Collaboration and Document Editing on Bandwidth-Limited Devices

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

[‡] Department of Electrical and Computer Engineering
[†] Department of Computer Science
Rice University

Abstract

This paper presents the design of CoFi, a novel architecture for supporting document editing and collaborative work over bandwidth-limited clients. CoFi combines the previously disjoint notions of consistency and fidelity in a unified architecture. CoFi enables bandwidth-limited clients to edit documents that are only partially present at the client (because parts of the documents were lossily transcoded, or only a portion of the document was fetched), and to propagate modifications incrementally by progressively increasing their fidelity.

1 Introduction

Research on mobile computing has made significant progress in supporting document browsing over bandwidth-limited devices [1, 2, 3]. Document editing and collaborative work on these platforms, however, remains an open problem.

We identify three factors that hinder document editing and collaborative work over bandwidth-limited devices:

1. The techniques used by adaptation systems to lower download latencies. These systems reduce network traffic by use of *subsetting* and *versioning*. In subsetting adaptations, only a subset of the components of the original document, for example the first page, is transferred. In versioning adaptations some of the components are transcoded into lower fidelity representations, for example low-resolution images. In either case, the documents present at the bandwidth-limited device are only partially loaded, and may be significantly different from the documents stored at the server. Naively storing user modifications to a partially loaded document may result in the deletion of components that were not included in the subset, or in the replacement of high-fidelity components with the lower-fidelity versions available at the bandwidth-limited device (even in cases where the user did not modify the transcoded components).
2. The potential for large updates. Users can produce large multimedia content (e.g., photographs, drawings, audio notes) that may incur long upload latencies over the bandwidth-limited link.
3. The possibility of update conflicts that result from the use of optimistic consistency models [4, 5], where clients modify their local copy of the document and propagate their modifications when they reconnect to the network.

Despite the above limitations, there is much to be gained from enabling users of bandwidth-limited devices to modify partially-loaded documents and to share (even low-fidelity versions of) their modifications with their peers. For example, a manager away on a conference, could make changes to a presentation for launching a new product by loading and editing just the text portions of the slides his employees are working on. Several engineers collaborating in inspecting a large facility could stay aware of each-other's progress by adding digital photographs of their findings to a shared report. Transcoded versions of these photographs could then be viewed by the other engineers.

This paper introduces CoFi, a unified architecture that combines the notions of **co**sistency and **fi**delity, and supports document editing and collaborative work on bandwidth-limited devices. CoFi supports editing partially-loaded documents by decomposing documents into their component structures (e.g., pages, images, paragraphs, sounds) and keeping track of changes made to each component by the user and those that result from adaptation. CoFi can then propagate just user modifications to components present with high-fidelity at the client, or when the data type allows it, merge (at the server) user modifications to low-fidelity components with the high-fidelity versions available at the server.

CoFi enables user of bandwidth-limited devices to share their modifications by using subsetting and versioning adaptations to support partial and incremental propagation of modifications. For example, under low bandwidth conditions, a user can choose to propagate only a portion of the modified components, or can transcode components and propagate lower-fidelity versions. Later, on reconnecting over higher-capacity links, the user can propagate the remaining components or upgrade the fidelity of components that were previously propagated to the server.

Two characteristics of CoFi reduce the likelihood of update conflicts. First, CoFi keeps track of modifications at the level of components, instead of the full documents, which reduces the size of the consistency unit. Secondly, the use of subsetting and versioning encourages clients to propagate modifications more frequently, increasing the awareness that users have about the modification being performed by other users and reducing the likelihood that two users will inadvertently modify the same component.

The rest of this paper is organized as follows. Section 2 presents the general design of CoFi. Section 3 discusses related work. Finally, Section 4 concludes the paper and discusses future plans for implementation.

2 CoFi

CoFi supports document editing and collaborative work over bandwidth-limited devices by decomposing documents into their components structures (e.g., pages, images, paragraphs, sounds), and keeping track of modifications to individual components with a model that incorporates optimistic consistency [4, 5] and fidelity. We based CoFi on an optimistic consistency model because we think this is the preferred mode of operation for bandwidth-limited devices. CoFi, however, can be easily adapted to operate with stronger consistency requirements, provided that the users are willing to pay the higher price of a more restrictive consistency model.

