44

Heterogeneous Synchronization

by

Walter V. vonKoch

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 26, 1998
[June 1998]

Author
Department of Electrical Engineering and Computer Science
May 20, 1998

Certified by [illegible]
John Chapin
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Heterogeneous Synchronization

by
Walter V. vonKoch

Submitted to the
Department of Electrical Engineering and Computer Science

May 26, 1998

In Partial Fulfillment of the Requirements for the Degree of
Bachelor of Science in Computer Science and Engineering
and Master of Engineering in Electrical Engineering and Computer Science

Abstract

Methods for sharing application data are currently more complex than necessary and crude at best. Users may utilize different applications for similar tasks. These applications do not have access to the same data. For example, many email clients do not share address books. Applications store their data in different formats, ranging from ASCII files to transactional databases.

Heterogeneous synchronization is a process for achieving eventual data consistency between these heterogeneous data sources. Mediators are used to translate the data into a generic record-field format for universal access by a synchronization process. The process makes no assumptions about the capabilities of the sources as long as the data can be read. In addition, there are no restrictions on the kind of data that can be synchronized. Each field consists of a name and value. The name is defined by the mediators and implicitly defines the type of the field's value. All operations on the value are performed by the mediators that define a given field name.

The SyncEngine is an implementation of the heterogeneous synchronization process. In the current version the SyncEngine synchronizes contact information between the address books of Eudora Pro and Outlook.

This thesis introduces a novel application of generic data access to weakly consistent synchronization for heterogeneous sources.

Thesis Supervisor: John Chapin
Title: Assistant Professor

Acknowledgements

I would like to thank Prof. John Chapin for supervising this thesis. His input and guidance has put this thesis many times back on track. Thanks go to Michelle Eng for her untiring energy and enthusiasm to push me to finish this thesis. I also would like to thank my parents who have made my studies at MIT possible over the last 5 years. Without them, I would not be where I am today.

Table of Contents

Heterogeneous Synchronization ____ *1*

Abstract ____ *i*

Acknowledgements ____ *ii*

Table of Contents ____ *iii*

List of Figures ____ v

List of Tables ____ v

I. Introduction ____ *1*

II. Related Work ____ *5*

1. Data Sharing Techniques ____ **5**

1.1 Voting ____ 5

1.2 Weakly Consistent Replication ____ 6

2. Data Sharing Systems ____ **7**

2.1 Windows Briefcase Replication ____ 7

2.2 IntelliLink Plus ____ 7

2.3 TSIMMIS ____ 8

III. Design Principles ____ *9*

1. KISS ____ **9**

2. Data Consistency ____ **9**

3. Standard Interfaces ____ **9**

4. Extensibility ____ **10**

IV. Methodology: Heterogeneous Synchronization ____ *10*

1. Overview ____ **10**

2. Sources ____ **12**

3. Mediators ____ **13**

4. Records ____ **15**

5. Named-Fields ____ **15**

6. Local Database ____ **17**

7. Record Equality Testing ____ **20**

8. Heterogeneous Synchronization Process ____ **21**

8.1 Initial Synchronization ____ 21

8.2 Change Identification ____ 23

9. Scheduling of Synchronization Runs ____ **33**

10. Failures ____ **34**

10.1 Read-Only Sources ____ 36

11. Local Database Maintenance ____ **36**

11.1 Database Collapsing ____ 36

11.2 Database Pruning 37

12. Advanced Sources 37

V. Implementation: SyncEngine 38

1. Overview 38

2. SyncEngine 38

2.1 Local Database 41

3. Drivers 44

4. Future Work 46

4.1 Record Comparison via Record Hashes 46

4.2 Callback for Record Comparison 47

4.3 Callback for Find-Similar-Record 47

4.4 Enhanced Links 47

5. Tools and Infrastructure 48

VI. Experiments 48

1. General Validation 49

2. Failure 51

3. Read-Only 53

4. Performance 54

4.1 Measurements 54

4.2 Analysis 57

VII. Conclusions 59

VIII. References 60

List of Figures

Figure 1 - Mediators - Abstract Data Access ____ *12*
Figure 2 - Mediators ____ *14*
Figure 5 – Local Database ____ *18*
Figure 6 - Timeline ____ *19*
Figure 7 - IndentifyChange(SrcRec) ____ *32*
Figure 8 - IdentifyChange(R) ____ *33*
Figure 9 - SyncEngine Overview ____ *39*
Figure 10 - Database Schema ____ *41*
Figure 11 - Scalability of Eudora driver ____ *55*
Figure 12 - Scalability of Outlook driver ____ *56*

List of Tables

Table 1 - Possible Changes ____ *24*
Table 2 - Possible Actions ____ *25*
Table 3 - Required Actions ____ *26*
Table 4 - Identification of Change Types of R ____ *27*
Table 5 - Identification of Change Types of SrcRec ____ *29*
Table 6 - Conditions ____ *30*
Table 7 - Summary of the synchronization algorithm ____ *30*
Table 8 - Sample Records ____ *49*

I. Introduction

Many desktop applications work with similar information but do not share this data with each other intelligently. For example, a fax program has a different address book database than an e-mail application. The user might use one client for business and another client for private email. If both clients shared one address book, the user would not have to maintain two sets of similar contacts. To address this problem some of these software products allow the user to import and/or export data. While this is a good start, a problem arises with data consistency. If some data changes, the change is not automatically applied to the other applications. The user has to manually export and then import the data into all other applications. As can be imagined, this process is unnecessarily complex and time consuming.

The problem of information sharing does not only apply to an application's address-book. Another example is schedule and calendar information. Personal Information Managers (PIMs) are digital versions of Franklin day-planners. A user might use several PIMs for different purposes but would prefer that they share the same calendar. Personal Digital Assistants (PDAs), which are replacing paper based filo-faxes, also contain the user's appointment schedule. Most PDAs let users synchronize data with particular applications, but do not provide a general data synchronization procedure. These three examples suggest that there is a need for a general data synchronization process.

Databases have provided synchronization and replication features for years. They allow tables and views of data to be replicated between two or more replicas. In most cases, replication is used homogeneously, between databases running the same database system software and created by the same manufacturer. The main focus of database replication services is to provide high-speed replication of changes from one database to another. The duplicate databases are often used to improve availability and/or load balancing. In these replication schemes, the source and destination have the same schema and provide proprietary access protocols in order to achieve high performance replication. The database software has replication and synchronization services built-in.

If productivity applications such as PIMs and PDAs used full-featured databases to store information, they could rely on the provided replication features to share information with each other. First, every application uses its own proprietary schema, they could store their data in one central database and there would be no information sharing problem. This solution is not realistic. Second, even Microsoft, with its influence on personal computer operating systems and its huge market share in applications, has been unsuccessful in establishing one central database for contact information. The Windows-Address-Book attempts to consolidate address book information in one place, but even Microsoft applications do not exclusively use the Windows-Address-Book. For example, Microsoft Word does not use the Windows-Address-Book for the envelope-printing feature.

Another attempt to consolidate application data storage is the Windows Registry. Windows 95 and Windows NT 4.0 both provide a central hierarchical database, namely the Windows Registry. It contains information from configuration information for the operating system to application specific information. However, the Windows Registry lacks structure and a predefined standard. Applications write data to the registry in their own format. Hence, applications cannot share information intelligently with each other.

An external general-purpose synchronization engine that synchronizes and replicates information between many applications in a user-friendly fashion would be a better solution to this problem. This is the solution described in this thesis.

The engine should periodically (or on user demand) be able to scan the different applications for new and updated information. It should also be able to transfer the changes to all other applications. This process should be as seamless as possible and require only minimal user interaction. Unfortunately, a synchronization engine in this environment has no internal knowledge of the application's data storage policy, nor can the engine rely on the application for support. For example, in a database environment, the replication engine can intercept transactions to identify which information needs to be transferred to the other replicas. A general-purpose synchronization engine is unable to hook into the internal workings of existing applications so it cannot observe changes as they happen.

One approach for the synchronization engine would be to supply a software developers kit (SDK) to application developers. The developers would use the SDK in order to allow information sharing between applications. While this could theoretically work, it is not feasible. There is no incentive for application developers to utilize the synchronization SDK because it does not add any value to their product. Instead, it makes it easier for the user to switch applications, decreasing the application's customer base. From the developer's perspective, if the user cannot access the information from another application, the user is less likely to switch and re-enter the information into the new application.

Not only must the synchronization engine operate without support from the applications, it also cannot assume that all applications support a minimum set of capabilities. For example, some applications support last names up to a maximum of 30 characters, while others support 50 characters. The engine needs to be aware of the limitations of each application and take those into account when synchronizing data among software products. In addition, an application might use Unicode encoding for text strings instead of ASCII encoding. Hence, the synchronization engine needs to translate between Unicode and ASCII strings in order to synchronize correctly.

One approach is to use the lowest common denominator. For example, one could synchronize 30 characters of text if that is the smallest limit of all of the software products. This limitation seems overly restrictive, since it should be possible to share data of greater length between applications that support longer fields.

Another restriction that should not limit synchronization is the engine’s potential inability to update one application's information, for example if a file is read-only or the application does not provide an interface to update its information. The synchronization engine will not be able to update that particular application's information, but it should be able to propagate updates among other applications. If a temporary restriction that prevents updates is lifted in the future, synchronization should proceed as if the restriction had never been in effect.

The SyncEngine described in this thesis addresses the problem of synchronizing information between different applications. The engine synchronizes data between heterogeneous applications, applications that provide different capabilities and have different restrictions. The SyncEngine operates without resorting to a lowest common denominator solution. The engine synchronizes data in a generic fashion: it makes no assumptions about the data except that the data can be accessed in a record-field fashion. The engine relies on mediators that provide access to the data in a common generic record-field format. The SyncEngine has been tested for synchronizing the address books of Eudora and Outlook.

II. Related Work

Some of the previously mentioned products, e.g. Outlook and Schedule Plus, have features to import information from other sources. These features are crude at best. They require the user to "clean up" the imported data and to delete duplicate records. This manual process is not suited for continuous data synchronization.

The following sections describe database replication schemes which are more relevant to this project. Most of the schemes add functionality to the database itself, which is not possible in this project. However, these examples give insight into mechanisms for synchronization, replication, and heterogeneous data access.

1. Data Sharing Techniques

1.1 Voting

A well-known replication method is voting [2], which is a protocol for maintaining consistency of replicated data. A read or write collects a quorum of votes, which are assigned to each copy of a replicated data item. These read and write quorums of votes must satisfy two constraints. First, quorums must intersect, guaranteeing that any read quorum has a current copy. Second, write quorums must intersect, imposing an order on updates. Voting masks failures, with no need for intervention to resume operation. However, resilience to failures is achieved at a high read/write cost since at least half of all nodes need to participate in each data access in order to partition them into majority and non-majority partitions. These ideas could be applied to the SyncEngine. It would not need its own database to store the current state. Instead the engine could read or write from or to a quorum of applications respectively. However, this approach is slower than writing to its own local database, so the design implemented in this thesis uses a local database.

1.2 Weakly Consistent Replication

The anti-entropy algorithm [13] also relies on weak consistency requirements. The algorithm supports arbitrary communication topologies with incremental replication progress, while guaranteeing eventual consistency. Replication takes place between two replicas that exchange write operations. Any database state can be reached by successively applying the write operation from an initial start state. A write log is stored with each database and contains all writes received by applications or other servers. It is easy to identify conflicts. If a write is not in one of the replica's logs, it is transmitted to that log. The storage and communication is very efficient.

A nice feature of the anti-entropy algorithm is that it allows incremental progress. If the replication process is interrupted, it can be restarted and finish from where it halted. This feature is very appealing to the SyncEngine, whose data sources can go off-line at any time. This property also allows replication over low bandwidth connections or even connectionless media like floppy disks. It is easy to support disconnected sources, e.g. mobile personal digital assistants or applications connected via the Internet. However, the anti-entropy algorithm does not provide the means to identify changes in application data sources. One possibility is to restrict the SyncEngine to data sources with change notifications or to implement a change-identification mechanism.

In this thesis, the SyncEngine implements change identification without application support, and uses a write propagation algorithm similar to but much simples then the anti-entropy algorithm.

2. *Data Sharing Systems*

2.1 Windows Briefcase Replication

The Windows Briefcase (part of Microsoft Windows 95 and Windows NT 4.0) allows application-independent replication. It replicates files between two locations. The user creates a copy of a file in a different location and the operating system tracks if either copy has been changed. Later, the user chooses to synchronize the two copies. If only one field has changed, it is copied to the other location. However, if both files have changed, the operating system leaves it to the user to resolve the conflict.

The Windows Briefcase process is very easy to use, but has many obvious drawbacks. It requires user interaction, and it is limited to a file by file synchronization. Synchronizing data such as address books requires automatic record by record synchronization, so the Windows Briefcase mechanism is not appropriate.

