Collaboration and Multimedia Authoring on Mobile Devices

Eyal de Lara[◇], Rajnish Kumar[†], Dan S. Wallach[†], and Willy Zwaenepoel[‡]

[◇]Deparment of Computer Science
University of Toronto
delara@cs.toronto.edu

[†]Department of Computer Science
Rice University
{rajnish,dwallach}@cs.rice.edu

[‡]School of Computer and Communication Science
Ecole Polytechnique Federale de Lausanne, Switzerland
willy.zwaenepoel@epfl.ch

Abstract

This paper introduces *adaptation-aware editing* and *progressive update propagation*, two novel mechanisms that enable authoring multimedia content and collaborative work on mobile devices. Adaptation-aware editing enables editing content that was adapted to reduce download time to the mobile device. Progressive update propagation reduces the time for propagating content generated at the mobile device by transmitting either a fraction of the modifications or transcoded versions thereof.

With application-aware editing and progressive update propagation, an object present at a mobile device is characterized not only by a particular version, as in conventional replication, but also by a particular fidelity. We demonstrate that replication models can be extended to account for fidelity independently of the mechanisms used for concurrency control and consistency maintenance. As a result, the two techniques described in this paper can easily be added to any replication protocol, whether optimistic or pessimistic.

We report on our experience implementing adaptation-aware editing and progressive update propagation. Experiments with two multimedia applications, an email reader and a presentation software package, show that both mechanisms can be added with modest programming effort and achieve substantial reductions in upload and download latencies.

1 Introduction

Research on mobile computing has made significant progress in adapting applications for viewing multimedia content on mobile devices [5, 8, 21]. Multimedia authoring and collaborative work on these platforms remain, however, open problems.

We identify three factors that hinder multimedia authoring and collaborative work over bandwidth-limited links:

1. Read adaptations. The adaptation techniques used to lower resource usage (e.g., energy, bandwidth) may result in situations where content present at the mobile device differs significantly from the versions stored at the server. Typical adaptation techniques adapt by downloading just a fraction of a multimedia document, or by trancoding content into lower-fidelity representations. Naively storing user modifications made to an adapted document may delete elements that were not present at the mobile device, or it may replace high-fidelity data with the transcoded versions sent to the mobile device (even in cases where the user did not modify the transcoded elements).
2. Large updates. Mobile users can generate large multimedia content (e.g., photographs, drawings, audio notes) whose propagation may result in large resource expenditures or long upload latencies over a bandwidth-limited link .
3. Conflicts. The use of optimistic replication models [12, 23] allows concurrent modifications that may conflict with each other. Conflicts can occur in other circumstances as well, but low bandwidth and the possibility of frequent disconnection make their occurrence more likely.

This paper introduces *adaptation-aware editing* and *progressive update propagation*, two novel mechanisms that enable document authoring and collaborative work over bandwidth-limited links. These mechanisms extend traditional replication models to account for the fidelity level of replicated content. Both mechanisms decompose multimedia documents into their component structure (e.g., pages, images, sounds, video), and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing enables editing adapted documents by differentiating between modifications made by the user and those that result from adaptation. Progressive update propagation reduces the time and the resources required to propagate components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of those modifications. Adaptation-aware editing and progressive update propagation also reduce the likelihood of update conflicts in two ways. First, by working at the component level rather than the whole-document level, they reduce the sharing granularity. Second, because both mechanisms lower the cost to download and upload component data, they encourage more frequent communication, hence increasing the awareness that users have of their collaborators' activities [3].

By reducing the cost of propagating multimedia content, adaptation-aware editing and progressive update propagation enable new types of applications and extend the reach of existing applications into the mobile realm. The following two examples illustrate the use of both mechanisms:

1. *Maintenance*. A work crew inspects damage to a plant caused by an explosion. They use a digital camera to take pictures of the problem area, and send the pictures over a wireless connection to the head office. Since bandwidth is low, and they want an urgent assessment of the seriousness of the situation, they use progressive update propagation to initially send low-resolution versions of the pictures. These initial images allow the head office to determine quickly that there is no need to declare an emergency, but that repair work nonetheless needs to be started immediately. The crew continues to use progressive update propagation to send higher-resolution versions of the pictures, sufficiently detailed to initiate repairs. The head office forwards these pictures to a trusted contractor and to the insurance company. The contractor uses adaptation-aware editing to indicate the suggested repairs on the pictures, and sends the marked-up pictures back to the head office and the insurance company. Both approve the repairs, and the contractor heads out to the site. When the work crew arrives back at the office, full-resolution pictures are saved for later investigation.
2. *Collaborative presentation design*. A team member on a mobile device takes advantage of adaptation-aware editing to reduce download time by downloading and editing an adapted version of a presentation. The adapted document consists of just a few slides of the original presentation and has low-fidelity images, sounds, and videos. The team member then uses progressive update propagation to share her modifications to the presentation, which include a photograph taken with a digital camera. Progressive update propagation reduces the time for uploading the photograph by sending a low-fidelity version of the image. When the team member reconnects over a high-bandwidth link, the system automatically upgrades the version of the photograph.

The previous scenarios cannot be handled by current adaptation systems that only handle adaptation of read-only content. They also cannot be supported by current replication systems. Propagating transcoded versions of components as described in the above examples, requires the replication model to account for the fidelity level of replicated content. Upgrading the fidelity of an image in a particular version of a document is different from creating a new version with (user) modifications to the document.

This paper shows that fidelity can be added to a replication protocol independently of the mechanisms used for concurrency control and consistency maintenance. Replication models are typically represented by state diagrams, and we follow this general paradigm. We present state diagrams that incorporate the presence of transcoded versions of components, for use with both optimistic and pessimistic replication. The introduction of transcoded component versions is orthogonal to the maintenance of consistency between replicas. More specifically, new states are added to represent transcoded versions, but the semantics of the existing states and the transitions between them remain unchanged. Therefore, fidelity can be added easily to any replication protocol, whether optimistic or pessimistic.

There are several possible implementations of adaptation-aware editing and progressive update propagation. We present a prototype implementation of these mechanisms that takes advantage of existing run-time APIs and structured document formats [5]. This implementation allows us to adapt applications for multimedia authoring and collaboration *without* changing their source code.

We demonstrate our implementation by experimenting with the Outlook email browser and the PowerPoint presentation software. Both applications see large reductions in user-perceived latencies. For Outlook, progressive update propagation reduces the time a wireless author has to stay connected to propagate emails with multimedia attachments. For PowerPoint, adaptation-aware editing and progressive update propagation reduce the time that wireless collaborators need to wait to view changes made to the presentation by their colleagues.

The rest of this paper is organized as follows. Section 2 introduces adaptation-aware editing and progressive update propagation and explores the implications of extending pessimistic and optimistic replication models to support these mechanisms. Sections 3 and 4 present the design and evaluation of our prototype implementation of adaptation-aware editing and progressive update propagation. Finally, Sections 5 and 6 discuss related work and conclude the paper.

2 Incorporating Fidelity in Replication Protocols

In this section, we describe the implications of extending traditional replication models to provide various degrees of support for adaptation-aware editing and progressive update propagation.

We assume that it is possible to decompose documents into their component structure (e.g., pages, images, sounds, video). Component decomposition can be guided by the document's file format, or by a policy set by the content provider or by the client. For example, HTML documents [24], as well as documents from popular productivity tools [4] use well defined tags to signal the presence of multimedia elements such as images, sounds, and videos.

Multiple versions of a component can co-exist in different replicas. Two versions of a component may differ because they have different creation times, and hence reflect different stages in the development of the component, or because they have different fidelity levels. We consider two fidelity classes: *full* and *partial*. For a given creation time, a component can have only one full-fidelity version but many partial-fidelity versions. A component is present at full fidelity when its version contains data that is equal to the data when the version was created. Conversely, a component is present with partial fidelity if it has been lossily transcoded from the component's original version. Fidelity is by nature a type-specific notion, and hence there can be a type-specific number of different partial-fidelity versions. We assume that it is possible to determine whether one version has higher fidelity than another one.

