Mimic: Raw Activity Shipping for File Synchronization in Mobile File Systems *

Tae-Young Chang, Aravind Velayutham, and Raghupathy Sivakumar
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332, USA
{key4078,vel,siva}@ece.gatech.edu

ABSTRACT

In this paper, we consider the problem of file synchronization when a mobile host shares files with a backbone file server in a network file system. Several *diff* schemes have been proposed to improve upon the transfer overheads of conventional file synchronization approaches which use full file transfer. These schemes compute the binary *diff* of the new file with respect to the old copy at the server and transfer the computed *diff* to the server for file-synchronization. However, Lee et al. have shown that the performance of *diff* can be significantly improved upon by shipping user operations as opposed to the data itself. Using this as motivation, we present a purely application-unaware approach called *Mimic* that relies on transferring raw user activity to the server for file synchronization. Through a simple prototype of the proposed approach, we show that Mimic can outperform *diff* under many common conditions. We also identify conditions under which *diff* based approaches do perform better than the proposed approach, but show that detection of such conditions is straightforward, thus enabling both schemes to be used in tandem with a mobile file system for bandwidth-efficient file synchronization.

Categories and Subject Descriptors

D.4.3 [**Operating Systems**]: File Systems Management—*Distributed file systems*

General Terms

Design, Experimentation, Performances

Keywords

Mobile File System, File Synchronization, Raw Activity Shipping

*This work is sponsored by the Georgia Electronic Design Center (GEDC).

MobiSys'04, June 6–9, 2004, Boston, Massachusetts, USA.

1. INTRODUCTION

The bandwidth usage efficiency of the file synchronization scheme in a distributed file system is important when the clients are connected to the file server through weakly connected, low-bandwidth links, such as in a wireless environment. An intuitive file synchronization strategy for such environments is one where only the *diff* or the differences between the original and updated files are sent across to the server. The original file at the server is thus patched with the differences. The performance of such an approach is obviously better than that of a *full file transfer*. The *low-bandwidth file system* (LBFS) is an example of a file system that uses such a strategy [8].

However, in [5], the authors show that the performance of *diff* can be improved upon considerably by adopting an *operation shipping* strategy. Briefly, for any file created or updated through the use of user-level shell commands, the commands, and not the *diff*, are sent across to the server. The server then re-executes the commands at its end to regenerate the corresponding updated files. The authors demonstrate that the transfer size for the commands is typically much smaller than that of the *diff*s, and hence motivate a new paradigm for file synchronization.

In this work, we present a similar strategy for more common interactive desktop applications. The target class of applications includes applications such as the Microsoft Office suite, Visio, Acrobat, xfig, AutoCAD, xv, etc. While the authors in [5] do identify the need for an operation shipping approach for interactive applications, and present a solution for the same, the solution is application-aware and requires changes to applications. In contrast, the strategy that we explore in this paper is fully application-unaware, and relies on *raw user-activity shipping* as opposed to activities with any higher level semantics.

The approach we present *records* raw user activity such as keyboard and mouse inputs at the client, maintains the records as long as the client is disconnected from the server, ships the records to the server during synchronization, and *replays* the activities at the server. We refer to this approach as *Mimic* since the strategy is to *mimic* the client-side user activity on the server. Mimic is not a *complete* file system, but is meant to be an add-on to an underlying file system such as LBFS or Coda [11], to reduce transfer sizes during file synchronization. We use a simple prototype of the Mimic strategy to show the considerable bandwidth usage benefits it can provide when used appropriately. We also identify conditions under which Mimic will not provide better performance than *diff*, and hence present an integration strategy that involves both Mimic and *diff* working in a loosely coupled fashion.

Thus, the contributions of this work are twofold:

- We motivate through arguments and performance results the need for an activity shipping based file synchronization strategy for file updates performed using common interactive applications.
- We present an application-unaware strategy called Mimic that, when loosely coupled with the file system, provides considerable performance benefits in terms of file synchronization bandwidth usage.

The rest of the paper is organized as follows: In Section 2, we summarize the approach proposed in [5] briefly, and outline the motivation for Mimic in the context of interactive applications. In Section 3, we present a high level overview of Mimic, and the envisioned file system integration strategy. In Section 4, we describe the key details of the Mimic approach. In Section 5, we evaluate Mimic's performance against that of *diff*. In Section 6, we discuss some issues with the Mimic design and scope. Finally, in Section 7, we discuss related works, and in Section 8 we conclude the paper.

2. MOTIVATION

In this section, we outline the motivation for performing activity shipping for interactive applications, and define the scope and goals of this work. However, before we delve into the motivation, we provide a brief summary of the *operation shipping* strategy presented in [5].

Note that the broad motivation for performing activity shipping, as opposed to *diff* shipping, was first identified in [5]. The authors in [5] refer to *diff* shipping as *delta shipping* or *value shipping*. Moreover, the high level design of Mimic in terms of the core components is also similar to the design presented in [5]. Through the summary we highlight both the similarities and the differences between operation shipping and Mimic.

2.1 Operation Shipping

The authors in [5] present an application-unaware operation shipping approach for non-interactive applications. The approach consists of four components: (i) logging of user operations, (ii) shipping of user operations to a surrogate on the server side, (iii) re-execution of the user operations at the surrogate to regenerate the updated file, and (iv) verify whether the regenerated file at the server is identical to the updated file at the client.

The logging is performed by appropriately instrumenting the *bash* shell [2] to notify the file system both before and after the processing of the user operations. The logs such as "tar -cvf update.tar", are then sent across to the server side surrogate for replay during synchronization. The surrogate replays the logs, and it is assumed that both the software and hardware environment of the surrogate and the client in terms of operating system, header files, and system libraries, and other system environment variables, are identical. Once the replay is performed, the regenerated file is verified through a *fingerprinting* technique [6] to ensure that it is identical to the updated file at the client. If the first verification fails, the server performs *forward error correction* (FEC) to correct non-repeatable actions [9], [12]. Nevertheless if the second verification also fails, the client is informed accordingly, and *diff* is used to synchronize the file.

The authors also present an operation shipping strategy for interactive applications, but the strategy is application-aware and requires changes to the application implementation to log operations.

In summary, the motivation for Mimic is identical to the motivation for operation shipping identified in [5]: *to exploit conditions under which the encoded user activity for a file update is much smaller in size than the changes to the file itself.* However, the application-unaware strategy presented in [5] applies only to non-interactive applications, and still requires changes to meta-applications such as the *bash* shell. In contrast, Mimic is fully application-unaware, does not require any changes other than to the file system itself, and applies to interactive applications. It is quite reasonable to consider operation shipping and Mimic as co-existing approaches in a file system providing support for both non-interactive and interactive applications. Mimic achieves its applications unaware property by shipping raw activity such as keystrokes and mouse-clicks as opposed to application-aware cases.

2.2 Motivation

While we have provided a high level motivation for activity shipping earlier in the section, we now present the key factors that contribute to the motivation in the specific context of interactive applications. The identification of the factors serves to highlight the commonality of the types of file updates for which Mimic will deliver better performance than *diff*.

We classify file updates for which activity shipping will deliver better performance into two types: (i) updates where small amounts of user-activity results in large amounts of content being *added* to the file; and (ii) updates where small amounts of the user-activity in large *changes* to the content of the file without necessarily increasing the size of the file itself. In the rest of the section, we use experimental results with Microsoft Word documents to highlight the inefficacy of *diff* for the above classes of updates, and therein motivate Mimic.

