // the sync_date_buffer keeps the current day/time of this sync
session.
      // if everything goes succefully, we will update the info rec of
the
      // mail database w/ this sync_date_buffer, thus saving the
day/time of
      // the last sync session
      BYTE *sync_date_buffer = (BYTE *)
malloc(sizeof(BYTE)*INFO_REC_SIZE);
      LNSetThrowAllErrors(TRUE);

      // In this current Alpha 2.0 version, we are not supporting
deletes,
      // so it is ok to just purge all deleted mail in the pilot.
      SyncPurgeDeletedRecs(m_RemHandle);


      // Finding the information record that is being used to store the
date of the
      // last sync session where the Mail was sync'ed.
      // For the time being, we distinguish the information record by
assigning it the
      // unique category number 10.

      try
      {
```

```
      // Opening Notes Session
      logfile << "-> Opening Notes Session" << endl;
      notesSession.Init();

      // the last_sync_date contains the same info contained in
sync_date_buffer,
      // except in Notes format, making it easier to compare
later.
      // Initializing the last_sync_date like this was sort of
      // a temp fix, since I was having problems setting the
      // day/time/month/year for a LNDatetime object that was
      // initialized w/ no arguments.
      LNDatetime last_sync_date("07/05/97 23:00:00");

      if (! (GetAndSetDateTime(sync_date_buffer, last_sync_date,
InfoRec)))
      {
         notesSession.Term();
         return -1;
      }

      // Getting and opening the Notes Mail database
      logfile << "-> Opening Notes Mail Database" << endl;
      notesSession.GetMailDatabase(&notesMailDB);
      notesMailDB.Open();

      // setting up the raw record that will collect the info
from the
      // remote record
      memset(&pilotRawRec, 0, sizeof(pilotRawRec));
      pilotRawRec.m_FileHandle = m_RemHandle;
      pilotRawRec.m_RecId = 0;
      pilotRawRec.m_RecIndex = 0;
      pilotRawRec.m_pBytes = (BYTE *)new char
[SIZE_OF_PILOT_RECORD];
      pilotRawRec.m_TotalBytes = SIZE_OF_TOTAL_BYTES;

      // Begin looping over the Pilot records that have been
modified.
      // These may include new messages that were composed,
messages that
      // were deleted, and the filed messages.

      while (err == SYNCERR_NONE)
      {
         logfile << "-> Getting the next modified record in
the Pilot" << endl;
         if ( (err = SyncReadNextModifiedRec(pilotRawRec) ) !=
SYNCERR_NONE )
         {
            break;
         }
         // logging the information about the modified record
being read
         logfile << "Reading Modified Remote Records" << endl;
         logfile << "Remote Record Prop.: " << endl;
         logfile << "RecID: " << pilotRawRec.m_RecId << endl;
```

```
                  logfile << "CatID:  " << pilotRawRec.m_CatId << endl;
                  logfile << "RecSize:  " << pilotRawRec.m_RecSize <<
endl;
                  logfile << "Attrib:  " << pilotRawRec.m_Attribs <<
endl;
                  logfile << "Body: " << pilotRawRec.m_pBytes << endl;

                  if ( pilotRawRec.m_CatId == OUTBOX_CAT_ID)
                  {
                        logfile << "-> Coverting the raw Pilot record
into PC format" << endl;
                        ConvertOutboxRecord (pilotMailRec,
pilotRawRec);
                        //logfile << "-> To: " <<
pilotMailRec.m_szToField << endl;
                        //logfile << "-> CC: " <<
pilotMailRec.m_szCCField << endl;
                        //logfile << "-> Subj: " <<
pilotMailRec.m_szSubjField << endl;
                        //logfile << "-> Body: " <<
pilotMailRec.m_szBodyField << endl;
                        logfile << "-----------------------------------
-" << endl << endl;

                        LNMailMessage notesMessage;
                        LNRichText bodyField;
                        LNRTCursor bodyCursor;

                        // creating a new notes mail message, and
populating the fieds using the
                        // information gathered from the pilot record

                        notesSession.CreateMailMessage(&notesMessage);

      notesMessage.SetRecipients(pilotMailRec.m_szToField);

                        if (pilotMailRec.m_szCCField[0]!=NULL)

      notesMessage.SetCC(pilotMailRec.m_szCCField);

                        if (pilotMailRec.m_szBCCField[0]!=NULL)

      notesMessage.SetBCC(pilotMailRec.m_szBCCField);

                        if (pilotMailRec.m_szSubjField[0]!=NULL)

      notesMessage.SetSubject(pilotMailRec.m_szSubjField);

                        if ( !
(notesMessage.GetItem("Body",&bodyField)))

      bodyField.Append(pilotMailRec.m_szBodyField);

                        notesMessage.Send();

                        //Obtaining the SentSaveOption from the
Registry, and determining whether
```

```
                        //to save the sent document or not.
                        LPBYTE SentSaveOption= (LPBYTE )
malloc(sizeof(char) + 1);
                        LPDWORD buffer_size = (LPDWORD)
malloc(sizeof(DWORD) + 1);
                        *buffer_size = 2;

      RegQueryValueEx(applicationmail,"SentSaveOption",NULL,NULL,SentSa
veOption,buffer_size);

                        // if the user chooses save every sent message,
then we just tell notes to
                        // save the message.
                        if ( !memcmp(SentSaveOption,"0",1) )
                              notesMessage.Save();

                        else if (!memcmp(SentSaveOption,"1",1))
                        {
                              // if the user chooses to prompt before
saving each sent message
                              CString mesg = "Would you like to save
the message to ";
                              mesg += pilotMailRec.m_szToField;
                              mesg += " regarding ";
                              mesg += pilotMailRec.m_szSubjField;
                              mesg += " in your Notes Mail Database?";
                              int save_result = MessageBox(NULL,mesg,
"NotesMail Conduit", MB_ICONQUESTION | MB_YESNO);
                              if (save_result == IDYES)
                                    notesMessage.Save();
                        }
                        else if (!memcmp(SentSaveOption,"2",1))
                              continue;
                        else
                              notesMessage.Save();

                        err = SyncDeleteRec(pilotRawRec);
                  }
            }

            logfile << "-> Scanning and Transferring Mail from Notes to
Pilot" << endl << endl;

            if (err && err != SYNCERR_FILE_NOT_FOUND)
                  lRet == err;
            else
            {
                  // The Code below takes care of downloading messages
from Notes to Pilot

                  // These are the notes variables needed to query the
messages inside the Notes
                  // Mail database.

                  LNViewFolder notesInbox;
                  LNVFEntry notesCurrentEntry;
                  LNMailMessage notesCurrentDoc;
                  // picking up the desired mode of downloading the
inbox from the registry
                  LPBYTE InboxOption= (LPBYTE ) malloc(sizeof(char) +
1);
                  LPDWORD temp_buffer_size = (LPDWORD)
malloc(sizeof(DWORD) + 1);
                  *temp_buffer_size = 2;

      RegQueryValueEx(applicationmail,"InboxOption",NULL,NULL,InboxOpti
on,
                        temp_buffer_size);

                  LPBYTE DownloadOption= (LPBYTE ) malloc(sizeof(char)
+ 1);
                  *temp_buffer_size = 2;

      RegQueryValueEx(applicationmail,"DownloadOption",NULL,NULL,Downlo
adOption,
                        temp_buffer_size);

                  LPBYTE ClearInboxOption= (LPBYTE )
malloc(sizeof(char) + 1);
                  *temp_buffer_size = 2;

      RegQueryValueEx(applicationmail,"ClearInboxOption",NULL,NULL,Clea
rInboxOption,
                        temp_buffer_size);

                  // Clearing the Pilot Inbox if the user chose to do
so.
                  if ( !memcmp(ClearInboxOption,"1",1) )
                  {
                        WORD record_count=0;
                        SyncGetDBRecordCount(m_RemHandle,
record_count);
                        logfile << endl << "--> DELETING INBOX RECORDS:
" << endl;

