Awareness Services for Digital Libraries

Arturo Crespo and Hector Garcia-Molina

Stanford University

Abstract. We propose an architecture for Digital Library repositories where one or more data stores persistently hold the digital objects (e.g., documents), and interact with clients that perform indexing, replication, intellectual property management, revenue management, and other functions. One of the most critical components in such stores is the awareness mechanism, used to notify clients of inserted, deleted or changed objects. In this paper we survey the various awareness schemes (including snapshot, timestamp and log based), describing them all as variations of a single unified scheme. This makes it possible to understand their relative differences and strengths. In particular we focus on a signature-based awareness scheme that we believe is especially well suited for digital libraries, and show enhancements to improve its performance.

1 Introduction

A Digital Library *repository* stores and manages a collection of *digital objects*, such as documents, movies, or simulations [2, 7]. At the core of a repository is what we call the *data store*, responsible for storing in a reliable and scalable way the bits that make up the digital objects. A store interacts with other repository components. For example, an indexer may support associative searches over the contents on the digital objects. A name manager may be responsible for translating external user names (e.g., report STAN-CS-456-97) into a local identifier for a digital object. A replica manager may be responsible for making copies of digital objects across multiple stores, for reliability and availability.

It is essential that the store offer a simple, well defined interface to the clients that access it, to provide the flexibility and power that is needed in modern digital library repositories. For example, Figure 1 shows a scenario with three data stores, each containing HCI, AI, and DB technical reports. The stores may belong to different repositories and may be administered by different entities. Yet, we may wish to have a Computer Science indexer provide access to the reports in all three stores. At the same time, a second indexer (perhaps in the same repository as the DB store) may wish to index only the DB technical reports. It may also be convenient for an index to only access a *fraction* of the reports at a store (e.g., the CS index may only be over reports that are five or less years old).

A data store offers two fundamental types of services:

1. *Object Access:* Clients store and retrieve digital objects. Access to an object is via its *handle*, a store specific string that identifies the object [7]. Note that

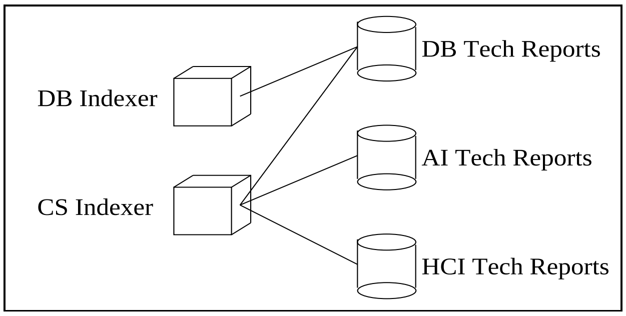

Fig. 1. A scenario with three stores and two indexers.

handles are typically not seen by end users. For instance, a user may search for a report by subject and author at some index, getting the external name of the report (e.g., STAN-CS-456-97 or some URL). To fetch the report, the name is translated by a name service into the handle for some specific object at some store.

2. *Object Awareness:* Clients must be aware of changes at the store, either new objects, deleted objects, or modified objects. Awareness schemes may use a push model, where the store informs interested parties, or a pull model, where clients ask "What's new?" Typically, awareness services are based on timestamps: e.g., all objects have a creation times and clients can ask for objects newer than some given timestamp.

The focus of this paper is on awareness services. They are fundamental services, needed if we want to decouple the storage of digital objects from their management (e.g., search, access rights, naming, payment). Furthermore, the traditional timestamp-based approach (used on the Web and in file system based stores), as well as other common schemes have some serious problems. Thus, in this paper we carefully consider the spectrum of options for awareness services and suggest some solutions that have important strengths for digital libraries.

To illustrate some of the problems with "traditional" awareness schemes, consider our experience with the CS-TR Project [6]. Under this project, Computer Science departments and research laboratories (currently about 50 participate) provide access to their technical reports. For each report, a bibliographic record (file) is created, giving the basic meta-data for the report (e.g., title, authors, date). Various index services need to get new bibliographic records on a regular basis. One of these indexes was SIFT, a selective dissemination service at Stanford where users could "subscribe" to new reports by giving keywords that described their interests.[1] Each CSTR store provides an interface to get new records, created after some given timestamp.

[1] SIFT is now operated by InReference, see `http://www.reference.com`. The CS-TR component was actually discontinued in 1996 when the service was commercialized.

First we considered using as the timestamp of a bibliographic record the creation time of the file that held the record. With this scheme, a store could simply look for new files when a client requested new records. This option was ruled out because we wanted to be able to migrate and replicate records. For example, when the hardware holding a collection becomes obsolete, we want to move the records to a new file system (keep in mind that digital libraries are long lived, so we need to plan to this!), and this would generate new timestamps for unchanged records. Similarly, we wanted to have replicas of a record at two stores to have the same timestamp, since otherwise an index service might consider one a new version of the other. For these reasons we ruled out file system generated timestamps.

Instead, we used application generated timestamps: a field within the bibliographic record indicated when the record was created or modified by a librarian. Unfortunately, this scheme lead to problems: librarians would modify records (e.g., to fix a typo in the author's name), and would forget to update the timestamp. Similarly, for one reason or another, records were created with old timestamps. All this lead services like SIFT to miss records and provide incorrect indexing.

Also, timestamps, either system or application generated, do not cope well with object deletion. That is, a store looking for objects with timestamps greater than some value, will not notice that some objects have disappeared entirely. This problem is of course clear to anyone who indexes Web sites. These sites are incapable of reporting deleted web pages, so we frequently see indexes that point to non-existent pages. While this may be acceptable for the Web, not all digital libraries will be satisfied with this type of awareness service.

An alternative to timestamp-based schemes that we carefully explore in this paper are content-based schemes. To illustrate one simple version, the store can compute a signature (e.g., large checksum) for each object it holds. It sends this full collection of signatures to a client, which can compare the signatures to the those for the objects it holds (or has indexed), yielding the identities of the changed objects. These schemes are especially attractive for digital libraries because they are not sensitive to how the bits are stored, only to the actual content of the object. However, it can be very inefficient to compute all the signatures and to send them to clients. Thus, in this paper we also explore techniques to make content-based schemes practical.

In summary, this paper makes two contributions:

- It surveys the spectrum of awareness options, highlighting the advantages and disadvantages of each. Interestingly enough, all the awareness options that we know of can be captured by a *single* algorithm with configurable options. We present this unifying algorithm, *UNI-AWARE*, and show how different choices for its parameters lead to the different awareness mechanisms such as timestamp-based, log-based, triggered updates, signatures, and so on.
- For signature-based schemes, we present several enhancements that reduce the computation and communication costs involved. As stated above, we

believe that these schemes have significant benefits in a digital library environment, where the way objects are stored may vary from one site to another, and over time.