Activity description	Activity size	File size [bytes]	*diff* size [bytes]	Activity record size [bytes]
Insert a line	98 keystrokes	29341	1543	236
Insert a paragraph	476 keystrokes	29356	2111	848
Copy and paste a paragraph from the same file	6 keystrokes + 12 mouse-clicks	33449	1119	72
Change the font type of a paragraph	7 mouse-clicks	40611	1660	30

Figure 1: Comparison between *diff* and activity record size

- The first class of updates is relatively straightforward to understand. An example of an update that results in a large amount of content being added to files with minimal user-activity is a *copy and paste* operation. In Figure 1, the activity "Copy and paste a paragraph from the same file" thus represents this class of updates. A paragraph in a 33 KB Word document is copied, and pasted onto another location in the document using keystrokes mouse operations. The *diff* between the updated and original files amounts to approximately 1 KB. However, the activity itself, in terms of mouse-clicks and locations, when encoded amounts to only 72 bytes. We elaborate on a simple encoding of raw user activity in Section 4.
- The second class of updates is relatively more non-intuitive, particularly in the context of interactive applications. In Figure 1, for the activity "Insert a line" by keystrokes, a single line of text is inserted into a 29 KB Word document. The additional textual content added is approximately 80 characters. However, the *diff* amounts to about 1.5 KB of data. This phenomenon is also observed for the activity "Insert a paragraph" by keystrokes, where a 5-line paragraph is typed

into the Word document. While the textual content added is approximately 400 characters, the *diff* amounts to approximately 2 KB. The interesting observation to be made in the above results is that while the file size or textual content itself does not change dramatically, there is considerable change in the binary representation of the file despite the minimal user-activity.

The reasons behind the observation is the complex file structures used by most interactive applications to provide the best interface and services to the user. In the specific case of Microsoft Word, the application maintains each document in the form of *six* different streams [7]. The *main* stream contains the document header and all textual information in the document, while the *data* stream contains information about all non-OLE objects (such as figures and tables) in the document, including their physical addresses in the binary representation of the file. The *object* stream similarly maintains information about all OLE objects (such as imported figures and spreadsheets). A *table* stream maintains the structure of the document itself, including the locations of the different objects in the visual representation of the document, and the relative dependencies between the objects (e.g. a figure and wrap-around text). Finally, two *summary* information streams maintain other information about the document including timestamps.

In this complex representation of a Word document, a single line of text insertion thus has the following impact: The main stream is updated to contain the modified textual information. In addition, the data and object streams are updated to reflect the new addresses of the different objects in the binary document file, with their positions altered due to the insertion of the line of text in the main stream. Moreover, the table stream also is updated to reflect all changes to the visual representation of the document in terms of the relative and absolute locations of the different objects in the document. Finally, any resulting changes in other attributes of the file including timestamps results in updates to the summary streams. Such a multi-fold update of the document file, when a single line of text is inserted, accounts for the substantial size in the *diff* despite the minimal user-activity. Note that such complex, and application tailored, binary representations of content is not specific to only Word, and is true for most interactive applications today. In Section 5, we consider other applications.

Finally, in Figure 1, for the activity "Change the font type of a paragraph", a paragraph is highlighted, and a font change is effected through only mouse operations. It can be observed that for such "meta" operations, the size of the activity itself is much smaller than that of the *diff*. The reasons are again similar to those described earlier in terms of updates to multiple streams within the file structure.

In summary, updates for which small magnitudes of user-activity result in large additions or changes to the file's content render activity shipping an attractive option for file synchronization. In the rest of the section, we outline the key goals of the Mimic approach for file synchronization.

2.3 Scope and Goals

In the next two sections, we present Mimic, a raw-activity shipping strategy for file synchronization. Mimic is specifically targeted toward reducing transfer sizes for files updated using interactive applications. The goal of the Mimic design is to achieve such reduction in transfer sizes while being fully application-unaware, and requiring no changes to applications. We assume that the hardware and software configurations of the client, and the surrogate on the server end that will replay the user-activities, are identical. For ease of presentation, we assume that the surrogate at the server end is colocated with the server. While this can be a potential security concern because of client activities being executed directly on the server, one solution is to decouple the surrogate from the server just as in [5]. However, we do not consider such decoupling in this paper. Also, we assume that Mimic can be interfaced with the underlying file system at the client end.

3. MIMIC OVERVIEW

In this section, we first present a high level overview of Mimic, and then describe how it loosely integrates with the underlying file system. We defer details of the Mimic approach to Section 4.

Mimic is a fully application-unaware strategy that consists of components at the client and the server respectively. At a high level, Mimic captures raw user-activity at the client that pertains to shared files, maintains such activity on a per-file basis, and ships the raw-activity to the server during file synchronization. The server then replays the activities to regenerate the updated files at the client. The realization of the above functionalities in Mimic are done with the goals of reducing the transfer file size, minimizing latencies involved, and incurring minimal overheads in terms of usage of system resources.

Mimic requires interfacing with both the underlying file system and window manager at the client, and with the window manager at the server end. The interfacing with the window manager, however, does not require any changes to the operating system, and instead is achievable through standard interfaces that most window managers export.

Briefly, the primary components of the Mimic approach are:

1. *Record:* This component is responsible for the effective capturing of raw-activity at the client end, classifying the activity, and maintaining per-shared-file records.

2. *Replay:* This component is responsible for replaying the activity records in the fastest manner possible while ensuring correctness.

3. *Verification:* Finally, this component is responsible for verifying whether the replay based re-creation of an updated file is correct. This component includes both forward error correction to correct non-repeatable actions (such as timestamps), and detecting any errors that arise due to improper playback. Since Mimic uses a verification scheme [9] that is identical to the one presented in [5], we do not elaborate on this component of Mimic any further. Interested readers are referred to [5].

While we elaborate on the realization of the *record* and *replay* mechanisms in detail in Section 4, in the rest of the section we present details of the assumed integration of Mimic with the underlying file system. For purposes of this discussion, we assume that the underlying file system is Coda, which is in turn equipped with a *diff* based synchronization strategy like the one presented in [8]. In the rest of the paper, we refer to the above file system as simply Coda.

3.1 Integration with File System

We now describe the loose coupling that Mimic requires with the underlying file system at the client. Mimic does not require any interfacing with the file system at the server. We refer to the coupling

