Distributed and Parallel Databases, 15, 95–116, 2004

Edison: Database-Supported Synchronization for PDAs

MATTHEW DENNY mdenny@cs.berkeley.edu
MICHAEL J. FRANKLIN franklin@cs.berkeley.edu
University of California, Berkeley

Recommended by: Ahmed Elmagarmid

Abstract. One of the most important features provided by personal digital assistants is the ability to synchronize device data with data on a user's PC. Unfortunately, current synchronization protocols have significant shortcomings that limit the availability, usability, and performance of synchronization. We present Edison, a service that leverages existing off-the-shelf ORDBMS technology to address these problems. Edison allows large numbers of users to synchronize handheld devices from any point on the internet with subsets of large, shared data sets. Edison supports this functionality while transferring the minimal amount of data to and from the device. We describe the implementation of the Edison data server and protocol, and show that Edison requires minimal overhead in terms of DBMS storage and additional time per synchronization.

Keywords: synchronization PDA handheld

1. Introduction

The use of personal digital assistants (PDAs) is growing at a torrid pace. A key benefit of PDAs is the convenience they provide by allowing users to bring copies of needed information with them as they move throughout their day. Central to this functionality is the ability to easily replicate and reconcile data between the handheld device and data sources that reside on fixed (i.e., not mobile) backing stores.

Since PDAs are disconnected from the network most of the time, transactional techniques cannot be used to achieve consistent replication between PDAs and backing stores. Instead, PDAs typically implement a weaker form of consistency control called *synchronization*. During the time that a PDA is disconnected from the network data can be created, modified, or deleted at either the PDA or at the backing store. When the PDA is reconnected to the network, the synchronization protocol can be invoked to reconcile the data on the device with the data on the backing store.

Each handheld operating system (e.g., PalmOS, PocketPC, etc.) provides its own protocol for synchronization, but these protocols have a great deal of commonality. They do not provide ACID semantics as such strong guarantees are not needed for most Personal Information Management (PIM) applications (calendars, address books, etc.) where updates are fairly infrequent and conflicts are even more rare.[1]

In PDAs, data is associated with particular applications. The data for an application is represented as a set of records. These records serve as the granularity of consistency. When a PDA is reconnected to the network and synchronization is initiated, the PDA and the data store exchange information to determine which updates need to be transfered in either direction (i.e., PDA to data store or vice versa). The synchronization protocols rely upon *synchronization metadata* to efficiently determine the relative states of the device-resident data and the data store. This metadata consists of "status bits" associated with each copy of each record, that indicate if that record copy has been created, modified, deleted, etc.

Records created or modified on one site are sent to the other. Records deleted on one site are removed from the other. Records for which conflicting operations were performed on both sites during the period of disconnection are dealt with through a *conflict resolution* procedure. Depending on the application, the conflict resolution may be performed automatically, according to predefined rules, or may require user intervention.

1.1. Limitations of current approaches

While there has been significant research on replication for both mobile file systems [5, 10, 16] and mobile database systems [12], the existing commercial synchronization protocols were developed largely independently of this work. The commercial protocols were designed originally for much simpler scenarios, effectively assuming that users would synchronize (i.e., "sync") only a single handheld device with user data stores on only one or two machines (e.g., a home or office PC). As a result, these protocols have significant limitations that have become increasingly apparent with the growth of network connectivity, device memory capacities, and user expectations. In particular:

1. A user can only synchronize with desktop PCs (or other machines) that have been explicitly preconfigured for synchronization of that particular user's data. This limits the usefulness of the devices when traveling, and can often result in availability problems, since the synchronization is done with client-class machines rather than with managed servers.
2. Even given such preconfiguration, users incur large performance penalties if they synchronize with multiple machines. This is penalty arises because the synchronization metadata stored at the various machines becomes mutually inconsistent, requiring entire data sets to be transmitted and compared in order to determine the proper synchronization actions to perform.
3. Synchronization with (parts of) large shared data stores (e.g., shared calendars) is not directly supported. The basic synchronization protocols have an extremely simplistic data model (i.e., flat, unshared files of records) and require a device to synchronize all of the records in each file.

Recently, web-based services such as FusionOne and Palm.net have been appearing to address the first two issues. While not yet widely used, these services allow users to synchronize their private PIM data from any site on the web. Such services, however, do not yet support synchronization for user-specific subsets of shared data stores. At the same time, various approaches for the problem of synchronizing shared data have been developed. However, these tend to be application-specific and often interact awkwardly with the generic synchronization protocols because of the simple data model on which the protocols are based. Still, it is clear that there is much effort underway to address the limitations of the basic synchronization protocols.

Our work stems from a simple realization about the basic protocols, namely, that all of the problems listed above are the result of poor management of synchronization metadata. For example, metadata is not shared among the sites at which a user might wish to synchronize, even if these sites have been identified ahead of time. Furthermore, the simple data model supported by the protocol does not allow the expression of subsets of interest within a larger data set, and the logic underlying the synchronization does not take into account the possibility of multiple users sharing the same data.[2]

1.2. Exploiting off-the-shelf database technology

From the above description, it should be clear that the current approach to the management of synchronization metadata is extremely primitive. What is needed is a system that can efficiently and reliably support synchronization metadata for many users accessing the system from anywhere in the network. Thus, we sought to investigate the use of an off-the-shelf commercial database system to manage the synchronization metadata for a large number of handheld devices and users. We have developed a system called Edison, that addresses the problems of availability, access, performance, and sharing for PDA-style synchronization.

Edison provides a shared repository where the synchronization metadata for each handheld device resides. Since synchronization is always performed using the same sync metadata at regardless of the network access point, the metadata will never becomes inconsistent, so synchronization can always be done without transmitting unnecessary data. Our current solution is a centralized one, but we have found that under current usage scenarios, such a solution can easily support many thousands of devices. By using off-the-shelf database technology, we expect to be able to scale the system to a cluster of servers, or if necessary to a small number of large data centers. In addition, we have developed metadata caching strategies that can offload much of the synchronization processing to other nodes in the network.

Edison supports several other features that current protocols do not provide. Edison has built-in support for device synchronization with large, shared data sources, such as those used in many groupware applications. Edison allows handheld users to replicate and update this data concurrently, ensuring that such updates are propagated to the correct users when those users synchronize. Edison also provides an extensibility framework for mobile application developers that allows them to easily add synchronization services to their applications.

In this paper, we focus on the Palm HotSync protocol and the accompanying Palm Desktop software as our primary example of a PDA synchronization protocol. This software is the default synchronization mechanism for PIM data on the most popular PDA operating system platform [4, 17]. The current implementation Edison has been developed to support the Palm HotSync protocol. As stated above, however, other vendor's protocols are essentially similar, and thus we expect the ideas behind Edison to carry over to such protocols as well.

