Format-Independent Change Detection and Propagation in Support of Mobile Computing

Michael Lanham, Ajay Kang, Joachim Hammer, Abdelsalam Helal, and Joseph Wilson

301 CSE Building, BOX 116120
University of Florida
Gainesville, FL 32611-6120, U.S.A

Abstract

We present a new approach to change detection and propagation in order to update copies of documents, which are stored on various devices. Our approach, called Format-Independent Change Detection and Propagation (FCDP), is capable of computing the changes that have been made to a document on one device and applying the net effect to an unedited copy, which uses a different format and representation. For example, this allows users of mobile devices to modify documents, which have been generated by more powerful applications on different platforms, and to have the modifications reflected in the originating documents.

Our algorithm, which has been implemented and tested, improves upon current change detection techniques since it does not require access to all content and metadata of the original document. Furthermore, the application of changes across differently formatted documents is automatic, requiring no user initiation like current desktop-to-PDA conversion products, and transparent, eliminating the user-directed translations steps found in commercial products.

1. Introduction

In today's networked computing environment, users demand constant availability of data and information which is typically stored on their workstations, corporate file servers, and other external sources such as the World Wide Web (WWW). An increasing population of mobile users is demanding the same when only limited network bandwidth is available, or even when network access is not available. Moreover, given the growing popularity of portables and personal digital assistants (PDA), mobile users are requiring access to important data regardless of the form-factor, rendering capabilities and computing power of the mobile device they are using. We see three related *challenges* imposed by mobility:

1. *Any-time, any-where access* to user data, regardless of whether the user is disconnected, weakly connected via high-latency, low-bandwidth networks, or temporarily completely disconnected.
2. *Device-independent access* to data to allow users to switch among different devices, even while mobile, and have access to the same set of files.
3. *Application-independent access* to data to allow users to modify portions of documents and files belonging to classes of related applications (e.g., the ability to modify parts of a document irrespective of the Word processing application that was used to create the file).

In [8], the authors have described a three-tier architecture as a basis for overcoming the aforementioned challenges. The heart of the architecture is a *mobile environment manager* to support the automatic hoarding of data from multiple, heterogeneous sources into a variety of different mobile devices. The mobile environment manager, which resides on a central

(master) repository server as well as on each of the computing devices that are used, eliminates the manual and tedious synchronization between the devices and the central repository on the one hand, and between multiple (mobile) devices on the other hand. The three-tier architecture and supporting algorithms provides any-time, any-where access to data, irrespective of which device is used (challenges 1 and 2) and enables the automation of synchronization tasks in both connected mode (following disconnection) and weakly connected mode.

In this paper, we describe our approach to overcoming the *third challenge*, namely to provide format-independent access to data in mobile computing environments. In particular, we describe algorithms for updating copies of content-rich text documents that reside on different devices in different formats based on the capabilities of the device. Our approach is called *Format-Independent Change Detection and Propagation (FCDP)*. FCDP is capable of computing the changes that have been made to a document on one device (e.g., a Microsoft Word document on the user's desktop) and applying them to a copy with minimal formatting instructions and structure on a different device (e.g., a PDA incapable of directly manipulating a full MS Word document). Selectively removing metadata or content allows users of mobile devices to not only read a document generated on another platform and with a more powerful application, but to modify it and have those modifications reflected in the originating document. The application of changes across applications is *automatic*, requiring no user initiation like current desktop-to-PDA conversion products. Our creation of cross-platform translations is also *transparent* and does not require the multi-step, user-directed translations found in commercial products.

To illustrate how FCDP works, let us explore a simple scenario. A busy executive is composing a document in his[1] office using a full-fledged word processor. To enable ubiquitous access to the file and automatic updates to a master copy for future changes, he adds the document to his "working file set" on the central repository server using the environment manager portion on his PC. Prior to leaving for a trip, he synchronizes his FCDP-enabled PDA with the repository server. The server-portion of FCDP automatically removes pictures, graphs, formatting, and other information that is not usable by the PDA's editing application (e.g., a simple text editor)—leaving only the ASCII text. This reduction speeds up transmission to the device, minimizes storage requirements on the PDA, yet retains enough information to proofread and edit the contents of the paper while traveling.

During his travel, the executive updates the document. When connecting to the company server in the evening, the FCDP synchronization client on his PDA transmits the detected changes (using an intermediate format) to the central repository for synchronization with the master copy. Since only the changes are transferred, the necessary bandwidth demands are low. Once the changes reach the central repository server, they are re-integrated into the original, content-rich document, which is in the format used by the word processing application. To the best of our knowledge, *this ability to apply the changes, which have been made to one document, to an unedited copy but of different format and content-richness is not currently possible.*

2. Approach

As previously stated, our approach to change detection and propagation is part of a larger research effort to develop an infrastructure and methodologies for automatic hoarding and synchronization of data across devices and nodes to enable ubiquitous computing. The hoarding and synchronization architecture is shown in Figure 1. The central component of this

[1] The terms he, his, etc. are used throughout this paper to refer to an individual which may be male or female.

architecture is the Mobile Environment Manager (MEM), which has two primary components: M-MEM and F-MEM (Mobile and Fixed respectively). It provides the smart filtering algorithms and communication links between clients and the central repository and synchronization server. Underneath F-MEM is the meta-data database (Oracle Relational Database Management System with XML support) that manages the information about the working sets for each user including the default rules for device capabilities and the rules and filters for converting data into various formats to support transmittal between different devices. The meta-data database also contains the links to the actual data files that make up the working set, which are stored in a CODA file system [18].