2.2 IntelliLink Plus

IntelliLink Plus for Windows by IntelliLink Corp. [7] goes beyond import and export to perform data transfers between Microsoft Schedule+ and a host of additional PC and handheld applications. It allows the user to import and export contact information between a variety of programs. The idea is similar to the SyncEngine in that it tries to keep several applications' data consistent. However, IntelliLink's approach is different. IntelliLink Plus requires the user to initiate the import/export from within Schedule+. The user is expected to export to the other applications whenever he or she updates information in Schedule+. Similarly, he or she has to import from another application when that application's data changes. To replicate the changes to yet additional applications, the user has to export to all of the desired applications. The process is cumbersome and requires effort on the user's part. There are no conflict-resolution schemes implemented in IntelliLink Plus; instead, the user is asked to resolve conflicts manually for each conflicting record. In contrast, the goal of the SyncEngine is to decrease the amount of user interaction. Ideally, a change made in one application should be automatically replicated to all other applications.

2.3 TSIMMIS

The TSIMMIS project [1] developed general techniques for accessing heterogeneous data sources. Each data record is converted to a common representation for data comparisons and manipulations. The conversion is performed by a software layer between database and data client called *Mediators*.

Mediators perform data translation and field masking. Mediators can also perform data manipulations depending on the client's requirements. The SyncEngine borrows the idea of data translation from TSIMMIS, but does not implement fully functional mediators as defined by TSIMMIS. The SyncEngine's application specific drivers will perform translation and data manipulation. However, the translation routines will be hard coded. In the future, the manipulation routines should be downloaded from a user defined location to the SyncEngine, so that they can be changed dynamically.

One shortcoming of the TSIMMIS system with regard to heterogeneous synchronization is that the capabilities and constraints of the original data source are not exposed by the mediators for use by the generic database engine. The SyncEngine needs to extend the TSIMMIS mediator concept to provide this information, since the SyncEngine needs to identify changes and merge fields in case of synchronization conflicts. In TSIMMIS the mediators hide the differences of the sources. The SyncEngine must be aware of the difference. For example, first name fields could be up to 30 characters in one source and up to 50 in another. The SyncEngine must be aware of the max character length when comparing first names from both sources.

III. Design Principles

1. *KISS*

One of the goals of this project was to create a generic and working SyncEngine. A generic design risks over-design and incompletion. The fundamental design principle to prevent over-design is the 'KISS' principle, or "keep-it-simple-stupid." When in doubt, the simpler option was chosen in the design of the SyncEngine.

2. *Data Consistency*

Database replication technology usually focuses on performance and transactional consistency. Replication products compete to minimize the time required to establish consistency between replicas. However, the primary concern of heterogeneous synchronization is not performance. Instead, the objective is to store the largest possible union of data from several sources in each source. The main design goal should be data consistency, since heterogeneous synchronization's goal is to store a consistent set of data in several sources. High performance is desirable but secondary. The SyncEngine implementation is targeted at the small data sizes characteristic of contact information making fine-tuned performance secondary.

3. *Standard Interfaces*

The design of the SyncEngine follows existing componentization guidelines. In particular, the interface of the mediators follows the Automation guidelines [10] of Microsoft's Component Object Model (COM) [14]. The guidelines define argument data types and suggest calling semantics. A feature of the Automation guidelines is the dual interface specification that allows late and early binding. The key benefit of the Automation guidelines is that they define a binary calling conventions, which is supported by several programming languages. For example, Visual Basic (VB), Java Script, Visual J++, Visual C++, and VB Script all support it. Hence, the mediators can be written in any language that supports the Automation specification. For instance, it is considerably simpler to access Microsoft Schedule Plus from Visual Basic than via C++. The Automation specification allows implementing the Schedule Plus driver in Visual Basic even though the main SyncEngine is written in C++.

4. Extensibility

The heterogeneous synchronization process should be extensible. Sources should be allowed to be added at a later time without requiring changes to the existing implementation. Similarly, the kinds of information synchronized and their data types should be extensible.

IV. Methodology: Heterogeneous Synchronization

1. Overview

This section describes the algorithm developed for heterogeneous synchronization, independent of any implementation issues. Section V illustrates the issues of the SyncEngine implementation.

Abstractly speaking, heterogeneous synchronization takes data from several sources, creates the union of this data, and stores the union back to every source. However, a source might not be able to store all types of data. In this case the goal is to store the largest possible subset of the overall union in that source. After a synchronization run, all sources should contain logically the same information. If information in any source is updated, added, or deleted, the next synchronization run will apply the same changes (additions, deletions, and updates) to all sources. Hence, after each synchronization run, the information in all sources is identical to the extent allowed by the capabilities of each source. This outlines the overall goal of the synchronization process.

Heterogeneous sources make the synchronization process difficult. The synchronization algorithm can make no assumptions about the capabilities of the sources, except that they support some method of data access that enables an external program to read the information stored inside the source. There can be complex situations where one source can be updated, another source is read-only, and a third source can only store a subset of the information.

The heterogeneous synchronization system consists of several components. As in the TSIMMIS system, the synchronization system needs a translation layer. Recall that the mediator in TSIMMIS generalizes the access to the source's data, as well as the data itself. The access to the information needs to be normalized because each source has a different information access interface. Further recall that each source needs a specific mediator. See Figure 1.

The information normalization property of the mediator allows the synchronization process to work with the data in an abstract fashion, without knowledge about its type or interpretation. The goal is to be able to synchronize any kind of data.

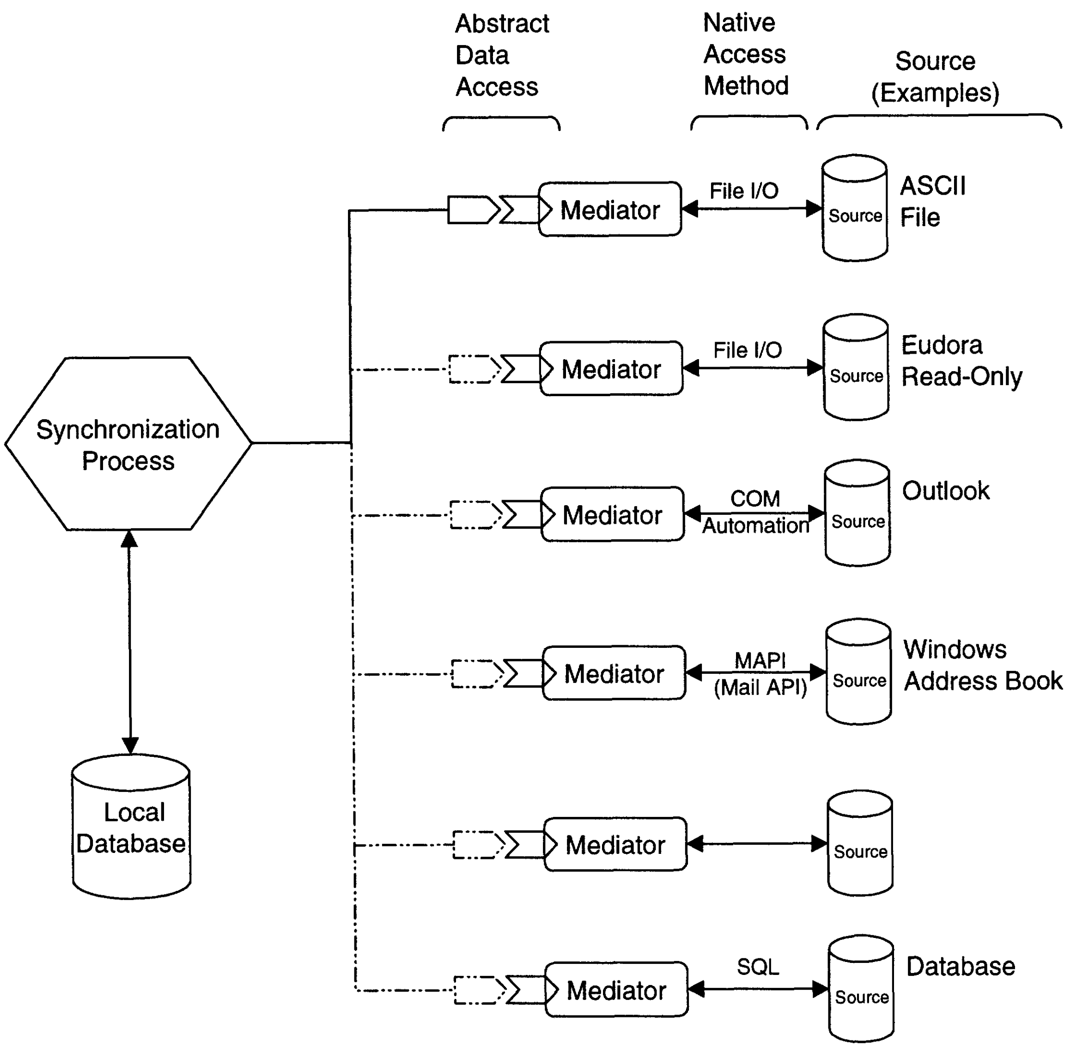

Figure 1 - Mediators - Abstract Data Access

2. Sources

The system synchronizes data between several sources. (For notation let these be sources A, B, and C.) Each source can be a file, or an application that provides methods to access its data. The exact makeup of the source is irrelevant to the synchronization process, because each source is accessed though a mediator specific to the source. If an application can store data in several places (i.e. files), then a customized mediator for each location (file) is needed.

3. Mediators

Each source is accessed via its mediator, making the words mediator and source synonymous from the perspective of the synchronization algorithm. (For notation let these mediators also be A, B, and C.) Each mediator exports the source's information as an unordered collection of records. The mediator assigns a unique identifier to each record it exports (see section 0 for details). The identifier is unique in time and unique with respect to that mediator. The mediator supports retrieval of individual records by identifier and iteration over the collection of records.

Each mediator defines a collection of named-fields it supports. (For the definition of named-fields see section 5.) The names of the supported named-fields are also exported as a collection, which can be accessed by iterating over it. Each mediator may support different named-fields. The following example clarifies the definition of the intersection of supported field names between sources.

- Let mediator A support fields named N, O, and P.
- Let mediator B support fields named O, and Q.
- Let mediator C support fields named N, O, P, and Q.

The intersection of supported named-fields between mediator A and B are fields named O. The intersection for mediator B and C are fields named O, and Q. Finally, the intersection between mediators A and C are fields named N, O, and P.

If all three intersections were empty, meaning no field name was shared by any pair of mediators, then there are no fields that can be synchronized between sources.

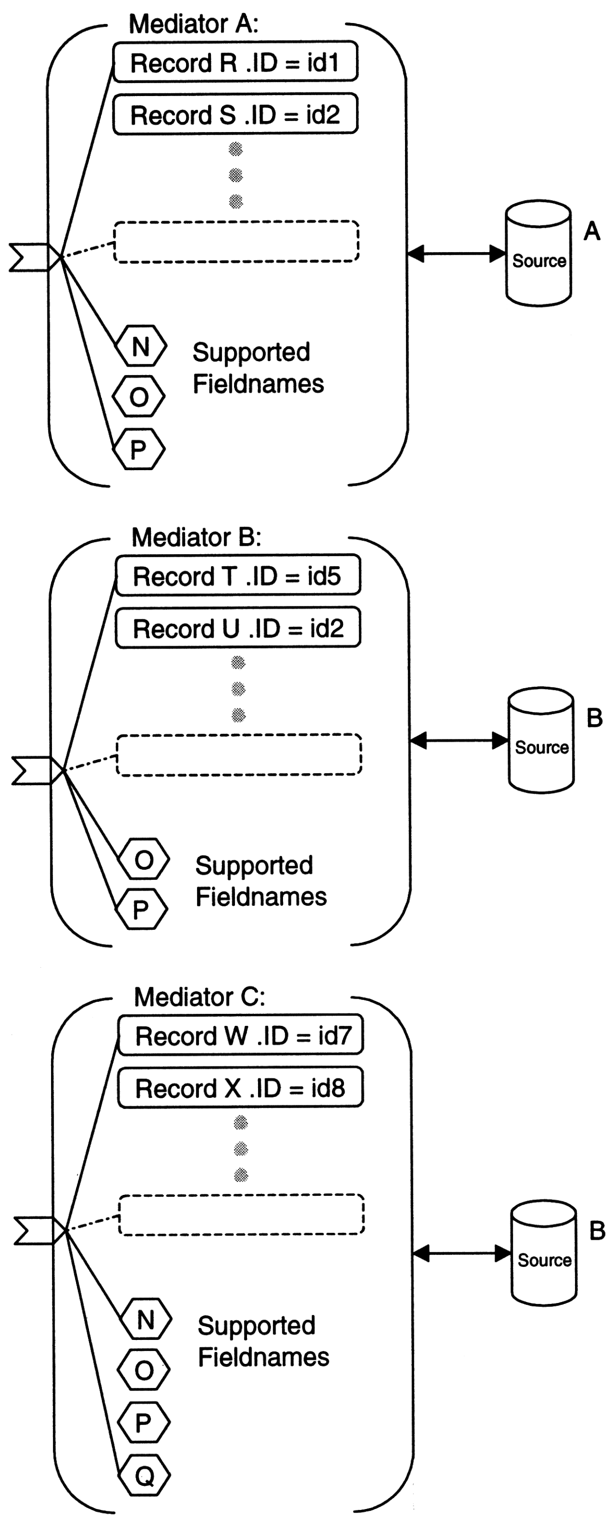

Figure 2 - Mediators

4. Records

The mediators export the source's data as records. (For notation let these records be R, S, and T.) Each record consists of an unordered collection of named-fields and a unique identifier assigned by the mediator as described above. The individual fields can be accessed by iterating over the named-field collection or by selecting a field by its name. There are no restrictions on the number of named-fields with the same name that can be in each record.