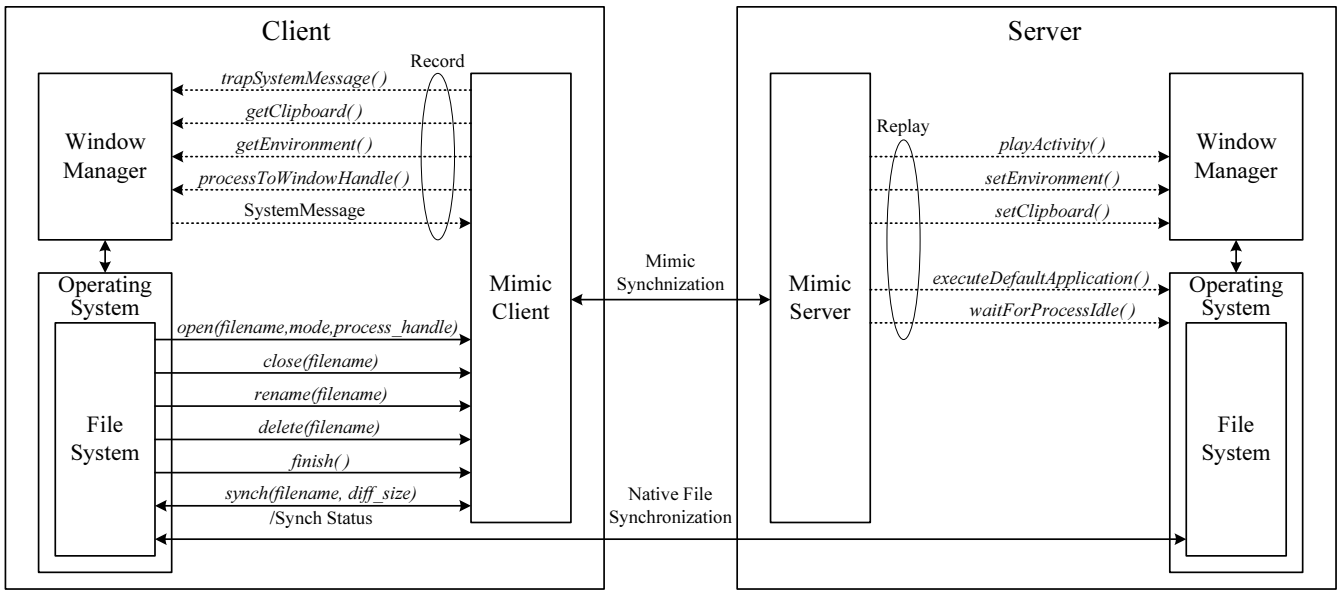


Figure 2: Mimic-file system-window manager interfaces

as loose because Mimic currently relies only on *informative* callbacks from the file system that is essential for its operations, and requires minimal changes to the file system design and logic. In Section 6, we discuss how Mimic can be tightly coupled with the file system for more effective performance.

The interface between Mimic and the underlying file system consists of six function calls (*open( )*, *close( )*, *rename( )*, *delete( )*, *finish( )*, and *synch( )*), all exported by Mimic, and invoked by the file system. The first five functions are *informative* in nature, and require no logic change inside the file system other than their mere invocation. The sixth function is used by the file system to preferentially use Mimic for the synchronization process, but falls back to its native synchronization mechanism if Mimic indicates a failure in its synchronization attempt. We now present the interface in two stages, based upon which aspect of Mimic the functions are useful for.

- The first set of functions which are part of the interface help in the recording component of Mimic, and consist of *open( )*, *close( )*, *rename( )*, and *delete( )*. All four functions are purely informative in nature, and are used by the file system to *inform* Mimic about the corresponding actions. The *open( )* function is used by the file system to inform Mimic about the opening of a *shared* file, and its parameters consist of the filename, opening-mode (read, write, or append), and the process-handle for the process that is performing the open. Mimic uses this information to initialize its recording activity pertaining to that file. In Section 4, we explain the filtering process that enables Mimic to not maintain records for files not being updated interactively, opened in read-only mode, etc. The *close( )*, *rename( )*, and *delete( )* functions all consist of the filename as the parameter (*rename( )* has both the old and new file names), and are used by Mimic to update the activity records corresponding to the file being closed, renamed, or deleted.

- The second set of functions that Mimic requires the file system to use consists of the *finish( )* and *synch( )* functions, which are used during the actual file system initiated synchronization process. For every file that needs to be synchronized, the file system preferentially calls the *synch( )* function with the filename, and the *diff* size as parameters. The return value for the *synch( )* call indicates to the file system whether or not Mimic's synchronization process was a success. Only in the event of a failure does the file system initiate its native synchronization process. We elaborate on the specific conditions under which Mimic will return an error in Section 4. Finally, the *finish( )* function is used by the file system to indicate to Mimic the termination of the synchronization process, after which Mimic performs simple clean-up operations.

Due to the purely informative nature of five of the functions (except *synch( )*), no changes are required in the file system logic when the functions are called. The functions are merely invoked by the file system when it is performing an open, close, rename, delete, and completion of synchronization respectively. The only change in the logic required when the *synch( )* function is invoked is a check for the return value of the function call, and conditionally invoking the native synchronization process.

4. MIMIC APPROACH

In this section, we present the details of Mimic mechanisms at the client and server respectively. Broadly, the *recording* and *shipping* tasks are performed at the client, while the *replaying* and *verification* tasks are performed at the server. We explain Mimic's interfacing with the window manager through *generic function names*. Interested readers can refer to [13] for a mapping of the generic function names to the corresponding native function names in Microsoft Windows and X Window.

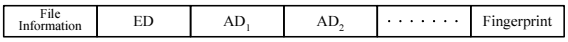
4.1 Mimic Client

4.1.1 Data Structures

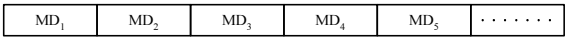
Mimic maintains three key data structures at the client:

- The *window-handle table* is used to maintain the mapping between window-handles and filenames. Multiple window-

handles could potentially map onto a single filename due to the complex windowing structures used by interactive applications today. For example, a single Microsoft Word document window is actually comprised of several windows, including the main text window and other menu related windows. The window-handle table is populated during the initialization process we describe next, and is used to map raw input activity to the appropriate files.



(a) FAR structure



(b) MAR structure

Figure 3: FAR and MAR structures

- The *file activity records* (FARs) are used to physically record the raw user activity. Each shared file that the user updates has its own FAR. The structure for a FAR is shown in Figure 3. It consists of the file information including the name and size of the file, followed by an *environment descriptor* (ED), and a sequence of *activity descriptors* (ADs). The ED captures the initial state of the system environment in terms of the keyboard layout, screen resolution, and color depth. Figure 4(c) illustrates the structure of the ED. Each AD consists of the type of activity, and the value of the activity. Figure 4(b) presents the different types of ADs.

- Finally, the *meta activity record* (MAR) is used to record any system configuration related activities. Specifically, when the user performs activities that change the system configuration such as the keyboard layout and screen resolution, such activities are recorded in the the MAR through meta descriptors (MDs). Note that any meta activity performed can also be realized through adding EDs to all FARs. However, the MAR allows for a more effective way of registering known meta activity, as only the concerned environment variable state needs to be recorded.

4.1.2 Initialization

When a file is opened through a file system call, the file system invokes the *open( )* function. When Mimic receives the call, and finds the file to have been opened in the *write* or *update* modes, it maps the process handle to the corresponding set of window handles through a *processToWindowHandles( )* call to the window manager. It then registers the returned window handles in the window-handle table with the corresponding file name.

If Mimic is unable to retrieve the window handles because the file is not being processed with an interactive application, Mimic simply ignores the *open( )* call. This *implicitly* ensures that Mimic does not handle such files. Note that any updates to such files will be handled by the native file synchronization strategy of the underlying file system.

Once the window-handle table is populated, Mimic also creates the FAR for the file if the FAR does not exist. If the FAR exists but is currently closed, it is re-opened. If the corresponding FAR is already opened, Mimic infers an error and aborts maintenance of the FAR for the file. Once the FAR is opened, an ED is created by fetching the state of the environment variables through a *getEnvironment( )* call to the window manager. This concludes the initialization phase for the file in Mimic.

4.1.3 Recording

Whenever user activity is performed, the activity is encoded in the form of system messages, and inserted into the system message queue. Mimic registers with the window manager through the *trapSystemMessage( )* interface to receive all dequeued messages from the queue. By default, without the trapping of the messages, the messages will be dequeued by the window manager, demultiplexed based on the window handle, and queued onto a thread-queue that is maintained on a per-window basis. An example format of the input system message called EVENTMSG in Windows operating systems is presented in Figure 4(a) [7].

When Mimic intercepts system messages, it passes the message onto the *message monitor* or the *environment monitor* based on whether the message corresponds to a specific file, or is a meta level system configuration message. Note that every system message is tagged with this information in the Windows operating system. However, this is not mandatory for the operational correctness of Mimic since all system environment messages will arrive with a window handle of *zero*, and hence can be detected accordingly.

