Operation Shipping for Mobile File Systems

Yui-Wah Lee, *Member*, *IEEE*, Kwong-Sak Leung, *Senior Member*, *IEEE*, and Mahadev Satyanarayanan, *Fellow*, *IEEE*

Abstract—This paper addresses a bottleneck problem in mobile file systems: the propagation of updated large files from a weakly-connected client to its servers. It proposes an efficient mechanism called *operation shipping* or *operation-based update propagation*. In the new mechanism, the client ships the *user operation* that updated the large files, rather than the files themselves, across the weak network. (In contrast, existing file systems use value shipping and ship the files.) The user operation is sent to a surrogate client that is strongly connected to the servers. The surrogate replays the user operation, regenerates the files, checks whether they are identical to the originals, and, if so, sends the files to the servers on behalf of the client. Care has been taken such that the new mechanism does not compromise correctness or server scalability. For example, we show how forward error correction (FEC) can restore minor reexecution discrepancies and, thus, make operation shipping work with more applications. Operation shipping can be further classified into two types: *application-transparent* and *application-aware*. Their feasibilities and benefits have been demonstrated by the design, implementation, and evaluation of a prototype extension to the Coda File System. In our controlled experiments, operation shipping achieved substantial performance improvements—network traffic reductions from 12 times to nearly 400 times and speedups in the range of 1.4 times to nearly 50 times.

Index Terms—Operation shipping, operation-based update propagation, mobile file systems, surrogate client, application-awareness, application-transparency, forward error correction, Coda File System.

1 Introduction

Mobile computers, unlike their stationary counterparts, are often at the mercy of weak connectivities—networks that are intermittent, low-bandwidth, expensive, or high-latency [31], [13]. A mobile file system is a distributed file system that works well even with these unpleasant networks. Previous research has demonstrated the feasibility and benefits of disconnected operation, in which a file-system client can continue to function even when it loses network connectivity to its server [12]. They have also demonstrated that weak connectivity can be exploited. A key technique for the latter is to decouple the slow update propagation from the foreground processing of file-system requests [25].

Unfortunately, even though update propagation is now a background activity, it is still a performance bottleneck in a weak network. Because it is traditionally done by shipping updated files in their entirety, and it can cause substantial network traffic, Large files are common and they can easily overwhelm a weak network. For example, it will take at least 13 minutes to ship a 1-Mbyte file in its entirety across a 9.6-Kbps wireless modem link. When the file is shared by more than one user, this imposes an unpleasant latency during which the latest version is not available to other users. Also, during that period, the client is expected to maintain its network connectivity, which may be impossible (the client may have wandered into a radio blind spot) or undesirable (high energy consumption or high connection charge). Obviously, there is a need to reduce the network traffic and latency. In the literature, *delta shipping* and *data compression* are often suggested as the solutions to the problem. As discussed later in this paper, they both have shortcomings that limit their usefulness.

In this paper, on the other hand, we propose a radically different technique: *operation shipping* (also called *operation-based update propagation*). We are motivated by two observations. First, although many files are large, the *user operations* that created or modified them are usually small and easy to intercept. Second, these user operations can often be reexecuted to regenerate the files at modest computation costs. We therefore propose that a weakly-connected client should, when appropriate, ship the user operation rather than the files it updated. Fig. 1 depicts the idea.

While operation shipping is conceptually simple, we emphasize that it is not as simple as `rsh file-server latex` (i.e., manually rerunning the command `latex` on a remote `file-server` using a remote shell `rsh`). Specifically, we need to address a number of issues before we can apply it in the context of mobile file systems. First, how can user operations be logged? And will this logging mechanism be backward-compatible with existing applications? Also, how can these operations be shipped in lieu of the files that they updated? Section 3 will discuss these issues in detail.

Second, we need to consider the location for reexecution to happen. We propose a new per-client entity in the system: *surrogate*. As discussed in Section 3.6, a surrogate is a special client that is strongly connected to the server, and is assigned as a helper of a weakly connected client. By using surrogates, we can avoid adding extra workload to the servers, and we can more easily ensure that their execution environments closely match those of the clients.

- Y.-W. Lee is with Bell Laboratories, Holmdel, NJ 07733. E-mail: leecy@bell-labs.com.
- K.-S. Leung is with the Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong. E-mail: ksleung@cse.cuhk.edu.hk.
- M. Satyanarayanan is with the Department of Computer Science, Carnegie Mellon University, Pittsburgh PA 15213. E-mail: satya+@cs.cmu.edu.

Manuscript received 20 Mar. 2001; revised 7 Dec. 2001; accepted 25 Mar. 2002.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number 113823.

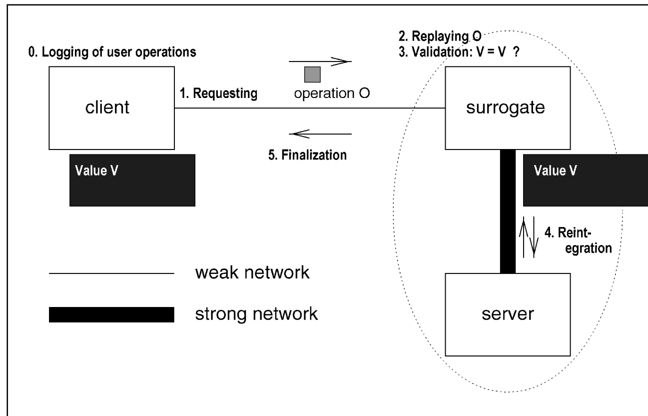


Fig. 1. Overview of operation shipping. This figure presents a high-level view of operation shipping. The zeroth step is for the logging of user operations. The first through the fifth step are for the shipping of updates by operation. The concept of the surrogate will be explained in Section 3.6.

Third, we have to make sure the correctness of the new mechanism. For example, the version of a file on the client is the authoritative copy while the version regenerated by the surrogate is not. Our file system must guarantee that only the authoritative version is eventually written back to the file server. Section 4 will explain our strategy. It will also present some interesting results that surprised us—we found that many common applications regenerate files that are not identical to the authoritative copies. We refer to these as nonrepeating side effects or reexecution discrepancies. Yet, even in these situations, our file system will still perform correct update propagation. We describe two techniques, one based on forward error correction (FEC), and the other by intelligently renaming temporary files, to accomplish this. (Sections 4.3, 4.4, and 4.5).

To validate our approach, we have designed, implemented, and evaluated a prototype system. The main component of the system is an extended version of the Coda File System [33], [12], [24], [7]. We also made minor extensions to a popular UNIX shell called the Bourne Again Shell (`bash`) and an image application called the GIMP. They serve as case studies showing how operation shipping will interact with existing noninteractive and interactive applications. The source code of the prototype system can be downloaded from [16]. We evaluated the system using controlled experiments, and found that operation shipping can achieve substantial performance improvements. In our experiments, the network traffic reductions were in the range of 12 times to nearly 400 times, and the speedups were in the range of 1.4 times to nearly 50 times.

As we will discuss in Section 3, there are two types of operation shipping: application-transparent and application-aware. An early version of this paper [18] has presented some results for the first type. In this paper, we will present a complete picture, discuss both types of operation shipping, and report the experimental results of application-aware operation shipping.

2 Coda Background

The Coda File System has been used as a research vehicle for a number of advanced file-system techniques, such as disconnected operation, replicated file servers using optimistic concurrency control, etc. It is still being actively developed and maintained by both the Carnegie Mellon University and a team of volunteer programmers around the Internet [7]. It has been well-documented in the literature [33], [12], [14], [25], [21], [3], [32], [30], so here we only provide a very brief background.

Low-level file-system operations	CML record logged
`chown`	`CHOWN`
`chmod`	`CHMOD`
`utimes`	`UTIMES`
`open`	`STORE` (see note below)
`write`	`STORE` (see note below)
`close`	`STORE` (see note below)
`mkdir`	`MKDIR`
`rename`	`RENAME`

Fig. 2. Examples of low-level operations and their corresponding CML records. A `STORE` record will be used to log a sequence of several operations: a mutating `open`, possibly interspersed with some `write` operations, and a `close`, all on the same file.

