Tancred Lindholm, Jaakko Kangasharju, and Sasu Tarkoma. 2005. A hybrid approach to optimistic file system directory tree synchronization. In: Ugur Cetintemel and Alexandros Labrinidis (editors). Proceedings of the Fourth ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE 2005). Baltimore, Maryland, USA. 12 June 2005. New York, NY, USA. ACM Press, pages 49-56. ISBN 1-59593-088-4.

http://doi.acm.org/10.1145/1065870.1065879

A Hybrid Approach to Optimistic File System Directory Tree Synchronization

Tancred Lindholm
Helsinki Institute for Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
tancred.lindholm@hiit.fi

Jaakko Kangasharju
Helsinki Institute for Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
jkangash@hiit.fi

Sasu Tarkoma
Helsinki Institute for Information Technology
P.O. Box 9800
FIN-02015 HUT, Finland
sasu.tarkoma@hiit.fi

ABSTRACT

There are two main approaches to optimistic file system synchronization: distributed file systems and file synchronizers. The former type is characterized by a log-based approach that depends on access to file system internals, the latter by a state-based approach that utilizes the standard file system interface, which limits the efficiency of change detection.

We propose a hybrid approach that 1) defines a minor extension to the semantics of the file system interface that enables efficient state-based file system change detection and 2) employs selectively instantiated XML documents to make the use of state-based algorithms for optimistic synchronization feasible on large file systems.

The hybrid approach is simple, well-suited for current file system architectures, and allows us to leverage existing state-based reconciliation algorithms. An initial implementation shows our approach to be feasible, lightweight, and interoperable and to have satisfactory performance.

Categories and Subject Descriptors

C.2.4 [**Computer Communication Networks**]: Distributed Systems; D.4.3 [**Operating Systems**]: File Systems Management

General Terms

Algorithms, Design, Performance

Keywords

optimistic synchronization, stackable file system, directory tree, state-based, reconciliation, XML

1. INTRODUCTION

Data synchronization is one of the fundamental research areas in mobile computing, with synchronization of file systems being of particular practical importance. In this work we focus on optimistic synchronization of file systems [3] on limited devices, such as mobile phones or PDAs.

These devices typically operate in a weakly connected environment, where connectivity is intermittent and bandwidth often limited, for both technical and monetary reasons. The optimistic approach to synchronization is preferred in this environment, due to the high data availability and relative resilience to network disturbances it offers [11].

The industry seems to mainly have focused on synchronization of databases. In contrast, we note that file storage capacity on mobile phones already reach several gigabytes (e.g. the Samsung SGH-i300 mobile phone). We believe that synchronization of file systems on mobile devices will become increasingly important in the future.

In this paper, we present the directory tree synchronization method we developed as a part of a framework for pairwise optimistic file synchronization between mobile devices. Specifically, of the three aspects of synchronization [3], we address the areas of *update detection* and *reconciliation*, while mentioning update propagation only cursorily.

Update detection, in the context of file system synchronization, is the task of finding the changes made to the directory tree since the last point of synchronization. The prevalent methods are to either keep a log of file operations or scan the directory tree for changes prior to synchronization. Reconciliation is performed either on the file system operation logs or on the states of the file systems at the time of synchronization.

We may roughly categorize systems for optimistic file system synchronization into two classes according to their update and reconciliation strategies: *distributed file systems*, that log file operations and reconcile these logs, and *file synchronizers* that scan the file system for changes and reconcile its state. The Coda distributed file system [12] is a well-known representative for the former category, and the Unison [10] and rsync [13] file synchronizers for the latter.

Our work represents a hybrid approach that performs update detection by scanning and reconciliation by state (like file synchronizers), but uses a minor file system interface augmentation in combination with state in a format that focuses on *changes* (like a distributed file system). The approach aims to offer acceptable synchronization performance on large file systems, while minimizing the amount of file system interface expansion.

Increasing storage capacity raises a demand for efficient update detection. The logging approach requires extended

MobiDE'05, June 12, 2005, Baltimore, Maryland, USA.

```
<tree id="0">
 <directory name="dev" id="01" />
 <directory name="bin" id="03">
   <file name="ls" id="04" />
 </directory>
 <directory name="home" id="05">
   <directory name="fred" id="09">
     <file name="dirmerge.lyx" id="0m" />
     <directory name="photos" id="0a" >
       <file name="01.jpg" id="0d" />
     </directory>
     <directory name="docs" id="0c" >
       <file name="paper.lyx" id="0n" />
     </directory>
   </directory>
 </directory>
 <directory name="usr" id="06">
   <directory name="src" id="07">
     <file name="linux-2.6.tgz" id="08" />
   </directory>
 </directory>
</tree>
```

Figure 1: The initial directory tree T_0 on d_1 and d_2

file system functionality for gathering the logs, and imposes constant CPU drain during file system usage. The scanning approach, being limited by the standard file system interface, suffers from the *excessive scanning problem*: the need to scan the entire file system regardless of the actual number of changes. The file system interface augmentation addresses both of these issues.