We start our discussion in Section 2 by defining the awareness problem more formally. Section 3 briefly surveys related work, and Section 4 surveys the design space for awareness mechanisms. Our unifying algorithm, UNI-AWARE, is presented in Section 5, while Section 6 discusses optimizations for the signature-based algorithms.

2 Problem Definition

A data store contains a set of digital objects $R = \{O_1, O_2, ...O_N\}$. Each object has a *handle*, $O_i.H$, that can be used by a client to retrieve the object from the store. In addition, each object may have other *attributes* such as:

- Timestamp, $O_i.TS$: a timestamp representing when the object was created or last modified.
- Store, $O_i.S$: a globally unique identifier for the store that contains the object.
- Name, $O_i.N$: a user readable name for the object, e.g., "STAN-CS-456-97." Names can often be used to retrieve objects, with the help of a name service that maps names into handles.
- Title, $O_i.TITLE$: the object title.
- Author, $O_i.AUTHOR$: the object's author(s).
- Body, $O_i.B$: the main body of the object, e.g., a postscript file for the object, or the full text in ASCII.

Various sets of attributes are used in practice, e.g., the Warwick Framework [11] or the Dublin Core [16]. Each defines a list of possible attributes and their types (e.g., strings, lists of strings, nested records). The attributes for an object (sometimes called the meta-data) can either be stored together with the object itself, can be stored in a linked but separate file, or can be computed on the fly from the object. (For example, the title may be stored explicitly in a meta-data file, or it could be extracted by scanning the document for the string that follows "Title:".) For simplicity, in this paper we assume that all attributes are simply part of each object O_i.

We model object removal and creation as deletions and insertions to set R. We model the modification of O_i (or any of its attributes) by the creation of a new object O_j. For modification, there are three cases to consider:

1. *Versions:* The old object O_i remains, as a previous version. The handle of the new object, $O_j.H$ is necessarily different from $O_i.H$ in order to distinguish the versions.
2. *In-Place Update:* The old object O_i is removed, and the handle of the new object is unchanged, i.e., $O_j.H = O_i.H$. This represents an "in-place" update of the object.

3. *Shadow Updates:* The old object is removed, and the new object has a new handle, i.e., $O_j.H \neq O_i.H$

A *client* to a data store needs to "learn" about all the digital objects in R that satisfy some filter predicate σ. The filter σ specifies what objects the client wishes to learn about. Learning about object O_i may mean receiving a complete copy of O_i and storing it locally (for reliability). Learning about O_i may mean receiving the digital object, extracting certain attributes of interest, adding those attribute values (together with $O_i.H$ or $O_i.N$) to an index, and discarding O_i. Learning may also mean simply receiving the attributes of interest, and not the full object. The client also has a "forget" operation for O_i that undoes the action performed when O_i was learned.

Given this framework, we can now define the goal of an awareness scheme: whenever a store creates a new object O_i that satisfies the filter of a client, that client must (at some not-too-distant future time) learn about O_i. Whenever a store removes an object O_i that satisfied the filter of a client, that client must forget O_i.

In closing this section, we note that in general, there are two types of handles: semantic and arbitrary ones. By looking at a semantic handle we may be able to infer something about one or more of the attributes of the identified digital object. For example, if handles are strings of the form `site.id`, then from $O_i.H$ we can determine $O_i.S$. An arbitrary handle conveys no information when examined by anyone other that the store that generated it. For the awareness problem, two types of semantic handles are of interest:

– *Sequential handles:* the lexicographical order of handles determines the order in which their digital objects were created. Thus, if $O_1.H < O_2.H$, we know that object O_1 was created (or last modified) before O_2. (Note that sequential handles cannot be used with in-place updates as defined above, since a modified object must keep its old handle.)
– *Content-sensitive handles:* Two digital objects O_1 and O_2 (and all their attributes) are identical if and only if their handles $O_1.H$ and $O_2.H$ are identical. In this case, we can detect duplicate objects simply by checking their handles. Similarly, if a store modifies an object in any way, its handle will change, making detection of the change easier. (Again, content-sensitive handles cannot be used with in-place updates.)

In Section 5 we discuss how sequential or content-sensitive handles can be used for awareness.

3 Related Work

The awareness problem is related to the problem of maintaining database replicas. Snapshots as a means to update replicas were first introduced in [1]. The *Differential Refresh* Algorithm updates a previous snapshot by using a log [9]. Recent work in data warehousing improves the efficiency of these algorithms [10].

Remote file comparison algorithms are also related to the awareness problem. These algorithms attempt to locate disagreements between two or more large remote data files [13]. A class of algorithms that use randomized signatures to compare remote file copies is presented in [5]. The difference between these algorithms and our work is that they operate on a fixed set of pages, rather than with object deletion and creation. They also assume that few pages (objects) differ, while our algorithms do not make this assumption.

A related problem is the deployment of programs over a network. Current solutions use a repository that maintains the master copy of the application code. Remote nodes check the central node periodically for updates to the master copy. In case of an update, the remote node replaces its local copy with the updated application. Commercial software that solves this problem includes Marimba [12], NetDeploy [14], and BackWeb [3]. Although solutions to the awareness problem are solutions to the program deployment problem, there is a significant difference in scale. Program deployment solutions use file granularity and assume that the "master copy"contains a relatively small number of files.

4 The Client-Store Design Space

The performance and complexity of an awareness mechanism very much depends on how the store interacts with the client. In this section we briefly discuss the possible store-client interaction modes and parameters.

Push vs. Pull. Either the client or the store can be responsible for keeping the client up to date. With the *push model*, the data store is responsible for notifying the client when changes have happened. With the *pull model*, it is the client's responsibility to query the data store for the changes. With a hybrid *hint-pull model*, the client is still responsible for requesting changes, but the store can on its own provide a "hint" to the client that it is time to make a query.

A push model simplifies the design of the client as the client only needs to wait for changes. Also, network traffic may be reduced since messages are sent only when there are indeed changes. On the negative side, with a push model the store needs to know the the identity of all clients and must make sure that they receive the changes. A second problem was evident in the TULIP experiment, in which Elsevier Science distributed through the Internet science and engineering journals to nine universities [4]. The TULIP project used initially a push strategy, based on the assumption that all universities would be able to start receiving documents within the same time frame and that there would be enough storage space available at the universities to receive the information. However, these assumptions did not hold in practice and during the course of the project the strategy was changed to a pull model. A pull model, on the other hand, has complementary characteristics: the store is simpler, and the client needs to do more work.

Statefull vs. stateless stores and clients. Data stores can be either statefull or stateless with respect to clients. A statefull store saves information regarding what objects clients have learned about. The information can be on a client per client basis, or can be global. For instance, the store may remember the timestamp of the latest object seen by each client, or it may simply remember a single timestamp, that of the latest object seen by *all* clients. In the later case, a client may receive duplicate information.