CoFi allows different versions of the same component, which we call *views*, to coexist in different parts of the system. Two views 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. CoFi supports two fidelity classes: *full* and *partial*. For a given creation time, a component can have only one full-fidelity view but many partial-fidelity views. We say that a component is present at full fidelity when its view contains data that is equal to the reference view (i.e., the original view) of the component for a given stage in the component's development. Conversely, we say that a component is present with partial fidelity if its view was lossily transcoded from the component's reference view. Fidelity is by nature a type-specific notion, and hence there can be a type-specific number of different partial-fidelity views. CoFi assumes, however, that all the views of a component can be arranged into a monotonically increasing order according to their fidelity, with the first view having the lowest-possible fidelity (maybe even an empty view), and the last being a full-fidelity view.

CoFi enables users of bandwidth-limited devices to reduce the latency for downloading components by loading partial-fidelity views. In a similar manner, users can reduce upload time by making available to other user partial-fidelity views of their updates.

In principle, CoFi does not assume a predetermined relationship between system nodes. CoFi nodes can be configured into client-server relationships or peer-to-peer groups. For ease of explanation, however, we assume for the remainder of this paper a system with a client-server configuration. In this configuration, documents are made persistent at the server (or servers in a replicated implementation). Clients can cache a subset of a document's component, or even the full document, but it is assumed that all client modifications will be eventually propagated to the server.

Servers operate with a simple consistency model. A server always has a consistent view of the components it serves, and can serve both full- and partial-fidelity views of components. Servers accept new views only if they are more recent than the current server view or they represent fidelity refinements. Clients by default fetch the most recent and highest-fidelity view available for a component. Client can, however, request older views (usually for conflict resolution), or request partial-fidelity views by specifying transcoding transformation for the component data.

The following discussions focus on the states of a single component on a client node. We first consider a model that includes a traditional implementation of optimistic consistency. We then extend this model to incorporate fidelity. While CoFi allows clients to cache multiple views of a component, the following discussions relate to the most recent and highest-fidelity view of the component.

2.1 Optimistic Consistency

The dark ovals in Figure 1 show the state transition diagram for a component in a client that supports an optimistic consistency model, but has no notion of fidelity. A component can be in one of five states: *Empty, Clean, Dirty, Obsolete*, or *Conflict*. Transitions between states are marked with dark arrows and occur by replacing the current view with a more recent one (*Replace*), modifying the current view (*Write*), pushing modifications to the server (*Push*), learning about the existence of a more recent view at the server (*New View*), or resolving a conflict (*Client Resolve* and *Server Resolve*).

New components are initially placed in the *Empty* state, which reflects that the client is aware of the existence of the component but does not yet have a view for it. Component created by the client, are then moved to *Dirty* to reflect that the client has data that needs to be propagated to the server. Components that exist initially only at the server transition to *Clean* after fetching the most recent view of the component. If the client modifies a component, its state moves to *Dirty*. The component goes back to *Clean* by pushing its modifications back to the server. If the client learns that the server has a newer view of

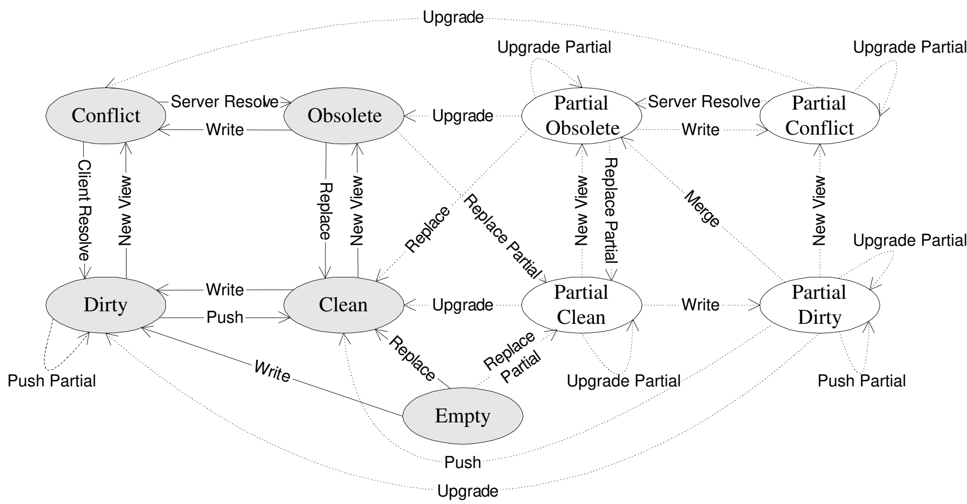

Figure 1: CoFi state transition diagram.

