VERSION HEADERS FOR FLEXIBLE SYNCHRONIZATION AND CONFLICT RESOLUTION

Ken Rimey

November 22, 2004

HIIT
Technical
Report
2004-3

Version Headers for Flexible Synchronization and Conflict Resolution

Ken Rimey

Helsinki Institute for Information Technology
Tammasaarenkatu 3, Helsinki, Finland
P.O. Box 9800
FIN-02015 TKK, Finland
http://www.hiit.fi/

HIIT Technical Reports 2004-3
ISSN 1458-9478

Version Headers for Flexible Synchronization and Conflict Resolution

Ken Rimey
Helsinki Institute for Information Technology
rimey@hiit.fi

Abstract

We propose a set of metadata fields to be included in object versions to enable a form of optimistic replication in which synchronization is performed by copying versions from site to site. To enable determining whether one version has been derived from another, we include in the metadata an explicit list of predecessor version IDs, utilizing range encoding to achieve a compact representation.

We allow conflicting versions to be copied from site to site, creating a possibility that multiple sites will attempt to resolve the conflict. The paper defines a scheme for consistently arbitrating among different resolutions of the same conflict, and for avoiding run-away cascades involving conflicts among conflict resolutions.

The proposed architecture enables the use of a variety of synchronization protocols and conflict resolution algorithms in the same distributed system. We are using it in a research prototype of a replicated XML database running on desktop computers and Symbian OS mobile phones.

Keywords

Mobile data management, optimistic replication, data synchronization, epidemic algorithms, versioning, conflict resolution, native XML database.

1. Introduction

Optimistic replication, in which updates to a replicated data set are accepted at any replica site without coordination with the other sites [2, 14], can be based on propagation of object versions from site to site [8, 3], or it can be based on propagation of updates [13, 6], such that the current version of an object at a site is implicitly determined by the log of updates known at that site. While both approaches are interesting (and hybrid approaches are feasible), this paper focuses on version propagation.

With update propagation, synchronization and conflict resolution are naturally intertwined, because even if arriving updates reflect the concurrent creation of different new versions of an object at different remote sites, the assembly of these updates into a single log will effectively define a single merged version. In that

approach, conflict resolution amounts to deciding how to order the log entries, and possibly how to omit or modify some of them, such that the preconditions of all the entries are satisfied and the log can be executed to produce a result.

With version propagation, it is possible to generalize by not requiring conflicts to necessarily be resolved when synchronizing [3]. Synchronization then primarily amounts to copying of object versions from site to site. If nobody chooses to resolve a conflict between two concurrently created versions of some object, both versions will eventually be copied to all sites, achieving convergence in the specific sense that all sites have the same data.

This opens up the possibility of conflict resolution being performed by entities other than the synchronization engines. Whereas simple conflicts can and should be resolved as soon as they are detected, trickier conflicts are better left to client applications, which know the semantics of the data but are not necessarily running at the time or place the conflict is detected. Some conflicts, moreover, will require manual resolution.

In any case, we reject the notion of letting unresolved conflicts partition the network with respect to the objects in question.

An important goal for us is to allow the use of a variety of synchronization protocols and conflict resolution algorithms in the same distributed system. Since the use of a particular synchronization protocol is essentially a matter to be agreed upon between the two synchronizing sites, we wish to limit the requirements imposed on other sites. In particular, we decline to require versions to be propagated in any particular order, or to exclude the possibility of some device storing and propagating only a partial data set.

Similarly, we wish to allow coexistence of a diversity of conflict resolution algorithms within the same system, including algorithms that produce different results. This requires a scheme for consistently arbitrating among the results, and for avoiding run-away cascades involving multiple attempts to resolve conflicts among conflict resolutions.

The cornerstone of our solution and the focus of this paper is the design of the version header, a small set of metadata fields attached to each object version. The metadata and data for an object version comprise an immutable document, which is the atomic unit in which data is copied from site to site in synchronization.

The main challenges are as follows:

- Assigning unique version IDs in a distributed system.
- Enabling determination of the relationship between two versions v and v' of an object—namely, whether v supersedes v', v' supersedes v, or neither supersedes the other.
- Arbitrating consistently among conflict resolutions and avoiding run-away cascades.