When the message monitor receives a message, it looks up the corresponding filename in the window-handle table, and if successful in the lookup appends the message in the format of an AD into the corresponding FAR as shown in Figure 4(b). If the window-handle is not found in the table, Mimic skips the record phase, and directly queues the message onto the corresponding thread-queue. When the environment monitor receives a message, it appends the message to the MAR in the format of an MD as shown in Figure 4(c). In addition, a pointer to the MD is appended to every open FAR.

Finally, a special type of message that needs to be handled uniquely in Mimic is a *paste* message, as the message requires to be capture along with the corresponding *clipboard* information. The environment monitor thus, upon detecting a system paste message, obtains the clipboard information through the *getClipBoard( )* window manager function call, and forwards the content to the message monitor, which then appends it to the FAR. Although Mimic handles clipboard-based paste operations in the above manner, in Section 5, we show that Mimic is efficient only for those paste operations for which the content is copied or cut from the same file, and thus the clipboard content is not essential.

4.1.4 Bookkeeping

The *rename( )*, *close( )*, *delete( )*, *finish( )* interface function calls invoked by the file system triggers appropriate bookkeeping operations in Mimic. For the *close( )* and *delete( )* calls, the actions involve closing and deleting the corresponding FARs respectively. The *rename( )* function call, however, is handled differently. If the rename is for a file created since the last synchronization, Mimic renames the filename in the corresponding FAR. However, if the rename is for an already existing file, Mimic disables the FAR maintenance for the file, and allows the native file synchronization process to handle the file. Note that the FAR is a record of the *incremental* activity performed since the file was opened. Hence, if a FAR is maintained for a file f_a, and f_a is later renamed to f_b, the server *cannot* recreate the updated f_b by replaying the FAR for f_a on the original copy of f_b.

Finally, when the *finish( )* call is received from the file system signifying the end of a synchronization session, Mimic cleans up its data structures by deleting all the FARs and the MAR, and clearing up its window-handle table.

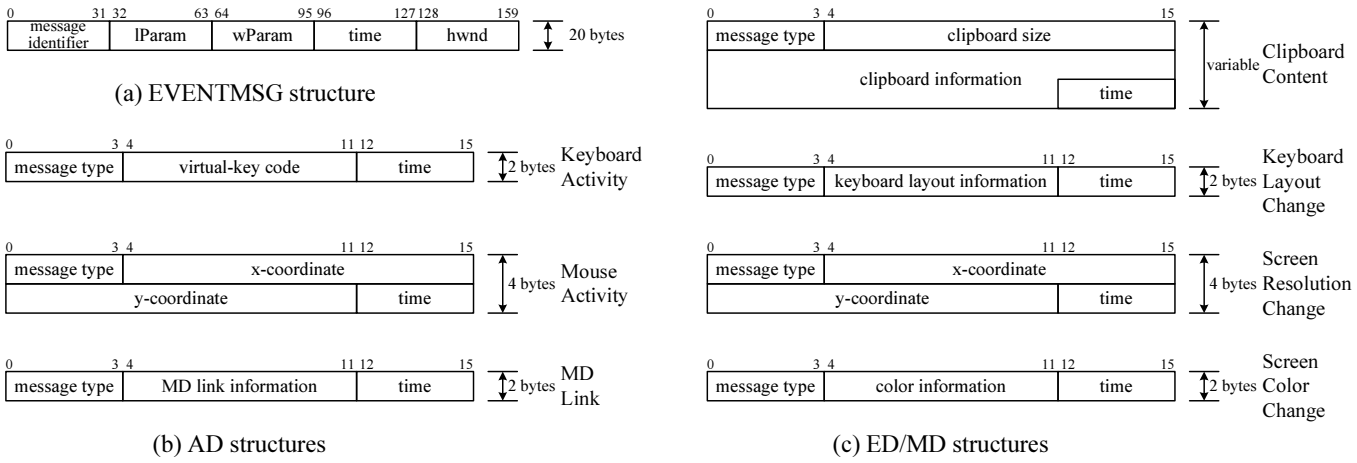


Figure 4: EVENTMSG and descriptor structures

4.1.5 Shipping

When Mimic receives a *synch( )* call from the file system, it goes through a two stage check to see if it can proceed with the synchronization. It first checks to see if a FAR has been maintained for the file. It then checks to see if the size of the FAR is less than that of the *diff* the file system will have to send as part of the native synchronization process. If the answer to either of the two checks is negative, Mimic returns a *SYNCH_FAIL* message back to the file system. The file system, as outlined in Section 4, will then proceed with its native synchronization strategy.

If both checks are successful, Mimic ships the corresponding FAR and MAR to the server. Note that the MAR needs to be shipped exactly once for a synchronization session. When Mimic attempts the synchronization, it forwards the result of the server side verification process as the return value for the *synch( )* call.

4.2 Mimic Server

Each replaying at the Mimic server requires the FAR corresponding to the file being synchronized, and the MAR since the last synchronization.

4.2.1 Initialization

During the initialization phase of the playback, Mimic uses the ED at the head of the FAR to set the environment variables at the server. Once the environment variable is set, Mimic invokes the interactive application corresponding to the file through the *executeDefaultApplication( )* interface to the operating system.

4.2.2 Replaying

Once the initialization process is complete, the system focus is then shifted to the application's window, the ADs read one record at a time, and played through the *playActivity( )* interface of the window manager. Recall that a single FAR can have multiple instances of EDs interspersed between multiple ADs. Such a phenomenon will occur when a file is opened-updated-closed multiple times between two synchronization sessions. For every open, Mimic's record component would have appended a new ED to the FAR. When the replay process thus encounters EDs in the FAR, it processes it just like the first ED and appropriately resets the environment variables.

Also, when an AD is a pointer to an MD in the MAR, the corresponding MD in MAR is looked up, and the corresponding environment variable is set appropriately. Thus, the time-wise integration of the FAR and MAR activities happens implicitly through the explicit pointers from the FAR to the appropriate entries in the MAR.

A critical issue with the Mimic synchronization process is the replay time at the server. At worst, the replay time for a file will be equal to the real time taken by the user to update the file at the client. Obviously, this is undesirable. At the same time, playing the records too fast can result in the *skipping* or *mis-interpretation* of user activity in the replay process. Specifically, problems may arise when the application is fed with inputs at a rate greater than the rate at which the actual user inputs were performed. This can be explained as follows. Inputs to an application can be thought of to change the *state* of the application. Also, certain inputs might be relevant to the application only for particular states, or might be interpreted differently for different states. Thus, when the replay rate is faster than the actual user input rate, inputs might be either ignored by the application, or might be wrongly interpreted. Consider an example with the following sequence of user-activities: (i) user *right-clicks* mouse in a Word document, (ii) *moves* pointer mouse to "Paste", and (iii) *left-clicks* on "Paste". The *correct* application state for the latter two inputs are: "application with the appropriate pop-up window open", and "application with the pop-up window open, and mouse focus on Paste", respectively. If the replay is performed faster than the rate of change of application state, the latter two inputs might be delivered to the application *before* the pop-up window is opened, in which case the correct activities will not be executed.

Thus, the challenge is to replay the records as fast as possible without introducing errors due to activity skipping, or misinterpretation. Mimic addresses this challenge by explicitly monitoring the process CPU utilization after every message playback through the *waitForProcessIdle( )* interface to the operating system. Only when the relevant process is idle does Mimic playback the next AD. In Section 5 we show how Mimic's latency is thus quite reasonable given the bandwidth usage benefits.