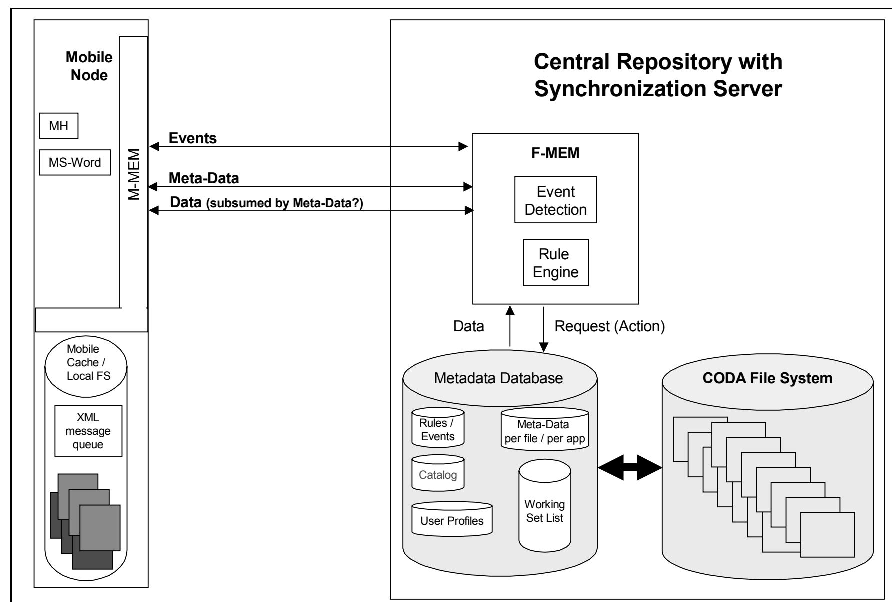

Figure 1: Conceptual overview of our hoarding and synchronization architecture.

A client's M-MEM modules maintain contact with the servers or, upon reconnection to a network, establish connection to the synchronization server. Along with the initial communication setup, M-MEM identifies its O/S and device type. M-MEM also transfers the information it has captured during connected and disconnected states, including interesting file activities that need to be reflected in the user's working set, changes in consistency desires, etc. to the F-MEM. Files not accessed within a certain amount of time are automatically removed from the working set. For active files, F-MEM receives either entire files or differential updates from the clients. Likewise, when a client first requests a file (either actively or through an access miss on the client), F-MEM sends the entire file.

When M-MEM reports a file access miss on the client, F-MEM must transfer the requested file to the client. Our architecture uses an optimistic replication scheme similar to the one used in the Coda file system [18]. Since the server stores all files under its control as XML encoded files, it must translate the files into a format understood by the requesting application. If the client is using an application/device incapable of utilizing the content-rich data, the server initiates a content reduction process (in addition to the translation process) in order to produce a format understood by the client.

For the following description of FCDP, we assume the mobile client is unable to render or edit content-rich documents. Furthermore, we assume that a low bandwidth, high-latency network connects the client to the data repository. The first step is to get the document from the repository to the mobile device. If necessary, F-MEM starts the conversion of the corresponding document into a XML-based format usable by the client. For complex, content-

rich documents, an F-MEM initiated converter may also omit objects that are either not useable by the client or which are too big for efficient transfer over the network. A graphical depiction of this process is shown in Figure 2.

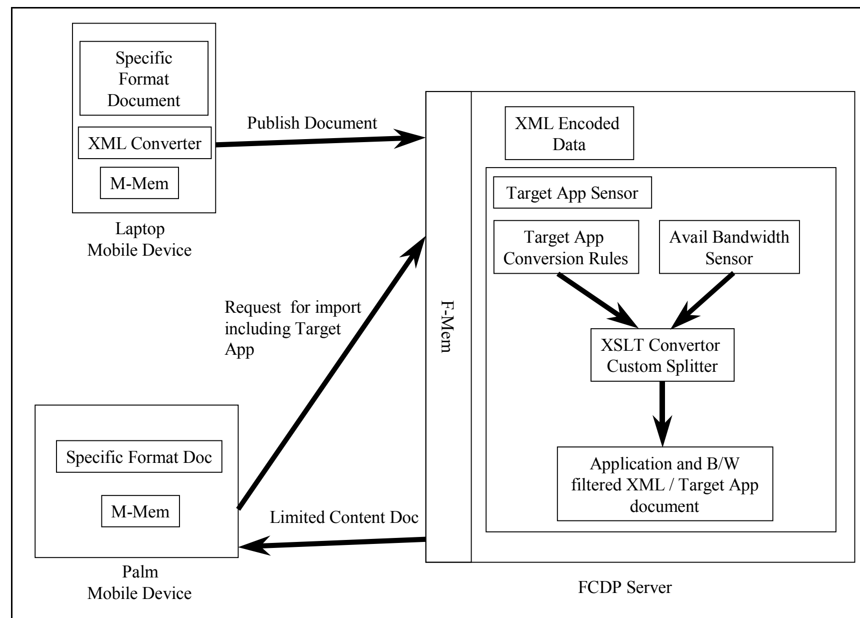

Figure 2: Conceptual overview of FCDP.

Of particular note is the use of XML to provide the canonical encoding of the documents in the warehouse. Discussion of converting proprietary formats to XML is beyond the scope of this paper. Using XML as the encoding format allows us to leverage the growing set of tools and automation techniques to transform XML into other formats. It also eases processing of the document in comparison to commercial proprietary formats.

On the client device (e.g., the Laptop or Palm shown on the right-hand side in Figure 2), an XML converter performs lossless, two-way conversion of the transmitted document into the format used by the client application and vice versa. After the file has been edited (e.g., upon file closing), it is re-converted into the intermediate XML document and compared to the downloaded version in order to identify the changes. This is done by the client-side FCDP which computes the minimum cost edit script [3, 5] that is transmitted to the FCDP server immediately (when in connected or weakly connected mode) or else upon reconnection. Note that re-conversion and change detection can also be a user directed event.

On the server, FCDP applies the edit script to the XML version of the original document stored in its repository. The outcome of this is a document with the original content plus all the changes that were made to its filtered counterpart document on the mobile device. If necessary, the server can transmit the edit script to other mobile devices to bring those copies into a consistent state. The metadata repository also stores the edit script to provide versioning control if previous versions of the document need recovering. The following sections describe our implementation of the client and server-side FCDP algorithms in a prototype system as well as the results of an initial set of experiments validating our approach.

3. Implementation of FCDP

3.1. Overview of FCDP

For this discussion, we refer to the original, unedited version of the document as the v_0

version; v_1 refers to the version of the document after a user has modified it. Table 1 shows a summary of the various document incarnations used in FCDP.