Coda uses a client—server model. In each Coda installation, there are many clients and a few servers.[1] On each client, a cache manager, called Venus, carefully manages and persistently stores cached objects (files, symbolic links, and directories). To support mobile computing, Coda clients can be used in *disconnected* and *write-disconnected* modes. In these two modes, Venus temporarily operates independently, and allows file-system operations to be performed on objects even where there is no, or merely some very weak, connectivities to the servers. In these cases, updates are applied immediately to locally cached objects, and are also logged in a client-modify log (CML). The logging mechanism allows updates to be eventually propagated to the servers that maintain the primary replica of the objects [11], [24]. This eventual propagation is called *reintegration*.

A CML consists of records called CML entries, each is recording the effect of a mutating file-system operation. For example, a `chmod` operation is logged as a `CHMOD` record, and a `mkdir` operation is logged as a `MKDIR` record. Fig. 2 lists some CML record types.

The record type `STORE` is special. First, it is recording the effect of a sequence of operations: a mutating `open`, interspersed possibly with several `write`'s, and a final `close` operation on a file. Coda maps the whole sequence of operations to a single `STORE` record because it uses a session semantics [11]. Second, a `STORE`'s associated data includes the content of the file being stored. In contrast, other record types do not include the content but only some directory attributes such as the owner or the name of an object. Note that file contents can be as large as many kilobytes or megabytes. Therefore, Venus does not log the content of a stored file in CML; rather, it only keeps a pointer to a container file, which is a regular Unix file serving a double role as both the cache copy of the file and the logged value of the store operation.

1. Coda supports the use of replicated servers [33], [14]. When an object is multiply replicated, a client needs to talk to multiple servers for processing the object. In the following, we use the plural form of the noun "servers," which actually covers also the case of a singly replicated server.

Traditionally, Coda uses a value-based approach for propagating STORE records. Container files, which represent the logged values of store operations, are shipped across the weak network [11]. This is exactly the performance bottleneck that can be optimized out by operation shipping. In the next section, we will see how Venus can perform update propagation without shipping the bulky container files.

Like in any distributed file systems, there are cases when two clients simultaneously read or write on the same file. Designed for high availability in environments where network disconnections are involuntary and unpredictable, Coda uses an optimistic approach for replica control. It allows read and write operations on all clients, and certifies these operations as conflict-free upon reintegration. When conflicts happen, Coda provides manual and automatic tools to resolve the conflicts.

3 Logging and Shipping of User Operations

3.1 User Operations versus Values

High-level user operations, the focus of this paper, should not be confused with *low-level file-system operations*. Examples of the former are invocations of noninteractive applications, such as `latex` or `make`, as well as application-specific commands of interactive applications, such as changing color balance of an image in a photo editor or replacing a string in an text editor. Examples of the latter are CHOWN, CHMOD, STORE, etc. Note that a user operation O can generate a number of low-level operations. We denote the latter as $V = \{V_1, V_2, \ldots, V_m\}$, the *values* of the former. For example, a user operation `latex usenix99.tex` will generate a number of CREATEs and STOREs; a user operation `ar rv libsth.a foo.o bar.o` will generate a sequence of CREATE, STORE, REMOVE, and RENAME. Also, user operations should not be confused with key strokes and mouse clicks. The latter are uninterpreted raw input and have little meaning if taken out of context. In contrast, the former are interpreted and their dependencies on context are much smaller.

User operations are logged when they are performed by users. The logging procedures involves the passing of three pieces of information to the file system: 1) A user operation O has happened, 2) O has been executed with a context C, and 3) O has generated a set of values V. A logging entity is responsible for passing this information to the file system.

3.2 Application-Transparent versus Application-Aware

There are two types of user operations. The first includes commands that invoke noninteractive applications, such as `make` or `latex`; while the second includes commands performed to certain interactive applications—examples are "change color balance" in an image editor and "replace string" in a text editor. It is important to make this distinction because they corresponds respectively to *application-transparent* and *application-aware* operation shipping. The former case is relatively easy to understand, and operation shipping can be made transparent to the applications. The latter case is no less important than the former, but it involves a slightly more complicated mechanism. In particular, applications in the latter case must be modified—they must have mechanisms to log user operations on the client and to replay user operations on the surrogate.

3.3 Logging of User Operations

In this section and the next, we will first discuss the application-transparent case, and then extend the discussion to the application-aware case.

3.3.1 Application-Transparent Logging

To log user operations transparently, our file system makes use of the Unix concept of process groups to associate user operations with the low-level operations that they generate. Note that a user operation is identified with its process group but not with individual processes since some applications (such as `make`) spawn child processes to carry out subtasks of the same user operation.

The new file system supports operation logging by extending its file-system interface, which can be used by a logging entity—a Unix shell in this case. The extension comprises two new system calls: `VIOC_BEGIN_OP` and `VIOC_END_OP`. Together, they define a logging session of the process group. During a logging session, the effect of every mutating file-system call from the same process group is regarded by Venus as a part of the value generated by the user operation. Fig. 3b shows how a logging shell would make use of the new interface, and how this is done transparently to the applications. For comparison, the actions of an ordinary (i.e., nonlogging) shell is shown in Fig. 3a.

During a logging session, regardless of the network connection quality, Venus will put itself into the write-disconnected mode (Section 2). That means it will not immediately write through the individual updates to the server. Instead, it will log these updates and propagate them altogether later, possibly using operation shipping if the validation succeeds. Section 3.4 will explain the propagation mechanism.

Besides instrumenting the command shell, it is also possible to instrument the kernel so that it logs all user operations when `exec` calls happen. However, we favor a nonkernel approach because of the following reasons. First, we believe that we should put into the kernel only those functionalities that are absolutely necessary to be there—operation logging is not such a functionality as it can be implemented in user space. Second, our approach makes it easier to implement flexible policies (such as opting out some applications).

The logging mechanism described previously can be applied to any Unix shell. As a case study, we apply it to the GNU Project's Bourne Again Shell (`bash`) [5]. The extension involves only a few lines of code (the source code is available for download from [16]). Our current implementation serves only as a proof of concept and is simplistic in some senses—it logs and replays all user operations, and it always uses operation shipping (unless it is forced to fall back on value shipping). A more realistic implementation can allow users to "opt-out" operation logging/shipping for some applications. The file system can also dynamically decide on the best mode of update propagation based on the network conditions, application execution time, etc. See [17] for some of these possible extensions.

