Operation-based Update Propagation in a Mobile File System *

Yui-Wah Lee Kwong-Sak Leung
The Chinese University of Hong Kong
{clement,ksleung}@cse.cuhk.edu.hk

Mahadev Satyanarayanan
Carnegie Mellon University
satya+@cs.cmu.edu

Abstract

In this paper we describe a technique called *operation-based update propagation* for efficiently transmitting updates to large files that have been modified on a weakly connected client of a distributed file system. In this technique, modifications are captured above the file-system layer at the client, shipped to a surrogate client that is strongly connected to a server, re-executed at the surrogate, and the resulting files transmitted from the surrogate to the server. If re-execution fails to produce a file identical to the original, the system falls back to shipping the file from the client over the slow network. We have implemented a prototype of this mechanism in the Coda File System on Linux, and demonstrated performance improvements ranging from 40 percents to nearly three orders of magnitude in reduced network traffic and elapsed time. We also found a novel use of forward error correction in this context.

1 Introduction

The use of a distributed file system on a mobile client is often hindered by poor network connectivity. Although *disconnected operation* [8] is feasible, a mobile client with an extensive amount of updates should not defer propagating them to a server for too long. Damage, theft or destruction of the client before update propagation will result in loss of those updates. Further, their timely propagation may be critical to successful collaboration and in reducing the likelihood of update conflicts.

Propagation of updates from a mobile client is often impeded by *weak connectivity* in the form of wireless or wired networks that are low-bandwidth or intermittent. Aggressive update propagation under these conditions increases the demand on scarce bandwidth. A major component to bandwidth demand is the shipping of large files in their entirety. Two obvious solutions to the problem are *delta shipping* and *data compression*. The former tries to ship only the incremental differences between versions of files, and the latter "compresses out" the redundancies of files before shipping the files. Unfortunately, as discussed later in this paper, both of these methods have shortcomings that limit their usefulness.

In this paper, we focus on a radically different solution, which is called *operation-based update propagation* (or *operation shipping* for brevity). It is motivated by two observations. First, large files are often created or modified by *user operations* that can be easily intercepted and compactly represented. Second, the cost of shipping and re-executing the user operations is often significantly smaller than that of shipping the large files over a weak network.

To propagate a file, the client ships the operation to a *surrogate client* that is well connected to the server, as shown in Figure 1. The surrogate re-executes the operation, validates that the re-generated file is identical to the original, and then propagates the file via its high-speed connection to the server. If the file re-generated by the surrogate does not match that from the original execution, the system falls back to shipping the original from the client to the server over the slow connection. This validation and fall-back mechanism is essential to ensure the *correctness* of update propagation.

*This research was partially supported by the Defense Advanced Research Projects Agency (DARPA), Air Force Materiel Command, USAF under agreement number F19628-96-C-0061, the Intel Corporation, and the Novell Corporation. The views and conclusions contained 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 U.S. Government.

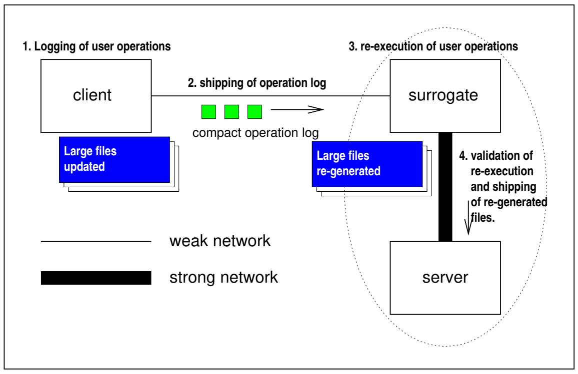

Figure 1: An overview of operation shipping

The use of a surrogate in our approach ensures that server scalability and security are preserved — servers are not required to execute potentially malicious and long-running code, nor are they required to instantiate the execution environments of the numerous clients they service. In our model, we assume each mobile client has pre-arranged for a suitable surrogate of an appropriate machine type, at an adequate level of security, possessing suitable authentication tickets, and providing an appropriate execution environment. This assumption is reasonable, since many users of mobile clients also own powerful and well connected personal desktops in their offices, and they can pre-arrange for these otherwise idle personal machines as the surrogates.

Even though the idea of operation shipping is conceptually simple, there are many details that have to be addressed to make it work in practice. In the rest of this paper, we describe how we handle these details by implementing a prototype based on the Coda File System. Our experiment confirms the substantial performance gains of this approach. Depending on the metric and the specific application, we have observed improvements ranging from a factor of three to nearly three orders of magnitude.

2 Coda background

We have implemented our prototype by extending the Coda File System [3]. Although our experience is based on Coda, the general principles should also be applicable to other mobile file systems. We briefly describe Coda in this section; more information is available in the literature [3, 8, 7, 14, 13].

The Coda model is that there are many clients and a few servers. On each client, a cache manager, called *Venus*, carefully manages and persistently stores cached file-system objects. To support mobile computing, Coda clients can be used in *disconnected* and *weakly connected* mode, where Venus emulates the servers and allows file-system operations on cached objects. Updates are applied immediately to locally cached objects, and are also logged in a *client-modify log (CML)*. The logging mechanism allows propagation of updates to servers to be deferred until a convenient time, such as when network connectivity is restored. Venus propagates these updates with a mechanism called *trickle reintegration* [14, 13]. When propagation is attempted, a prefix of the log is shipped to the server in temporal order, the size of the prefix being determined by available bandwidth.

The effect of each mutating file-system operation is represented as a record in the CML. For example, a `chmod` operation is logged as a `CHMOD` record, a `mkdir` operation is logged as a `MKDIR` record. Furthermore, if there have been some intervening `write` operations made to an `open`'ed file, a subsequent `close` operation will be logged as a `STORE` record.

Records of the type `STORE` are special, because the associated data include the contents of the files. Therefore, these records are much bigger than records of other types. `STORE` records can be as large as several kilobytes or even megabytes, whereas other records are typically smaller than a few hundred bytes. Although the contents of a file logically constitute a part of the CML, they are physically stored in a separate *container file* in the client's local file system.