This discussion considers both pessimistic and optimistic replication models. A pessimistic replication model guarantees that at most one replica modifies a component at any given time, and that a replica does not modify a component while it is being read by some other replica. The mutual exclusion guarantee can be realized by various mechanisms, such as locks or invalidation messages. With optimistic replication, replicas may read and write components without any synchronization. A manual or automatic reconciliation procedure resolves conflicts caused by concurrent writes on different replicas.

Replication models are typically represented by state diagrams, and we follow this general paradigm. The states and transitions for a replication model are independent of the specific mechanisms used for consistency maintenance and depend only on whether the replication model is pessimistic or optimistic. The discussion in this section is therefore independent of specific mechanisms used for consistency maintenance, such as invalidations, leases, or timeouts. The mechanisms for consistency maintenance only determine what events trigger specific transitions (e.g., transition from Clean to Empty on receiving an invalidation message). This discussion is also independent of the specific mechanisms used to propagate versions between replicas. The use of data or operation shipping, as well as optimizations, such a version diffing, are implementation decisions that do not affect the underlying replication model.

We consider both primary replica and serverless approaches. In a primary replica approach, a server holds the primary replica of the document. Clients can replicate subsets or all of the document's components by reading them from the server's primary replica. Client modifications are sent to the server, and there is no direct communication between clients. In contrast, in a serverless configuration there is no centralized server or primary replica and replicas communicate directly.

In the rest of this section, we first describe the implications of supporting adaptation-aware editing and progressive update propagation in isolation. We then describe replication models that support both mechanisms. The initial discussion assumes a primary replica. Serverless systems are discussed afterwords.

2.1 Adaptation-Aware Editing

The simplest form of adaptation-aware editing limits users to modifying only components that are loaded with full fidelity at the bandwidth-limited device. Such an implementation requires the replication system to keep track of which components are available at the bandwidth-limited device and whether these components have been transcoded into partial-fidelity versions. This information is normally already present in replication systems or can be easily added. The replication system then prevents users from modifying any component that is not present with full fidelity.

A simple extension to the previous model is to allow users to (completely) overwrite or delete partial-fidelity components or components that were not included in the client's replica subset. In this scenario, the user can (completely) replace the content of a component that was not loaded or that was loaded at partial fidelity with new full-fidelity content generated at the bandwidth-limited device. The user can also remove a component from the document altogether. Adding this functionality does not require keeping extra state.

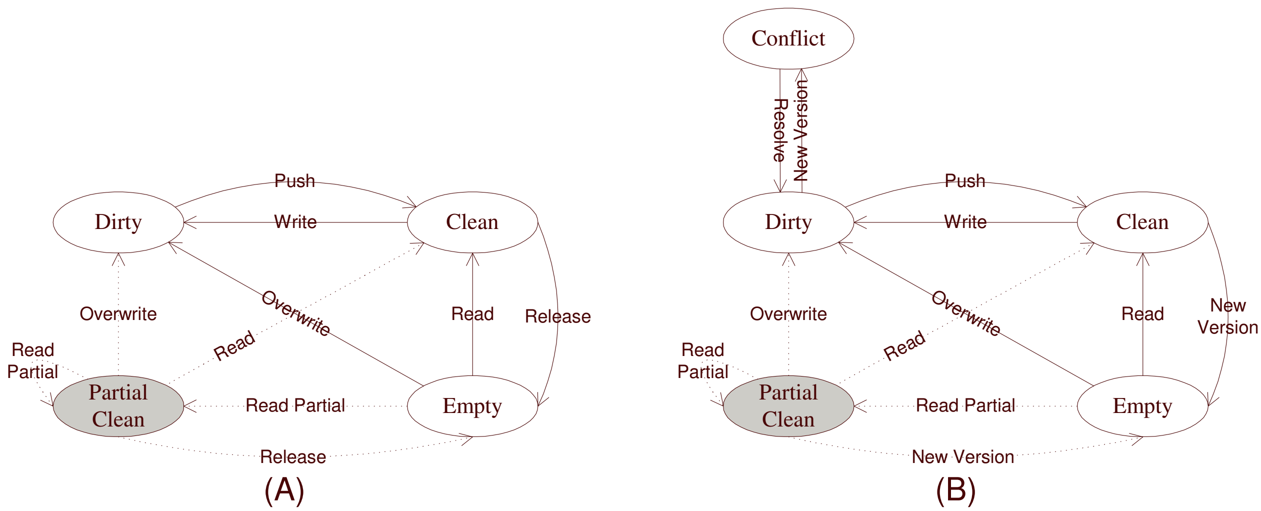

Figure 1: State transition diagram for a pessimistic (A) and an optimistic (B) replication model with support for adaptation-aware editing. Partial-fidelity states and the transitions in and out of these states are represented with gray ovals and dotted arrows, respectively. In contrast, states present in traditional replication models and their transitions are represented with clear ovals and full arrows, respectively.

2.1.1 Pessimistic Replication

Figure 1(A) shows the state transition diagram for individual components of a client replica for a pessimistic replication model that supports modifying full-fidelity component versions and overwriting partial-fidelity component versions. In our state diagrams, we represent new partial-fidelity states by gray ovals and the new transitions in and out of these states by dotted arrows. In contrast, we represent the states present in traditional replication models by clear ovals and their transitions by full arrows. The state diagram for the primary replica (not shown) stays the same as without support for adaptation-aware editing. This diagram contains two states, Empty and Clean, with the obvious meanings.

In the client replica state transition diagram, a component can be in one of four states: Empty, Partial-Clean, Clean, and Dirty. A component is in Empty when it is being edited by some other client replica or when the client chooses not to read it. A component transitions into Partial-Clean when the client replica reads a partial-fidelity version. This version can be further refined by reading higher-fidelity partial-fidelity versions (i.e., Read-Partial) or the component can transition into Clean by reading a full-fidelity version. The component transitions into Dirty when the client replica either modifies a full-fidelity version (i.e., component in Clean state) or overwrites an unloaded component or a partial-fidelity version (i.e., component in Empty or Partial-Clean). The component transitions back to Clean when the client replica propagates a full-fidelity version to the primary replica. Finally, a component transitions back to Empty when the client replica no longer wishes to read the component. Transitions to Empty depend on the specific mechanisms used to guarantee mutual exclusion, and can occur, for example, when the client replica releases a lock or receives an invalidation.

2.1.2 Optimistic Replication

Figure 1(B) shows the state transition diagram for an optimistic replication model. The optimistic replication diagram differs from the pessimistic diagram (Figure 1(A)) in two ways: First, it has an extra state for conflict resolution. The component transitions to the Conflict state when the replica detects the primary replica version and the client replica version are concurrent (i.e., it is not possible to determine a partial ordering for the two versions) [15]. The component transitions back to the Dirty state once the client replica reads the conflicting version and resolves the conflict. Second, transitions from the Clean and Partial-Clean states to the Empty state occur when the client replica learns that the primary replica has a more recent version for the component. The decision of when to transition to Empty is left to the implementation. Some implementations may eagerly invalidate the current version, while others may allow the user to keep working with the current version. In other words, it is the implementation's responsibility to decide how eagerly it wants to act on the consistency information it receives.