Table 1: Document legend in FCDP

v_0	Rich Content, XML Doc	$v_1(-)$	Changed version of $v_0(-)$
$v_0(-)$	Text only version of v_0	$v_1(-)$’	v_1^- with XML structure imposed
		v_1	v_0 with modifications from $v_1(-)$

The first version of our FDCP algorithms focuses on the important and large class of word processing documents. Recall the scenario in Section 1 in which the executive wants to use his PDA to proofread and edit files created on his desktop computer. We graphically depict the use case in Figure 3 below.

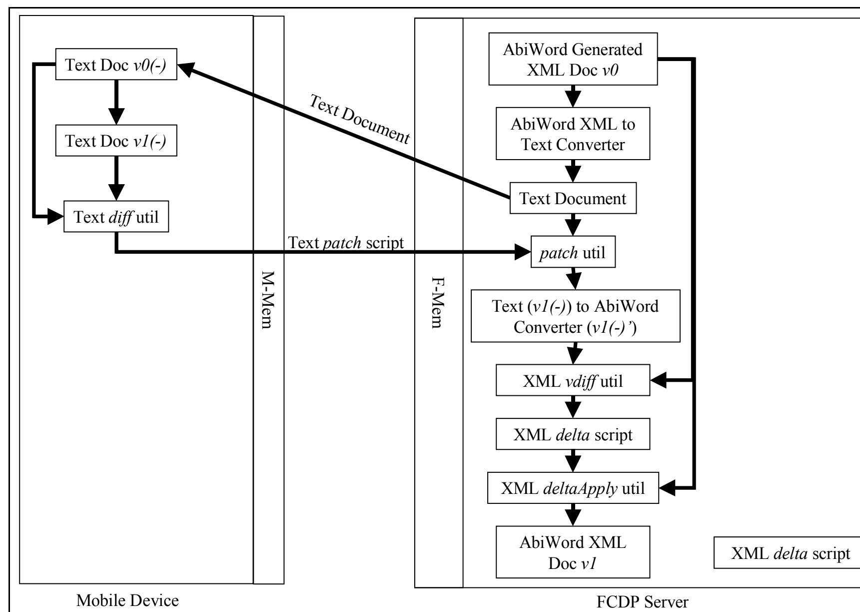

Figure 3: FCDP overview with a text-only mobile client.

We have developed tools to convert the canonical XML formatted document (v_0) on the repository server into a text-only document ($v_0(-)$). F-MEM transmits the document to the PDA. The executive then edits the document and creates version ($v_1(-)$). The PDA’s M-MEM executes a text-based change detection program (GNU *diff()* [7]) between $v_0(-)$ and $v_1(-)$. M-MEM transmits the edit script to the repository server where FCDP integrates the changes into its copy of $v_0(-)$ to create $v_1(-)$. It must convert the updated text-only document into a form ($v_1(-)$’)usable by our XML change detection engine called *vdiff* (verbose diff) engine. The server proceeds to detect the changes between $v_1(-)$’ and v_0 to create an XML based edit script. Finally, the server applies the edit script to v_0 to create v_1. We use two iterations of difference detection (GNU *diff()* and *vdiff()*) only when the mobile device is unable to support the processing demands of the vdiff-engine. In Figure 4, we depict a less extreme content-transformation example. We expect the client to perform a single change detection exclusively using the vdiff-engine we describe here.

To allow change detection across different text/word processing applications we chose to rely on XML to provide a universally accessible format. We found it easier to transform and reduce XML documents than transforming proprietary formats. The conversion of v_0 documents into $v_0(-)$ documents formats requires we track the transformations performed on the structure and contents of v_0. Lossless conversion requires no such tracking but it is

essential to allow cross-format change-detection.

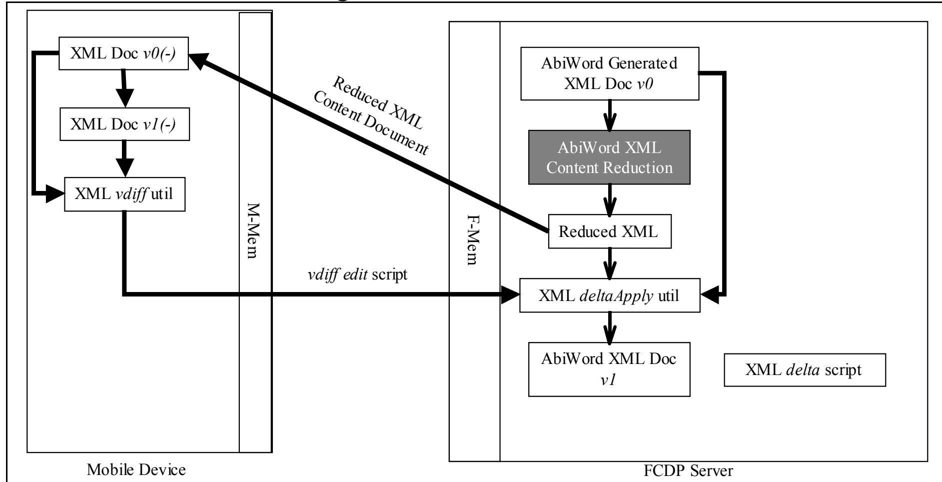

Figure 4: FCDP overview with a generic mobile client.

3.2. The Symmetric Difference Map File

Word processors create documents with that contain a lot of metadata such as style and formatting information, page definitions, etc. We must identify the metadata, document structures, and document contents that, due to rendering limitations or bandwidth restrictions, mobile devices will not have access to. Once identified and removed, this missing data will not interfere with the change detection and propagation. Without this information, a "roundtrip" between a transformed and edited document back to the originating document would not be possible. To track this information we rely on externally stored data. We have created a vdiff schema, shown in Figure 5 that defines an intersection or *symmetric difference map file* between the document structure of an XML file and its corresponding transformation.