                        // clearing the Pilot inbox
                        for (int index = record_count-1; index >= 0;
index--)
                        {
                              CRawRecordInfo tempRawRec;
                              memset(&tempRawRec,0,sizeof(tempRawRec));
                              tempRawRec.m_FileHandle = m_RemHandle;
                              tempRawRec.m_RecIndex = index;
                              SyncReadRecordByIndex(tempRawRec);
                              if (tempRawRec.m_CatId == INBOX_CAT_ID)
                                    SyncDeleteRec(tempRawRec);
                        }
                  }


                  if ( !memcmp(DownloadOption,"0",1) )
                  {
                        // this is the case when the user wants to
download messages
                        // from the NotesMail inbox.  We first open the
($Inbox) view,
                        // and go to its first entry.


      notesMailDB.GetViewFolder("($Inbox)",&notesInbox);  // getting
the Notes Mail Inbox view
                        notesInbox.Open();            // opening the
Notes Mail Inbox view
                        notesInbox.GotoFirst(&notesCurrentEntry);
                        if (notesCurrentEntry.IsCategory())

      notesInbox.GotoNextNonCategory(&notesCurrentEntry);

                        if ( (!memcmp(InboxOption,"0",1)) ||
(!memcmp(InboxOption,"1",1)) )
                        {

                              // Creating the list of mail received
since last sync session
                              CString formula = "Select (DeliveredDate
> [";
                              formula += (LNString) last_sync_date;
                              //formula += "]) & (Form = \"Memo\" |
Form = \"Reply\")";
                              formula += "])";
                              LNDocumentArray selected_docs;
                              LNSearchOptions search_opts;
                              search_opts.SetBeginDate(last_sync_date);
                              LNFormula selectionFormula(formula);
                              notesMailDB.Search(selectionFormula,
&selected_docs,&search_opts);
                              int selected_count =
selected_docs.GetCount();
                              logfile << "COUNT!!!! = " <<
selected_docs.GetCount() << endl;

                              if ( (!memcmp(InboxOption,"0",1)))
                              {
                                    // Creating the list of unread
messages
                                    LNDocumentArray unread_docs;

      notesMailDB.GetUnreadList(&unread_docs,0);

                                    for (int z = 0; z <
selected_count;z++)
                                    {
                                          notesCurrentDoc =
selected_docs[z];
                                          notesCurrentDoc.Open();
                                          if
(unread_docs.Contains(notesCurrentDoc.GetNoteID()))
                                          {
                                                temp_result =
ConvertInboxRecord(notesCurrentDoc,pilotRawRec);
                                                if (temp_result)

      SyncWriteRec(pilotRawRec);
                                          }
                                          notesCurrentDoc.Close();
                                    }
                              }
                              else if ( (!memcmp(InboxOption,"1",1)))
                              {
                                    for (int z = 0; z <
selected_count;z++)
                                    {
                                          notesCurrentDoc =
selected_docs[z];
                                          notesCurrentDoc.Open();
                                          temp_result =
ConvertInboxRecord(notesCurrentDoc,pilotRawRec);
                                          if (temp_result)

      SyncWriteRec(pilotRawRec);
                                          notesCurrentDoc.Close();
                                    }
                              }

                        }

                        else if (!memcmp(InboxOption,"2",1) )
                        {
                              // this is the case when the user want to
download all message
                              // into the inbox, replacing the old
Pilot inbox.

                              WORD record_count=0;
                              SyncGetDBRecordCount(m_RemHandle,
record_count);
                              logfile << endl << "--> DELETING INBOX
RECORDS: " << record_count << endl;

                              // clearing the Pilot inbox
                              for (int index = record_count-1; index >=
0; index--)
                              {
                                    CRawRecordInfo tempRawRec;

      memset(&tempRawRec,0,sizeof(tempRawRec));
                                    tempRawRec.m_FileHandle =
m_RemHandle;
                                    tempRawRec.m_RecIndex = index;
                                    SyncReadRecordByIndex(tempRawRec);
                                    if (tempRawRec.m_CatId ==
INBOX_CAT_ID)
                                          SyncDeleteRec(tempRawRec);
                              }
                              do
                              {

      notesCurrentEntry.GetDocument(&notesCurrentDoc);
                                    notesCurrentDoc.Open();
                                    temp_result =
ConvertInboxRecord(notesCurrentDoc, pilotRawRec);     // converting the
doc into Pilot format
                                    if (temp_result)
                                          SyncWriteRec(pilotRawRec);
                                    // writing the converted doc into
Pilot
                                    notesCurrentDoc.Close();
                              }
                              while (
!(notesInbox.GotoNextNonCategory(&notesCurrentEntry)));
                        }
                  }

                  else if ( !memcmp(DownloadOption,"1",1) )
                  // this is the download option where the user
specifies the folder
                  // from which he/she/it wants to download the
messages from.
                  {

                        // obtaining the view/folder name from the
registry
                        LPDWORD buffer_size2 = (LPDWORD)
malloc(sizeof(DWORD));
                        *buffer_size2 = MAX_VIEWFOLDER_SIZE;
                        LPBYTE ViewFolderName = (LPBYTE)
malloc(sizeof(char) * MAX_VIEWFOLDER_SIZE);

      RegQueryValueEx(applicationmail,"ViewFolderName",NULL,NULL,ViewFo
lderName,buffer_size2);
                        char *temp = (char *) malloc(sizeof(char) *
(*buffer_size2));
                        memcpy (temp,ViewFolderName,*buffer_size2);

                        try
                        {

      notesMailDB.GetViewFolder(temp,&notesInbox);
                        }
                        catch (...)
                        {
                              CString mesg = "Unable to read from
view/folder ";
                              mesg += temp;
                              mesg += ".  Please make sure that it is a
valid view/folder.";
                              MessageBox(NULL,mesg,"NotesMail Conduit
Error!",MB_OK|MB_ICONEXCLAMATION);
                              LogAddEntry(mesg,slWarning,FALSE);
                              notesSession.Term();
                              return -1;
                        }

                        notesInbox.Open();
                        notesInbox.GotoFirst(&notesCurrentEntry);
                        do
                        {

        notesCurrentEntry.GetDocument(&notesCurrentDoc);
                                notesCurrentDoc.Open();
                                //logfile << "--> Coverting Doc" << endl;
                                temp_result =
ConvertInboxRecord(notesCurrentDoc, pilotRawRec);
                                if (temp_result)
                                        SyncWriteRec(pilotRawRec);
                                notesCurrentDoc.Close();
                        }
                        while (
!(notesInbox.GotoNext(&notesCurrentEntry)));
                }
            }
        }
        catch(LNSTATUS error)
        {
            char errorBuf[1000];
            LNGetErrorMessage(error, errorBuf);
            CString mesg = "!!!! Lotus Notes Error:  ";
            mesg += errorBuf;
            logfile << mesg << endl;
            LogAddEntry(mesg,slWarning,FALSE);
            notesSession.Term();
            return -1;
        }

        // Updating the current sync time in the Info Record
        InfoRec.m_pBytes = sync_date_buffer;
        InfoRec.m_RecSize = INFO_REC_SIZE;
        InfoRec.m_TotalBytes = INFO_REC_SIZE;
        InfoRec.m_CatId = 10;
        SyncWriteRec(InfoRec);

        notesSession.Term();  // Terminating Notes Session
        return lRet;
}

void CMailConduitMonitor::ConvertDeleteRecord(CMailRecord &toRecord,
CRawRecordInfo &fromRecord)
{
        int pos = 0;
        int count = 0;
        while (count != 9)
        {
            if (fromRecord.m_pBytes[pos]=='\0')
                count++;
            pos++;
        }
        ReadByteStream(pos, toRecord.docUNID, fromRecord.m_pBytes);
        return;
```

```
}


void CMailConduitMonitor::ConvertOutboxRecord (CMailRecord &toRecord,
CRawRecordInfo &fromRecord)
{
      toRecord.m_RecId = fromRecord.m_RecId;
      logfile << "--> Setting the Rec ID: " << toRecord.m_RecId <<
endl;

      logfile << "--> Setting the Cat ID" << endl;
      toRecord.m_CatId = fromRecord.m_CatId;

      logfile << "--> Setting the Status Bits" << endl;
      toRecord.SetStatus(fldStatusNONE);
      if (fromRecord.m_Attribs & DELETE_BIT)
            toRecord.SetStatus(fldStatusDELETE);
      else if (fromRecord.m_Attribs & DIRTY_BIT)
            toRecord.SetStatus(fldStatusUPDATE);

      if ( !(fromRecord.m_Attribs&DELETE_BIT) )
      {
            // Setting the delivery type
            toRecord.m_DeliveryType = fromRecord.m_pBytes[4];
            int pos = 6;