Our version IDs are local counter values prefixed by a probabilistically unique string identifying the counter. We enable determination of the relationship between two versions by including a complete list of the ancestor version IDs in the version header. We use range encoding to achieve a compact representation of this list in most practical circumstances. We arbitrate among multiple resolutions of the same conflict and avoid run-away cascades by labeling versions created as a result of automatic conflict resolution with a description of the conflict being resolved, and by discarding all but one resolution of each distinct conflict in a consistent manner across replicas.

We have applied the ideas described in this paper in a research prototype of a replicated XML database, which stores collections of small, well-formed XML documents and provides an XPath [16] query capability, enabling client applications to select a subset of a collection to fetch and then monitor. Application areas with which we are experimenting include personal information management (PIM) and management of digital media files. In calendaring, for example, a collection represents a calendar and the objects in the collection represent calendar entries, which take the form of little XML (*xcal*) documents interconvertible to and from IETF iCalendar format [1]. In collaborative photo archive management, to take another example, the objects are metadata records [15] recording information about photographs and tracking the locations of the actual image files.

2. System Model

We assume a set of *repositories*, which operate as the nodes of a distributed system. Each repository contains a number of named *collections*.

Each collection C defines a set C_t of *object versions* at time $t = 0, 1, 2\ldots$ All collections start empty at $t = 0$.

The contents of a collection C can evolve by adding an object version v:

$$C_t = C_{t-1} \cup \{v\}$$

This can happen in one of two ways: A new object version v can be created at C, or an existing object version $v \in C'_{t-1}$ can be copied into C.

We denote the object identity of a version v as object(v). That is, v and v' are versions of the same *object* if and only if object(v) = object(v').

When an object version v is first created, it is assigned a set of zero or more *parent versions*, parents(v), which must be versions of the same object as v:

$$v' \in \text{parents}(v) \Rightarrow \text{object}(v') = \text{object}(v)$$

Most object versions will have either one parent or no parents.

The set of *ancestors* of an object version v, ancestors(v), includes the parents, the parents of the parents, and so on. In other words, ancestors(v) is the minimal set such that

$$\text{parents}(v) \subseteq \text{ancestors}(v)$$

and

$$v' \in \text{ancestors}(v) \Rightarrow \text{parents}(v') \subseteq \text{ancestors}(v)$$

If $v' \in$ ancestors(v), we say that v *supersedes* v'. Note that *supersedes* is a transitive relation.

If $v \in C_t$ and there is no $v' \in C_t$ such that v' supersedes v, we say that v is *current* in collection C at time t. Client applications accessing a collection will, for most purposes, only be interested in the current object versions.

When an object version v is first created, it is assigned a *logical clock value* [10], lclock(v) $\in \{1, 2, 3\ldots\}$. We require that

$$v' \in \text{parents}(v) \Rightarrow \text{lclock}(v) > \text{lclock}(v')$$

Furthermore, if v is created at a collection in a given repository at time t, it is natural to choose lclock(v) such that, for all collections C in that repository,

$$v' \in C_{t-1} \Rightarrow \text{lclock}(v) > \text{lclock}(v')$$

We will permanently designate each object version v as either an *ordinary* version, a *tombstone*, or a *join*.

```
<vevent meta:key="321b63a93a98"
        meta:version="5fdb1c:7"
        meta:parents="5fdb1c:6"
        meta:ancestors="5fdb1c:1-6 1ed41f:1-2,5"
        meta:lclock="44392"
        xmlns:meta="…">
  <summary>Afternoon meeting</summary>
  <dtstart>20040615T140000</dtstart>
  <duration>PT1H</duration>
  …
</vevent>
```

Figure 1. Calendar entry with versioning metadata.

Tombstones serve to provide deletion functionality of a sort, without requiring removal or modification of object versions. Tombstones always have exactly one parent, and they are never parents themselves.

Joins are versions that have been created automatically for the express purpose of eliminating or reducing a *conflict*. Joins always have more than one parent.

A *conflict* is a set of two or more current non-tombstone versions of the same object in some collection.

Finally, we also permit removal of an object version from a collection:

$$C_t = C_{t-1} - \{v\}$$

However, we disallow removing current versions.