Although trickle reintegration has been shown to be effective in decoupling the foreground file-system activities from the slow propagations of updates, it still suffers from an important limitation: updated files are propagated in their entirety. In other words, although the users perceive a fast response time from the file system, the actual propagations of the updates are very slow, and they generate heavy network traffic. On a weak network, we need a more efficient scheme for shipping updates.

3 Design and Implementation

3.1 Overview

We organize our discussion according to the four steps shown in Figure 1:

1. **Logging of user operations.** When a file is updated on a client, the client keeps the new file value V (the contents of the file) and also logs the user operation O.

2. **Shipping of operation log.** If the network bandwidth is low, the client does not ship V to the server. Instead, it ships O, the fingerprint of V, and other meta-data to a *surrogate* client.

3. **Re-execution of operations.** The surrogate re-executes O and produces a file value V'.

4. **Validation of re-execution.** The surrogate validates the re-execution by checking the fingerprints and meta-data. If it accepts the re-execution, then it will present V' to the server; otherwise, it will report failure to the client, which will then fall back to value shipping and ship V directly to the server.

3.2 Logging of user operations

3.2.1 Level of abstraction

We need to find the right level of abstraction for logging operations, such that the operation logs are compact. Currently, Venus logs only the low-level *file-system operations*. File-system operation logs are not compact enough for efficient operation shipping, since they contain the contents of the files.

On the other hand, computer users tend to think at a higher level. Typically, when they are working with a computer, they issue a series of *commands* to it. Since human beings type relatively slowly, the commands must be compact; hence, we focus on this level for operation shipping. When we speak of *user operations*, we mean the high level commands of computer users that can be intercepted, logged, and replayed later.

There must be an entity that intercepts the user operations and supplies the relevant information to the file system. This entity can be an interactive shell, or it can be the application itself. We refer to the two cases as *application-transparent* and *application-aware* logging.

3.2.2 Application-transparent logging

Application-transparent logging is possible if the application is *non-interactive*. In this case, an interactive shell can intercept the information related to the user operation. For example, operation logging can be transparent to the following types of applications (examples are listed in parentheses): compiler and linkers (`gcc, yacc,` and `ld`), text processors (`latex` and `troff`), file-format converters (`dvips` and `sgml2latex`), software-project management tools (`make`), and file packagers (`tar`).

For application-transparent operation shipping, we need not modify the applications, we simply need to modify some interactive shells – the meta-applications – such as `bash` and `tcsh`. Fortunately, there are much fewer meta-applications than applications. We assume, of course, that the users are willing to use a modified shell to take advantage of operation shipping.

We extend the file-system API (application-programmer interface) with two new commands for the `ioctl` system call: `VIOC_BEGIN_OP` and `VIOC_END_OP`. They delineate the beginning and the end of a user operation. The user operation is tagged with the process group , so forked children in the same group are considered part of the same user operation. When the file system receives a file-system call, it can determine whether the call comes from a tagged user operation by examining the process-group information of the caller. The information necessary for re-execution, including the name of the user command, the command-line arguments, current working directory, and so on, is passed to the file system with the `VIOC_BEGIN_OP` command.

Our file system provides the logging mechanism, but the logging entities can choose the appropriate logging policies. For example, an interactive shell may allow users to specifically enable or disable operation logging based on certain criteria.

We have experimented with the `bash` shell, a common Unix shell. We added a few lines of code so that the modified shell issues the `VIOC_BEGIN_OP` and `VIOC_END_OP` commands appropriately. Currently, we implement the most straightforward policy: the shell logs every user operation. In the future, we may implement a more flexible policy.

3.2.3 Application-aware logging

Application-aware logging is needed when the application is interactive. In this case, the application is involved in capturing the user operations. The following types of applications fall into this category (examples are listed in parentheses): text editors and word processors (`emacs, vi, Applix Word` and `Microsoft Word`), drawing tools (`xfig` and `Corel Draw`), presentation software (`Applix Present` and `Microsoft PowerPoint`), computer-aided-design tools (AutoCAD and magic), and image manipulators (xv and GNU Image Manipulation Program).

In this paper, we focus on application-transparent operation shipping. The mechanism that we have designed, and the evaluation that we have performed, is limited to non-interactive applications. We plan to study application-aware operation shipping next, and we will report our findings in the future.

3.3 Shipping of operation log

3.3.1 Shipping mechanism

The *reintegrator*, which is a user-level thread within Venus, manages update propagation. It periodically selects several records from the head of the CML and ships them to the servers. For records with no user-operation information attached, the reintegrator uses value shipping and makes a ViceReintegrate remote procedure call (RPC) to the server. The server, when processing the RPC, *back-fetches* the related container files from the client. If the reply of the RPC indicates success, the reintegrator will locally commit the updates. Local commitment of updates is the final step of successful update propagation, and includes updating the states of relevant objects, such as version vectors and dirty bits, and truncating the CML.

If user-operation information is available for a record, the reintegrator will attempt operation shipping first. All the records associated with the same user operation will be operation shipped altogether. The reintegrator selects the records, packs the operation log, and makes a UserOpPropagate RPC to the surrogate. If the reply indicates success, the reintegrator will locally commit the updates. However, if the reply indicates failure, the reintegrator will set a flag in each of the records indicating that it has tried and failed propagation by operation shipping. These records will then be value shipped.

3.3.2 Cost model

The current version of our prototype attempts operation shipping for a record whenever there is user-operation information available. This *static* approach implicitly assumes that the connectivity between a mobile client and its servers is always weak. In real life, a mobile client may have strong connectivity occasionally. During that time, as explained in the following paragraphs, value shipping is more efficient than operation shipping. We plan to enhance our prototype so that mobile clients *dynamically* decide whether they should use operation shipping or value shipping.

Our cost model compares the costs of value shipping with that of operation shipping. For each case, there are two different costs involved: network traffic and elapsed time.

For value shipping, assuming the overhead is negligible, the network traffic is the total length L of the updated files, and the elapsed time is $T_v = L/B_c$, where B_c is the bandwidth of the network connecting the client to the server.