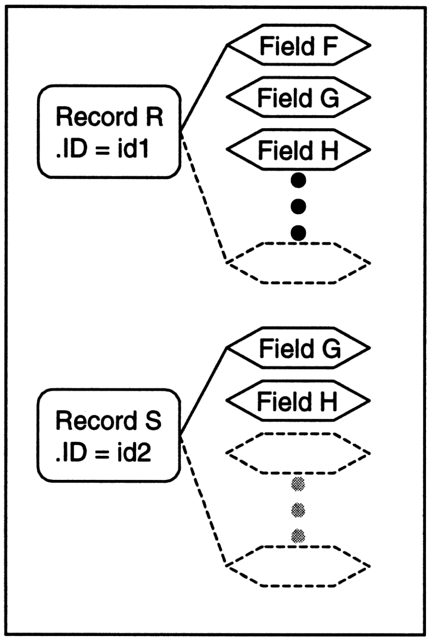

Figure 3 - Records

5. Named-Fields

A record consists of a collection of named-fields. (For notation let these fields be F, G, H and I.) Each named-field consists of two properties, a name and a value. (For notation let these names be N, O, P, and Q, and let the values be V1, V2, V3, ...)

Each mediator declares a collection of field names. Each field name implicitly defines the type of the value property of the named-field. For example, mediator A declares field name "Email" as a field with the value property being a zero-terminated string in ASCII encoding. The synchronization algorithm assumes that all mediators that declare the field name "Email" agree on the value type. The algorithm makes no attempt to ensure that the mediators are in compliance. The assumption is safe, since name clashes are unlikely if only mediators from one origin are used in any one system. For example, the implementers of the Eudora and the Outlook mediators have a common interest to avoid name clashes, namely to synchronize between Outlook and Eudora.

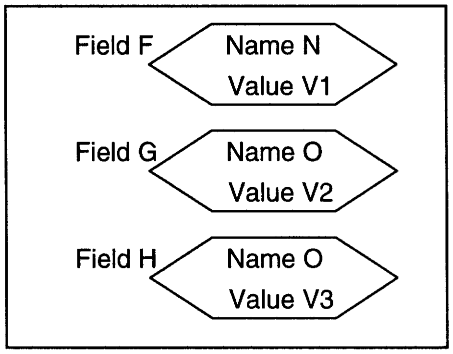

Figure 4 - Named-Fields

This design makes the synchronization process extensible. If new kind of information should be synchronized, the mediators responsible for this new information must define new named-fields and agree on their value representation. For example, if postal addresses should be synchronized, the mediators that synchronize addresses must agree the following field names: "Street Name", "City", "State", "Zip Code", and "Country". The mediators must also agree on the type of the value property of each field. An example for the type of the value property could be Unicode strings. But alternatively, the type of the value property of the field named "Zip Code" could be a binary encoded decimal. The synchronization process does not put any restriction on the choice of field name or representation.

Each record exported from a specific mediator contains only fields named with one of the names supported by that mediator. For example, if mediator A supports field names N and M, then all records exported by mediator M contain only fields named N or M. However, if a record is sent to a mediator, that mediator is required to accept records with fields having any name, not just the ones supported by the mediator. The mediator is expected to disregard the fields with names that it does not support, meaning that it treats these records as if they did not contain any fields with names different from the ones the mediator supports. This feature allows mediators to be extended in the future to support other field names. It is important that mediators disregard any field with names that it does not support, otherwise it would start performing operations on data whose type it does not know. This could lead to corruption. For example, it might look at a field's value as of type ASCII string, when in fact it is a binary encoded decimal.

6. Local Database

The synchronization algorithm requires a local database in order to determine which, if any, records have changed in a source. The functionality of the database is defined in this section; section 8 describes how the algorithm uses these features. The local database stores records (including their unique identifiers), status and timestamp for each record.

The unique identifier assigned by the mediators are used to establish a logical link between equivalent records in different sources. For example, assume that the contact information for "Bill Clinton" is stored in sources A and B. In source A, it stored as record R and in source B as record S. Let R have identifier R.ID and S have identifier S.ID. Logically they both represent the same information, namely the contact "Bill Clinton." The identifiers R.ID and S.ID together form a logical link between the same contact "Bill Clinton" across source boundaries. If the contact is updated in source A, the identifier link {R.ID, S.ID} can be used to determine that record S in source B needs to change as well. The exact process is explained later.

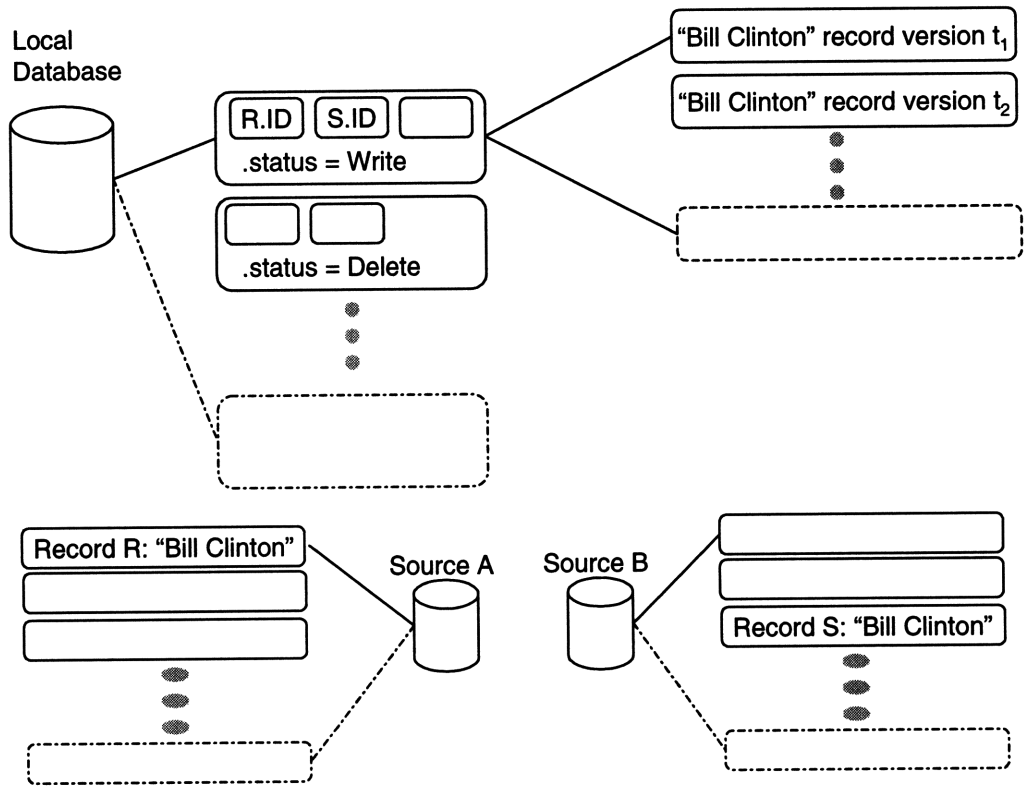

Figure 5 – Local Database

The local database also stores a status field for each logical records that helps when a records is missing in a source, because the synchronization process does not know if the record has been deleted from the source or if the record never existed in the source. For example, let the contact "Bill Clinton" be record R in source A. Another source C does not have a contact "Bill Clinton." This information is not sufficient to determine if source C never had a contact "Bill Clinton" or if it used to have a contact "Bill Clinton" that has been deleted. This is important, since if the contact has been deleted in C, then the record R in source A must be deleted as well. If C never had a contact "Bill Clinton," then the contact must be created in source C by adding the record R from source A to source C.

To resolve the creation-deletion ambiguity, each record stored in the local database has a status R.status. By default R.status = Write, meaning that the record exists. If the record has been deleted it is marked as R.status = Delete, which means that the record has been deleted from at least one source and should be deleted from the others.

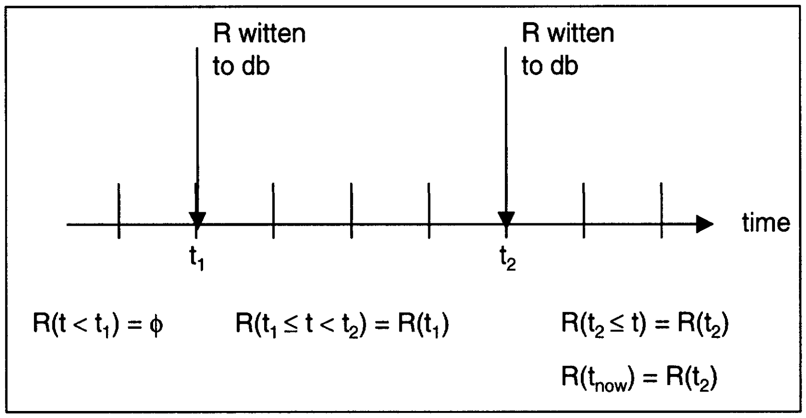

Figure 6 - Timeline

The synchronization process must also determine whether a record has changed in a source. To make this possible, the local database needs to keep track of all changes to each record. Assume that a record R was stored in the local database at time t_1. At time t_2 the same record is updated. Instead of overwriting the record $R(t_1)$ with the new information, a new record $R(t_2)$ is added to the local database. Both versions of record R can be retrieved from the local database by specifying a time for the record. For example, $R(t_1)$ will be retrieved if a time t is specified with $t_1 \leq t < t_2$. Similarly, $R(t_2)$ will be retrieved if a time t' is specified with $t_2 \leq t'$. If there is no version of record R with time greater than t_2, then time t' does not have an upper bound, i.e. $t_2 \leq t'$. Obviously, the retrieval of $R(t_0)$ with $t_0 < t_1$ will fail, since there exists no version of record R at any time before t_1. For notation, the latest version of a record is named $R(t_{now})$ since all version have times smaller than the current time by definition. For an illustration see Figure 6.

Unfortunately, the local database can grow large if the information changes often. A proposal for reducing the number of versions of records stored the local database is described in section 11.

7. Record Equality Testing

The synchronization process tests records for equality as part of change identification. The tested records are considered to be equal if they contain the same number of fields and the fields are pair-wise equivalent. Pair-wise field equivalence means that the fields in the first record and in the second record have the same field name and the same field value.

The values of two fields cannot be compared directly, since the data type of that value is unknown to the synchronization algorithm. Only the mediators that support a specific field name know what type it is. For example, the value of field name N might an ASCII string while the value of field name O might be a UNICODE string. Similarly, the value of field name O might be an ASCII string whose capitalization does not matter, meaning that "xyz" is equivalent to "XYZ".

Therefore, each mediator must provide an equivalence-testing function for each field name that it supports. The synchronization process can then use these functions to compare two fields and thereby test the equality of two records. The synchronization algorithm calls on the mediator that it is currently synchronizing with to perform the record comparison. It is of course necessary that all mediators supporting the same field name provide logically identical equivalence-testing functions for that field's values.

8. Heterogeneous Synchronization Process

Now that the components have been defined, the actual synchronization algorithm is explained in this section. As stated earlier, the goal of synchronization is to merge all records from all sources into each source and to keep them consistent. The synchronization process eventually achieves consistency. Eventually means that the algorithm guarantees consistency if a synchronization run with all sources completes in which no changed records were identified. This method is called eventual consistency [13].

Unlike in [13], in heterogeneous synchronization the limitations of a given source may prevent it from storing all records of all sources. Therefore, we use a more relaxed requirement than complete consistency, requiring instead that the largest possible fraction of the union (of all records from all sources) is stored in each source. More formally:

- Construct union U = records from A $\cup$ records from B $\cup$ …
- A stores largest possible $V_A \subseteq U$ given the limitations of A
- B stores largest possible $V_B \subseteq U$ given the limitation of B

Note that $V_A \neq V_B$ if the limitations for A and B are different.

We call this *heterogeneous consistency.*

Once heterogeneous consistency has been initially achieved, the sources need to be kept consistent. When the user changes a record in one source, the SyncEngine should propagate the change to all corresponding records in all other sources. The following paragraphs outline the synchronization algorithm that achieves this goal.

8.1 Initial Synchronization

The synchronization algorithm processes each source one by one, starting with source A, then source B, and so forth. The first time the synchronization process is started, all records from a source A are stored in the local database. They are stored along with their identifiers R.ID, which are created by the mediator for source A. Next, all records from a source B are similarly stored in the local database.

At this step one could search for "similar" records, link them, and avoid duplicates. Consider what happens if both sources A and B contain a record for "Bill Clinton". Let "Bill Clinton" in source A be record R and let "Bill Clinton" in source B be record S. One could devise an algorithm that finds that R and S are conceptually the same record, except that they originate in different sources. These "similar" records could be linked in the local database via their unique identifiers. This would reduce the number of duplicates of the same contact "Bill Clinton" in each source. However, this optimization is not implemented in the current version of the SyncEngine. Not searching for "similar" records means that the user has to manually detect the duplicates and merge or delete them. Consequently, the algorithm is not as automatic as it could be.

After all the records from sources A and B have been stored in the local database, the records from the remaining sources are similarly stored in the local database. As a result, the local database contains the union of all records from all sources. Next, all records in the local database that are not in a particular source need to be written to that source.