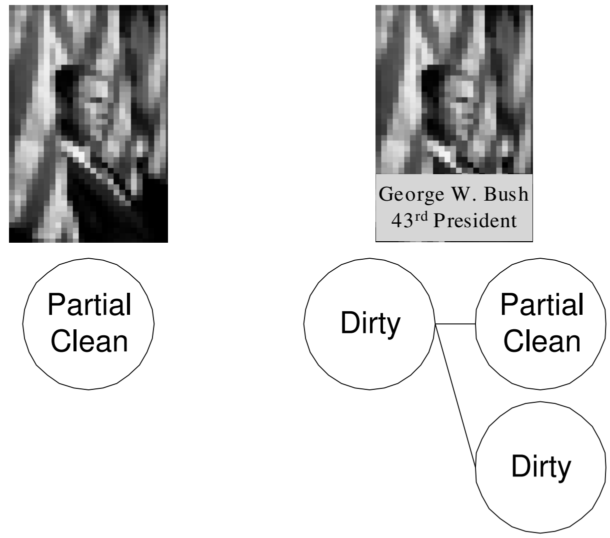

Figure 2: A component is split after modifications to a partial-fidelity version.

2.1.3 Modification of Partially-Loaded Components

A more ambitious form of adaptation-aware editing allows users to modify just a portion of a partial-fidelity version, for example, to replace parts of transcoded images, audio recordings, or video streams. Such modifications result in a component version that contains a mixture of partial- and full-fidelity data, which contravenes our initial assumption that the replication system keeps track of fidelity at the component granularity. While the semantics of some data types, such as images, may support modifications to just parts of the component's version, these semantics are not visible to the replication system. To reflect the changes to the replication system, the component has to be split into two subcomponents as shown on Figure 2. The first subcomponent holds the partial-fidelity data, which was not modified by the user, and the second subcomponent holds the full-fidelity modifications made by the user. The original component (now turned into a container for the two subcomponents) transitions to the Dirty state, to reflect the change in the document's component structure. The subcomponent holding unmodified partial fidelity data remains in the Partial-Clean state. In contrast, the subcomponent holding new full-fidelity data transitions to the Dirty state.

A partial-fidelity version can also be changed by an operation that does not produce any full-fidelity data, for instance, by applying a gray-scale filter to a partial-fidelity image. For these cases, the operation rather than the resulting data has to be propagated to the primary replica, and applied there [16]. After the operation has been propagated and applied to the server's version, the client's version transitions into Partial-Clean if the client's version can be lossily trancoded from the server's version, and to Empty otherwise. In other words, the client's version transitions to Partial-Clean only if the lossy trancoding algorithm used to derive the client's version and the operation being propagated are commutative.

In either of the above cases, reflecting the modifications made to the partial-fidelity version on the full-fidelity version available at the server requires data-type specific instrumentation (i.e., code that knows how to extract the modifications to a partial-fidelity version, and merge them with the full-fidelity version at the server).

2.2 Progressive Update Propagation

A replication system supports progressive update propagation by propagating a subset of the modified components and/or by propagating partial-fidelity versions of modified components.

In this section, we consider the implications of an implementation that supports progressive update propagation but does not support transcoding components on read or editing partial-fidelity components. In such an implementation, client replicas have by default full-fidelity versions of the components they replicate. A client replica has a partial-fidelity version for a component only when the component is being updated by some other client and the updates are being progressively propagated. In other words, the decision to propagate partial-fidelity data is made by the replica that is writing the component and not by the reader, as was the case in the previous section. Moreover, independently of whether we implement a pessimistic or optimistic approach to replication, once a partial-fidelity version has been propagated to the primary replica, it can only be replaced with another version created by the same writer (i.e., a higher-fidelity version or a more recent version). Replacing a partial-fidelity version with a version created by a different writer would require editing partial-fidelity components, which is not allowed by the implementation discussed in this section. In Section 2.3 we describe an implementation that allows a writer to replace a partial-fidelity version created by another writer.

2.2.1 Pessimistic Replication

Supporting progressive update propagation requires adding one new state to the primary replica's state transition diagram (Partial-Clean) and two new states to the client replica's state transition diagram (Pseudo-Dirty and Partial-Clean).

Figures 3 (A) and (B) show the state transition diagram for an individual component in a pessimistic replication model at a client replica and at the primary replica, respectively. The transition diagram for the primary replica

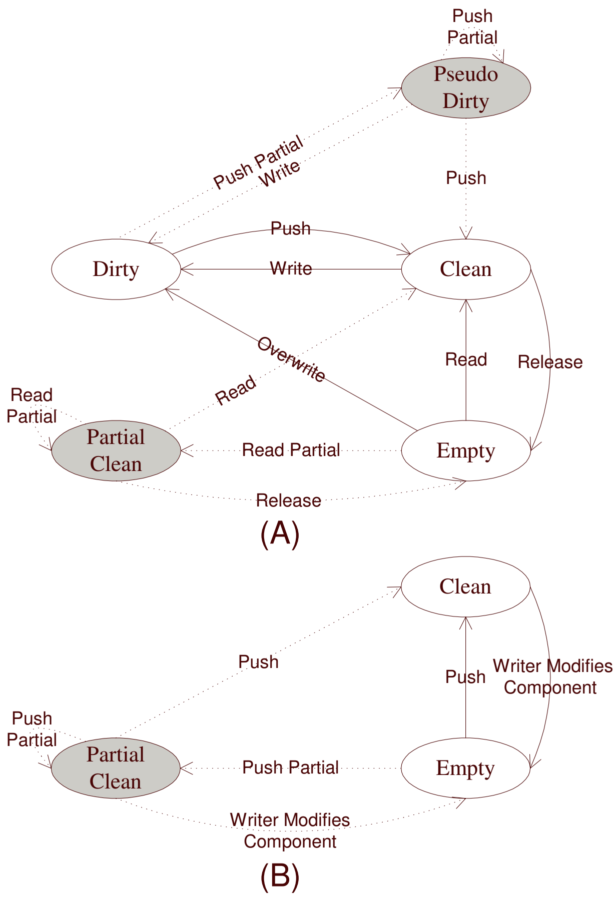

Figure 3: Pessimistic replication state transition diagram for components of the client (A) and primary (B) replicas.

is simple. A component at the primary replica can be in one of three states: Empty, Partial-Clean, and Clean. A component is in the Empty state while it is being edited by a client replica. The component transitions to the Partial-Clean and Clean states when the writer pushes a partial-fidelity or a full-fidelity version of the components, respectively.

A component in a client replica can be in one of five states: Empty, Clean, Partial-Clean, Dirty, and Pseudo-Dirty. A component is in the Empty state either while it is being modified by some other client replica or when the client has chosen not to read it. A component transitions to Clean by reading a full-fidelity version. If only a partial-fidelity version is available at the primary replica because the last writer has not propagated a full-fidelity version yet, the client replica can read this version and transition to Partial-Clean. The component transitions from Clean to Dirty after the client modifies its content. The client can then propagate modifications to the primary replica in two ways. First, the writer can push a full-fidelity version of the modifications, forcing the component at the primary and writer's replica to transition to Clean. Second, the writer can propagate a partial-fidelity version of the component, forcing the component to transition to Partial-Clean in the primary replica, and to Pseudo-Dirty in the writer's replica. The various replicas remain in these states until the writer pushes a full-fidelity version and the component at both the primary and writer's replica transition to Clean. At this time, other client replicas can read the full-fidelity version and transition to Clean. Alternatively, a writer in Pseudo-Dirty can modify the component for a second time and transition to Dirty. If other replicas are to obtain access to a partial-fidelity version, it is imperative that the writer relinquishes exclusive access rights. This enables other replicas to read the partial-fidelity version, but requires the replica in Pseudo-Dirty to re-acquire exclusive access to the component before it can modify it again and transition to Dirty.

2.2.2 Optimistic Replication

In an optimistic replication scheme, before propagating modifications to a component, the writer has to determine if his modifications conflict with other modifications previously reflected at the primary replica. If there is a conflict, the client replica has to resolve it by merging (in a type-specific way) the full-fidelity versions of the conflicting modifications. After resolution, the client replica can propagate a full- or partial-fidelity version of the component to the primary replica. If, however, the primary replica has only a partial-fidelity version for a conflicting component (i.e., the concurrent writer has not propagated a full-fidelity version of its modifications), the two versions cannot be merged as this would violate the restriction on editing partially-loaded components. In this case, conflict resolution has to be delayed until the client replica, which propagated the conflicting partial-fidelity version, propagates a full-fidelity version of its modifications. This problem demonstrates the limitations of implementing partial update propagation for optimistic concurrency control in the absence of adaptation-aware editing. The next section describes how to implement this combination.