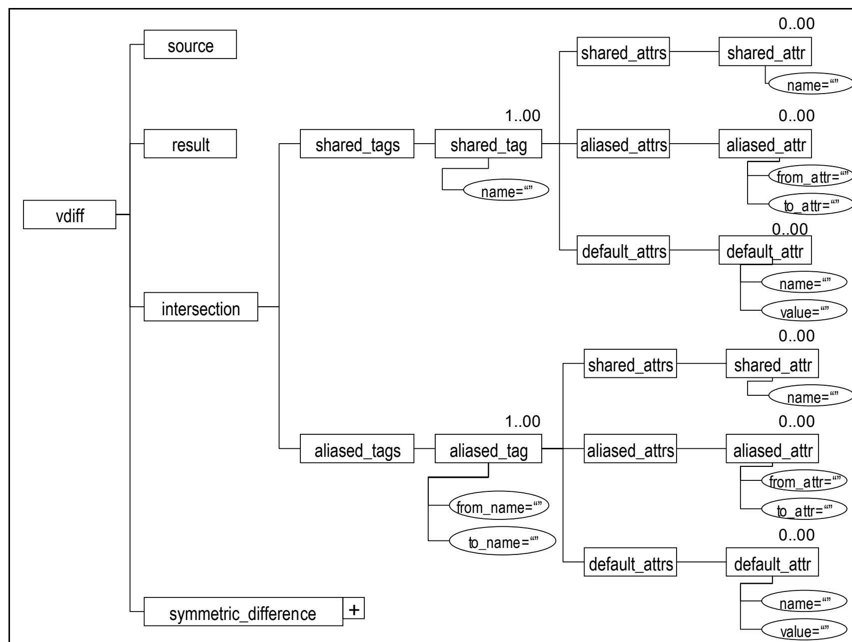

Figure 5: Diagram of vdiff schema.

The symmetric difference map file contains the meta-data that reports the name of the originating application and the format of the targeted conversion. The schema then requires a tag-by-tag enumeration of shared and aliased tags or a tag-by-tag enumeration of unshared tags and unshared attributes. The ability to list only the intersection of identical tags between XML formats prevents having to enumerate every possible tag name in the domain of the v_0 document. The alternative is to list only the unshared tags (the symmetric difference) between two XML documents. Figure 5 shows the vdiff schema with the intersection element expanded.

3.3. The vdiff() Algorithm

The portion of FCDP responsible for detecting changes to the canonical XML-based representations, is *vdiff()*. Our implementation of *vdiff()* is based on *XyDiff()* [6] created by the VERSO team at INRIA.

As in XyDiff(), we use XIDs (eXternal IDs) to provide permanent identifiers for every node in the v_0 document. The pseudo-code below provides the top-most level of the *vdiff()* algorithm and assumes three arguments are provided by the invocation, the name of the v_0 document, the name of the $v_1(-)'$ (referred to as "v1minus" for the remainder of this section) document, and the name of the map file describing structural similarities or differences.

```
1    v0DOM = ParseAndLabel(v0document,isSource);
2    v1DOM = ParseAndLabel(v1minusDocument,isNotSource);
3    StructuralMapInfo = ParseMapFile(mapFile);
4    BuildSubTreeLookupTable(v0DOM);
5    FindAndUseIDAttrs(v0DOM);
6    TopDownMatchHeaviestTrees (v1minusDOM);
7    PeepHoleOptimization (v0DOM); //force matches if reasonably safe
8    MarkOldTree(v0DOM, StructuralMapInfo);
9    MarkNewTree(v1minusDOM, StructuralMapInfo);
10   AdjustForUnSharedChildren (v0DOM, v1minusDOM, StructuralMapInfo);
11   BuildLeastCostEditScriptForWeakMoves (v0DOM,v1minusDOM);
12   DetectUpdatedNodes(v1minusDOM, v0DOM);
13   ConductAttributeOperations(v0DOM, v1minusDOM,StructuralMapInfo);
14   WriteXIDmapFile (v1minusDOM);
15   WriteDiffInfoToFile ();
```

We briefly explain the logic embedded in this code. In lines 1 and 2, the `ParseAndLabel` method uses the Xerces [2] XML parser to parse and validate the input XML files. Immediately after parsing, the method traverses the in-memory DOM and builds a mapping between each node and its XID. The `ParseMapFile` method in line 3 processes the contents of the map file and instantiates the appropriate classes and data structures. The data structures are primarily STL maps that ensure *O(1)* lookup. This ensures little performance penalty when seeing if an element tag is shared between the two documents.

The `BuildSubTreeLookupTable` method of line 4 traverses the *v0DOM-tree* and builds an average case *O(1)* lookup table of sub-trees. The key for each sub-tree is a hash value created from the content of the sub-tree's root plus the cumulative hash values of its children. Line 5's `FindAndUseIDAttrs` determines if any elements in the *v0DOM-tree* and *v1minusDOM-tree* have ID attributes and attempts to find the appropriate matched node. The identical value in the two ID attributes is *prima fascia* evidence of a match.

In line 6, the `TopDownMatchHeaviestTrees` method uses the sub-tree lookup table built in line 4 to search for matches starting at the top of the *v1minusDOM-tree*. An obvious and mandatory match of the root nodes happens first, then a breadth-first search. If a match

occurs at a non-leaf node, the method recursively assigns all the descendants of the matched nodes to their respective peers.

Line 7 contains the critical `PeepHoleOptimization` method. This is an attempt to increase the number of matched nodes without creating false matches. For each matched node in the *v0DOM-tree* and its matched node in the *v1minusDOM-tree*, it builds a list of unique unmatched children. If there is only one instance of each tag name in each child list, match the child nodes. If there is more than one instance of the tag name, there is insufficient data to force a match.

`The MarkOldTree` and `MarkNewTree` methods in lines 8 and 9 are straightforward. The `MarkOldTree` method traverses the *v0* tree and marks every unmatched and shared-tag node as deleted. It also marks nodes as strong moved if they and their matched node do not have parents that are themselves matched to each other. The method `MarkNewTree` marks unmatched and shared of *v1minusDOM* as inserted.

