The Case for Semantic Aware Remote Replication

Xiaotao Liu, Gal Niv, K.K. Ramakrishnan†, Prashant Shenoy, Jacobus Van der Merwe†
University of Massachusetts / †AT&T Labs-Research

Abstract

This paper argues that the network latency due to synchronous replication is no longer tolerable in scenarios where businesses are required by regulation to separate their secondary sites from the primary by hundreds of miles. We propose a semantic-aware remote replication system to meet the contrasting needs of both system efficiency and safe remote replication with tight recovery-point and recovery-time objectives. Using experiments conducted on a commercial replication system and on a Linux file system we show that (i) unlike synchronous replication, asynchronous replication is relatively insensitive to network latency, and (ii) applications such as databases already intelligently deal with the weak persistency semantics offered by modern file systems. Our proposed system attempts to use asynchronous replication whenever possible and uses application/file-system "signals" to maintain synchrony between the primary and secondary sites. We present a high-level design of our system and discuss several potential challenges that need to be addressed in such a system.

1 Introduction

In recent years there has been increased awareness of the need for business continuity in the face of disasters [9]. Of particular importance is the protection and availability of business data in such circumstances. The need for data availability is typically addressed by replicating business data on a local/primary storage system, to some remote location from where it can be accessed in case of a disaster.

From a business/usability point of view, such remote replication is driven by two metrics [3]. First is the *recovery-point-objective* which is the consistent data point to which data can be restored after a disaster. Second is the *recovery-time-objective* which is the time it takes to recover to that consistent data point after a disaster.

Remote replication can be broadly classified into the following two categories:

- Synchronous replication: every data block written to a local storage system is replicated to the remote location before the local write operation returns.
- Asynchronous replication: in this case the local and remote storage systems are allowed to diverge. The amount of divergence between the local and remote systems is typically bounded by either a certain amount of data, or by a certain amount of time.

For the remote replication metrics mentioned above, synchronous replication would therefore appear to be the ideal replication mechanism on both counts. First, from a recovery-point-objective, for synchronous replication, the local and remote storage systems are in lock step and therefore data committed to disk is guaranteed to be available at the remote system. Similarly, from a recovery-time-objective, since the local and remote systems are in sync, no time consuming procedures are required to bring the remote system to a consistent state in the event of a local failure.

For these reasons, synchronous replication is normally recommended for applications, such as financial databases, where consistency between local and remote storage systems is a high priority. However, these desirable properties come at a price. First, because every data block needs to be replicated remotely, synchronous replication systems can not benefit from any local write coalescing of data if the same data blocks are written repeatedly [6]. Second, because data have to be copied to the remote location before the write operation returns, synchronous replication has a direct performance impact on the local system, since both lower throughput and increased latency of the path between the primary and the remote systems are reflected in the time it takes for the local disk write to complete. This requires that network resources be engineered to have sufficient throughput to accommodate the peak, bursty load.

Because the local and remote systems are allowed to diverge, asynchronous replication always involves some data loss in the event of a failure of the primary system. On the other hand, since write operations can be batched and pipelined, an asynchronous replication system can more smoothly and fully utilize the available network capacity between local and remote sites, regardless of the latency between sites. As such, in terms of the rate of data transfer, asynchronous replication systems move data in a much more efficient manner than synchronous replication systems.

The performance penalty due to network latency in synchronous replication is usually tolerable when the secondary site is separated from the primary by only a few miles or tens of miles. However, to ensure that large disasters do not have catastrophic business consequences, many critical businesses today are required by regulations to maintain a secondary site that is separated from the primary by a few hundred miles. The increased network latency due to this larger geographic separation can have a drastic performance (latency) impact on applications, and it is no longer clear

whether the higher penalty of synchronous replication is tolerable even for stringent business continuity needs. The goal of our work then is to investigate whether a replication system can be developed that (i) embodies the desirable properties of both synchronous and asynchronous approaches, while removing (or at least mitigating) the undesirable properties of each and (ii) is particularly suitable for replication over large distances.