Note that an object version that is present but not current in a collection C at time t will never be current in C at any later time $t' > t$.

3. Metadata Fields

On a more concrete level, we actually define five metadata fields for each ordinary object version: `key`, `version`, `parents`, `ancestors`, and `lclock`. In our XML database application, these correspond to attributes on the root element of each document. Figure 1 shows an example.

Tombstones and joins use these same attributes. Tombstone versions are distinguished by a special root tag (`meta:tombstone`). Joins are distinguished by the presence of an additional `join` attribute.

The metadata fields included in an object version, like the data itself, are determined when the version is first created and never modified thereafter.

3.1. key

Every object version v has a `key` attribute defining the value of object(v). This *object ID* is an arbitrary, opaque Unicode string. Two object versions are versions of the same object if and only if they have the same object ID.

An application program creating an initial version of an object might specify the object ID explicitly, or it might allow the repository to generate a probabilistically unique object ID.

3.2. version

Every object version also has a `version` attribute defining a *version ID* that must distinguish it from all other versions of the same object. The version ID consists of a *version prefix* and a decimal counter value, separated by a colon ("`:`").

The object ID and the version ID can be combined with an intervening slash ("`/`") to form the *full ID* of the object version, which will look something like this:

```
321b63a93a98/5fdb1c:7
```

Each repository maintains a counter for each object for which it has allocated one or more version IDs. (We actually maintain a separate counter for each collection when an object appears in several collections in the same repository.) Each counter has an associated version prefix uniquely identifying it among all others associated with the same object.

Our version prefixes are simply pseudorandom strings of hexadecimal digits. Proper seeding of the random number generator is essential here for minimizing the likelihood of a collision. Examples of appropriate seeds include a Universal Unique Identifier (UUID) or a sufficient number of bits of environmental noise (as in Linux's `/dev/random`) [9].

An alternative approach, which for many people comes to mind first, is to allocate a globally unique *repository ID*—perhaps again a pseudorandom string of hexadecimal digits—and to use this as the version prefix for all local counters. However, this has the effect of indelibly linking all object versions created at the same repository, even for disparate objects, which may be a privacy concern.

3.3. parents

The `parents` attribute of an object version is simply a space-separated list of the version IDs of the parent versions. The attribute is omitted for an initial version of an object. Otherwise it will most typically specify just one version ID. When there is more than one parent version, as in a *join*, their order is retained.

This attribute actually turns out to be of little practical use, but we store it anyway for reference.

3.4. ancestors

The `ancestors` attribute is fundamental for computing the *supersedes* relation and determining whether object versions are current. Its value for an object version v encodes all of the version IDs of object versions in ancestors(v). The attribute is omitted if this set is empty. Otherwise the encoding is constructed as follows[1]:

1. Group the version IDs by prefix.
2. Sort the groups in lexicographic order by prefix.
3. Sort the counter values in each group in ascending order and maximally combine them into ranges, including ranges of length 2.
4. Encode each group as a string along the lines of the following example:

```
0305f7:1-3,5,10-11
```

[1] In practice, one might allow or possibly require the use of certain abbreviations to reduce the redundancy among the metadata attributes. For instance, one might omit the parent versions from the `ancestors` attribute.

5. Combine the encodings into a space-separated list.

3.5. lclock

The `lclock` attribute of an object version v is simply lclock(v) expressed in decimal.

Because it is possible for different object versions to share the same logical clock value, defining a total order requires a tie-breaking rule. Our *standard total order* on object versions sorts on lclock(v) and the components of the full ID, in the following order:

1. by logical clock value (numerically),
2. by counter value (numerically),
3. by version prefix (lexicographically),
4. and lastly, by object ID (lexicographically).

We use the standard total order to do *redundancy suppression* in a globally consistent manner, as described in the next section.

3.6. join

The `join` attribute is only present if the object version is a *join*. It is used in *redundancy suppression* to avoid run-away cascades in conflict resolution.