In a statefull store, the state can be persistent or volatile. If the state is volatile, then a failure can destroy the information, in which case clients will have to use some other awareness scheme. For example, it the store loses the latest timestamp information, then clients will have to learn about all objects in the store (many not for the first time), instead of just the new ones.

If the store is stateless, then each client can save the state of its interaction with the store. For example, each client can remember the largest object timestamp it has seen. When it pulls updates from the store, it can ask for objects with a timestamp greater than this one.

Cognizant Clients and Sources. A cognizant store knows the identities of all its clients. It can use this knowledge to authorize pull requests from clients, or to decide what clients to push information to. A statefull store must be cognizant, but a stateless store need not be. An http server is an example of a non-cognizant, stateless store.

Although it may sound surprising, clients may or may not be cognizant of the stores they access. A Web crawler, for example, does not necessarily have a list of stores to access. Instead, it performs source discovery by examining the links in existing objects (e.g., URLs), and then contacting the appropriate stores for the web pages.

Number of Clients per Data Store. The ratio of the number of stores to the number of clients is a parameter that strongly affects the performance of the various awareness schemes. If a store serves many clients, it may be hard for the store to keep state information on a client per client basis, forcing us to use a less effective awareness scheme. Also, the frequency of pushing or pulling may have to be reduced.

A client may have to interact with more than one store, if for instance, it is building an index over multiple collections. If the stores use different awareness schemes, then the client must be prepared to run the different protocols. The unifying algorithm we present next, UNI-AWARE, may actually be a convenient framework for such a client to coordinate the different interaction styles.

5 The UNI-AWARE Algorithm

In this section we present *UNI-AWARE*, a unified awareness algorithm that "covers" known schemes. Casting all schemes in this same framework makes it possible to clearly see their differences and strengths. In Section 6 we present an enhanced version of UNI-AWARE.

Figure 2 presents Algorithm UNI-AWARE. We stress that the pseudo-code in the figure is a logical explanation and not necessarily the best implementation of the algorithm. In the code, all functions prefixed with "Store" run on the data store, while functions prefixed with "Client" run on the client; however, these functions can be started remotely through remote procedure calls.

```
func Store_UniAware(func σ_pre, func σ_post, func Compress ):
        i_obj = σ_pre( S ); // S contains all data store objects
        c_obj = Compress( i_obj );
        vector_store = σ_post( c_obj );
        return vector_store;

func Client_UniAware(func Client_σ_pre, func Client_σ_post,
                     func Store_σ_pre, func Store_σ_post,
                     func Compress, func Decompress ):
        i_obj = Client_σ_pre( C ); //C has all objects the client has learned about
        c_obj = Compress( i_obj );
        vector_client = Client_σ_post( c_obj );
        vector_store = Store_UniAware( Store_σ_pre, Store_σ_post, Compress );
        Learnobj= vector_store - vector_client;
        ForgetObj = vector_client - vector_store;
        NewObjs = Decompress( LearnObj );
        Learn NewObjs;
        OldObjs = Decompress( ForgetObj );
        Forget OldObjs;
```

Fig. 2. UNI-AWARE Algorithm

Algorithm UNI-AWARE uses six different *Custom Functions*; an instanciation of these functions tailors UNI-AWARE to execute a particular awareness *instance* (e.g., timestamps, signatures). A custom function takes as input a list of elements, that we call a *vector*, performs some operation on those elements, and constructs another vector. We now discuss the six functions.

The *Compress*() function receives a vector of digital objects and returns a vector of "compressed" objects. Intuitively, the compressed version of an object is a summary containing all that is necessary for detecting changes under a particular scheme. Two examples of valid compression are: a function that returns the LZW compression of a object [8], and a function that returns a pair $< object\_handle, update\_time >$. After a change is detected, the client may need to get the changed or new object (e.g., to make a replica). For this it uses the *Decompress*() function, an inverse of Compress. This function receives a vector of compressed objects and returns a vector of objects or object handles. For our examples above, the first decompression function would apply the LZW decompression algorithm, which will return the object itself. In the second example, Decompress would be a function that returns the object handle.

The store pre-selection function, $Store\_\sigma_{pre}()$, is used to eliminate from consideration objects the client is not interested in. For example, in the CS-TR environment, a client may indicate interest only in objects containing the word "database." The store post-selection function $Store\_\sigma_{post}$ is used to eliminate from consideration objects the client already has learned about, e.g., those with a timestamp greater than some value.

In the UNI-AWARE Algorithm, we conclude that an object needs to be forgotten when we do not receive its compressed version from the data store that sent the object originally. Since in this case the store did not apply its pre and post filtering function on this missing object, the client must check if it was interested in the original object (else, it did not learn about it, so there is no need to forget it). For this it uses functions $Client\_\sigma_{pre}()$ and $Client\_\sigma_{post}()$, which are analogous to those on the store side.

A cycle in the awareness process starts with a call to $Client\_UniAware()$. If we are using the pull model, the client makes this initial call; if we we are using a push model, the store remotely invokes this call at the client. (With the push model, the call can be initiated periodically, or can be triggered by an update to the store.) First, the $Client\_UniAware()$ function generates a vector of compressed objects, $vector_{client}$, representing those objects the client has learned about in the past. (Keep in mind that this is a conceptual description; for instance, the client may simply cache $vector_{client}$ so it does not have to recompute it.)

Next the client will ask the data store to generate, in a similar fashion, the $vector_{store}$. This vector is sent to the client which compares it with its own. The objects represented by $vector_{store}$ but not in $vector_{client}$ are ones the client must learn about. Those represented in $vector_{client}$ but not in $vector_{store}$ are ones the client must forget. Finally, the client decompresses the two difference sets to obtains handles for the objects that need to be learned or forgotten.

In Figure 3 we present a simple UNI-AWARE example. The objects at the store are shown in the top left. Let us assume the client has built an object cache, and its contents are shown in the top, right. Let us assume that the client is interested in all objects ($Store\_\sigma_{pre}()$ and $Client\_\sigma_{pre}()$ always evaluate to true), and that it is not possible to know from compressed objects if they have been learned by the client ($Store\_\sigma_{post}$ and $Client\_\sigma_{post}$ always return true). When UNI-AWARE runs, the $vector_{client}$ and $vector_{store}$ vectors are computed as shown in the figure, assuming that an object is compressed into its handle and some attribute C. When $vector_{store}$ is sent to the client, the client finds the handles of the inserted objects by performing $vector_{store} - vector_{client}$, and the handle of the deleted objects from $vector_{client} - vector_{store}$. Note that the update of the object with title "Paper2" generates a forget operation on the old version followed by a learn operation of the new version. (We can optimize the algorithm to detect these deletion-instertion actions on the same object, in order to do a single modify operation. However, we do not discuss this in this paper.)