For operation shipping, the network traffic is the length of the operation log, L_{op}, and the elapsed time is T_{op}. The latter is composed of four components: (1) the time needed to ship the operation log (L_{op}/B_c), (2) the time needed for re-executing the operation (E), (3) the time needed for additional computational overhead (H_{op}) such as computing checksum information and encoding and decoding of forward-error-correction codes, and (4) the time needed to ship the updated files to the servers. There are two cases for the last component. If the re-execution passes the validation (accepted), the updated files will be shipped from the surrogate (the time cost will be L/B_s, where B_s is the bandwidth of the network connecting the surrogate to the server); if the re-execution fails the validation, the updated file will be shipped from the client (the time cost will be L/B_c). The following equation summarizes the time costs involved:

$$T_{op} = \begin{cases} L_{op}/B_c + E + H_{op} + L/B_s & \text{if accepted} \\ L_{op}/B_c + E + H_{op} + L/B_c & \text{if rejected} \end{cases} \quad (1)$$

Therefore, operation shipping is more favorable than value shipping only in certain condition. Operation shipping saves network traffic if the operation log is more compact than the updated files ($L_{op} < L$). Also, it speeds up the update propagation ($T_{op} < T_v$) if the following five conditions are true: (1) the re-execution is accepted, (2) the operation log is compact ($L_{op} \ll L$), (3) the re-execution is fast ($E \ll L/B_c$), (4) the time needed for additional computational overheads is small ($H_{op} \ll L/B_c$), and (5) the surrogate has a much better network connectivity than the client ($B_s \gg B_c$).

3.4 Re-execution of user operations

3.4.1 Re-execution mechanism

Although we anticipate most re-executions will be successful (execute completely and pass the validation), we have to prepare for the possibility that they may fail (cannot execute completely or fail the validation). Therefore, we have to ensure that re-executions are abortable transactions such that failed re-executions will have no lasting effect. We implement this as follows.

Upon receiving a `UserOpPropagate` RPC from the client, Venus on the surrogate will temporarily put the affected volume [1] into *write-disconnected* mode, and then re-executes the user operation via a spawned child called the *re-executor*. During the re-execution, since the volume is write-disconnected, input files can be retrieved from the servers, but file-system updates are not written immediately to the server. These updates are tagged with the identity of the re-execution and are logged in the CML. If the re-execution later passes the validation, the surrogate will re-connect the volume with the servers and reintegrate the updates. At the end, and only when the reintegration succeeds, the surrogate will locally commit the updates, and indicate a success to the client in a RPC reply. On the other hand, failures may happen any time during the re-execution, the validation, or the reintegration. If any such failures occur, the surrogate will discard all the updates and indicate a failure to the client in a RPC reply.

It is possible that a reintegration completes successfully at the servers, but the RPC response fails to arrive at the client in spite of retransmission. This can happen when there is an untimely failure of the surrogate or the communication channels. We make use of the existing Coda mechanism of ensuring atomicity of reintegrations [13] to handle this kind of failures. In short, the client presumes an reintegration has failed if it does not receive a positive response from the surrogate, and it will retry the reintegration. At the same time, the server retains the information necessary to detect whether a record has already been reintegrated earlier. If a client attempts such an improper retry, the server will reply with the error code `EALREADY` (Operation already in progress), and the client will then know that the records have already been successfully reintegrated in a previous attempt, and it will simply locally commit those records.

3.4.2 Facilitating repeating re-executions

A re-execution of a user operation is accepted when it produces a result that is identical to that of the original execution. We say the re-execution is *repeating* the original execution. Only these repeating re-executions are useful to operation shipping. We know that a deterministic procedure will produce the same result in different runs provided that it has the same input and the same environment in each run. Our file system makes use of this principle to facilitate repeating re-executions.

First, the re-executor runs with the four Unix-process attributes identical to that of the original execution: (1) the working directory, (2) the environment-variable list, (3) the command-line arguments, and (4) the file-creation mask [23].

Second, our file system expects that the surrogate machine has software and hardware configurations similar to the client machine. Two machines are said to be identically configured if they have the same CPU type, the same operating system, the same system-header files, the same system libraries, and any other system environments that can affect the outcomes of computations on the two machines.

Third, if a re-execution requires an input file stored in Coda, we can rely on the file system to ensure that the client and the surrogate will use an identical version of the input file. Coda can ensure this because a client ships updates to its servers in temporal order, and the surrogate will always retrieve the latest version of a file from the servers. For example, consider a user issuing three user operations successively on a client machine: (Op1) update a source file using an editor, (Op2) compile the source file into an object file using a compiler, and (Op3) update the source file again. When the surrogate re-executes Op2, the client must have shipped Op1 but not Op3, and the re-executor will see exactly the version updated by Op1.

We emphasize that our file system does not guarantee that all re-executions will be repeating their original executions; it just increases the probability of that happening. More importantly, it ensures that only repeating re-executions will be accepted. This is achieved by the procedure of validation (Section 3.5).

In the evaluation section, we will see that many applications exhibit repeating re-executions. Although some other applications exhibit non-repeating side effects during re-executions, these side effects can be handled in simple ways. Therefore, we believe we can use operation shipping with a large number of applications.

3.4.3 Status information

In addition to updating contents of files, user operations also update some meta-data (status information). Some of the meta-data are internal to the file system and are invisible to the users (e.g., every `STORE` operation has an identity number for concurrency control); some

[1] In Coda, a volume is a collection of files forming a partial subtree of the Coda name space.

are external and are visible to the users (e.g., the last-modification time of a file). In both cases, to make a re-execution's result identical to that of the original execution, the values of the meta-data of the re-execution should be reset to those of the original execution. Therefore, the client needs to pack these meta-data as part of the operation log, and the surrogate needs to adjust the meta-data of the re-execution to match the supplied values.