The value is a space-separated list of version IDs (sorted by prefix and then by counter value) of the ordinary versions effectively merged by the join. This set, joined(v), is defined for any object version v as follows:

$$v \text{ is a } \textit{join} \;\Rightarrow\; \text{joined}(v) = \bigcup_{v' \in \text{parents}(v)} \text{joined}(v')$$

$$v \text{ is not a } \textit{join} \;\Rightarrow\; \text{joined}(v) = \{v\}$$

If joined(v) = joined(v') for two joins v and v', we say v and v' are *redundant*. If a collection comes to contain a redundant pair of current joins, we require the repository to immediately *suppress* the one that comes earlier in standard total order by creating a tombstone for it.

4. The Synchronization Problem

Although designs for synchronization protocols are outside the scope of this paper, we will define the synchronization problem and comment on the implied requirements for the version header design.

We define synchronization as a procedure operating on a pair of collections, C and C', which may be hosted at different repositories. Its job is to synchronize those object versions v satisfying a given *filter predicate*, matches(v). In the XML database application, this predicate is determined by an XPath expression [16]. If matches(v) ≡ *true*, we speak of *full synchronization*.

The actions performed by a synchronization procedure are limited to copying of object versions from C to C' and vice versa (and any suppression of redundant joins that this implies). The actual procedure will generally take the form of a distributed algorithm executing at C and C'.

We require the effect of performing a full synchronization of a pair of collections in a quiescent system to be to make them identical with respect to current, non-tombstone versions. More generally, we require the effect of synchronizing with

respect to a filter predicate to be to make the collections identical with respect to the current, non-tombstone versions matching the predicate. Note that achieving this may require copying some tombstones and some object versions that do not match the predicate.

A simple distributed algorithm for full synchronization might operate as follows. First, each node sends to the other the full IDs of all current object versions. Whenever either node receives an ID, it checks whether the named object version is *needed* in the local collection, and if so, requests a copy of it from the other node. An object version is needed in a collection if it is not present and it is not superseded by an object version that is present. Finally, if the synchronization session can be left open, change notifications can be used to efficiently keep the two collections in sync for the duration of the session.

First of all, this algorithm requires that it be possible to determine which of the object versions in a collection are current. For this, it is sufficient, given two versions of an object, to be able to determine whether one supersedes the other. (Naturally, one might choose to maintain an index to the current object versions.) However, the algorithm also imposes a stronger requirement, namely that it be possible to determine whether an object version v supersedes another object version v', given the version header for v and just the full ID of v'. This is of course straightforward with the proposed version header design. After checking whether v and v' represent the same object (which is apparent from the full ID), one simply checks whether the version ID of v' is present among those encoded in the `ancestors` attribute of v.

We have confirmed in our XML database prototype that the `key`, `version`, and `ancestors` attributes also suffice to implement synchronization with respect to a filter predicate. Moreover, we believe that the proposed version header will enable probabilistic algorithms (based on hash trees) with data transfer requirements that are sublinear in the number of object versions in the collections, given a bound on the number of differences (versions that are present in just one of the collections).

By including in the version header the identity of the collection at which the object version was originally created, and maintaining at each collection a *vector clock* [4, 11] recording the last logical clock value (or a local serial number) seen from each other collection, it would be possible to quickly determine what needs to be copied over in a full synchronization by simply exchanging vector clocks, much as is done for updates in `refdbms` [5], Bayou [13], and other systems [14]. However, this approach does not work with filter predicates, and we do not wish to require that all devices that store data carrying our version headers store all collections in their entirety.

5. Compactness of the Ancestor List

Here are two rather different examples of how the size in bytes of the `ancestors` attribute can grow linearly with the number of ancestor versions encoded:

- Version 1 is created at some repository, version 2 is derived from version 1 at a different repository, version 3 is derived from version 2 at yet another repository, and so on. Version n will have an `ancestors` attribute of the form

  `<prefix 1>:1 <prefix 2>:1 …<prefix n-1>:1`

In general, the size of the `ancestors` attribute will grow linearly with the number of repositories at which new versions of the object are created.

- Version 2 is derived from version 1, and then new versions are alternately derived from either the last odd version or the last even version, all at the same collection. The result is a pair of versions as follows[2]:

Version ID	`ancestors`
<prefix>:<2*n*+1>	<prefix>:1,3,5,…,<2*n*−1>
<prefix>:<2*n*+2>	<prefix>:1,2,4,6,…,<2*n*>

Note that a join with these versions as parents would have an `ancestors` attribute of "<prefix>:1-<2*n*+2>".