In the following subsections we explore different alternatives for the Custom Functions. By changing the compression, decompression and post-selection

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2'
TR4	Paper 4

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3

$vector_{store}$: [TR1|C1] [TR2|C2'] [TR4|C4]

$vector_{client}$: [TR1|C1] [TR2|C2] [TR3|C3]

$LearnObj = vector_{store} - vector_{client}$: [TR2|C2'] [TR4|C4]

$ForgetObj = vector_{client} - vector_{store}$: [TR2|C2] [TR3|C3]

Fig. 3. Example of the UNI-AWARE Algorithm

functions we will produce four families of strategies: snapshot algorithms, timestamp algorithms, log-based algorithms, and signature algorithms. Note that the pre-selection functions do not change in these instances. They are only used to support client filtering, and they will not be included in our discussion.

5.1 Snapshot Algorithms

The Snapshot Algorithms use the simplest definitions of the Custom Functions. They only require that a store produce a snapshot, i.e., a vector of all objects at the store. We present two instances in this family.

The Simple Snapshot Algorithm uses the Customs Functions in Figure 4. These Custom Functions simply return their argument without change. This implies that the store sends all of its objects to the client (whether they changed or not) in order to update it. This is clearly wasteful, but on the positive side, the algorithm requires very little functionality from the store, and could be useful for small stores.

```
func Compress ( obj_vector ):            func Decompress ( obj_vector ):
   return obj_vector;                        return obj_vector;
 func Store_σ_post( obj_vector ):          func Client_σ_post( obj_vector ):
   return obj_vector;                        return obj_vector;
```

Fig. 4. Custom Functions for the Simple Snapshot Algorithm

The *Handle Snapshot Algorithm* only works with versions or shadow updates (no in-place updates), and when handles of deleted objects are not reused.

(Content-sentisitve handles are a special case of this scenario.) In this case, the snapshot need only include the handles of all store objects. When comparing with the previous snapshot, a new handle in the current snapshot will indicate an insertion, while a missing handle will indicate a deletion. The Customs Functions for this algorithm are given in of Figure 5. (Recall that $O.H$ is the handle of object O.)

```
func Compress ( obj_vector ):
    result = [ ];
    for obj in obj_vector:
        result += obj.H;
    return result;
func Store_σpost( obj_vector ):
    return obj_vector;

func Decompress ( obj_vector ):
    return obj_vector;
func Client_σpost( obj_vector ):
    return obj_vector;
```

Fig. 5. Custom Functions for the Handle Snapshot Algorithm

The Handle Snapshot Algorithm can be extended to include other object attributes in the compressed vector, in addition to just the handle. The additional attributes can then help us detect modifications, in case the handle is not sufficient. This variation is actually the Handle-Signature Algorithm described later on.

5.2 Timestamps

If timestamps at the store can be used to order objects by their insertion or modification time, then we can reduce the number of objects sent to the client. The first instance that uses this idea is the *Timestamp Algorithm*, whose functions appear in Figure 6. This instance only works with *no-delete* stores. A store is no-delete if it never removes objects. Modifications are allowed, either with versions or in-place. (Shadow modifications are not allowed since the old versions cannot be removed.)

In the Timestamp Algorithm, the store only includes objects with a timestamp greater than T_m, a marker timestamp. This marker was computed in the previous invocation of the algorithm, and tells us what objects the client has already learned of. The code in Figure 6 computes the maker timestamp for the next invocation, T'_m, in two places. This is not really needed: the client can compute T'_m and pass it to the store as part of function $Store\_\sigma_{post}$, or the store can compute it and save it for the next iteration.

In Figure 7 we present a sample execution of the Timestamp Algorithm. In the figure, we can see that the client has learned about all object except the one with handle TR3. Thus, the marker computed in the last invocation is T2.

func Compress (obj_{vector}):
  return obj_{vector}
func $Store\_\sigma_{post}$(obj_{vector}):
  T'_m = max timestamp in obj_{vector}
  return objects O with $O.TS > T_m$

func Decompress (obj_{vector}):
  T'_m = max timestamp in obj_{vector}
  return obj_{vector};
func $Client\_\sigma_{post}$(obj_{vector}):
  return [ ];

Fig. 6. Custom Functions for Timestamp Algorithm

The algorithm starts with the store creating $vector_{store}$ with all objects with timestamps greater than T2. The client then compares this vector to its own version to discover that it is missing TR3. In the process, the next marker is computed to be T3.

Data Store

Handle	Title	Timestamp
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3

Client

Handle	Title	Timestamp
TR1	Paper 1	T1
TR2	Paper 2	T2

Marker = T2

$vector_{store}$: TR3 | Paper3 | T3

$vector_{client}$: $empty$

$LearnObj = vector_{store} - vector_{client}$: TR3 | Paper3 | T3

$ForgetObj = vector_{client} - vector_{store}$: $empty$

Next Updated Marker = T3

Fig. 7. Example of the Timestamp Algorithm

A variation of the Timestamp Algorithm where we do not ship full objects is the *Handle-Timestamp Algorithm*. (We do not show its Custom Functions.) Here the Compress function extracts the handle and the timestamp of each object. (Notice that if handles are sequential (see Section 2) handles can play the role of timestamps, so we actually only need handles.) The timestamps are used as in the Timestamp Algorithm to eliminate objects the client is already aware of. When the client compares $vector_{store}$ and $vector_{client}$, it compares both the handles and timestamps. The benefit with respect to the Handle Snapshot Algorithm, is than now we can cope with in-place modifications or handle reuse. This is because the timestamp lets us detect modifications when the handle does not change. So, in summary, the Handle-Timestamp Algorithm works with no-delete stores and either version or in-place updates.

The reason we restrict the Timestamp and the Handle-Timestamp Algorithms to no-delete stores is that they cannot handle deletions. The problem with deletions is that after an object is removed, the data store does not keep any record of it. Therefore, the data store cannot tell the client to delete it.

There are two approaches for handling deletions. With the first approach, the client performs *lazy deletions*. That is, the client keeps all deleted entries until an error occurs, or until the client runs an algorithm, such as the Snapshot Algorithm, that can handle deletions. An error occurs, for example, if a user searches the client index to get the handle for an object of interest, and then goes to the store looking for that deleted object. In such an event, the user can ask the client to delete the index entry for the deleted object.

The second approach is for the data store to keep *tombstones* for the deleted objects. When using tombstones, for each deleted object D_i, the store inserts a tombstone object D_i' with the same handle as D_i ($D_i'.H = D_i.H$), and an attribute indicating that D_i' is a tombstone. The timestamp of D_i' is its creation time (i.e., D_i's deletion time). The only change to the Timestamp Algorithm (or to the Handle-Timestamp Algorithm) is that when the client learns of the creation of tombstone object D_i', it forgets D_i.

