Docx2Go: Collaborative Editing of Fidelity Reduced Documents on Mobile Devices

Krishna P. N. Puttaswamy[*]
Microsoft Research Silicon Valley
Mountain View, CA 94043
krishnap@cs.ucsb.edu

Catherine C. Marshall
Microsoft Research Silicon Valley
Mountain View, CA 94043
cathymar@microsoft.com

Venugopalan Ramasubramanian
Microsoft Research Silicon Valley
Mountain View, CA 94043
rama@microsoft.com

Patrick Stuedi
Microsoft Research Silicon Valley
Mountain View, CA 94043
pstuedi@microsoft.com

Douglas B. Terry
Microsoft Research Silicon Valley
Mountain View, CA 94043
terry@microsoft.com

Ted Wobber
Microsoft Research Silicon Valley
Mountain View, CA 94043
wobber@microsoft.com

ABSTRACT

Docx2Go is a new framework to support editing of shared documents on mobile devices. Three high-level requirements influenced its design — namely, the need to adapt content, especially textual content, on the fly according to the quality of the network connection and the form factor of each device; support for concurrent, uncoordinated editing on different devices, whose effects will later be merged on all devices in a convergent and consistent manner without sacrificing the semantics of the edits; and a flexible replication architecture that accommodates both device-to-device and cloud-mediated synchronization. Docx2Go supports on-the-go editing for XML documents, such as documents in Microsoft Word and other commonly used formats. It combines the best practices from content adaptation systems, weakly consistent replication systems, and collaborative editing systems, while extending the state of the art in each of these fields. The implementation of Docx2Go has been evaluated based on a workload drawn from Wikipedia.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*

General Terms

Algorithms, Design, Experimentation, Performance

1. INTRODUCTION

Sophisticated mobile devices have revolutionized the kinds of work people can do away from their offices; smart phones, netbooks, and niche devices such as e-book readers allow people to use workplace applications opportunistically in a range of settings. Mobile broadband networks mean that users need not plan ahead to work effectively in locations away from their offices; by design, they are able to access their documents from wherever they are. Cafes have become libraries and airplanes have become cubicles that move at 500 miles per hour. With our documents in the cloud, collaborative work appears seamless.

[*]Krishna is a Ph.D. candidate at the University of California at Santa Barbara. He was an intern at Microsoft Research Silicon Valley during this work.

But is it? Often real-world situations constrain normal collaborative activity: connectivity may be intermittent, for example, or bandwidth imposes fixed costs. Transmission may be interrupted, leaving a frustrated user with a partial copy of a long document. Some devices are more capable than others–a half-page figure that enhances the final version of an article may be unnecessary or inappropriate when the penultimate version is reviewed from an iPhone-sized screen. Collaborators may be co-located away from their offices, ready to exchange documents, but unable to reach a central server to do so. Multiple authors may make uncoordinated changes to their local copies of a document without consulting one another. In other words, there are invariably small snags in the seamless fabric of many collaborative activities.

In this paper, we introduce an application framework that addresses the requirements of one such collaborative activity, 'editing on the go.' We aim to support the kind of editing collaborators might perform using a broad range of devices as they move opportunistically between reading, reviewing, and writing or between data gathering, aggregating, and analysis.

To design such an application, we build on previous work in three somewhat independent areas of mobile computing and semi-synchronous collaboration: (1) methods for content adaptation; (2) platforms for weakly consistent replication; and (3) mechanisms that support collaborative editing. Content adaptation methods allow each participant and each device that is involved in a collaboration to store and manipulate the portions of a document that are relevant to the participant and that meet the constraints introduced by the situation at hand such as high-cost bandwidth or limited storage [3, 5, 6, 9]. Weakly consistent replication platforms enable data to be synchronized among loosely-organized networks in a way that guarantees that changes originating from any device can propagate throughout the network [2, 11, 13]. Finally, two decades' worth of collaborative editing research has resulted in reliable mechanisms for ordering edits to a shared document and for resolving concurrent changes [7, 12, 14, 18].

Docx2Go weaves these three technologies together in an application framework that addresses the challenges of collaboratively editing XML documents in a mobile environment. By supporting XML, Docx2Go can handle many modern formats, including formats that conform to OOXML and ODF standards and documents (including slides and spreadsheets) produced using the latest version of Microsoft Office, which runs on a variety of mobile devices. It also means that the application can be used to collaborate over Web-based documents, such as the interlinked pages that are the basis for wikis or other Web sites.

Each document in Docx2Go is represented as a collection of XML elements. This document structure enables us take advantage of–and harmonize–key elements of the three independent component technologies. Content adaptation, conflict detection and resolution, and synchronization may each be based on functional elements of an XML document such as paragraphs, headings, or figures.

For example, the user who is paying for her smart phone service by the number of kilobytes transmitted can specify that she only wants to see the set of paragraphs that constitute the introduction of a paper she is working on with her co-authors. The collaborators editing the first paragraph of the introduction and last paragraph of the conclusion will not incur the overhead of a costly merge, since conflicts do not arise between different uniquely identified XML elements. Devices can synchronize updated elements directly with other devices over a peer-to-peer network, keeping synchronization overhead low (only updated elements are synchronized) and flexible (a centralized service need not be contacted). Using a fine-grained representation based on a document's XML structure allows the technologies to be woven together in an effective way.

Similarly, biodiversity field data that is gathered by experts and citizen-scientists and stored in Excel spreadsheets can be selectively exchanged, updated, cleaned, and consolidated without the need to contact a central server. Small mobile devices may be used in the field, while more powerful computing capabilities may be brought to bear on large, aggregated datasets. It is easy to envision many different scenarios that take advantage of XML documents used in situations away from the normal workplace.

In this paper, we demonstrate the efficiencies obtained by exploiting XML document formats in the Docx2Go implementation, and discuss the kinds of metadata that are necessary for maintaining document structure in this environment. We then evaluate aspects of system performance using a workload derived from the edit history of a set of Wikipedia pages over a specified period.

The contributions of this research are threefold. One, we identified the requirements of this important family of applications, editing on the go, and confirmed that emerging XML-based standards can be leveraged to meet these requirements. Two, we designed a mobile document editing framework, Docx2Go, that supports collaborative editing on incomplete documents and achieves eventual consistency by incorporating advancements in content adaptation, optimistic replication, and groupware systems. Three, we demonstrated the specific advantages of our approach through a prototype implementation of the proposed framework and an application for editing Wikipedia pages.

2. MOTIVATION, REQUIREMENTS AND CHALLENGES

We use a scholarly writing scenario (Figure 1) to highlight requirements for a mobile collaborative document editing application. We have chosen this scenario because of its familiarity; in practice, we expect Docx2Go to be useful in a variety of less familiar settings such as eScience, in which many datasets are assembled and manipulated in Excel, or in domains that make extensive use of other types of XML documents such as PowerPoint slide sets, wikis, or Web pages.

2.1 A paper writing scenario

Alice and Bob, two of Professor Smith's graduate students, are writing a journal submission together. Alice is the lead author. She has written most of its sections and will ensure that the paper takes a coherent shape before submission. Bob, a junior graduate student who helped Alice with experiments, is producing the sections on the experiments and their implications.

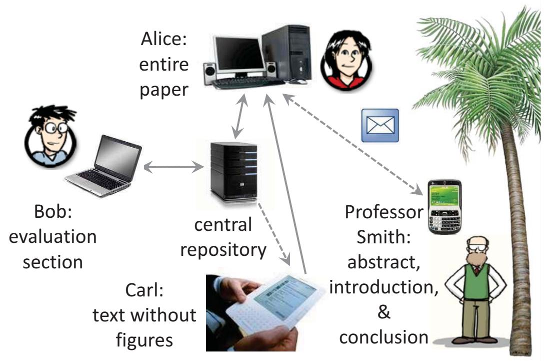

Alice: entire paper
Bob: evaluation section
central repository
Carl: text without figures
Professor Smith: abstract, introduction, & conclusion

Figure 1: **Example scenario to illustrate collaborative editing activities from mobile devices.**

Although he is on vacation in Fiji, Professor Smith is overseeing the submission. He is particularly concerned about how the paper is framed, so he plans to take a close look at the paper's abstract, introduction, and conclusion. However, he is equipped only with a smart phone that has limited editing capabilities, and to make matters worse, he is using an international roaming data connection that charges by the byte. The unfortunate Professor Smith has requested that Alice and Bob send him the relevant sections of the paper via email; he will return his edits by email as well.