After adding all records that do not exist in a source to that source, it is now consistent with the union of all records from all sources. This process is continued until all sources are consistent with the union of all records from all sources. Hence, the sources eventually reach heterogeneous consistency. See section 10 for what happens if a failure happens before all records have been be added to a given source.

8.2 Change Identification

Before changes in a given source can be applied to other sources, the algorithm has to determine what changes happened in the given source. This section explains how the synchronization process determines which records have changed in a given source as well as which changes from other sources need to be applied to this given source.

In the following description, SrcRec refers to a record from a given source and record R refers to the corresponding record from the local database. The correspondence is determined by the identifier link which is stored in the local database. Recall the explanation on identifier links in section 6.

8.2.1. Possible Changes

The information in the sources can change over time as the user works in different applications. The changes can be updates, additions, or deletions of individual fields or entire records. The records in the local database change because the changes in another source are applied to the corresponding record in the local database.

The synchronization process looks at each source individually. The possible changes are the cross product of the changes to a SrcRec and the changes to the corresponding record R in the local database. Table 1 outlines all possible changes. Some permutations are marked as N/A for not available, since these permutations cannot occur in the synchronization process. The reasons are explained below.

Possible changes since last sync with given source		Type of change to SrcRec				
		not exist	no change	add	delete	update
Type of change to R in local database	not exist	N/A 1	N/A 2		N/A 1	N/A 2
	no change	N/A 5		N/A 3		
	add		N/A 4	N/A 3 N/A 4	N/A 4	N/A 4
	delete	N/A 5		N/A 3		
	update	N/A 5		N/A 3		

Table 1 - Possible Changes

N/A 1: If a record exists neither in the source nor in the local database, then it does not exist; the synchronization process does not need to synchronize it.

N/A 2: If there is a source record with no corresponding record in the local database, then this source record must have been added since the last sync with the given source. Therefore it cannot be in the No Change or Update states.

N/A 3: If a SrcRec has just been added to its source, then SrcRec cannot have a corresponding record in the local database. As previously mentioned, one could conceivably search for a record "similar" to SrcRec in the local database, but this search algorithm is not part of the synchronization procedure implemented in this thesis. Consequently, the cases marked N/A 3 do not exist.

N/A 4: Similarly to N/A 3, if a record R has just been added to the local database, then R cannot have a corresponding record in the source.

N/A 5: If there is a record in the local database with no corresponding record the source, then this database record must have been added since the last sync with the given source. Therefore it cannot be in the No Change, Update, or Delete states.

8.2.2. Possible Actions

For each combination of source record change type and database record change type, there is an action that must be taken. Table 2 describes the possible actions. Each possible entry in table 1 requires one or several of these possible actions to be executed. This list of actions is complete.

nop	no operation, no action is necessary
add SrcRec to local database	add SrcRec to local database
update R	update record R in local database with data from SrcRec. This creates a new version $R(t_{now})$ of record R with the information from SrcRec.
add R to source	add record R from local database to the source
reconcile	reconcile SrcRec with R. For example, if R and SrcRec have both changed, the user is required to merge changes.[1]
mark R delete	mark R as "Delete" in the local database
del SrcRec	delete SrcRec in the source. [2]
update SrcRec	overwrite SrcRec in the source with $R(t_{now})$
confirm	if one record has been updated and the corresponding record has been deleted or marked deleted, either the user is asked for which action is correct or a new record based on the updated record is created

Table 2 - Possible Actions

Table 3 outlines what actions need to be taken under which circumstances. The table explains which changes in the source and/or changes in the local database to corresponding records trigger which actions.

[1] Another possibility is to create two records based on R and SrcRec.

[2] In the future, the algorithm should ask the user to confirm the deletion of a source record. All deletion should also be logged so that the user could restore a deleted record manually.

Required actions for changes since last sync with given source		Type of change to SrcRec in source				
		not exist	no change	add	delete	update
Type of change to R in local database	not exist	N/A	N/A	add SrcRec to local da-tabase	N/A	N/A
	no change	N/A	nop	N/A	mark R delete	update R
	add	add R to source	N/A	N/A	N/A	N/A
	delete	N/A	del SrcRec	N/A	nop	confirm
	update	N/A	update SrcRec	N/A	confirm	reconcile

Table 3 - Required Actions

If a record changed in a source, on the next synchronization run, the change should be applied to all corresponding records in all other sources in order to keep all sources synchronized. Before the change can be applied to other sources, the change needs to be identified.

8.2.3. Type of Change of R

The type of change to a record R in the local database can be determined by looking only at the local database. Table 4 lists the type of changes and how they can be identified.

Type of Change to R in Local database	Identified by
not exist	$R = \phi$ (i.e. there exists no record R corresponding to SrcRec in local database)
no change	$R(tnow) = R(tlastsync)$ $\wedge R(tlastsync) \neq \phi$ $\wedge R.status = Write$
add	$R(tlastsync) = \phi$ $\wedge\ SrcRec = \phi$ $\wedge R.status = Write$
delete	$R.status = Delete$ $\wedge R(tlastsync) \neq \phi$
update	$R(tnow) \neq R(tlastsync)$ $\wedge R(tlastsync) \neq \phi$ $\wedge R.status = Write$

Table 4 - Identification of Change Types of R

The remainder of this section explains the entries in Table 4. Let source A be the source that is currently being synchronized. Let $t_{lastsync}$ be the time of the last synchronization with source A. $R(t_{lastsync})$ is the version of record R as of time $t_{lastsync}$. Let t_{now} be the current time.

No Change: A database record R is unchanged (change type "No Change") with respect to source A if the $t_{lastsync}$ version is the same as the newest version of record R. Obviously, the $t_{lastsync}$ version must exist and R must not be a deleted record.

Add: A database record R has been added (change type "Add") to the local database since the last synchronization with source A if R is not marked deleted and a $t_{lastsync}$ version of R does not exist. Additionally, a corresponding source record SrcRec must not exist.

Delete: A database record R has been deleted (change type "Delete") from the local database since the last synchronization with source A, if R is marked deleted and a

$t_{lastsync}$ version of R exists. The second requirement is subtle, but if a $t_{lastsync}$ version does not exist, then R has been added to the local database (and marked deleted) since the last synchronization with source A. This implies that there exists no corresponding source record in A because a corresponding source record would have only been created during a synchronization run. Hence, nothing needs to be done, since there is no corresponding source record.

Update: A database record R has been updated (change type "Update") in the local database since the last synchronization with source A if R is not marked deleted and the $t_{lastsync}$ version is not equal to the t_{now} version of record R. Obviously, the $t_{lastsync}$ version of R must exist, otherwise R would have change type "Add" and not "Update".

8.2.4. Type of Change of SrcRec

Identifying the change to SrcRec in the source would be simple if the source supported a method to query for all records that have changed since the last synchronization. The heterogeneous nature of the sources prevents the assumption that all sources support such a query for changed records. To identify which records have changed, all source records SrcRec must be read from the source and compared to the information stored in the local database. Table 5 outlines how changes in the source are identified.

Type of Change to SrcRec in Source	Identified by
not exist	SrcRec = ϕ (i.e there exists no record SrcRec corresponding to R)
no change	SrcRec = $R(t_{lastsync})$
add	$R = \phi \wedge$ SrcRec $\neq \phi$
delete	$R(t_{lastsync}) \neq \phi \wedge$ SrcRec $= \phi$
update	SrcRec $\neq R(t_{lastsync})$

Table 5 - Identification of Change Types of SrcRec

The remainder of this section explains the entries in Table 5. Let source A be the source that is currently being synchronized. Let $t_{lastsync}$ be the time of the last synchronization with source A. $R(t_{lastsync})$ is the version of record R as of time $t_{lastsync}$. Let t_{now} be the current time.

No Change: A source record SrcRec is unchanged (change type "No Change"), if SrcRec is equivalent to the $t_{lastsync}$ version of the corresponding record R. They correspond since the $t_{lastsync}$ version was updated from SrcRec (or vice versa) at the last synchronization.

Add: A source record SrcRec has been added (change type "Add") to A, if there exists no record R in the local database corresponding to SrcRec. Obviously the source record SrcRec must exist.

Delete: A source record SrcRec has been deleted (change type "Delete") from A, if a $t_{lastsync}$ version of the corresponding database record R exists and source record SrcRec no longer exists.

Update: A source record SrcRec has been updated (change type "Update") in A, if SrcRec is not equivalent to the $t_{lastsync}$ version of the corresponding record R, since the $t_{lastsync}$ version is the version to which SrcRec was equivalent at the last synchronization. Note that $R(t_{lastsync}) = \phi$ cannot occur since in order for the change type to be "Update," there must have been a record R in the local database at the time of the last synchronization.

Using Table 4 and Table 5, it is possible to determine which changes have occurred. Combining the conditions from both tables leads to a complete list of the conditions that must be tested in the synchronization process, shown in Table 6.

Conditions		Inverse Conditions
C1	$R = \phi$	$R \neq \phi$
C2	$R(t_{lastsync}) = \phi$	$R(t_{lastsync}) \neq \phi$
C3	$R(t_{now}) = R(t_{lastsync})$	$R(t_{now}) \neq R(t_{lastsync})$
C4	R.status = Write	R.status = Delete
C5	SrcRec = ϕ	SrcRec $\neq \phi$
C6	SrcRec = $R(t_{lastsync})$	SrcRec $\neq R(t_{lastsync})$

Table 6 - Conditions

8.2.5. Type of Change, Conditions, Actions

Combining the conditions with the possible changes results in Table 7, summarizing the synchronization algorithm.

Changes and their conditions		Type of change to SrcRec in source since last sync				
		not exist	no change	add	delete	update
Type of change to R in local database since last sync	not exist	N/A	N/A	C1 $\wedge$ ~C5 add SrcRec to local database	N/A	N/A
	no change	N/A	~C2 $\wedge$ C3 $\wedge$ C4 $\wedge$ C6 nop	N/A	~C2 $\wedge$ C3 $\wedge$ C4 $\wedge$ C5 mark R delete	~C2 $\wedge$ C3 $\wedge$ C4 $\wedge$ ~C6 update R
	add	C2 $\wedge$ C4 $\wedge$ C5 add R to source	N/A	N/A	N/A	N/A
	delete	N/A	~C2 $\wedge$ ~C4 $\wedge$ C6 del SrcRec	N/A	~C2 $\wedge$ ~C4 $\wedge$ C5 nop	~C2 $\wedge$ ~C4 $\wedge$ ~C6 confirm
	update	N/A	~C2 $\wedge$ ~C3 $\wedge$ C4 $\wedge$ C6 update SrcRec	N/A	~C2 $\wedge$ ~C3 $\wedge$ C4 $\wedge$ C5 confirm	~C2 $\wedge$ ~C3 $\wedge$ C4 $\wedge$ C6 reconcile

Table 7 - Summary of the synchronization algorithm

The following algorithm identifies all changes and takes the appropriate actions. First, all source records are retrieved from a source A. Each source record SrcRec is put through a series of tests to identify changes. Then the appropriate actions are taken. Thereafter, all records R that do not have corresponding source records in source A are sequentially read and put through a series of tests to identify if they changed, and the appropriate actions are taken. Now the local database and the source A should be consistent again. The same procedure is applied to the remaining sources until there are no more changes to be applied. Once all sources have been processed in this manner, an entire synchronization run is complete. Heterogeneous consistency is only reached if no changes happen to any source between the time it is processed and the end of the sync run. For the types of applications considered in this thesis, the change rate is low enough that heterogeneous consistency will almost always be achieved.