The state approach to reconciliation is in many cases the more straightforward and obvious design choice. It is, however, of limited applicability on large file systems, due to the size of the states that need to be processed and communicated. Our approach significantly alleviates this inefficiency.

Furthermore, by choosing to reconcile state rather logs, we were able to leverage and build upon previous work on using three-way merging for mobile reconciliation [6]. In particular, we may reconcile using the three-way merging algorithm [7] that is part of our synchronization framework, despite the size of the directory tree exceeding working memory by several orders of magnitude.

The remainder of the paper is organized as follows. We start by giving basic definitions and introducing a straightforward XML model for directory trees. We then present a directory tree synchronization use case, which is used as a running example throughout the rest of the paper. In sections 3 and 4 we present our method for update detection, and a format for expressing its output in a compact manner. In section 5 we describe how to efficiently reconcile the updated directory trees, and also briefly look at the issue of directory tree versioning. We examine the implementation in section 6 and look at initial results on the feasibility of our approach in section 7. We discuss and relate our approach to existing work in section 8, and conclude in section 9.

2. NOTATION AND EXAMPLE SCENARIO

We model the file system directory trees as XML documents. The root element is <tree>, directories are expressed with the <directory> element, and files with the <file> element. Each element has a unique identifier, which relates the same elements across documents. The identifier is encoded

```
<tree id="0">
 ⋮ Subtrees 01, 03 same as T0
 <directory name="home" id="05">
   <directory name="fred" id="09">
     <!-- dirmerge.lyx moved away from here -->
     ⋮ Subtree 0a same as T0
     <directory name="docs" id="0c" >
       <file name="dirmerge.lyx" id="0m" />
       <file name="paper.lyx" id="0n" />
     </directory>
   </directory>
 </directory>
 ⋮ Subtree 06 same as T0
</tree>
```

Figure 2: The modified directory tree T_1 on d_1

```
<tree id="0">
 ⋮ Subtrees 01, 03 same as T0
 <directory name="home" id="05">
   <directory name="fred" id="09">
     <file name="mobide05.lyx" id="0m" />
     <directory name="photos" id="0a" >
       <!-- 01.jpg deleted -->
       <file name="show.jpg" id="0i" />
     </directory>
     ⋮ Subtree 0c same as T0
   </directory>
 </directory>
 ⋮ Subtree 06 same as T0
</tree>
```

Figure 3: The modified directory tree T_2 on d_2

in the id attribute, and the names of files and directories in the name attribute.

We illustrate our directory synchronization method with an example synchronization scenario. In the scenario, we assume two devices, d_1 and d_2, which initially exhibit the same directory tree, denoted T_0. This directory tree is then independently changed on d_1 and d_2. The changed trees on d_1 and d_2 are denoted T_1 and T_2, respectively.

The initial directory tree of the example is shown in figure 1, and the trees T_1 and T_2 are shown in figures 2 and 3. On d_1, we move the file dirmerge.lyx from /home/fred to /home/fred/docs. On d_2, we change the /home/fred/photos directory by deleting 01.jpg and inserting snow.jpg. We also rename dirmerge.lyx in /home/fred to mobide05.lyx.

The example entails each of the four file operations that we recognize: inserting, deleting, moving, and renaming a file. We emphasize that the directory trees in the example are only for the purpose of illustrating our method as they are several orders of magnitude smaller than the trees found on typical workstation and server file systems.

The synchronization process proceeds as shown in figure 4 (assuming reconciliation happens on d_1). The star and plus forms (T_0^*, T_0^+ etc.) of the trees indicate different representation formats of the same tree. The trees T_1 and T_2 are implicitly stored in the file system directory trees, whereas the base tree T_0 is kept in a simple database.

First, (step 1) the updated tree is read off the file system on d_1 and d_2 in an optimized form, called XML-with-

references (see section 4). The updated tree from d_2 is propagated to d_1 over the network, after which the trees undergo a normalization phase (step 2, see section 5.1). The trees are then fed into the XML three-way merger (step 3).

The task of the three-way merger is to *detect* the changes between T_1^+ and T_0^+, as well as between T_2^+ and T_0^+, then *merge* these changes, and apply them to T_0^+, thus producing the reconciled directory tree T_{m}^+. As the three-way merger we use the "3dm" algorithm described in [7].

The merged tree is then applied (step 4) to the file systems on d_1 and d_2; in the case of d_2 it first has to be propagated through the network. The synchronization cycle is completed by designating the merged tree T_{m}^+ as the new base tree T_0 (step 5).

After synchronizing the directory tree, the synchronization framework continues by synchronizing any changed files in a similar fashion. The file synchronization procedure is, however, not described in this paper.

A practical implementation of the XML directory tree model on any hierarchical file system is straightforward, except for the maintenance of unique element identifiers. A simple approach on Unix-like systems is to use the inode number as identifier, as long as there are no two entries linked to the same inode (hard links). For conciseness, we assume this to be the case throughout this paper. Hard links are discussed further in section 6.