We observe that most current replication mechanisms primarily operate at the disk block level. That means that the replication process is completely oblivious to the application and indeed to the file system operating on it. This is attractive for the replication mechanism; "all" it has to do is to faithfully replicate each block of data that is handed to it. This means that such replication works with any application, any file system and any operating system. The downside though of such a simple system is that every block of data is treated with the same importance, regardless of the importance and/or urgency applied to it by the application and/or file system.

The key insight of our approach is that, from an application/file system perspective, all writes are *not* treated equally. For performance reasons modern file systems typically do not commit each write request to disk [7]. Rather, writes normally go into a write buffer in memory where it can spend a significant amount of time (15-30 seconds on Unix-like file systems and even longer on others) before being written out to disk. The consistency/protection semantics offered by such a file system are clearly undesirable for many applications (e.g., databases) where there is a need to store data persistently to deal with crashes. Applications deal with this by explicitly forcing data to disk when needed. For example this can be done by opening a file in synchronous mode (not to be confused with synchronous replication), or by indicating to the file system through system calls (e.g., fsync/fflush) that data should written to disk.

The question addressed in this work is whether the importance that the application or file system attach to a write (indicated by the "signals" described above), can be used to inform the replication system regarding the importance of a write and whether it should be replicated synchronously or not. We call this **semantic-aware remote replication**. In particular, our approach attempts to make use of asynchronous replication, unless the application/file system "signals" the need for synchronous semantics. In the latter case, the replication of the data is performed with synchronous semantics. Although one can envisage applications being modified to specifically make use of this system, our initial work indicates that this does not appear to be necessary—we expect the file system to be able to infer the application semantics through existing mechanisms.

The outline of the remainder of the paper is as follows. In Section 2, we put our work into perspective by considering related work. Our approach involves the replication system providing a semantic aware interface to the file system, which in turn deduce the relative importance of data writes from application behavior. Understanding the interaction between application, file system and storage system and remote replication is therefore crucial to our approach. In Section 3, we consider the interactions of these different components in detail. Finally, in Section 4 we outline the essence of our approach and consider some of the potential problem areas that need to be addressed in a detailed design.

2 Related work

Seneca [4] uses write records combined with the notion of barriers in order to coalesce writes and reduce network throughput. The protocol records each write into the primary log, while periodically inserting send barriers into it. Any blocks written after the last send barrier can be overwritten, eliminating the need for their retransmission. Write-ordering is ensured by receive barriers, which guarantee that any blocks written between the old block and the end of the log will be considered one atomic unit, and will be written to disk as such. The Ursa Minor system argues that no single fault model is optimal for all applications and proposed supporting data-type specific selections of fault models and encoding schemes [1].

The Veritas Volume Replicator uses a log and employs transactions to perform asynchronous replication. It first logs writes into a Storage Replicator Log (SRL), then writes the data to the primary volume, and finally transmits it to the secondary volume. A write is reported as complete to the file system as soon as it is logged into the SRL, but it is only marked as complete in the SRL itself once an acknowledgment has been received that the data has been written to the secondary volume [8]. Hence the system guarantees a consistent state, since the log is always aware of which writes have been replicated successfully. In contrast, when operating in synchronous mode, the Veritas Volume Replicator waits until it has received an acknowledgment that the data has been received (although not yet written to disk) at the secondary before reporting the write as complete to the file system. In this case extra reliability is earned at the cost of speed, since the file system must wait for data to propagate over the network and acknowledged each time a write is performed.

Network Appliance SnapMirror uses a technique known as snapshotting to keep the secondary volume up to date. Using the WAFL filesystem, which supports snapshot operations, the system takes an initial snapshot–essentially a read-only copy–of the primary volume and uses it to initialize the secondary volume. After some time elapses or a threshold is reached, a new snapshot is taken and the differences between the previous and the new snapshot are transferred to the secondary volume [2].