The following is a proposed implementation of the synchronization algorithm. Figure 7 and Figure 8 outline a proposed implementation of the `IndentifyChange(xxx)` algorithms.

```
Sync()
Begin
        For each Source S
        Begin
                For each SrcRec in S
                Begin
                        IdentifyChange(SrcRec)
                End
                For each R in the local database with no corresponding SrcRec
                Begin
                        IdentifyChange(R)
                End
        End
End
```

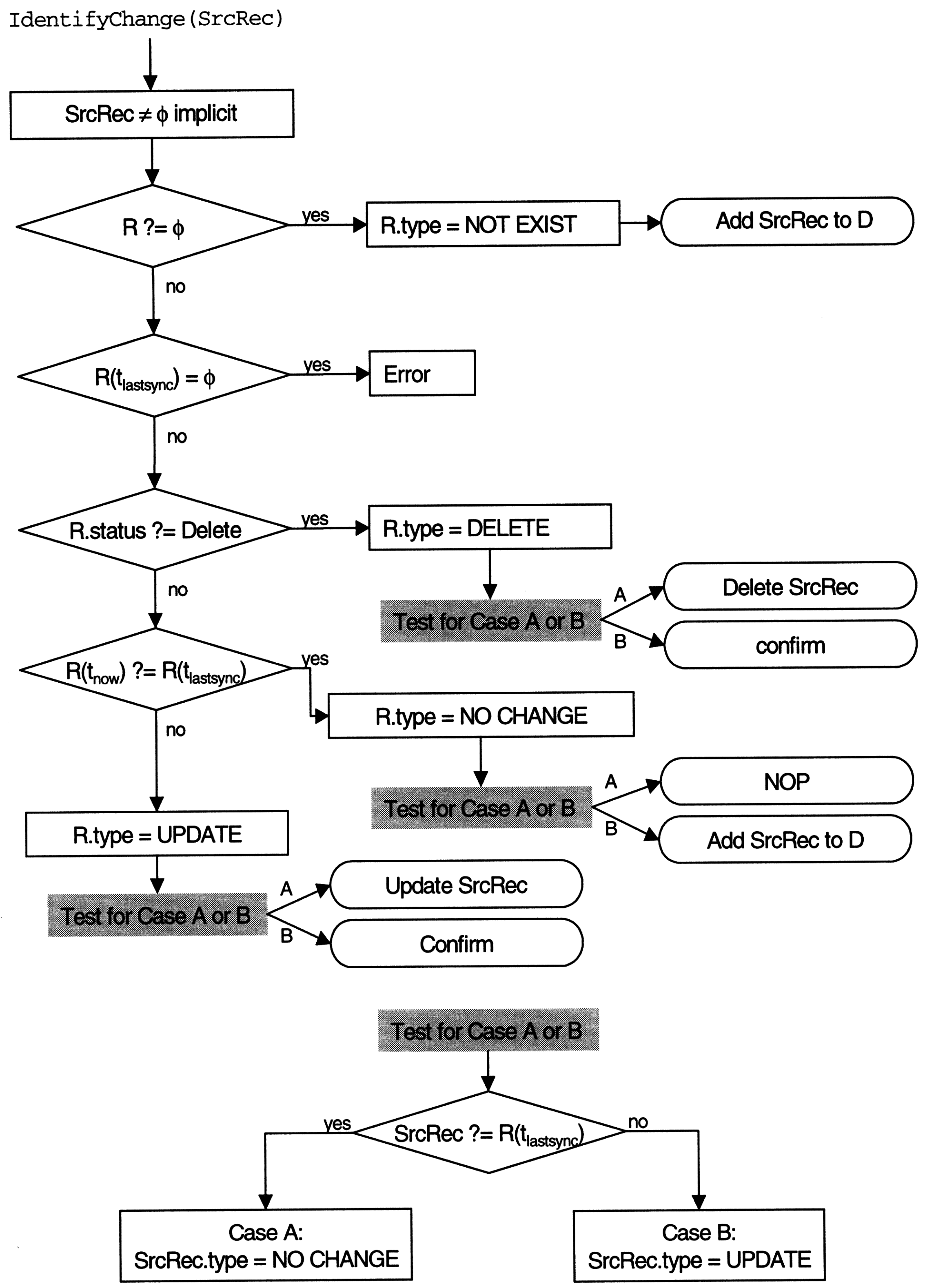

Figure 7 - IndentifyChange(SrcRec)

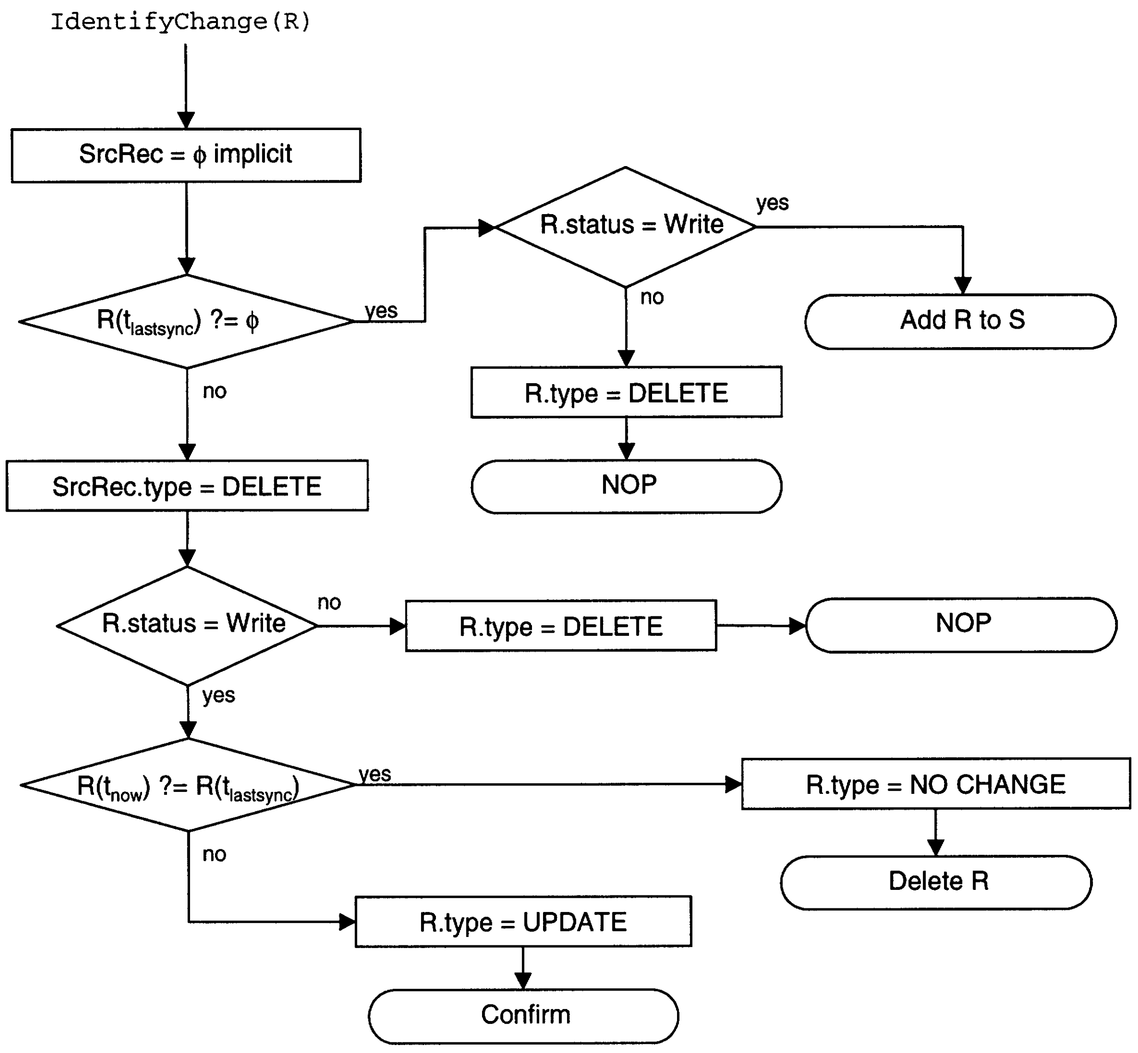

Figure 8 - IdentifyChange(R)

9. Scheduling of Synchronization Runs

The synchronization algorithm needs to run often enough that the user has the perception that the sources are almost always consistent. For example, in the domain of address books that means that a change in one address book should be applied to another within roughly a minute. To support this requirement, the user should be able to specify a fixed interval between automatic executions of the synchronization process. To improve this fixed scheduling, advanced mediators could notify the algorithm that a change has occurred in a source. The mediator need not guarantee that a change did take place, nor specify exactly what changed. A mediator could for example use the Windows NT file change notification feature. This operating system notification fires whenever a file has been modified. This notification does not guarantee that the source data has changed, but it is a strong hint which the mediator can pass on to the algorithm. The algorithm in turn buffers incoming notifications, and once a user defined delay time has passed without further modifications, a synchronization run is started. This approach assures that user modifications to a source results in a sooner than scheduled synchronization. It also avoids a notification propagation problem, since a synchronization run is not started immediately after a mediator notification. If this were not the case, a notification from mediator A could result in a synchronization run which immediately updates source B. This update in turn, could result in another notification by mediator B, starting another synchronization run. Since each run can take significant amount of time, the cascade of runs would degrade the user's system performance noticeably.

10. Failures

The synchronization process can take a long time to complete. Failures can occur during this period. For example, the network connection to a source might be interrupted, power may fail, a hard disk may crash, etc. The longer the synchronization algorithm runs, the greater the probability of a failure occurring. In case of a failure the synchronization run is interrupted. Assuming that the local database is accessed using transactions, and assuming that mediators write and update the sources in a transactional fashion, then the algorithm can recover from a failure.

Recovering also requires that the implementation of the algorithm follows a fail-fast design. Fail-fast means that the algorithm stops when it encounters an error rather than corrupting its local database or any of the sources.

Fail-fast for the local database is assured by the use of transactions. The mediators have to be implemented to guarantee fail-fast. For example, instead of updating a file, the mediator can update a copy of the file. Upon completing of the update, the new file is renamed to the original name. This method assures file level atomicity. The atomicity requirement is reduced to the rename operation. It is assumed that the operating system guarantees atomicity for the rename operation.

Given a fail-fast algorithm, achieved through transactional updates, recovery is straightforward. After the error has been eliminated, the synchronization process is started again. The last synchronization time of the record that was being synchronized at the time of the failure will not have been updated, since the synchronization time is part of the transactional update to the local database. Hence, after a failure the algorithm treats that record as if it had not been considered since the last completed synchronization. Additionally, the changes applied to sources during the failed run do not have to be reapplied, since the change identification algorithm recognizes that $R(t_{now}) == R(t_{lastsync})$ for those records.

For an example, assume that the algorithm is synchronizing with source A. It identifies that a record R has changed in the source A, that a record S has changed in the local database, and that a record T has changed in the source as well as in the local database. The algorithm applies the changes for record R and S before a failure occurs. The synchronization run stops leaving the change for record T unapplied. After the error is corrected, the algorithm is started again. Assume that records R, S, and T have not changed again since the failed synchronization run. The algorithm notices that the records R and S did not change, because the changes have already been applied during the partial synchronization run. Record T is recognized as changed, and this time the appropriate synchronization behavior occurs. Similarly, if records R and S have changed since the failed synchronization run stopped, then records R and S will be identified as changed by the algorithm, and the appropriate changes will be applied.

10.1 Read-Only Sources

An interesting example of a failure is the case of temporarily or permanently read-only sources. At first glance it might not make much sense to synchronize with a source that is read-only; however, the source may be read-only with respect to the synchronization process, but updateable by another method. Read-only sources can be considered as another form of failure, in which non of the changes are applied to the source. In the following next synchronization will try to apply the same changes to the source. It is clear, that if the read-only restriction was ever removed, then heterogeneous consistency would eventually be reached for the given source.

11. Local Database Maintenance

This section describes several enhancements to the local database that make the overall operation more efficient, but are not crucial to the synchronization process of the algorithm.

11.1 Database Collapsing

This local database can grow large when data changes often since it stores all versions of all records. To reduce this problem, the algorithm can be improved to store only the latest version of a record. For previous versions, a reverse-delta is stored. The reverse-delta is used to restore the original version of the record. For example, let version t_{now} be stored entirely. To retrieve version $R(t_l)$ with $t_l < t_{now}$ the reverse delta of t_l is applied to $R(t_{now})$ to generate version $R(t_l)$. This concept can be applied sequentially, such that each previous version is stored as the reverse delta of the following version. To generate a previous version all intermediate reverse-deltas have to be applied.

Applying reverse-deltas takes time. Using this method trades retrieval speed off for storage size. Unfortunately, this method is only a partial solution, since the overall size of the local database still increases over time, retaining old versions of records that will never be accessed again by the algorithm.

11.2 Database Pruning

The database can also be pruned of old versions of records that are no longer needed. Recall that only the versions $R(t_{now})$ and $R(t_{lastsync})$ of each record are needed for synchronization with each source. Consequently, all versions of records that are not the t_{now} or $t_{lastsync}$ versions of that record for each source can be periodically pruned from the database. Database pruning is an efficient way to keep the local database at the minimum number of versions of records. In most problem domains, especially the one considered in this thesis (email address books) pruning should be sufficient and database collapsing is not necessary. For problem domains with records that are individually very large, collapsing might be an important enhancement.

12. Advanced Sources

Some sources provide more capabilities than others. One example is Outlook, which can be queried for records that have changed since a specific date. Querying for changed records improves synchronization performance since the algorithm does not need to retrieve all records from each driver. Since the algorithm still has to work with mediators that cannot be queried for changed records, the algorithm must first ask the mediator if it can be queried for changed records. If it does, the synchronization speed for this source can be increased.

This concept can be extended to allow mediators to send a notification to the algorithm, saying that a change in their source might have happened. For more details see section 9.

V. Implementation: SyncEngine

1. Overview

The SyncEngine is a prototype implementation of the previously outlined synchronization process. The goal is to build a generic engine that synchronizes information between heterogeneous data sources. This engine can be used by the user to synchronize information between different applications or between the same application on different computers, such as a desktop and a laptop. The engine should allow other sources to be added and removed at the users' discretion. The engine needs to be extensible in the sense that in the future it can synchronize new types of information without changing the SyncEngine itself. All that should need to be changed are the mediators. (In the context of the SyncEngine the mediators are called drivers; one driver for each source.) A new source is integrated into the synchronization process by adding a new driver for that source. Different kinds of information from an existing source are added by modifying the existing driver for that source.

2. SyncEngine

The current implementation of the SyncEngine allows the user to synchronize the address book of Qualcomm's Eudora Pro 4.0 email-client with the contact store of Microsoft's

Outlook 97 personal information manager. This is an interesting scenario because Eudora's address book is an ASCII file, while Outlook's address book is an extensible database. Additionally, the SyncEngine cannot write to Eudora's address book since it is a combination of an ASCII file with a binary index file with unknown file format. The ASCII file could be updated, but that would leave the index file out of date, and Eudora would be unable to read its address book. Therefore, from the SyncEngine's perspective, Eudora's address book is read-only. These two sources represent two extremes, an ASCII file that can only be read via file I/O on the one hand, and a query based database on the other.