The contributions of the paper are the following:

- We describe in some detail, the way that existing PDA synchronization protocols work, and identify the inherent limitations in their approach to metadata management.
- We develop an approach to managing such metadata for large numbers of handheld devices that leverages existing ORDBMS technology.
- We develop an architecture for the deployment of a synchronization service in wide area networks that incorporates the metadata repository and offloads much of the synchronization processing to other sites on the network.
- We present a performance analysis of the metadata repository that indicates that existing off-the-shelf database technology can support large numbers of handheld devices under current usage scenarios.

Section 2 describes current synchronization technology, concentrating on the Palm HotSync protocol. Section 3 outlines the Edison protocol, focusing on its network deployment. Section 4 provides an analysis of performance issues in Edison. Section 5 discusses related work, and Section 6 presents future research directions and conclusions.

2. Palm synchronization

2.1. Overview and data structures

We begin by describing the basic synchronization solution for the Palm OS. The Palm synchronization solution consists of code and data in three components: the handheld, the Palm Desktop PIM software with its associated conduits, and the HotSync Manager.[3]

2.1.1. The handheld. The handheld keeps its data and synchronization metadata in what we call handheld databases (*HHDBs*). This is the primary file abstraction for PalmOS. Each HHDB is associated with a certain application, which is identified in the HHDB header by an application-specific *creatorID*.[4] An HHDB consists of a set of records which are NVRAM chunks. Each HHDB record contains (in addition to the actual data) a record identifier (*HHrecordUID*) and a set of *status bits* that indicate modifications to the record (e.g., insertion, deletion, update). This storage structure is very similar to that of other handhelds. For instance, the PocketPC synchronizes groups of objects, which have similar associated identifiers, data, and status information.

2.1.2. The HotSync Manager. The HotSync Manager is a PC service that listens on a communications port for synchronization requests, performs the low-level communication, and allows desktop applications to synchronize with selected HHDBs. Each application that will be synchronizing with HHDBs must register a *conduit* with the HotSync Manager. A conduit is a piece of downloadable code in the form of a Windows DLL or Java class that executes the synchronization process for that application. The HotSync Manager uses the creatorID of each HHDB to determine the appropriate conduit to call during synchronization. The HotSync Manager also generates a pseudo-random *PC ID* to uniquely identify the

Desktop PC. It writes the PC ID to the handheld on each synchronization, which identifies the last PC that the handheld synchronized with.

2.1.3. The Palm Desktop. The Palm Desktop is the default desktop software for the Palm PIM applications. It provides storage and manipulation of the desktop replica of the Palm PIM data, and also provides a conduit for the applications. The Palm Desktop stores a replica of the data from each PIM HHDB (which we will call *desktop data sets*), along with the associated HHrecordUIDs and its own copy of the status bits for each record. These status bits indicate changes to the desktop-resident copies of the data since the last synchronization. Also, the Palm Desktop maintains a snapshot of each desktop data set taken immediately after the most recent synchronization. This snapshot is used to determine which updates are needed for synchronization in situations where the local status bits cannot be trusted (so called "slow sync", as is described in the following section). The Palm Desktop runs on a single PC without communication from any other nodes on a network. It does not share data or metadata with any other instances of itself or other data sources.

2.2. *The protocol*

2.2.1. Overview. The synchronization process is initiated by the handheld, which sends the creatorIDs of its HHDBs to the HotSync Manager. The HotSync manager then synchronizes each HHDB in turn by invoking the appropriate conduit. The goal of the synchronization is to ensure that updates made on the desktop but not on the handheld are installed at the handheld, and vice versa.

Depending on the state of the synchronization metadata, a conduit can engage in either a fast sync or a slow sync to determine the status of replicas on the desktop and the handheld. Given this status information, the synchronization action performed for each updated record is determined by a set of rules called the *synchronization logic*. The default synchronization logic for the Palm Desktop conduit is shown in Table 1. Note that in this table, "No Record" indicates that a record with that HHrecordUID is not present at the site. Also note that for the Palm Desktop, all of the conditions can be handled automatically, except for the case in which a record was updated at both locations. In this latter case, user intervention is required.

The conduit applies the actions dictated by the synchronization logic, updates the desktop data set, and sends any needed updates to the handheld. The synchronization ends by removing any records at either site that are still marked deleted, clearing all the status bits on the HHDB and desktop data set, and storing a new snapshot of all the data in the desktop data set at the desktop.

2.2.2. Fast sync vs. slow sync. When the Palm Desktop conduit is called by the HotSync Manager, it first checks the PC ID on the handheld to see if the handheld's last synchronization was with this desktop. If so, then a *fast sync* occurs, where the handheld need only upload those records whose status bits are set on the device.[5] In this case, the conduit can determine the state of records simply by comparing the status bits of the uploaded records to those in the corresponding desktop data set records.

Table 1. Synchronization logic.

Handheld record status	Desktop record status	Palm desktop action
Add	No record	Add the handheld record to the desktop.
No record	Add	Add the desktop record to the handheld.
Delete	No change	Delete record on the desktop.
No change	Delete	Delete record on the handheld.
No record	Delete	No Action.
Delete	No Record	No Action.
Delete	Delete	No Action.
Delete	Change	Send updated desktop record to handheld (handheld record is "un-deleted").
Change	Delete	Send updated handheld record to desktop (desktop record is "un-deleted").
Change	No Change	Update desktop record with handheld value.
No change	Change	Send desktop record to the handheld.
Change	Change	Create two copies reflecting each update on both the handheld and desktop and notify user of conflict.

In contrast, if an HHDB's last synchronization was with a different desktop, the relative states of the desktop and handheld copies of the records cannot be determined simply by comparing status bits. This scenario is quite common. It can arise for example, if a user synchronizes with a home PC and then an office PC. In the mobile computing literature, this is known as the multiple endpoints problem.

For example, consider the case where an HHDB is first synchronized with desktop A, then with desktop B. Recall that after each synchronization, the status bits on both the desktop and the handheld are cleared. Thus, when the HHDB attempts to synchronize with desktop B, its status bits only represent the changes since its synchronization with desktop A. In this case, simply comparing status bits would result in incorrect synchronization actions. Thus, the HotSync protocol requires a *slow sync* in this second synchronization. In a slow sync, the handheld uploads *the entire HHDB* to the desktop. The conduit compares every record from both the HHDB and the desktop data set against the snapshot taken the last time the handheld synchronized with desktop B. This 3-way comparison enables the conduit to determine which records have changed at the handheld and/or at the desktop since the pair's last synchronization. The protocol then applies the synchronization logic and finishes as described above.