3. UPDATE DETECTION

To remedy the excessive scanning problem we propose a simple extension to the semantics of the modification timestamp maintained by legacy file systems. Instead of indicating the last time a file or directory was modified, we propose that the timestamp should indicate the last time an entry or (for directories) *any of its descendant entries* was modified.

To implement this semantics we need to percolate modification time changes all the way up to the directory tree root. To reflect this, we call time stamps adhering to this extended semantics *bubbling modification timestamps* (BMTs).

When using BMTs we still scan the file system for changes from the root, but only need to enter directories whose modification timestamp is more recent than the time of the last synchronization run. Other directories need not be visited since, by definition, there will not exist any changes more recent than the directory timestamp in the descendants of that directory. We call this method *selective scanning*.

An entry whose modification timestamp is more recent than the time of the last synchronization is said to be *tainted*. Selective scanning may also be used to find updates to the contents (as opposed to the name) of files by tainting any file entry whose storage object has been modified.

Consider our example synchronization scenario. On d_1, we changed the contents of the /home/fred and /home/fred/docs directories. According to the selective scanning method we taint these directories, as well as the directories / (the root) and /home. On d_2, we changed the contents of /home/fred and /home/fred/photos. Here, the selective scanning method require that we, in addition to these, taint the / and /home directories.

The main drawback of the selective scanning method is that a single file system operation normally causes several directories to be scanned, making the method more expensive in terms of file system operations than the logging method. On the other hand, file system usage usually exhibits a great amount of locality in terms of which files and directories are modified. For the selective scanning method, this means that we expect the average amount of directory entries scanned per file system operation to decrease as the number of operations increase. Furthermore, the directory tree needs to remain unchanged during scanning, as e.g. a move may show up as a delete if first the target directory is scanned, followed by the move, and then scanning of the move source directory.

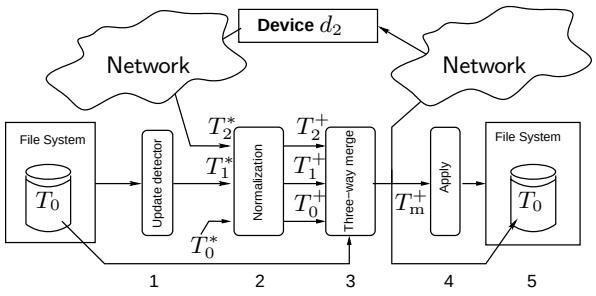


Figure 4: The directory synchronization process

A benefit of the selective scanning method is that it is limited by the size of the file system: in the worst case the full file system needs to be scanned. The logging method, on the other hand, may theoretically generate an arbitrary amount of data that is not correlated to the size of the final file system state.

A typical example of this is heavy temporary file activity between synchronization runs. Here, the selective scanning method may need to only scan a single directory and its parents, whereas the corresponding log may contain a large amount of file inserts and deletes that cancel each other out. Indeed, file systems that log file operations usually implement some level of log optimization (e.g. [12]) to remove entries that are not relevant to the final outcome of the file system state. Selective scanning, on the other hand, requires no such functionality.

We are not aware of any file system that implements the proposed extension to the modification timestamp. It appears that a full implementation would be quite expensive in terms of file operations, as any modifying operation would need to not only update the timestamp of the corresponding entry, but also the timestamp of each directory on the path to the root.

In our case, we may perform selective scanning with less specific information, as we only need to know whether an entry has been tainted since the last synchronization or not. A method to implement this "taint bit", which we call *relaxed BMT*, is to use a cache of recently tainted entries. When an entry is modified, we traverse towards the root, updating timestamps and adding entries to the cache as we go along. The traversal stops as soon as an entry already in the cache is encountered. The cache is purged (e.g. by an ioctl) after each synchronization run. We chose to implement this method due to its simplicity and small overhead.

4. XML-WITH-REFERENCES

In order to be able to process T_1 and T_2, we need to express these compactly, as well as in a manner compatible with the selective scanning process. We make use of the fact

```
procedure selectiveScan(d: Entry)
 1: if ¬tainted(d) then
 2:    output(<ref-tree... />)
 3: else if isfile(d) then
 4:    output(<file ... />)
 5: else
 6:    output(<directory ... >)
 7:    for all e ∈ entries(d) do
 8:       selectiveScan(e)
 9:    end for
10:    output(</directory>)
11: end if
endproc
```

Figure 5: XMLR selective scanning

```
<tree id="0">
  <ref-tree id="01" />
  <ref-tree id="03" />
  <directory name="home" id="05">
    <directory name="fred" id="09">
      <ref-tree id="0a" />
      <directory name="docs" id="0c" >
        <file name="dirmerge.lyx" id="0m" />
        <ref-tree id="0n" />
      </directory>
    </directory>
  </directory>
  <ref-tree id="06" />
</tree>
```

Figure 6: The XMLR directory tree T_1^* on d_1.

that untainted entries contain no changes from the base version, and express these using elements that reference roots of subtrees in the base version T_0.

More specifically, let the <ref-tree id="*i*" /> tag signify a reference to the subtree rooted at the element identified by i in T_0. A reference element may not have any children, as it implicitly states its descendants.