3.4.4 Non-repeating side effects

We focus on applications that perform deterministic tasks, such as compiling binaries or formatting texts, and exclude applications such as games and probabilistic search that are randomized in nature. In an early stage of this project, we expected the re-executions of these applications to repeat their original executions completely. However, we found that some common applications exhibited non-repeating side effects. So far we have found two types of such side effects: (1) time stamps in output files, and (2) temporary names of intermediate files. Fortunately, we are able to handle these side effects automatically, so we are still able to use operation shipping with these applications. We will discuss the handling methods in the two following subsections.

We also anticipate a third possible kind of side effect: external side effects. For example, if an application sends an email message as the last step of execution, then the user may be confused by the additional message sent by re-execution. To cope with this kind of side effect, we plan to allow users to disable the use of operation shipping for some applications.

3.4.5 Side effects due to time stamps

Some applications, such as `rp2gen`, `ar`, and `latex`, put time stamps into the files that they produce. `rp2gen` generates stubs routines for remote procedure calls, `ar` builds library files, and `latex` formats documents. They use time stamps for various reasons. Because of the time stamps, a file generated by a re-execution will differ slightly from the version generated by the original execution. Observing that only a few bytes are different, we can treat the changed bytes as "errors" and use the technique of forward error correction (FEC) [6, 4] to "correct" them. (We are indebted to Matt Mathis for suggesting this idea to us.)

Our file system, therefore, does the following. Venus on the remote client computes an error-correction code (we use the Reed-Solomon code) for each updated file that is to be shipped by operation, and it packs the code as a part of the operation log. Venus on the surrogate, after re-executing the operation, uses the code to correct the time stamps that may have occurred in the re-generated version of the file.

Note that this is a novel use of the Reed-Solomon code. Usually, data blocks are sent together with parity blocks (the error-correction code); but our clients send only the parity blocks. The data blocks are instead re-generated by the surrogate. Whereas others use the code to correct communication errors, we use it to correct some minor re-execution discrepancies. If a discrepancy is so large that our error-correction procedure cannot correct it, our file system simply falls back to value shipping. This entails a loss of performance but preserves correctness.

The additional network traffic due to the error correction code is quite small. We choose to use a (65535,65503) Reed-Solomon block code over $GF(2^{16})$. In other words, the symbol size is 16 bits, each block has 65,503 data symbols (131,006 bytes) and 32 parity symbols (64 bytes). The system can correct up to 16 errors (32 bytes) in each data block.

However, the additional CPU time due to encoding and decoding is not small. We discuss this in more detail in Section 4.4. Also, the Reed-Solomon code cannot correct discrepancies that change length (for example, the two timestamps "`9:17`" and "`14:49`" have different lengths). The rsync algorithm [24] can handle length change, but we favor the Reed-Solomon code because it has a smaller overhead on network traffic.

3.4.6 Side effects due to temporary files

The program `ar` is an example of an application that uses temporary files. Figure 2 shows the two CMLs on the client and the surrogate after the execution and re-execution of the following user operations:

```
ar rv libsth.a foo.o bar.o.
```

The user operation builds a library file `libsth.a` from two object modules `foo.o` and `bar.o`.

Note that `ar` used two temporary files `sta09395` and `sta16294` in the two executions. The names of the temporary files are generated based on the identity numbers of processes executing the application, and hence they are time dependent. Our validation procedure (Section 3.5) might naively reject the re-execution "because the records are different."

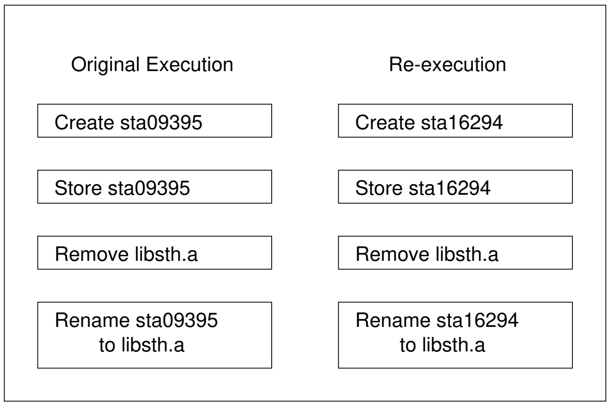

Figure 2: CMLs of two executions of `ar`

Temporary files appear only in the intermediate steps of the execution. They will either be deleted or renamed at the end, so their names do not affect the final file-system state. An application uses temporary files to provide execution atomicity. For example, `ar` writes intermediate computation results to a temporary file, and it will rename the file to the target filename only if the execution succeeds. This measure is to ensure that the target file will not be destroyed accidentally by a futile execution.

If a temporary file is created and subsequently *deleted* during the execution of a user operation, its modification records will be deleted by the existing *identity cancellation* procedure [7]. They will not appear in the two CMLs and will not cause naive rejections of re-execution.

However, if a temporary file is created and subsequently *renamed* during the execution of a user operation, its modifications records will be present in the CMLs, and might cause our validation to reject the re-execution. Our file system uses a procedure of *temporary-file renaming* to compensate for the side effect. This procedure is done after the re-executor has finished the re-execution and before the surrogate begins the validation.

The idea of the temporary-file renaming is to scan the two CMLs and identify all the temporary files as well as their final names. We identify temporary files by the fact that they are created and subsequently renamed in an user operation. For each temporary file used by the surrogate, our file system determines the temporary file name N used by the client in the original execution. It thus renames the temporary file to N. In our `ar` example, the temporary file `sta16294` will be renamed to `sta09395`.

3.5 Validation of re-executions

3.5.1 Validation mechanism

Validation is done after the handling of side effects, and the adjustments of status information. By that time, if a re-execution is repeating its original execution, the set of mutations incurred on the surrogate should be the same as that on the client. Since mutations are captured on CMLs, our file system can validate a re-execution by comparing the relevant portion of the CML of the surrogate to that of the client.

To facilitate the comparison, the client packs every record in the relevant portion of the CML as part of the operation log. However, the container files, which are associated with `STORE` records, are not packed; otherwise they would incur a heavy network traffic for shipping the operation log, amounting to the traffic needed for value shipping. Instead, the client packs the fingerprint of each container file. When comparing CMLs, the surrogate asserts that two container files are equal if they have the same fingerprint.