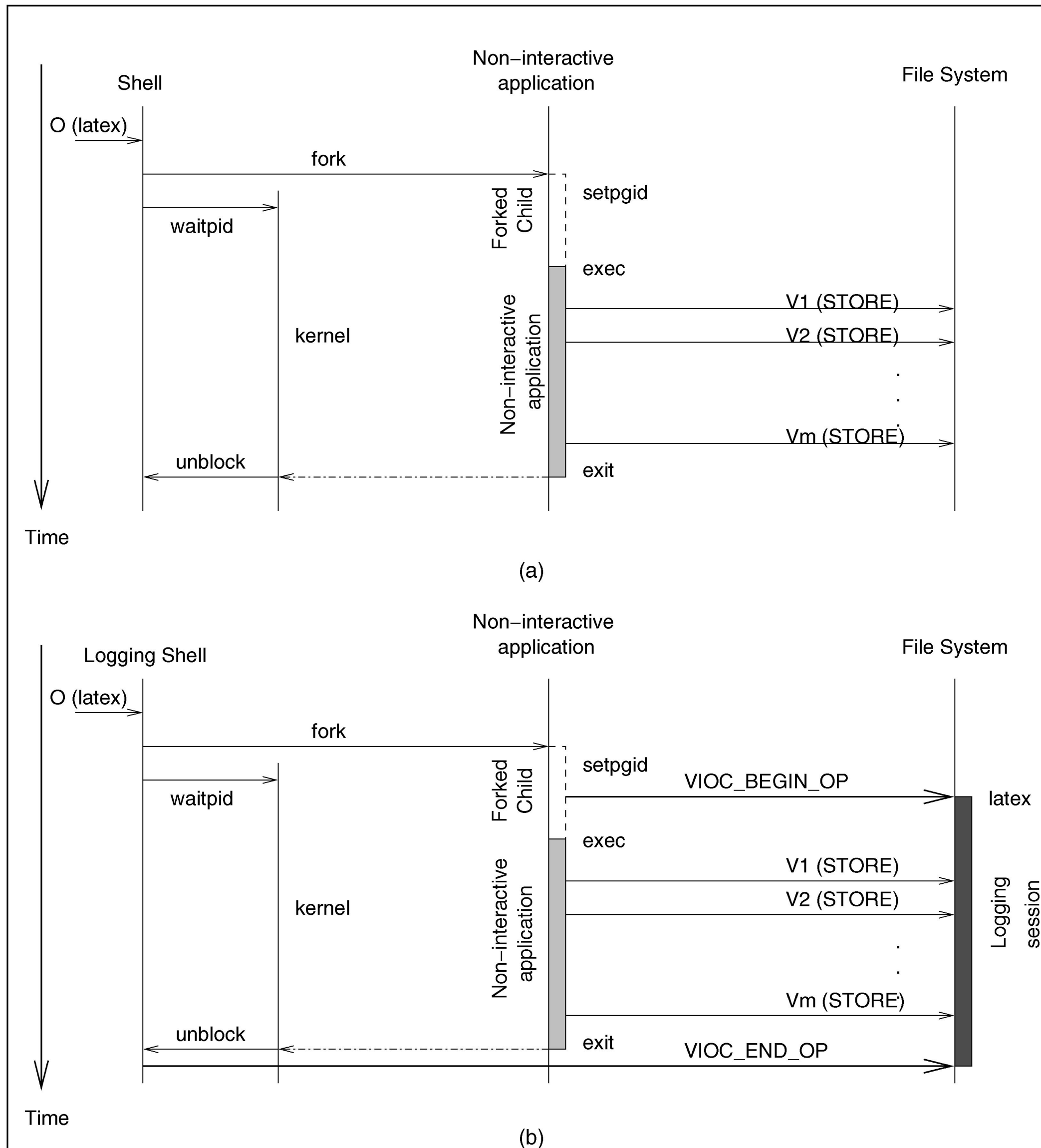


Fig. 3. Logging of user operations. (a) The typical sequence of an ordinary (i.e., nonlogging) shell executing an application. (b) A logging shell logs the user operation (such as `latex usenix.tex`) by inserting the `VIOC_BEGIN_OP` and `VIOC_END_OP` syscalls before and after the execution of the noninteractive application. $V_1, V_2, \ldots, V_m$ are the file-system operations that the application generates. Examples of them are `CHOWN`, `CHMOD`, and `STORE`.

3.3.2 Application-Aware Logging

For application-aware operation logging and shipping, a new element—the *application-specific operation log*—is required. The application logs the user operations, prepares the log, and passes the log to the file system using a new syscall `VIOC_PUT_APP_OPLOG`. Fig. 4 illustrates the interaction between the application and the file system. The file-system client does not interpret the log but just forwards it to the surrogate. On the surrogate, a reexecuting instance of the same application retrieves the log using a new syscall `VIOC_GET_APP_OPLOG`.

Note that, in this case, the application must be made aware of the new update propagation mechanism. That is, to participate in operation shipping, an existing application has to be modified. To demonstrate this process in an example, we modify an existing application—GIMP, or the GNU Image Manipulation Program, which is an open-source program popular for tasks such as photo retouching, image composition, and image authoring [6], [15]. Being an interactive application, GIMP has to be modified before it can participate in operation shipping.

Fortunately, we found that the needed modification is moderate [17]. It does not involve major changes in the internal logic of the application but only some minor alternations on the user-interface modules—each GIMP command being logged will need a few lines of code for logging. The key insight here is that it is feasible yet not exceedingly difficult to apply operation logging to interactive applications. Our prototype currently can log 30 different user commands, such as `jpeg_load`, `normalize`, `brightness/contrast`, `text`, etc. The number represents about one sixth of the total number of user commands that exist in GIMP. In terms of the number of lines of code, we added about 2,000 lines, while GIMP has about 314,000 in total. In terms of time, it took us a few weeks to understand the structure of the system and enable

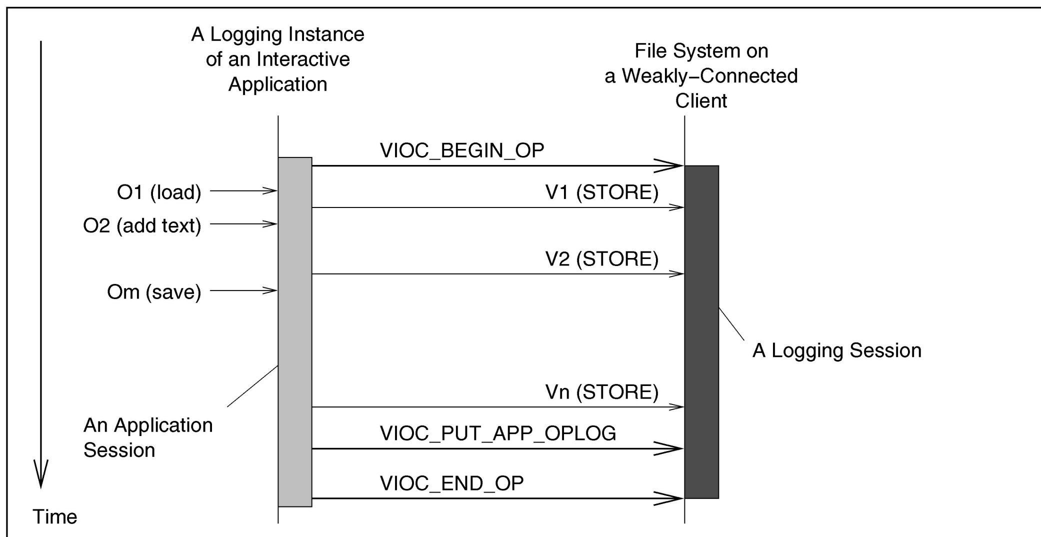


Fig. 4. Application-aware operation logging. This figure shows what happens when a user interacts with an interactive application (such as GIMP). Examples of user operations are to `load` an image, to `add text` to the image, and to `save` the image. $V_1, V_2, \ldots, V_m$ are file-system operations that the application issues. Examples of these are `STORE`, `RENAME`, and `MKDIR`. The application also captures the user operations on an application-specific log, and passes the log to the file system using the syscall `VIOC_PUT_APP_OPLOG`.

logging for the first few commands, but enabling more commands becomes straightforward after that.

The extended GIMP[2] can run in two special modes: `oplog` and `reexec`. They are for use on a client and a surrogate respectively. The GIMP-specific operation logs use Script-Fu scripting facilities and the Scheme language [15]. Fig. 5 shows an example.

3.4 Shipping of Logged Operations

3.4.1 Application-Transparent Shipping

The shipping stage has five phases and involves all three parties in the system—the client, the surrogate, and the servers. The five phases are:

1. requesting,
2. replaying,
3. validation,
4. reintegration/aborting, and
5. finalization.

Fig. 6 depicts an overview of the shipping stage; [17] gives more details. The mechanisms are very similar for both application-transparent and application-aware cases. We will first present the mechanism for the former, and then discuss the additional mechanism needed for the latter.

Among the five phases, the key is the replaying phase, in which Venus reexecutes a user operation by `fork`ing and `exec`ing the application. The hope is that a reexecuted user operation O (such as `latex`) will regenerate a set of values $V' = \{V'_1, V'_2, \ldots, V'_m\}$ (such as {`CREATE`, `STORE`, ..., `RENAME`}) that is identical to the original ($V = \{V_1, V_2, \ldots, V_m\}$). To associate a user operation with the effects of the file-system calls that it emits, Venus puts itself into a replaying session. As in a logging session, every call that comes from the same process group of the reinvoked application will be regarded by Venus as from the same user operation. Before `exec`ing the application, Venus will also restore the execution context, which includes the command-line arguments, environment variables, working directories, and file-creation mask (`umask`).

Similar to the case in a logging session, Venus will put itself into the write-disconnected mode during the replaying phase. Here, the reason is that we want to make sure that all reexecutions are abortable transactions—their effects will be propagated to the server only if they can pass the validation phase (Section 4). In other words, the write-disconnected mode is being exploited here to provide failure atomicity. If an reexecution cannot be validated, all its effect will simply be discarded from the system, and the client will fall back to use value shipping.

3.4.2 Application-Aware Shipping.

The shipping stage for the application-aware case is very similar to the previous case. There are two main differences. First, the replaying entity is the application itself but not Venus. Second, the syscall `VIOC_GET_APP_OPLOG` is needed here. It is retrieved by the application so as to replay all the user operations and to hopefully regenerate the same set of low-level operations.