4.2.3 Verification

After all ADs in the FAR are played back at the server, the corresponding application is terminated, and the verification process is begun. The verification process, as mentioned in Section 4, is identical to the three phase - file size check, forward error correction, and fingerprinting - verification process presented in [5], and hence we do not delve into it any further. If the verification pro-

cess fails, the Mimic server returns a *SYNCH_FAIL* message to the Mimic client, which then propagates it to the local file system.

5. PERFORMANCE EVALUATION

In this section, we use a simple prototype of Mimic on Microsoft Windows platform to evaluate its performance against that of *diff*. We primarily focus on the transfer size overheads and latency as the metrics for the comparison.

5.1 Experimental Setup

- *Network:* We consider wireless wide area networks (WWANs) as a representative of weakly connected networks in our experiments. The client is connected to the network through a CDMA2000-1X cellular network. The measured effective data rate on the WWAN interfaces is about 17 Kbps, and the round-trip time between the client and the server is about 300ms.
- *Hosts:* The mobile client used in the experiments is a HP Pavilion N5430 laptop computer with a 850 MHz AMD Duron CPU, 128 MB RAM, and a Sprint PCS Merlin C210 WWAN network interface card. The server is a Dell Dimension 4400 desktop computer with a 1.6 GHz Intel Pentium IV CPU, 256 MB RAM, and a 3COM 10/100 Mbps 3C905CX-TXM NIC, and it runs Windows 2000 Advanced Server operating system. Both the client and the server are equipped with Microsoft Office 2000.

Application Index	Content	Feature	Application
W1	text-based	keyboard intensive	Word
W2	text + other	keyboard and mouse	PowerPoint
W3	structured text	keyboard and mouse	Excel
W4	graphics	mouse intensive	Visio

Figure 5: Application index for experiments

- *Applications:* We use Microsoft Office suite of applications for the experiments. Specifically, we use Word, PowerPoint, Excel, and Visio for word-processing, presentation, spreadsheet, and graphic editing, respectively. Figure 5 shows the task labels that are used later in this Section for convenience. Figure 6 describes the different operations and the corresponding indices which will be used in the subsequent results.
- *Metrics:* For all the experiments, we use 1024×768 (XGA) screen display resolution. We measure the transfer size and latency as performance metrics. Note that the total synchronization time of Mimic is composed of three components: (i) the transfer latency for the FARs and MAR, (ii) the playback time, and (iii) the verification time. On the other hand, the synchronization time of *diff* is composed of (i) the time taken by *diff* to compute the differential patch, (ii) the transfer latency for the patch, and (iii) the time take at the server to use the patch to recreate the updated file.
- *Other information:* Windows operating systems uses Object Linking and Embedding (OLE), which provides means for integrating objects from diverse applications [7]. An object is a block of information that could come from a word processor, a spreadsheet, a graphic content, an audio clip, or an executable program itself. To provide compatibility with other applications, the content copied to the clipboard is stored in the form, which supports OLE taking much larger space than the original content in the memory. However, when the object is integrated with an application file, it undergoes compacting whereby the size of the file becomes significantly less than that in the memory. We assume that all data except *diff*, which is already compressed by its own algorithm, are compressed before being transmitted.

  In the experiments, we assume that there is no environmental change while recording and replaying.

5.2 Transfer Size Performance

5.2.1 Impact of User Activity Type

Figure 7 presents the experiment results of the transfer size for various user activity types in (W1-W4).

- *Insert:* In Figure 7, it can be seen that the transfer size in Mimic is usually equal or smaller than that for *diff* when an insertion (A1-A3) is performed. It is because each user input activity is translated into more complicated operations within the applications, and as a result the size of binary change becomes much larger than the input activity size.

  The figures also show that the update size in Mimic is always proportional to the magnitude of insertion in any application, while the *diff* size is unpredictable in text-centric applications (W1-W2). As explained in Section 2, word-processing and presentation applications have much more complicated file structures in order to embed various types of external objects. Hence, regardless of user activity size, each insertion affects the entire file structure. For example, the location of an insertion is one of the main factors that decides the magnitude of file structure change. When a character is inserted in the first page of a document, its *diff* overhead can be several times larger than that of a similar insertion in the last page of the document.

  On the contrary, the *diff* update size in graphic and spreadsheet applications (W3-W4) shows more modest performance improvements for Mimic. It is because those applications employ relatively simpler file structures and each insertion is translated simply as an addition of objects, paving the way for reasonable overheads when using *diff*.
- *Modify:* Files can be modified by changing either the attribute of the content such as font type, or the content itself. In the following experiments, we consider only the latter. This modifications consist of a combinations of multiple deletions and insertions. The former type of file modification is classified as meta data change, and explained later.

  In Figure 7, it is shown that the Mimic overhead is smaller than that of *diff* for most modification operations (A4-A5). However, the *diff* size in (W1-W2) does not increase linearly with the amount of activity as is the case with Mimic's record size for the same reason provided for the effect of insertions.

  On the other hand, the modifications in (W3-W4) show quite different results from those of insertions. Especially in (W4), the *diff* update size of modifications is even larger than that of insertions, while the Mimic overhead has the same size. This is because in graphic applications the size of an object is proportional to the complexity of the object.

Activity Type	Activity Index	Size of Change			
		W1	W2	W3	W4
Initial file state		5 full pages with 176 lines	8 full slides	83 rows	1 page with 10 small graphs
Insert	A1	1 line (0.6%)	1 new blank slide	1 row (1.2%)	1 new blank page
	A2	1 paragraph (3%)	1 paragraph (6%)	10 rows (12%)	1 small graph (10%)
	A3	1 full page (20%)	1 text slide (17%)	50 rows (60%)	1 large graph (17%)
Modify	A4	1 paragraph (3%)	1 paragraph (6%)	1 row (1.2%)	1 small graph (10%)
	A5	1 full page (20%)	1 slide (17%)	10 rows (12%)	1 large graph (17%)
Delete	A6	1 paragraph (3%)	1 paragraph (6%)	10 rows (12%)	1 small graph (10%)
	A7	1 full page (20%)	1 slide with a large picture (17%)	50 rows (60%)	1 small graph (17%)
Copy and paste	A8	1 paragraph from the same file (3%)	1 paragraph from the same file (6%)	1 row from the same file (1.2%)	1 small graph from the same file (10%)
	A9	1 full page from the same file (20%)	1 page from the same file (17%)	10 rows from the same file (12%)	1 large graph from the same file (17%)
	A10	1 page of an external file (20%)	1 page with a picture from an external file (17%)	1 row from an external file (1.2%)	1 small graph from an external file (10%)
	A11	1 picture from an external file	1 picture from an external file	10 rows from an external file (12%)	1 small picture from an external file
Meta data	A12	Change the font type of a paragraph	Change the font type of a slide	Change the font type of a row	Change the font type of a graph

Figure 6: User activity index for experiments

- *Delete:* When a user performs a delete or copy operation (A6-A7), the file structure can be dramatically changed even for a few number of deletes. Content deletions can be categorized into two types, full paragraph deletion and partial paragraph deletion. Generally, within the file structure, file contents are stored in paragraphs with each paragraph having its own content attributes. Therefore, if only some parts of a paragraph are removed, the file size reduction is not as large as that in full paragraph deletions. However, both cases are considered in the experiments.

  Figure 7 shows that Mimic's overhead is significantly smaller compared to the *diff* size in all the applications (W1-W4). This is because the deletion operations incur relatively large changes in the file structure while Mimic's overhead is proportional only to the activity size.

- *Copy and Paste:* Copying content means converting the content, which exists inside the file itself or externally, into another form that have compatibility with any other applications and loading it into the memory as a form of meta data. Once the content is copied into the clipboard, it can be pasted into any application.