Line 10 lists a critical piece of FDCP. This method, `AdjustForUnSharedChildren`, ensures proper ordering of inserted and moved children. Without compensating for the offsets caused by unshared children, a node's child list will not be in a correct sequence. For example, an inserted child may appear to be the *i*th child of a *v1minus* node. When incorporated back into the original document however, it should rightly be in the *j*th position. If left with an incorrect insertion position, the diff script will insert the node at the *i*th position. This incorrect positioning will cause errors as minor as wrongly ordered paragraph/picture sequences and as major as violating the DTD or Schema of the source document.

The `BuildLeastCostEditScriptForWeakMoves` method of line 11 is a straightforward longest common sub-sequence problem. The task is to determine the least expensive means by which each node can turn its old child sequence into its new child sequence. The cost for inserts and deletes are proportional to the weight of the sub-tree each child represents. Line 12, `DetectUpdatedNodes`, is also uncomplicated in implementation. If a node and its matched node have only singular text node children, match the text nodes. Consider the text nodes updated and assign new XIDs to them. The method of line 13, `ConductAttributeOperations`, looks at every matched node and determines if its attributes represent inserted, deleted, or updated values. We again use the `StructuralMapInfo` and its *O(1)* lookups to minimize the expense of determining if the absence of attributes in a *v1minusDOM-tree* node are meaningful. In the domain of our test sets, this function also infers attributes for inserted nodes. This inference is critical to proper rendering of paragraphs in AbiWord [20].

3.4. Improvements to the vdiff() Algorithm

Based on the performance of our initial implementation of the *vdiff()* algorithm, we incorporated several improvements to be able to handle more complex document structures with lower error rates. We refer to the new version of *vdiff()* as *vdiff2()* and its algorithm is outlined below. The least cost edit script referred to in line is based on the algorithm developed by Myers [16]:

```
1 Match the nodes in the v0 and v1minus documents using the least cost
  edit script match algorithm;
2 Adjust the structure of the v1minus document to make it isomorphic to
  the v0 document;
3 Optimize matches;
4 Construct delta script;
```

The primary problem in the node matching phase in line 1 is that a sub-tree of text nodes (which store the document content) in the v0 document will represented as a single node in the v1minus document since all document structure has been lost. Hence, a simple matching of text nodes in the two documents would lead to unsatisfactory results. In order to solve this problem, a sub-tree match phase has been introduced as one of the phases of the match procedure. Another problem encountered in this phase is that text nodes also contain information other that the text content of the document in the v0 document. As an example of this consider inline images in Abiword documents. The image data is encoded into base-64 characters and stored in an ordinary text node. The exclusion map mentioned above enables the algorithm to exclude such information from the match phase since each match has a run time linear in the size of the input nodes and image data tends to be very large.

The primary steps involved in the matching procedure are:

1. Construct a list of text nodes from the v0 document by excluding any sub-trees specified in the exclusion map.
2. Construct a list of text node sub-trees from the v0 document using the list constructed in step 1. All sub-tree members are removed from the first list as a result.
3. Construct a list of text nodes from the v1minus document
4. For each text node in the v1minus list, use the least cost edit script match algorithm to pair it with a text node in the v0 document list. During this process, all match values are retained in a hash table for fast lookup. When a match is found, i.e., the script length is below a threshold determined by the input lengths, a check is made to determine that the v0 document node was not already matched. If so, the better match is selected.
5. For each node in the v0 document sub-tree list, if the size of the subtree is below a threshold, compute all permutations of the sub-tree nodes and store them in a list. The nodes in this list are then matched against all nodes in the v1minus list to find the best match. If the subtree exceeds the threshold size, we do not compute any permutations and compare the natural node order of the v0 document to the nodes in the v1minus list. Hence the overall runtime of *vdiff2()* is closer to *O(NM)*, where N and M are the number of text nodes in both trees. Once a best match is determined, a trace through the edit graph of the two strings is determined. This trace combined with our knowledge of the end points of the v0 sub-tree nodes, allows us to determine the matching sub-strings in the v1minus node. Thus we have performed an approximate sub-string match that can associate sub-strings with their edited versions within a certain threshold.

Line 2 of the *vdiff2()* algorithm is responsible for adjusting the structure of $v_1(-)$’ to make it isomorphic to that of v_0. The steps involved in this phase are:

1. Apply the sub-string matches by replacing the node in the v1minus document by its corresponding sub-strings.
2. Adjust the document for unshared ancestors by inserting all unshared ancestors of a matched node in the v0 document into the v1 document. However, the condition where unshared ancestors of sibling nodes in the v0 document are being inserted would cause two separate sub-trees to be generated. To handle this, the sibling sub-tree of the candidate node in the v1minus document is checked for matches against the ancestors being inserted. If matches exist and the v0 document ancestors are already matched to these nodes, the existing v1minus nodes are used.

3. Adjust the document for unshared nodes by traversing the trees and importing all unshared nodes of matching parents.

Optimization of matches in line 3 and the construction of the delta script in line 4 are unchanged and performed in the same manner as in the original *vdiff()* algorithm.

4. Experimental Results

4.1. Experimental Setup and Description of Data Sets

The testing platform consisted of an AMD Duron 650Mhz processor, 128 Mb RAM and an IBM-DTLA-307030 hard drive running Mandrake Linux 8.2 (kernel 2.96) with gcc version 2.96 and Xerces-C++ 1.4. Test data for the experiments consists of ten AbiWord generated term papers, forms, memos, letters and pages from documents such as E-books and brochures. We conducted insert, delete, move and update operations on 0%-50% (at 10% increments) of the paragraphs in the original document and 10%-40% (at 10% increments) of the content of each paragraph. Hence, each test case resulted in 96 modified documents. In the following test cases, *vdiff2()* is used instead of the original *vdiff()* algorithm.

We lack empirical evidence to demonstrate that this sequence is reflective of real world editing patterns, but intent to establish empirical evidence in future work. The test documents contained no graphics or other embedded objects, just formatted and styled text. We deliberately kept the test set small to better control the fluctuation of document structure within our collection of documents. The primary goal of the experiments was to prove the viability of cross-format change detection. A secondary goal consisted of proving the assumption that content reduction can result in transmission savings.