There exist numerous other commercial products that perform replication, such as IBM Extended Remote Copy, HP Continuous Access XP, and EMC RepliStor. EMC prod-

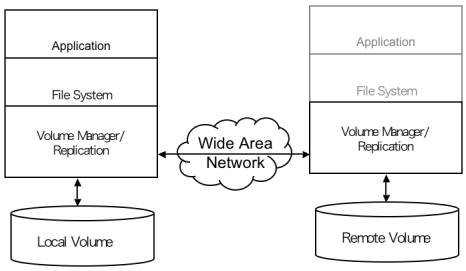


Figure 1: Remote Replication

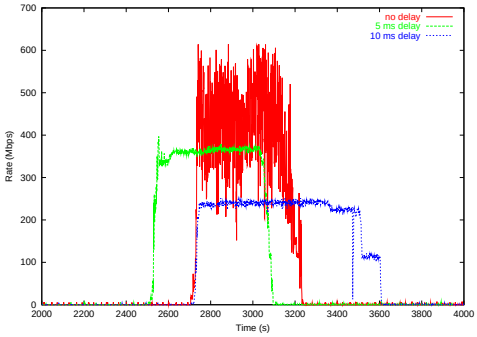


Figure 2: Synchronous replication: FCIP throughput for different delay values

ucts, in particular, are based on the Symmetrix Remote Data Facility (SRDF), a family of solutions which include synchronous and asynchronous replication protocols. An excellent description of these and others, as well as a detailed taxonomy of the different approaches for replication can be found in [4].

3 Remote Replication

In this section we consider the components involved with remote replication, end-to-end. In Figure 1, we show a generic remote replication system. At the primary site, data written by an application passes through the file system before it is handed over to a volume manager to write the data to disk. In a remote replication system, the volume manager will also be responsible for replicating the data via the wide area network to a remote system, where a reciprocal volume manager will take part in the remote replication protocol and write the data to disk at the remote site. We investigate the interaction between these different components by first considering the remote replication of a commercial replication system in Section 3.1. Then in Section 3.2 we look in detail at the functioning of the default Linux ext3 file system and in particular the interaction of a database with this file system.

3.1 Remote Replication on IP networks

In this section we present results from a case study involving remote replication over IP networks using a commercial storage system with remote replication capabilities. For this case study, the storage arrays were interconnected over a testbed consisting of a Gigabit Ethernet IP network using storage switches implementing the Fibre Channel over IP (FCIP) [5] protocol. Our IP testbed was equipped with a commercial network emulator which allowed us to introduce various network anomalies (in particular delay) in a controlled fashion for our experiments.

To understand the difficulties of having strict synchronous replication, especially when the secondary (replication) site is a long distance away from the primary site, it is useful to examine the fundamental limitations imposed by the system and protocols. For this purpose, we examined the performance of our test system using both synchronous and asynchronous replication while varying the delay introduced by the network emulator in our testbed.

We first consider the case where the storage systems were configured to perform synchronous replication. Figure 2 shows the instantaneous network throughput (averaged over 1 second intervals) achieved by the system for three different delay settings. The plots show the activity for the initial remote replication phase where the local and remote disks are being synchronized. (I.e., the amount of data transfered in each of the three runs are the same.) The impact of even this modest increase in delay, when performing the replication synchronously, is evident from the significant decrease in the throughput achieved. Further, even in the case where no delay was introduced, the throughput achieved is significantly lower than the 1 Gbps available in the testbed network.

The second example we show is where the systems were configured to perform a type of asynchronous replication, namely, the replication of ”point-in-time” copies. The remote replication application uses the functionality in the storage array to perform coherent, recurrent, background copy of data by maintaining several copies of the data that represent a consistent snapshot of the data volume. An old version of the data on the remote system is not deleted before a new copy has been replicated to the remote site in its entirety. Once all the volumes are synchronized for the first time, only incremental transfers are typically performed so that only data that has changed since the previous transfer is copied across the network link.

When the distances between the local and remote sites are several hundred to a few thousand miles, the latency implies that a large amount of data can be in flight at any instant between the source and destination. Because the local and remote copies are allowed to diverge, asynchronous replication can effectively fill this pipeline with data before an acknowledgment is needed from the remote end and is therefore much less sensitive to the increase in delay.