  In the figure, the Mimic overhead for internal copying and pasting (A8-A9) is considerably smaller than that of *diff* in all applications (W1-W4). The reason is as follows. When an object that exists inside the same file is copied, Mimic does not have to send the content of the clipboard to the server, and instead it lets the object to be copied automatically at the server through the user activity itself. Therefore, it eliminates the clipboard overhead as seen in Figure 7.

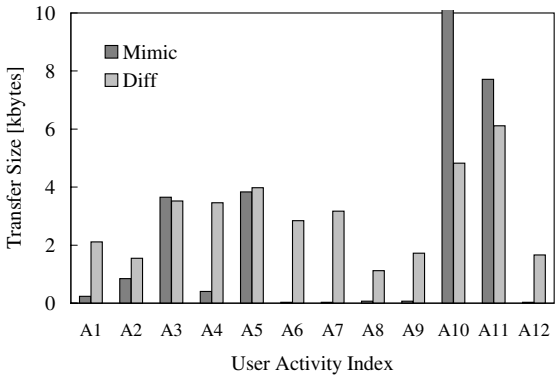
  However, if the object is copied externally (A10-A11), the Mimic overhead is significantly increased in (W1-W4). This is because Mimic captures and compresses the content of the clipboard as binary data, and then transmits it to the server along with the FAR. However, the copied objects that follow the OLE format have much larger size in the memory than the originally generated objects. In this scenario, *diff* usually shows better performance than Mimic due to large overheads for transferring the clipboard content.

- *Meta Data:* A user may change the configuration of textual information such as the font size or font type instead of the text content itself.

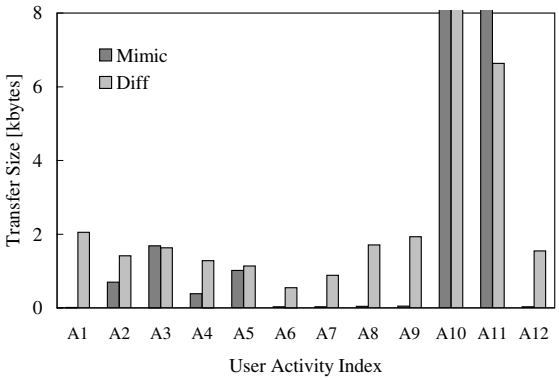
  In the experiments, modifications to meta data (A12) change the file structures drastically even though the file size change itself is small. It can be seen that Mimic shows much better performance than *diff* in all the applications (W1-W4). This is because changes to meta data are similar in nature to simple text insertion (both affect the entire file structure), with potentially a larger impact.

5.2.2 Impact of Application Type

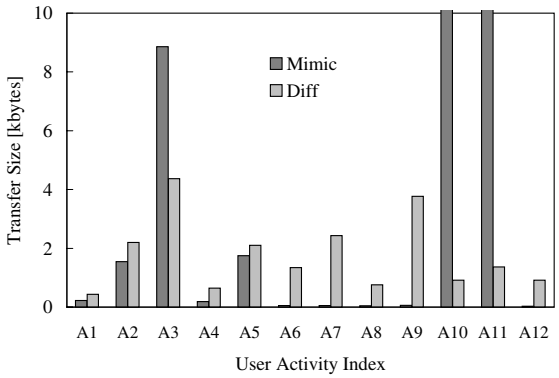
- *Word-processing:* Word-processing applications are basically text-centric programs that are able to embed various types of external objects such as pictures, graphs, tables, and equations. Hence, the file structure consists of a large number of data objects and their corresponding links. In Figure 7(a), *diff* in (W1) shows somewhat unpredictable performance results in (A1-A5) than Mimic due to its complicated file structure. Again, for activities (A6-A9,A12), the improvement brought about by Mimic is considerable. However, when a user copies a picture from another program, due to the clipboard problem discussed earlier, Mimic's overhead is dramatically increased.



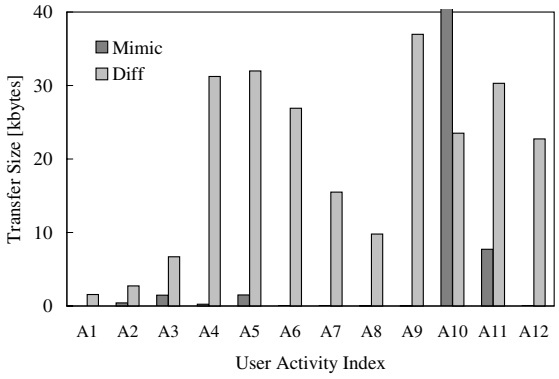
(a) Transfer Size for W1



(b) Transfer Size for W2

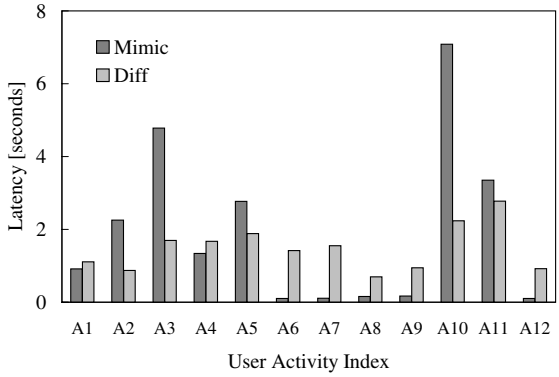


(c) Transfer Size for W3

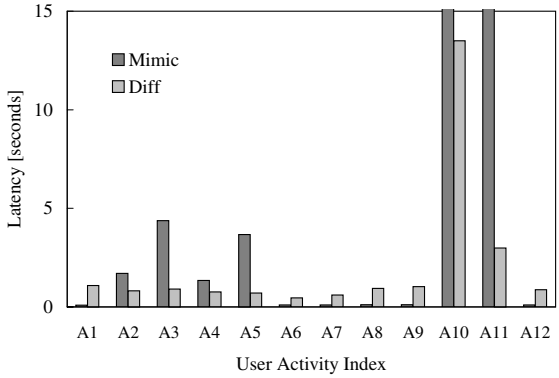


(d) Transfer Size for W4

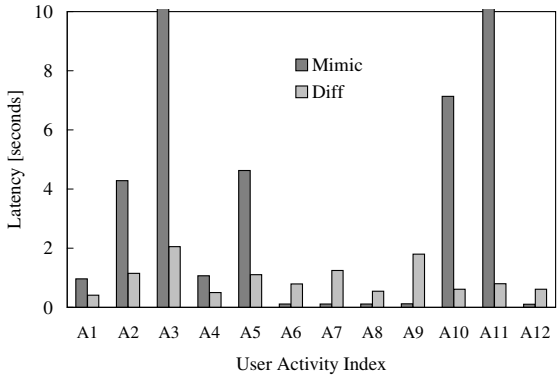
Figure 7: Transfer size



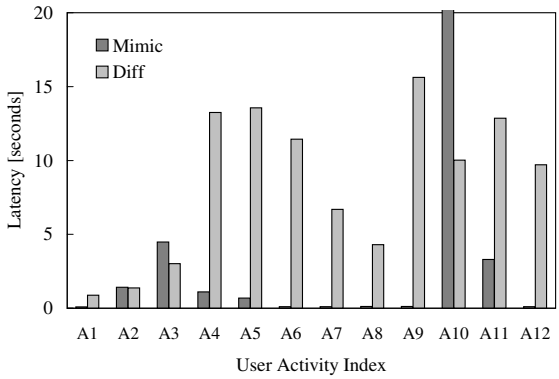
(a) Latency for W1



(b) Latency for W2



(c) Latency for W3



(d) Latency for W4

Figure 8: Latency

- *Presentation:* Presentation programs are popular applications for visual presentation where a user intends to present various types of data. Copying objects from other applications is one of the most important functions in addition to text editing. In Figure 7(b), in most cases of text editing and meta data change, Mimic shows equal or better performance than *diff*.