            // Setting the Subject field
            pos = ReadByteStream (pos, toRecord.m_szSubjField,
fromRecord.m_pBytes);
            pos++;

            // Setting the To Field
            pos = ReadByteStream (pos, toRecord.m_szToField,
fromRecord.m_pBytes);

            // Setting the CC Field
            pos = ReadByteStream (pos, toRecord.m_szCCField,
fromRecord.m_pBytes);

            // Setting the BCC Field
            pos = ReadByteStream (pos, toRecord.m_szBCCField,
fromRecord.m_pBytes);
            pos= pos+2;

            // Setting the Body Field
            pos = ReadByteStream (pos, toRecord.m_szBodyField,
fromRecord.m_pBytes);


      }
}



int CMailConduitMonitor::ConvertInboxRecord (LNMailMessage &doc,
CRawRecordInfo &toRecord)
{
```

```
int pos=0;
int count,q;
LNText temp_text;
LNString temp_string;
LNString recipients = "";
LNString cc = "";
LNSTATUS result;
BYTE *buffer = (BYTE *) calloc(MAX_BUFFER_SIZE,sizeof(BYTE));
BYTE *buf_pointer = buffer;
try
{
      // filling in the header w/ the date information

      LNDatetime notesDeliveredDate;
      LNDatetime notesPostedDate;
      LNDatetime notesDate;
      LNDatetimes temp_field;
      doc.GetItem("DeliveredDate",&temp_field);
      notesDeliveredDate = temp_field[0];
      doc.GetDateSent(&notesPostedDate);

      if (notesDeliveredDate.IsNull())
            notesDate = notesPostedDate;
      else
            notesDate = notesDeliveredDate;

      int day,month,year,hour,minute;
      day = notesDate.GetDay();
      month = notesDate.GetMonth();
      year = notesDate.GetYear() - YEAR_OFFSET;
      hour = notesDate.GetHour();
      minute = notesDate.GetMinute();

      //Writing the year/month/date
      WORD wTemp = 0;
      wTemp = (year << 9) | (month << 5) | day;
      WriteByteStream(buf_pointer,(WORD) wTemp,pos);

      //Writing the time
      buf_pointer += 2;
      wTemp = 0;
      wTemp = (minute << 8) | hour;
      memcpy(buf_pointer,&wTemp,2);
      buf_pointer += 2;

      *buf_pointer++=0x54;
      *buf_pointer++='\0';


      result = doc.GetSubject(&temp_text);
      if (!result)
      {
            temp_string = temp_text[0];
            count = strlen(temp_string);
            //logfile << "Subject: " << temp_string << endl;
            memcpy (buf_pointer,temp_string,count);
            buf_pointer += count;
```

```
}
*buf_pointer++ = '\0';

result = doc.GetSender(&temp_string);
if (!result)
{
      ConvertName(temp_string);
      count = strlen(temp_string);
      //logfile << "From: " << temp_string << endl;
      memcpy (buf_pointer, temp_string, count);
      buf_pointer += count;
}
*buf_pointer++ = '\0';

result = doc.GetRecipients(&temp_text);
if (!result)
{
      count = temp_text.GetCount();
      for(q=0;q<count;q++)
      {
            temp_string = temp_text[q];
            ConvertName(temp_string);
            recipients.Append(temp_string);
            if(q!=(count-1))
                  recipients.Append(", ");
      }
      count = strlen(recipients);
      //logfile << "To: " << recipients << endl;
      memcpy (buf_pointer, recipients, count);
      buf_pointer += count;
}
*buf_pointer++ = '\0';

result = doc.GetCC(&temp_text);
if(!result)
{
      count = temp_text.GetCount();
      for(q=0;q<count;q++)
      {
            temp_string = temp_text[q];
            ConvertName(temp_string);
            cc.Append(temp_string);
            if(q!=(count-1))
                  cc.Append(", ");
      }
      count = strlen(cc);
      //logfile << "CC: "<< temp_string << endl;
      memcpy(buf_pointer, cc,count);
      buf_pointer += count;
}
*buf_pointer++ = '\0';
*buf_pointer++ = '\0';
*buf_pointer++ = '\0';

result = doc.GetRecipients(&temp_text);
if (!result)
{
```

```
            count = strlen(recipients);
            memcpy (buf_pointer, recipients, count);
            buf_pointer += count;
        }
        *buf_pointer++ = '\0';

        LNRichText body;
        LNString body_LN_text;
        result = doc.GetBody(&body);
        if (!result)
        {
            body.GetText(&body_LN_text);
            count = strlen(body_LN_text);
            if (count > (MAX_BUFFER_SIZE - (buf_pointer -
buffer)))
            {
                temp_string = "<This message was
truncated!>\n\n";
                count = (MAX_BUFFER_SIZE - (buf_pointer -
buffer) - 40);

    memcpy(buf_pointer,temp_string,strlen(temp_string));
                buf_pointer += strlen(temp_string);
            }

            memcpy(buf_pointer,body_LN_text,count);
            //logfile << "Body:  " << buf_pointer << endl;
            buf_pointer += count;
        }
        else
        {
            temp_string = "<Unable to get body of message!>";
            count = strlen(temp_string);
            memcpy(buf_pointer,temp_string,count);
            buf_pointer += count;
        }
        *buf_pointer++ = '\0';

        pos = buf_pointer - buffer;
        logfile << "Message size: " << pos << endl;
        logfile << endl << endl;

        toRecord.m_pBytes = (BYTE *) malloc(pos);
        toRecord.m_RecSize = pos;
        toRecord.m_TotalBytes = pos;
        toRecord.m_RecId = 0;

        for (int j = 0; j < pos;j++)
            toRecord.m_pBytes[j] = buffer[j];
    }
    catch(LNSTATUS error)
    {
        char errorBuf[1000];
        LNGetErrorMessage(error, errorBuf);
        logfile << "!!!! Lotus Notes Error while Converting Doc:  "
<< errorBuf << endl;
        result = doc.GetSender(&temp_string);
```

```
            LNString mesg = "  --> Unable to download the message from
";

            if (!result)
            {
                  ConvertName(temp_string);
                  mesg += temp_string;
            }

            result = doc.GetSubject(&temp_text);
            if (!result)
            {
                  temp_string = temp_text[0];
                  mesg += " regarding ";
                  mesg += temp_string;
            }

            mesg += ".";
            LogAddEntry(mesg,slWarning,FALSE);
            return 0;
      }

      return 1;
}

void CMailConduitMonitor::ConvertName(LNString &name)
{
      unsigned long *dummy = (unsigned long *) calloc(1,sizeof(unsigned
long));
      int total = name.GetCharacterCount();
      if (name.Find("CN=",0,dummy))
            return;
      else
      {
            name.Delete(0,3);
            name.FindChar('/',0,dummy);
            name.Delete(*dummy, total-*dummy-3);
      }
      return;
}


void CMailConduitMonitor::GetCurrentSystemDateTime(BYTE *buffer)
{
      CTime sync_date = CTime::GetCurrentTime();
      WORD day,month,year,hour,minute;
      day = sync_date.GetDay();
      month = sync_date.GetMonth();
      year = sync_date.GetYear() - NOTES_YEAR_OFFSET;
      hour = sync_date.GetHour();
      minute = sync_date.GetMinute();


      logfile << "Setting today's date" << endl;
      logfile << month << "/" << day << "/" << year << "   " << hour <<
":" << minute << endl;
      //Writing the year/month/date
```

```
        WORD wTemp = 0;
        wTemp = (year << 9) | (month << 5) | day;
        memcpy(buffer,&wTemp,2);

        //Writing the time
        wTemp = 0;
        wTemp = (hour << 8) | minute;
        memcpy(buffer+2,&wTemp,2);
        return;
}


int CMailConduitMonitor::GetAndSetDateTime(BYTE *sync_date_buffer,
LNDatetime &last_sync_date, CRawRecordInfo &InfoRec)
{

        try
        {
                LNDatetime current_time =
notesSession.GetCurrentDatetime();
                WORD day,month,year,hour,minute;
                day = (WORD) current_time.GetDay();
                month = (WORD) current_time.GetMonth();
                year = (WORD) current_time.GetYear() - NOTES_YEAR_OFFSET;
                hour = (WORD) current_time.GetHour();
                minute = (WORD) current_time.GetMinute();

                logfile << "Setting today's date and time: ";
                logfile << month << "/" << day << "/" << year << "  " <<
hour << ":" << minute << endl;
                //Writing the year/month/date
                WORD wTemp = 0;
                wTemp = (year << 9) | (month << 5) | day;
                memcpy(sync_date_buffer,&wTemp,2);