We consider both examples atypical of expected usage.

Consider, on the other hand, typical circumstances where versions of an object are only created at N different collections, each at a different repository, and where conflicts never arise. Then, the `ancestors` attribute of each version of the object will take the form

<prefix 1>:1-<k_1> <prefix 2>:1-<k_2> …<prefix n>:1-<k_n>

where $n \leq N$. For fixed N, this grows logarithmically with the number of ancestor versions encoded.

While we cannot make any such guarantees for arbitrary circumstances, we do expect the `ancestors` attribute to remain reasonably compact in a broad range of practical usage.

6. The Conflict Resolution Problem

Like synchronization protocols, conflict resolution procedures are outside the scope of this paper, but we will again comment on their requirements on the version headers.

We define conflict resolution as a procedure that adds *joins* to a collection C, where the parents of each join must be current in C when the join is added. That is, each join with n parents will reduce the number of object versions participating in conflicts by $n - 1$. The conflict resolution procedure is permitted to leave some conflicts unresolved.

Because synchronizations can copy a join to other collections, the action of a particular conflict resolution process might result in the elimination of a conflict globally, even if other conflict resolution processes decline to handle that conflict. However, it is also possible for a number of conflict resolution processes to create joins for the same object such that the system eventually converges to a state in which each collection contains more conflicting versions of that object than before. The redundancy suppression mechanism defined in Section 3.6 is intended to at least partially control this phenomenon. The following section will elaborate on what it achieves.

Thus far the main focus of our work on conflict resolution has been on enabling the system to behave well in the presence of multiple conflict resolution processes, arbitrating among them when they produce inconsistent results, and avoiding run-away cascades.

[2] Repositories can avoid this problem if they wish by allowing multiple counters per object and basing each new version ID on a counter for which the value immediately following the last one appearing among the ancestor IDs is available.

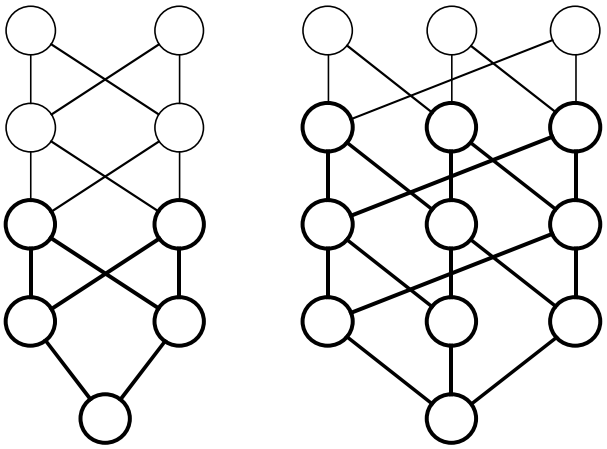

Figure 2. Conflict resolution cascades prevented by redundancy suppression.

The specific conflict resolution procedures with which we have begun to experiment are based on three-way merging of XML documents. Given a collection and an appropriate three-way merge algorithm, we select a pair of conflicting versions v' and v'', attempt to determine their *most recent common ancestor*, v, and then, if three-way merging of $v \rightarrow v'$ and $v \rightarrow v''$ is successful, add the result to the collection as a join with v' and v'' as parents. By common ancestor, we mean the version coming latest in standard total order among those that are ancestors of both v' and v''. If this is not present in the collection, or is not identifiable, we generally leave the conflict unresolved.

7. Convergence of Conflict Resolution

Consider a system with N collections C^i ($i = 1..N$), where the only sources of new object versions from some time t_0 onwards are conflict resolution (Section 6) and redundancy suppression (Section 3.6). That is, object versions may be copied from collection to collection, joins may be created for current parent versions, tombstones may be created by redundancy suppression, and versions may be removed if they are not current, but new ordinary versions are not created.

Let U_t be the set of all object versions that exist in some collection at time t:

$$U_t = \bigcup_{i = 1..N} C_t^i$$

Let U be the set of object versions that exists in some collection at any point in time:

$$U = \bigcup_{t \geq 0} U_t$$

Claim: U is finite. (See the Appendix for a proof.)