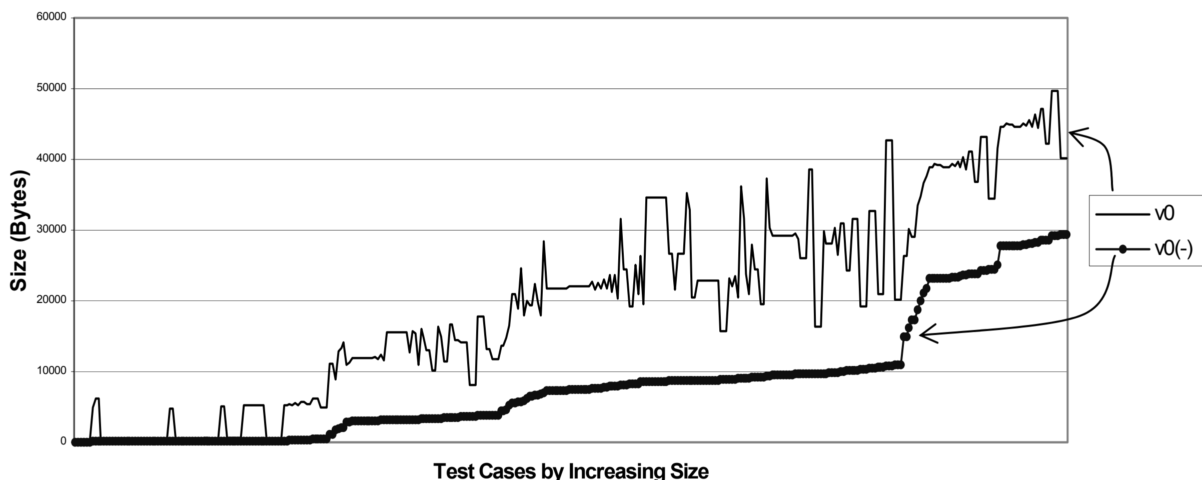

Figure 6: Bytes to transmit v_0 vs. $v_0(-)$ documents (generated by AbiWord)

4.2. Bandwidth Savings

Transmitting text-only versions of documents to PDAs can result in large reductions in the amount of data transmitted. Although the AbiWord test set was relatively small (960 test cases derived from 10 source documents), results from the Puppeteer project [11] also substantiate the savings achievable by content reduction as shown in Figure 6.

Another indicator of how FCDP reduces the amount of data that mobile devices need to transmit is shown in Figure 7. If the mobile device has to transmit the entire $v_1(-)$ document, it

will have to transmit far more data than just the edit script generated by *GNU diff()*.

Figure 7: Bandwidth savings when using GNU diff versus value shipping of entire file

The cumulative savings generated by *vdiff2()* are shown in Figure 8. The *vdiff2()* edit script sizes are only 7.53% of the total size for XyDiff [6]over all 960 test cases, which is a significant savings. Of course the more significant factor is that XyDiff generated files lose almost all document formatting.

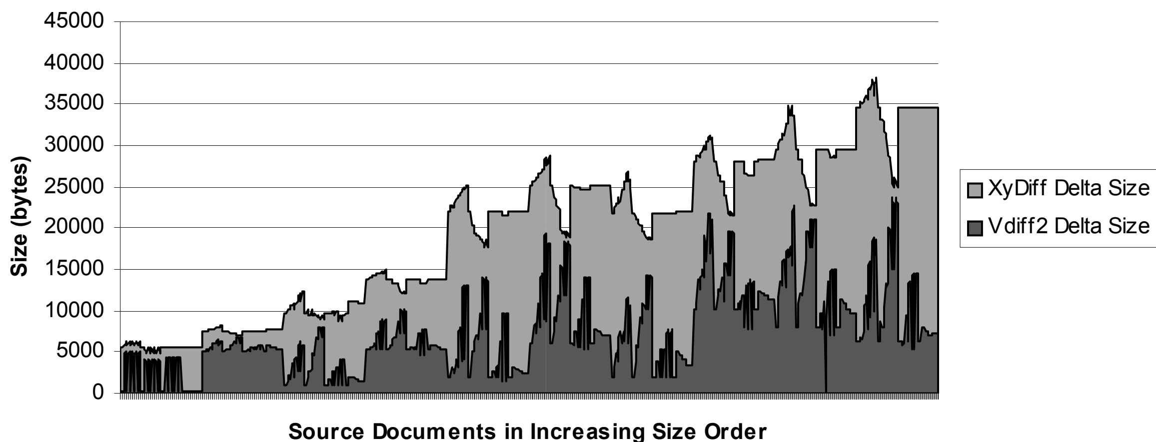

Figure 8: Comparison of *vdiff2()* edit script sizes with those generated by XyDiff.

4.3. Error Rates

In order to compute the error rates, we applied the edit operations on the source document to generate edited documents (we denote these as "reference modified documents"). These documents were then converted to text and provided as an input to *vdiff2()* along with the original source documents. The resulting documents after applying the generated deltas were compared to the modified reference documents for quantification of error. Figure 9 shows a plot of the number of missing nodes of various types using the reference documents as the expected result. As can be seen, *vdiff2()* error rates are significantly lower than those for XyDiff, having only 5.7% of the total number of missing nodes. Please note that negative node counts indicate those errors where *vdiff2()* created "new" nodes that are not part of the

original document.

It is important to point out that these performance benefits are achieved at the cost of runtime performance: whereas the original *vdiff()* algorithm is almost 5% faster than XyDiff, *vdiff2()* is now roughly five times slower than XyDiff. However, we believe that the slower performance of vdiff2 does not impact its usefulness, since accuracy is much more important.