                //Writing the time
                wTemp = 0;
                wTemp = (hour << 8) | minute;
                memcpy(sync_date_buffer+2,&wTemp,2);
        }
        catch(LNSTATUS error)
        {
                // if we can't get the time from Notes, then we'll just get
the time
                // from the system
                GetCurrentSystemDateTime(sync_date_buffer);
                char errorBuf[1000];
                LNGetErrorMessage(error, errorBuf);
                CString mesg = "!!!! Unable to get current time from Notes.
Using system time. ";
                mesg += "Lotus Notes Error:  ";
                mesg += errorBuf;
                logfile << mesg << endl << endl;
        }

        WORD temp_record_count=0;
        SyncGetDBRecordCount(m_RemHandle, temp_record_count);
```

```
      logfile << "Total Record Count = " << temp_record_count << endl;

      for (int temp = 0; temp < temp_record_count; temp++)
      {
            memset(&InfoRec,0,sizeof(InfoRec));
            InfoRec.m_FileHandle = m_RemHandle;
            InfoRec.m_pBytes = (BYTE *)new char [INFO_REC_SIZE];
            InfoRec.m_TotalBytes = INFO_REC_SIZE;
            SyncReadRecordByIndex(InfoRec);
            if (InfoRec.m_CatId == INFORMATION_CAT_ID)
                  break;
      }

            // if we can't find a record w/ catId = 10, then it means
that we haven't created one yet.
            // if this is the case, then we create one.
      if (temp == temp_record_count)
      {
            InfoRec.m_pBytes = sync_date_buffer;
            InfoRec.m_RecSize = INFO_REC_SIZE;
            InfoRec.m_TotalBytes = INFO_REC_SIZE;
            InfoRec.m_CatId = 10;
            InfoRec.m_RecId = 0;
            SyncWriteRec(InfoRec);
      }

      // if the info record does exist, then we extract the date/time
info out of it, and
      // put the info in the last_sync_date variable.

      WORD *year_month_day,*hour_minute;
      year_month_day = (WORD *)calloc(1,sizeof(WORD));
      hour_minute = (WORD *)calloc(1,sizeof(WORD));
      memcpy(year_month_day,InfoRec.m_pBytes,2);
      memcpy(hour_minute,InfoRec.m_pBytes+2,2);

      try
      {
            last_sync_date.SetDay( (int) (*year_month_day & 0x01F ) );
            last_sync_date.SetMonth( (int) (*year_month_day >> 5) &
0x0F);
            last_sync_date.SetYear( (int) ((*year_month_day >> 9) &
0x7F) + NOTES_YEAR_OFFSET) ;
            last_sync_date.SetHour ( (int) *hour_minute >> 8);
            last_sync_date.SetMinute(  (int) *hour_minute & 0x0FF);
      }
      catch (...)
      {
            // the code gets here if for some reason, the special
record
            // in the pilot became corrupted.  In that case, we just
rewrite
            // it w/ the last hotsync date set to the current date.

            InfoRec.m_pBytes = sync_date_buffer;
            InfoRec.m_RecSize = INFO_REC_SIZE;
            InfoRec.m_TotalBytes = INFO_REC_SIZE;
            InfoRec.m_CatId = 10;
            SyncWriteRec(InfoRec);


            WORD *year_month_day,*hour_minute;
            year_month_day = (WORD *)malloc(sizeof(WORD));
            hour_minute = (WORD *)malloc(sizeof(WORD));
            memcpy(year_month_day,InfoRec.m_pBytes,2);
            memcpy(hour_minute,InfoRec.m_pBytes + 2,2);
            last_sync_date.SetDay( (int) (*year_month_day & 0x01F ) );
            last_sync_date.SetMonth( (int) (*year_month_day >> 5) &
0x0F);
            last_sync_date.SetYear( (int) (*year_month_day >> 9) +
NOTES_YEAR_OFFSET) ;
            last_sync_date.SetHour ( (int) *hour_minute >> 8);
            last_sync_date.SetMinute(  (int) *hour_minute & 0x0FF);

            CString warning = "Warning!  The special record that keeps
the";
            warning += " last mail hotsync date in the pilot was
corrupted! ";
            warning += "The problem has been fixed, and next time you
hotsync, ";
            warning += "everything should return to normal.";
            LogAddEntry(warning,slWarning,FALSE);
      }
      LNString date = last_sync_date;
      const char *test = date;
      logfile << "Last Sync Date: " << test << endl;

      // doing a sanity check... useful b/c the old way of reading the
date/time
      // actually gives 1977 for 1997...
      if (last_sync_date.GetYear()<1997)
      {
            InfoRec.m_pBytes = sync_date_buffer;
            InfoRec.m_RecSize = INFO_REC_SIZE;
            InfoRec.m_TotalBytes = INFO_REC_SIZE;
            InfoRec.m_CatId = 10;
            SyncWriteRec(InfoRec);

            CString warning = "Warning!  The special record that keeps
the";
            warning += " last mail hotsync date in the pilot is
outdated! ";
            warning += "The problem has been fixed, and next time you
hotsync, ";
            warning += "everything should return to normal.";
            LogAddEntry(warning,slWarning,FALSE);
            return 0;

      }
      return 1;
}
```

```
/*
      File:       MailConduitMonitorClass.h
*/
#include "basemon.h"
#include "fstream.h"
#include <lncppapi.h>
#include "winreg.h"
class CMailRecord;
//
//  Error codes for the Conduit DLL's range  0x5000 - 0x5FFF
//
#define CONDERR_NONE                             0x0000
#define CONDERR_FIRST                            0x1000
#define CONDERR_BAD_REMOTE_TABLES                CONDERR_FIRST + 4
#define CONDERR_BAD_LOCAL_TABLES                 CONDERR_FIRST + 5
#define CONDERR_BAD_LOCAL_BACKUP                 CONDERR_FIRST + 6
#define CONDERR_ADD_LOCAL_RECORD                 CONDERR_FIRST + 7
#define CONDERR_ADD_REMOTE_RECORD                CONDERR_FIRST + 8
#define CONDERR_CHANGE_REMOTE_RECORD             CONDERR_FIRST + 9
#define CONDERR_RAW_RECORD_ALLOCATE              CONDERR_FIRST + 0x0A
#define CONDERR_REMOTE_CHANGES_NOT_SENT          CONDERR_FIRST + 0x0B
#define CONDERR_LOCAL_MEMORY_ALLOC_FAILED        CONDERR_FIRST + 0x0C
//#define CONDERR_BAD_SYNC_TYPE                    CONDERR_FIRST + 0x1D
#define CONDERR_UNSUP_TYPE                             CONDERR_FIRST + 0x1E

extern "C"
{
      typedef  long (*PROGRESSFN) (char*);
};

class CMailConduitMonitor : public CBaseConduitMonitor
{
public:
      CMailConduitMonitor     ( PROGRESSFN pFn, CSyncProperties
&rProps);
      ~CMailConduitMonitor();
      long StartSync();
      long ObtainRemoteTable();
      long DoMailSync();
      void LogApplicationName(char* appName, WORD len);
      void ConvertOutboxRecord (CMailRecord &a, CRawRecordInfo &b);
      void ConvertDeleteRecord (CMailRecord &a, CRawRecordInfo &b);
      int  ConvertInboxRecord (LNMailMessage &doc, CRawRecordInfo
&toRecord);
      void ConvertName(LNString &name);
      void GetCurrentSystemDateTime(BYTE *buffer);
      int GetAndSetDateTime(BYTE *sync_date_buffer, LNDatetime
&last_sync_date, CRawRecordInfo &rawRecord);

protected:
      fstream logfile;
      HKEY applicationmail;
      LNNotesSession notesSession;  // Create Notes Session
      LNDatabase notesMailDB;                   // Create Notes DB
Object

};
```

```
/*
      File:       MailRecordClass.cpp
*/

#include "stdafx.h"
#include "MailRecordClass.h"