It follows from the claim that the system will eventually stabilize in a state such that all conflict resolution processes decline to take any action with regard to the remaining conflicts.

Without the redundancy suppression mechanism, the claim would be false, and the system might never stabilize. Figure 2 shows two examples of cascading joins of joins. In the first, two joins are independently introduced to resolve a two-way

conflict, and then the process repeats ad infinitum. Redundancy suppression prevents this by creating a tombstone for one of the two joins as soon as they appear in the same collection.

The second example demonstrates that it really is necessary to define joins v and v' to be redundant when joined(v) = joined(v'), and not just when parents(v) = parents(v'). Here, three two-way joins are independently introduced to partially resolve a three-way conflict, and then this repeats ad infinitum. Redundancy suppression stops this after the second round.

8. Related Work

Optimistic replication and epidemic algorithms for maintenance of database replicas [2] have a rich history, which has been well surveyed by Saito and Shapiro [14]. Update propagation, as used, for instance, in Bayou [13] and more recently by Hupfeld [6], has been somewhat more widely investigated than version propagation, as used here and, for instance, in Notes [8] and Bengal [3].

As in our architecture, Bengal allows conflict resolution to be postponed, such that "conflicts can be resolved on nodes other than the one that detected the conflict" [3].

The hash history approach [7] is closely related to ours, in that it also labels each version with an explicit list of superseded versions. Using content hashes as version IDs has the advantage of securely binding the version ID to the contents of the object version, providing some protection against introduction of a copy of the object version with modified data. On the other hand, it has the disadvantage that the lists of superseded versions do not compress, requiring an aging policy to control their size.

9. Ongoing Work

We have implemented a replicated XML database in Python along the lines described in this paper. We have experimented with a couple of alternative underlying *native XML databases* [12]: Sleepycat Berkeley DB XML, and a simple dbm-based store without index-based acceleration of XPath queries.

We have ported the latter configuration of our prototype to Nokia's *Python for Series 60* run-time environment for high-end mobile phones, where we run our synchronization protocol over GPRS. We are working on utilizing Bluetooth for synchronizing directly from phone to phone.

Our current synchronization protocol supports filter predicates and is based on exchanging lists of the full IDs of object versions matching the predicate. We intend to further optimize for the case of synchronizing large collections that are almost identical.

10. Conclusion

This paper has described a scheme for labeling versions of objects with a small set of metadata fields in order to enable update-anywhere replicated data storage, such that any data set can be synchronized with any other at any time, in whole or in part. Instead of proposing a specific conflict resolution procedure, the paper has defined a framework to enable simultaneous and uncoordinated execution of multiple conflict resolution processes.

Kenneth Oksanen, Torsten Rüger, Pekka Kanerva, Tero Hasu, and Juha Päivärinta have contributed to the prototype implementation. This work was done in the *Personal Distributed Information Store* (PDIS) project funded by the National Technology Agency of Finland (Tekes) and corporate sponsors Nokia, Hewlett-Packard, Movial, Innofactor, and Fathammer.

11. References

[1] F. Dawson and D. Stenerson, *Internet Calendaring and Scheduling Core Object Specification (iCalendar)*, RFC 2445, Internet Engineering Task Force, 1998.

[2] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry, "Epidemic Algorithms for Replicated Database Maintenance", *Proceedings of the 6th Symposium on Principles of Distributed Computing*, 1987.

[3] T. Ekenstam, C. Matheny, P. Reiher, and G. Popek, "The Bengal Database Replication System", *Distributed and Parallel Databases*, vol. 9, no. 3, 2001.

[4] C. J. Fidge, "Timestamps in Message-Passing Systems that Preserve the Partial Ordering", *Australian Computer Science Communications*, 10(1):56–66, February 1988.

[5] R. A. Golding, "A Weak-Consistency Architecture for Distributed Information Services", *Computing Systems*, 5(4):379–405, 1992.

[6] F. Hupfeld, "Log-Structured Storage for Efficient Weakly-Connected Replication", *Proceeding of the 2004 ICDCS Workshops*, 2004.

[7] B. Kang, R. Wilensky, and J. Kubiatowicz, "The Hash History Approach for Reconciling Mutual Inconsistency", *Proceedings of the 23rd International Conference on Distributed Computing Systems*, 2003.