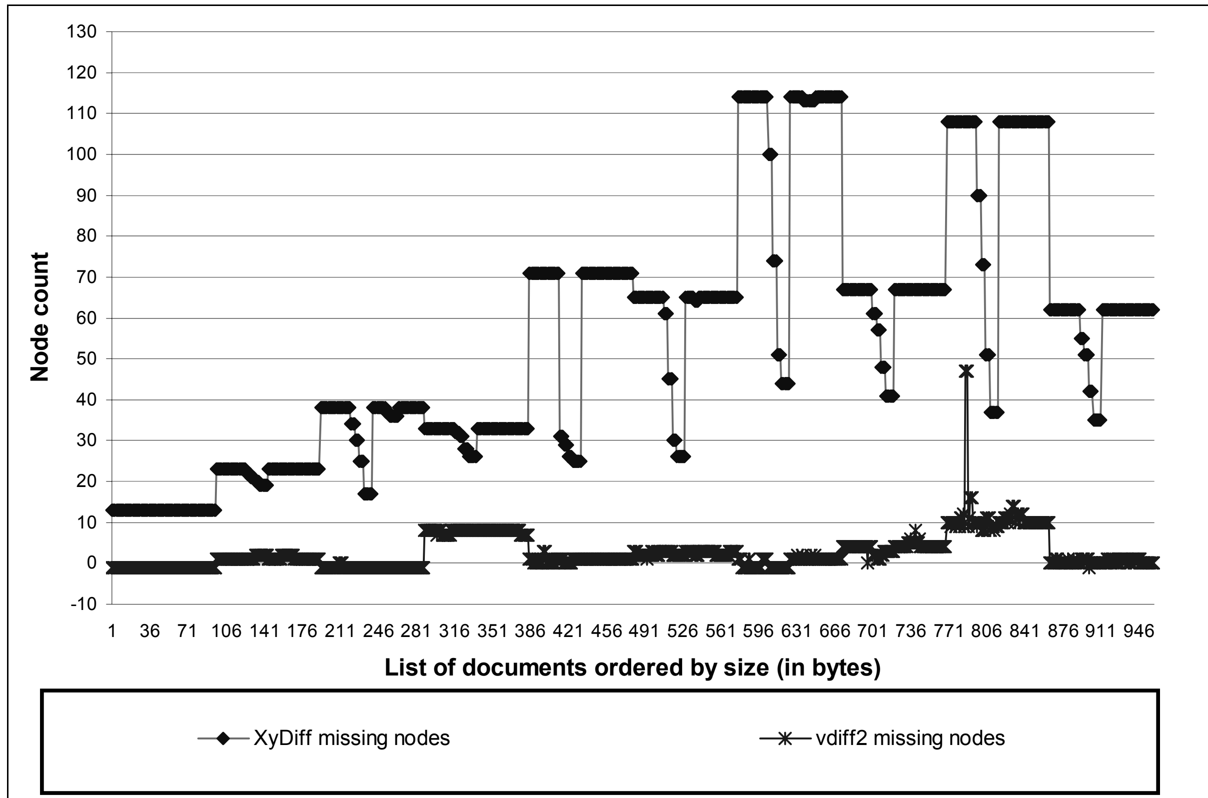

Figure 9: Missing node counts for *vdiff2()* and XyDiff against the reference modified documents.

5. Related Research

The research described in this paper falls into two categories: XML difference algorithms and bandwidth adaptation.

5.1. XML difference algorithms

Our approach to change detection and propagation in XML documents is based on tool developed by the VERSO Team at INRIA for their Xyleme Project [6, 14]. This tool has the original name of XyDiff and now has the name verbose-diff (vdiff): accounting for documents' differing levels of structural verbosity and content. XyDiff expanded on the capabilities of Sun's [21], IBM's [9] tools by incorporating the ability to capture move and update semantics. Like [3, 4], XyDiff is one of the few XML tools to utilize the move semantic for *diff* scripts. This takes advantage of the hierarchical nature of XML and allows movements of entire sub-tree to new locations with a single entry in a *diff* script.

The original XyDiff algorithm [14] utilizes external identifiers (Xyleme IDs) to permanently identify each node in the *v0* XML document. These identifiers correlate to a post-order traversal of the DOM tree created by parsing the XML document. The remainder of the XyDiff algorithm shares 85% commonality with the previous version of *vdiff()*. The

algorithm shown in Section 3, minus all details associated with the `StructuralMapInfo`, applies for XyDiff.

5.2. Bandwidth Adaptation Through Content Reduction, Editing, and Propagation

The research described here is part of an overall effort to build the infrastructure to support device independent mobile computing. Other device-independent computing efforts are under way at Stanford [1], DARPA [19], IBM [10], and at Texas A&M [13].

Our effort to allow the executive to propagate text only changes into a complex format (like OpenOffice's StarWriter, AbiWord, and MS Word) expands on work done in the area of change detection. GNU's `diff()` utility, for example, compares flat text files using the algorithm described in [16]. There are also numerous front ends to `diff()` that present the results in various formats. The difficulty with `diff()` and similar utilities is they use line breaks as record delimiters: however, line breaks often do not exist nor have real meaning in binary data files. In addition, `diff`-based utilities do not recognize hierarchically structured data and are unable to discover movement of data from one location to another. Due to these shortcomings, we are adapting existing research on finding minimum cost edit distances of structured data (see, for example, [5, 23]). This line of research transforms the data into a tree structure. An edit script can change tree A into tree B with a sequence of inserts, deletes and moves of A's nodes so that it looks like B in both shape and content. A minimum cost edit script is one that is least expensive with respect to I/O operations, time to completion, and other desired metrics [3].

XML specific diff algorithms include laDiff [4], IBM's *XML Diff Merge Tool* and *XML treeDiff* [9], and Sun's DiffMk [21]. Their shortcoming in this proposal is they all require a 100% overlap in tag and attribute set domains for both XML documents. They each mandate that absence of data in the modified file is a delete operation imposed on the original file. We do not impose such a restriction.

Research on common/intermediate representations using XML has been ongoing at Stanford University with the Lore Project [15]. In addition, the open source community and Linux developers have created numerous XML converters for popular word processing formats [22] that we are leveraging. Our goal is to make any and all translations invisible to the user, and automatically apply detected changes between file versions.