The disadvantage of using tombstones is that they take up space at the data store. However, tombstone space D_i' can be reclaimed when the store knows that all clients have learned of D_i'. This occurs when all client marker timestamps are greater than $D_i'.TS$. (This requires knowledges of all clients by the store.) Alternatively, the data store can unilaterally discard all tombstones older than a certain time marker T_{cut}. If client then uses a marker T_m smaller than T_{cut}, the request fails and the client must use a simple algorithm such as the Snapshot Algorithm.

5.3 Logs

A common metadata structure kept by stores (mainly for crash recovery) is a log. The log is a sequence of records that registers all the update activities in the database [15].[2] Each log record is conceptually of the form $< LSN, H, A, B >$, where LSN is a unique sequential number, H is handle of the affected object, A is the action performed (e.g. update, deletion, insertion), and B is some information about the new object body (such as a delta, the new object body).

There are essentially two ways to use logs. The simplest is to use the LSNs as timestamps for objects. That is, for each object with handle $O.H$, we can use the largest LSN in log records that contain $O.H$ as its "timestamp." For

[2] For our purpose, we will consider a log at the level of objects. If the system provides a lower granularity log, we can simulate an object-level log by coalescing several log records. Similarly, we will ignore log records that are not related to updates in objects.

example, say that the record with LSN = 21 registers the creation of object $O_1.H$, that record with LSN = 35 registers its modification, and there are no other records with $O_1.H$. Then the number 35 is O_1's current timestamp. The Timestamp or the Handle-Timestamp Algorithms can be used directly, under this interpretation of timestamps.

A second way to use logs is to have the store report changes by shipping the log records that describe the changes. This gives us the Log Algorithm, whose Custom Functions are given in Figure 8. A marker log sequence number L_m (computed in the previous invocation) tells us what records need to be shipped (analogous to T_m in the Timestamp Algorithm). When the client gets a new NewObjs vector to learn, it replays the log actions (in LSN order) and takes appropriate actions on its local state. For example, if a log record says object O_1 was deleted, then the client forgets O_1. If the record says some text was inserted into O_2, then the client may index the new text.

```
func Compress ( obj_vector ):
   result = [ ];
   for obj in obj_vector:
      result + = [obj.H, obj.H log records];
   return result;
func Store_σ_post( obj_vector ):
   discard all log records LSN ≤ L_m
   if object has no log records left
      remove object from obj_vector
   L'_m = max LSN in obj_vector
   return the new obj_vector;

func Decompress ( obj_vector ):
   L'_m = max LSN in obj_vector
   return obj_vector;
func Client_σ_post( obj_vector ):
   return [ ];
```

Fig. 8. Custom Functions for the Log Algorithm

The Log Algorithm requires substantial sophistication on the part of the client, since it needs to understand the store actions. Thus, in may only be applicable in limited situations, but in those cases if can effectively reduce the amount of data that must be sent to the client. As with tombstones in the Timestamp Algorithm, there is also the problem of limiting the size of the log. Similar techniques to those described in Section 5.2 can be used.

5.4 Signature Algorithm

The Timestamp and Log Algorithms use metadata (timestamps, log records) to reduce the amount of data sent to the client. As discussed in the Introduction, it is difficult to consistently manage timestamps (or logs) as stores fail or objects migrate from one computer to another. An algorithm like the Simple Snapshot one does not rely on metadata for change detection, and is hence much more robust in the face of operating system changes or failures. In this section we enhance that algorithm so that all full objects need not be sent to the client.

The idea is to send, for each store object, its handle and a *signature* that "summarizes" its content. A signature is a token that (i) has a high probability of being unique for each object in the data store, and (ii) changes when the content of the object changes. Several functions comply with these requirements including Cyclic Redundancy Checks (CRC) and checksums. The probability that a signature is unique for a given object content can be made arbitrarily low by using more bits in the signature. With say 128 bits, the probability that two objects will have the same signature is insignificant, so they can be used safely for detecting changes (unless one is the type of person that worries about our Sun going supernova). Signatures do not have to be recomputed each time they are needed; instead they can be cached. However, the cache need not be reliable, since if the signatures are lost or corrupted, we can always re-generate them by re-applying the signature algorithm to all the objects. Incidentaly, note that if handles are content-sensitive (see Section 2), then we do not need signatures, as handles can play the role of signatures.

Timestamps also have the two properties of the previous paragraph, so they can be used as signatures (assuming the system generates unique timestamps as each object is created or modified). However, with timestamps we do need system support since they cannot be inferred from the content, so in this sense they are less robust than content-based signatures.

The Custom Functions for the Simple Signature Algorithm are presented in Figure 9. The compression function produces a vector with tuples containing the object handle and signature. This vector is sent to the client, who compares it to its own vector to detect changes. For new or modified objects, the client uses the handle to request the object so it can learn about it. In Figure 10 we present an example for the Simple Vector Algorithm.

func Compress (obj_{vector}):
  result = [ ];
  for obj in obj_{vector}:
    result + = [obj.H,signature(obj)];
  return result;
func $Store\_\sigma_{post}$(obj_{vector}):
  return obj_{vector};

func Decompress (obj_{vector}):
  result = [ ];
  for obj in obj_{vector}:
    result + = obj.H;
  return result;
func $Client\_\sigma_{post}$(obj_{vector}):
  return obj_{vector};

Fig. 9. Custom Functions for the Signature Algorithm

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2'
TR4	Paper 4

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3

$vector_{store}$:

TR1	signature(Paper1)	TR2	signature(Paper2')	TR4	signature(Paper4)

$vector_{client}$:

TR1	signature(Paper1)	TR2	signature(Paper2)	TR3	signature(Paper3)

$LearnObj = vector_{store} - vector_{client}$:

TR2	signature(Paper2')	TR4	signature(Paper4)

$ForgetObj = vector_{client} - vector_{store}$:

TR2	signature(Paper2)	TR3	signature(Paper3)

Fig. 10. Example of the Signature Algorithm

Even though the Simple Vector Algorithm is more efficient than the Simple Snapshot Algorithm, it still must send many signatures to the client, one for each object in the store. In Section 6 we discuss improvements to reduce this cost.

5.5 UNI-AWARE Summary

The following table summarizes the UNI-AWARE options we have discussed.

Algorithm	Bytes Transmitted	Need Metadata	Limitations
SimpleSnapshot	$R + DN$	No	None
Handle Snapshot	$R + LN$	No	No in-place updates; no handle reuse
Timestamp	$R + DU$	Yes	no-delete store; no shadow updates
Handle-Timestamp	$R + 2LU$	Yes	no-delete store; no shadow updates
Handle-Timestamp with Tombstones	$R + 2LU$	Yes	None
Log	$R + 2LU$	Yes	None
Signature	$R + 2LN$	No	May miss updates (arbitrarily low probability)

The "Bytes Transmitted" column gives a *very rough* estimate of the communication cost per invocation. For this, we assume that the client must first send R bytes to the store, that there are a total of N objects, each of size D, that the client is interested in, and that of these, U have been inserted, deleted or updated since the previous invocation. We also assume that timestamps, LSNs, log records and signatures are all L bytes long. The two other columns in the table indicate if the algorithm requires system-supported metadata, and if there are any logical (i.e., not performance related) limitations.