2.3 Combining Adaptation-Aware Editing and Progressive Update Propagation

In this section we explore the implications of extending pessimistic and optimistic replication models to support both partial document editing and progressive update propagation. We consider replication systems that support all the features presented in Sections 2.1 and 2.2.

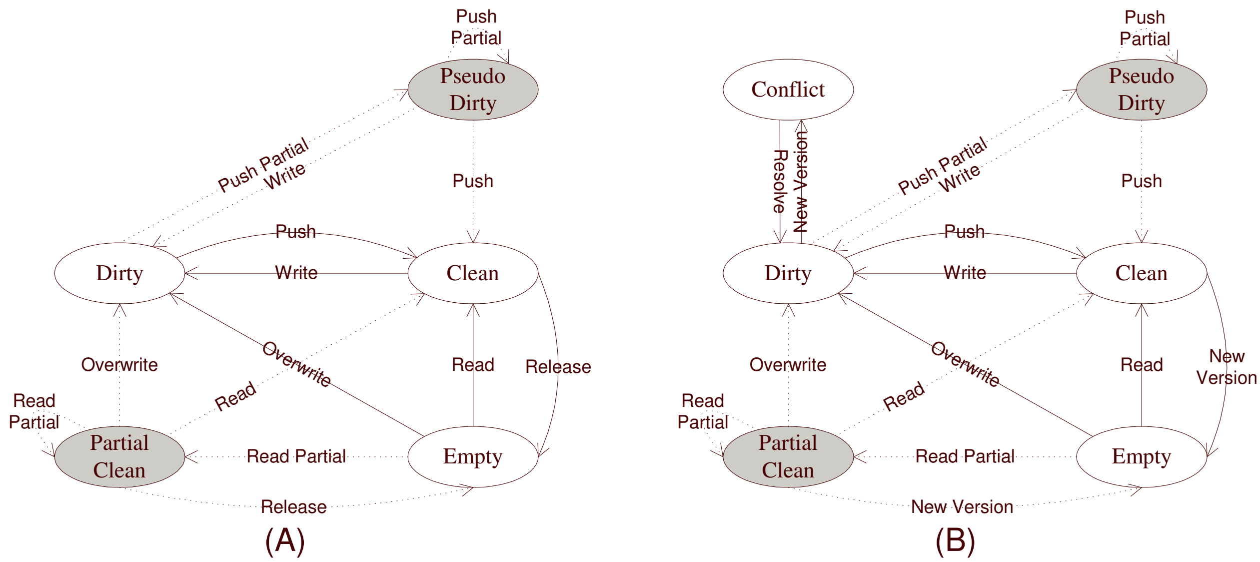

Figure 4: State transition diagrams for individual components of client replicas that support partial document editing and progressive update propagation based on pessimistic (A) and optimistic (B) replication models.

2.3.1 Pessimistic Replication

Figure 4 (A) shows the state transition diagram for components at the client replica for a pessimistic replication system that supports adaptation-aware editing and progressive update propagation. The state transition diagram for components at the primary replica is the same as the one shown in Figure 3 (B).

The diagram in Figure 4 (A) is similar to that of Figure 3 (A) and most states have similar semantics. The semantics of Partial-Clean and Partial-Dirty are, however, a little different. A component may be in the Partial-Clean state because the client requested a partial-fidelity version to reduce its network usage, or because only a partial-fidelity version of the component is available at the primary replica. As was the case in Section 2.2.1, if other replicas are to obtain access to a partial-fidelity version, it is imperative that the writer relinquishes exclusive access rights. Moreover, if another replica is to replace the partial-fidelity version with a later version with new data, the current writer should relinquish all access.

Because adaptation-aware editing is supported, a second client replica can read and modify a component as soon as a partial-fidelity version is available at the primary replica. Two scenarios are possible. First, the second client replica can delete or completely overwrite the component. In this case, the second writer propagates the new version of the component to the primary replica (in either full or partial fidelity), where it supersedes all previous versions, including any version propagated by the first writer. Based on the implementation, any further versions propagated by the first writer are either stored for archival purposes or discarded. Second, the second client replica modifies just a portion of the component. As was the case in Section 2.1.3, propagating the modifications to the partial-fidelity version requires data-type specific instrumentation. This instrumentation may or may not require waiting for the first writer to propagate a full-fidelity version of its modifications. Alternatively, for some data types it may be possible to propagate the second writer's modifications to the primary replica, and merge them lazily with the first writer's modification as they arrive.

2.3.2 Optimistic Replication

Figure 4 (B) shows the state transition diagram for an optimistic replication model that supports partial document editing and progressive update propagation. The state transition diagram is similar to that of the pessimistic replication model we discussed in Section 2.3.1, with states and transitions with the same names having equivalent semantics. The optimistic replication diagram differs in two ways: First, it has an extra state for conflict resolution. Second, it transitions from Clean and Partial-Clean to Empty and from Dirty to Conflict when the client replica learns about a more recent or concurrent component version. As was the case in Section 2.1.2, the eagerness with which the transitions to the Empty state are taken is an implementation decision.

Supporting both adaptation-aware editing and progressive update propagation, also enables client replicas to resolve conflicts even when the server's primary-replica just has a partial-fidelity version for the component. In such case, the client replica reads the conflicting partial-fidelity version, resolves the conflict, and chooses whether to propagate a full- or partial-fidelity version of the modifications. Resolving conflicts using partial-fidelity versions requires data-type specific functionality similar to that described in Section 2.1.3 for reflecting modifications made to partial-fidelity versions.

2.4 Serverless Replication

The earlier state diagrams can be carried over from a primary replica configuration to a serverless configuration without any change. In a serverless configuration, when a replica modifies a component it becomes the source for distributing these modifications to other replicas. In practice, however, not all replicas have to read the modifications directly from the source replica and replicas can get these modifications from some other replica that in turn got the modifications from the source replica.

Independently of how the modifications are propagated, the last writer has a full-fidelity version of the component and is perceived by other replicas as the source for this version. Hence, the replica that writes the component last becomes effectively a temporary "primary replica" for the component that it modified. The states and state transitions otherwise remain the same. If a replica wants to progressively read the component from the primary replica, it needs to maintain a Partial-Clean state. If the temporary primary replica for a particular component wishes to progressively propagate its modifications (i.e., in a push-based implementation), it needs to maintain a Pseudo-Dirty state. If it wants to update more than one replica concurrently, it needs to maintain the progress of each individual transmission as part of that state.

2.5 Summary

We have described the changes necessary to the state diagrams for pessimistic and optimistic replication models in order to support adaptation-aware editing and progressive update propagation. In general, the changes involve adding states and transitions. The existing states and transitions remain with their original semantics. Some complications arise if we allow modifications to partial-fidelity versions, requiring components to be split to reflect old partial-fidelity data and new full-fidelity data. Additionally, data-type specific instrumentation may be required to extract the modifications and reflect them on the full-fidelity version.

3 The CoFi Prototype

This section describes CoFi, a prototype implementation of adaptation-aware editing and progressive update propagation. We named our prototype CoFi because it keeps track of both *co*nsistency and *fi*delity. We first discuss CoFi's system architecture. We then present our optimistic primary replica implementation of adaptation-aware editing and progressive update propagation (see Section 2.3.2).