3.5 Cancelation Optimization

Sometimes a file is updated many times before it is reintegrated. Intuitively, the intermediate states are not needed since they have no effects on the final states. To exploit this intuition, Coda has a mechanism called *cancelation optimization* that will cancel CML records that have no final effects. An example of these records are the nonfinal `STORE` records for a file that is updated many times before reintegration (see Chapter 6.3 of [11] for the formal treatment on this subject). For operation shipping, we need to slightly modify the standard procedure so as to preserve the needed information for validation. Specifically, an otherwise canceled record will be kept around as a ghost record so as to preserve the necessary information for

2. The prototype is based on the mainstream version 1.0.2. The source code of the extended GIMP is available for download from [16].

```
(define (script-fu-oplog-reexec)
  (let*
      ( ; declaring local variables
        (theImage_1)
        (theLayerDrawable_2)
        (theLayerDrawable_3)
      ) ; end of local variables
      (set! theImage_1 (car (file-jpeg-load 1
        "gibralter2.jpg" "gibralter2.jpg")))
      (set! theLayerDrawable_2 (car
        (gimp-image-get-active-layer theImage_1)))
      (set! theLayerDrawable_3 (car (gimp-text-ext theImage_1
         theLayerDrawable_2 263.000000 571.000000
         "Gibralter" 0 1 15 0
         "*" "helvetica" "*" "*" "*" "*" "*" "*")))
      (gimp-layer-translate theLayerDrawable_3 23 0)
      (gimp-floating-sel-anchor theLayerDrawable_3)
      (file-jpeg-save 1 theImage_1 theLayerDrawable_2 "
         /coda/usr/c.clement/tmp/test2/t31.jpg" "t31.jpg"
         1.000000 0.000000 1)
  )
)
```

Fig. 5. An example GIMP-specific operation log. This figure shows a portion of an example GIMP-specific operation log. It records the following user operations. The user loaded a jpeg file name “gibralter2.jpg”. He then added a text string “Gibralter,” moved the text layer to the lower right corner of the image, and “anchored” (pinned) the layer. Finally, he saved the annotated image to another file called “t31.jpg.” The log was automatically generated by the extended GIMP, but here it was slightly edited (with long lines split into shorter lines) for better presentation in the figure.

validation (Section 4.2). This modified procedure increases the overhead slightly but preserves the effectiveness of cancelation optimization (see Chapter 4.4 of [17]).

3.6 Surrogate

The key roles of a surrogate[3] are to reexecute user operations and to reintegrate results with the servers on behalf of its weakly-connected client. There are three reasons why we suggest that reexecutions should *not* be done on the servers but rather on a surrogate. First, we want to avoid adding extra workload to the servers, which, being the focal points of the systems and the “hot spots” of activities, are probably already the bottleneck of the systems. Second, by using per-client surrgoates, we can more easily ensure that their execution environments closely match those of their respective clients. Third, we can avoid requiring the servers to execute arbitrary and potentially malicious binaries supplied by users.

The following are the desired properties of a surrogate machine:

1. It should be strongly connected to the server, so that the shipping of the reexecution result to the server can be done cheaply.
2. It should provide an execution environment as similar as possible to the weakly-connected client that it services.
3. It should be at an adequate level of security, and process suitable authentication tokens for the user requesting services.

We currently propose that each weakly-connected client will have its own dedicated and statically assigned surrogate. This is conceptually most simple to the users and system administrators. There may be a concern about the need of extra hardware for surrogates. However, in many cases, the surrogate machines are already in place and can be used for free. This is because many users own both a desktop machine and a notebook machine—the former usually sits idle while the user travels. In any case, it is possible to extend our model so that a surrogate machine will be shared by multiple clients, but we leave this as future work.

3. We pick “surrogate” as the name, instead of its synonyms proxy and agent, to differentiate our concept from other computer concepts, such as “web proxy” or “Internet search agent.”

4 Correctness

4.1 Strategies for Preserving Correctness

We define a round of operation shipping to be correct if the set of values (CML records) that the servers received from the surrogate is totally identical to what the servers would have received from the client using value shipping.

There are five components in our strategies for preserving the correctness. First, we increase the likelihood that a user operation will be repeated on the surrogate (i.e., it will produce exactly the same set of values as its original execution). We do this by having the surrogate be identically configured as the client, and by restoring the execution context upon reexecution (Sections 3.4). Second, the surrogate will try to fix any nonrepeating side effects that may have happened (Sections 4.3, 4.4, and 4.5). Third, the surrogate will adjust the regenerated file-system objects' metadata, such as modification time, `fid` (which is the low-level identifier of file-system objects), and store ID (which is used for concurrency control). This adjustment is needed since these metadata are time-dependent and will not repeat upon reexecution. For each of them, the reference value for adjustment is the one that was used in the original execution. Fourth, after that, the surrogate validates that the final set of values in the surrogate is identical to the original

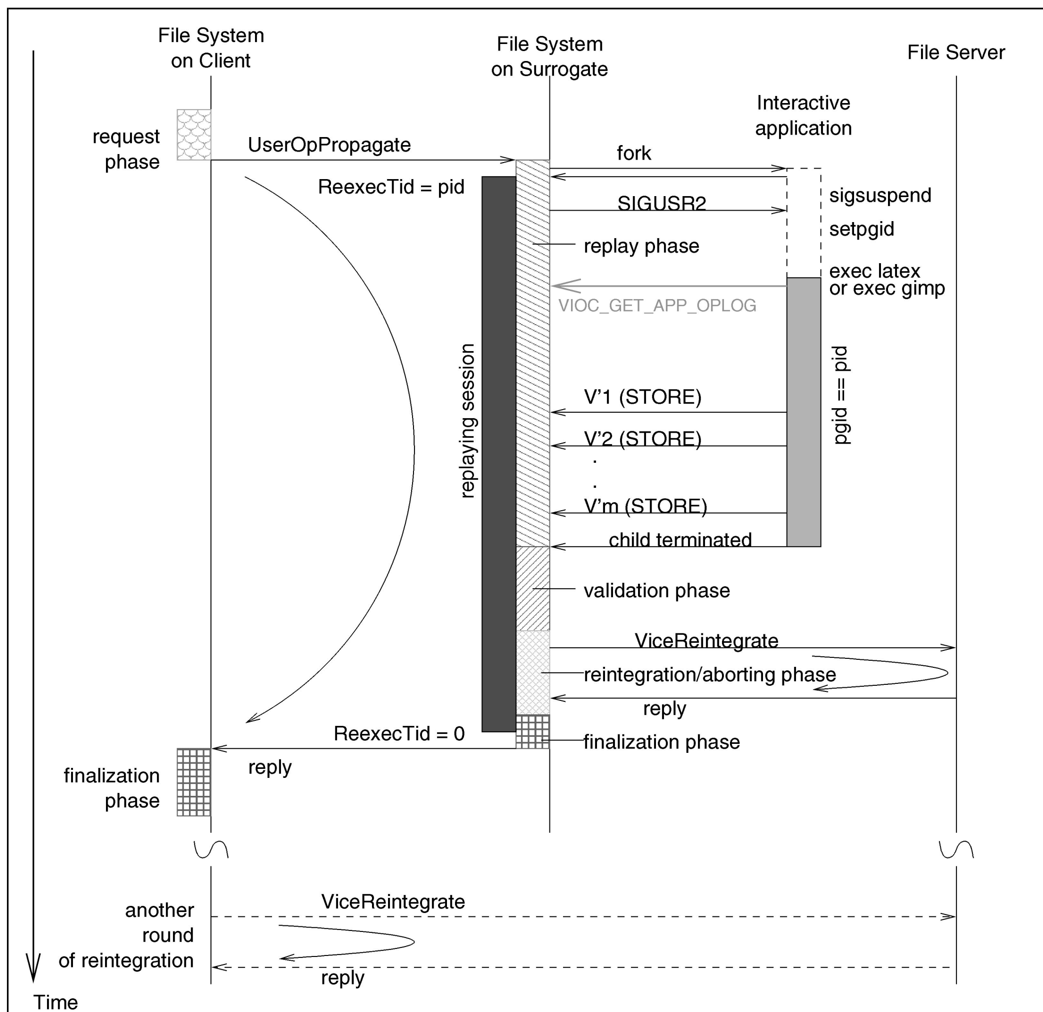


Fig. 6. Shipping stage. This figures shows the five phases of operation shipping: requesting, replaying, validation, reintegration/aborting, and finalization (shown as differently shaded bars). Also, shown in the lower part of figure is the fallback mechanism, which is needed only when the shipping of a user operation is not successful. Examples of the user operation being reexecuted are `latex` or `gimp`. Examples of file-system operations $V_1, V_2, \ldots, V_m$ are `STORE`, `RENAME`, and `MKDIR`. Note that the `VIOC_GET_APP_OPLOG` syscall applies to only application-aware operation shipping.