The SyncEngine itself is general and uses an abstract interface to the driver of each source to access the source's information. It does not assume that only contact information is synchronized. By changing the drivers, it is possible to synchronize other kinds of information, such as appointments. However, these modifications have not yet been implemented.

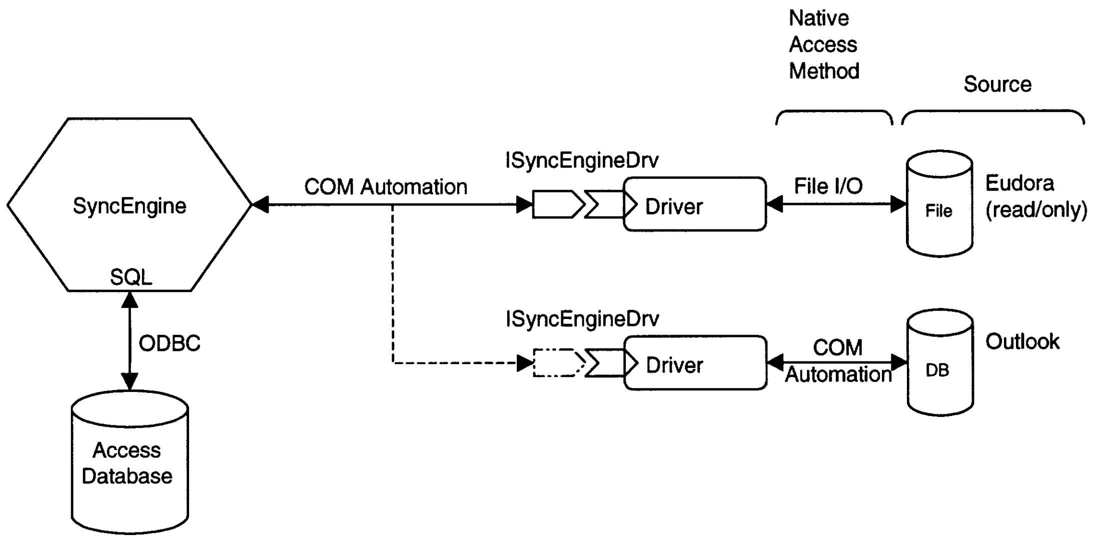

Figure 9 - SyncEngine Overview

Figure 9 shows the major components surrounding the SyncEngine. Depicted are the sources Outlook and Eudora, along with their address books, the drivers for each source, the SyncEngine, and the database local to the SyncEngine. The SyncEngine does not access the address books directly but via the driver for each source. Outlook exposes a COM Automation interface for access to its address book, which is used by the Outlook driver. The Eudora driver uses Windows file I/O calls to access the address book. The SyncEngine uses Structured Query Language (SQL) to access its local database.

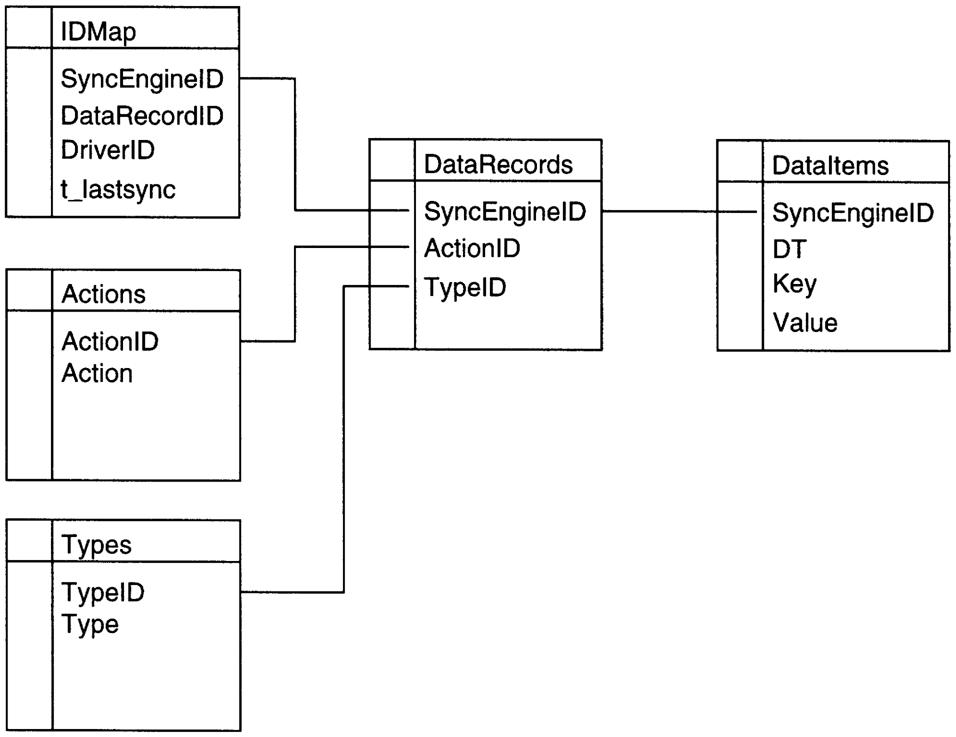

Figure 10 - Database Schema

When the SyncEngine starts the user can choose which sources to synchronize. Once selected, the user starts the synchronization process by clicking a button. The engine starts with the first source, dynamically loads its driver DLL and starts synchronizing that source. After that source is synchronized, the driver is unloaded to reduce the overall memory footprint of the engine. Unloading becomes important for drivers that cache the source data in memory such as the Eudora driver. Then the driver for the next source is dynamically loaded and synchronized. This procedure iterates over all selected sources. It is clear that after enough synchronization runs that all sources will eventually be consistent.

2.1 Local Database

The local database is a Microsoft Access database. A graphical representation of the schema is depicted in Figure 10.

2.1.1. IDMap Table

Each record is assigned a unique identifier by the driver where the record originated. This identifier is called DataRecordID. This id is only valid in combination with the driver id. The DataRecordID and DriverID pair uniquely identifier a record from a given driver.

Inside the database, all corresponding records are stored only once. For example, let the contact "Bill Clinton" from source A correspond to the contact "Bill Clinton" from source B. Instead of storing both records, the information is stored only once and is assigned a unique identifier with respect to the database. This id is called SyncEngineID. The IDMap table is responsible for maintaining the correspondence relationship (the "identifier link" in section IV.6) across drivers. The table contains the one to many relationship of on SyncEngineID to many DataRecordID/DriverID pairs. Lastly, each row has an t_lastsync field whose value is the time $t_{lastsync}$ for that record, meaning the last time this record identified by DataRecordID/DriverID was made or checked to be equivalent to the corresponding record in the source identified by DriverID. This field t_lastsync needs to be updated each time consistency has been established between the database and corresponding the source record.

2.1.2. DataRecords Table

The DataRecords table contains the status for each SyncEngineID record. The status is either "Write" or "Delete". The status is stored in the ActionID field. For extensibility and efficiency reasons, the ActionID field contains an index into the Actions table. (The Actions is assumed to contain at least two rows, one for "Write" and one for "Delete".) The TypeID field is a placeholder for future enhancements. See section 4.

2.1.3. DataItems Table

The DataItems table contains the actual values of all SyncEngineID records. For each SyncEngineID there are many rows, each row containing the Key and Value of a field of that record. The key is the type of the field's value property. For example, if a record has four fields than there will be four rows in the DataItems table for that record. But this table must also store all version of the same record. Therefore, each row contains a DT which is a date/time value. The correct record can be identified by the SyncEngineID along with the DT of that record.

2.1.4. Example

Here is an example, of what steps are necessary to retrieve a particular record. Let R be a record with ids DataRecordID and DriverID. To retrieve R(t), meaning R as of time t, first the SyncEngineID of R must be determined by a lookup in the IDMap table. Once, the SyncEngineID is retrieved, it can be used to look up status (ActionID) in the DataRecords table. Then the SyncEngineID is used to retrieve all fields for R, by retrieving all rows in table DataItems with SyncEngineID and DT = t. Putting all rows together results in R(t). This retrieval can be easily performed using inner-join SQL statements. The full SQL statement is:

SELECT DataRecords.SyncEngineID AS SyncEngineID, IDMap.DriverID, IDMap.DataRecordID, IDMap.t_lastsync, DataRecords.ActionID, DataRecords.TypeID, DataItems.DT, DataItems.Key, DataItems.Value

FROM (DataRecords INNER JOIN IDMap ON DataRecords.SyncEngineID = IDMap.SyncEngineID) INNER JOIN DataItems ON DataRecords.SyncEngineID = DataItems.SyncEngineID

WHERE (DataRecords.SyncEngineID = *id* AND DataItems.DT = *t*)

2.1.5. Interface

Inside the SyncEngine, record storage to and retrieval from the local database is abstracted into a class. The following is the interface:

```
class CSyncEngineDB
{
public:
    //.......................................................
    // Ctor/Dtor
    CSyncEngineDB();                         // ctor
    virtual ~CSyncEngineDB();                // dtor

    //.......................................................
    // Open/Close database and all member recordsets
    HRESULT Open();                          // open database and member record sets
    void    Close();                         // close database and member record sets

    //.......................................................
    // record manipulation methods
    HRESULT AddDataRecord(                   // add record to database or new version if
                                             // record already exists
        const CDataRecord& rDataRecord,      // record to add to db
        const COleDateTime& dt,              // date/time of record to add
        /*[OUT]*/ long& rlSyncEngineID       // return SyncEngineID of the added record
        )                                    // return S_OK on success
                                             // return E_OUTOFMEMORY
                                             // return E_INVALIDARG if no entry or no
                                             // dataitems for lSyncEngineID with time dt
        throw();                             // any CDatabaseException

    HRESULT GetDataRecord(                   // retrieve record from database by
                                             // SyncEngineID
        long lSyncEngineID,                  // ID of record in db
        const COleDateTime& dt,              // date/time of record to retrieve
        bool bExactDT,                       // records with exact dt or <= dt
        /*[OUT]*/CDataRecordDB* &rpDBDataRecord // pointer to retrieved record
        )                                    // return S_OK on success
                                             // return E_OUTOFMEMORY
                                             // return E_INVALIDARG if no entry or no
                                             // dataitems for lSyncEngineID with time dt
        throw();                             // any CDatabaseExecption

    HRESULT GetDataRecord(                   // retrieve record from database by
                                             // DriverID/DataRecordID and time
        LPCTSTR szDriverID,                  // syncengine driver name
        LPCTSTR szDataRecordID,              // record ID specific to syncengine driver
        const COleDateTime& dt,             // date/time of record to retrieve
        bool bExactDT,                      // records with exact dt or <= dt
        /*[OUT]*/CDataRecordDB* &rpDBDataRecord // pointer to retrieved record
        )                                   // return S_OK on success
        throw();                            // any CDatabaseException

    HRESULT AddMapping(                     // add mapping between source and existing db
                                            // record
        long lSyncEngineID,                 // SyncEn-gineID of existing record
        LPCTSTR szDriverID,                 // DriverID to map to SyncEngineID
        LPCTSTR szDataRecordID)             // DataRecor-dID to map to SyncEngineID
        throw();
    //.....................................................
    // Get/Set methods for driver last sync time
    HRESULT GetDrvSyncTime(                 // get driver last sync time
        LPCTSTR szDriverID,                 // syncengine driver name
        /*[OUT]*/COleDateTime& rdtDrvSyncTime // date/time of last sync with specified
                                            // driver
        ) const;                            // return S_OK on success
                                            // return S_FALSE on no last sync time entry
                                            // for drv meaning first sync with drv
                                            // return E_FAIL on failure

    HRESULT SetDrvSyncTime(                 // set driver last sync time
        LPCTSTR szDriverID,                 // syncengine driver name
        const COleDateTime& dtDrvSyncTime   // date/time of last sync with specified
                                            // driver
        );                                  // return S_OK on success
                                            // return E_FAIL on failure
};
```