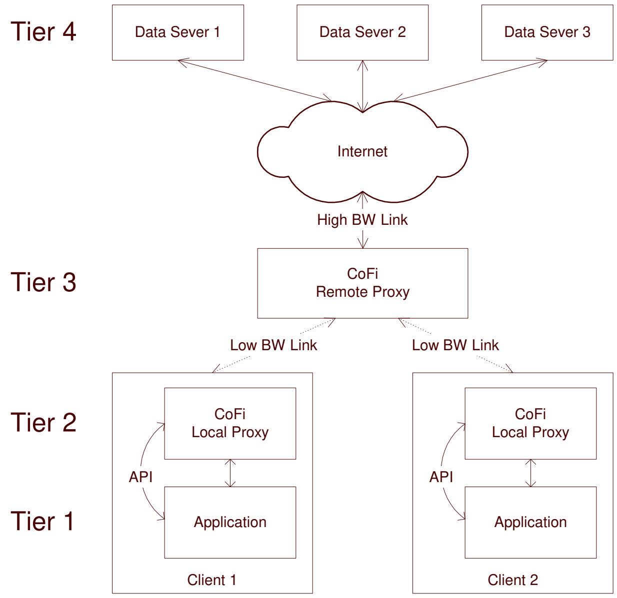

Figure 5: CoFi architecture.

3.1 System Architecture

CoFi adapts applications for collaborative and multimedia authoring over bandwidth-limited networks *without* modifying their source code or the data repositories. CoFi follows the philosophy introduced in Puppeteer for read-only adaptation [5], which takes advantage of the exposed runtime APIs and structured document formats of modern applications.

Figure 5 shows the four-tier CoFi system architecture. It consists of the application(s), a local and a remote proxy, and the data server(s). The application(s) and data server(s) are completely unaware of CoFi. Data servers can be arbitrary repositories of data such as Web servers, file servers, or databases. All communication between the application(s) and the data server(s) goes through the CoFi local and remote proxies that work together to implement adaptation-aware editing and progressive update propagation. The CoFi local proxy runs on the bandwidth-limited device and manipulates the running application through a subset of the application's exported API. The local proxy is also in charge of acquiring user modifications, transcoding component versions, and running the adaptation policies that control the download and upload of component versions. The CoFi remote proxy runs on the

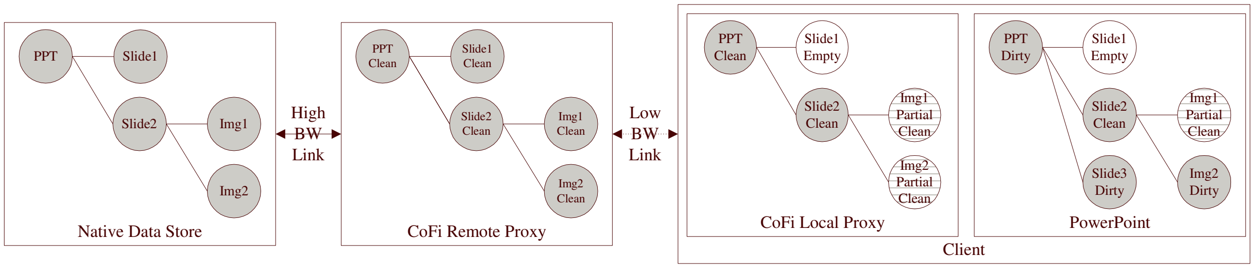

Figure 6: The native data store, CoFi remote and local proxies, and the application can have versions of the document that differ in their component subsets and fidelities.

other side of the bandwidth-limited link and is assumed to have high-bandwidth and low-latency connectivity (relative to the bandwidth-limited device) to the data servers. The CoFi remote proxy is responsible for interacting with the native store and transcoding component versions. Because applications differ in their file formats and run-time APIs, the CoFi proxies rely on component-specific drivers to parse documents and uncover their component structure, to detect user modifications, and to interact with the application's run-time API.

CoFi supports subsetting and versioning adaptation policies. Subsetting policies communicate a subset of the elements of a document, for example, the first page. Versioning policies transmit a less resource-intensive version of some of the components of a document, for example, a low-fidelity version of an image. CoFi adapts applications by extracting subsets and versions from documents. CoFi uses the exported APIs of the applications to incrementally increase the subset of the document or improve the fidelity of the version of the components available to the application. For example, it uses the exported APIs to insert additional pages or higher-fidelity images into the application.

3.2 Optimistic Primary Replica Replication

CoFi implements an optimistic primary replica replication model as described in Section 2.3.2. The prototype consists of a group of bandwidth-limited nodes, each running a CoFi local proxy, that collaborate by exchanging component data over a single CoFi remote proxy, which stores the primary replica of the document. When a document is first opened, it is imported from its native data store into the CoFi remote proxy. Further accesses to the document, both reads and writes, are then served from the CoFi remote proxy's version. In a more complete implementation, there would be multiple remote proxies communicating between each other, but this communication can be implemented by known methods and the current prototype allows us to focus on the novel aspects of CoFi. To enable communication with non CoFi-enabled programs, CoFi exports document modifications back to their native data store.

In CoFi, several versions of a document co-exist in various parts of the system. Figure 6 exemplifies the state of the system for a single client editing a PowerPoint document. The figure shows that there is one version of the document in each of the native data store, the CoFi remote and local proxy, and the application. Moreover, the figure shows that these versions differ in their component subsets and component fidelities. In the example, the native data store and the CoFi remote proxy have complete versions of the document. In contrast, both the CoFi local proxy and the application have just incomplete versions: the first slide is empty and the images of the second slide are only present in partial fidelity. Finally, the application version has an extra slide component.

The differences between the versions in the remote and local proxy result from subsetting and versioning adaptations. In contrast, the differences between the versions in the local proxy and the application result from user modifications. The native store and remote proxy versions can differ because the two versions have not been synchronized (i.e., modifications have not been propagated to the native store), or when the native store version is being updated using out-of-band mechanisms (i.e., outside of the CoFi system).

The rest of this section describes how versions of the document converge by exchanging component data. First, we describe how CoFi propagates user modifications from the bandwidth-limited device to the CoFi remote proxy and the data store. Second, we describe how the CoFi local proxy refreshes the application's document version with newer or higher-fidelity component versions.

3.2.1 User Modification Propagation

Adaptation policies running on the CoFi local proxy (as described in Section 3.1) control the propagation of user modifications to the CoFi remote proxy. Update propagation involves four stages: acquiring user modifications, resolving conflicts, transmitting modifications to the remote proxy, and synchronizing the modifications with the document's native store.

Acquire Modifications CoFi acquires user modifications by comparing the local proxy's document version to the application's version. Ideally, CoFi would use the application's exported API to acquire any user modifications. When such functionality is not provided by the application's API, CoFi instructs the application to save a temporal version of the document in the local file system. CoFi then parses the temporary document and compares it to the local proxy version.

Conflict Resolution CoFi detects conflicting modifications by tagging component versions with version numbers, which determine the partial order of modifications in the system. CoFi implements both client- and server-based conflict resolution. In client-based resolution, the local proxy fetches the conflicting version from the remote proxy, resolves the conflict and creates a new version that dominates the two conflicting versions. In server-based resolution, the client pushes its version to the remote proxy, and a resolver executing in the remote proxy creates a new version that merges the conflicting modifications.

When user intervention is necessary to resolve a conflict, conflict resolution is client-based. To facilitate conflict resolution, the application-specific resolution policy can use the application's exported API to present the conflicting component versions in the context of the application's environment.

Modification Transmission A policy running on the local proxy selects the subset of components for which to propagate modifications, as well as the fidelity level for each component in the subset. The policy can later increase the fidelity level of a previously propagated component by re-selecting the component and pushing a higher-fidelity version.

Synchronization with Native Storage CoFi's remote proxy exports documents to their native storage to enable information sharing with clients outside of the CoFi system and to leverage the mechanisms that these storage systems may implement (e.g., availability, fault tolerance, security, etc). Before exporting modifications, CoFi needs to detect if the native store has been modified by an application outside of CoFi's control. CoFi detects and resolves conflicts created by out-of-band modifications using similar mechanisms to those explained above (i.e., by comparing the last-known and current states of the native store).