3.5.2 Fingerprint

A fingerprint function produces a fixed-length fingerprint $f(M)$ for a given arbitrary-length message M. A good fingerprint function should have two properties: (1) computing $f(M)$ from M is easy, and (2) the probability that another message M' gives the same fingerprint is small. For our purpose, the messages for which we find the fingerprints are the contents of the container files.

Our file system employs MD5 (Message Digest 5) fingerprints [17, 21]. Each fingerprint has 128 bits, so the overhead is very small. Also, the probability that two different container files give the same fingerprint is very small; it is in the order of $1/2^{64}$.

The fact that the probability is non-zero, albeit very small, may worry some readers. However, even value shipping is vulnerable to a small but non-zero probability of error. That is, there is a small probability that a communication error has occurred but is not detected by the error-correction subsystem of the communication channel. We believe people can tolerate the small probabilities of errors of both operation shipping and value shipping.

Test	Name	Nature	NF	Size (KB)	SE1	SE2
T1	rp2gen callback.rpc2	RPC2 stub generator	5	27.5	•	
T2	rp2gen adsrv.rpc2	RPC2 stub generator	5	76.3	•	
T3	yacc parsepdb.yacc	compiler compiling	1	23.5		
T4	c++ -c counters.cc -o counters.o	compiling	2	26.0		
T5	c++ -c pdlist.cc -o pdlist.o	compiling	2	62.4		
T6	c++ -c fso_daemon.cc -o fso_daemon.o	compiling	2	265.3		
T7	c++ parserecdump.o -o parserecdump	linking	1	23.0		
T8	ar rv libdir.a ...	library building	1	70.2	•	•
T9	ar rv libfail.a ...	library building	1	363.1	•	•
T10	tar xzvf coda-doc-4.6.5-3-ppt.tgz	extracting files	5	269.5		
T11	make coda (in coda-src/blurb)	compiling/linking	3	69.9		
T12	make coda (in coda-src/rp2gen)	compiling/linking	10	237.1		
T13	tar cvf update.tar ...	packaging files	1	60.2		
T14	sgml2latex guide.sgml	translator	1	41.8		
T15	sgml2latex rvm_manual.sgml	translator	1	270.0		
T16	latex usenix99.tex	text formatter	3	93.4	•	

We ran 16 tests using nine applications with some real-life files. For each test, we list the name, nature, the number of the files that were updated (NF), and the total size of the files. Some of the applications exhibited non-repeating side-effects due to time stamps (SE1) and temporary files (SE2), they are marked by bullet points (•) in the table.

Figure 3: Selected tests and applications

4 Evaluation

At this time, we do not sufficiently understand client usage patterns to accurately model overall performance improvement. However, we have selected a set of commonly used non-interactive applications that allow us to focus on the following three questions:

1. *Is operation shipping transparent to the applications?*
2. *What is the extent of network-traffic reduction that can be achieved by using operation shipping?*
3. *What is the extent of elapsed-time reduction that can be achieved by using operation shipping?*

We shall answer these questions in the following subsections, after we briefly describe the experimental setup.

4.1 Experimental setup

The client, the surrogate, and the server machine used in the experiments were a Pentium 90MHz, a Pentium-MMX-200MHz, and a Pentium 90MHz machine respectively. All three machines were running the Linux operating system (kernel version 2.0.35). 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`) [18] to emulate different network bandwidths on a 10-Mbps Ethernet.

We performed 16 different tests using nine common non-interactive applications (Figure 3). We used real-life input files, found in our environment, for the tests. We selected the tests such that the data size in each test was close to one of the three reference sizes: 16, 64, and 256 Kbytes. The data size is defined as the total size of the files updated by an operation. The 16 tests were labeled as $T1, T2, \cdots, T16$. Each test was repeated three times.

4.2 Transparency to applications

We make no claim that operation shipping can be used transparently with all non-interactive applications. For example, we anticipate that operation shipping probably cannot be used with the `-j <n>` mode of `GNU Make`, which runs `n` jobs in parallel. Fortunately, so far all nine selected applications can be used transparently with operation shipping. Three of them exhibit non-repeating side effects, but these side effects can be compensated by our handling techniques.

Test	Nature	Traffic by value-shipping (Kbytes) L_v	Traffic by operation-shipping (Kbytes) L_{op}	Traffic reduction by operation-shipping L_v/L_{op}	Expected traffic by data-compression (Kbytes) $L_{v,gz}$	Expected traffic reduction by data compression $L_v/L_{v,gz}$
T1	rp2gen	28.7	2.0	14.4	6	4.8
T2	rp2gen	77.5	1.9	40.8	9.6	8.1
T3	yacc	23.7	1.0	23.7	5.4	4.4
T4	c++ -c	27.1	1.9	14.3	7.9	3.4
T5	c++ -c	63.4	1.8	35.2	17.9	3.5
T6	c++ -c	266.3	2.0	133.2	71.6	3.7
T7	c++	23.9	2.0	12.0	8.4	2.8
T8	ar	70.2	1.9	36.9	22	3.2
T9	ar	364.0	2.2	165.5	78.6	4.6
T10	tar x	271.8	4.7	57.8	71.5	3.8
T11	make	71.6	2.3	31.1	25.3	2.8
T12	make	242.0	5.9	41.0	81.8	3.0
T13	tar c	60.2	1.0	60.2	10	6.0
T14	sgml2latex	42.0	1.0	42.0	13.8	3.0
T15	sgml2latex	270.3	1.1	245.7	71.0	3.8
T16	latex	94.1	1.4	67.2	35.5	2.7

In column 5, we list the network traffic reduction factors by operation shipping L_v/L_{op}, where L_v and L_{op} is the network traffic by value shipping and by operation shipping respectively. In column 7, we also list the expected network traffic reduction factors $L_v/L_{v,gz}$ if we used data compression ($L_{v,gz}$ is the expected size of the compressed traffic).