CMailRecord::CMailRecord( int nRecId, WORD wStatus, WORD CatId, WORD
DeliveryType,
      const char *szToField, const char *szCCField, const char
*szBCCField,
      const char *szSubjField, const char *szBodyField, UNID id)

{
      strcpy( m_szToField, szToField );
      strcpy( m_szCCField, szCCField );
      strcpy( m_szBCCField, szBCCField );
      strcpy( m_szSubjField, szSubjField );
      strcpy( m_szBodyField, szBodyField );
      m_RecId = nRecId;
      m_CatId = CatId;
      m_wStatus= wStatus;
      m_DeliveryType = DeliveryType;
      docUNID = id;
}

CMailRecord::CMailRecord()
{
      m_RecId = 0;
      m_CatId = 0;
      m_wStatus = fldStatusNONE;
}

CMailRecord::CMailRecord( CMailRecord &rRec )
{
      *this = rRec;
}

CMailRecord &CMailRecord::operator=( CMailRecord &srcRec )
{

      strcpy( m_szToField, srcRec.m_szToField );
      strcpy( m_szCCField, srcRec.m_szCCField );
      strcpy( m_szBCCField, srcRec.m_szBCCField );
      strcpy( m_szSubjField, srcRec.m_szSubjField );
      strcpy( m_szBodyField, srcRec.m_szBodyField );
      m_RecId = srcRec.m_RecId;
      m_wStatus= srcRec.m_wStatus;
      m_CatId = srcRec.m_CatId;
      m_DeliveryType = srcRec.m_DeliveryType;
      docUNID = srcRec.docUNID;
      return *this;
}
```

```
/*
      File:       MailRecordClass.h
*/

#include <iostream.h>
#include <lncppapi.h>
#define MAX_TO_LEN 50
#define MAX_CC_LEN 50
#define MAX_BCC_LEN 50
#define MAX_SUBJ_LEN 50
#define MAX_BODY_LEN 4000

#define fldStatusNONE     0
#define fldStatusADD      0x01
#define fldStatusUPDATE   0x02
#define fldStatusDELETE   0x04

class CMailRecord
{
public:

      CMailRecord( int nRecId, WORD wStatus, WORD CatId, WORD
DeliveryType,
            const char *szToField, const char *szCCField, const char
*szBCCField,
            const char *szSubjField, const char *szBodyField,UNID id);
      CMailRecord();
      CMailRecord( CMailRecord &rRec );
      CMailRecord &operator=(CMailRecord &rRec );

      // Inline helpers
      void SetStatus( WORD wStatus ) { m_wStatus = wStatus; }

      DWORD m_RecId;
      DWORD m_CatId;
      WORD m_wStatus;
      WORD m_DeliveryType;
      char *m_szToField;
      char *m_szCCField;
      char *m_szBCCField;
      char *m_szSubjField;
      char *m_szBodyField;
      UNID docUNID;

};
```

Appendix B - Source code for PalmPilot application for Notes password entry

```
#include <Pilot.h>                        // all the system toolbox
headers
#include "PasswdDlgRsc.h"                 // application resource
defines

#define RECORD_SIZE 100
/**********************************************************************
*
 * Global variables for this module

**********************************************************************/
static MenuBarPtr       CurrentMenu = NULL;     // P2. ptr to current
menu
static Word                   CurrentView;                   // P3. id
of current form
static LocalID                PasswordDbID;
static DmOpenRef        PasswordDbRef;
static LocalID                HotSyncDbID;
static UInt                   cardNo;
static VoidHand               mailInfoRecordHdl;
static UInt                   mailInfoRecordIndex;
static UInt                   ValidMailInfoRec=0;            // this
variable is just used to double check that the mail info rec exists
static UInt                   ValidStoredPassword=0;  // this variable
is used to check whether there's a valid stored password.
/**********************************************************************
*
 * Prototypes for internal functions

**********************************************************************/
static void StartApplication(void);
static Boolean MainDialogBoxFormHandleEvent(EventPtr event);
static Boolean ConfigDialogBoxFormhandleEvent( EventPtr event);
static Boolean ApplicationHandleEvent (EventPtr event);     // P3.
handle form load events
static void EventLoop(void);

static VoidPtr GetObjectPtr (Word objectID)
{
      FormPtr frm;

      frm = FrmGetActiveForm ();
      return (FrmGetObjectPtr (frm, FrmGetObjectIndex (frm,
objectID)));

}

/**********************************************************************
*
```

```
 *
 * FUNCTION:     StartApplication
 *
 * DESCRIPTION:  This routine sets up the initial state of the
application.
 *
 * PARAMETERS:   None.
 *
 * RETURNED:     Nothing.
 *

***********************************************************************
/
static void StartApplication(void)
{

      Err                                  err;
      int                            i;
      DmSearchStateType       searchState;
      ULong
      numRecordsP,totalBytesP,dataBytesP;
      Ptr                                  recordPtr;
      char                           key = 'm';
      static VoidHand                tempHdl;
      CurrentView = PasswdMainDialogForm;

      // Get the card number and database id of the sync application.
      err = DmGetNextDatabaseByTypeCreator (true, &searchState,
            sysFileTApplication, sysFileCSync, true, &cardNo,
&HotSyncDbID);

      // Getting the database id of the Password database
      PasswordDbID = DmFindDatabase(cardNo, "NotesInfoDB");

      // if the database doesn't exist, then create it, and also create
a mail info record
      if (!PasswordDbID)
      {
            err = DmCreateDatabase(cardNo, "NotesInfoDB", 0x4C525332,
0x01, false);
            PasswordDbID = DmFindDatabase(cardNo, "NotesInfoDB");
            PasswordDbRef = DmOpenDatabase(cardNo, PasswordDbID,
dmModeReadWrite);
            numRecordsP = 0;
            mailInfoRecordHdl = DmNewRecord(PasswordDbRef, (unsigned
short *) &numRecordsP, RECORD_SIZE);
            mailInfoRecordIndex = (unsigned short) numRecordsP;
            recordPtr = MemHandleLock(mailInfoRecordHdl);
            DmSet(recordPtr, 0,RECORD_SIZE, '\0');
            DmWrite(recordPtr,0, &key, 1);
            MemHandleUnlock(mailInfoRecordHdl);
            DmReleaseRecord(PasswordDbRef, mailInfoRecordIndex, false);
            ValidMailInfoRec = 1;
            ValidStoredPassword = 0;
            CurrentView = PasswdConfigDialogForm;
      }
      else
      {
            PasswordDbRef = DmOpenDatabase(cardNo, PasswordDbID,
dmModeReadWrite);
            DmDatabaseSize(cardNo, PasswordDbID, &numRecordsP,
&totalBytesP, &dataBytesP);
            // Going through the database, and determining if we have a
valid mail info record.

            if (numRecordsP)
            {
                  for (i=numRecordsP-1; i >= 0; i--)
                  {
                        tempHdl = DmQueryRecord(PasswordDbRef, i);
                        recordPtr = MemHandleLock(tempHdl);
                        if (*recordPtr == 'm')
                        {
                              mailInfoRecordHdl = tempHdl;
                              mailInfoRecordIndex = i;
                              ValidMailInfoRec = 1;
                              MemHandleUnlock(tempHdl);
                        }

                        if (*recordPtr == 'p')
                        {
                              ValidStoredPassword = 1;
                              MemHandleUnlock(tempHdl);
                        }

                        if (*recordPtr == 'a')
                        {
                              MemHandleUnlock(tempHdl);
                              DmRemoveRecord(PasswordDbRef, i);
                        }

                  }
            }
            else
            {
                  ValidMailInfoRec = 0;
                  ValidStoredPassword = 0;
                  CurrentView = PasswdConfigDialogForm;
            }
      }
}



/*********************************************************************
*
 *
 * FUNCTION:            MainFormHandleEvent
 *
 * DESCRIPTION:   Handles processing of events for the ÒmainÓ form.
 *
 * PARAMETERS:          event       - the most recent event.
 *
 * RETURNED:            True if the event is handled, false otherwise.
 *

***********************************************************************
/
static Boolean MainDialogBoxFormHandleEvent(EventPtr event)
{
      Boolean           handled = false;
      //EventType newEvent;
      FormPtr           frm;

      switch (event->eType)
      {

      case ctlSelectEvent:  // A control button was pressed and
released.