- *Spreadsheet:* A spreadsheet file consists of hybrid objects that are numerical values and graphs. In the experiment, we consider only the former type of data. Figure 7(c) shows the transfer size per user activity in Excel. Again, several activities including (A6-A9), and (A12) result in Mimic exhibiting large benefits in the transfer file size.

  However, Mimic in (W3) does not show the best performance for insertions and modifications (A1-A5) due to lots of redundant messages. Basically, a spreadsheet file visually consists of lots of cells, and a user usually moves the cursor to another cell using mouse-clicks. This is the reason why the Mimic size in (W3) is relatively much larger that that in (W1-W2) even though (W3) is a text-based application.

  While the external copy activities are detrimental to Mimic's performance as usual, the results for activity C3 is also interesting. Based on the results in the figure, it can be observed that when the degree of user activity is a substantial portion of the actual content of the file (20% in these experiments), Mimic's performance will start equaling that of *diff*, and sometimes even become worse.

- *Graphic:* In Figure 7(d), the performance of Mimic is *always significantly better* that that of *diff*. This is because *graphic editors* manage various complicated objects that include large number of environmental data. For example, when a user draws a circle, the circle has many parameters related to its setting such as display priority with a respect to the other objects or grouping with the other objects. Further its binary level complexity is much higher than text based data.

5.2.3 Impact of Update Interval

Figure 9 shows the overhead results with different update intervals when a user accesses multiple files (W1,W4) spending the same time per file, and performs various input activities. The input rate is about 200 operations per minute for (W1) and 85 operations per minute for (W4). The dominant operations used are insertions.

It can be observed that both Mimic and *diff* overhead increases in proportion to the update interval for mixed activities. This is because the overall overhead is dominated by the transfer size of (W4), whose overhead is increases almost linearly with a larger interval. Thus, as the interval becomes larger, the overhead difference is also increased linearly. In the experiments, Mimic reduces the size of overhead by about 40%.

5.2.4 Summary

In Figure 7, it can be seen that the transfer size in Mimic is generally equal or smaller than that in *diff* except when copying from outside the file. Overall performance in *diff* and Mimic can be characterized as follows.

Generally, Mimic's overhead is proportional only to the activity size. However, there are some exceptions, in which even Mimic overheads do not seem to be proportional to the magnitude of user activity, such as copying and pasting from an external source. Similarly, delete or modify operations can incur smaller overheads than their insert counterparts for the same magnitude of user activity.

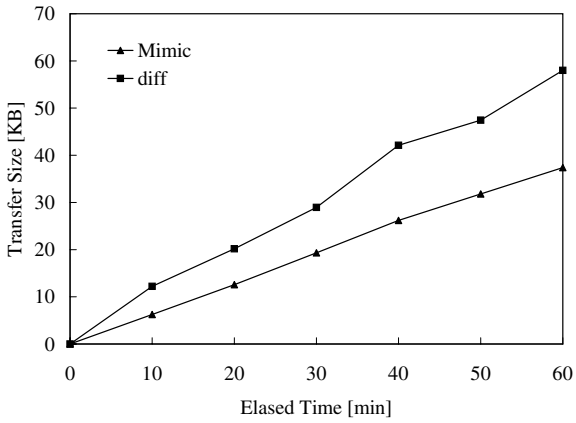


Figure 9: Transfer size for large-scale user activity

Equally interestingly, the single line insertion (A1) in *diff* consumes more bandwidth than a single paragraph insertion (A2) in Figure 7. This phenomenon is again due to the impact of application-specific storage semantics as discussed in Section 2.

5.3 Latency Performance

The synchronization latency of Mimic generally depends on the playback performance, which is decided by overall system capability such as CPU processing power. In the experiments, we assume that a user types about 200 characters per minute. The CPU idle check based playback mechanism in Mimic results in a maximum replaying speed of approximately 90 times the original speed in the experiments. If the server is equipped with more processing power, the maximum playback speed can further be increased, and the latency thus reduced. Figure 8 is the latency results for (W1-W4) when the synchronization is performed over a WWAN.

5.3.1 Impact of User Activity Type

In the experiments, for small insertions, deletions, internal copies, and meta data changes (A1,A6-A9,A12), Mimic performs better in terms of latency. Even though the latency in Mimic includes its playback time besides transmission time, its total update time does not exceed that of *diff* because the benefit of small transfer size for those operations is larger than playback overhead.

However, for the remaining types of activities such as large insertions, modifications, and external copies (A3-A5,A10-A11), Mimic performs worse than *diff* in latency in (W1-W3). Especially, moderate insertions (A2) in Mimic shows larger latencies even though its overhead is smaller because the playback latency becomes relatively large. Hence, if the transfer size itself in Mimic is already larger than that of diff, Mimic cannot show better latency performance.

This brings out an interesting trade-off: Can increase in latencies be tolerated if it reduces the total transfer size? For WWANs especially, where users may have to pay on a per-MB basis, this is arguably so.

5.3.2 Impact of Application Type

In the experiments, the latency performance in Mimic follows the trend of the transfer size performance approximately because the latency is closely relevant to the transfer size.

Finally, in Figure 8(d), Mimic shows overall better performance than *diff*, and it is because of both the lower transfer sizes, and the faster replaying speed used by Mimic. Hence, Mimic brings significantly larger benefits for such a mouse-centric graphic application that has a large ratio of file change to activity size.

5.4 System Overheads

Recording and CPU utilization monitoring are important components in Mimic. However, Mimic subsystems consume only a negligible portion of the system resources. In our experiments, the average CPU utilization for the recording process is less than 1%, and memory usage is about 4 MB. This is small compared to the memory used by an application such as Microsoft Word that can occupy up to 20 MB of memory and up to 50% of CPU utilization, when active.

6. OPEN ISSUES

- The Mimic approach currently works for applications, for which every file processing window has its own unique window handle. However, for applications where a single window can contain multiple file processing contexts, Mimic will simply revert back to the native file synchronization (already open FARs will be indexed, and hence the condition detected). While most interactive applications do not fall under this category of applications, there do exist applications which exhibit this phenomenon. Examples include the emacs text editor where multiple files can be manipulated within the same window, and it is the application that keeps track of which file is currently in context. Another class of applications that Mimic currently does not handle is one where the applications, in the context of a single thread, manipulate multiple files. This condition would again be detected when already opened FARs need to be opened again, and the synchronization of the files will be handled by the native synchronization process. We currently investigate how such applications can be handled.

- Mimic currently does not support synchronization of files, the updates of which depend on the updates of other shared files. As an example, consider the following sequence of activities: (i) the user opens shared file A, which is a Word document, and begins updating it; (ii) without closing file A, the user then opens shared file B, which is a Visio graphic file, and updates it, saves it, and closes it; (iii) the user then returns back to file A, and *inserts* the Visio graphic file as an embedded object in the Word document. When such file inter-dependencies exist, the replay of the dependent files should be done simultaneously and chronologically. However, the challenge is to be able to detect such inter-dependencies even as they are created. While a time-ordered playback of all FARs at the same time would not require such detection, we are currently investigating ways of detecting the dependencies so that only dependent files need to be played back together. A more difficult problem is when such dependencies exist between shared and non-shared files. Mimic does not handle such dependencies either.