3. Drivers

The drivers are implemented as dynamic link libraries (dlls). The SyncEngine dynamically loads each driver at run time and initializes it. Once initialized it can call the driver to return each record of the source. Following is the interface that each driver exposes.

```
class ISyncEngineDrv : virtual public IUnknown
{
public:
    STDMETHOD(Initialize)() PURE;           // initializes the driver
                                            // this method should be called
                                            // by the factory to initialize
                                            // an instance of this object.

    //--------------------------------------------------------------------
    // General driver properties:
    STDMETHOD_(BSTR,GetName)() PURE;        // returns unique driver name

    STDMETHOD_(BSTR,GetDescription)() PURE; // returns driver description

    //--------------------------------------------------------------------
    // Property for supported field enumeration
    STDMETHOD_(ISyncEngineDrvFieldsEnum*,GetFieldsEnum)() PURE;

    //--------------------------------------------------------------------
    // Property for records enumeration

    STDMETHOD_(ISyncEngineDrvRecordsEnum*,GetRecordsEnum)() PURE;

    //--------------------------------------------------------------------
    // Record retrieval methods:
    STDMETHOD_(long,GetCount)(              // returns the number of records in driver
        ) PURE;                             // return number of records
    STDMETHOD(GetDataRecord)(               // gets a specific record
        long i,                             // record to get: 0 thru GetCount()-1
        /*[OUT]*/IDataRecordWirerep* &rpDataRecord  // pointer to retrieved record
        ) PURE;                             // return S_OK on success

    STDMETHOD(AddDataRecord)(               // adds record to driver
        IDataRecordWirerep *pDataRecord     // record to add
        ) PURE;                             // return S_OK on success
        // Note: updates the record's driver ID and  record ID

    STDMETHOD(UpdateDataRecord)(            // update record in driver
        const IDataRecordWirerep *pDataRecord // record to update
        ) PURE;                             // return S_OK on success

    STDMETHOD(DeleteDataRecord)(            // delete record in driver
        const IDataRecordWirerep *pDataRecord // record to delete
        ) PURE;                             // return S_OK on success
};

class ISyncEngineDrvRecordsEnum : virtual public IUnknown
{
public:
    STDMETHOD_(long,GetRecordsCount)(       // returns the number of records in driver
        ) PURE;                             // return number of records

    STDMETHOD(GetDataRecord)(               // gets a specific record
        long i,                             // record to get: 0 thru GetCount()-1
        /*[OUT]*/IDataRecordWirerep* &rpDataRecord  // pointer to retrieved record
        ) PURE;                             // return S_OK on success

    STDMETHOD(AddDataRecord)(               // adds record to driver
        IDataRecordWirerep *pDataRecord     // record to add
        ) PURE;                             // return S_OK on success
        // Note: updates the record's driver ID and  record ID

    STDMETHOD(UpdateDataRecord)(            // update record in driver
        const IDataRecordWirerep *pDataRecord // record to update
        ) PURE;                             // return S_OK on success

    STDMETHOD(DeleteDataRecord)(            // delete record in driver
        const IDataRecordWirerep *pDataRecord // record to delete
        ) PURE;                             // return S_OK on success
};

class ISyncEngineDrvFieldsEnum : virtual public IUnknown
{
public:
    STDMETHOD_(long,GetFieldsCount)() PURE; // returns number of supported field names
                                            // return S_OK on success

    STDMETHOD_(BSTR,GetFieldName)(          // get a specific supported field name
        long i                              // field name to get: 0 thru GetCount()-1
        ) PURE;                             // return S_OK on success
};
```

The interface exposed by each driver conforms to the requirements of Microsoft's Component Object Model. However, the design should be modified in the future to conform more closely the to COM Automation guidelines. In particular the IxxxEnum interfaces are not compatible with the Automation guidelines. The goal is to make the interface conform well enough that it can be implemented using any language such as Java, VisualBasic or C++. Currently, because the interface does not conform, the drivers cannot be called from VBScript or JavaScript.

4. Future Work

To improve the efficiency of the local database, the enhancements described in section IV.11 should be implemented in future versions of the SyncEngine. The following sections describe other extensions to the current implementation that would improve the efficiency of the SyncEngine.

4.1 Record Comparison via Record Hashes

For domain with large record size the following proposal improves performance. The comparison of two records can be a bottleneck, because the corresponding fields of both records need to be compared. A future implementation could speed up this process by storing a digital signature [15] of all fields along with each record. The digital signature is a well-chosen one-way hash function of all field names and values of the particular record. The mediator/driver will be responsible for generating the hash for each record, and a hash will be stored for each record in the local database. When comparing two records the hashes of the two records can be compared.

4.2 Callback for Record Comparison

In the current implementation the SyncEngine compares the fields of two records using a simple string compare. However, as described previously, this might not be the correct behavior for all field value types. Only the mediators/drivers know the type of the field's value property for each field name that they support. In future implementations the driver should expose a method that takes two records as arguments and compares them. Since the driver understands the representation of all fields it can correctly compare the two records.

4.3 Callback for Find-Similar-Record

In the initial synchronization run, all records from all other sources are added to each source. If several sources contain similar records, which is likely since the user will have manually replicated the records, it will be important to avoid creating duplicates in these sources by “linking” the similar records. But the SyncEngine cannot find similar records, since it is unaware of the representation of the record’s fields. A solution might be to call the driver to locate a similar record for each record from another source during the initial synchronization. In case of contact information the driver might look for a match in first name, last name and email address.

4.4 Enhanced Links

The TypeID in the Types table of the local database a placeholder for future extensions. Currently all records are synchronized with all sources. The corresponding records are “linked”. In the future, it should be possible to define records that are not-linked, meaning they are not synchronized with other sources. This feature can be handy when a record is deleted in one record, but the corresponding records in other sources should not be deleted. The sources should then be consistent except for this “non-linked” record.

5. Tools and Infrastructure

The target platform for the engine is Win32, meaning Microsoft Windows 95/NT. The development environment is Microsoft VisualStudio 97 on Windows NT 4.0. VisualStudio includes VisualC++ as well as VisualBasic. C++, including language features like templates and exceptions, is used for the implementation of the SyncEngine and the drivers. To reduce development time, the Microsoft Foundation Classes (MFC) as well as the Standard Template Library (STL) are used. These libraries provide frameworks for user interface development as well as simple and advance data structures. The local database is created using Microsoft Access 97 and is accessed from the engine via the Open Database Connector (ODBC) using Structured Query Language (SQL) queries. MFC has ODBC support classes which facilitate the development of the database access.

NuMega BoundsChecker was used during development to identify run-time errors and memory leaks. Visual Quantify from Rational Rose was used for performance analysis.

VI. Experiments

This section gives the results of experiments performed to test the functionality of the SyncEngine as well as to identify shortcomings of the engine.

Information about the performance of the SyncEngine was gathered by creating a sample Eudora and Outlook address book. Table 8 shows sample records for each address book.

	Field Name	Sample
Eudora	FullName	EudoraContact0019
	Email	email0019@eudora.com
Outlook	FullName	Outlook Contact0193
	FirstName	Outlook
	LastName	Contact0193
	HomeAddressStreet	99 Magazine St, Apt 999
	HomeAddressCity	Cambridge
	HomeAddressPostalCode[3]	MA
	HomeAddressCountry	USA
	HomePhone	(617) 555-5555
	Email	Contact0193@outlook.com

Table 8 - Sample Records

[3] The Outlook driver incorrectly puts the state in the field named PostalCode. This will be corrected in the future.

Each record was created using a script. The records are identical except for the number following the word 'Contact'. The number is used to differentiate each record

1. General Validation

The first experiment with the SyncEngine was run to validate the functionality of the algorithm. The functionality of the engine was validated by comparing the number of records and fields in the local database after each synchronization run with the expected number of records and fields.

The setup was a clean state, where no synchronization had been run before, meaning that local database was empty. The Eudora and the Outlook address book contained one hundred records each. After two synchronization runs (two synchronizations with each driver), the local database contained 200 records and 1600 fields. The total number of records was 200, since each source contributed 100 records. The total number of fields was 1600, because each Eudora record had 3 fields initially (See Table 8.) Each Outlook record had 9 fields initially. Consequently, 100 Eudora records contributed 300 fields and 100 Outlook records contributed 900 fields for a total of 1200 fields.

At first glance, there are 400 more fields then expected. During the synchronization with Outlook the 100 Eudora records were added to Outlook. If a record was added to Outlook that has only a 'FullName' field but not a 'FirstName' or 'LastName' field, then Outlook assigned a 'FirstName' and/or 'LastName' field automatically. In this experiment, the records originating in Eudora only had a 'FullName' and 'Email' field. When these records were added to Outlook, each of these records was automatically assigned a 'FirstName' field by Outlook. During the next synchronization with Outlook, these records were identified as changed in Outlook. Accordingly, they were added to the local database as new versions. Each new version had four fields instead of originally three. Since there were 100 records with new versions, there were 400 new fields, for a total of 1600 as previously mentioned.

Subsequently, the “EudoraContact0010” record’s ‘Email’ field was changed to “TESTemail0010@eudora.com” from within Eudora. In the next synchronization with Eudora, this record was noticed as “Change” in the source and the corresponding record in the local database was “No Change”. Consequently, a version of the record was added to the local database, resulting in 1604 fields, since the new version had four fields. During the next synchronization run with Outlook, this record was noticed as “No Change” in the source Outlook and “Change” in the local database. The record in the source Outlook was updated with the latest version of the local database record.

2. Failure

One claim of the SyncEngine is that it is able to synchronize even if it is interrupted. The design assumes that the drivers can update the sources atomically and that the local database is updated atomically. The Eudora driver does not need to provide atomic updates, since it is read-only. The Outlook driver delegates updates to Outlook. It is assumed that Outlook provides atomicity for these updates on a record level. The SyncEngine only requires record level atomicity.

The SyncEngine’s behavior after failures has been tested as follows. During a synchronization run, a failure is simulated by terminating the SyncEngine’s process at a random time. To verify the assumption that Outlook provides atomic updates, a script validates that each record in Outlook complies to the sample record given in Table 8 differing only in the number after the word ‘Contact’. This does not prove the assumption to be true in the general case, since all simulated crashes of the SyncEngine could have happened while not updating Outlook. However, it ensures that the assumption is valid for the tests.

Neither of the two sources is changed after the interruption. This restriction is without loss of generality of the simulation, since changes to the sources after the interruption are handled identically to changes from before the interrupted synchronization that have not been applied before the interruption. This case is explained in more detail below.

After the interruption, the SyncEngine is started again, and synchronization starts by identifying which records have changed. There are two cases of changed records:

(a) Records whose changes have been applied before the interruption.

(b) Records whose changes have not been applied before the interruption.

In case (a), a change had been identified in the interrupted synchronization run and the correct action had been completed before the interruption. Completing the correct action means that the SrcRec is equal to $R(t_{now})$. In the following synchronization run the change identification algorithm will compare the SrcRec to $R(t_{now})$. Since the correct action had previously been completed, SrcRec is equal to $R(t_{now})$ and no further action needs to be taken.

In case (b), a change had either not been identified in the interrupted synchronization run or the correct action had not been completed before the interruption. In the following synchronization run that change identification algorithm will recognize the change and apply the correct action as it would have in the interrupted synchronization run.

The experiment was initialized with 50 records in the source according to the sample records in Table 8. The local database was empty. During synchronization the 50 source records need to be added to the local database. The SyncEngine was run inside the debugger. During the run, the SyncEngine process was stopped in debugger. At that point the number of records in the database was recorded. This number corresponds to the number of records in case (a), the records whose changes have been applied before the interruption. The remaining records are in case (b). Then the synchronization process was restarted. Upon completion the number of records in the database was compared to the expected number of records (50).

This test run was performed 15 times for synchronizing the Eudora source and 15 times for the Outlook source. Out of the 30 runs, the number of records in the local database after completion of the restarted synchronization was 50 records (as expected) except for the 11th Outlook synchronization. In that instance the restarted run did not complete because the SyncEngine terminated with an unexpected exception from Outlook's Automation interface. The cause of that exception could not be determined, nor could the failure be reproduced. Since writing Automation code from C++ is complex the most likely cause is that the Outlook driver is not using Outlook's Automation interface quite right. The scenario could not be reproduced.

The number of (a) and (b) cases was not centered around the expected value of 25 for each, assuming a uniform distribution over all 50 records. It was roughly centered around 36. The reason might be that the SyncEngine was stopped manually, and that the user stopped the process later rather than earlier.

The simulations have shown that the SyncEngine can be restarted after interruptions and proceed with the synchronization.

3. Read-Only

The SyncEngine is designed to be able to synchronize data sources, some of which might be temporarily or permanently read-only. The general validation described in section 1 did not validate synchronization with read-only sources.

Starting from the endpoint of the general validation experiment, the same record "EudoraContact0010" was changed. In particular its email field was altered to "MORE-TESTemail0010@eudora.com". The next synchronization run with Outlook noticed this change and added a new version of this record to the local database, resulting in 1608 fields in the local database. The following Eudora synchronization run noticed the change, and tried to update the record in Eudora. Since Eudora was read-only the update failed and the $t_{lastsync}$ time of that record for source Eudora was not updated to the time of synchronization. On subsequent synchronization runs, the SyncEngine noticed that the Eudora source record was "No Change" and that the local database record was "Change". The engine tried to update the source record on every subsequent synchronization run, but each time the update failed, and the $t_{lastsync}$ time of that record for source Eudora was never updated. At this point, the record was manually updated in Eudora. The subsequent synchronization run noticed that both corresponding records were changed. Consequently, the source and the local database records needed to be merged. The SyncEngine displayed a dialog box informing the user that the records needed to be merged. The experiment was aborted at this point. The experiment showed that the SyncEngine does correctly synchronize even if one of the two sources is read-only.

4. Performance

A series of tests was performed to measure the performance of the SyncEngine implementation. As stated earlier, performance was not a major design issue and consequently the implementation has not been optimized. The measurements were performed on a Pentium Pro 200 Mhz computer with 128 MB of random access memory running Windows NT 4.0 SP3.

4.1 Measurements

This section summarizes the results for several test runs to evaluate the scalability of the SyncEngine. Figure 11 and Figure 12 show the speed of synchronization with Eudora and Outlook respectively.