We call an XML document containing such reference elements an *XML-with-references* (XMLR) document. An XMLR document should be interpreted as the document obtained when all references are replaced with the subtrees they refer to, including ones revealed by replacement.

The reference elements can be thought of as placeholders for subtrees whose evaluation has been delayed. In accordance with this, we refer to the reference elements as *placeholders*, and any element that belongs to the subtree referenced by such a placeholder as *virtual*. Any other element in the document is said to be *instantiated*.

Figure 5 shows a simple recursive algorithm for selective scanning that outputs the scanned tree as an XMLR document. Applying the algorithm on d_1 in our example yields the XMLR document T_1^* shown in figure 6. Similarly, we obtain T_2^* on d_2. The starred (i.e. T^*) notation is used throughout the paper to identify XMLR documents.

Virtual elements in figure 6 are e.g. those identified by 07 and 08, whereas the element 06 is a placeholder, and 0 is instantiated. In the example selective scanning saves traversing 5 of the 9 directories on both d_1 and d_2.

Although the concept of reference elements is very simple, we find that it quite effectively allows us to focus XML processing on the *changed* parts of the documents. Furthermore, the technique is usable throughout the whole of the directory synchronization process, as we will see in the following sections.

5. RECONCILIATION

The next step (step 3 in figure 4) is to reconcile the XMLR documents for the updated trees T_1 and T_2 into the merged directory tree T_m. As an aid for the reconciliation process we have the base tree T_0.

The straightforward approach of merging these documents will not work. Consider, for instance, a case where
T_0=<r id="r"><b id="b"/></r>,
T_1^*=<ref-tree id="a" /> , and
T_2^*=<r><ref-tree id="b" /></r>.
Being unable to relate elements in their placeholder and instantiated forms, we find that the root node has been changed to <ref-tree>, and that the root node children have been erased in T_1^* and replaced with <ref-tree id="b" /> in T_2^*. These findings are clearly flawed, and hence the merge fails (3dm emits a conflict on this input).

To resolve the incompatibilities between XMLR documents and the merging algorithm we may either change the algorithm to account for reference elements, or alternatively, pre-process the XMLR documents into a format suitable for merging with an unmodified version of the algorithm.

We chose the latter alternative. The task now becomes finding a format that is compatible with the merge, but yet retains the compactness of XMLR.

5.1 XMLR Document Normalization

The main issue to address is the failure to recognize equality between elements in their instantiated and placeholder forms. Another is how to deal with the implicit structural relationships that exist between references when one XMLR document references an element that is virtual in another document.

Both may be remedied by introducing the following concept of normalizing a set of XMLR documents:

Definition 1 A set of XMLR documents $\{T_0^*, T_1^*, \ldots\}$ with uniquely identified elements is normalized when, for each element identifier i, the state (*instantiated*, *placeholder* or *virtual*) of the elements identified by i is consistent across all documents in the set that contain i. The members of a set that has been normalized are denoted $T_0^+, T_1^+, \ldots$

Consider the set $\{T_0, T_1^*\}$ in the context of our example. The elements identified by 09 are instantiated in both documents, so their state is consistent. The elements identified by 01, on the other hand, do not have a consistent state (placeholder in T_1^*, but instantiated in T_0), and hence the set is not normalized.

Normalization removes the need to recognize equality between elements in their placeholder and instantiated forms, as elements will appear in one form only in all documents in the set. Furthermore, there will be no references to virtual elements, which means that each placeholder element (equating copies across the documents) is a unique reference to the subtree it refers to.

We believe that these properties make the normalized XMLR document set suitable for direct input to a wide variety of XML processing algorithms. In our own research we

procedure instantiate(inout T, T_0: Tree, i: node id)
1: instantiate(T,parent(T, i)) **if** i not instantiated
2: $n' := T_0[i]$ with children in placeholder state
3: replace($T_0[i], n'$)
endproc

procedure normalize(inout $\mathcal{T}$: setof Tree, in T_0: Tree)
4: **repeat**
5: i :=id of node in $\mathcal{T}$ with inconsistent state
6: **if** $\exists n \in \mathcal{T}[i]$ so that $n.state$ = instantiated **then**
7: instantiate(T, T_0, i) **forall** $T \in \mathcal{T}$
8: **else** { *Force i to placeholder state* }
9: instantiate(T, T_0,parent(T, i)) **forall** $T \in \mathcal{T}$
10: **end if**
11: **until** all nodes in $\mathcal{T}$ in consistent state
endproc

Figure 7: Normalization algorithm

have, for instance, used this property when building XML delta documents.

A straightforward way of implementing normalization is shown in figure 7. The *instantiate*() procedure is used to force nodes into instantiated and placeholder states, the latter being achieved by instantiating the parent of a node. The normalization procedure looks for nodes in an inconsistent state, and instantiates these into instantiated or placeholder states, until no more inconsistent nodes are found. Note the similarity between BMT maintenance and node instantiation: both visit the nodes on the path to the root.