Since the slow sync transfers much more data between the handheld and the desktop than fast sync, users will see poor performance in the slow sync case. Figure 1 shows the synchronization times for fast sync and slow sync with 25 2 KB records updated on a Handspring Visor using a USB connection as the HHDB size is increased from 100 KB to 1.5 MB. While the fast sync time is fairly insensitive to the HHDB size, slow sync time grows linearly.[6] When the HHDB size is held constant, other experiments (not shown here)

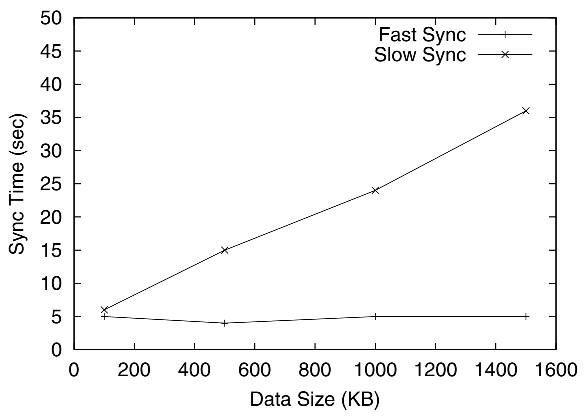

Figure 1. Sync times vs. HHDB size.

show that the gap between fast sync and slow sync remains almost constant as the number of updates between synchronizations increases from 10 to 100.

If the gap between fast syncs and slow syncs is so large, why has this performance problem not already impeded handheld use? Most of these protocols were designed for applications which are not hampered by these performance problems. Most of these applications (e.g. PIM) have a small amount of data (less than 1 MB), and most users synchronize while at their PCs. Thus, most users are not significantly inconvenienced by waiting 25 sec. or less for a synchronization while working at their PCs. However, this will change with the advent of handheld multimedia applications and wireless networks. For instance, one major application for palmtops is playing MP3s. If we extrapolate on figure 1, a handheld containing 10 3 MB MP3s will take at least 10 minutes to slow sync! Also, synchronizations over wireless networks are much less amenable to slow sync. On wireless networks, users are generally away from their desktops and may need data immediately. Thus, long synchronization times can present a severe inconvenience. Also, many wireless users pay for service on a per kilobyte basis, making large data transfers for synchronization unacceptable.

2.3. Other synchronization protocols

Most current PDA synchronization protocols use a framework similar to HotSync. Microsoft's ActiveSync for Pocket PCs has an ActiveSync Manager that manages application-specific "service providers" in the same way the HotSync Manager manages conduits. Like Palm, ActiveSync requires users to synchronize with pre-configured machines—the location of the ActiveSync desktop metadata files for a PC are maintained locally in that machine's registry. ActiveSync also suffers from the multiple endpoints problem. If a handheld synchronizes with alternate PCs as outlined in Section 2.2.2, ActiveSync engages in a synchronization similar to slow sync. In this case, the handheld sends all objects to the data source and the service provider becomes responsible for detecting and resolving conflicts. Alternatively, the service provider can consider all handheld objects deleted, and the service provider propagates every object from the desktop to the handheld.[7] Unlike the single user Palm Desktop, ActiveSync allows direct synchronizations with an Exchange server, which can access shared Exchange server data [14]. However, these synchronizations are limited to Exchange data, and we have little information about the implementation or efficiency of these synchronizations.

In addition to handheld vendors, third-party vendors such as [22, 23] provide solutions for synchronizing handhelds with specific applications such as Microsoft Exchange and Lotus Notes. These solutions do not help users synchronize with generic data stores such as ODBC data sources. Palm has a separate product called Palm HotSync Server for synchronizing with shared data sources [18]. However, this solution has the drawback of requiring a significant amount of code to ensure that shared data updates get propagated to the correct handhelds. See Section 5 for a comparison of Palm HotSync Server and Edison. As stated before, there is little information in the literature on how these synchronization systems are implemented.

The SyncML industry consortium has released a specification for a synchronization standard [25]. SyncML provides a device-independent markup language used for synchronizing between handhelds and servers. This language, however, specifies neither the metadata format for keeping track of changed data nor how this metadata should be managed.

2.4. Problems with the protocols

As stated before, current PDA synchronization protocols have three main deficiencies:

- *Synchronization is only Allowed with Stand-Alone Replicas Pre-Configured to Synchronize with a User's Data*: Often, this stand-alone replica resides on a user's PC. This limits the availability of data when synchronizing remotely.
- *Performance Problems when Synchronizing with Multiple Replicas* A handheld that synchronizes between alternate replicas will often have to transfer all of the handheld's data during a synchronization. This can lead to unacceptable performance.
- *Lack of Support for Shared Data Sources* Since users synchronize with stand-alone replicas, current synchronization infrastructure cannot directly support multiple users sharing data.

With increasing wireless connectivity and application data requirements, these problems are becoming unacceptable. Users should be able to synchronize their handhelds from any connection on the Internet, including multiple PCs, wireless networks, and handheld modems. From these access points, users should be able to synchronize with and concurrently update shared data sources with reasonable performance. The Edison service has been designed to address these issues.

3. The Edison synchronization protocol

The problems discussed in Section 2 come from the fact that metadata and data are stored on stand-alone replicas that are not kept consistent with one another. To solve this, we present Edison, a service that utilizes off-the-shelf ORDBMS technology to allow users to synchronize from any network access point. Edison relies on its protocol and ORDBMS transactions to keep metadata consistent, which ensures that only the changed data and metadata is uploaded by the handheld on each synchronization. Since Edison is a networked service, multiple users can share data. When shared data is updated, Edison ensures that these updates are propagated to all interested users on their next synchronization.

Edison also provides an extensibility framework for application developers who need to add synchronization to their applications. Our current Edison prototype is intended for PalmOS devices, but it can be extended to support any record-based synchronization with a similar consistency model.

In this section, we first outline the main components of the Edison architecture, and describe the synchronization protocol itself. We then describe how Edison is deployed to support new synchronization services. Finally, we then discuss the fault tolerance aspects of the protocol.

3.1. Edison components

The basic architecture of Edison is shown in figure 2. Edison consists of a shared data server (DS) that exists on a fixed (wired) network. The handhelds (HH) gain access to this network via *sync nodes* (SNs), which are the access points where synchronization occurs. The components that are new to Edison are the DS and SN nodes; the handheld software remains unchanged. The only impact Edison has on the handheld is that it embeds two identifiers (the “sync matrix ID” and “sync vector ID” described below) in the header information of each HHDB.

Currently, the Edison data server is implemented using a single machine running an off-the-shelf ORDBMS. Since the actual synchronization logic is offloaded to the SNs,

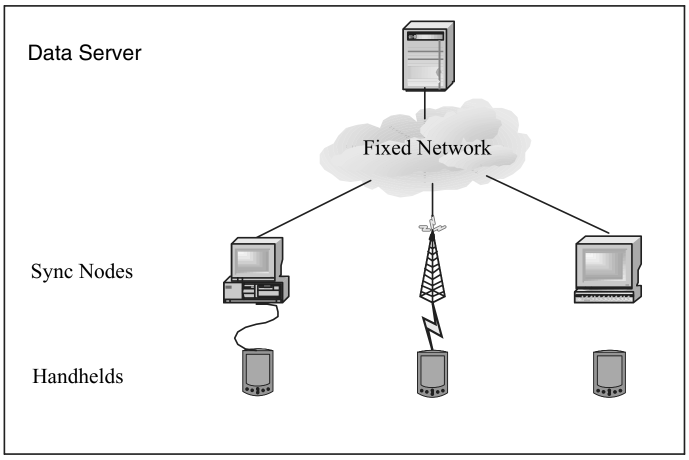

Figure 2. Edison Topology.

the demands on the ORDBMS are not large for this application (data server performance is addressed in Section 4). Furthermore, as described below, we can employ caching at sync nodes to further offload work from the server. Finally, it is important to note that by using an off-the-shelf database system, we can obtain further scalability by leveraging the cluster-based and distributed aspects of the ORDBMS.