Figure 4: Network traffic reductions by operation shipping (and by data compression)

4.3 Network traffic reduction

For each test, we measured the traffic required for propagating the update by value shipping and by operation shipping. Both the file data and the overhead are 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, meta-data, and fingerprints, and so on, were counted towards the traffic.

In Figure 4, we show the traffic reduction L_v/L_{op}, where L_v and L_{op} are the traffic volumes required for the update propagation by value shipping and by operation shipping respectively.

Previous Coda projects [7, 14] have shown that cancellation optimization is effective in reducing the network traffic needed for propagating updates. For example, if a file is stored several times, then only the last `STORE` record is needed to be shipped. When we took the measurements, we did not wait for any possible cancellation optimization to happen, therefore, the measured traffic reductions achieved by operation shipping alone represent the best-case numbers. We excluded cancellation optimization because its effectiveness depends on usage patterns, and should be studied using file-reference traces. At this stage, we do not have file-reference traces performed at the level of user operations, so we have to evaluate operation shipping in isolation of cancellation optimization.

Nevertheless, the traffic reductions achieved by operation shipping were much more substantial than that achieved by cancellation optimization. [2] In 13 out of the 16 tests, the reduction exceeded a factor of 20; the highest reduction factor was 245.7 (T15); the smallest reduction was 12 (T7). In other words, operation shipping reduced the network traffic volumes by one to nearly three orders of magnitude.

[2] In their study of file-reference traces, Mummert, Ebling and Satyanarayanan [14] have reported that about 50 % of network traffic was saved when modification records were allowed to stay in the modification log for 600 seconds, waiting for possible cancellations to happen.

Test	Nature	Data size (Kbytes)	Elapsed time (msecs) 9.6-Kbps T_v	9.6-Kbps T_{op}	28.8-Kbps T_v	28.8-Kbps T_{op}	64-Kbps T_v	64-Kbps T_{op}
T1	rp2gen	27.5	27,921	8,282	9,666	6,404	4,539	5,637
			(312)	(73)	(8)	(50)	(20)	(37)
T2	rp2gen	76.3	71,937	9,322	24,294	7,358	11,416	6,706
			(27)	(61)	(39)	(9)	(133)	(90)
T3	yacc	23.5	22,025	3,215	7,563	2,364	3,506	2,049
			(31)	(60)	(13)	(34)	(0)	(9)
T4	c++ -c	26.0	25,112	5,098	8,683	3,491	4,164	2,928
			(64)	(31)	(38)	(88)	(176)	(107)
T5	c++ -c	62.4	59,144	7,546	19,899	5,927	9,591	5,377
			(254)	(48)	(51)	(12)	(93)	(43)
T6	c++ -c	265.3	257,143	15,645	88,274	13,877	39,167	13,181
			(23,989)	(82)	(8,418)	(92)	(233)	(9)
T7	c++	23.0	22,218	4,425	7,637	2,874	3,599	2,297
			(27)	(27)	(12)	(30)	(12)	(20)
T8	ar	69.3	65,473	5,613	22,059	4,104	10,571	3,646
			(58)	(21)	(77)	(25)	(343)	(138)
T9	ar	363.1	345,241	13,143	118,929	11,634	55,725	10,944
			(24,172)	(142)	(7,402)	(74)	(3,472)	(91)
T10	tar x	269.5	247,674	12,825	85,041	9,448	39,954	8,448
			(327)	(156)	(276)	(96)	(182)	(60)
T11	make	69.9	67,113	8,839	22,723	6,793	10,633	6,115
			(580)	(79)	(354)	(25)	(142)	(30)
T12	make	237.1	224,135	22,272	77,279	18,098	36,256	17,085
			(2,452)	(132)	(73)	(39)	(293)	(396)
T13	tar c	60.0	55,355	3,674	18,826	2,978	8,802	2,602
			(36)	(8)	(33)	(92)	(7)	(74)
T14	sgml2latex	41.8	38,602	5,433	13,160	4,648	6,209	4,439
			(15)	(18)	(12)	(25)	(87)	(235)
T15	sgml2latex	270.0	245,709	13,780	83757	12852	39414	12600
			(266)	(103)	(162)	(42)	(32)	(107)
T16	latex	93.4	86,619	8,522	29,429	7,194	13,869	6,243
			(30)	(646)	(54)	(669)	(49)	(39)

Elapsed time, in milliseconds, for update propagation using value shipping (T_v) and operation shipping (T_{op}) under three different network conditions. Figures in parentheses are standard deviations from three runs.

Figure 5: Elapsed time for value shipping and operation shipping.

4.4 Reduction of elapsed time

We also measured the elapsed time for propagating an update by value shipping and by operation shipping. The elapsed time is the time to complete the respective remote procedure calls: `ViceReintegrate` for value shipping, and `UserOpPropagate` for operation shipping. For the latter, the elapsed time comprises the time for shipping the operation log, re-executing the operation, and other overhead, such as checking the fingerprints. Since the elapsed time depends heavily on the network bandwidth, we measured it under three different network bandwidths: 9.6, 28.8, and 64.0 kilobits per second. The measurements are shown in Figure 5.

We summarize the speedups for the tests in Figure 6. The speedup is defined to be the ratio T_v/T_{op}, where T_v and T_{op} are the elapsed time for value shipping and operation shipping respectively.

We found that the speedups were substantial. They were the most substantial in the 9.6-Kbps network. Eight out of the 16 tests were accelerated by a factor exceeding 10. The maximum speedup was 26.3 (T9); the minimum speedup was 3.4 (T1). In the other two networks, the speedups ranged from a factor of 1.4 to 10.2. (There was one exception: test T1 slowed down when using operation shipping at 64 Kbps.)

However, we also found that the speedups were smaller than the numbers that we got from the previous version of our system, where forward error correction was not used. We performed some initial profiling of the time spent for operation shipping and found that the overhead of FEC was not small, sometimes as high as 80% of the total elapsed time. Although FEC is useful in handling the side effects of timestamps, it does not justify such a large overhead. We plan to use two optimizations to reduce the overhead: (1) we could use FEC on only those applications that need it, using hints from the users, and (2) we could choose to use a smaller number of parity symbols (says, 16) and substantially reduce the amount of computation needed.