The instantiated node forms are fetched from the common reference target tree T_0. Since nodes may be fetched by an arbitrary id (line 2 in figure 7), the T_0 persistent storage mechanism needs to support looking up elements by their id. The persistent stores for T_1 and T_2, on the other hand, only need to support a depth-first traversal access pattern, which allows the file system directory trees to be used to store these directly without any additional indexing information.

The computational complexity of normalization is determined by the need to look up nodes from T_0. With a lookup cost of $O(\log m)$ and k lookups, the cost of normalization becomes $O(k \log m)$, where m is the size of T_0. Assuming a constant number of input trees, each node may only be instantiated $O(1)$ times, yielding $k \leq n + O(m)$, where n is the combined size of the input XMLR trees.

5.2 Merging, Conflicts, and Validation

We are now set to merge the changed trees. One detail not to overlook is that we cannot use T_0, but have to use its XMLR counterpart T_0^*, i.e. <ref-tree id="0" /> in the document set to normalize. We do this for two reasons: T_0 is a huge document that we should never instantiate, and T_0 in the set-to-normalize would cause all placeholder elements to be instantiated (since all elements in T_0 are instantiated). Thus, to reconcile we let

$$\begin{aligned} \{T_0^+, T_1^+, T_2^+\} &= \text{normalize}(\{T_0^*, T_1^*, T_2^*\}, T_0) \\ T_{\text{m}}^+ &= \textsf{3dm}\ (T_0^+, T_1^+, T_2^+) \end{aligned}$$

The normalized trees T_0^+, T_1^+, T_2^+, are shown in figure 8, where boxed nodes represent references, and shading nodes added by normalization. The merged tree T_{m}^+ is shown in figure 9. The merger is responsible for performing conflict

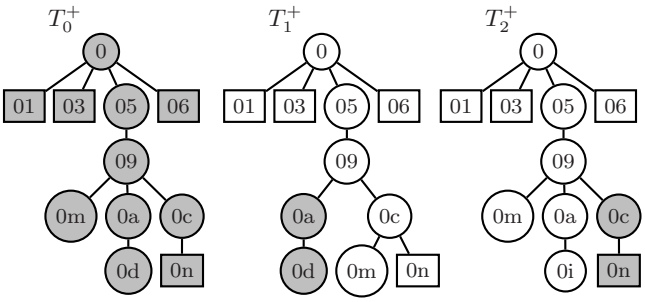


Figure 8: The normalized trees T_0^+, T_1^+, and T_2^+.

```
<tree id="0">
 <ref-tree id="01" />
 <ref-tree id="03" />
 <directory name="home" id="05">
  <directory name="fred" id="09">
   <directory name="photos" id="0a">
    <file name="snow.jpg" id="0i" />
   </directory>
   <directory name="docs" id="0c">
    <file name="mobide05.lyx" id="0m" />
    <ref-tree id="0n" />
   </directory>
  </directory>
 </directory>
 <ref-tree id="06" />
</tree>
```

Figure 9: The merged tree T_{m}^+.

detection, and (potentially) resolution, as well as checking the result for validity.

The default 3dm conflict processing needs only minor augmentations to deal with directory trees [6]. The main augmentation is the validity check, which handles e.g. name collision situations, where entries with different element ids but the same name occur in a directory.

The name collision issue is the hardest check to perform, because the normalized form obtained with the default entry tainting will not necessarily include enough instantiated nodes to access the required name attributes. To remedy this, we may opt to always taint all entries in a directory whenever a directory is changed, or to look up the missing names from T_0. The latter approach, which we implemented, is somewhat dissatisfactory in that it requires the validation step to be XMLR-aware.

5.3 Applying the Merged Document

In the final phase of synchronization, we apply the merged tree T_{m}^+ on d_1 and d_2. Here, we are faced with a mismatch in the nature of the file system interfaces: whereas the standard read interface is state-based, the write interface is operation-based. Thus, we need to find a set of file operations that restructures the directory trees on the target devices according to T_{m}^+.

To accomplish this, we start traversing T_{m}^+ and the target directory tree from their respective roots. At each node, the children of the node are compared to the contents of the corresponding directory. Entries only in T_{m}^+ are either inserted or the destination of a move, and entries only in the target directory are either deleted or the original location of a moved entry. Entries whose name does not

procedure apply($n \in T_{\mathrm{m}}^{+}$: Node; p: FilePath)
1: $n_c := \text{children}(n)$, $n_p := \text{children}(p)$
2: **for all** entries $e \in n_c \cup n_p$ **do**
3: **if** $e \in n_c$ and $e \in p_c$ and renamed(e) **then**
4: rename(p/e)
5: **else if** $e \notin n_c$ and $e \in p_c$ and $e \notin T_{\mathrm{m}}^{+}$ **then**
6: delete(p/e)
7: **else if** $e \in n_c$ and $e \notin p_c$ and $e \notin T_1^{+}$ **then**
8: insert(p/e)
9: **else if** $e \in n_c$ and $e \notin p_c$ and $e \in T_1^{+}$ **then**
10: move(getPath(e, T_1^{+}),p/e)
11: **end if**
12: **if** isdirectory(p) **then**
13: apply(e,p/e)
14: **end if**
15: **end for**
endproc

Figure 10: Algorithm for applying T_{m}^{+} on d_1. On d_2, substitute T_2^{+} for T_1^{+}.

match have been renamed. To disambiguate between moves and inserts/deletes, we look up the entry in the normalized XMLR document for the original tree. The apply algorithm is shown in figure 10. The initial call is $apply(root(T_{\mathrm{m}}^{+}),/)$.