3.2.2 Refreshing the Application's Version

Adaptation policies can refresh the application's document version to reflect changes made by other users or to increase the fidelity of a component present at partial fidelity. This process involves three steps: fetching newer or higher-fidelity versions, detecting any user modifications to the components about to be updated, and using the application's API to update the application's document version. If the update process detects that the components have been modified by the user, then a conflict has occurred and the modifications have to be merged with the new version fetched from the remote proxy in a component-specific way – following the techniques for conflict resolution described in Section 3.2.1.

3.3 Implementation Details

3.3.1 User Interaction with CoFi

In our current prototype, the applications' toolbars are extended with extra fields for selecting an adaptation policy that determines the fidelity level at which a document is opened or saved. Eventually, CoFi could rely on monitoring of bandwidth or other resources to automatically choose a particular fidelity level.

CoFi also provides a *Component Viewer* window that shows the current state of components in a document. Using this window, users can determine what components are currently loaded in the application, what components are in progress of being loaded, whether modifications to a component have been propagated to the remote proxy and with what fidelity, whether a newer version of a component is known to be available at the remote proxy, and whether a conflict has been detected for any component. Users can also interact with the Component Viewer to control the propagation of component versions.

3.3.2 Client-Server Interactions

The current prototype implementation is client-driven. That is, clients specify when they want to read or write a document and at what fidelity. The client also indicates when it wants to get or send a refinement of an earlier transcoded version. The server does not notify the clients of new versions or new refinements. Such a facility could be added through a callback mechanism, but would leave other aspects of the implementation unchanged.

3.3.3 Relationship to Puppeteer

CoFi shares some of the code base of Puppeteer [5], namely the code to parse document formats, some of the code in the local proxy to interact with the application, and the protocol to interact between the local and remote proxies.

4 Experimental Results

In this section, we report on our experience using CoFi to add adaptation-aware editing and progressive update propagation to the Outlook email client and the PowerPoint presentation system. We implement progressive update propagation for both applications. For PowerPoint we also support adaptation-aware editing. The CoFi drivers and policies we implement for Outlook and PowerPoint consist of 2,365 and 3,315 lines of Java code, respectively.

We measure the performance of CoFi on an experimental platform consisting of three 500 MHz Pentium III machines running Windows 2000. Two of the machines are configured as clients and one as a server. Client machines run the user application and the CoFi local proxy. The server machine runs the CoFi remote proxy. Clients and the CoFi server communicate via a fourth PC running the DummyNet network simulator [26]. This setup allows us to control the bandwidth between clients and server to emulate various network technologies. We use our departmental NFS and IMAP servers as the native stores for our experiments with PowerPoint and Outlook, respectively.

4.1 Outlook

We developed an email service that supports the progressive propagation of images embedded in or attached to emails. On the sender side, the progressive email services use Outlook's email client to generate emails. On the receiving end, we support both CoFi-enabled clients running Outlook and standard third-party email readers.

We implemented emails as CoFi shared documents that are written only by the email's sender but are read by one or more recipients. Our sender adaptation policy propagates the text content of new emails, transcodes images into a progressive JPEG representation and sends only portions of an image's data. The sender can propagate fidelity refinements for images by selecting the email from a special Outlook folder and re-sending it. The image fidelity refinements are available to CoFi-enabled recipients as soon as they reach the CoFi remote proxy. For CoFi-enabled recipients, our adaptation policy fetches the email's text content and transcoded versions of its image attachments. Readers request fidelity refinements by clicking a refresh button added on Outlook's toolbar. Finally, we support third-party email readers by composing a new email message once all images have reached full fidelity.

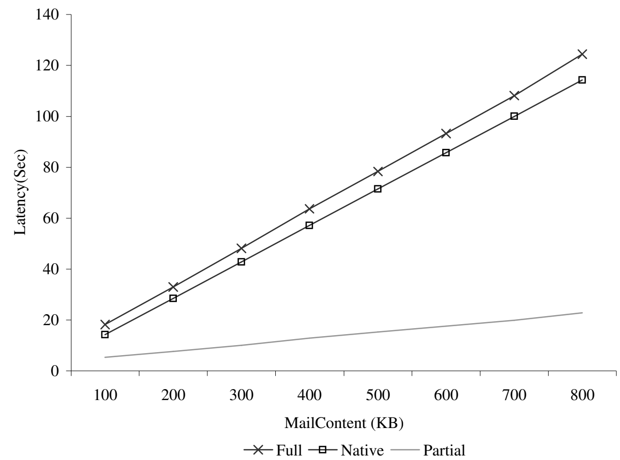

Figure 7: Latency for sending emails with and without progressive update propagation over 56 Kb/sec.

4.1.1 Progressive Update Propagation

Figure 7 plots the latencies for transmitting a set of synthetic emails consisting of a few text paragraphs and a variable number of image attachments each of size 100 KB over a 56 Kbps link. The plots show results for a run that uses Outlook without any adaptation support (*Native*), and two CoFi runs, one that sends the full images (*Full*), and a second that uses versioning to propagate partial-fidelity versions of the images (*Partial*). In this experiment, a partial-fidelity version correspond to the initial $1/7$ of the content of an image encoded in a progressive JPEG representation.

For the *Native* run, we measure only the time it takes to transmit the emails between the mobile client running Outlook and an SMTP server on the other end of the bandwidth-limited link. This accounts for the time that the mobile client has to wait before disconnection in order to propagate the email. We do not include the time it takes for the SMTP server to deliver the email to the recipients, as these operations can be done asynchronously and do not require the mobile client to remain connected. Similarly, for CoFi runs we measure only the time it takes to transmit the emails between the CoFi local and remote proxies and do not include the time needed to compose and send emails to third-party email recipients, or the time it takes for CoFi-enabled recipients to read the email adaptively.

Full demonstrates that the CoFi overhead is small, averaging less than 5% over all emails. In contrast, *Partial* shows that progressive propagation of the attachments reduces the latency by roughly 80%. The 5% overhead in CoFi corresponds to the cost involved in parsing the email content to find its structure, exchanging the control information, and transcoding the images.

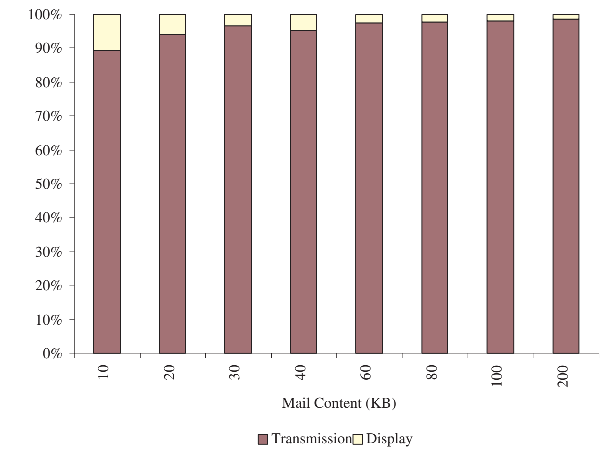

Figure 8: Breakdown of partial update propagation latency while sending emails.

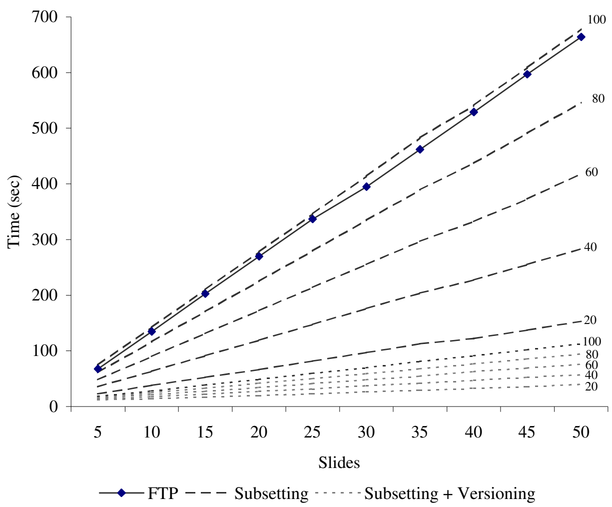

Figure 9: Latency for saving modifications to PowerPoint presentations with and without adaptation over 56 Kb/sec.