3.1.1. Data server. A data server (DS) is a networked, shared data source responsible for storing and maintaining data and metadata for many handhelds. Each handheld can be configured to synchronize all or a subset of its HHDBs with a particular data server.

For each type of HHDB (e.g. Palm address book, datebook, etc.) it supports, the data server maintains a *Master Data Table* (MDT). An MDT contains a replica of each HHDB record from every HHDB of its type that synchronizes with this data server. For instance, an address book MDT would store replicas of the address book entries of all handhelds that synchronize their address book with a given data server.

Each individual record is stored once in the MDT. Handhelds that share a record (e.g., a common date book appointment) use the same MDT tuple as a backing store for that record. If the record is changed at the server, both handhelds will receive a new copy when they synchronize. Likewise, if one handheld changes this record and synchronizes, the change will be sent to the other handheld when that handheld next synchronizes. The MDT can hold data in any format as long as an identifier field (the *RID*) is used to uniquely identify each record within the table, and the application developer provides methods to convert data between the MDT and HHDB record format. MDTs are currently implemented as tables in an ORDBMS, but Edison allows them to point to external data as well.

The data server also stores synchronization metadata for each handheld that uses Edison. The metadata is stored in structures called *sync matrices*. There is one sync matrix for each handheld. Each sync matrix has a unique ID (*sync matrix ID*) and contains a set of *sync vectors*, one for each HHDB on the handheld that is configured to synchronize with Edison. Each sync vector has a unique ID (*sync vector ID*), a monotonically increasing version number, and a set of entries, one for each record in the corresponding HHDB. A sync vector entry contains the status bits for the corresponding MDT record as well as a mapping between the HHrecordUID (i.e., the ID for the record on the device) and the RID for the corresponding tuple in the MDT.

For an application with a large shared data set, any one handheld will likely synchronize with only a subset of the tuples in the master data table. Thus, for each MDT, Edison maintains an *interest table*, which keeps track of which tuples are relevant to each handheld. The interest table contains mappings between HHDBs and sets of MDT tuples identified by their RIDs. We refer to the set of tuples associated with an HHDB as the *interest set* for the HHDB. The Edison API provides hooks to allow applications to specify and adjust the interest sets.

In the current implementation, interest sets are explicitly represented as sets of RIDs. We intend to extend the mechanism to allow interests to be expressed as predicates, which would greatly simplify the job of the application developer. Maintaining such predicates however, is a form of view maintenance; the design of a scalable mechanism to support such views is a challenging aspect of our future work (see Section 6).

3.1.2. Sync node. In Edison, we define a sync node (SN) to be any point on the network where synchronization occurs. Sync nodes can take the form of anything from a desktop machine with a PDA cradle to a process running at a wireless basestation. Typically, the SN is the point where the handheld accesses the fixed network. However, the handheld can communicate with the SN through another network access point if the handheld uses standard network protocols (e.g. TCP/IP).

Sync nodes maintain only *soft-state* data, which means that none of the data used at an SN needs to be maintained durably there—we can always obtain the needed information from the data server.

In order to perform a synchronization for a handheld, an SN must have the most current copy of that handheld's sync matrix. One option is for the SN to download the appropriate sync matrix for every synchronization. Alternatively, sync matrices can be cached at sync nodes. If caching is used, the data server must track which matrices are cached at which sync nodes, so it can propagate changes to the SNs. A "check-on-access" protocol based on version numbers is used to ensure correctness. The server also sends the SN copies of any relevant data records that have been created or modified at the data server or by another handheld. Otherwise, these records would have to be requested from the DS during the synchronization process, reducing the benefits of caching.

3.2. Protocol

Having described the basic components of the system, we can now describe the Edison synchronization protocol. As with any synchronization protocol, Edison must make two guarantees in order to work correctly: the synchronization logic must generate the proper updates for the handheld and the data server, and these updates must be reliably applied to both components. In order for the synchronization logic to generate the correct updates, the logic must use metadata that accurately reflects the state of the handheld and data server. Since the HHDBs that synchronize with an Edison system only synchronize with one data server, the HHDBs' metadata will always reflect the state of the HHDB data since its last sync. Thus, Edison's challenge is to ensure that up-to-date sync vectors are always used on synchronizations.

To ensure that updates are applied to both the handheld and the data server, Edison performs what we call *atomic synchronizations*. We define atomic synchronizations to be those that either apply all synchronization logic updates to the handheld and data server, or none at all. In this protocol description, we show how Edison keeps metadata up-to-date and supports atomic synchronizations. Note that the description in this section only covers normal operation; we defer the discussion of fault tolerance to Section 3.4.

To initiate a synchronization, the handheld connects to an SN and sends its corresponding sync matrix ID. Note that in order to minimize changes to the handheld, Edison follows the approach of the Palm protocol where each HHDB is synchronized in turn by the appropriate conduit. If Edison is implemented as a Palm conduit, this conduit gets the sync matrix ID when the conduit is executed for the first HHDB to be synchronized (recall that the ID is embedded in the header of each HHDB). The SN checks to see if it has a cached copy of the handheld's sync matrix and associated data records. If so, the SN proceeds using its cached copy. Otherwise, it requests a copy of the sync matrix and data from the DS. Note that the DS processes the sync matrix request inside an ACID transaction. This ensures that the sync vector version numbers are consistent with their associated entries and data.

After the SN has the sync matrix, it synchronizes each HHDB in the handheld that is configured to sync with Edison. The process of synchronizing an HHDB can be split into three phases. Phase 1 and Phase 3 occur primarily on the sync node, while Phase 2 occurs on the data server:

Phase 1: Sync Action Generation. Phase 1 is similar to the initial stages of the *fast sync* of the Palm protocol described earlier. The handheld uploads only the status bits and data for the records that have been updated at the handheld since the last synchronization. The sync node then applies the synchronization logic to the sync vector and the uploaded records. Since the master data table resides at the server rather than at the sync node, the sync logic generates a set of "update actions" to be sent to the data server. These actions contain the type of update, the RID and HHrecordUID of the affected record, and the HHDB record data if the record was added or changed at the handheld. These actions are collected into a single SYNC_REQUEST message and sent to the data server. This message also contains the sync matrix ID, the sync vector ID, and version number for the sync vector that was used by the synchronization logic.