One important detail is that it is sufficient to look at only non-virtual elements in the trees when looking up entries, and hence no index of the nodes in T_{m} or T_1 is needed. This property, which is of large practical importance, follows from the normalization of the input trees and the fact that all nodes that are looked up by the algorithm are non-virtual.

As an alternative to applying the merged tree as a list of file operations, a state-based write interface for file system directories could be imagined. With such an interface, tree application would essentially be a matter of writing the child list of each directory node in T_{m}^{+} into the corresponding directory file.

5.4 Directory Tree Revision Control

In the synchronization framework there is a need to retain a version history of directory trees. The normalized XMLR documents calculated for the reconciliation phase may be used to obtain sufficiently compact XML deltas for the directory trees.

As is common in version control systems, we maintain the version repository as a chain of reverse deltas, i.e. deltas that when applied to version n yields version $n-1$. In our case, the tree T_0 acts as version $n-1$ and T_{m}^{+} as version n. T_{m}^{+} effectively expresses a forward delta from version $n-1$ to n. To obtain the reverse delta, we take T_0^{+} and instantiate the subtrees for all placeholder elements that are not present in T_{m}^{+} (i.e. subtrees deleted in T_{m}^{+}). The easy generation of deltas is an additional benefit of the normalization process.

6. IMPLEMENTATION

The BMT functionality was implemented as a stackable file system [4] using the FiST [17] framework on Debian GNU/Linux 3.0 with a series 2.4 kernel. The framework provides an identity stackable file system code template, base0fs, to which we added code for maintaining both full and relaxed BMTs. This required around 150 lines of C code, half of which was dedicated to keeping a simple 32-entry LRU-like cache of recently tainted inode numbers.

File systems do not generally maintain information in an inode about which directories contain entries for that inode. This may cause problems with implementations, as file modifications may become "anonymous": we only know the inode number of a modified object, but not any pathname that resolves to that particular object. The Linux kernel directory entry cache is, however, structured in such a way that we were always able to obtain pathnames for modified inodes.

When this is not the case, a solution is to maintain a set of modified file inode numbers. After the updated directory tree has been read off disk, we may obtain pathnames to the modified files by looking up the modified inode numbers from T_1^{*} (complemented with T_0 for virtual elements).

The other parts of the synchronizer are written in Java. To provide random access by element id, we used a simple persistent string dictionary for T_0, and in-memory lookup tables for the XMLR trees, as these are of limited size and need to be kept in memory anyway.

A practical matter of the update detection phase is how to handle the modification timestamp for file entries. As pointed out in [3], changing the name of a directory entry will by default not mark it as modified. Our BMT implementation could trivially implement the required tainting, but then another problem would arise: marking renamed file entries modified would imply that their content has changed, thus breaking content modification detection.

We solve this dilemma by keeping the standard semantics for the modification timestamp for file entries. To compensate, we identify renamed entries during the update detection phase by comparing the names of the scanned entries to the names stored in T_0.

We do not currently support storage objects (inodes) for which more than one entry exists. The occurrence of such storage objects break the use of inode numbers as unique entry identifiers, forcing us to find alternate methods to obtain these. We have yet to identify one that works to our satisfaction. We note, however, that symbolic links are allocated unique inodes, and thus the issue occurs with "hard" links only.

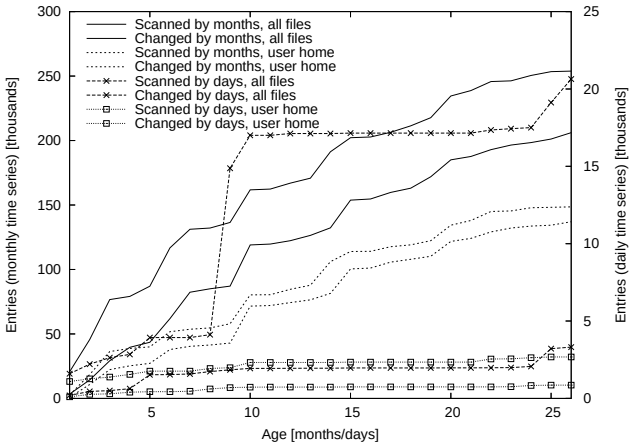
The synchronizer has been used to synchronize files across the Linux and Windows operating systems. The selective scanning mechanism is, however, currently only implemented on Linux.

7. EXPERIMENTAL EVALUATION

To test our approach, we did some initial evaluation of the amount of directory tree traversal and of the cost of maintaining BMTs.