Because Alice and Bob want feedback on the entire paper, they recruit Carl, an experienced researcher in their group, to review the whole submission. Carl is happy to help them; he will be able to try out his new e-book reader during his commute home by train. He makes comments and annotations on the document and uploads them to Alice and Bob through a rather spotty mobile data connection. Meanwhile, Alice has set up a centralized repository so her co-author Bob can upload his contributions to the paper, while she takes responsibility for merging Professor Smith's revisions and Carl's suggestions into a new version of the paper.

2.2 Requirements

This scenario illustrates some common situations that arise when people edit documents on mobile devices. These situations highlight three broad requirements of applications intended to support this activity.

- **Uncoordinated multi-author editing.** Despite the advent of mobile broadband networks, mobile data connectivity remains unreliable even in urban areas, and the latency of accessing services over 3G networks remains high. Consequently, users want interactive applications, such as editors, to run directly on their mobile devices and to continue operating if the device becomes disconnected. Local editing leads to the possibility that multiple authors may make simultaneous, uncoordinated changes to the same document. Users expect that concurrent operations will be identified and merged automatically when possible, and that they will be able to override any inappropriate automatic merges.

- **Dynamic content adaptation.** Because mobile devices may have bandwidth, power, and storage limitations, users may find it advantageous to work with partial copies of longer documents. They may only be interested in portions of the document, or it may be impractical to work with a complete copy locally. Furthermore, transmission problems sometimes lead to truncated or incomplete documents; this need not prevent users from editing the content they have successfully received. In general, users must be able to edit and synchronize changes to these partial copies.

- **Flexible sharing and synchronization.** Users may find themselves in a variety of situations when they collaborate; thus they may prefer different mechanisms for sharing documents and synchronizing their changes. Systems such as cloud services that offer a single point of synchronization may seem risky to some users: the service may be unavailable or may violate privacy. Such users may instead prefer to exchange updated documents through other means such as e-mail, peer-to-peer replication platforms, or removable media [8]. Therefore, our mobile collaborative editing application must have a flexible synchronization architecture.

Although each of these challenges is familiar to systems researchers, to-date they have not been addressed in a unified application framework. For example, our previous work on Polyjuz [17] supports peer-to-peer weakly consistent replication on content-adapted, or reduced fidelity, data. However, Polyjuz is designed to handle well-structured data (such as simple database records consisting of attribute-value pairs), rather than textual content, and does not support the fine-grained updates and concurrent conflicts that arise in the course of collaborative editing. In the rest of this section, we elaborate on the three requirements from the perspective of system design and point out the challenges that have not been fully addressed by prior work.

2.3 System implications and challenges

Uncoordinated multi-author editing

Our scholarly writing scenario–along with other collaborative editing scenarios–suggests the following general system architecture. A document is replicated on multiple devices owned by the same or different authors. Devices may be mobile and are supported by different types of network connectivity, depending on availability. Users can edit a document as presented to them on the device, even when the device is not connected to any network. When the opportunity arises, a device may synchronize with another device, automatically or user-initiated, exchanging updates to the document.

Weakly consistent replication systems such as Bayou [11], Cimbiosys [13], and PRACTI [2] support this system architecture. They provide efficient mechanisms for distinguishing *conflicting* updates, which occur when users make updates to the same document from two different devices, from *sequential* updates, which occur when a user makes an update to the document after obtaining an updated copy of the document from another user. Conflict detection enables such replication systems to determine when to apply changes received from a remote device, thereby overwriting outdated local data, and when to report or resolve a conflict.

In collaborative editing scenarios in which co-authors modify a shared document on multiple devices, concurrent updates to the document will be the norm, rather than an exception, unless the authors take turns editing the document. Resolving such conflicts is problematic in current replication systems; author intervention may be required if an application-specific merge procedure fails to resolve conflicting modifications to the document.

Ideally, concurrent updates to unrelated or distant portions of the document should be merged automatically. For instance, if Alice edits the introduction and Bob changes the evaluation section, the application should simply merge the two updates without intervention from either author. On the other hand, if Professor Smith revises the abstract while Alice corrects a typo in it, a purely automated merge strategy might lead to incoherent writing or lost edits. Conflict detection and resolution should take into account the semantics of operations such as insert, delete, update and move.

Current collaborative editing systems allow operations to be tracked at fine granularity. Multiple authors can change the same document simultaneously without immediate coordination; the system ensures that the document state on each device converges over time. However, such systems are not designed to handle content that users are editing at different fidelity levels. Our scenario introduces the notion that not all devices have the complete document.

Dynamic content adaptation

Users may only want (or be able to get) an incomplete or partial copy of a document on a device. The example scenario illustrates two (among many) potential reasons for content adaptation: (1) to reduce the bandwidth consumed, saving money for users with pay-per-use contracts, and (2) to make data transmission more efficient by eliminating unnecessary parts of the document, thus providing a better user experience.

Previous research has explored dynamic content adaptation to address devices' bandwidth and form factor constraints. Projects such as Odyssey [9], PageTailor [3], and Puppeteer [5] have primarily focused on omitting embedded multimedia or reducing content size through transcoding. In this paper, we take content adaptation a step further and focus on adapting textual content for widely used document formats. Specifically, we dynamically adapt the content structure of documents so that partial content can be useful to users.

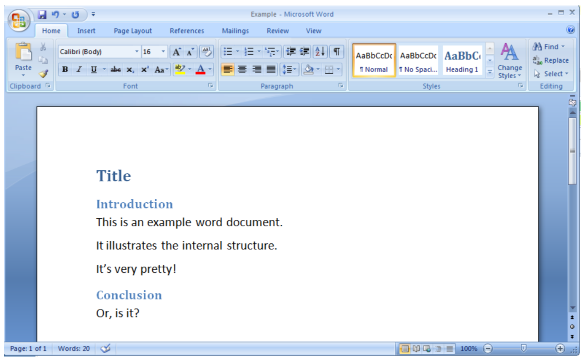
Using XML structure to adapt textual content provides several benefits. Often people are only concerned with a limited portion of a longer document or large spreadsheet when they are working away from the office; they are focused on a particular task that makes sense away from many of their usual resources. Downloading the entire document or exchanging it during subsequent synchronizations can be costly or frustrating if the network connection is slow. Moreover, fine-grained content adaptation may be used in other ways, such as preserving privacy, implementing a security policy, or reducing complexity. For example, mobile devices used outside the workplace may only receive the portions of a document that do not contain sensitive information.

Yet to achieve these benefits, we must address some countervailing challenges. Specifically, it is more difficult to specify which portions of a semi-structured XML document are of interest to particular users or devices. One way to adapt content is to divide the document into well-defined portions and specify which subset of the portions each device gets. In fact, it is common practice for authors to break a longer document into files that correspond to the document's sections. This work-around allows authors to work on the sections more or less independently (although the more pieces the document is divided into, the more painful it is to manage them). Many document editors provide facilities to help authors reassemble these sections after the editing is finished.

Docx2Go allows devices to selectively download and edit portions of a text document, and in so doing must necessarily address the following research questions. First, how does a device or user define the portion(s) of a document to download? Second, how does the system track the changes to textual elements and synchronize these changes with the full or partial copies of the document on other devices? Third, do the full and partial copies of the collaboratively edited document converge to a consistent state on all devices?

Flexible sharing and synchronization

Currently users can collaboratively edit documents several ways. They may store documents on a central server (as they would if they



Title

Introduction

This is an example word document.

It illustrates the internal structure.

It's very pretty!

Conclusion

Or, is it?