(Section 4.2). Finally, if a reexecution does not pass the validation procedure, the file system will fall back on value shipping. Note that, in this case, performance will suffer but correctness will not.

4.2 Validation

Validation is the procedure for a surrogate to ensure that the CML records resulting from the replaying of a user operation are identical to their counterparts on the client. The surrogate receives a copy of all the client records, except `STORE` records, for which only their fingerprints (defined below) are received. The surrogate then compares its own set of records with that of the surrogate. A reexecution is validated if and only if the two sets of records match each other. For `STORE` records, only their fingerprints are compared.

A fingerprint function is also known as a one-way hash function. It produces a fixed-length fingerprint $f(M)$ for a given arbitrary-length message M. In our application, the content of a file is the messages for which a fingerprint is computed. A good fingerprint function should have two properties: 1) computing $f(M)$ from M is easy and 2) the probability $P_{collision}$ that another message M', $M' \neq M$, will give the same fingerprint is small. Our file system employs MD5 (Message Digest 5) fingerprints. Each fingerprint has 128 bits, so the overhead is very small. Also, the probability $P_{collision}$ is very small and is in the order of $1/2^{64}$ [29], [34].

4.3 Nonrepeating Side Effects

In the early stage of this project, we expected the reexecutions of all target applications would repeat their original executions since we focus only on applications that perform deterministic tasks. To our surprise, we found that some of our target applications actually exhibit *nonrepeating side effects*. Fortunately, as discussed in the next two sections, we find that there are techniques to handle these side effects and, thus, we can still use operation shipping with these applications. Note that these handling techniques are done on a best-effort basis only. That is, if they fail, the file system simply falls back on value shipping. Note also that we do not claim that we can handle all types of side effects, but we can indeed handle the two common ones that we found. In effect, we think the principle illustrated here is the following: If a surrogate finds that a reexecution is not repeating, before it

gives up and falls back to value shipping, it should first try to fix the reexecution discrepancies.

4.4 Nonrepeating Side Effects Due to Time Stamps

Some applications (such as `rp2gen`, `ar`, and LaTeX) put time stamps into the files that they produce. These time stamps cause trouble to the validation of reexecutions because a regenerated file will have a few bytes different from the original. To maintain correctness of update propagation, one naive solution is to reject the reexecution.

However, we found that we can avoid naive rejections by viewing the changed bytes as if there were transmission "errors." With such a view, we use the existing technique of forward error correction (FEC) [10], [8] to restore the discrepancies.[4] The client precomputes FEC parity blocks and sends them to the surrogate. Upon reexecution, if the surrogate finds that a regenerated file does not give the same fingerprint as its counterpart, it invokes a FEC procedure: The parity block is the one precomputed by the client, and the data block is the regenerated file. For many cases, FEC can restore the reexecution discrepancies, and restore the file to the client's version.

The beauty of the technique is that the file system does not need to know the exact location of the time stamps embedded. Also, the file system can always fall back to value shipping if the FEC procedure cannot restore the discrepancies (for example, when the time stamps have too many bytes). Note that this is a novel use of FEC: while traditionally FEC is used for correction of communication errors, here we use it to restore reexecution discrepancies. In other words, while traditionally the data blocks are sent together with the parity blocks, here the data blocks are regenerated by reexecutions.

The additional network traffic due to the error correction code is quite small. In our implementation, we use Reed-Solomon code, for which the parameters can be chosen according to the desired error-correction capability. We choose to use a symbol size of 16 bits (2 bytes), and a correction capability of 16 errors (32 bytes); therefore, each block has 65,503 data symbols (131,006 bytes) and 32 parity symbols (64 bytes). The overhead is thus $\frac{32}{65503} = 0.049\%$. Reed-Solomon code fits our purpose well, but it has a weakness: It cannot correct discrepancies that change length (which may happen, for example, when timestamps are represented as human readable strings such as "`9:17`" or "`10:17`"). We still favor it over other algorithms, such as the rsync algorithm [36] (which can handle length change), since it has a smaller overhead on network traffic.

4.5 Nonrepeating Side Effects Due to Temporary Files

Some applications, for example `ar`, use temporary files in their executions. At the end of executions, some of these files are not deleted but only renamed. Fig. 7 shows the CML records on a client and a surrogate after two executions of a hypothetical user operation "`ar rv libsth.a foo.o bar.o`," which builds a library file `libsth.a` from two object modules `foo.o` and `bar.o`. Here, `ar` uses two temporary files `sta09395` and `sta16294`, whose name are generated pseudorandomly.

4. We are indebted to Matt Mathis of Pittsburgh Supercomputer Center for suggesting this idea to us.

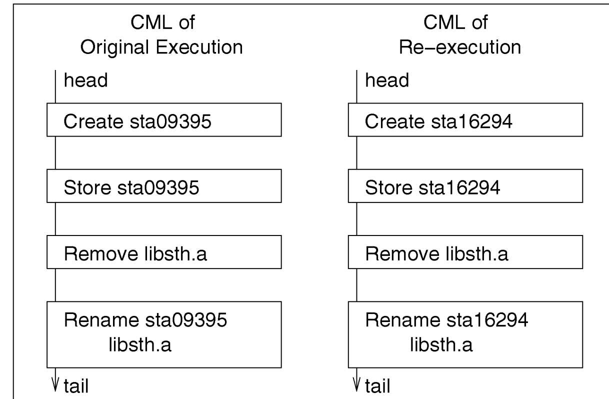


Fig. 7. CMLs of two executions of `ar`. The client-modify logs of two executions of the application `ar`. The log is implemented as a list of entries, each of which records some low-level file-system operations performed on a client. In this example, the operations are `Create`, `Store`, `Remove`, `Rename`, and they operate on files named `sta09395`, `sta16294`, and `libsth.a`.

To maintain correctness of update propagation, our file system might have to reject the reexecution since the CML records are different. However, again, we can avoid this naive rejection by noting that the difference is only in the intermediate states. That is, the temporarily files are both renamed to `libsth.a` at the end of the executions, so the final states of the file systems will actually be the same. With this observation, we add a procedure of temporary-file renaming to compensate for the intermediate nonrepeating side effects. In the procedure, the surrogate scans the sets of records and identifies all the temporary files by noting that they are created and, subsequently, renamed within a user operation. It renames the temporary files of the surrogate using the respective names chosen by the client. In our `ar` example, the temporary file `sta16294` will be renamed to `sta09395`.

Before we conclude this section, let us explain why we choose to validate at the CML level. Some may suggest that we can validate after CML records are applied on the servers (i.e., after reintegration) and then by comparing the final file-system states—conceivably this alternative approach would incur less false negatives and avoid those we have seen above. There are two reasons behind our design. First, CML records capture the essence of the changes to be made on the file-system states, and comparing them is easy. Second, and more importantly, we need to keep the option of aborting an operation-shipping transaction should the validation fail. Aborting these transactions can be done more easily before reintegration—we can simply discard those CML records on the surrogate.

5 Evaluation

We have performed two sets of controlled experiments for the application-transparent and application-aware cases respectively. The first set has been reported before ([18]), so we may omit some details for the first set and focus more on the second set.