In the directory tree traversal tests we compared the number of modified entries to the number of entries that were visited when detecting changes using the selective scan approach. The data was gathered from the live file system on one of our workstations. The size of the full file system was around 400,000 entries, of which 200,000 entries were in a user's home directory.

Figure 11 shows the relation between modified and selectively scanned entries during cumulative periods of 1–26 months and 1–26 days. We see that the absolute overhead remains fairly constant, and thus the relative overhead increases as the number of modified entries decrease. We also



Note: The series form pairs of changed and scanned values, as emphasized with the line styles. For each pair, changed $\leq$ scanned.

Figure 11: Results of the directory traversal test.

tested by scanning the daily changes during a 10-day period, which revealed a quite large variance in the ratio of scanned to modified entries: 2–30 times for modification batches of up to 3000 entries.

We observe that selective scanning may be quite sensitive to the location of the modifications. The large jump in scanned entries around day 9 in figure 11 was due to a new binary being installed, an operation which added a few files to otherwise large directories like /usr/bin (2000 entries) and /var/lib/dpkg/info/ (4500 entries).

Due to such events, we find it hard to generalize our results. An intuitive estimate is that we may expect a baseline cost of scanning a few percent of the file system, after which an overhead of scanning up to about one tenth of the entries gradually sets in.

We note that the number of modified entries is directly proportional to the size of an optimized operation log. Similarly, the number of scanned entries correspond to the size of the XMLR input trees. Use of the test results yield that tainting/instantiating n nodes from T_0 would yield a lookup set of $n+c$ nodes, where c is the constant overhead observed. Normalization would then require $O(n \log m)$ time and produce output trees of size $O(n)$, since $k \leq 2n + c$. As the subsequent merging costs $O(n \log n)$ [7], we may argue that our approach has a complexity of $O(n \log m)$ and, assuming a lower bound of $\Omega(n)$ for logging implementations, that the relative speed of our approach compared to logging should be bounded by $O(\log m)$.

To test the cost of maintaining the BMTs, as well as to stress-test our implementation, we timed the build process of the Linux 2.4 kernel, as distributed in the Debian kernel-source-2.4.18 package version 14.3. The test consisted of rounds of consecutive executions of make clean, make dep and make using the default Debian kernel configuration. We tested by performing a warmup round, followed by the timed execution of 9 consecutive rounds. The test was run on an otherwise empty ext2 file system, as well as with base0fs and our update detector in full and relaxed BMT configurations stacked on top of the ext2 file system.

The results are shown in table 1. We find that the baseline stackable file system incurs an overall task slowdown of

Table 1: Cost of maintaining BMTs.

	System + I/O time			Wallclock time		
File system	$\bar{t}$ [s]	σ [s]	η [%]	$\bar{t}$ [s]	σ [s]	η [%]
ext2	8.87	0.19		120.26	0.20	
base0fs	10.22	0.40	15.22	121.77	0.28	1.26
BMT, full	10.16	0.26	14.54	121.65	0.02	1.16
relaxed	10.15	0.23	14.43	121.58	0.02	1.10

$\bar{t}$=average time, σ=standard deviation, η=overhead to ext2

around 1%, and about 15% increase in system and I/O time, which is consistent with numbers reported in [17]. Comparing our approach to the baseline, we find that the overhead of the relaxed update detector is dwarfed by that of the baseline — in fact, some numbers show a slight *increase* in performance.

The numbers for the full BMT approach are quite surprising: the overhead remains minimal, despite the large number of directory tainting. We theorize that the frequent accesses to the inodes on the path to the root helped retain relevant entries in the directory entry cache, thereby compensating for the frequent updates. This potentially also explains the lower standard deviations of the BMT approaches.

The kernel build benchmark, although easily reproducible, is not ideal for testing file system performance due to the significant amount of user CPU time. Nonetheless, it still gives us a good indication of the overhead of our approach in relation to the baseline. In order to measure the impact of BMT only, one would need to measure natively implemented BMT extensions, which would not suffer from the overhead of the stacking layer.

8. DISCUSSION AND RELATED WORK

Synchronization in distributed file systems [12, 14] is generally a built-in part of a specialized file system that performs file operation logging and has extensive access to file system internals. Implementations typically provide a one-stop solution for local file storage, data distribution model, security, and administration. Interoperability may be limited by dependencies on file system internals specific to a particular operating system.

In contrast, we find that our approach allows for a highly modular implementation, where one may freely add synchronization functionality on top of any file system. Furthermore, our synchronization protocol remains highly interoperable, since our extension to the file interface does not expose any internal data from the file system.

Looking at file synchronizers, we find the Unison model of synchronization [3, 10] highly relevant to our research. The model gives a theoretical framework for file synchronization, and defines a set of desirable properties for synchronizers within this framework. Update detection and reconciliation are described from theoretical and practical points of view.

Our notion of a tainted entry is in many ways similar to the Unison *dirty* predicate for file pathnames. Both concepts indicate paths to changed entries, and have the property of being up-closed, i.e. ancestors of a dirty/tainted entry must also be dirty/tainted.