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<w:document xmlns:w="http://schemas.openxmlformats.org/wordprocessingml/2006/main" >
 <w:body>
    <w:p w:rsidR="00CC2B60" w:rsidRDefault="00CC2B60" w:rsidP="00CC2B60">
        <w:pPr><w:pStyle w:val="Heading1" /> </w:pPr>
        <w:r><w:t>Title</w:t></w:r>
    </w:p>
    <w:p w:rsidR="00B20CDB" w:rsidRDefault="00B20CDB" w:rsidP="00B20CDB">
        <w:pPr><w:pStyle w:val="Heading3" /> </w:pPr>
        <w:r><w:t>Introduction</w:t></w:r>
        </w:p>
    <w:p w:rsidR="00B20CDB" w:rsidRDefault="00B20CDB">
        <w:r><w:t>This is an example word document.</w:t></w:r>
    </w:p>
    <w:p w:rsidR="000B4459" w:rsidRDefault="000B4459">
        <w:r><w:t>It illustrates the internal structure.</w:t></w:r>
    </w:p>
    <w:p w:rsidR="00FF49CC" w:rsidRDefault="00FF49CC">
        <w:r><w:t>It's very pretty!</w:t></w:r>
    </w:p>
    <w:p w:rsidR="00B20CDB" w:rsidRDefault="00B20CDB" w:rsidP="00B20CDB">
        <w:pPr><w:pStyle w:val="Heading3" /></w:pPr>
        <w:r><w:t>Conclusion</w:t></w:r>
    </w:p>
    <w:p w:rsidR="0015144E" w:rsidRDefault="0015144E">
    <w:r><w:t>Or, is it?</w:t></w:r>
    </w:p>
 </w:body>
</w:document>
```

Figure 2: An example Microsoft Word document rendered on the screen and its underlying XML structure.

used Google Docs or Office Live) and either edit the documents directly on the server, or edit them locally and upload them later to maintain consistency. These services may provide centralized locking (in lieu of collaborative editing) or some support for disconnected, concurrent operation. Alternatively, authors can share documents through a weakly consistent file replication system, such as Microsoft Sync Framework or Live Mesh, which tracks updates to items and synchronizes updated files among the devices. Such systems often provide peer-to-peer synchronization and conflict detection but not at the fine granularity required by synchronous collaboration. In practice, co-authors often develop their own ad hoc mechanisms for coordinating their writing (using email, for example, or mixed modes of communication) [8].

Because people who work together don't necessarily use the same platforms or connect to the network the same way, an effective collaborative editing application should be agnostic to the transport medium used for sharing and synchronizing updates. Some users adopt the concurrent editing functionality offered by Google Docs or Office Live. Others edit documents using local editors and synchronize the files using versioning systems such as CVS or replication frameworks such as Live Mesh. Others simply exchange updates through email and merge them by hand in the local file system. Each method has its own strengths and shortcomings. In the next section, we show how elements of these different techniques can be assembled–in conjunction with some additional innovations that we will discuss–to build a practical and useful document editing system for mobile devices.

3. DESIGN

Docx2Go supports collaborative editing of a document[1] within a system consisting of multiple devices. One of the devices initially creates the document, which is then replicated on other devices. Each device, called a *replica*, stores a local copy of the document and allows users to perform editing operations on it. A replica obtains its initial copy of the document and subsequent changes to the document by synchronizing with other replicas.

A replica might have a *complete* or a *partial* (content-adapted) local copy of the document. Adaptation of content to generate a partial copy happens during synchronization. The content adaptation process can be driven by the user, by an automated tool that employs clever heuristics for content selection, or may occur naturally as a result of a disrupted network connection. Section 4 describes an example of a content adaptation heuristic that can be applied to Wikipedia pages.

[1]We talk about a single document in this section for clarity; the techniques apply equally well to replicating multiple documents.

An *application* on top of the Docx2Go framework enables editing for a particular document type, such as Word, Excel or PowerPoint. This application understands the document format and implements format-specific operations. It may also provide a custom, collaboration-aware editing tool for users to edit the document. Alternatively, it might just support an off-the-shelf editing tool such as Microsoft Word or PowerPoint, while it serves as the bridge between the editing tool and the Docx2Go framework. Section 4 describes the application's role in greater detail.

3.1 Document structure and decomposition

The Docx2Go framework supports XML-based document types. XML is the de-facto basis for recently proposed open standards, most notably ODF and OOXML. Popular office application suites such as OpenOffice and Microsoft Office have already adopted these standards—ODF and OOXML respectively—for representing text documents, presentations, and spreadsheets.

Figure 2 illustrates the XML structure of a Microsoft Office 2007 Word document, stored as a "docx" file. The document file is essentially a compressed directory that contains additional sub-directories and XML files. A few of the XML files are used to store document metadata, such as formatting styles, fonts, and application settings. The key content file is called *document.xml*; an example of this type of file is shown in Figure 2. The outer layers consist of elements called *document* and *body*. The *body* contains a sequence of sub-elements, each representing a paragraph or other content such as a figure or a table (not shown in the figure). Each of these inner elements may contain additional nested sub-elements. Other documents, such as PowerPoint presentations and Excel spreadsheets, have a similar internal XML representation albeit with a different structure.

The key design decision we made in Docx2Go is to treat a well-formed XML element as the unit of granularity for content adaptation, conflict detection, and update exchange. For example, in a Word document, the above operations can be supported in terms of paragraph, figure, or table elements inside the *body* element. This choice provides a uniquely advantageous point in the tradeoff between flexibility, user experience, and system efficiency, as described below.

Content adaptation. A natural basis for selecting content from a document is to use the visible, logical units of organization in a document: for example, chapters or sections. Unfortunately, a document's logical structure is often simply indicated using visual style. Structural demarcations such as section or chapter headings are only distinguished from other paragraphs by using, for example, a different style or font, as shown in Figure 2. An automated system might therefore require human assistance to identify and separate the logical units.

Element-level content adaptation provides a flexible alternative. A partial document can be now derived in multiple ways: 1) A user can generate the partial version she wants by culling sections from the original. 2) The partial document can be generated automatically using rules (for example, to include all text but omit tables, figures and embedded objects). 3) Or, a truncated version of the document may be the natural result of a terminated network connection that produces a partial document composed of whatever well-formed elements that were downloaded.

Concurrent changes. The research literature describes two different approaches to conflict detection. Weakly consistent replication systems [16] have traditionally treated all concurrent changes to an item as conflicting. By contrast, some collaborative editing systems have avoided conflict detection entirely by breaking down a document into characters [7, 12, 14].

Conflict detection at the level of fine-grained elements provides a balance between the two extremes. Concurrent changes to distinct elements in independent parts of the document—a common occurrence in collaborative editing—need not trigger annoying conflict notifications for the user to inspect and resolve. At the same time, concurrent changes to contextually related units of the document can be identified and the users notified. This unit of conflict detection may vary according to the element type: a paragraph for text, a cell for a table, and the entire figure element for figures. Moreover, the replication metadata required for tracking changes and detecting conflicts can be stored as a special attribute in the XML element.

Update exchange. Finally, recognizing and exchanging updates at the unit of fine-grained elements leads to a bandwidth-efficient synchronization protocol. It enables replicas to exchange just the updated units instead of the entire document each time. Moreover, sophisticated "diff-exchange" protocols can be avoided because the exchanged elements are already small.

Unfortunately, current work on fidelity-aware replication do not support the adaptation of text content. CoFi [4] and Polyjuz [17] work on componentized data that have strict and static structure. XML elements at the granularity we have identified are fluid—they are likely to be moved around—and dynamic—new elements may be created and current elements deleted at any time. The remainder of this section shows how Docx2Go handles replication at the granularity of elements.

3.2 Docx2Go basics

Docx2Go internally represents a document as an *ordered* collection of *elements*. When a document's content is adapted to create a partial document, the elements in the partial document follow the same relative order as in the parent document. Once the partial document is on a user's device, the user can change the order of the document elements through normal editing.

Docx2Go supports four types of editing operations, all performed at the element level: *insert* adds a new element at a specific location in the relative order; *delete* removes a specified element; *update* changes the internal contents of an element; and *move* changes the relative order of a specified element with respect to other elements. We use the term *update* liberally to represent any of the above operations, except in places where distinction is required.

Docx2Go draws upon basic techniques from weakly consistent replication to track updates and identify conflicts [16]. Each element has a unique *element identifier* and a *version number*. A replica issues a new version number upon each operation. This version number consists of the unique *replica identifier* of the updating replica and the replica's *update counter*, which the replica increments after each update. Each element also has a *version vector* that represents its *made-with knowledge* used to detect conflicts. The version vector is simply a list, with one update counter per each replica, indicating the last update performed on that replica for this element. Two different versions of the same element conflict if the made-with knowledge of these two versions are disjoint. Docx2Go adds these replication metadata fields as new attributes to the XML element.

Replicas exchange updated elements through a synchronization process. The synchronization process might be invoked manually, or opportunistically whenever two replicas come in network proximity, or through a specific periodic pattern. Our framework does not impose any restrictions on how synchronizations are initiated. The replicas adhere to the following standard synchronization protocol: A *target* replica, which is the receiver of updates, sends the *source* replica a summary of the updates it already knows about. The source uses this summary to identify the updated elements that the target needs and sends them to the target.

Docx2Go also supports a less efficient alternative to facilitate manual synchronization. The source just sends its entire document to the target along with the replication metadata. The target then invokes Docx2Go locally to merge the received document with the local copy and produce an updated document. This alternative synchronization mechanism is useful when the document is transported via e-mail or copied from a thumb drive.

Three questions remain to be answered: 1) How does Docx2Go deal with changes in the relative order of the elements? 2) How does it handle each of the different operations and merge conflicts between them? And, 3) what can be done to redress the overhead imposed by the replication metadata? The remainder of this section describes how Docx2Go addresses these questions.

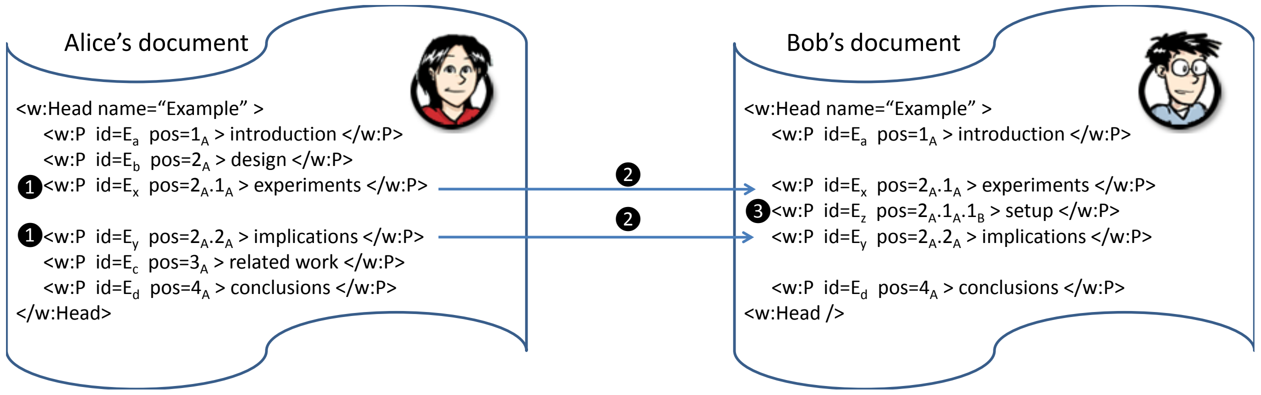
3.3 Order consistency

Docx2Go needs to ensure that the elements in all replicas of a document have a consistent order. The following scenario illustrates the problem: Let replica S have an initial copy of the document consisting of four elements, E_a, E_b, E_c, and E_d, and let replica T have a partial version of the same document that consists of two elements, E_a and E_d. Now suppose that replica S inserts a new element E_x between E_b and E_c. When replica T later receives the element E_x from S, how does T know where to position E_x in its copy of the document?

To date, collaborative editing research has produced two principled approaches to solving this problem. One approach is called *Operational Transforms* (or OT): OT associates a numerical position index with each operation based on the position where the operation was performed on the local state of a replica. Later, a remote replica transforms the operation's position index into a new index—appropriate for the remote replica's state of the document—through a complex algebra of transformation operations [7, 14]. However, OT requires that each replica ultimately receive all operations in the replicated system and thereby excludes efficient partial document editing.

The alternative approach for maintaining order consistency is based on *Concurrent Replicated Data Types* (or CRDT). CRDT associates with each element an extensible position indicator that is persistent across all replicas [12, 18]; that is, unlike the OT approach, CRDT does not require that the position indicators be transformed on different replicas. Docx2Go adopts the CRDT approach for maintaining order consistency, extending it to support partial document editing efficiently.

Order consistency maintenance in Docx2Go works as follows. Each element has an additional metadata attribute called the *posi-*