Phase 2: Sync Action Processing. Phase 2 takes place at the data server. When the data server receives a SYNC_REQUEST message, it initiates an ACID transaction and performs the following steps inside that transaction. First, it compares the sync vector version number it receives from the SN to the current version number of that sync vector. If the SN's version is out-of-date, the data server ends the transaction, and sends the sync node a SYNC_ABORT message containing a copy of the current version of the sync vector and the modified data records. This version check guarantees that the data server only successfully processes synchronizations that use up-to-date metadata. Since atomic transactions are used when generating sync matrix request responses and SYNC_ABORT messages, the version numbers at the sync node sync vectors are always consistent with their corresponding entries.

If the sync node's sync matrix is up to date, Phase 2 continues. First, the data server applies all the update actions received from the SN to the proper MDT. The data server then checks the corresponding interest table to see which other sync vectors share these updated MDT tuples. The synchronization logic dictates how these sync vectors are updated. For example, for a record that was updated on a handheld, the default synchronization logic clears the handheld's sync vector entry status for that record, and sets the status bits for the entries in all other interested sync vectors to "changed". The data server then increments the version numbers of all involved sync vectors. This process ensures that the update will be reflected in the next sync of the other handhelds that share the record.

The data server gathers all updates to master data table tuples that affect the handheld and temporarily logs them along with an indication that the synchronization has committed. This information is kept for recovery purposes until it is known that the updates have safely been installed on the handheld (Fault tolerance is addressed in Section 3.4.). If the transaction aborts at any time before this information is completely written to the log, the data server aborts the transaction and sends a SYNC_ABORT to the SN as described above. Otherwise, it commits the transaction and sends the SYNC_COMMIT message to the sync node. This transaction ensures that the DS applies all actions from a synchronization atomically to both its data and metadata. Finally, for each of the sync vectors changed during this phase, a CACHE_UPDATE message containing the relevant data and metadata updates is sent to each of the sync nodes currently caching that vector.

Note that the SYNC_COMMIT and CACHE_UPDATE messages are sent *outside* of the transaction boundaries. This behavior is necessary to reduce concurrency conflicts at the data server; the DBMS only holds locks during Phase 2, and not during the entire synchronization. Correctness is enforced through the version checking mechanism, which ensures that SNs use up-to-date metadata during Phase 1. The version checking mechanism allows Phase 1 itself to occur outside of a transaction. If, while an SN is executing Phase 1 for some HHDB, conflicting updates are installed at the data server, the version checking mechanism will detect the problem when the SYNC_REQUEST arrives. In this case, the DS will send a SYNC_ABORT message containing the new information to the SN.

Phase 3: Sync Response Processing. If the sync node receives a SYNC_ABORT message, it updates its sync matrix with the new sync vector, installs copies of any new data records, and returns to the beginning of Phase 1 without fetching records from the handheld again. If it receives the SYNC_COMMIT message, it sends the updated HHDB record values to the handheld and expunges the HHDB records that are marked deleted. Since the handheld does not allow updates to its data while synchronizing, the updates in the SYNC_COMMIT message are applied atomically to the handheld provided that no failures occur. The sync node finishes the sync by sending a SYNC_END message to the server, which tells the DS that it can clean up its log. The DS acknowledges this deletion with a SYNC_END_REPLY.

After all HHDBs have been synchronized, the sync node has the option of either discarding the sync matrix and data records or caching them and receiving propagation messages. Recall that even if the SN decides to cache them, they are considered to be soft-state; the SN can discard them and request new replicas at any time.

In addition to updates from handhelds, updates may originate from other sources. For instance, users will want to make updates to their groupware calendar from their desktop applications as well as their PDAs. When an update occurs from an external source, the DS applies this update with a modified form of the transaction from Phase 2. Inside the transaction, the update is treated as a single action, and the DS updates both the data and the metadata for all handhelds that share this record. Since the transaction does not involve a handheld, the version check and logging are omitted. The next time that a HHDB that shares this data synchronizes, its metadata will reflect this change.

3.3. Application deployment

In order for Edison to be successfully deployed, application developers must be able to easily integrate new and existing applications with the system. Edison's extensibility framework reduces these tasks to implementing a few functions and creating a few database tables.

Using this framework, the authors were able to add data server support for a groupware address book and datebook for Palm with about half a day's work.

To enable synchronization services at the data server, an application developer creates the appropriate master data tables and the interest tables. The developer is responsible for populating the interest table with the appropriate mappings.

To support application-specific synchronization logic, the developer must define two methods at the data server for each data table: doAction() and changeSVEntries(). DoAction() is called once per action during Phase 2 processing. It takes as arguments, the action type and any data needed for this action. This function is responsible for applying the action to the corresponding master data table. DoAction() performs any needed conversions between the HHDB and MDT formats, as well as any interaction with external data sources. DoAction is also responsible for updating the interest tables if updates to a record can change a user's interest in the record. ChangeSVEntries() is called once per action per interested sync vector in Phase 2. It is responsible for making the appropriate update to the corresponding sync vector entry. ChangeSVEntries() also generates the entries in the CACHE_UPDATE messages for sync vectors that are cached at sync nodes.

At the sync node, the developer needs to replace one method per HHDB type to extend the sync logic. This method takes an HHDB record and its corresponding sync vector entry, and returns the action generated, if any. Note that this method can return any action type, so long as the corresponding doAction() and changeSVEntries() methods to support it have been implemented at the data server. This method also takes an optional argument, which is another set of status bits. This argument is used to provide a modified synchronization logic used in failure recovery, as described below.

3.4. Fault tolerance

The previous subsections described how, in the absence of failures, Edison ensures the use of up-to-date synchronization metadata and provides atomic synchronizations. In this section, we describe how Edison copes with failures during the synchronization protocol. We proceed by examining the impact of failures during each of the three phases of the protocol.

In Phase 1, no modifications are made to the data or metadata on the handheld or data server. Thus, only the sync node can be affected by a failure during this phase. Since the sync node keeps only soft state, it can recover from a failure by requesting a new sync matrix from the data server.

In Phase 2, all SYNC_REQUEST processing runs in an ACID transaction; the data server atomically applies all of the actions in the SYNC_REQUEST message to its data and metadata. If the data server fails, we depend on the DBMS's transaction mechanism to ensure that any partially completed SYNC_REQUESTs are rolled back. The corresponding sync node will then re-send the SYNC_REQUEST as described below.

Recall that while the updates on the data server are applied in a transaction, the transaction does not include the sending of replies to any of the sync nodes. This is a key performance optimization that limits the scope of such transactions, thereby improving concurrency. In particular, we do not want a transaction to block on the completion of Phase 3 because transactions would then include the relatively slow communication between the handheld and the sync node.

This optimization, however, opens up a window of vulnerability. Recall that after preparing the SYNC_COMMIT message to send to the sync node, the data server logs the MDT record inserts, updates, and deletes that need to be propagated to the handheld before committing the transaction. Only then is the message sent to the sync node. If a failure occurs after this point, it is quite possible that the necessary updates do not make it to the handheld, or even to the sync node.