5.1 Experimental Setup

The two sets of experiments were performed in two different time periods, so the hardware used was slightly different. For

Test	Nature	App	L_v Kbytes	L_{op} Kbytes	L_v/L_{op}
T1	rp2gen callback.rpc2	$\surd^{(1)}$	28.7	2.0	14.4
T2	rp2gen adsrv.rpc2	$\surd^{(1)}$	77.5	1.9	40.8
T3	yacc parsepdb.yacc	$\surd$	23.7	1.0	23.7
T4	c++ -c counters.cc -o counters.o	$\surd$	27.1	1.9	14.3
T5	c++ -c pdlist.cc -o pdlist.o	$\surd$	63.4	1.8	35.2
T6	c++ -c fso_daemon.cc -o fso_daemon.o	$\surd$	266.3	2.0	133.2
T7	c++ parserecdump.o -o parserecdump	$\surd$	23.9	2.0	12.0
T8	ar rv libdir.a ...	$\surd^{(1,2)}$	70.2	1.9	36.9
T9	ar rv libfail.a ...	$\surd^{(1,2)}$	364.0	2.2	165.5
T10	tar xzvf coda-doc-4.6.5-3-ppt.tgz	$\surd$	271.8	4.7	57.8
T11	make coda (in coda-src/blurb)	$\surd$	71.6	2.3	31.1
T12	make coda (in coda-src/rp2gen)	$\surd$	242.0	5.9	41.0
T13	tar cvf update.tar ...	$\surd$	60.2	1.0	60.2
T14	sgml2latex guide.sgml	$\surd$	42.0	1.0	42.0
T15	sgml2latex rvm_manual.sgml	$\surd$	270.3	1.1	245.7
T16	latex usenix99.tex	$\surd^{(1)}$	94.1	1.4	67.2

Fig. 8. Application-transparent operation shipping: traffic reduction. Sixteen tests were run using nine applications with real-life files. The column labeled App indicates the applicability of operation shipping to these tests: $\surd$ means applicable, $\times$ otherwise. (In this test set, operation shipping was applicable to all tests.) If there are numbers in bracket in the same column, they mean techniques for handling nonrepeating side effects were needed—1) for time stamps (Section 4.4) and 2) for temporary files (Section 4.5). The column labeled L_v and L_{op} show the network traffic, in Kbytes, of value and operation shipping, respectively; the final column L_v/L_{op} shows the traffic reduction.

the first set, the client, the surrogate, and the server machine were a Pentium 90MHz, a Pentium MMX 200MHz, and a Pentium 90MHz machine respectively, all running Linux kernel 2.0.35. For the second set, they were a Pentium MMX 200MHz, a Pentium II 300MHz, and a Pentium 90MHz respectively, and were running Linux kernel version of 2.2.5, 2.2.5, and 2.0.34 respectively. The network between the surrogate and the server was a 10-Mbps Ethernet. The network bandwidth between the remote client and the surrogate varied in different tests, and we used the Coda failure emulation package (`libfail` and `filcon`) [32] to emulate different network bandwidths on a 10-Mbps Ethernet.

As listed in Figs. 8 and 9, there were 16 tests, $T1, T2, \cdots, T16$, in the first test set, and 11 tests, $T30, T31, \cdots, T40$, in the second. Each of them represented a certain user task and comprises of a group or a single user operations. Each test was repeated three times. We are mainly interested in three aspects of operation shipping: applicability, reduction of network traffic, and speedup. These will be discussed one by one in the following sections.

5.2 Applicability

When the file system can use a user operation to ship an update, we say operation shipping is *applicable* to the user operation. We are interested to know the applicability with common user operations. However, we anticipate that there are cases when operation shipping is not applicable to some user operations. These can happen when a user operation does not repeat on the surrogate, yet its nonrepeating side effects cannot be restored by techniques like those in Sections 4.4 and 4.5. For example, we do not expect that operation shipping will work with the `-j <n>` mode of `GNU make`, which runs `n` jobs in parallel.

In our tests, for the application-transparent case, operation shipping was applicable to all user operations, although three applications did exhibit nonrepeating side effects, which were restored by our handling techniques (Fig. 8). For the application-aware case, operation shipping was applicable to all user operations except one in $T40$, which involved a function `blur` (Fig. 9). The function used current time value as a random seed and produced globally different images upon reexecution. We could have easily fixed the nonrepeating behavior by modifying the interface of the function, but we chose not to do so since the function serves well as an illustration on a limitation of operation shipping. We thus continued our experiments with $T40$ dropped.

5.3 Network Traffic Reduction

We measured the traffic required for propagating the update by value shipping and by operation shipping (L_v and L_{op}). Both the file data and the overhead were included in the traffic. In particular, for operation shipping, all fields in the operation logs: command, command-line arguments, current working directory, environment list, file-creation mask, metadata, fingerprints, and FEC parity blocks, were all counted towards the traffic. We list the result in Figs. 8 and 9, which also show the traffic reduction L_v/L_{op}. They show that operation shipping can achieve very substantial traffic reduction. For the application-transparent case, the highest reduction factor was 245.7 ($T15$); the smallest reduction was 12 ($T7$). For the application-aware case, the corresponding numbers were 396.6 ($T34$) and 33.9 ($T39$).

5.4 Speedup

Since the elapsed time of update propagation depends heavily on the network bandwidth, it was measured under three different network bandwidths: 9.6, 28.8, and 64.0 kilobits per second. We denote the elapsed time as T_v and T_{op} for value shipping and operation shipping. They were the end-to-end measurements for completing a round of update propagation using the respective approach. Specifically, T_{op} included the time needed for shipping the operation

Test	Nature	App	L_v Kbytes	L_{op} Kbytes	L_v/L_{op}
T30	Embossment load, emboss, save	√	244.0	2.1	116.2
T31	Annotation load, text, move, anchor, save	√	184.8	2.3	80.3
T32	Color inversion load, invert, save	√	128.9	2.1	61.4
T33	Color adjustment load, color_bal, bright_contr, save	√	261.1	2.3	113.5
T34	BMP conversion load, save as bmp	√	951.8	2.4	396.6
T35	Gradient Map load, choose map, gradient map, save	√	118.4	2.1	56.4
T36	Canvas load, canvas, save	√	305.4	2.2	138.8
T37	Mosaic load, tile, save	√	284.4	2.3	123.7
T38	Oil Painting load, oil_painting, save	√	350.4	2.2	159.3
T39	Poster load, posterize, save	√	71.2	2.1	33.9
T40	Blurring load, blur, save	×	—	—	—

Fig. 9. Application-aware operation shipping: traffic reduction. The 11 tests selected for evaluating the performance for application-aware operation shipping. User operations invoked in each test are shown in smaller print. The column labeled App indicates the applicability of operation shipping to these tests: $\surd$ for applicable, $\times$ otherwise. The columns labeled L_v and L_{op} show the network-traffic, in Kbytes, of value and operation shipping, respectively; the final column L_v/L_{op} shows the traffic reduction.

log, reexecution, MD5 computations, possible FEC procedures, validation, final reintegration between the surrogate and the server, etc. We also calculated the speedup T_v/T_{op}. Our results show that operation shipping can achieve very substantial speedups. For the application-transparent case, these ranged from 1.4 times to 26.3 times. (Due to space limitation, here we omit the individual numbers since they have been reported before [18]). For the application-aware case, the numbers are shown in Fig. 10. The speedups were the most substantial in the 9.6-Kbps network, where eight out of the 10 tests were accelerated by a factor exceeding 10, and the maximum speedup was 48.8 ($T34$) and the minimum 8.8 ($T39$). In the other two networks, the speedups ranged from a factor of 1.7 ($T30$, 64-Kbps) to 21.9 ($T34$, 28.8-Kbps).

In addition, operation shipping has another advantage: the elapsed time of update propagation can be much less sensitive to the network condition than that of value shipping. This can be seen by a closer examination of Fig. 10. For example, in $T34$, when the network bandwidth degraded from 64 Kbps to 9.6 Kbps, the elapsed time for value shipping increased almost proportionately (from 134 to 889 seconds), whereas the elapsed time for operation shipping is affected only slightly (from 14.6 to 18.2 seconds). This advantage, of course, comes from the fact that operation shipping causes much less network traffic, and it makes the performance of the file system more predictable under various network conditions.

6 Related Work and Alternative Solutions

6.1 Related Work

To the best of our knowledge, this is the first work that attempts to propagate file updates by operations. However, some general ideas and techniques resemble those used in some previous work.

6.1.1 Database

The idea of operation-based update propagation has been used in the database community [27]. However, our work is distinctive since the context is different. First, logging and shipping of operations in our case have to be done at a level higher than the low-level file-system operations (such as `open`, `write`, `close`), which are not compact enough for our purpose. Therefore, we focus on the level of *user operations*. Second, several new concepts are required in the new context: replaying of user operation on the surrogate, adjustment of status information, validation of replayed operations, and the handling of nonrepeating side effects, etc. We are also the first to use FEC to restore reexecution discrepancies. Third, since we always have a fall-back mechanism of value shipping, we can attempt operation shipping more aggressively than in other contexts.