```
<w:Head name="Example" >
    <w:P  id=E_a  pos=1_A > introduction </w:P>
    <w:P  id=E_b  pos=2_A > design </w:P>
 ❶<w:P  id=E_x  pos=2_A.1_A > experiments </w:P>

 ❶<w:P  id=E_y  pos=2_A.2_A > implications </w:P>
    <w:P  id=E_c  pos=3_A > related work </w:P>
    <w:P  id=E_d  pos=4_A > conclusions </w:P>
</w:Head>
```

```
<w:Head name="Example" >
    <w:P  id=E_a  pos=1_A > introduction </w:P>

    <w:P  id=E_x  pos=2_A.1_A > experiments </w:P>
 ❸<w:P  id=E_z  pos=2_A.1_A.1_B > setup </w:P>
    <w:P  id=E_y  pos=2_A.2_A > implications </w:P>

    <w:P  id=E_d  pos=4_A > conclusions </w:P>
<w:Head />
```

Figure 3: Position indicators (*pos*) establish a consistent order between document elements. The numbers reflect the order of operations: Steps 1 and 3 are inserts, and step 2 is a synchronization.

tion indicator, which determines the element's position in the document relative to its surrounding elements. Therefore, a consistent order for any partial or complete set of elements can be obtained by merely sorting the elements using the position indicator as the key. In the previous example, the document at S has the position indicators 1, 2, 3, and 4 for elements E_a, E_b, E_c, and E_d respectively. The same position indicators 1 and 4 for E_a and E_d appear in the partial document at T. Now, when S inserts E_x between E_b and E_c, E_x gets a new position indicator 2.1, which lies in the middle of E_b's and E_c's position indicators. This position assignment enables S to send to T just the updated element; its embedded position indicator determines the correct order of E_x in T's copy.

The position indicator is extensible and variable in length. For instance, if S had to insert another new paragraph E_y between E_x and E_c, E_y would get the position 2.2; a subsequent insert of E_z between E_x and E_y will result in the position 2.1.1. In this fashion, two position indicators can spawn a new position indicator that lies in the middle. Although a position indicator could ultimately grow to be intractably large, previous evaluations have shown that such growth requires peculiar, worst case insert patterns that rarely occur [12, 18].

We now formalize the position indicator generation process in Docx2Go. A position indicator consists of a concatenated sequence of *position units*, where each position unit is a tuple consisting of a *sequence number* and a *replica identifier*. The replica identifier is used to break ties while ordering two position units and helps Docx2Go to handle concurrent editing scenarios, for example when two or more replicas concurrently insert a new paragraph each between the same two paragraphs (thereby generating the same sequence number). The following operations on position indicators control the ordering process: *compare*, *dense*, *extend* and *generate*. To describe these, we define a term called *corresponding pair* to denote a pair of position units that appear in the same place on two different position indicators.

The *compare* operation computes the relative order of two position indicators, p_1 and p_2. It works as follows. p_1 and p_2 are equal if every corresponding pair in p_1 and p_2 is equal. Otherwise, the position units of p_1 and p_2 are compared in sequence until an unequal corresponding pair is found. The final result is the value obtained from comparing the unequal corresponding pair.

The *dense* operation determines whether two position units are sequential with no intermittent space between them; that is, the sequence numbers are equal or differ by one. This operation supports the *generate* operation.

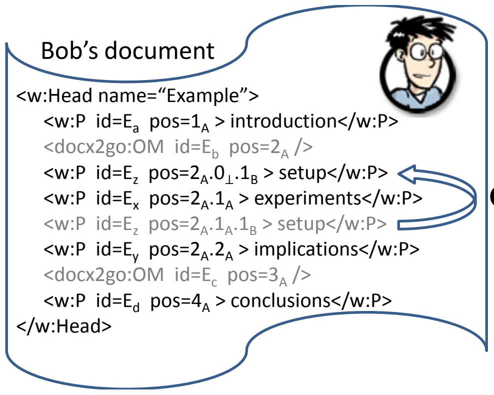
The *extend* operation extends a position indicator with additional position units so that it may have a desired number of position units without changing the position it indicates. The extension is done by concatenating the required number of position units, each with sequence number 0 and a special replica identifier $\perp$. This operation also supports the *generate* operation.

The *generate* operation computes a new position indicator that lies in between two position indicators, p_1 and p_2. It works by first extending the position indicators using the *extend* operation so that they have the same number of position units. Then it traverses the corresponding pairs of position units in sequence until it finds an unequal pair. If the unequal pair is not dense, it creates a new position unit with a sequence number in between that of the unequal pair and the current replica's identifier, and appends it to the previously traversed position units. If the unequal pair is dense, it creates a new position unit with the sequence number one and the current replica's identifier. It then appends the smaller of the two unequal position units to the previously traversed position units, followed by the newly created position unit.

Figure 3 shows how Docx2Go would assign position indicators in the example scenario. Initially Alice's full document has four elements (E_a, E_b, E_c, and E_d) and Bob's partial document has two elements (E_a and E_d). In Step 1, Alice adds two elements E_x and E_y in between E_b and E_c and synchronizes with Bob in Step 2. Bob inserts a new element E_z between E_x and E_y in Step 3.

This position generation process works for full document replication. However, complications arise when replicas have only a partial local copy. For instance in the example shown in Figure 3, if the user (Bob) on replica T moves element E_z to someplace between E_a and E_x, what should the element's new position indicator be? Answering this question correctly requires knowledge of the user's intention. The user could have intended to insert the paragraph either immediately after E_a, immediately before E_x, or in a specific place in the middle.

Docx2Go resolves this problem by keeping a skeleton structure of the complete document, regardless of whether it is full or partial. It maintains the element identifier and position indicator for each element in a data structure called *order metadata*, and synchronizes the order metadata along with the document. Figure 4 illustrates how the order metadata, shown as OM, helps Bob's replica T move the element E_z in front of the element E_x. With the help of the order metadata, a collaboration-aware editing tool can visually display the missing portions in the document to the user, enabling the user to specify the intended location of the new element. Alternatively, the editing tool (collaboration-aware or not) can stick to a default policy about where the new element goes (immediately before or immediately after the closest boundary element) and use information in the order metadata to derive the position indicator.