            // The "Enter password on the Pilot" button was pressed
            if (event->data.ctlEnter.controlID ==
PasswdMainDialogUsePilotPasswordCheckbox)
            {

                  ControlPtr button;
                  ControlPtr checkbox;
                  FieldPtr passwdfield;

                  button =
GetObjectPtr(PasswdMainDialogClearTextButton);
                  checkbox =
GetObjectPtr(PasswdMainDialogStorePasswordCheckbox);
                  passwdfield = GetObjectPtr
(PasswdMainDialogPasswordField);

                  CtlSetEnabled(checkbox,true);
                  CtlSetEnabled(button, true);
                  FldSetUsable(passwdfield,true);
            }


            // The "Use password stored in Desktop" was checked
            else if (event->data.ctlEnter.controlID ==
PasswdMainDialogUseStoredPasswordCheckbox)
            {

                  ControlPtr button;
                  ControlPtr checkbox;
                  FieldPtr passwdfield;
                  button =
GetObjectPtr(PasswdMainDialogClearTextButton);
                  checkbox =
GetObjectPtr(PasswdMainDialogStorePasswordCheckbox);
                  passwdfield = GetObjectPtr
(PasswdMainDialogPasswordField);

                  CtlSetEnabled(checkbox, false);
                  CtlSetEnabled(button, false);
                  FldSetUsable(passwdfield,false);
            }

            // The "Show Password in Clear Text" button was pressed
            else if (event->data.ctlEnter.controlID ==
PasswdMainDialogClearTextButton)
            {
                  FieldPtr    passwordField;
                  ControlPtr  button;
                  FontID            font;
                  CharPtr           clear = "Show Password";
                  CharPtr           cypher = "Hide Password";

                  button =
GetObjectPtr(PasswdMainDialogClearTextButton);
                  passwordField =
GetObjectPtr(PasswdMainDialogPasswordField);
                  font = FldGetFont(passwordField);
                  if (font == symbolFont)
                  {
                        //FldSetFont(passwordField, stdFont);
                        passwordField->fontID = stdFont;
                        FldEraseField(passwordField);
                        FldDrawField(passwordField);
                        CtlSetLabel(button, cypher);
                  }
                  else if (font == stdFont)
                  {
                        //FldSetFont(passwordField, symbolFont);
                        passwordField->fontID = symbolFont;
                        FldEraseField(passwordField);
                        FldDrawField(passwordField);
                        CtlSetLabel(button,clear );
                  }

                  handled = true;
            }

            // P3. If the new button is pressed, go to the edit form.
            else if (event->data.ctlEnter.controlID ==
PasswdMainDialogLocalsyncButton)
            {
                  Err                                 err;
                  FieldPtr                      passwordField;
                  CharPtr                             passwordText;
                  VoidHand                      RecordHandle;
                  UInt                          index = 10;
                  ControlPtr                    pilotChkBox,
storeChkBox, localChkBox;
                  Ptr                                 p;
                  int                                 password_length;
                  char                          a = ’a’;
                  char                          flag;       // this
variable is used to indicate which passwd the user wants to use,

      // and also to determine if the user wants to store the password
locally.
```

```
                  char                          null = '\0';
                  // Setting the entered password in the NotesInfoDb.

                  // this just double checks that there's indeed a
valid mail info record.
                  // if for some reason the record got corrupted, warn
the user to try
                  // configuring it again!

                  if (!ValidMailInfoRec)
                  {
                              // Load the info form, then display it.
                              frm = FrmInitForm(PasswdErrorDialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                              break;
                  }

                  localChkBox =
GetObjectPtr(PasswdMainDialogUseStoredPasswordCheckbox);
                  pilotChkBox =
GetObjectPtr(PasswdMainDialogUsePilotPasswordCheckbox);
                  storeChkBox =
GetObjectPtr(PasswdMainDialogStorePasswordCheckbox);

                  if (CtlGetValue(pilotChkBox))
                  {

                        passwordField = GetObjectPtr
(PasswdMainDialogPasswordField);
                        passwordText = FldGetTextPtr (passwordField);
                        FldReleaseFocus (passwordField);

                        // Release any free space in the memo field.
                        FldCompactText (passwordField);
                        if (passwordText)
                              password_length = StrLen(passwordText);
                        else
                        {
                              // Load the info form, then display it.
                              frm =
FrmInitForm(PasswdError3DialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                                    break;
                        }
                        if (CtlGetValue(storeChkBox))
                              flag = 's';
                        else
                              flag = 'p';
                  }
                  else if (CtlGetValue(localChkBox))
                        flag = 'l';



                  // Creating new record
                  RecordHandle = DmNewRecord(PasswordDbRef, &index,
RECORD_SIZE);

                  p = MemHandleLock(RecordHandle);
                  DmWrite(p,0, &a, 1);
                  DmWrite(p,1, &flag, 1);
                  DmWrite(p,2, &null, 1);
                  if (CtlGetValue(pilotChkBox))
                  {
                        DmWrite(p,3,passwordText, password_length);
                        DmWrite(p,3+password_length, &null, 1);
                  }
                  else
                  {
                        DmWrite(p,3,&null,1);
                  }
                  MemHandleUnlock(RecordHandle);
                  DmReleaseRecord(PasswordDbRef, index, false);

                  err = SysUIAppSwitch (cardNo,
HotSyncDbID,sysAppLaunchCmdSyncRequestLocal, NULL);

                  handled = true;
            }
            else if (event->data.ctlEnter.controlID ==
PasswdMainDialogModemsyncButton)
            {

                  Err                                   err;
                  FieldPtr                        passwordField;
                  CharPtr                               passwordText;
                  VoidHand                        RecordHandle;
                  UInt                            index = 10;
                  ControlPtr                      pilotChkBox,
storeChkBox, localChkBox;
                  Ptr                                   p;
                  int                                   password_length;
                  char                            a = 'a';
                  char                            flag;       // this
variable is used to indicate which passwd the user wants to use,

      // and also to determine if the user wants to store the password
locally.
                  char                            null = '\0';
                  // Setting the entered password in the NotesInfoDb.

                  // this just double checks that there's indeed a
valid mail info record.
                  // if for some reason the record got corrupted, warn
the user to try
                  // configuring it again!
                  if (!ValidMailInfoRec)
                  {
                              // Load the info form, then display it.
                              frm = FrmInitForm(PasswdErrorDialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                              break;
                  }

                  localChkBox =
GetObjectPtr(PasswdMainDialogUseStoredPasswordCheckbox);
                  pilotChkBox =
GetObjectPtr(PasswdMainDialogUsePilotPasswordCheckbox);
                  storeChkBox =
GetObjectPtr(PasswdMainDialogStorePasswordCheckbox);

                  if (CtlGetValue(pilotChkBox))
                  {

                        passwordField = GetObjectPtr
(PasswdMainDialogPasswordField);
                        passwordText = FldGetTextPtr (passwordField);
                        FldReleaseFocus (passwordField);

                        // Release any free space in the memo field.
                        FldCompactText (passwordField);
                        if (passwordText)
                              password_length = StrLen(passwordText);
                        else
                        {
                              // Load the info form, then display it.
                              frm =
FrmInitForm(PasswdError3DialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                                    break;
                        }
                        if (CtlGetValue(storeChkBox))
                              flag = ’s’;
                        else
                              flag = ’p’;
                  }
                  else if (CtlGetValue(localChkBox))
                        flag = ’l’;



                  // Creating new record
                  RecordHandle = DmNewRecord(PasswordDbRef, &index,
RECORD_SIZE);
```

```
                  p = MemHandleLock(RecordHandle);
                  DmWrite(p,0, &a, 1);
                  DmWrite(p,1, &flag, 1);
                  DmWrite(p,2, &null, 1);
                  if (CtlGetValue(pilotChkBox))
                  {
                        DmWrite(p,3,passwordText, password_length);
                        DmWrite(p,3+password_length, &null, 1);
                  }
                  else
                  {
                        DmWrite(p,3,&null,1);
                  }