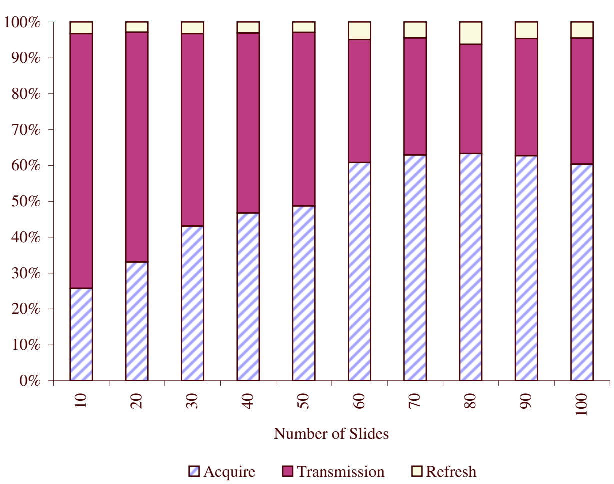

Figure 10: Latency breakdown for upgrading the fidelity of a single image in PowerPoint documents of various sizes.

4.1.2 Fidelity Upgrade

We measured the time it takes for a CoFi-enabled recipient to upgrade a partial-fidelity image to full fidelity. Figure 8 shows that the largest fraction of the time necessary for this fidelity upgrade is due to transmission (Transmission), and that only a small fraction of the time is spent on displaying the upgraded images in the application (Display). In other words, the overhead caused by CoFi's use of the API is very small. While we were not expecting the overhead to be significant, these results confirm that CoFi supports these kinds of adaptations in practice.

4.2 PowerPoint

We adapted PowerPoint to support adaptation-aware editing and progressive update propagation. Adaptation-aware editing reduces download time by enabling mobile clients to edit PowerPoint presentations that have been aggressively adapted. Previous work [5] demonstrated that loading text-only versions of PowerPoint presentations can reduce download latency over bandwidth-limited links by over 90% for large presentations. The rest of this section evaluates the benefits of progressive update propagation, quantifies the latency for updating a PowerPoint presentation with higher-fidelity data, and discusses our conflict resolution policy.

4.2.1 Progressive Update Propagation

CoFi-enabled PowerPoint propagates modifications progressively by saving back just subsets of the modified slides or embedded objects, or by transcoding embedded images into a progressive JPEG representation and saving just portions of the images' data. We implemented an adaptation policy that propagates modifications every time the user saves the document. The policy propagates the text content of any new or modified slides and transcoded versions of new or modified embedded images. Image fidelity refinements are then propagated on every subsequent save request until all images at the CoFi remote proxy reach full fidelity.

We evaluate the effectiveness of progressive update propagation by measuring the latency for saving modifications to a set of synthetic PowerPoint documents. We constructed our synthetic documents by replicating a single slide that contained 4 KB of text and a 80 KB image.

Figure 9 shows latency measurements for saving PowerPoint documents with up to 50 slides over a 56 Kb/sec network link. The figure shows latency results for transferring the documents over FTP and adaptation policies that use subsetting and versioning to reduce the data traffic. The FTP measurements give us a baseline for the time it takes to transfer the full document without any adaptation. We use this baseline to determine the effectiveness of our adaptation policies.

Figure 9 plots the results of 5 experiments that use subsetting to reduce latency. The numbers on the right hand side of the plot, next to each line, show the proportion of document slides that was saved back to the remote proxy. In this manner, the top most line corresponds to documents that were saved in their entirety, while the lowest subsetting line corresponds to documents where only modifications to 20% of the slides were saved. In all experiments, we assume that both the slide's text and single image were modified and had to be saved back. The top most subsetting line, which corresponds to saving the full document, shows that the CoFi overhead is small, averaging less than 5% over all documents. In contrast, all other subsetting experiments show significant reductions in upload latency. The last five lines in Figure 9 show the results for an adaptation policy that uses subsetting and versioning of images to further reduce upload latency. This policy converts images embedded in slides into a progressive JPEG representation and transfers only the initial $1/7$ of the image's data; achieving even larger reductions in upload latency.

4.2.2 Fidelity Upgrade

Figure 10 shows the breakdown of the execution time for updating a single image with higher fidelity data. The figure shows that the API calls to replace the image (Display) account for a small portion of the overall latency, similar to what we saw for Outlook in Figure 8. More significantly, the figure shows that for large documents, roughly 60% of the time for upgrading the fidelity of an image is spent making sure the user did not modify the image we are about to update (Detect). Detecting modifications is time-consuming because PowerPoint's API does not support querying whether a component has been changed. Instead, we detect modifications by saving a temporary copy of the presentation on disk, parsing this copy, and comparing it with the local proxy copy of the presentation. Writing out a copy of the presentation to disk dominates the cost of all other factors in the total time taken for detecting modifications.

This experiment represents a worst-case scenario of having to write out the entire document to upgrade a single image. Under normal operation, we expect modification detection to benefit from PowerPoint's ability to write out modifications incrementally, as well as from the possible batching of multiple component upgrades into a single operation (i.e., updating a set of images at a time). Detecting modification is also only necessary if we allow editing of partial-fidelity images. If we disallow editing partial-fidelity images, then the cost to upgrade an image is just a few milliseconds over the time it takes to transmit the image over the bandwidth-limited link.

In any case, the large modification detection time results from specific limitations of the current PowerPoint API (which could be easily fixed by adding an API call for checking if a component has been changed), and is *not* a fundamental limitation of either adaptation-aware editing or progressive update propagation.

4.2.3 Conflict Resolution

We consider the following conflicts: one user modifies a slide while another user deletes it, two users move a slide to different positions in the presentation, or two users concurrently modify the same slide. We refer to these conflicts as *edit-delete*, *move-move*, and *edit-edit*, respectively. For simplicity, for the rest of this section, we refer to the copy of the presentation available at the remote and local proxies as the *remote* and *local* copies, respectively.

Our PowerPoint policy resolves *edit-delete* and *move-move* conflicts automatically. For *edit-delete* conflicts, our policy prioritizes editing over deletion, recreating the slide in the replica where it was deleted. For *move-move* conflicts, our policy gives priority to the local copy, moving the slide in the remote copy to reflect its position in the local copy. Finally, we resolve *edit-edit* conflicts by using the CoFi PowerPoint driver to present the two conflicting slides to the user, and prompting the user to resolve the conflict by either choosing one of the slides or by merging their content. The above policy is, however, just one of the possible ways to resolve conflicts, and we can easily envision variations or extensions to this simple policy.

The cost of conflict detection is highly dependent on the size of the documents. The bulk of the cost stems from writing out the document (see Section 4.2.2) and from transmitting the data over the network. All other aspects, including the execution of the conflict detection algorithm and the use of the APIs to display conflicts to the user, are insignificant.

5 Related Work

Support for partial propagation of modifications made to a shared database or file system has been provided before. This paper, however, is the first to introduce mechanisms that support propagating partial-fidelity versions of modifications, as well as their progressive improvement. WebDAV [30], and LBFS [20] implement file systems for wide-area and low-bandwidth networks. Coda [14], Ficus [25], and Bayou [29] provide support for document editing on disconnected devices. These systems differ from CoFi in that they are not aware of the fidelity level of the objects they replicate.