6.1.2 Directory Operations

For directory operations, operation shipping is not new to Coda [14]. In fact, the Coda client-modify log can be viewed as an operation log for directories and a value log for files. When a directory is updated on a Coda client (e.g., when a new file is created), instead of shipping the whole new directory to the server, the client ships only the update operation (e.g., an insertion operation). Directory operations are more like database operations since they can be mapped directly to insertion, deletion, and modification of directory entries. Our work is distinctive since we apply operation shipping to file updates, which demands a number of new concepts as we have already seen.

Test	Name (Date size in Kbytes)	9.6-Kbps			28.8-Kbps			64-Kbps		
		T_v (s.d.)	T_{op} (s.d.)	T_v/T_{op}	T_v (s.d.)	T_{op} (s.d.)	T_v/T_{op}	T_v (s.d.)	T_{op} (s.d.)	T_v/T_{op}
T30	Embossment (243.8)	226,487 (613)	22,852 (52)	9.9	76,640 (325)	21,662 (353)	3.5	34,285 (74)	20,300 (2,028)	1.7
T31	Annotation (184.6)	171,485 (471)	8,333 (562)	20.6	58,267 (890)	6,767 (305)	8.6	25,927 (23)	6,064 (570)	4.3
T32	Color inversion (128.7)	119,853 (84)	7,650 (560)	15.7	40,566 (261)	6,223 (285)	6.5	18,090 (3)	5,701 (299)	3.2
T33	Color adjustment (260.9)	242,607 (177)	11,757 (590)	20.6	82,303 (383)	10,395 (568)	7.9	36,679 (65)	10,377 (273)	3.5
T34	BMP conversion (951.6)	889,491 (11,550)	18,228 (1,550)	48.8	317,380 (32,531)	14,467 (16)	21.9	134,009 (608)	14,587 (1,172)	9.2
T35	Gradient Map (118.2)	113,546 (5,198)	7,830 (591)	14.5	37,278 (468)	6,253 (591)	6.0	16,670 (67)	5,728 (583)	2.9
T36	Canvas (305.2)	286,422 (5,124)	10,052 (315)	28.5	95,924 (150)	8,418 (269)	11.4	42,881 (65)	8,024 (489)	5.3
T37	Mosaic (284.2)	263,713 (356)	16,876 (306)	15.6	90,062 (380)	16,013 (870)	5.6	39,888 (9)	15,646 (286)	2.5
T38	Oil painting (350.2)	324,745 (156)	19,519 (41)	16.6	110,228 (23)	18,051 (300)	6.1	49,167 (44)	17,417 (154)	2.8
T39	Poster (71.0)	66,179 (144)	7,506 (281)	8.8	22,531 (69)	5,788 (14)	3.9	10,059 (86)	5,260 (7)	1.9

Fig. 10. Application-aware operation shipping: elapsed time and speedup. Each test is performed in three different network conditions: networks with bandwidth of 9.6, 28.8, and 64.0 Kbps, respectively. For each network condition, there are three columns of data: the elapsed time for update propagation using value shipping (T_v) and application-aware operation shipping (T_{op}), and the speedup (T_v/T_{op}). All time measurements are shown in milliseconds. Each test/network combination was repeated three times, and the standard deviation (s.d) is shown in a parenthesis underneath each time measurement. The data size of each test involved is shown in a parenthesis underneath the name of the test.

6.1.3 Reexecutions

Also, several previous research projects have made extensive uses of reexecutions for different purposes, such as fault tolerance, load balancing, and consistency guarantees. For fault tolerance, a Unix process P can be backed up by another process P_b. If P crashes, then P_b will repeat the execution of P from a recent checkpoint, and will thereafter assume the role of P [2]. For load balancing, a Unix process can migrate to another host to reduce the load imposed on the original host [4]. For consistency guarantees, a previous Coda project proposed the notion of Isolation-Only Transaction. Users can delimit portions of executions using this notion. When an update conflict happens, Coda will reexecute the transaction [19], [20] to resolve the conflicts. Our work is different to these previous works in the specific goals and contexts.

6.2 Delta Shipping

To reduce the network traffic for shipping a file, sometimes we can ship only the incremental difference, also called the *delta*, between different versions of a file. This is the idea behind utilities and algorithms such as `diff`, which works for text files, and `rsync`, which works for binary files [36], or even file systems such as LBFS ([26], [9]). It is also used in web proxies [23], file archives [22], and source-file repositories [35], [28].

However, delta shipping has several limitations. First, newly created files have no previous version (or we can say the delta of a newly created files is as big as the whole file). Second, the effectiveness of delta shipping largely depends on how similar the two versions of a file are, and how those incremental differences are distributed in the file. In pathological cases, a slightly changed file may need a huge delta. This can happen, for example, when there is an global substitution of string in a text file, or when there is a global brightness or contrast adjustment in an image file. In general, we believe operation shipping can achieve a larger reduction of network traffic.

Having that said, we believe that delta shipping can complement our technique and improve the performance of value shipping, which is still an essential mechanism of any file system.

6.3 Data Compression

Data compression reduces the size of a file by taking out the redundancy in the file. This technique has been used in file systems [9], [1] and web proxies [23]. In general, however, the reduction factors achieved by data compression are smaller than those of operation shipping. This is because the former operates generically, while the latter exploits high-level information of user operations. For example, when we compressed the traffic for value shipping L_V in Fig. 8 using the popular `gzip` utility, which uses the Lempel-Ziv coding (LZ77), we could reduce the traffic by a factor of 2.7 to 8.1 (see [18]). These reductions were good but not as substantial as those achieved by operation shipping, which ranged from 12.0 to 245.7 times. Nevertheless, like delta shipping, data

compression can complement operation shipping and enhance the performance of value shipping in a file system.

6.4 Operation Shipping without Involving the File System

Can we use operation shipping without involving the file system? We can imagine a shell that logs every command a user types, and, without involving the file system, the shell remotely executes the same commands on a surrogate machine. We believe, however, such a system will suffer from severe limitations. First, if the file system had no knowledge that the second execution was a reexecution, it would treat the files produced by the two executions as two distinct copies, and would force the client to fetch the surrogate copy. Second, it might even think that there was an update/update conflict. Finally, it cannot ensure the correctness of the reexecution using the procedures described in Section 4. In our opinion, operation shipping must involve the file system.

7 Conclusion

This paper reports our experience with operation shipping for both the application-transparent and the application-aware cases. We have implemented a prototype. The main component of the system is an extended version of the Coda File System. We also make minor extensions to the Bourne Again Shell and the GIMP. They demonstrate that, to participate in operation shipping, existing noninteractive applications need no modification, and existing interactive applications need only minor modifications. We have also evaluated the prototype in controlled experiments and demonstrated that operation shipping can achieve substantial performance improvements.

In a broader sense, we carry out our work with the following philosophy. When mobile computers are at the mercy of weak connectivity, it is the file system rather than the users who should adapt to the environment. In our context, without an efficient update-propagation scheme, users would have to adapt their behaviors to the unpleasant weak network environment. For example, a mobile user would choose to work on a local file system rather than a distributed file system (and he would manually copy files over when needed). Our goal is to save users from these troubles. The ideal of mobile computing is to let users to carry out their work everywhere, without having to worry about the constraints imposed by the environments. With this work, we hope we are one more step closer to this ideal.

Acknowledgments

The authors would like to thank Matt Mathis for giving them the idea of using FEC for handling the side effect of time stamps, and Phil Karn for allowing them to incorporate his Reed Solomon Code implementation in their prototype. This work is, of course, built upon the enormous work done by other current and past members of the Coda File System group. They would also like to thank Robert Baron, Peter Braam, Maria Ebling, David Eckhardt, Jan Harkes, James Jay Kistler, Puneet Kumar, Qi Lu, Hank Mashburn, Lily Mummert, Brian Noble, Henry Pierce, Josh Raiff, and David Steere for their great effort. They also thank the anonymous reviewers for their input and comments. This research was partially supported by the Defense Advanced Research Projects Agency (DARPA), Air Force Material Command, USAF under agreement number F19628-96-C-0661, the Intel Corporation, and the Novell Corporation. The views herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA, Intel, Novell, or the US government.

References

[1] D. Bachmann, P. Honeyman, and L. Huston, "The Rx Hex," *Proc. First IEEE Workshop Services in Distributed and Networked Environments,* June 1994.

[2] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle, "Fault Tolerance Under UNIX," *ACM Trans. Computer Systems,* vol. 7, no. 1, Feb. 1989.

[3] P.J. Braam, "The Coda Distributed File System," *Linux J.,* June 1998.