                  MemHandleUnlock(RecordHandle);
                  DmReleaseRecord(PasswordDbRef, index, false);

                  err = SysUIAppSwitch (cardNo,
HotSyncDbID,sysAppLaunchCmdSyncRequestRemote, NULL);

                  handled = true;
            }
            break;

      case frmOpenEvent:      // P3. The form was told to open.
                  // It has already been loaded and activated so just
draw it.
                  {
                        ControlPtr checkbox;
                        checkbox =
GetObjectPtr(PasswdMainDialogUseStoredPasswordCheckbox);
                        if (!ValidStoredPassword)
                              CtlHideControl(checkbox);
                        FrmDrawForm (FrmGetActiveForm());
                        handled = true;
                        break;
                  }
      case menuEvent:         // P2. A menu item was selected.
                  // The Òget infoÓ being the only menu item, display
the info form.

                  switch (event->data.menu.itemID)
                  {
                        case (OptionsAbout):
                        {
                              // First clear the menu status from the
display.
                              MenuEraseStatus(CurrentMenu);

                              // Load the info form, then display it.
                              frm = FrmInitForm(PasswdAboutDialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                              break;
                        }
                        case (OptionsConfigureNotesInfo):
                        {
                              // Clear Menu
                              FrmPopupForm (PasswdConfigDialogForm);
                              handled = true;
                              break;
                        }
                  }

            }
      return(handled);
}


static Boolean ConfigDialogBoxFormhandleEvent (EventPtr event)
{
      Boolean handled = false;
      FormPtr frm;

      if (event->eType == ctlSelectEvent)
            {
            switch (event->data.ctlSelect.controlID)
                  {
                  case PasswdConfigDialogOKButton:
                  {
                        FieldPtr tempField;
                        CharPtr     tempText;
                        int   tempLength, current;
                        Ptr   p;
                        char null = '\0';


                        tempField = GetObjectPtr
(PasswdConfigDialogMailServerNameField);
                        tempText = FldGetTextPtr (tempField);
                        FldReleaseFocus (tempField);

                        // Release any free space in the field.
                        FldCompactText (tempField);
                        if (tempText)
                              tempLength = StrLen(tempText);
                        else
                              break;

                        mailInfoRecordHdl = DmGetRecord(PasswordDbRef,
mailInfoRecordIndex);
                        p = MemHandleLock(mailInfoRecordHdl);


                        DmWrite(p,1, &null, 1);
                        DmWrite(p,2, &null, 1);
                        DmWrite(p,3,tempText, tempLength);
                        DmWrite(p,3+tempLength, &null, 1);
                        current = 4 + tempLength;
                        tempField = GetObjectPtr
(PasswdConfigDialogMailFileNameField);
                        tempText = FldGetTextPtr (tempField);
                        FldReleaseFocus (tempField);

                        // Release any free space in the field.
                        FldCompactText (tempField);
                        if (tempText)
                              tempLength = StrLen(tempText);
                        else
                              break;

                        if (!tempLength)
                        {
                        // if user didn't type in a mail file name,
warn him/her!

                              MemHandleUnlock(mailInfoRecordHdl);
                              DmReleaseRecord(PasswordDbRef,
mailInfoRecordIndex, false);
                              // Load the info form, then display it.
                              frm =
FrmInitForm(PasswdError1DialogForm);
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                              break;
                        }

                        DmWrite(p,current,tempText, tempLength);
                        DmWrite(p,current+tempLength, &null, 1);
                        current = current + tempLength + 1;

                        tempField = GetObjectPtr
(PasswdConfigDialogNotesIDNameField);
                        tempText = FldGetTextPtr (tempField);
                        FldReleaseFocus (tempField);

                        // Release any free space in the field.
                        FldCompactText (tempField);
                        if (tempText)
                              tempLength = StrLen(tempText);
                        else
                              break;
                        // if user didn't type in an id file name, warn
him/her!

                        if (!tempLength)
                        {
                              MemHandleUnlock(mailInfoRecordHdl);
                              DmReleaseRecord(PasswordDbRef,
mailInfoRecordIndex, false);
                              // Load the info form, then display it.
                              frm =
FrmInitForm(PasswdError2DialogForm);
```

```
                              FrmDoDialog(frm);

                              // Delete the info form.
                              FrmDeleteForm(frm);
                              handled = true;
                              break;
                        }
                        DmWrite(p,current,tempText, tempLength);
                        DmWrite(p,current+tempLength, &null, 1);

                        MemHandleUnlock(mailInfoRecordHdl);
                        DmReleaseRecord(PasswordDbRef,
mailInfoRecordIndex, false);
                        CurrentView = PasswdMainDialogForm;
                        FrmGotoForm(CurrentView);
                        handled = true;
                        break;
                  }

                  case PasswdConfigDialogCancelButton:

                        CurrentView = PasswdMainDialogForm;
                        FrmGotoForm(CurrentView);
                        handled = true;
                        break;

                  }
            }

      else if (event->eType == frmOpenEvent)
            {
                  FieldPtr                      servernameFieldPtr,
mailpathnameFieldPtr, idfilenameFieldPtr;
                  VoidHand                      servernameHdl,
mailpathnameHdl, idfilenameHdl;
                  CharPtr                             servernamePtr,
mailpathnamePtr, idfilenamePtr;
                  Ptr                                 recordPtr;
                  ULong
      numRecordsP,totalBytesP,dataBytesP;
                  int                           i, temp;
                  int                                 flag = 0;
                  char                          key = 'm';


                  servernameFieldPtr = GetObjectPtr
(PasswdConfigDialogMailServerNameField);
                  mailpathnameFieldPtr =
GetObjectPtr(PasswdConfigDialogMailFileNameField);
                  idfilenameFieldPtr =
GetObjectPtr(PasswdConfigDialogNotesIDNameField);
                  DmDatabaseSize(cardNo, PasswordDbID, &numRecordsP,
&totalBytesP, &dataBytesP);

                  if (ValidMailInfoRec)
                  {
                        recordPtr = MemHandleLock(mailInfoRecordHdl);
                        recordPtr += 3;
                        temp = StrLen(recordPtr);
                        servernameHdl = MemHandleNew(temp+1);
                        servernamePtr = MemHandleLock(servernameHdl);
                        MemSet(servernamePtr, temp + 1, ’\0’);
                        for (i = 0; i < temp+1;i++)
                              servernamePtr[i]=  recordPtr[i];
                        MemHandleUnlock(servernameHdl);
                        FldSetTextHandle(servernameFieldPtr,
servernameHdl);

                        recordPtr += temp + 1;
                        temp = StrLen(recordPtr);
                        mailpathnameHdl = MemHandleNew(temp+1);
                        mailpathnamePtr =
MemHandleLock(mailpathnameHdl);
                        MemSet(mailpathnamePtr, temp + 1, ’\0’);
                        for (i = 0; i < temp+1;i++)
                              mailpathnamePtr[i]=  recordPtr[i];
                        MemHandleUnlock(mailpathnameHdl);
                        FldSetTextHandle(mailpathnameFieldPtr,
mailpathnameHdl);

                        recordPtr += temp + 1;
                        temp = StrLen(recordPtr);
                        idfilenameHdl = MemHandleNew(temp+1);
                        idfilenamePtr = MemHandleLock(idfilenameHdl);
                        MemSet(idfilenamePtr, temp + 1, ’\0’);
                        for (i = 0; i < temp+1;i++)
                              idfilenamePtr[i]=  recordPtr[i];
                        MemHandleUnlock(idfilenameHdl);
                        FldSetTextHandle(idfilenameFieldPtr,
idfilenameHdl);

                        MemHandleUnlock(mailInfoRecordHdl);
                  }
                  else
                  {
                        mailInfoRecordHdl = DmNewRecord(PasswordDbRef,
(unsigned short *) &numRecordsP, RECORD_SIZE);
                        mailInfoRecordIndex = (unsigned short)
numRecordsP;
                        recordPtr = MemHandleLock(mailInfoRecordHdl);
                        DmWrite(recordPtr,0, &key, 1);
                        MemHandleUnlock(mailInfoRecordHdl);
                        ValidMailInfoRec = 1;
                  }

                  frm = FrmGetActiveForm ();
                  FrmDrawForm (frm);
                  handled = true;
            }

      return (handled);
}
```

```
/**********************************************************************
*
 *
 * FUNCTION:    ApplicationHandleEvent
 *
 * DESCRIPTION: P3. This routine loads a form resource and sets the
event handler for the form.
 *
 * PARAMETERS:  event  - a pointer to an EventType structure
 *
 * RETURNED:    True if the event has been handled and should not be
 *                                  passed to a higher level handler.
 *