the component (created by some other client), the component transitions to *Obsolete* when the component was not changed at the client, and to *Conflict* when the client edited the component. A component in *Obsolete* transitions to *Clean* by fetching the new view available at the server. A component in *Obsolete* moves to *Conflict* if the client modifies it. To resolve conflicts, the two conflicting views need to be merged. Depending on the implementation, merging can be done at the client or at the server. When the merging happens on the client, the component moves to *Dirty* to reflect that the new view needs to be propagated to the server. When merging happens at the server, the component moves to *Obsolete* to reflect that the server has a more recent view than the client.

2.2 Consistency and Fidelity

We extend our model to account for fidelity by adding four new states to the diagram of Figure 1. We assume that the states described in the last section reflect a component with a full-fidelity view, and that the new states represent a component with a partial-fidelity view. We denote partial-fidelity states by pre-pending the word *Partial* to the state's name. In this manner a component with a partial-fidelity view can be in one of four states: *Partial Clean, Partial Dirty, Partial Obsolete*, and *Partial Conflict*. The new states have similar meaning to the states introduced in the previous section. For example, both the *Dirty* and *Partial Dirty* states reflect that a component has data that needs to be propagated to the server. The states differ in that in the former, the user modified a full-fidelity view, while in the latter she modified a partial-fidelity view.

We extend the model with five new transitions: *Replace Partial, Upgrade, Upgrade Partial, Push Partial*, and *Merge*. We denote these transitions in Figure 1 with dotted lines. A component transitions into a partial-fidelity state by fetching a partial-fidelity version of the most recent view (i.e., Replace Partial). For example, a component in *Obsolete* transitions to *Partial Clean* by fetching an up to date partial-fidelity view. A component transitions from a partial-fidelity state to a full-fidelity state by one of three ways: It can upgrade its view with a full-fidelity refinement (i.e., *Upgrade*); or replace its view with a more-recent full-fidelity view (i.e., *Replace*); or it can overwrite the server view with its modifications to a partial-fidelity view (i.e., *Push*), effectively making the client's view the up to date full-fidelity view. For example, a component in *Partial Clean* transitions to *Clean* by fetching refinements to upgrade its view to full fidelity. A component in a partial-fidelity state can improve the fidelity of its view (without reaching full fidelity) by fetching partial-fidelity improvements (i.e., Upgrade Partial). For example, a user modifying a partial-fidelity image (component is *Partial Dirty*) can fetch fidelity refinements and merge them with his modifications. This, of course, assumes that the application can safely merge the user modifications with the refinement improvements. Finally, CoFi enables pushing partial-fidelity views of modified components. For example, a user can push a partial-fidelity view of a digital photograph acquired at the client (i.e., component in *Dirty* state). The component will remain in *Dirty* until a full fidelity view is pushed to the server (or we detect a conflict).

Partial Dirty is a particularly interesting state. While the view fetched form the server is partial fidelity, the modification made by the client are full fidelity. CoFi supports three methods for propagating these modifications. First, the client can set its view as the up to date full-fidelity view, replacing the server view (i.e., Push), and transitioning to *Clean*. Second, the client can ask the server to merge the full-fidelity modifications with the server full-fidelity view (i.e., *Merge*). A *Merge* creates a new view on the server, and the client transitions to *Par-*

tial Obsolete to reflect that a more-recent view exists on the server. Third, the client can propagate a partial-fidelity view of the modifications (i.e., *Push Partial*).