6 Distributed UNI-AWARE Algorithm

The Signature Algorithm is desirable because it does not use system meta-data such as timestamps or logs. However, it does send substantial data to the client, even if few changes occurred. In this section we present a distributed algorithm, DIST-UNI-AWARE, that reduces this cost. We also present this algorithm as a unified algorithm that can be tailored to a variety of specific instances.

Intuitively, DIST-UNI-AWARE starts by sending a single "compressed" version of the objects in the store (as opposed to a compressed version of each object). If this summary (e.g., signature) matches what the client has, then the client knows immediately that there were no changes and no further messages are needed. If there were changes, the client asks the store to split the objects into groups and to send it a compressed summary of each group. The process then repeats by having the client check if the objects in each group have changed. Figure 11 presents Algorithm DIST-UNI-AWARE.

In order to form groups of objects, both the store and the client agree on a conceptual ordering of the objects. For example, the objects can be ordered by their handle or their title. Given this ordering, the store can describe a particular group (subset) by the minimum and the maximum objects accoring to the ordering. In addition, since the store usually wishes to describe multiple groups, it is convenient to include in each group descriptor the maximum object of the previous group, and the minimum object of the next group. Thus, in algorithm DIST-UNI-AWARE the store will send to the client a vector of group descriptors, VG. Each descriptor $VG[i]$ applies to a group of store objects and contains two attributes: $VG[i].Data$, the compressed objects (e.g., their combined signature), and $VG[i].GroupInfo$, the bounds record. A $VG[i].GroupInfo = I$ record contains in turn $I.Min$, an identification (e.g., handle) of the minimum object in group $VG[i]$; $I.Max$, an identification for the maximum object; $I.PrevMax$, the maximum of the previous group; and $I.NextMin$, the minimum of the next group.

Algorithm DIST-UNI-AWARE uses the following Custom Functions. The pre-selection functions work as before, except that for simplicity we now have a single function σ_{pre} used at both the store and the client. The $Compress()$ function receives a vector of digital objects and returns a single "compressed" data item that represents all the objects in the vector. There is no Decompress function; instead there is $Group()$ function that packages a set of objects into groups of objects. Finally,the post-selection function, σ_{post} operates on group descriptors (as opposed to a vector of objects).

func Store_DistUniAware($\ldots$ *custom functions*$\ldots$):
    $i_{obj} = \sigma_{pre}($ S); // S contains all objects in the data store
    $c_{obj}.GroupInfo.Min = \min(i_{obj}.H)$;
    $c_{obj}.GroupInfo.Max = \max(i_{obj}.H)$;
    $c_{obj}.GroupInfo.NextMin =$ NULL;
    $c_{obj}.GroupInfo.PrevMax =$ NULL;
    $c_{obj}.Data =$ Compress(i_{obj});
    $vector_{store} = [\sigma_{post}(\ c_{obj}\ )]$;
    return $vector_{store}$;

func Store_VectorAs(GI, $\ldots$ *custom functions*$\ldots$):
    $i_{obj} = \sigma_{pre}($ S); // S contains all objects in the data store
    $vector_{store} =$ Group(i_{obj}, GI);
    return $vector_{store}$;

func Client_DistUniAware($\ldots$ *custom functions*$\ldots$):
    $vector_{store} =$ Store_DistUniAware($\ldots$ *custom functions*$\ldots$);
    Client_ProcessVector($vector_{store}$, $\ldots$ *all custom functions*$\ldots$);

func Client_VectorAs(GI):
    if GI.Min not in $\sigma_{pre}(C)$: // C contains all objects in the client
        Learn GI.Min;
    if GI.Max not in $\sigma_{pre}(C)$:
        Learn GI.Max;
    Forget handles in $\sigma_{pre}(C)$ inside interval (GI.PrevMax...GI.Min);
    Forget handles in $\sigma_{pre}(C)$ inside interval (GI.Max...GI.NextMin);
    $i_{obj} =$ objects in $\sigma_{pre}(C)$ with handles in interval [GI.Min...GI.Max];
    $c_{obj}.Data =$ Compress(i_{obj});
    $c_{obj}.GroupInfo =$ GI;
    $vector_{client} = \sigma_{post}(\ c_{obj}\ )$;
    return $vector_{client}$;

func Client_ProcessVector($vector_{store}$, $\ldots$ *custom functions*$\ldots$):
    for $elem_{store}$ in $vector_{store}$:
        $vector_{client} =$ Client_VectorAs($elem_{store}.GroupInfo$);
        if ($elem_{store}.Data \neq vector_{client}.Data$):
            if $elem_{store}.GroupInfo.Min == elem_{store}.GroupInfo.Max$:
                Learn $elem_{store}.GroupInfo.Min$;
            else:
                v_{store}=Store_VectorAs($elem_{store}.GroupInfo$,$\ldots$ *custom functions*$\ldots$);
                Client_ProcessVector(v_{store},$\ldots$ *custom functions*$\ldots$);

Fig. 11. DIST-UNI-AWARE Algorithm

A cycle in the awareness process starts with a call to $Client\_DistUniAware$. The client requests to the data store a group vector by calling the function $Store\_DistUniAware()$. The data store produces and sends to the client an initial group vector that spans all of its objects. Then, the client generates a vector equivalent to the one sent by the data store. An equivalent vector is a vector that has the same grouping information (i.e. boundaries) as another. When generating an equivalent vector, the client may need to learn about an object when it does not have one of the group boundaries, or forget objects that are outside the boundaries of the group and the boundaries of the adjacent groups. The client equivalent vector is compared with the vector sent by the data store. If their Data attributes are equal, the client concludes that the group of documents has not changed, and no more processing is needed. On the other hand, if they are different, the client asks the data store to split the group vector into multiple vectors, and proceeds recursively with the same algorithm. The recursion stops, when groups have only one object. At this stage, the client can decide if it needs to learn or not about the object by comparing the data store compressed version of the object with its own compressed version.

6.1 Hierarchical Signatures

In this section we describe a DIST-UNI-AWARE instance, the *Hierarchical Signatures* Algorithm, that uses signatures of objects in a group.

The algorithm uses the Custom Functions of Figure 12. The compression function produces a single signature from a vector of digital objects. The grouping function receives a set of objects and information about a group, and produces two vector groups each with half the content of the original group. Digital objects are grouped together based on the value of their handles.

Figure 13 presents an execution example the Hierarchical Signatures Algorithm. In this example the client has learned about all the digital objects in the data store, with the exception of the object with handle TR9 that was just inserted, and TR6 which was updated; the client also needs to forget the object with handle TR5 that was deleted. In the first iteration of the algorithm, the data store groups all digital objects in a single group, and sends a vector containing the group information and the signature of $< Paper1...Paper9 >$ to the client. This group information is shown as the first four boxes in $vector_{store}$ under Round 1. It includes the handle of the last object in the previous group (null because there is no previous group in this case), the handle of the first object in the group (TR1), the handle of the last object in the group (TR9), and the handle of the first object in the next group (also null).