The Puppeteer [12] project at Rice University is the closest to our vision of systems that support ubiquitous computing without adapting the user's applications. They use public Microsoft APIs to parse original MS Office documents into OLE-based DOMs. The DOMs nodes are individual structures (e.g., page, slide, worksheet) from within the MS document. They offer the ability to transmit content in low- and high-fidelity modes, and switch between the two. The mobile device's MS Office applications then manipulate the data as usual. Puppeteer also is beginning the process of allowing edits to the low-fidelity components on a mobile device and integrating those changes into the high-fidelity version. Puppeteer does not yet have the cross-application design goal that we are attempting to implement.

Another line of research in bandwidth adaptation is the Odyssey project [17]. Odyssey also adapts application data to the current state of the network connection. Unlike Puppeteer, Odyssey requires applications be customized to support its implementation scheme. This is in contrast to Puppeteer, which uses public APIs of applications to manipulate that application's data files. It is also in contrast to our own system, which utilizes a common format for supported applications: XML.

6. Conclusion and Status

In this paper we have presented change detection and propagation methods to synchronize documents which are stored on various computing devices. Our approach, called *Format-Independent Change Detection and Propagation (FCDP),* is capable of computing the changes that have been made to a document on one device and applying the net effect to an unedited copy, which uses a different format and representation. This allows users of mobile devices to modify documents, which have been generated by more powerful applications on different platforms, and to have the modifications reflected in the originating documents.

Our work is contributing to the state-of-the-art in the following important ways. We presented algorithms and techniques for converting XML documents into a client-usable format (in this case an ASCII text editor). We have developed algorithms and techniques to convert client-usable formats into minimalist XML documents based on the originating documents' XML DTDs/Schemas. We have also developed tools to track where and how the minimalist XML representations intersect with the XML tag&attribute set of the originating document. We developed a technique to infer, whenever appropriate, attributes for new XML nodes when the modifying application cannot generate those attributes. Finally, we presented a heuristic to integrate shared and unshared nodes across XML documents to ensure useful ordering of document content.

Despite the promising results of our experiments, our system is still work in progress. We are currently refining the approach to provide more capabilities and improve performance. To improve performance we want to develop customizable peephole optimization routines that rely on specific knowledge of the XML data structure. We also will use placeholders in the bandwidth-adapted files to mark the location and existence of stripped elements. That step will help us improve performance (both in time, and size of the edit script). Placeholders will allow movement, deletion, resizing, and other manipulation of the placeholder (though not its actual contents). We are also studying the runtime performance of the algorithm and are trying to develop more efficient alternatives for the most expensive phases.

References

[1] T. Ahmad, M. Clary, O. Densmore, S. Gadol, A. Keller, and R. Pang, "The DIANA Approach to Mobile Computing," presented at MOBIDATA: Workshop on Mobile and Wireless Information Systems, Rutgers, NJ, 1994.

[2] Apache Software Foundation, "Xerces C++ Parser."

[3] S. Chawathe, "Comparing hierarchical data in external memory," presented at Twenty-fifth International Conference on Very Large Data Bases, Philadelphia, PA, 1999.

[4] S. S. Chawathe and H. Garcia-Molina, "Meaningful Change Detection in Structured Data," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, pp. 26-37, 1997.

[5] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 25, pp. 493--504, 1996.

[6] G. Cobéna, S.Abiteboul, and A. Marian, "Detecting Changes in XML Documents," presented at International Confernce on Data Engineering, San Jose, CA, USA, 2002.

[7] Free Software Foundation, "Diffutils," Web Site, http://www.gnu.org/software/diffutils/diffutils.html.

[8] A. Helal, J. Hammer, A. Khushraj, and J. Zhang, "A Three-tier Architecture for

Ubiquitous Data Access," presented at First ACS/IEEE International Conference on Computer Systems and Applications, Beirut, Lebanon, 2001.

[9] IBM Corp., IBM AlphaWorks Emerging Technologies - XML Utilities, Web Site, www.alphaworks.ibm.com.

[10] IBM Corp., Pervasive Computing Website, Web Site, http://www-3.ibm.com/pvc/pervasive.shtml.

[11] E. d. Lara, D. Wallach, and W. Zwaenepoel, "Opportunities for Bandwidth Adaptation in Microsoft Office Documents," presented at 4th USENIX Windows Systems Symposium, Seattle, Washington, 2000.

[12] E. d. Lara, D. Wallach, and W. Zwaenepoel, "Position Summary: Architectures for Adaptation Systems," presented at Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany., 2001.

[13] D. Li, "Sharing Single User Editors by Intelligent Collaboration Transparency," presented at Third Annual Collaborative Editing Workshop, ACM Group, Boulder, CO, 2001.

[14] A. Marian, S. Abiteboul, and L. Mignet, "Change-Centric Management of Versions in an XML Warehouse.," presented at 27th International Conference of Very Large Databases, Rome, Italy, 2001.

[15] J. McHugh, S. Abiteboul, R. Goldman, D. Quass, and J. Widom, "Lore: A Database Management System for Semistructured Data," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 26, pp. 54-66, 1997.

[16] E. Myers, "An O(ND) difference algorithm and its variations," *Algorithmica*, vol. 1, pp. 251-266, 1986.

[17] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker, "Agile application-aware adaptation for mobility," *Operating Systems Review (ACM)*, vol. 51, pp. 276-287, 1997.

[18] M. Satyanarayanan, J. Kistler, P. Kumar, M. E. Okasaki, E. H. Seigel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," *IEEE Transactions on Computers*, vol. 39, pp. 447-459, 1990.

[19] J. Scholtz, "Ubiquitous computing in the military environment," Defense Advanced Research Projects Agency (DARPA), 3701 North Fairfax Drive, Arlington, VA 22203, Technical Report February 15, 2001.

[20] SourceGear Corporation, "AbiWord: Word Processing for Everyone."

[21] N. Walsh, "Making all the difference," Sun Microsystems, XML Technology Center, Menlo Park, CA, Technical Report February 2000.

[22] H. Watchorn and P. Daly, "Word And XML: Making The 'Twain Meet," presented at XML Europe 2001, Internationales Congress Centrum (ICC), Berlin, Germany, 2001.

[23] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems," *SIAM Journal of Computing*, vol. 18, pp. 1245-1262, 1989.