```
<w:Head name="Example">
    <w:P  id=E_a  pos=1_A > introduction</w:P>
    <docx2go:OM  id=E_b  pos=2_A />
    <w:P  id=E_z  pos=2_A.0_⊥.1_B > setup</w:P>
    <w:P  id=E_x  pos=2_A.1_A > experiments</w:P>
    <w:P  id=E_z  pos=2_A.1_A.1_B > setup</w:P>
    <w:P  id=E_y  pos=2_A.2_A > implications</w:P>
    <docx2go:OM  id=E_c  pos=3_A />
    <w:P  id=E_d  pos=4_A > conclusions</w:P>
</w:Head>
```

Figure 4: Illustrates a move operation (step 4) on Bob's partial document. The order metadata (OM) skeleton structure helps Docx2Go determine the new position.

3.4 Edit operations and conflict management

Docx2Go executes user-initiated operations as follows: For an *insert*, Docx2Go creates replication metadata for the element consisting of a new element identifier, version number, position indicator, and an empty made-with knowledge. For an *update*, it assigns a new version number and adds the old version number to the made-with knowledge. A *move* operation is similar to an *update*, but also sets the position indicator to correspond to the new position.

Docx2Go handles a *delete* operation in a special manner. Although it removes the contents of the element, it does not immediately remove the element itself from the document. Instead, it keeps a record of the delete operation in the form of a *tombstone* so that other replicas can learn about the delete operation. The tombstone is thus a regular document element that appears in the same position as the deleted element. It consists of the replication metadata (with a new version number and updated made-with knowledge) and an additional "deleted" attribute indicating that the element has been deleted. Docx2Go garbage-collects tombstones through a standard protocol used in weakly consisted replication systems [11].

In addition to the above operations, Docx2Go provides a *merge* operation to resolve concurrent edits (*conflicts*). The merge operation takes multiple, conflicting versions of an element and produces a new, merged version. Docx2Go assigns a new version number to this element, combines the made-with knowledge of the conflicting versions, and adds the conflicting version numbers to the made-with knowledge. The content and position indicator for this merged version is supplied by the user or the editing tool.

Docx2Go supports delayed conflict resolution. It provides two properties for managing conflicts: a) all concurrent operations to the same element will be detected; b) the effects of conflicting operations will be retained until explicitly merged. That is, replicas retain conflicting versions of an element and synchronize them all with other replicas that are interested in that element. A user can then invoke an appropriate merge operation after inspecting the concurrent changes. User-mediated conflict resolution is consistent with the common scenario in which documents have a lead author who is responsible for viewing and merging conflicts. Docx2Go ensures conflicting versions will eventually appear on the lead author's replica, enabling her to resolve them.

Alternatively, a collaboration-aware editing tool can resolve conflicts automatically based on policies specified by the user. The following is an example set of policies, which a collaboration-aware editing tool can employ to highlight conflicts to the user or resolve them automatically. We describe them on a case by case basis.

Insert. An insert operation cannot conflict with any other operation since by definition an insert causally precedes a delete, move, or an update.

Delete. Delete-delete conflicts are idempotent (they both result in the same effect of removing the element); therefore, merging them is trivial. For delete-update or delete-move conflicts, the editing tool can either automatically resolve the conflict in favor of the delete or the update. Alternatively, it can highlight the conflict to the user through a visual cue such as a strike through [1].

Update and move. Users can specify which co-author's update operations take precedence over another's, and thereby enable automated conflict resolution. In fact, a few collaborative editing systems [7, 15] impose this policy by default. In many cases, however, users may want to make a case-by-case decision. For conflicts that do not involve *moves*, conflicting versions of an element would appear sequentially in the document—enabling easy visualization, perhaps through background coloring.

3.5 Metadata compaction

Docx2Go assigns replication metadata to each element for conflict detection. The relative size of this metadata may be considerable compared to the contents of the element, especially when the application divides the document at a fine granularity. Among the constituents of the replication metadata, the made-with knowledge, being composed of a version vector, has a size proportional to the number of replicas in the system. Per-element made-with knowledge imposes a considerable overhead even with a moderate number of authors and modest document size.

Recent advances in weakly consistent replication systems help us reduce the overall size of made-with knowledge. A few full replication systems [10, 11] employ a compaction protocol, which reduces made-with knowledge to a single entry that applies to all items in a replica instead of a separate made-with knowledge for each item. Our recent work called Cimbiosys [13] showed how to extend this property called *knowledge singularity* to a partial replication system without sacrificing eventual consistency.

We observe that our approach to support collaborative editing of a partial document is equivalent to Cimbiosys' replication of a partial collection. Each element in the document can be viewed as an independent Cimbiosys item while a Docx2Go replica with a partial document is equivalent to a Cimbiosys partial replica, which replicates only a subset of items in the collection.

Docx2Go reduces metadata overhead by adopting Cimbiosys' techniques for knowledge compaction. In stable state, it keeps one made-with knowledge, consisting of a single version vector, for the whole document. At any particular moment, elements with unsynchronized updates and unresolved conflicts might have individual made-with knowledge. Compaction ensues as the replicas synchronize with each other, exchange updates, and resolve conflicts.

4. IMPLEMENTATION

We have implemented the Docx2Go framework as a layer on top of the Cimbiosys replication platform. Docx2Go, in turn, provides developers with an interface for building collaborative document editing applications. We present an example of such an application, CimWiki, at the end of this section. But first, we describe three important aspects of our Docx2Go implementation: 1) how Docx2Go interacts with Cimbiosys, 2) metadata management, and 3) Docx2Go's application interface.

Cimbiosys provides Docx2Go with replication functionality, so copies of a document can be edited on multiple devices and synchronized opportunistically. Updates to each copy propagate to the

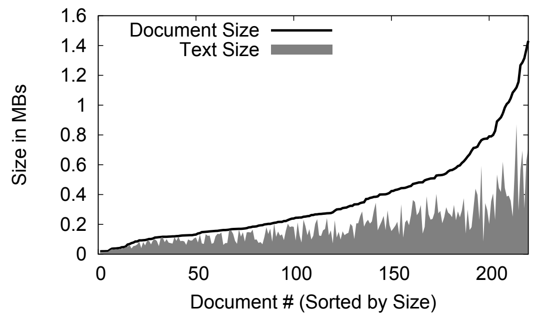


Figure 5: **The distribution of the size of text content in a Wikipedia page compared to its total size.**

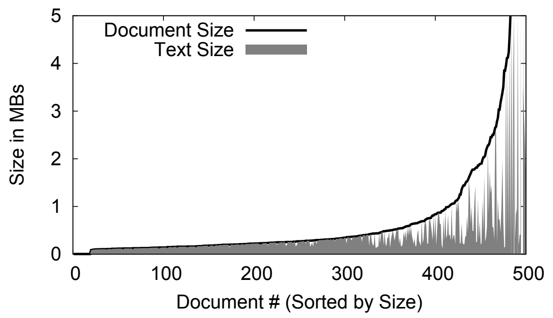


Figure 6: **The distribution of the size of text content in a Word 2007 (docx) document compared to its total size.**

other devices in the system; eventually all of the copies are up-to-date and consistent. When Cimbiosys detects concurrent updates to the document, it notifies the Docx2Go layer so that it can merge the updates appropriately.

Docx2Go acts as an intermediary between Cimbiosys and the document editing application. In this role, it performs three kinds of operations. First, it decomposes the XML document that it receives from the application into its constituent elements so that Cimbiosys can replicate the document on each device as a collection of items. Second, it detects local edits to the document and inserts, updates, and deletes the appropriate elements within Cimbiosys. Finally, it takes the changes that Cimbiosys receives during a synchronization and reconstructs the updated document for the application.

Document decomposition. A *depth* parameter specifies the level of the XML hierarchy to be used in the decomposition operation. Docx2Go treats each distinct element at that depth (or less) in the XML hierarchy as an independent item for replication. Cimbiosys assigns a unique identifier and replication metadata to each item.

Change detection. Docx2Go uses Microsoft's XMLDiffPatch library[2] to detect changes in the document by comparing the application's updated version of the document with a shadow copy that Docx2Go maintains. It then registers the changes it has detected with Cimbiosys and updates the shadow copy to reflect the document's current state.