To ensure that a handheld receives the needed updates, Edison uses these logged updates at the DS. Recall that the data server removes these updates in Phase 3 only after the SN successfully updates the handheld and sends the SYNC_END message. If the DS receives a SYNC_REQUEST for an HHDB for which there exist logged updates, the DS sends these updates to the sync node. These updates are sent regardless of the version check result in Phase 2, and the data server also sends sync vector and data updates to the sync node if the version check fails. This process ensures that the sync node receives all updates for this handheld that were committed in Phase 2 of the failed synchronization but may not have been applied in Phase 3.

Unfortunately, the sync node cannot simply replay the logged updates in this situation because the user could have performed additional updates on its HHDB records since the failed synchronization. These updates could be overwritten if the logged updates were just blindly applied. Also, the current sync vector for this HHDB may now reflect updates that should have precedence over the updates in the logged records. When the sync node receives logged updates in response to a SYNC_REQUEST message, it must run a modified form of the synchronization logic. This synchronization logic generates actions based on three pieces of information for each record: the status bits in the current sync vector entry, the HHDB record's status bits, and the logged update if one exists.

The recovery synchronization logic works as follows. If the sync vector entry status bits are not clear, the sync vector entry takes precedence over the logged update and the logic runs just as the default synchronization logic. If the sync vector entry status bits are clear, the default synchronization logic uses the logged update type instead of the sync vector entry status bits.

Although the recovery synchronization logic will make the handheld and data server consistent, it differs from the default synchronization protocol in one subtle way. If a synchronization fails in Phase 3 and the handheld synchronizes again, Edison has no way to tell if HHDB record status bits were set before or after the failed synchronization. Regardless, the recovery synchronization logic has to use these status bits. Consider a record where the sync vector entry is clear, an update log record exists, and the HHDB is set to updated or deleted. Since the synchronization logic does not know if the handheld updated or deleted the record before or after the failed synchronization, it generates a change-change or delete-change action, respectively. These actions behave the same as normal change-change or delete-change actions, except that the user will be notified of the delete-change action. If the DS update had been propagated to handheld during the failed synchronization, the handheld subsequently changed the status bits and the conflict message is superfluous. In this case, no data will be lost, and user can always correctly respond to the notification. If the DS update was not propagated, however, the handheld needs the actions to receive the updated record from the server.

After running the recovery synchronization logic, the sync node sends a REQUEST_SYNC with the newly generated actions, and the protocol continues as it would in normal operation. If consecutive phase 3 failures occur, the recovery synchronization logic works correctly. Within the REQUEST_SYNC transaction, the data server deletes all old log updates and generates the new log updates. Since the modified synchronization logic generates actions for any record that needs updated on the handheld, all logged updates necessary for recovery will be generated in Phase 2.

4. Data server implementation and performance

In this section, we first describe the prototype, focusing on the tables used to implement the data server. We then discuss the overhead of this implementation in terms of DBMS storage and synchronization time.

4.1. Data server schema and storage requirements

The Edison data server has been written using a commercial ORDBMS.[8] The schema for the main tables needed to support the data server and a single application is shown in figure 3.

Synchronization metadata is maintained in two tables: the *SyncVectorTable* and *SyncVectorEntryTable*. The *SyncVectorTable* contains a row for each sync vector (recall that there is a sync vector for each HHDB on each handheld supported by the system). This row contains the ID and current version number for the sync vector, the ID of the sync matrix it is a part of, a reference to the corresponding MDT, a count of replicas cached at SNs, and some internal bookkeeping information. The *SyncVectorEntryTable* contains the actual *status bits* for each of the records in the master data table, as well as the mapping between RIDs for MDT tuples and HHrecordUIDs for HHDB records.

In terms of size, the number of rows in the *SyncVectorTable* is equal to the number of HHDBs being supported (i.e., handhelds $\times$ number of databases per handheld). The number of rows in the *SyncVectorEntryTable* is equal to the number of records in all of those HHDBs (i.e., # of sync vectors $\times$ number of records per HHDB). Considering an enterprise-size system to support 1000 handheld users, assuming 4 HHDBs per user and 1000 records per HHDB, we would expect 4000 rows in the *SyncVectorTable* and 4,000,000 rows in the *SyncVectorEntryTable*.

Edison supports synchronization of subsets of shared data sources through the use of interest tables (one such table per shared MDT). In the current implementation, interests are represented explicitly using RIDs. Thus, in this implementation the aggregate number of entries in the interest tables (assuming all MDTs are shared) is the same as the number of entries in the *SyncVectorEntryTable*. As stated in Section 3, we ultimately intend to represent interests as predicates; in this case, the number of entries would be proportional to that of the much smaller *SyncVectorTable*.

In addition to the tables described above, there is an additional table, called the *PushSVLocationsTable*, that is used to support caching. It contains information on which sync vectors

```
CREATE ROW TYPE EdisonDataTableRowType (        ;; all MDT row types must inherit
                                                ;; from this row
        rid INT8 PRIMARY KEY);

CREATE ROW TYPE EdisonInterestRowType ( ;; all interest tables use this row type
        syncVecID INT,
        rid INT8);

CREATE TABLE SyncVectorTable (
        syncVecID INT PRIMARY KEY,
        versionNum INT DEFAULT 1,
        masterDataTblName VARCHAR(128);
        replicaCount INT,        ;; the next replica number to be assigned to
                                 ;; a cached SV replica
        syncMatrixID INT,        ;; must be unique between sync matrices
        nextHHRecordUIDAlloced INT);  ;; the HHRecordUID that the DS uses if it
                                      ;; inserts a record into the handheld

CREATE TABLE SyncVectorEntryTable(
        syncVecID INT,
        versionNum INT,    ;; last sync vector version that updated this sync vector entry
        rid INT8,
        HHRecordID INT,
        StatusBits INT,
        PRIMARY KEY (syncVecID, rid));

CREATE TABLE PushSVLocationsTable (    ;; maps cached sync vectors to IP addresses
        syncVecID INT,
        replicaNum INT,
        ipAddr BIT(32),
        PRIMARY KEY (syncVecID, replicaNum));

;; a sample MDT and interest table used in the tests.
;; for simplicity, sample MDT only has one 2 KB field

CREATE ROW TYPE TestMasterDataTableRowType (
        testBody CHAR(2000))
        UNDER EdisonDataTableRowType;

CREATE TABLE TestMasterDataTable OF TYPE
        TestMasterDataTableRowType;

CREATE TABLE TestInterestTable OF TYPE
        EdisonInterestTableRowType;
```

Figure 3. Sample schema for an Edison system.

are cached at which SNs. Its size is a function of the number of sync vector replicas that are cached at any given time.

Finally, The actual data itself is stored as tuples in an MDT. In general, there will be one MDT for each application type for which the data server supports synchronization. In the example of figure 3, there is a single MDT called *TestMasterDataTable*. Note that if the MDT is private to a single handheld, then there will be one entry in the MDT for each record in the HHDB. In contrast, if the MDT is shared among multiple HHDBs, there is one entry for each record in the *union* (i.e., with no duplicates) of those HHDBs. Thus, the number of entries in the MDT is bound by (and often less than) the number of entries in the *SyncVectorEntryTable*.