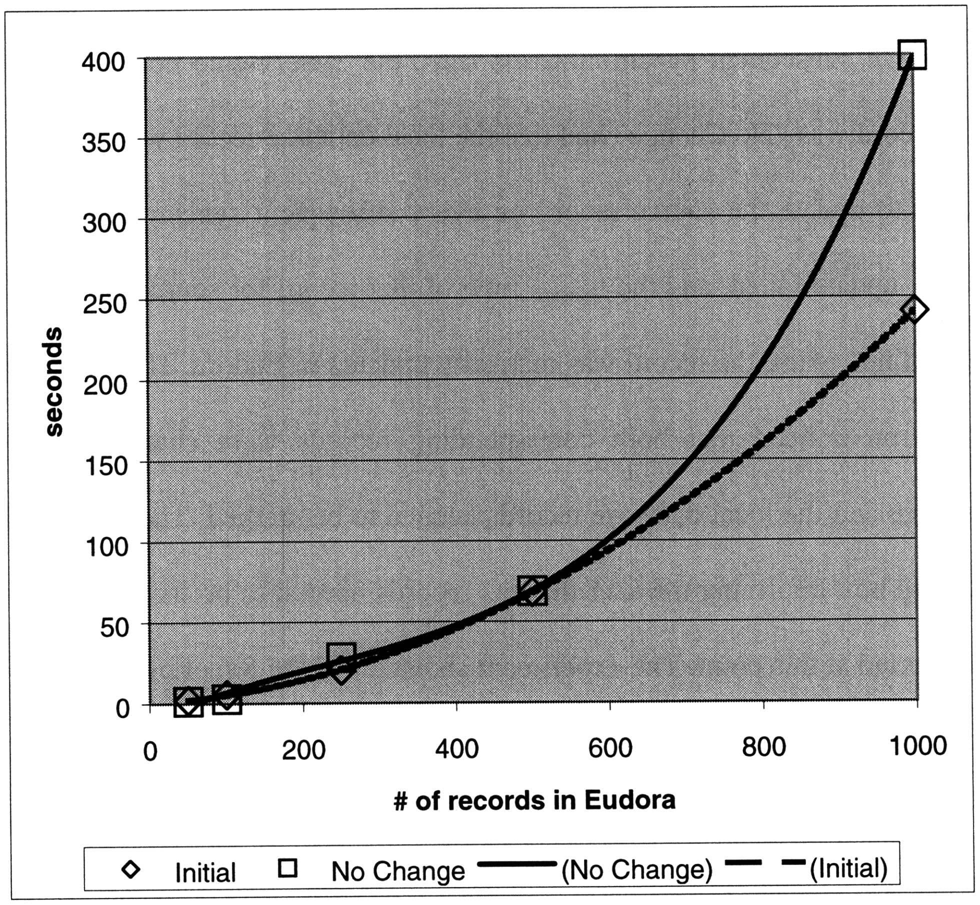

Figure 11 - Scalability of Eudora driver

Each figure contains two trend lines, one for an “initial sync” and another for a “no change sync”. Both are with respect to the number of records in the source. The “initial sync” line is the performance of synchronizing with the source for the first time when the local database is still empty. This synchronization mostly reflects the speed of retrieving records from the source and the speed of the local database for storing all retrieved records. The “no change sync” line is the performance of synchronizing with the source when there are no changes in the source nor in the local database. This synchronization mostly reflects the speed of retrieving records from the source and the local database and the performance of the change identification algorithm.

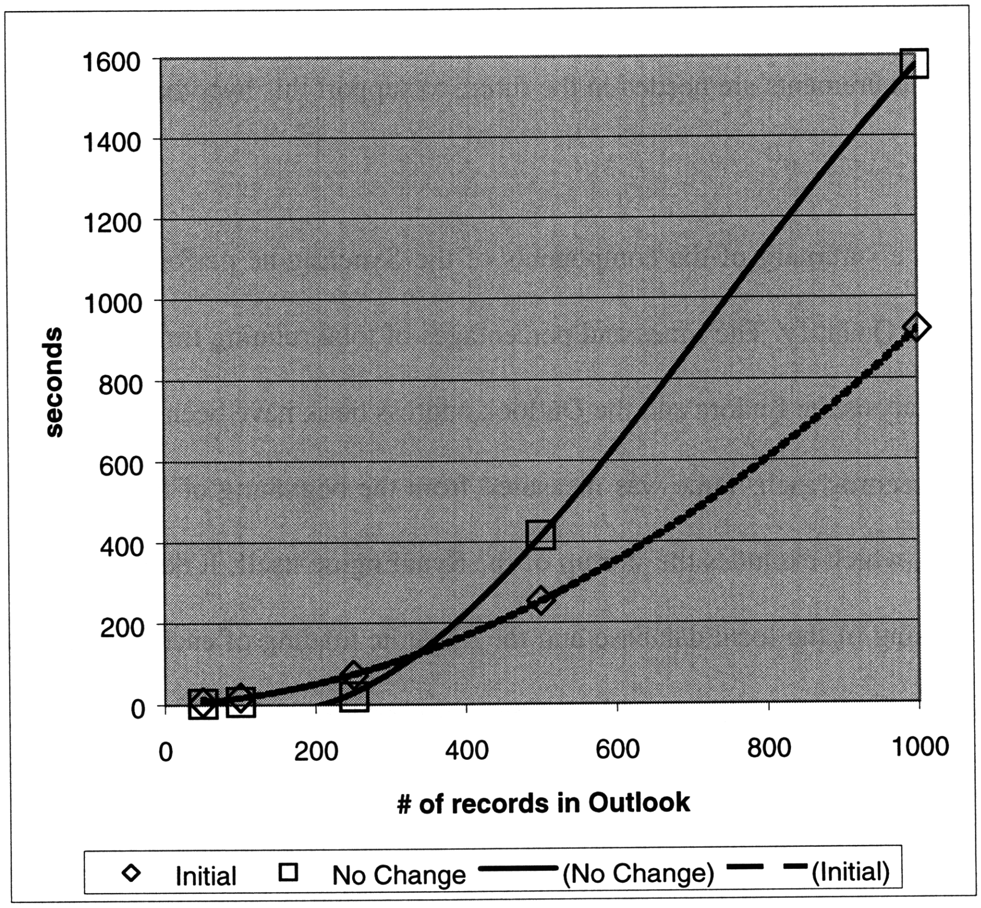

Figure 12 - Scalability of Outlook driver

It is obvious that the performance is not practical for real world usage. Performance can be improved by optimizing the drivers as well as the SyncEngine itself. More importantly, the trend with respect to increasing number of records in the source is linear in the range from 50 to 500 records for the Eudora driver and in the range from 50 to 250 for the Outlook driver. However, increasing the record range to 1000 records shows that the "no change sync" is close to $O(n^2)$, while the "initial sync" is close to $O(n^3)$. It was observed that the disk is operating at an extremely high number of seeks per second. A hypothesis is that the local database scales as $O(n^3)$ for inserts and as $O(n^2)$ for reads (plus few updates) with time dominated by disk seeks. (Recall that the local database is updated even in the no-change synchronization to keep the $t_{lastsync}$ values correct.) However more detailed measurements are needed in the future to support this hypothesis.

4.2 Analysis

The following is a summary of the components of the SyncEngine performance, measured using Visual Quantify. The times and percentages of total running time exclude idle times. In each analysis the Eudora and the Outlook address book have been pre-populated with 50 sample records each. Time was measured from the beginning of the actual synchronization run, which excludes the startup of the SyncEngine itself. It does include the opening and closing of the local database and the dynamic loading of each driver. Upon dynamic loading the Eudora driver reads in the entire address book file and creates a collection of records from which it serves all record retrieval calls. The loading time of the Outlook driver includes the initialization of an Automation COM connection to the running Outlook application, but no records are pre-cached.

4.2.1. Initial Synchronization Run

This run shows a typical behavior when a source is synchronized for the first time. The source is pre populated and the local database is empty.

The results for Outlook: Total running time: 33.9 seconds		
Time	% of Total Sync Time	Description
2.8	8.2%	opening local database
0.2	0.6%	dynamically loading Outlook driver
2.4	7.2%	retrieving records from Outlook driver (Estimated per record time: 0.048 seconds)
24.1	71.2%	adding records to local database (Estimated per record time: 0.482 seconds)

The results of Eudora: Total running time 4.59 seconds		
Time	% of Total Sync Time	Description
2.5	54.2%	opening local database
0.2	5.8%	dynamically loading Eudora driver
2.4	7.2%	retrieving records from Eudora driver (Estimated per record time: 0.048 seconds)
1.3	29.2%	adding records to local database (Estimated per record time: 0.026 seconds)

4.2.2. No Change Synchronization Run

This run shows typical behavior when no changes have been made to the source or to the local database. It is interesting to know where the time is spent during a synchronization run that does nothing except that it discovers that it has nothing to do for each record.

No Change synchronization with Outlook: Total running time: 19.8 seconds		
Time	% of Total Sync Time	Description
2.5	12.5%	opening local database
0.2	1.0%	dynamically loading Outlook driver
4.5	23.1%	retrieving records from Outlook driver (Estimated per record time: 0.09 seconds)
11.1	56.1%	retrieving records from local database and updating $t_{lastsync}$ (Estimated per record time: 0.222 seconds)

No Change synchronization with Eudora: Total running time: 4.8 seconds		
Time	% of Total Sync Time	Description
2.7	57.1%	opening local database
0.3	5.6%	dynamically loading Eudora driver
0.0	0.7%	retrieving records from Eudora driver (Estimated per record time: 0.0 seconds)
1.4	30.1%	retrieving records from local database and updating $t_{lastsync}$ (Estimated per record time: 0.028 seconds)

VII. Conclusions

The synchronization process described in this document guarantees eventual heterogeneous consistency between heterogeneous sources. The power of heterogeneous synchronization is its extensibility and that it works with any kind of information while not requiring changes to existing data sources. New sources and new kinds of data can be added to the system at anytime.

The techniques described in this thesis are most closely related to methods and techniques of the TSIMMIS system. The TSIMMIS system uses more sophisticated mediators. In TSIMMIS they are active components with rules for data translation and extraction. The mediators are advanced enough to access other mediators and query for information on the users behalf. This thesis only builds upon the concept of abstract data access of heterogeneous data sources.

It is important to note that,

- consistency is only reached if a synchronization run completes without simultaneous changes made by the sources.
- synchronization of read-only data sources is handled by the same algorithm that allows the synchronization process to be interrupted at any time due to failures.

The major limitation of the prototype implementation of heterogeneous synchronization described in this thesis is performance. Each synchronization run takes a considerable amount of time, up to over 20 minutes for a thousand records on the system examined here. Performance can be improved for sources whose mediator can tell the algorithm which records have change in the source.

The SyncEngine is a proof of concept implementation that successfully establishes consistency between the address books of Outlook and Eudora. More importantly, the SyncEngine shows that heterogeneous synchronization works for sources that are read-only as well. The implementation needs more work to make it truly generic and to increase performance.

VIII. References

1. Garcia-Molina, H., Hammer, J., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J. *Integrating and Accessing Heterogeneous Information Sources in TSIMMIS.* Department of Computer Science, Stanford University, CA.
2. Gifford, D. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles* (Dec 1979).
3. Hammer, J., Garcia-Moline, H., Cho, J., Aranha, R., Crespo, A. *Extracting Semistructured Information from the Web.* Department for Computer Science, Stanford University, CA.

4. Hammer, J., Garcia-Moline, H., Ireland, K., Papakonstantinou, Y., Ullman, J., Widom, J. *Information Translation, Mediation, Mosaic-Based Browsing in the TSIMMIS System.* Department for Computer Science, Stanford University, CA.
5. Helal, A., Heddaya, A., Bhargava, B. *Replication Techniques in Distributed Systems* (1996). Kluwer Academic Publishers, Boston/London/Dordrecht.
6. Howes, T., Smith, M. *LDAP Programming Directory Enabled Application with Lightweight Directory Access Protocol.* Macmillan Technical Publishing, Indianapolis, Indiana, 1997.
7. IntelliLink Corp. *IntelliLink Plus of Windows User's Guide.* IntelliLink Corp., Nashua, NH, 1995.
8. Ladin, R., Liskov, B., Shrira, L., Ghemawat, S. Providing High Availability Using Lazy Replication. In *ACM Transactions on Computer Systems, Vol. 10, No. 4* (Nov 1992).
9. Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., Williams, M. Replication in the Harp File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (Oct. 1991). ACM, New York.
10. Microsoft. *Automation Programmer's Reference.* Microsoft Press, Redmond, Washington, 1997.
11. Oki, B. M., Liskov, B. *Viewstamped Replication for Highly Available Distributed Systems.* Tech. Rep. MIT/LCS/TR-423, MIT Lab. for Computer Science, Cambridge, MA, 1988.
12. Oki, B., Pfluegl, M., Siegel, A., Skeen, D. *The Information Bus – An Architecture for Extensible Distributed Systems.* SIGOPS. ACM, Dec, 1993.
13. Petersen, K., Spreitzer, M., Terry, D., Theimer, M., Demeres, A. *Flexible Update Propagation for Weakly Consistent Replication.* Computer Science Laboratory, Xerox Palo Alto Research Center, CA.
14. Rogerson, D. *Inside COM - Microsoft's Component Object Model.* Microsoft Press, Redmond, Washington, 1997.
15. Schneier, B. *Applied Cryptography.* John Wiley & Sons, Inc., New York, New York, 1996.

5719 . 4