Moved entries are handled differently in our approach. Since objects are identified by their path in [3], there is the possibility of a re-mapping between a path and its storage object (inode), leading to an implicit content update. For

instance, consider the operations deltree /home/alice and move /home/fred /home/alice. Here, the contents of any paths starting with /home/alice are implicitly updated by the change of storage object, which implies that paths starting with /home/alice have to be marked dirty.

In our model, we do not have to do this. We may consider the content that is synchronized to be a set of mappings from inodes to (parent inode, name, modtime) triplets. Since moving a subtree does not change any mappings for nodes *inside* the subtree, we find that we need not mark descendant entries dirty.

The use of data structures with delayed evaluation, of which XMLR is an example, is a basic technique in computer science. Its application to the domain of XML has been studied formally [9] as well as implemented in practice in the "lazy" mode of the Xerces [2] XML parser.

The dynamic XML documents of the Active XML framework [1] define an XML encoding for delayed content by providing tags for embedded web service invocations. These act as a generalized version of our placeholder elements. Furthermore, the XML linking and inclusion standards XInclude [16] and XLink [15] both provide the semantics to express our placeholder elements, should this prove beneficial.

The selective instantiation of placeholder elements so that a set of XMLR documents is processable by XMLR-unaware algorithms is a distinguishing feature of our approach. In particular, our notion of normalizing the references in a set of documents is, to the best of our knowledge, a new concept.

Our approach allows a variety of merging algorithms to be used during the reconciliation phase, as long as they are based on three-way merging. Applicable algorithms are the 3dm algorithm, DeltaXML [5], as well as (to a lesser extent) the ubiquitous text-based tools diff and patch. Structured three-way merging on a more general level has been surveyed in [8]. We used 3dm because it easily accommodates unordered trees, supports moves, and is familiar to us.

9. CONCLUSIONS AND FUTURE WORK

The hybrid update detection and XML processing techniques presented in this paper form a novel approach to directory tree synchronization. Our update detection method, based on introducing the "bubbling" semantics for directory modification timestamps, is a feasible and scalable way to locate changes to file systems, while introducing only minimal changes below the file system level. The XML-with-references (XMLR) techniques allow us to process documents that mainly encode *changes* to state, rather than the full state, throughout the synchronization process. The techniques are compatible with the state-based XML reconciler that is part of our synchronization framework.

Our approach should be well suited for file synchronization on limited devices, where storage capacity is expected to grow quickly. On such devices, we think it will be easier and more feasible to implement our update detection method rather than methods for logging file system changes.

In future work, we intend to look at methods to execute our approach incrementally. File transfers over the network may in several cases be initiated as soon as the first modified file entry is encountered, and directory tainting allows us to locate such an entry quickly. Furthermore, the state of large change sets may not fit into memory, in which case it would be beneficial if an incremental approach were available.

10. REFERENCES

[1] S. Abiteboul, A. Bonifati, G. Cobéna, et al. Dynamic XML documents with distribution and replication. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 527–538. ACM Press, 2003.

[2] *Xerces: XML parsers in Java and C++*. `http://xml.apache.org`.

[3] S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom)*, pages 98–108, 1998.

[4] J. S. Heidemann and G. J. Popek. File-system development with stackable layers. *ACM Trans. Comput. Syst.*, 12(1):58–89, 1994.

[5] R. la Fontaine. Merging XML files: a new approach providing intelligent merge of XML data sets. In *Proceedings of XML Europe 2002*, Barcelona, Spain.

[6] T. Lindholm. XML three-way merge as a reconciliation engine for mobile data. In *Third ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 93–97, September 2003.

[7] T. Lindholm. A three-way merge for XML documents. In *Symposium on Document Engineering (DocEng) 2004*, Milwaukee, WI, USA, October 2004.

[8] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, May 2002.

[9] M. L. Noga, S. Schott, and W. Löwe. Lazy XML processing. In *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*, pages 88–94. ACM Press, 2002.

[10] B. C. Pierce and J. Vouillon. What's in Unison? A formal specification and reference implementation of a file synchronizer. Technical Report MS-CIS-03-36, Dept. of Computer and Information Science, University of Pennsylvania, 2004.

[11] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[12] M. Satyanarayanan and J. Kistler. Disconnected operation in the Coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, February 1992.

[13] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Department of Computer Science, The Australian National University, Canberra 0200 ACT, Australia, June 1996.

[14] B. von Hagen. *Using the InterMezzo Distributed Filesystem — Getting Connected in a Disconnected World*, August 2002. `http://www.linuxplanet.com/linuxplanet/reports/4368/1`.

[15] W3C. *XML Linking Language (XLink) Version 1.0*, June 2001. [Recommendation] `http://www.w3.org/TR/xlink`.

[16] W3C. *XML Inclusions (XInclude) Version 1.0*, December 2004. [Recommendation] `http://www.w3.org/TR/xinclude`.

[17] E. Zadok and J. Nieh. FiST: A Language for Stackable File Systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 55–70, San Diego, CA, June 2000.