While various adaptation systems [1, 5, 7, 8, 11, 12, 17, 19, 21, 28] use subsetting and versioning to reduce document download time, CoFi is the first to provide adaptation support for multimedia authoring and collaborative work over bandwidth-limited devices.

Several efforts [3, 9, 18] have used component-based technologies to implement collaborative applications that adapt to variations on network connectivity, or have implemented collaborative applications that use the document's component structure to reduce conflicts or limit the amount of data that need to be present at the device [2, 6, 13, 22, 27]. These efforts, however, do not allow the propagation of partial-fidelity versions of modifications. MASSIVE-3 [10] uses transcoding to reduce data traffic necessary to keep users of a collaborative virtual world aware of each other. MASSIVE-3, however, implements a pessimistic single-writer consistency model.

6 Conclusions

We have described adaptation-aware editing and progressive update propagation, two novel mechanisms for supporting multimedia authoring and collaborative work on bandwidth-limited devices. Both mechanisms decompose documents into their components structures (e.g., pages, images, paragraphs, sounds) and keep track of consistency and fidelity at a component granularity. Adaptation-aware editing lowers download latencies by enabling users to edit adapted documents. Progressive update propagation shortens the propagation time of components created or modified at the bandwidth-limited device by transmitting subsets of the modified components or transcoded versions of the modifications.

We demonstrate that support for adaptation-aware editing and progressive update propagation can be added to optimistic and pessimistic replication protocols in an orthogonal fashion. Specifically, new states are added to the state machines that describe the replication protocols, but the existing states and transitions remain unaffected.

We have described the implementation of our CoFi prototype, which supports adaptation-aware editing and progressive update propagation for optimistic client-server replication. We have presented performance results for experiments with multimedia authoring and collaboration with two real world applications. For these applications, the ability to edit partially loaded documents and progressively propagate fidelity refinements of modifications substantially reduce upload and download latencies.

While the experiments in this paper focus on document-centric applications, the same principles can be extended to applications with real-time requirements, such as video or audio. Adaptation-aware editing could be used to support video editing, while progressive update propagation would be useful in situations where there is a benefit in retransmitting a higher-fidelity version of a video or audio stream, such as when a user listens to a recording multiple times.

References

[1] David Andersen, Deepak Basal, Dorothy Curtis, Srinivasan Srinivasan, and Hari Balakrishnan. System support for bandwidth management and content adaptation in Internet applications. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, October 2000.

[2] Rhonda Chambers, Dean Crockett, Greg Griffing, and Jehan-Francois Paris. A Java tool for collaborative editing over the Internet. In *Proceedings of the 1998 Energy Sources Technology Conference and Exhibition (ETCE '98)*, Houston, TX, February 1998.

[3] Keith Cheverst, Gordon Blair, Nigel Davies, and Adrian Friday. Supporting collaboration in mobile-aware groupware. *Personal Technologies*, 3(1):33–42, March 1999.

[4] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Opportunities for bandwidth adaptation in Microsoft Office documents. In *Proceedings of the Fourth USENIX Windows Symposium*, Seattle, Washington, August 2000.

[5] Eyal de Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems (USITS)*, San Francisco, California, March 2001.

[6] Dominique Decouchant, Vincent Quint, and Manuel Romero Salcedo. Structured cooperative authoring on the World Wide Web. In *Proceedings of the Fourth International World Wide Web Conference*, Boston, Massachusetts, December 1995.

[7] Christos Efstratio, Keith Cheverst, Nigel Davies, and Adrian Friday. Architectural requirements for the effective support of adaptive mobile applications. In *Proceedings of the Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[8] Armando Fox, Steven D. Gribble, Eric A. Brewer, and Elan Amir. Adapting to network and client variability via on-demand dynamic distillation. *SIGPLAN Notices*, 31(9):160–170, September 1996.

[9] Julian Gallop, Christopher Cooper, Ian Johnson, David Duce, Gordon Blair, Geoff Coulson, and Tom Fitzpatrick. Structuring for extensibility - adapting the past to fit the future. In *Proceedings of The*

CSCW2000 workshop on Component-based Groupware, Philadelphia, Pennsylvania, December 2000.

[10] Chris Greenhalgh, Jim Purbrick, and Dave Snowdon. Inside Massive-3: Flexible support for data consistency and world structuring. In *Proceedings of the Third International Conference on Collaborative Virtual Environments*, San Francisco, California, September 2000.

[11] Anthony D. Joseph, Joshua A. Tauber, and M. Frans Kaashoek. Building reliable mobile-aware applications using the Rover toolkit. In *Proceedings of the 2nd ACM International Conference on Mobile Computing and Networking (MobiCom '96)*, Rye, New York, November 1996.

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

[13] Michael Koch and Jurgen Koch. Application of frameworks in groupware - The Iris group editor environment. *ACM Computing Surveys*, 32(1es), March 2000.

[14] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Winter 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[15] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[16] Yui-Wah Lee, Kwong-Sak Leung, and Mahadev Satyanarayanan. Operation-based update propagation in a mobile file system. In *Proceedings of the USENIX Annual Technical Conference*, Monterrey, California, June 1999.

[17] Baochun Li and Klara Nahrstedt. Qualprobes: Middlewate QoS profiling services for configuring adaptive applications. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, New York, New York, April 2000.

[18] Radu Litiu and Atul Prakash. Developing adaptive groupware applications using a mobile component framework. In *Proceedings of the ACM 2000 Conference on Computer Supported Cooperative Work (CSCW 2000)*, Philadelphia, Pennsylvania, December 2000.

[19] Lily B. Mummert, Maria R. Ebling, and Mahadev Satyanarayanan. Exploiting weak connectivity for mobile file access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, December 1995.

[20] Athicha Muthitacharoen, Benjie Chen, and David Mazieres. A low-bandwidth network file system. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, Banff, Canada, October 2001.

[21] Brian D. Noble, Mahadev Satyanarayanan, Dushyanth Narayanan, James Eric Tilton, Jason Flinn, and Kevin R. Walker. Agile application-aware adaptation for mobility. *Operating Systems Review (ACM)*, 51(5):276–287, December 1997.

[22] Francois Pacull, Alain Sandoz, and Andre Schiper. Duplex: A distributed collaborative editing environment in large scale. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, pages 165–173, Chapel Hill, North Carolina, October 1994.

[23] Gerald J. Popek, Richard G. Guy, Thomas W. Page, Jr., and John S. Heidemann. Replication in Ficus distributed file systems. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–25, Houston, Texas, November 1990.

[24] Dave Ragget, Arnaud Le Hors, and Ian Jacobs, editors. *HTML 4.01 Specifications*. December 1999. `http://www.w3.org/TR/html4/`.

[25] Peter Reiher, John Heidemann, David Ratner, Gregory Skenner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *Proceedings of the Summer USENIX Conference*, pages 183–195, Boston, Massachusetts, June 1994.

[26] Luigi Rizzo. DummyNet: a simple approach to the evaluation of network protocols. *ACM Computer Communication Review*, 27(1):13–41, January 1997.

[27] Martina Angela Sasse, Mark James Handley, and Shaw Cheng Chuang. Support for collaborative authoring via email: The MESSIE environment. In *Proceedings of 3rd European Conference on Computer Supported Cooperative Work*, pages 249–264, Milan, Italy, sep 1993.

[28] Mahadev Satyanarayanan, Jason Flinn, and Kevin R. Walker. Visual proxy: Exploiting OS customizations without application source code. *Operating Systems Review*, 33(3):14–18, July 1999.

[29] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, pages 172–183, Cooper Mountain, Colorado, December 1995.

[30] E. James Whitehead and Yaron Y. Goland. WebDAV: A network protocol for remote collaborative authoring on the Web. In *Proceedings of the Sixth European Conference on Computer Supported Cooperative Work (ECSCW'99)*, pages 291–310, Copenhagen, Denmark, September 1999.