CoFi supports conflict resolution for components in partial-fidelity states at both the client and the server. Resolving conflicts on the client requires fetching full-fidelity versions of the conflicting views, which has the effect of transitioning from *Partial Conflict* to *Conflict*. Conflict resolution then occurs as described in Section 2.1 and the component transitions to *Dirty*. For server-based resolution, a full-fidelity version of the modifications is transfered to the server, where it is merged with the conflicting view. The component in the client transitions to *Partial Obsolete* to reflect that the server has a more recent view.

2.3 Implementation Considerations

The CoFi architecture presented in the previous section allows for several possible implementations. One possibility is to write or modify applications to implement CoFi natively. An alternative is to implement CoFi in an adaptation system. We believe that the later is a more profitable proposition. Most parts of the CoFi architecture are bound to be common for most applications. By implementing CoFi as part of the general adaptation infrastructure we get to leverage the coding effort across a wider set of applications.

CoFi does not assume a predetermined method for propagating modifications between nodes and can support implementations based on data shipping, operation shipping, or a combination of data and operation shipping.

3 Related Work

While several adaptation systems [1, 2, 3] use subsetting and versioning to reduce document download time, to the best of our knowledge, CoFi is the first to provide adaptation support for document editing and collaborative work over bandwidth-limited devices.

Coda [6], Ficus [7], and Bayou [8] provide support for document editing on disconnected devices. CoFi differs from these previous efforts in that it assumes that modifications propagation and conflict resolution can occur over bandwidth-limited connections and do not have to wait for the device to be strongly connected.

Several efforts on collaborative applications (e.g., Alliance [9] and Duplex [10]) have used the documents component structure to reduce conflicts and limit the amount of data that need to be present at the device. These efforts, however, do not allow the propagation of low fidelity versions of modifications. MASSIVE-3 [11] 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.

4 Conclusions and Future Work

We described CoFi, a novel architecture for supporting document editing and collaborative work on bandwidth-limited devices. CoFi supports editing partially-loaded documents by decomposing documents into their components structures (e.g., pages, images, paragraphs, sounds) and keeping track of changes made by the user and those that result from adaptation. CoFi enables users of bandwidth-limited devices to share their modifications by using subsetting and versioning adaptations to support partial and incremental propagation of modifications.

In future work, we plan to implement CoFi in the Puppeteer adaptation system [1].

References

[1] E. de Lara, D. S. Wallach, and W. Zwaenepoel, “Puppeteer: Component-based adaptation for mobile computing,” in *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, (San Francisco, California), Mar. 2001.

[2] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir, “Adapting to network and client variability via on-demand dynamic distillation,” *SIGPLAN Notices*, vol. 31, pp. 160–170, Sept. 1996.

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

[4] J. J. Kistler and M. Satyanarayanan, “Disconnected operation in the Coda file system,” *ACM Transactions on Computer Systems*, vol. 10, pp. 3–25, Feb. 1992.

[5] G. J. Popek, R. G. Guy, T. W. Page, Jr., and J. S. Heidemann, “Replication in Ficus distributed file systems,” in *Proceedings of the Workshop on Management of Replicated Data*, (Houston, Texas), pp. 20–25, Nov. 1990.

[6] P. Kumar and M. Satyanarayanan, “Flexible and safe resolution of file conflicts,” in *Proceedings of the USENIX Winter 1995 Technical Conference*, (New Orleans, Louisiana), Jan. 1995.

[7] P. Reiher, J. Heidemann, D. Ratner, G. Skenner, and G. Popek, “Resolving file conflicts in the Ficus file system,” in *Proceedings of the Summer USENIX Conference*, (Boston, Massachusetts), pp. 183–195, June 1994.

[8] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing update conflicts in Bayou, a weakly connected replicated storage system,” in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles*, (Cooper Mountain, Colorado), pp. 172–183, Dec. 1995.

[9] D. Decouchant, V. Quint, and M. R. Salcedo, “Structured cooperative authoring on the World Wide Web,” in *Proceedings of the Fourth International World Wide Web Conference*, (Boston, Massachusetts), Dec. 1995.

[10] F. Pacull, A. Sandoz, and A. Schiper, “Duplex: A distributed collaborative editing environment in large scale,” in *Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW)*, (Chapel Hill, North Carolina), pp. 165–173, Oct. 1994.

[11] C. Greenhalgh, J. Purbrick, and D. 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.