***********************************************************************
/
static Boolean ApplicationHandleEvent (EventPtr event)
{
      FormPtr     frm;
      Word        formId;
      Boolean     handled = false;

      if (event->eType == frmLoadEvent)
            {
            // load the form resource specified in the event then
activate the form.
            formId = event->data.frmLoad.formID;
            frm = FrmInitForm(formId);
            FrmSetActiveForm(frm);

            // set the event handler for the form.  The handler of the
currently
            // active form is called by FrmDispatchEvent each time it
receives an event.
            switch (formId)
                  {
                  case PasswdMainDialogForm:
                        FrmSetEventHandler(frm,
MainDialogBoxFormHandleEvent);
                        break;
                  case PasswdConfigDialogForm:
                        FrmSetEventHandler(frm,
ConfigDialogBoxFormhandleEvent);
                        break;
                  }
            handled = true;
            }

      return handled;
}


/**********************************************************************
*
 *
 * FUNCTION:            EventLoop
 *
 * DESCRIPTION:     A simple loop that obtains events from the Event
 *                                   Manager and passes them on to
various applications and
 *                                   system event handlers before
passing them on to
 *                                   FrmDispatchEvent for default
processing.
 *
 * PARAMETERS:            None.
 *
 * RETURNED:              Nothing.
 *

***********************************************************************
/
static void EventLoop(void)
{
      EventType    event;
      Word              error;

      do
            {
            // Get the next available event.
            EvtGetEvent(&event, evtWaitForever);

            // Give the system a chance to handle the event.
            if (! SysHandleEvent(&event))

                  // P2. Give the menu bar a chance to update and
handle the event.
                  if (! MenuHandleEvent(CurrentMenu, &event, &error))

                        // P3. Give the application a chance to handle
the event.
                        if (! ApplicationHandleEvent(&event))

                              // P3. Let the form object provide
default handling of the event.
                              FrmDispatchEvent(&event);
            }
      while (event.eType != appStopEvent);
}


/**********************************************************************
*
 *
 * FUNCTION:              PilotMain
 *
 * DESCRIPTION:     This function is the equivalent of a main() function
 *                                   under standard ÒCÓ.  It is called
by the Emulator to begin
 *                                   execution of this application.
 *
 * PARAMETERS:            cmd - command specifying how to launch the
application.
```

```
 *                                 cmdPBP - parameter block for the
command.
 *                                 launchFlags - flags used to
configure the launch.
 *
 * RETURNED:            Any applicable error codes.
 *

**********************************************************************
/
DWord PilotMain(Word cmd, Ptr cmdPBP, Word launchFlags)
{
      // Check for a normal launch.
      if (cmd == sysAppLaunchCmdNormalLaunch)
            {
            // Initialize the application's data structures, ...
            StartApplication();

            // P3. Start the first form
            FrmGotoForm(CurrentView);

            // Start the event loop.
            EventLoop();
            }
      return 0;
}
```

Appendix C - Notes Mail Conduit User Survey

1) General configuration of system being used:

a) Notes Version used: ___
b) PalmPilot type used: ___
c) Operating System: ___
d) Type of Computer: (Pentium, Mac, etc..) ___
e) Speed of Processor: ___
f) Amount of RAM: ___

2) On average, how many times do you Hotsync each day?

3) Below is a list of the options available in the configuration menu of the Notes Mail conduit. Please check which options you usually use.

General Synchronization Options:
a) Send Messages in PalmPilot Outbox [ ]
b) Propagate Draft modifications from Pilot to Notes [ ]
c) Propagate deletions from Pilot to Notes [ ]
d) Synchronize messages from Notes Inbox to Pilot Inbox [ ]
e) Synchronize messages from custom Notes view/folder [ ]
f) Propagate Draft modifications from Notes to Pilot [ ]
g) Propagate deletions from Notes to Pilot [ ]

Outbox Options - Save Mail sent from Handheld in Notes?

a) Always Save [ ]
b) Prompt for each message [ ]
c) Never Save [ ]

Inbox Options - Transfer from Notes Inbox

a) All new unread messages since last Hotsync [ ]
b) All new messages since last Hotsync [ ]
c) All messages since a particular date [ ]
d) All messages in Inbox [ ]
e) Clear Handheld Inbox before transferring messages [ ]

Custom Options - Transfer all messages from a Notes view/folder to:

a) Inbox category in handheld [ ]
b) Filed category in handheld [ ]
c) Clear the selected handheld category before hotsyncing [ ]

4) Are there any other options that you would find useful in terms of selecting the Notes messages that you want transferred to your PalmPilot?

5) On average, how many daily email messages do you transfer from Notes to your PalmPilot, and how many messages do you compose and send from your PalmPilot?

6) What is the average size of messages that you transfer to your PalmPilot?

a) less than 5K [ ]
b) between 5-10K [ ]
c) between 10-20K [ ]
d) more than 20K[ ]
e) other __________

7) How long does it usually take for the NotesMail conduit to finish synchronizing?

a) less than 30 seconds [ ]
b) between 30sec-1 min [ ]
c) between 1-2 min [ ]
d) between 2-5 min [ ]
e) sometimes it takes forever and Hotsync manager times out [ ]
f) other ________

8) Currently, the NotesMail conduit does not convert certain elements of a Notes document to the PalmPilot. Rank in order of importance which of the following parts of a Notes document you would like to see supported:

a) Graphics [ ]
b) Doclinks [ ]
c) Attachments [ ]
d) Editable Fields/Buttons [ ]
e) Other ______________________________

9) Currently, you can only use the Notes Mail conduit by synchronizing with the same machine every time. How useful would it be if you could use the Notes Mail conduit by synchronizing with any machine that had a replica of your Mail database?

10) Using Network Hotsync, one can hotsync with a particular machine from any machine connected to the network. One alternative synchronization model consists of allowing users to directly hotsync with the Notes Server that contains the data that they with to have in their PalmPilots. Such model allows a Pilot user on the road to directly access information in a Notes Server by simply connecting the Pilot to the corporate network using a modem. In your corporate setting, how useful would such server synchronization model be, and how do you think users would use such feature?

11) Are there any aspects of the current version of the Notes Mail conduit that you think needs to be improved or modified?

12) What are some additional features that you would like to see supported in future versions of the Notes Mail conduit?

References

[1] Neeraja Sankaran, "Looking Back At ENIAC: Computers Hit Half-Century Mark", The Scientist, Vol:9, #16, August 21, 1995

[2] Diana Hwang and Randy Giusto, "Boom or Bust? Worldwide Smart Handheld Devices Market Review and Forecast Update, 1997-2001", International Data Corporation, 1998

[3] Kim Girard, "Apps show PalmPilot's Versatility", ComputerWorld, Jan. 26, 1998

[4] James Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System", Carnegie Mellon University

[5] Peter Reiher, John Heidemann, David Ratner, Greg Skinner, and Gerald Popek, "Resoling File Conflicts in the Ficus Files System", UCLA

[6] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer and Carl Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System", Xerox Palo Alto Research Center, 1995

[7] Anthony Joseph, Joshua Tauber and M. Frans Kaashoek, "Mobile Computing with the Rover Toolkit", MIT

[8] Kawell, et al., "Replicated Document Mangement in a Group Communication System," Lotus Technical Report 88-01, Lotus Development Corporation, 1988

[9] Fox, et al., "Experience With Top Gun Wingman: A Proxy-Based Graphical Web Browser for the USR PalmPilot," Submitted for publication, U.C. Berkely, 1998

[10] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. "The Notion of Consistency and Predicate Locks in a Database System," *CACM*, 19(11):624–633, November 1976.

[11] H. T. Kung and J. T. Robinson. "On Optimistic Methods for Concurrency Control," *ACM TODS*, 6(2):213–226, June 1981.

[12] Atul Adya, Robert Gruber, Barbara Liskov and Umesh Maheshwari, "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clocks," *ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995

[13] D. Agrawal, A. J. Bernstein, P. Gupta, and S. Sengupta, "Distributed Multi-version Optimistic Concurrency Control with Reduced Rollback," *Distributed Computing*, 2(1), 1987.

[14] Lotus Notes Server Administration Guide, "About how validation and authentication work", 1997.