4.2. Data server overhead

Given that the storage requirements of the Edison, we now examine the impact of the extra processing that Edison imposes on the overall synchronization time. Recall that a major

Table 2. Edison overhead vs. number of updates (2 KB records).

Number of updates	Palm HotSync time (sec.)	Data server link time (sec.)	Data server time (sec.)
0	3	.25	.13
25	4	.70	.30
50	6	1.31	.44
100	11	2.28	.72

advantage of Edison is that all synchronizations can be done as *fast syncs*. For Edison to be useful, the ORDBMS transactions and the protocol network traffic must not add significant time over current Palm *fast syncs*.

In order to examine the performance of the data server, we constructed a simulation environment that allows us to drive the data server with an artificial synchronization load. The simulation environment models the behavior of the sync nodes and the handhelds, performing the various phases of the synchronization protocol with the data server. We used a network simulator to model a topology in which sync nodes reside on LANs (20 ms latency, 10 Mb/s), which are linked (via a gateway) to the data server by a backbone (10 ms latency, 1Gbps). The data server and simulator was run on a Linux machine with two Pentium III 800 MHz processors and 750 MB of RAM.

We measured the average execution time for the data server and average time spent on the link between the sync node and the data server per synchronization. The sum of these two values represents the overhead that Edison incurs over a current Palm *fast sync*. Table 2 shows these measurements as the number of updates is increased for one set of parameter settings, namely: 1000 handhelds, 2 KB records, 250 records per HHDB, and 1 HHDB per handheld. Also, on average, each MDT record was shared among 5 handhelds (i.e., each update required four other sync vectors to be updated), and sync node caching was turned off for these experiments. The performance results are averaged over approximately 3000 synchronizations. Also shown in the table (for comparison purposes) is the corresponding *fast sync* time for a Handspring Visor with a USB link synchronizing directly with a desktop PC. As can be seen in the table, neither the data server nor the communication between the sync node and the data server take much time compared to the basic *fast sync* time, even over USB.

In our experiments we varied many parameters, including the number of handhelds (from 500 to 5,000), the number of HHDBs per handheld (1 to 4), the number of records per HHDB (50 to 1000), and the size of the records (500 bytes to 2KB). Varying the parameters within these ranges did not change these results in any meaningful way. For example, even with 5000 handheld clients, the data server executes in less than a half a second when 25 updates are performed between synchronizations.

The results of our experiments indicate that the time required by the data server to perform a synchronization and for communication between the sync node and the data server is small compared to the current fast sync protocol. Since Edison does not change the handheld and Edison sync nodes run logic that is very similar to that of the basic Palm protocol, we expect that those components will perform similarly in the Edison environment. Thus, we believe that Edison will be able to provide performance similar to that of the current *fast sync*, even when users synchronize at multiple locations and use shared data. It is also important to note that the current data server is a relatively un-tuned proof-of-concept implementation that runs on one machine. A well-tuned implementation running on a cluster should provide much more scalability, if necessary.

5. Related work

PDA synchronization, while tremendously popular, has received little attention in the academic literature. Thus, much of the related work has been in the commercial realm. As mentioned in Section 2, a growing number of services such as FusionOne [7] and AvantGo [2] are providing web-based synchronization functionality to handheld users. Such companies are too numerous to mention here, and any such list would be almost immediately out-of-date. Few companies, however, currently provide synchronization services for generic shared data sources (e.g., ODBC sources). One exception is Palm HotSync Server [18]. HotSync Server maintains mappings between HHDB records and the backing store records, but the application developer is responsible for ensuring that a handheld is notified of any updates made at the data server between synchronizations. In contrast, Edison provides this functionality by keeping the sync matrix data structures consistent as described in Section 3. As mentioned before, very little information is available on how such services are implemented, and none of it appears in the database literature.

In addition to the PDA and synchronization vendors, most database vendors also provide mobile data management solutions. Handheld clients running a small DBMS can replicate, update, and synchronize with data from a larger server. In the case of Oracle [15], the mobile client and server maintain consistency by shipping transaction logs from the mobile client. Once the server integrates a client's log, the server sends updated data values back to the client. The server rolls back any client transactions that conflict with transactions already committed at the server, and no conflict resolution is used.

Transaction logs can be inefficient if many updates are made to the same item on the handheld. For this reason, Informix's mobile solution uses logs of "work units", which are logical updates to be applied at the server using application-specific methods [9]. Microsoft's mobile database provides synchronization via either log shipping or "Merge Replication", a record-by-record synchronization using conflict resolution [13]. Sybase's Ultralite solution [24] also resolves conflicts at the record level by using user-defined scripts at the database. Unlike Edison, all of these solutions require that handheld data be stored in a proprietary client DBMS.

In the operating systems research community, several projects have studied synchronization for intermittently connected devices in the context of file systems. The Coda file system [10] allows users to replicate files to mobile devices and to update these files while disconnected from the network. Upon reconnection, the clients run application specific resolvers (ASRs) to resolve conflicts. Coda performs synchronizations at the granularity of a file, instead of a record like Edison.

Like Coda, the Bayou file system uses application specific methods to resolve conflicts. However, Bayou is a peer-to-peer architecture, where mobile devices can synchronize with each other as well as any number of replicas on fixed servers. Under this model, a mobile device synchronizing with a fixed server cannot be sure that it has the most recent updates if other devices synchronize with other servers. Bayou resolves conflicts at the level of a write, and Bayou synchronizes files by write log shipping.

The Roma Service takes a slightly different approach to the file synchronization problem. Roma allows users to keep multiple replicas of files on different devices, and keeps track of these replicas using a *personal metadata store*. For each of a user's files, the personal metadata store keeps track of each replica's location and version metadata. This metadata store is used to locate the most recently updated replica of a given file. However, Roma does not guarantee that the client can actually fetch the most updated copy. Also, Roma does not handle files shared between users.

In the database research community, much of the work that is related to Edison comes from the mobile transaction processing literature. The idea of a *tentative transaction* is described in [8]. Tentative transactions are those that can commit locally on the client, but may be rolled back upon reintegration. Tentative transactions are used with log shipping by the Microsoft and Oracle mobile DBMSs as described above. Phatak and Badrinath [19] presents a system where tentative transactions are integrated with the server using application-specific conflict resolvers. Pitoura and Bhargava [21] divides data into clusters, where strong consistency is maintained within a cluster, and weaker consistency is used between clusters. Chrysanthis [3], Yeo and Zaslavsky [26], and Dunham and Kumar [6] execute transactions over wireless links using transaction schemes derived from multidatabase techniques. As noted in Section 1, each of these transaction schemes provides its own level of consistency, which is different than that of PDA synchronization protocols.

The closest database research to Edison is probably the work described in [11]. In this work, mobile clients synchronize with "view holders", which store and maintain view replicas from a server. Edison differs from this approach by only requiring a small amount of metadata instead of materialized views.