Document reconstruction. To reconstruct the document from its constituent items, Docx2Go maintains the order metadata described in Section 3. The order metadata (an element identifier and its position indicator) is stored and replicated as a separate Cimbiosys item. Note that concurrent edits to the document—even to unrelated portions—could lead to concurrent updates to the order metadata. Cimbiosys detects and notifies Docx2Go of order metadata conflicts; Docx2Go resolves the conflict automatically by simply combining the conflicting metadata. For conflicting updates to document elements, the current implementation of Docx2Go includes all concurrent updates in the reconstructed document, which it sends to the application.

CIMWiki application

CIMWiki is a sample Docx2Go application for collaboratively editing pages downloaded from Wikipedia. Wikipedia pages are HTML documents, and as such, fit easily within Docx2Go's XML editing paradigm. Thus CIMWiki is a good application for evaluating the performance of Docx2Go and the efficacy of our approach.

All Wikipedia pages have a straightforward hierarchical structure. At the first level is an element called "bodyContent"; at the next level are standard HTML elements such as "div," "h1", "table", "p", and "ul." Although Wikipedia pages may have additional structure embedded in bodyContent's immediate children (for example, a paragraph may contain an embedded table, or a table cell may contain a paragraph of text), we only consider this coarse level of structure–bodyContent and its immediate children–and do not use any more detailed HTML document structure. Thus, when a Wikipedia page is edited in the CIMWiki application, each direct child of bodyContent constitutes a separate item in the Docx2Go framework.

CIMWiki performs content adaptation at the coarse granularity of sections, where a section may consist of multiple HTML elements. Sections are the conventional way of dividing a Wikipedia article into major topics; each section generally has its own heading so a long page may be navigated easily. For example, a politician's biographical page may include sections headed "Early life" and "Political career." By adapting content this way, CIMWiki allows a long, complicated Wikipedia article to broken into more tractable units of sections and edited in coherent chunks.

Because Wikipedia articles are typically broken into sections this way, it is fairly straightforward to parse a Wikipedia page using heuristics. For example, an HTML H2 header tag followed by a SPAN element usually signals the presence of a section name (e.g. "<h2><span ...>Early life</span></h2>" tells us that there is a section titled "Early life"). We can then assume that the material between one section heading and the next corresponds to the section's content. CIMWiki uses patterns of this sort to identify sections and number sections; this parsed version of the HTML document is then passed on to Docx2Go.

CIMWiki uses the content-filtering feature of Cimbiosys to ensure that the appropriate sections of the Wikipedia document are replicated on each device during the synchronization process. In other words, CIMWiki sets a replica's filter to accept the explicit set of sections that the user wants to edit on the local device.

5. EVALUATION

We evaluate Docx2Go using the CIMWiki application for editing Wikipedia pages. Wikipedia is a natural candidate for our evaluation because it provides a large readily available dataset of collaboratively edited documents along with their revision histories. This section focuses on two topics. First, we analyze the characteristics of the Wikipedia workload and discuss their implications for collaborative editing. Next, we present the results of running Docx2Go on this derived workload.

5.1 Analysis of Wikipedia content

We downloaded the complete revision history of 600 Wikipedia pages using Wikipedia's API. We collected these pages using a

[2]`http://msdn.microsoft.com/en-us/library/aa302294.aspx`

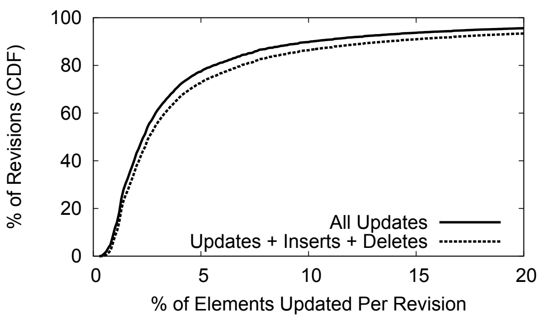


Figure 7: **Distribution of the number of elements updated in one revision of a Wikipedia page.**

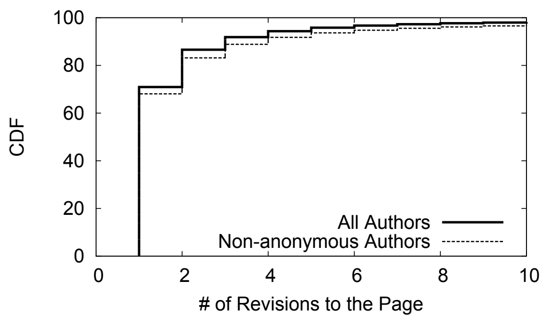


Figure 8: **Distribution of the number of revisions contributed by a single author.**

breadth-first search crawler initiated with a few root pages. The root pages included Wikipedia's Main Page and popular pages such as 'Baseball', 'Harry Potter', and 'University of California'. At least 220 out of the 600 pages had a history of 500 revisions, the maximum number of revisions that Wikipedia's API allows us to download. The remaining pages had 60 to 500 revisions.

Text vs media in documents

We first examined our dataset to understand the contribution of textual content to the total size of the document. Large documents that predominantly consist of text are likely to benefit more from text content adaptation than documents whose text component is small.

Figure 5 compares the quantity of text in a Wikipedia page with the total size of the page. For easy visualization, the figure plots the distribution in increasing page size order. The shaded areas in the figure illustrate the quantity of text in the pages. In general, the relative proportion of text in a Wikipedia page decreases as the page size increases. Yet many large pages still appear to contain a significant amount of text. Overall, the 220 Wikipedia pages had a total size of 81 MB, with 42 MB of text.

We also examined the text to document size ratio for Word 2007 ("docx") documents. We downloaded a sample of 500 Word documents from the Web by searching in Google for keywords 'mobile computing,' 'introduction,' 'documentation,' 'how to,' and 'tutorial,' with "filetype:docx." We selected the top 100 documents from each set of search results and examined their relative text content.

Figure 6 displays the quantity of text in our sample Word documents relative to their total size. The results are similar in quality to those revealed by our analysis of the collection of Wikipedia pages, although the Word documents we analyzed tend to be larger than Wikipedia pages. Small Word documents (less than 0.5 MB) are mostly text. Even though the relative amount of text decreases in larger documents, there are still many large documents with a significant quantity of text. Together, the 500 Word documents had a total size of 450 MB, of which 210 MB was text. Of these, about 70 documents had greater than 0.5 MB of text, and about 30 of them contained over 1 MB of text. Thus, mobile editing applications will benefit from text-based content adaptation.

Characteristics of Wikipedia workload

We next present the characteristics of Wikipedia workload that are relevant to the design of a collaborative editing system. In particular, we examine: 1) the number of elements in a typical Wikipedia page, 2) the number of elements updated in a typical revision of a page, and 3) the number of authors that revise a page and the number of revisions they perform on it. For the purpose of this analysis, the page elements consist of those we identified in our description of the CIMWiki application.

Number of elements in a page. The number of elements in the Wikipedia pages in our dataset varied from 7 to 359. The average number of elements per page was about 127, while over 75% of the pages had more than 81 elements. This result suggests that Docx2Go needs to deal with only a moderate number of elements (hundreds) for replication and conflict management. Other collaborative editors that work with smaller units [7, 12, 15, 18] (words or characters) will face an overhead several-fold higher.

Number of elements updated per revision.

We applied the XMLDiff tool to compare consecutive revisions of a Wikipedia page. That is, we computed the difference between the 1st and the 2nd revisions, the 2nd and the 3rd revisions, and so on until the 499th and 500th revision. We define two metrics to quantify the changes reported by XMLDiff: The *number of elements updated* counts each changed element present in both of the compared revisions. The *number of elements touched* adds the number of new elements and the number of deleted elements to the number of elements updated.

Figure 7 shows the distribution of the metrics. The figure reveals that during most revisions, only a small percentage of the elements are updated. In nearly 77% of the revisions, fewer than 5% of the elements are updated. Even when we add the number of inserted and deleted elements, 72% of revisions affect fewer than 5% of the elements. Overall, the number of elements inserted or deleted is quite small compared with the number of updates to existing elements. In the Wikipedia dataset, we found a total of 521,454 updates, but only 50,657 inserts and 53,430 deletes.

These results corroborate two observations we made earlier in the paper. First, synchronizing only updated elements instead of the entire document will reduce bandwidth consumption substantially. Second, supporting update as a full-fledged operation is beneficial because it provides the required context to understand collaborative edits to elements. By contrast, many collaborative editing systems break an update into an independent insert and a delete, retaining all concurrently inserted elements instead of flagging them as conflicts [7, 12, 15, 18]. This approach may cause more confusion when there are a large number of updates.