[8] L. Kawell Jr., S. Beckhardt, T. Halvorsen, R. Ozzie, and I. Greif, "Replicated Document Management in a Group Communication System", *Proceedings of the Second ACM Conference on Computer-Supported Cooperative Work*, 1988.

[9] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator", *Proceedings of the Sixth Annual Workshop on Selected Areas in Cryptography (SAC 99)*, Springer Verlag, 2000.

[10] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", *Communications of the ACM*, 21(7):558-565, July 1978.

[11] F. Mattern, "Virtual Time and Global States of Distributed Systems", *International Workshop on Parallel and Distributed Algorithms*, 1989.

[12] W. Meier, "eXist: An Open Source Native XML Database", *Web, Web-Services, and Database Systems (NODe 2002 Web- and Database-Related Workshops)*, Erfurt, Germany, October 2002. Springer LNCS Series, 2593.

[13] K. Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers, "Flexible Update Propagation for Weakly Consistent Replication", *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, 1997.

[14] Y. Saito and M. Shapiro, *Optimistic Replication*, Microsoft Research Technical Report MSR-TR-2003-60, October 2003.

[15] E. Swierk, E. Kiciman, N. C. Williams, T. Fukushima, H. Yoshida, V. Laviano, and M. Baker, "The Roma Personal Metadata Service", *Mobile Networks and Applications (MONET)*, vol. 7, no. 5, September/October 2002.

[16] *XML Path Language (XPath)*, W3C Recommendation, 1999.

Appendix: Proof of the Claim in Section 7

Let U_{ordinary}, U_{join} and $U_{\text{tombstone}}$, respectively, be the *ordinary*, *join*, and *tombstone* object versions in U:

$$U = U_{\text{ordinary}} \cup U_{\text{join}} \cup U_{\text{tombstone}}$$

We will show that each of these three sets is finite.

Ordinary versions

U_{ordinary} is finite because no ordinary object versions are created from time t_0 onwards.

Joins

Using the definition of joined(v) from Section 3.6, let

$$V_1 = U_{\text{ordinary}}$$

and

$$V_n = \left\{ v \in U_{\text{join}} \mid \left|\text{joined}(v)\right| = n \right\} \quad (n = 2, 3, \ldots)$$

Observe that $\text{joined}(v) \subseteq U_{\text{ordinary}}$ and thus $\left|\text{joined}(v)\right| \leq \left|U_{\text{ordinary}}\right|$ for all $v \in U_{\text{join}}$. Thus $U_{\text{join}} = \bigcup_{i=2..m} V_i$ for some m (namely $\left|U_{\text{ordinary}}\right|$), and to show that U_{join} is finite, it will suffice to show that V_n is finite for all n. This we do by induction on n.

Let $\text{f} : U_{\text{join}} \mapsto U$ such that $\text{f}(v) \in \text{parents}(v)$ and $\text{joined}(\text{f}(v)) \neq \text{joined}(v)$. Such a function must exist because the application of redundancy suppression before creation of a join v by conflict resolution means that $\text{joined}(v') \neq \text{joined}(v'')$ for distinct $v', v'' \in \text{parents}(v)$.

Let V_n^C be the set of object versions in V_n that were originally created at collection C. Because joins are only created for current parents, after which the parents will never again be current in that collection (see the end of Section 2), it follows for distinct $v, v' \in V_n^C$ ($n \geq 2$) that

$$\text{parents}(v) \cap \text{parents}(v') = \varnothing$$

and thus

$$\text{f}(v) \neq \text{f}(v')$$

Moreover, observe that

$$v \in V_n^C \;\Rightarrow\; \text{f}(v) \in \bigcup_{i=1..n-1} V_i$$

In other words, there is a one-to-one mapping from V_n^C to a subset of a finite set, and thus V_n^C is finite. It follows that $V_n = \bigcup_{j=1..N} V_n^{C_j}$ is finite.

Tombstones

From time t_0 onwards, tombstones are only created by redundancy suppression, and only for current joins. The number of tombstones created at a given collection from time t_0 onwards is thus bounded by $\left|U_{\text{join}}\right|$. It follows that $U_{\text{tombstone}}$ is finite.