Note that Edison's consistency model seems to differ at least somewhat with all systems described here. However, this consistency model is defined by the constraints of the storage and synchronization systems of current PDAs, which are the devices that users use to store disconnected replicas of their data. To our knowledge, no other database research project has examined the flaws in synchronization protocols used by real devices and proposed a solution using off-the-shelf ORDBMS technology.

6. Future work and conclusions

In this paper, we described the current protocols for synchronizing data between PDAs and backing stores, and showed how current techniques of synchronizing with stand-alone replicas places severe limitations on availability, usefulness, and performance. We then presented a new service, Edison, which leverages existing database functionality to provide synchronizations from any Internet access point. Edison does this while ensuring that handhelds only have to upload changed metadata and data on each synchronization. Edison also allows handheld devices to synchronize with subsets of large shared data sources. We described the fault tolerant aspects of the Edison protocol and demonstrated that the use of the data server does not add a significant amount of overhead to the existing fast sync protocol.

In terms of future work, we are currently designing the second generation of Edison that incorporates the following features:

- *Support for Multiple Devices.* Although this protocol is designed for the Palm OS, the basic ideas can be used to implement a device-independent synchronization protocol. Such a system is especially attractive given the emergence of SyncML as a platform-independent synchronization language. For such a protocol, the sync vector entries must be extended to accommodate additional device-specific sync metadata. The challenge in such a system involves efficiently storing, querying, and updating the sync metadata given metadata heterogeneity.
- *Better Support for Shared Data Sources.* In the current implementation, Edison represents interest in shared data sources as sets of RIDs. A major challenge is to extend this mechanism to allow the specification of interests through predicates. We believe that we can build upon existing techniques developed for materialized view maintenance and predicate indexing, as in [1, 11, 20]. However, the challenge is to do this in a way that does not significantly detract from the performance of the data server.
- *Incremental Synchronization.* The current Edison protocol synchronizes entire HHDBs at a time. If a user is synchronizing over a constrained wireless link or stores large records on her PDA, the user may want to perform *incremental synchronization*, which synchronizes only a few records at a time. Given the ability to perform incremental synchronizations, a handheld user may wish to prioritize the data to be synchronized based on network or time constraints, or by other concerns such as data quality. Users may also want different data depending on their current locations and activities.

We plan to solve these problems by building a *Synchronization Broker*, a service which handles all synchronization needs. Synchronization brokers would be responsible for receiving updates from multiple data sources, updating the sync metadata for user devices, and scheduling these updates for synchronization according to user preferences. We believe that adding this level of functionality will greatly enhance the handheld computing experience.

Notes

1. Note that other classes of mobile applications can benefit more from transactional semantics. Mobile transaction models have been developed for such applications (e.g., [5, 9]) but such applications are beyond the scope of this paper.
2. While this description is focused on the basic vendor-supplied PDA synchronization protocols, it is interesting to note that the new industry-wide SyncML synchronization initiative [25] does not address the issue of how to efficiently keep replicas of shared data items consistent.
3. The Palm documentation is somewhat vague about what components actually constitute the HotSync protocol, so for purposes of exposition we will consider the HotSync Protocol to be defined by the combined functionality of these three components.
4. CreatorIDs are guaranteed to be unique to each application by Palm Inc., which keeps a registry of such IDs.
5. For deleted records, the actual data does not need to be sent.

6. We ran similar experiments (not shown here) with the much slower serial link used by most PalmOS devices today. As would be expected, using a serial link results in an even greater difference between slow and hot sync times.
7. This is called the "Combine/Discard" process in the ActiveSync documentation.
8. Since we are showing performance numbers, our license agreement does not permit us to name the system used.

References

1. M. Altmel and M.J. Franklin, "Efficient filtering of XML documents for selective dissemination of information," in VLDB 2000.
2. AvantGo Inc, AvantGo Enterprise Products, http://avantgo.com.
3. P.K. Chrysanthis, "Transaction processing in a mobile computing environment," in IEEE Worskhop on Advances in Parallel and Distributed Systems, 1993.
4. CNNFn.com. Pda sales soar in 2000. http://cnnfn.cnn.com/2001/01/26/technology/handheld/index.htm, 2001.
5. A.J. Demers, K. Petersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and B.B. Welch, "The Bayou architecture: Support for data sharing among mobile users," in 1st IEEE Workshop on Mobile Computing Systems and Applications, 1994.
6. M.H. Dunham and V. Kumar, "Impact of mobility on transaction management," in ACM International Workshop on Data Engineering for Wireless and Mobile Access, 1999.
7. FusionOne Inc., FusionOne Web Site, 2000, http://www.fusionone.com.
8. J. Gray, P. Helland, P. O'Neill, and D. Shasha, "The dangers of replication and a solution," in SIGMOD, pp. 173–182, 1996.
9. Informix Inc., Cloudscape Synchronization Guide, http://www.informix.com.
10. J.J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," ACM Transactions on Computer Systems, vol. 10, no. 1, 1992.
11. S.W. Lauzac and P.K. Chrysanthis, "Utilizing versions of views within a mobile environment," Journal of Computing and Information, Special Issue: ICCI-98.
12. S. Lee, C. Hwang, and H. Yu, "Supporting transactional cahce consistency in mobile database systems," in ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE), 1999.
13. Microsoft Corp., Microsoft SQL Server Hands-On Lab: Replication, http://www.microsoft.com.
14. Microsoft Corp., Mobile Access in Exchange Server Tour, http://www.microsoft.com/exchange/evaluation/ti/tour/MA_1.asp.
15. Oracle Corp., Oracle Lite Replication Guide, http://technet.oracle.com.
16. T.W. Page, R.G. Guy, J.S. Heidemann, D.H. Ratner, P.L. Reiher, G.J. Popek A. Goel̃, and G.H. Kuenning, "Perspectives on optimistically replicated peer-to-peer filing," in Software-Practice and Experience, vol. 28, no. 2, 1998.
17. Palm Inc., Conduit Development Kit, http://www.palm.com/.
18. Palm Inc., Palm HotSync Server FAQ, http://www.palm.com.
19. S. Phatak and B.R. Badrinath, "Conflict resolution and reconclilation in disconnected databases," in MDDS 99, June 1999.
20. S. Phatak and B.R. Badrinath, "Data partitioning for disconnected client-server databases," in MobiDE '99.
21. E. Pitoura and B. Bhargava, "Maintaining consistency of data in mobile distributed environments," 1995.
22. PUMATECH Inc., IntelliSync Anywhere for Lotus Notes, http://www.pumatech.com/.
23. Starfish Software, Inc., TrueSync Technology, http://www.starfish.com.
24. SyBase Inc., UltraLite Developer's Guide, http://sybooks.sybase.com.
25. SyncML, SyncML Synchronization Protocol Specification, http://www.syncml.org/.
26. L.H. Yeo and A. Zaslavsky, "Submission of transactions from mobile workstations in a cooperative multidatabase processing environment," in International Conference on Distributed Computing Systems, 1994.