Next, the client uses the received group information to build the equivalent vector, $vector_{client}$, with all digital objects it has learned about with handles between TR1 to TR9. As the client does not have the object with handle TR9, it can immediately conclude that it needs to be learned. Assuming the client does so, it can then construct the equivalent vector, and notices that the signature in it (the data component) differs. Thus, the client starts Round 2 by requesting the data store to split the initial group. In Round 2, the data store forms two group

func Group(obj_{vector}, GI):
s_{handle} = sorted list of handles of obj_{vector};
delete from s_{handle} all handles $<$ GI.Min;
delete from s_{handle} all handles $>$ GI.Max;
// split divides list s_{handle} in two equal parts
(s1, s2) = split(s_{handle});
$v_{low}.GroupInfo.PrevMax$ = GI.PrevMax;
$v_{low}.GroupInfo.Min$ = GI.Min;
$v_{low}.GroupInfo.Max$ = max(s1);
$v_{low}.GroupInfo.NextMin$ = min(s2);
low_{data}=objects of obj_{vector} with handles in s1;
$v_{low}.GroupInfo.Data$=Compress(low_{data});
$v_{high}.GroupInfo.PrevMax$ = max(s1);
$v_{high}.GroupInfo.Min$ = min(s2);
$v_{high}.GroupInfo.Max$ = GI.Max;
$v_{high}.GroupInfo.NextMin$ = GI.NextMin;
$high_{data}$=objects of obj_{vector} with handles in s2;
$v_{high}.GroupInfo.Data$=Compress($high_{data}$);
return $[v_{low}, v_{high}]$;

func Compress (obj_{vector}):
return signature(obj_{vector});
func $Client\_\sigma_{post}(group_{vector})$:
return $group_{vector}$;
func $Store\_\sigma_{post}(group_{vector})$:
return $group_{vector}$;

Fig. 12. Custom Functions for the Hierarchical Signature Algorithm

vectors and sends them to the client. With the grouping information of those vectors, the client can immediately conclude that TR5 needs to be forgotten. (This is because TR5 is in between the maximum handle of the first vector and the minimum handle of the second vector. The client forgets about TR5 and builds the two equivalent vectors. This time, one of the group vectors is the same as the vector sent by the store and no more processing is needed for it. However, as the other vector is different, the client starts Round 3 by requesting the data store to subdivide it. In Round 3, the store sends information on group TR6, TR7, and on the group TR8, TR9. The second group matches what the client has, but the store is asked to further split the first group. The store does so, and in Round 4 sends individual signatures for TR6 and TR7. When the client receives the information on these atomic groups, it proceeds as with UNI-AWARE and concludes that it needs to forget the old version of TR6 and learn about the new version of TR6.

6.2 Hierarchical Timestamps

The *Hierarchical Timestamp Algorithm* is similar to the Hierarchical Signature Algorithm, except that the "signature" of a group of objects is now the largest timestamp of the group. If none of the objects in the group have been modified, then the group timestamp (i.e., the maximum timestamp) will match the group timestamp at the client. Otherwise the client will request that the group be split.

Data Store

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3
TR4	Paper 4
TR6	Paper 6'
TR7	Paper 7
TR8	Paper 8
TR9	Paper 9

Client

Handle	Title
TR1	Paper 1
TR2	Paper 2
TR3	Paper 3
TR4	Paper 4
TR5	Paper 5
TR6	Paper 6
TR7	Paper 7
TR8	Paper 8

Round 1:

Vectors from Store:

NULL	TR1	TR9	NULL	s(Paper1..Paper9)

Equivalent Vectors in Client:

NULL	TR1	TR9	NULL	s(Paper1..Paper9)

Client concludes: LEARN TR9 and start round 2 as s(Paper1..Paper9) $\neq$ s(Paper1..Paper9)

Round 2:

Vectors from Store:

NULL	TR1	TR4	TR6	s(Paper1..Paper4)
TR4	TR6	TR9	NULL	s(Paper6'..Paper9)

Equivalent Vectors in Client:

NULL	TR1	TR4	TR6	s(Paper1..Paper4)
TR4	TR6	TR9	NULL	s(Paper6..Paper9)

Client concludes: FORGET TR5 and start round 3 with the second vector as s(Paper6'..Paper9) $\neq$s(Paper6..Paper9)

Round 3:

Vectors from Store:

TR4	TR6	TR7	TR8	s(Paper6',Paper7)
TR7	TR8	TR9	NULL	s(Paper8..Paper9)

Equivalent Vectors in Client:

TR4	TR6	TR7	TR8	s(Paper6,Paper7)
TR7	TR8	TR9	NULL	s(Paper8..Paper9)

Client concludes: start round 4 with the first vector as s(Paper6',Paper7) $\neq$ s(Paper6,Paper7)

Round 4:

Vectors from Store:

TR4	TR6	TR6	TR7	s(Paper6')
TR6	TR7	TR7	TR8	s(Paper7)

Equivalent Vectors in Client:

TR4	TR6	TR6	TR7	s(Paper6)
TR6	TR7	TR7	TR8	s(Paper7)

Client concludes: LEARN TR6

Fig. 13. Example of the Hierarchical Signature Algorithm

We do not show the Custom Functions for this algorithm since they are very similar to those of the Hierarchical Signature Algorithm.

Timestamps are more effective than signatures for detecting changes because there the timestamps of different objects will never match. Signatures of different objects could match, but the probability can be made arbitrarily small by making the signatures large (e.g., 128 bits). Of course, as we have discussed, timestamps depend on the correct operation of the underlying operating or file system.

6.3 Performance Issues

The performance of the DIST-UNI-AWARE Algorithms depends very much on the number of new or changed objects between invocations. If there are no changes, only one round is required. If there is a single change, it takes roughly $\log_2 N$ rounds, where N is the number of digital objects. If there are two changes, we still require $\log_2 N$ rounds, but we may need to send twice as much data. As the number of changes increases, we end up having to split more and more groups. Eventually, with enough changes, every single group must be split, and the cost is much higher than if we had just sent all object signatures (or timestamps) outright.

There are many enhancements that can improve the performance of the algorithm as we have presented it. For example, groups can be split by factors of f instead of 2, reducing the number of rounds to $\log_f N$. This means that the store will send information for f groups at a time, but the cost of this may not be much higher for relatively small values of f. The initial message may also include f groups, instead of a single group as done in DIST-UNI-AWARE.