Even without the planned optimizations, our current result has already shown that operation shipping is useful. Our result also indicates another advantage of operation shipping. That is, the speed of update propagation is much less sensitive to the network condition. This can be seen from the elapsed-time–bandwidth curves for test T9, plotted in Figure 7, in which the curves for value shipping is steep and that for operation shipping is flat. (Curves for other tests show similar trends.)

Combining the results of these two subsections, we conclude that operation shipping can reduce network traffic very substantially, can accelerate update propagation substantially, and can make the elapsed time of update propagation much less dependent on the network condition.

Test	Nature	Data size (Kbytes)	Speedup 9.6	Speedup 28.8	Speedup 64
T1	rp2gen	27.5	3.4	1.5	0.8
T2	rp2gen	76.3	7.7	3.3	1.7
T3	yacc	23.5	6.9	3.2	1.7
T4	c++ -c	26.0	4.9	2.5	1.4
T5	c++ -c	62.4	7.8	3.4	1.8
T6	c++ -c	265.3	16.4	6.4	3.0
T7	c++	23.0	5.0	2.7	1.6
T8	ar	69.3	11.7	5.4	2.9
T9	ar	363.1	26.3	10.2	5.1
T10	tar x	269.5	19.3	9.0	4.7
T11	make	69.9	7.6	3.3	1.7
T12	make	237.1	10.1	4.3	2.1
T13	tar c	60.0	15.1	6.3	3.4
T14	sgml2latex	41.8	7.1	2.8	1.4
T15	sgml2latex	270.0	17.8	6.5	3.1
T16	latex	93.4	10.2	4.1	2.2

Speedups for update propagation under three different network speeds: 9.6-Kbps (9.6), 28.8-Kbps (28.8), and 64-Kbps (64).

Figure 6: Speedups for update propagation

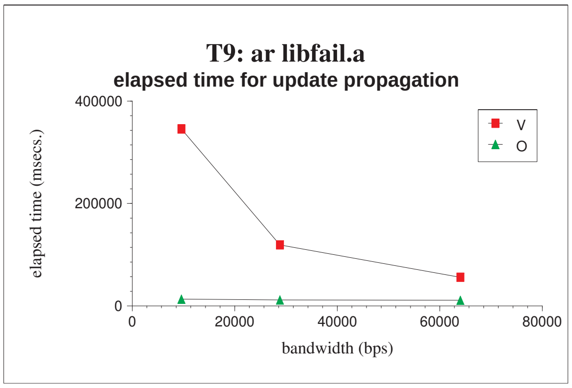

Figure 7: Elapsed time vs. bandwidth

First of all, we need to log and ship operations at a level higher than the file system itself, because the low-level file-system operations are not appropriate for operation shipping. Therefore, we need cooperation between the applications (or meta-applications) and the file system. Also, the new context requires several new concepts: re-execution by surrogate, adjustment of meta-data, validation of re-execution, and handling of non-repeating side effects. Finally, our file system can attempt operation shipping more boldly, because it has a fall-back mechanism of value shipping.

Directory operations. Logging and shipping of directory operations have been implemented in Coda prior to this

5 Related work and alternative solutions

5.1 Related work

To the best of our knowledge, this is the first work that attempts to propagate file updates by operation. However, some ideas and techniques used in this work have been studied in previous research.

Uses in database. The idea of operation-based update propagation is not new to the database community [15], but we apply it in a new context: distributed file system.

work [20, 19]. When a directory is updated on a Coda client (e.g., a new entry is inserted), instead of shipping the whole new directory to the server, the client ships only the update operation (e.g., the insertion operation). Directory operations are more like database operations, since they can be mapped directly to insertion, deletion, and modification of directory entries. In contrast, this work focuses on operation shipping of general user operations.

Repeatable re-execution. Several previous research projects have investigated the conditions for repeatable re-executions. Repeatable re-executions were desired for fault tolerance [1] or load balancing [2]. In the former case, a process P can be backed up by another process P_b. If P crashes, then P_b will repeat the execution of P since a recent checkpoint, and will thereafter assume the role of P. In the latter case, a process can migrate to another host to reduce the load imposed on the original host. In our work, repeatable re-executions are used to re-produce some file modifications that are identical to those produced by the original executions.

Re-execution for transactional guarantee. A previous Coda project has implemented a mechanism for re-execution of operations [9] [10]. It addresses the update conflicts that may be incurred in optimistically controlled replica. It proposes that a user can declare a portion of execution as an Isolation-Only Transaction (IOT). If an update conflict happens, Coda will re-execute the transaction. Our work is different in that we focus on performance improvement. Also, in our work, re-executions take place in a different host, whereas re-execution of IOTs take place in the same host. This implies that we must handle the case where re-execution does not produce the same results as the original execution.

5.2 Alternative solutions

There are other possible solutions to the problem that we are addressing. We are going to discuss four of them.

Delta shipping. The idea is to ship only the incremental difference, which is also called the *delta*, between different versions of a file. It has been proposed by many people and is currently being used as a general mechanism [24] or in specific systems including file systems [5], web proxies [12], file archives [11], and source-file repositories [22, 16].

It is possible to compute deltas not only for text files but also for binary files. We would like to mention the `rsync` algorithm [24] in particular. When shipping a file, the sending host suppresses the shipping of some blocks of data if they are found to be present on the receiving host already. It determines whether they are already present on the receiving host by using the checksum information supplied by the receiving host. The algorithm exploits a rolling checksum algorithm so that the blocks being matched can be started at any offset, not just multiples of block size.

Delta shipping has several limitations. First, a newly-created file has no previous version. 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 case, a slightly changed file may need a huge delta. This could happen, for example, if there are some global substitutions of strings, or if the brightness or contrast of an image is changed. In general, we believe operation shipping can achieve a larger reduction of network traffic.