Number of authors per page.

We counted the number of unique authors that revise a page using the *username* field. We found that a typical page has many authors. There were at least 86 authors for each page in our trace, the average being about 249 authors per page. Even after we eliminated the anonymous authors (authors whose IP address appears on the pages instead of their username), a minimum of 58 authors, and an average of 129 authors, edited each page. Clearly, many authors revise each popular page, suggesting that the pages undergo a significant amount of collaborative editing.

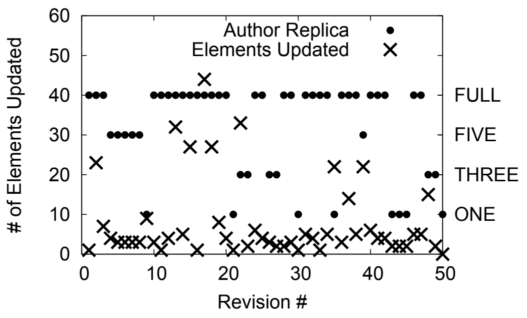


Figure 9: Wikipedia workload. The authoring replica and the number of elements affected for each revision.

We further examined the number of revisions each author contributed to a page. Figure 8 plots the number of revisions that each author made to a page in our dataset. It shows that most authors only make a small number of revisions, and only a few revise the page frequently. Nearly 70% of the authors revised a page just once, whether or not we include the anonymous authors. About 2% of the authors revised the same page more than 10 times during the entire history of 500 revisions. About 36.5% of the total revisions were performed by anonymous authors, but the distribution of the number of revisions remains nearly the same as when these anonymous revisions are ignored.

This revision pattern implies that Docx2Go will need to scale to at least a few hundred users so it can handle the workload introduced by a major collaborative endeavor like Wikipedia. In such cases, knowledge compaction is essential to ensure that the replication metadata overhead does not become unduly large.

5.2 Evaluation of Docx2Go

We evaluated Docx2Go using a workload derived from the Wikipedia dataset. The evaluation was designed to answer the following questions:

- Does Docx2Go achieve eventual convergence in the presence of concurrent edits?
- How much does text-based content adaptation reduce bandwidth and latency?
- How much overhead does Docx2Go metadata entail, and how effective is knowledge compaction in reducing this overhead?

Setup

We set up a system with four replicas, each running the CIMWiki Docx2Go application and collaboratively editing a Wikipedia document. We selected the 'Harry Potter' Wikipedia page, including its history of 500 revisions, as the candidate document for the evaluation. Our experiments with this candidate document provide qualitative answers to the evaluation questions. A precise quantification of the actual convergence time and the bandwidth saved depends on several variables such as the size of the document, the size of the adapted portions, the frequency and number of elements edited, and the efficiency of the network protocol. This simulated workload is sufficient to give us a sense of the efficacy of our approach and its potential benefits when it is used in a range of real-world situations.

The replicas each had different levels of text-based content adaptation. One replica called *full* included all of the sections in the document. The other three consisted of the first *one*, *three* and *five* sections respectively. Note that a section of the document usually spans multiple elements. Replicas *full* and *one* represent two extremes of content adaptation—*full* selects all the elements in the document (with ten sections total) and *one* selects only the few elements that constitute the first section. The remaining replicas represent intermediate adaptation levels.

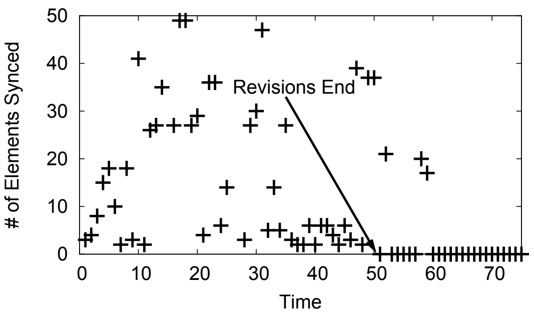


Figure 10: Eventual convergence in the presence of concurrent edits and peer-to-peer synchronization. Each time interval represents one revision and one synchronization.

The experiments used the two different synchronization topologies described below.

Master-slave setup. In the master-slave setup, all revisions to the document were played at a special *master* replica with full content adaptation level. It replayed revisions of the candidate document starting from version 1 until version 500 in the document's revision history. The other four replicas synchronized with the master periodically to fetch updates; the replicas did not synchronize among themselves. The replicas were timed to synchronize once after every revision played by the master. This master-slave setup helped us measure the impact of text-based content adaptation on the amount of data transferred, the latency of synchronization, and the advantage of knowledge compaction.

Peer-to-Peer setup. This setup was based on the scenario described in Section 2 and included peer-to-peer synchronization and concurrent changes. In this setup, there is no master replica; the four replicas in the system replayed the revisions directly to the candidate Wikipedia page. For each revision in the page's revision history, we picked the replica at the lowest adaptation level that included all sections affected by the changes in the revision as the author for that revision. For instance, if only the first section was changed in a revision, then we chose replica *one* as the author for replaying that revision. For each revision replayed, we also set up a synchronization between two randomly selected replicas.

Document convergence

We ran an experiment using the peer-to-peer setup but with all replicas hosted on the same computer. We replayed 50 consecutive revisions of the candidate page at replicas with suitable adaptation level, as described earlier. After the 50th revision was replayed, we continued to invoke synchronizations between randomly chosen pairs of replicas, in order to enable the system to converge.

Figure 9 illustrates the editing operations that occur in this experiment. This figure shows two things for each time unit of one revision: the replica at which the revision was performed and the number of elements affected by that revision. It shows that all replicas performed revisions in our workload each at a different content adaptation level: 30 revisions at *full*, 6 revisions at *five*, 6 at *three*, and 8 revisions at *one*.

Figure 10 shows the number of elements exchanged when two replicas synchronize with each other. It shows that when replicas were replaying updates in the system, a few elements were transferred during a synchronization. This data transfer continues for some time even after the revisions are stopped at time interval 50.

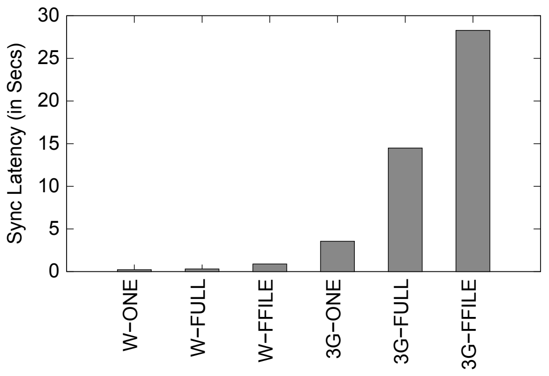


Figure 11: Average synchronization latency for wired and 3G Internet connections experienced by replicas at content adaptation level *one* and *full*, and a strawman replica that synchronizes complete files.

However, the system quickly converges by time interval 60 (there are no data transfers beyond this point). We verified that the elements and their order in the document were consistent on all the replicas at the end of the test and matched the state of the Wikipedia page in the revision history.

Impact of content adaptation

We used two metrics to evaluate the benefits of text-based content adaptation: 1) the latency experienced by a user downloading (synchronizing) updates from the master and 2) the total amount of data transferred over the network during synchronization.

Latency measurements on mobile broadband.

We set up a master replica in Santa Barbara, CA and the slaves on a laptop in San Francisco, CA. The slaves synchronized periodically with the master, while the master was replayed 50 revisions of the candidate Wikipedia page. We ran this experiment in two modes: first, by connecting the laptop to a wired network and second, by using Sprint's 3G Mobile Broadband network (via a laptop connect card).

Figure 11 shows the average synchronization latency in these experiments for *full* and *one*. In addition, we also measured the latency for a scenario called *ffile*, in which the replicas downloaded the entire document from the master during each synchronization.

For both the wired and the 3G network experiments, the download latency increased from *one* to *full* to *ffile*. This trend is expected because there was a corresponding increase in the amount of data downloaded; *one* only fetched updates to elements in the first section, *full* fetched updates to all elements, and *ffile* fetched all elements, regardless of whether they had been changed or not.

The more important trend, however, appears between the wired and the 3G network latencies. The synchronization latency increases from barely noticeable, a few seconds in the wired network, to between fifteen and thirty seconds in the 3G network. The improvement in latency due to text-based content adaptation is therefore more pronounced for the 3G network. For bigger documents and in areas with spotty broadband coverage, the latency may improve substantially.