Another improvement is to have the client send information on its objects when it requests that a group be split. To illustrate, consider once again the example of Figure 13. There, at the end of Round 1 the client asks the store to split the initial group into two. Instead, the client could split *its* objects into two groups, TR1 through TR4, and TR5 through TR9, and send signatures for them. This way the store could find out that objects TR1 through TR4 exist unchanged at the client, so it then splits the other group into two, thereby saving one round. (That is, the store goes directly to Round 3.)

Another possible optimization is to try to "cluster" the changed objects into small groups. For example, suppose that most of the changes at the store are appends of new objects. Then, for grouping the objects we first sort them by timestamp (as opposed to by signature, as done by function Group in Figure 12). This means that most changed objects will be contiguous as we break them into groups. This implies that with little effort we can then pinpoint the few changed groups, and for each send more detailed individual object signatures. This is much more effective that the original scheme, where changed objects are randomly distributed across the sorted objects. It is also important to notice that even though this scheme uses timestamps, its correctness does not depend on correct timestamps. That is, if the timestamps are incorrect (and even different than those the client has for the same object), the algorithm will still work because it detects differences using signatures, not the timestamps. The timestamps are only used as "hints" for clustering the objects to reduce the data exchanged. Figure 14 illustrates this idea.

Data Store

Handle	Title	TS
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3
TR4	Paper 4	T4
TR7	Paper 7	T7
TR8	Paper 8	T8
TR9	Paper 9	T9
TR6	Paper 6'	T10

Client

Handle	Title	TS
TR1	Paper 1	T1
TR2	Paper 2	T2
TR3	Paper 3	T3
TR4	Paper 4	T4
TR5	Paper 5	T5
TR6	Paper 6	T6
TR7	Paper 7	T7
TR8	Paper 8	T8

Round 1:

Vectors from Store:

NULL	TR1,T1	TR6,T10	NULL	s(Paper1..Paper9)

Equivalent Vectors in Client:

NULL	TR1,T1	TR6,T10	NULL	s'(Paper1..Paper9)

Client concludes: LEARN TR6 and start round 2 as s(Paper1..Paper9) $\neq$ s'(Paper1..Paper9)

Round 2:

Vectors from Store:

TR1,T1	TR2,T2	TR4,T4	TR7,T7	s(Paper1..Paper4)
TR4,T4	TR7,T7	TR9,T9	TR6,T10	s(Paper7..Paper9)

Equivalent Vectors in Client:

TR1,T1	TR2,T2	TR4,T4	TR7,T7	s(Paper1..Paper4)
TR4,T4	TR7,T7	TR9,T9	TR6,T10	s(Paper7..Paper9)

Client concludes: FORGET TR5, LEARN TR9 and no more iterations are needed as both pairs of signatures are equal.

Fig. 14. Example of the Clustering Technique

In Figure 14, the contents of the store and client are shown at the top, with the timestamps indicating when objects were last modified. Initially the store orders all its objects by timestamp, and forms the initial group with all objects. The descriptor for the group includes the timestamps of the minimum and maximum objects. When the client notices that the signature for this first group does not match what it has, it asks the store to split the group. In Round 2 the store forms two groups and sends the descriptors. Notice that because the objects were sorted by timestamp, all the changes are located in the second group. If that group had to be split, again we would expect changes to be localized. (In the example,the group does not need to be split since the algorithm terminates quickly.) In general, ordering by timestamp tends to avoid having to split many groups, reducing the communication traffic.

The example also illustrates another improvement. Since the store sent as end points of the first group TR1 and TR6, it does not need to use them as end points in subsequent rounds. Thus, in Round 2, the groups are TR2 through TR4, and TR7 through TR9. Sending these smaller groups lets the algorithm terminate in Round 2.

7 Conclusions

In this paper we have studied awareness mechanisms for repositories of digital objects. We have argued that it is important to cleanly separate the storage functionality from the rest (indexing, replication, and so on), so that many and varied components can use the stores. With such a split, it is critical to have effective awareness schemes that can work correctly even as the objects in a store migrate from one "environment" to another (e.g., as operating systems, file systems and hardware gets upgraded). The signature or content-based schemes we presented are especially well suited for such a scenario, and we believe that the performance enhancements we discussed can make them practical.

We also believe it is instructive to view awareness schemes as variants of unifying algorithms such as UNI-AWARE or DIST-UNI-AWARE. This makes differences in assumptions and performance very apparent. Furthermore, such a unified view of awareness mechanisms could be extremely important for a client that must deal with stores that implement different schemes.

References

1. M.E. Adiba, and B.G. Lindsay: Database Snapshots. In *Proceedings of the International Conference on Very Large Databases*, 1980.
2. William Arms: Key Concepts in the Architecture of the Digital Library. *D-Lib Magazine*, July 1995. At http://www.cnri.reston.va.us/home/dlib/July95/07arms.html.
3. BackWeb. At http://www.backweb.com/.
4. Marthyn Borghuis, Hans Brinckman, Albert Fischer, Karen Hunter, Eleonore van der Loo, Rob ter Mors, Paul Mostert, and Jaco Zijlstra: *TULIP. Final Report.* Elsevier Science, July 1996.
5. D. Barbara and R.J. Lipton: A class of randomized strategies for low-cost comparison of file copies. In *IEEE Transactions on Parallel and Distributed Systems*, volume 2(2), pages 160-170. April 1991.
6. Corporation for National Research Initiatives: *Computer Science Technical Reports (CS-TR)*. At http://www.cnri.reston.va.us/home/cstr.html.
7. Robert Kahn and Robert Wilensky: A Framework for Distributed Digital Object Services. Technical Report tn95-01. CNRI, May 1995.
8. G. G. Langdon: A Note on the Ziv-Lempel Model for Compressing Individual Sequences. In *IEEE Transactions on Information Theory*, volume 29(2), pages 284-287, 1983.

9. B.G. Lindsay et al.: A Snapshot Differential Refresh Algorithm. In *Proceedings of the ACM SIGMOD Annual Conference*, 1986.
10. Wilburt Juan Labio and Hector Garcia-Molina: Efficient Snapshot Differential Algorithms for Data Warehousing. In *VLDB*, 1996.
11. Carl Lagoze, Clifford A. Lynch, and Ron Daniel Jr.: The Warwick Framework: A Container Architecture for Aggregating Sets of Metadata. Technical Report TR96-1593. Cornell University, June 1996.
12. MARIMBA: *Castanet*. At http://www.marimba.com.
13. J.J. Metzner: Efficient replicated remote file comparison. In *IEEE Transactions on Computers*, volume 40(5), pages 651-660. May 1991.
14. Open Software Associates: *NetDeploy*. At http://www.osa.com/products/netdploy
15. A. Silberschtaz and H. F. Korth and S. Sudarshan: *Database System Concepts*. Mc Graw Hill, 1996.
16. Stuart Weibel, Jean Godby, Eric Miller, and Ron Daniel: OCLC/NCSA Metadata Workshop Report, March 1995.