- In Section 3, we described the interface between Mimic and the underlying file system. Even though Mimic has been designed to be loosely coupled with the file system, it is conceivable that a tightly coupled integration with the file system can actually improve performance. For example, decisions between FARs and *diff*s can be taken at a smaller granularity than between two consecutive instances of file synchronization. Our ongoing work is investigating such a tightly coupled integration with the file system.

- The Mimic design requires that the operating environment at the client and the server be the same in terms of the operating system, system settings such as screen resolution decided by the hardware, and application types and versions. This is because of the fact that the interpretation of raw user activity in terms of keyboard and mouse inputs interpretation is closely dependent on the operating environment of the system. There are multiple levels of complexity in tackling environment synchronization:

  1. The initial environment of the client and the server is the same before any user-activity begins, and the user's activity is restricted to solely the context of the file, and no changes are made to the operating environment. This is the simplest scenario where operating environment need not be captured for accurate playback of user activity. The FAR is sufficient to tackle this class of updates.
  2. The next level of complexity is when the user changes not just the file's content, but also the settings of the application that is used to update the file. An example of the latter class of updates can be a reorganization or customization of the application menus. Such activities in Mimic are also captured as part of the FAR, and are replayed as is at the server.
  3. The third level of complexity is when the user changes the operating environment beyond the scope of the file's content or the processing application. A good example of such an update is a change of the keyboard layout, or the screen resolution. Such changes in Mimic are recorded as part of the MAR, as the changes are independent of any specific file, and will impact all files that are updated from that point onward. The time-wise integrated FAR/MAR replay at the server ensures that such meta-activities are also reflected in the correct recreation of files.
  4. Finally, the most complex scenario is when the initial desktop environment settings are different at the client and the server. This requires the synchronization of the environment before starting the Mimic process in the first place, so that subsequent FAR/MAR playbacks can successfully recreate any activities at the client. Most operating environments have system defining files (eg. Word.pip, PowerPoi.pip, etc.) that can be shipped offline to the server to achieve such synchronization.

- Conflict resolution is necessary for environments where a single shared file can be manipulated by different clients at the same time. While simple conflict resolution techniques can involve prioritized merging (where one client is given priority over the others), or latest update merging (where the latest update is given preference), most conflict resolutions might have to be performed through manual intervention. We believe that Mimic's *user-activity* replay technique is more

amenable to such manual conflict resolutions. This is because of the fact that conflict decisions can now be performed at a semantic level that is understandable by the user (e.g. do I cut this paragraph or not?), as opposed to at a binary level. We are currently exploring these benefits of Mimic in more detail.

- The Mimic file synchronization scheme is not necessarily applicable only for upstream synchronization (client to server), and can be used for downstream synchronization (server to client) as well. However, two issues render the use of Mimic less attractive in the downstream direction when compared to the upstream direction: (i) The downstream bandwidths are typically much larger than the upstream bandwidths in most WWAN environments, rendering the savings in bandwidth the Mimic delivers less critical; and (ii) The computational and latency overheads of Mimic might be less tolerable when the replaying is performed by the client itself. Note that when the replay processing is ongoing, the user will be unable to actively work on the client.

- We have used Microsoft Office suite of applications for evaluating the performance of Mimic. These are representative of binary file formats and we have shown that Mimic achieves smaller overhead compared to *diff* based schemes for such file formats. For other file formats such as text-based formats (eg. XML), Mimic will not achieve such high improvements as in the case of non-text based formats such as Microsoft PowerPoint.

7. RELATED WORKS

While we have discussed the *operation shipping* strategy proposed in Section 2, we now briefly discuss two other related works that pertain to the problem studied in this paper.

The Low-bandwidth network file system (LBFS) [8] exploits cross-file similarities between files just as the Unix command `diff` does for text files. It exploits the fact that updated files often contain a number of segments in common with previous versions of the same files. But, the patching algorithm is different. The LBFS file server divides the files into chunks and indexes a large persistent file cache. When transferring a file, LBFS identifies chunks of data that the server already has. Then, the client sends only non-overlapped chunks to the server. Off-the-shelf software packages that compute *diff*s are also available, and examples include xDelta [14], .RTPatch [10], exeDiff [1], and BSDiff [3]. However, all such strategies will have the same limitations identified earlier in the paper for *diff* based approaches.

The Prayer file system (PFS) [4] also performs differential updates, but requires applications to store data in a pre-specified file format. Once applications are adapted to store data as *records* with a given template, synchronization is performed by shipping only records that have been updated. However, the strategy requires full application support, and the authors do not explore whether or not the proposed approach is feasible beyond simple applications such as *e-mail*.

8. CONCLUSIONS

In this paper, we consider the problem of file synchronization when a mobile host shares files with a backbone file server in a network file system. We show that *diff* based file synchronization schemes incur substantially more overheads than necessary. We then propose an application-independent approach called *Mimic* that relies on transferring user activity records to the server, where the new file is recreated through a playback of the user activity on the old copy of the file. We show that Mimic performs much better than *diff* in most scenarios in terms of the transfer file sizes. The trade-off is that the latency incurred by Mimic due to its replay mechanism can be larger than the overall latency incurred by *diff* schemes. We also identify some conditions under which Mimic incurs more transfer size overheads than *diff*. Despite the trade-offs, we conclude that Mimic can be used in tandem with *diff* to substantially improve file synchronization performance, especially when the bandwidth available on the network connection is low and expensive.

9. ACKNOWLEDGMENTS

We gratefully acknowledge Professor Mahadev Satyanarayanan for his insightful help and comments. He shepherded us through a complete re-write of the paper to significantly improve it in terms of focus, readability, and details. We also would like to thank the anonymous MobiSYS'04 reviewers for their comments.

10. REFERENCES

[1] B. S. Baker, U. Manber, and R. Muth, "Compressing Differences of Executable Code," in *ACM SIGPLAN Workshop on Compiler Support for System Software*, Apr. 1999.

[2] BASH-GNU Project-Free Software Foundation (FSF), http://www.gnu.org/software/bash/

[3] Binary Diff/Patch Utility, http://www.daemonology.net/bsdiff/

[4] D. Dwyer and V. Bharghavan, "A Mobility-Aware File System for Partially Connected Operation," in *ACM Operating Systems Review*, vol. 31, no. 1, pp. 24-30, Jan. 1997.

[5] Y. Lee, K. Leung, and M. Satyanarayanan, "Operation Shipping for Mobile File Systems," in *IEEE Transactions on Computers*, vol. 51, no. 12, pp. 1410-1422, Dec. 2002.

[6] R. Rivest, MD5 Message-Digest Algorithm, http://www.faqs.org/rfcs/rfc1321

[7] Microsoft Developer Network (MSDN), http://msdn.microsoft.com/

[8] A. Muthitacharoen, B. Chen, and D. Mazieres, "A Low-Bandwidth Network File System," in *Proceedings of the 18th Symposium on Operating Systems Principles*, Banff, Canada, Oct. 2001.

[9] I. S. Reed and G. Solomon, "Polynomial Codes over Certain Finite Fields," in *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, pp. 300-304, Jun. 1960.

[10] RTPatch, http://www.pocketsoft.com/rtpproducts.html

[11] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere, "Coda: A Highly Available File System for a Distributed Workstation Environment," in *IEEE Transactions on Computers*, vol. 39, no. 4, pp.447-359, 1990.

[12] A. Tridgell, "Efficient Algorithms for Sorting and Synchronization," PhD thesis, Australian National University, 1999.

[13] T. Chang, A. Velayutham, and R. Sivakumar, ""Mimic for Microsoft Windows and X Window," Technical Report, GNAN Research Group, Georgia Institute of Technology, Nov. 2003.

[14] xDelta, http://www.eng.uwaterloo.ca/ ejones/software/xdelta-win32.html