On the other hand, delta shipping does not involve re-execution of applications and pre-arrangement of surrogate clients, as operation shipping does. Therefore, it is simpler in terms of system administration. We believe delta shipping and operation shipping can complement each other in a distributed file system. In particular, delta shipping can be used when operation shipping has failed for some updates, and when the file system has resorted to use value shipping.

Data compression. Data compression reduces the size of a file by taking out the redundancy in the file. This technique can be used in a file system or a web proxy [12]. However, the reduction factors achieved by data compression may be smaller than that of operation shipping. We did a small performance study using a representative implementation: the `gzip` utility, which uses the Lempel-Ziv coding (LZ77). We ran gzip with the updated files of the 16 tests in Section 4, and listed the expected traffic volume and expected traffic reduction by compressing the files before shipping them. The results are shown in Figure 4 (the sixth and seventh column). The expected traffic reductions by data compression ranged from 2.7 to 8.1, substantially smaller than that achieved by operation shipping, which ranged from 12.0 to 245.7. We were not surprised by the results, since we know operation shipping exploits the semantic information of the user operation, whereas data compression operates only generically on the files. Like delta shipping, data compression can complement operation shipping, and be used when our file system has resorted to value shipping.

Logging keystrokes. A file system may log keystrokes and mouse clicks, ship them, and replay them on the surrogate. As such, it may be transparent to an application even if the application is interactive. However, we are pessimistic about this approach, because it is very difficult to make sure the logged keystrokes and mouse clicks will produce the identical outcome on the surrogate machine. Too many things can happen at run-time that could cause the keystrokes to produce different results.

Operation shipping without involving the file system. Can we use operation shipping without involving the file system? We can imagine that someone may design a meta-application that logs every command a user types, and, without involving the file system, remotely executes the same commands on a surrogate machine. We believe such a system would not work, for the following reasons. If the file system had no knowledge that the second execution was a re-execution, it would treat the files produced by the two execution as two distinct copies, and would force the client to fetch the surrogate copy. It might even think that there was an update/update conflict. Besides, it cannot ensure the correctness of the re-execution. We therefore believe that the file system plays a key role in useful and correct operation shipping,

6 Conclusion

Our experience with operation shipping, although it is limited only to the application-transparent case, is entirely favorable. We have implemented a prototype by extending the Coda File System, and have demonstrated that operation shipping can achieve substantial performance improvements.

Efficient update propagation is important for insulating users from the unpleasant consequences of low bandwidth networks. Indeed, without this capability, performance may be sufficiently degraded that users are tempted to forgo the transparency benefits of a distributed file system, and rely on explicit copying of local files instead. Our results suggest that operation shipping can play an important role in the design of future distributed file systems for bandwidth-challenged environments.

Acknowledgements

Matt Mathis gave us the idea of using forward error correction for handling the side effect of time stamps. Phil Karn allowed us to use his Reed-Solomon-Code package in our prototype. Maria Ebling and David Eckhardt provided major input for the high-level design of the system. Peter Braam insisted that operation logging must be as transparent as possible to non-interactive applications, and made us think more carefully about the issue. Many people, including John C. S. Lui, Peter T. S. Tam, K. Y. So, Eric Tilton, and Kip Walker, helped in improving the presentation of the paper. Finally, this work would not have been possible without the excellent support of the Coda community.

References

[1] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault Tolerance Under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, February 1989.

[2] F. Douglis and J. Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software–Practice and Experience*, 21(8):757–785, August 1991.

[3] The Coda Group. Coda File System. Available from
http://coda.cs.cmu.edu.

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

[5] Airsoft Inc. Powerburst – The First Software Accelerator That More Than Doubles Remote Node Performance. Available from
http://www.airsoft.com/comp.html, Cupertino, CA.

[6] P. Karn. Error Control Coding, a Seminar handout. Available from
http://people.qualcomm.com/karn/dsp.html.

[7] J. J. Kistler. *Disconnected Operation in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1993.

[8] J. J. Kistler and M. Satyanarayanan. Disconnected Operation in the Coda File System. *ACM Transactions on Computer Systems*, 10(1), February 1992.

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

[10] Q. Lu and M. Satyanarayanan. Improving Data Consistency in Mobile Computing Using Isolation-Only Transactions. In *Proceedings of the Fifth IEEE HotOS Topics Workshop*, Orcas Island, WA, May 1995.

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

[12] J.C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proceeding of the ACM SIGCOMM'97*, 1997.

[13] L. B. Mummert. *Exploiting Weak Connectivity in a Distributed File System*. PhD thesis, Carnegie Mellon University, School of Computer Science, 1996.

[14] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, Copper Mountain Resort, Colorado, December 1995.

[15] K. Patersen, M. J. Spreitzer, D. B. Terry, M. M. Theimer, and A. J. Demers. Flexible Update Propagation for Weakly Consistent Replication. In *Proceedings of the 16th ACM Symposium on Operatiog Systems Principles*, Saint-Malo, France, October 1997.

[16] The FreeBSD Documentation Project. CVSup: in FreeBSD Handbook. Available from http://www.freebsd.org/handbook/cvsup.html.

[17] R. Rivest. The MD5 Message-Digest Algorithm, Internet RFC 1321. Available from http://theory.lcs.mit.edu/~rivest/publications.html, April 1992.

[18] M. Satyanarayanan, M. R. Ebling, J. Raiff, and P. J. Braam. *Coda File System User and System Administrators Manual*. School of Computer Science, Carnegie Mellon University, August 1997. version 1.1.

[19] M. Satyanarayanan, J. J. Kistler, P. Kumar, and H. Mashburn. On the Ubiquity of Logging in Distributed File Systems. In *Third IEEE Workship on Workstation Operation Systems*, Key Biscayne, FL, Apr 1992.

[20] 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 Transaction on Computers*, 39(4), April 1990.

[21] B. Schneier. *Applied Cryptography*. John Wiley & Sons, Inc., second edition, 1996.

[22] Cyclic Software. Concurrent Versions System (CVS). Available from http://www.cyclic.com/.

[23] W. R. Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[24] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, Available from http://samba.anu.edu.au/rsync/, June 1996.