[4] F. Douglis and J. Ousterhout, "Transparent Process Migration: Design Alternatives and the Sprite Implementation," *Software-Practice and Experience,* vol. 21, no. 8, pp. 757-785, Aug. 1991.

[5] *Free Software Foundation. BASH—The Official Web Site,* http://www.fsf.org/software/bash/bash.html. 2002.

[6] http://www.gimp.org. 2002.

[7] *Coda Group, Coda File System—The Official Web Site,* http://coda.cs.cmu.edu. 2002.

[8] A. Houghton, *The Engineer's Error Coding Handbook.* Chapman & Hall, 1997.

[9] K. Scott, "Review: AirSoft's AirAccess Keeps Everything in Sync," *Network Computing,* vol. 6, no. 7, June 1995, also available from http://www.networkcomputing.com/607/607rev3.html.

[10] P. Karn, *Error Control Coding, a Seminar handout,* available from http://people.qualcomm.com/karn/dsp.html. 2002.

[11] J.J. Kistler, "Disconnected Operation in a Distributed File System," PhD thesis, Carnegie Mellon Univ., School of Computer Science, 1993.

[12] J.J. Kistler and M. Satyanarayanan, "Disconnected Operation in the Coda File System," *ACM Trans. Computer Systems,* vol. 10, no. 1, Feb. 1992.

[13] L. Kleinrock, "Nomadic Computing—An Opportunity," *Computer Comm. Rev.,* vol. 25, no. 1, Jan, 1995.

[14] P. Kumar and M. Satyanarayanan, "Log-Based Directory Resolution in the Coda File System," *Proc. Second Int'l Conf. Parallel and Distributed Information Systems,* Jan 1993.

[15] O.S. Kylander and K. Kylander, *GIMP: The Official Handbook,* The Coriolis Group, Also available from http://manual.gimp.org, 1999.

[16] Y.W. Lee, *Prototypes for Operation Shipping - Download Location,* http://www.cse.cuhk.edu.hk/~clement/source_code/, 2002.

[17] Y.W. Lee, "Operation-Based Update Propagation in a Mobile File System," PhD thesis, The Chinese Univ. of Hong Kong, Dept. of Computer Science and Eng., Jan. 2000.

[18] Y.W. Lee, K.S. Leung, and M. Satyanarayanan, "Operation-Based Update Propagation in a Mobile File System" *Proc. USENIX 1999 Ann. Technical Conf.,* June 1999.

[19] Q. Lu, "Improving Data Consistency for Mobile File Access Using Isolation-Only Transaction," PhD thesis, Carnegie Mellon Univ., School of Computer Science, May 1996.

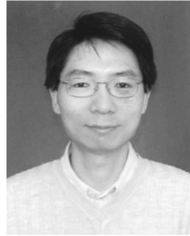
[20] Q. Lu and M. Satyanarayanan, "Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions," *Proc. Fifth IEEE HotOS Topics Workshop,* May 1995.

[21] Q. Lu and M. Satyanarayanan, "Resource Conservation in a Mobile Transaction System," *IEEE Trans. Computers,* vol. 46 no. 3, Mar. 1997.

[22] J. MacDonald, "Versioned File Archiving, Compression, and Distribution," submitted for the Data Compression Conf., an earlier version is available from http://www.XCF.Berkeley.edu/jmacd/xdelta.html. 1998.

[23] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential Benefits of Delta Encoding and Data Compression for HTTP," *Proc. ACM SIGCOMM '97,* 1997.

[24] L.B. Mummert, "Exploiting Weak Connectivity in a Distributed File System," PhD thesis, Carnegie Mellon Univ., School of Computer Science, 1996.
[25] L.B. Mummert, M.R. Ebling, and M. Satyanarayanan, "Exploiting Weak Connectivity for Mobile File Access," *Proc. 15th ACM Symp. Operating Systems Principles,* Dec. 1995.
[26] A. Muthitachareon, B. Chen, and D. Mazières, "A Low-Bandwidth Network File System," *Proc. 18th ACM Symp. Operating Systems Principles (SOSP '01),* Oct. 2001.
[27] K. Patersen, M.J. Spreitzer, D.B. Terry, M.M. Theimer, and A.J. Demers, "Flexible Update Propagation for Weakly Consistent Replication," *Proc. 16th ACM Symp. Operating Systems Principles,* Oct. 1997.
[28] *The FreeBSD Documentation Project. CVSup: in FreeBSD Handbook,* available from http://www.freebsd.org/handbook/cvsup.html, 2002.
[29] R. Rivest, *The MD5 Message-Digest Algorithm, Internet RFC 1321,* available from http://theory.lcs.mit.edu/~rivest/publications.html, Apr. 1992.
[30] M. Satyanarayanan, "The Evolution of the Coda File System," *ACM Trans. Computer Systems,* vol. 20, no. 2, pp. 85-124, May 2002.
[31] M. Satyanarayanan, "Fundamental Challenges in Mobile Computing," *Proc. Fifteenth ACM Symp. Principles of Distributed Computing,* May 1996.
[32] M. Satyanarayanan, M.R. Ebling, J. Raiff, P.J. Braam, and J. Harkes, "Coda File System User and System Administrators Manual," School of Computer Science, Carnegie Mellon Univ., available from http://www.coda.cs.cmu. edu/doc/html/index.html. 2000.
[33] M. Satyanarayanan, J.J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere, "Coda: A Highly available File System for a Distributed Workstation Environment," *IEEE Trans. Computers,* vol. 39, no. 4, Apr. 1990.
[34] B. Schneier, *Applied Cryptography,* second ed. John Wiley & Sons, Inc., 1996.
[35] *Cyclic Software. Concurrent Versions System (CVS),* available from http://www.cyclic.com/, 2002.
[36] A. Tridgell and P. Mackerras, "The RSYNC Algorithm," Technical Report TR-CS-96-05, The Australian Nat'l Univ., available from http://samba.anu.edu.au/rsync/, June 1996.



Yui-Wah Lee received the BSc and the MPhil degrees both in electrical and electronic engineering, from the University of Hong Kong, and the PhD degree in computer science and engineering from the Chinese University of Hong Kong in 2000. He is currently a member of the technical staff in the Networking Research Laboratory at Bell Laboratories Research of Lucent Technologies, Inc, and was a visiting scholar and a member of the Coda File System research group at Carnegie Mellon University. Dr. Lee's research interests are system design and implementation in general, and mobile computing, and networking in particular. He is a member of the IEEE.

Kwong-Sak Leung (M'77-SM'89) received the BSc in engineering and PhD degrees in 1977 and 1980, respectively, from the University of London, Queen Mary College. He worked as a senior engineer on contract R&D at ERA Technology and later joined the Central Electricity Generating Board to work on nuclear power station simulators in England. He joined the Computer Science and Engineering Department at the Chinese University of Hong Kong in 1985, where he is currently professor and chairman of the department. Dr. Leung's research interests are in soft computing, data and knowledge engineering, and software systems. He has published more than 160 papers and two books. He has been a chair and a member of many program and organizing committees of international conferences. He is on the Editorial Board of *Fuzzy Sets and Systems* and an associate editor of International *Journal of Intelligent Automation and Soft Computing*. He is a senior member of the IEEE, a chartered engineer, a member of IEE and ACM, and a fellow of HKCS and HKIE.

Mahadev Satyanarayanan received the Bachelor's and Master's degrees from the Indian Institute of Technology, Madras, India, and the PhD in computer science from Carnegie Mellon. He is an experimental computer scientist who has pioneered research in the field of mobile information access. One outcome of this work is the Coda File System, which supports disconnected and bandwidth-adaptive operation. Key ideas from Coda have been incorporated by Microsoft into the IntelliMirror component of Windows. Another outcome is Odyssey, a set of open-source operating system extensions for enabling mobile applications to adapt to variation in critical resources such as bandwidth and energy. Coda and Odyssey are building blocks in Project Aura, a research initiative at Carnegie Mellon to build a distraction-free ubiquitous computing environment. Earlier, Satyanarayanan was a principal architect and implementor of the Andrew File System, which was commercialized by IBM. Dr. Satyanarayanan is the Carnegie Group Professor of Computer Science at Carnegie Mellon University. He is currently on partial sabbatical, serving as the founding director of Intel Research Pittsburgh. He is the founding Editor-in-Chief of *IEEE Pervasive Computing*. He is a fellow of the IEEE and a member of the IEEE Computer Society.

▷ **For more information on this or any computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.