Bandwidth measurements.

Figure 12 shows the total amount of data transferred between replicas during synchronization. It confirms the expected trend that the amount of data transferred decreases with more selective content adaptation. In this experiment, the replica at level *one* received nearly 8 times less data compared to the replica at level *full*. Other replicas with an intermediate content adaptation level have a proportional decrease in bandwidth consumption as well.

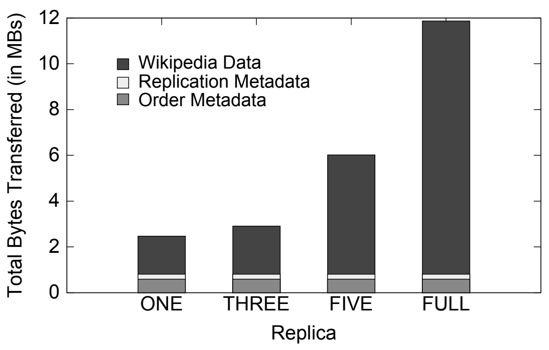


Figure 12: Bandwidth consumption for transferring data, order metadata, and replication metadata to replicas with different content adaptation levels.

Metadata overhead

Figure 12 also plots the total metadata overhead broken down into replication metadata and order metadata. The replication metadata is proportional to the content adaptation level since it depends on the number of elements. By contrast, the order metadata imposes a constant overhead across all replicas, since it carries the same skeleton document structure for every replica. In comparison to the amount of data transferred, the metadata overhead is relatively high for replicas with selected content (*one* and *three*) but quite low for more complete replicas (*full* and *five*).

Finally, to investigate the benefit of metadata compaction, we ran an experiment with compaction disabled. In this experiment, the master revised the page 250 times, once per interval, and the slaves synchronized with the master, again once per interval. We measured the amount of replication metadata transferred to the slaves during synchronization and plotted it for the *full* replica.

Figure 13 shows how replication metadata grows over time in this setup, with and without compaction. Under the condition without compaction, the size of the replication metadata increases with the number of revisions; initially it grows very quickly, and eventually converges. Under the other condition, with compaction, the replication metadata remains roughly the same size. At the end of our test, the size of the replication metadata remained below 1KB with compaction enabled, but grew to nearly 1MB without compaction. Thus compaction helps in keeping the metadata overhead low.

6. CONCLUSION

Supporting editing on the go requires a close look at complex real-world situations. In an era of near-ubiquitous connectivity, simply storing the shared document on a server and editing it in place diminishes the possibility of conflicting edits, out-of-date copies, or platform incompatibilities. But many examples of breakdowns in this approach are readily identified, especially when we factor in the great diversity of mobile device capabilities, and the less than perfect network connectivity users invariably experience when they are away from their usual workplaces. Editing a local copy of a document using the mobile devices and network capacity at hand is often more convenient and flexible than relying on a centralized repository and continuous access to coordinate synchronous changes.

Docx2Go is a framework for editing XML documents in this type of heterogeneous mobile environment. Docx2Go uses XML's fine-grained representation of document structure as a linchpin for its specific innovations that draw on techniques from weakly consistent replication, content adaptation, and collaborative editing. First, this representation is used as a basis for promoting the efficient syn-

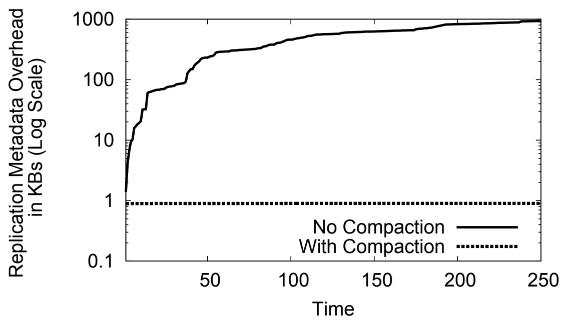


Figure 13: Growth of replication metadata with and without compaction.

chronization of independent local copies of a document. It also enables the system to support textual content adaptation—the ability to reduce document resolution to match device and network affordances—to improve users' mobile editing experience. Finally, Docx2Go resolves conflicts between different devices' independent editing operations using a method of identifying and ordering the XML elements. By bringing together these three techniques under the umbrella of a common document representation, we have been able to address the complex requirements of editing on the go.

Thus far we have implemented and evaluated the performance of the general framework described in this paper. Our future work includes extending the Docx2Go framework by developing applications to handle other types of XML documents, specifically those produced by Microsoft Office tools such as Word, PowerPoint, and Excel. In the scenarios, examples, and evaluation presented in this paper, we have focused on familiar types of XML documents such as scholarly papers and Wikipedia pages; in future work, we intend to investigate more diverse application domains such as eScience, which sometimes rely on large Excel spreadsheets, and may benefit in new ways from Docx2Go's flexible content adaptation capabilities. By developing and deploying a range of Docx2Go applications, we will be able to test the efficacy of our approach using real documents, networks, devices, and collaborations.

7. REFERENCES

[1] ALSHATTNAWI, S., CANALS, G., AND MOLLI, P. Concurrency awareness in a p2p wiki system. In *Proc. of International Symposium on Collaborative Technologies and Systems (CTS)* (Irvine, CA, May 2008).

[2] BELARAMANI, N., DAHLIN, M., GAO, L., NAYATE, A., VENKATARAMANI, A., YALAGANDULA, P., AND ZHENG, J. PRACTI replication. In *Proc. of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (San Jose, CA, May 2006).

[3] BILA, N., RONDA, T., MOHOMED, I., TRUONG, K. N., AND DE LARA, E. PageTailor: Reusable end-user customization for the mobile web. In *Proc. of the ACM Conference on Mobile Systems, Applications and Services (MobiSys)* (San Juan, Puerto Rico, June 2007).

[4] DE LARA, E., KUMAR, R., WALLACH, D. S., AND ZWAENEPOEL, W. Collaboration and multimedia authoring on mobile devices. In *Proc. of the ACM Conference on Mobile Systems, Applications and Services (MobiSys)* (San Francisco, CA, May 2003).

[5] DE LARA, E., WALLACH, D. S., AND ZWAENEPOEL, W. Puppeteer: Component-based adaptation for mobile computing. In *Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS)* (San Francisco, CA, Mar. 2001).

[6] FOX, A., GRIBBLE, S. D., BREWER, E. A., AND AMIR, E. Adapting to network and client variability via on-demand dynamic distillation. In *Proc. of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Cambridge, MA, Oct. 1996).

[7] LI, R., AND LI, D. A landmark-based transformation approach to concurrency control in group editors. In *Proc. of the International ACM Conference on Supporting Group Work (GROUP)* (Sanibel Island, FL, Nov. 2005).

[8] MARSHALL, C. C. From writing and analysis to the repository: Taking the scholars' perspective on scholarly archiving. In *Proc. of the ACM/IEEE-CS Joint Conference on Digital Libraries (JCDL)* (June 2008).

[9] NOBLE, B. D., SATYANARAYANAN, M., NARAYANAN, D., TILTON, J. E., FLINN, J., AND WALKER, K. R. Agile, application-aware adaptation for mobility. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)* (Saint Malo, France, Oct. 1997).

[10] NOVIK, L., HUDIS, I., TERRY, D. B., ANAND, S., JHAVERI, V., SHAH, A., AND WU, Y. Peer-to-peer replication in WinFS. Technical Report MSR-TR-2006-78, Microsoft Research, June 2006.

[11] PETERSEN, K., SPREITZER, M. J., TERRY, D. B., THEIMER, M. M., AND DEMERS, A. J. Flexible update propagation for weakly consistent replication. In *Proc. of the ACM Symposium on Operating Systems Principles (SOSP)* (Saint Malo, France, Oct. 1997).

[12] PREGUICA, N., MARQUES, J. M., SHAPIRO, M., AND LETIA, M. A commutative replicated data type for cooperative editing. In *Proc. of International Conference on Distributed Computing Systems (ICDCS)* (Montreal, Canada, June 2009).

[13] RAMASUBRAMANIAN, V., RODEHEFFER, T. L., TERRY, D. B., WALRAED-SULLIVAN, M., WOBBER, T., MARSHALL, C. C., AND VAHDAT, A. Cimbiosys: A platform for content-based partial replication. In *Proc. of the USENIX Conference on Networked Systems Design and Implementation (NSDI)* (Boston, MA, Apr. 2009).

[14] SUN, C., AND ELLIS, C. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of the ACM Conference on Computer Supported Cooperative work (CSCW)* (Seattle, WA, Nov. 1998).

[15] SUN, C., JIA, X., ZHANG, Y., YANG, Y., AND CHEN, D. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interactions 5*, 1 (1998).

[16] TERRY, D. B. *Replicated Data Management for Mobile Computing*. Morgan & Claypool Publishers, 2008.

[17] VEERARAGHAVAN, K., RAMASUBRAMANIAN, V., RODEHEFFER, T., TERRY, D. B., AND WOBBER, T. Fidelity-aware replication for mobile devices. In *Proc. of the ACM Conference on Mobile Systems, Applications and Services (MobiSys)* (Krakow, Poland, June 2009).

[18] WEISS, S., URSO, P., AND MOLLI, P. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *Proc. of International Conference on Distributed Computing Systems (ICDCS)* (Montreal, Canada, June 2009).