Figure 3 shows the throughput achievable for asynchronous replication for different values of delay in our testbed. Notice that the achieved throughput is quite close to the 1 Gbps link capacity in all cases, with appropriate tuning

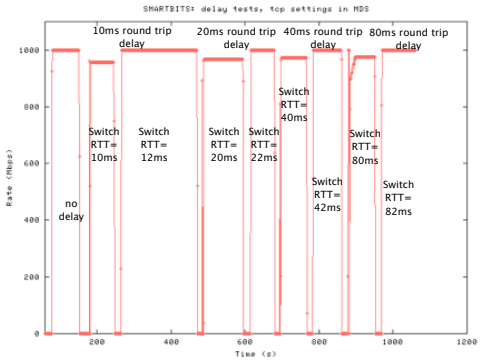


Figure 3: Asynchronous replication: FCIP throughput with different delay values and FCIP switch parameters settings.

of TCP and Fibre Channel switch and protocol parameters. This remains true, even if the round trip delay is increased to 80 ms, which is equivalent to a round trip delay across the continental US. Thus, asynchronous replication can be made relatively insensitive to distance, which is clearly very attractive.

3.2 Behavior of Applications on a File System

To better understand how applications interact with the underlying file system and the disk, we conducted two sets of experiments. We ran a synthetic application that issued writes to a journalled Linux file system and observed how the file system handled these writes at the kernel level. Next, we ran the Postgresql database on the journalled file system and observed how it issued writes to the file system upon each database transaction. We used Linux's *systemtap* kernel-profiling tool to obtain a detailed trace of kernel events, including application-issued system call, execution of kernel daemons, and writes to the disk driver.

All experiments were carried out using Linux 2.6.9, with ext3 as the underlying filesystem and using Postgresql 7.4.13. We experimented with various file system journalling options but only report the results on the default option ($journal = ordered$). Note that *no replication was performed in this setup.*

3.2.1 Filesystem Write Handling

Like any flavor of Unix, Linux allows each file to be opened either in the synchronous or asynchronous mode by specifying the appropriate `O_SYNC` or `O_ASYNC` flag in the `open` system call. This determines how writes will be handled by the filesystem.

Any application-level write can trigger three types of writes at the kernel level:

1. *Journal writes*, which may include logging of metadata, data, or both, depending on the journal configuration. The default is to only log metadata.

2. *Data writes*, which write the file data blocks to disk.

3. *Metadata writes*, which write the file metadata to disk.

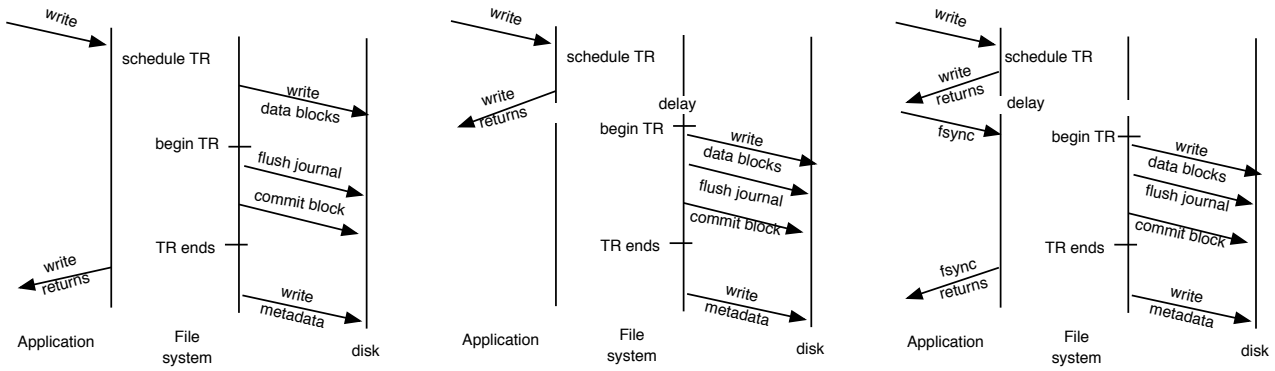
Like most operating systems, Linux employs a kernel-level file system buffer cache to optimize performance. The OS employs two daemons— `kjournald` and `pdflush`—that flush dirty blocks to the disk in the background. `kjournald` is responsible for writing journal blocks to disk, while `pdflush` is responsible for flushing dirty data and metadata buffer blocks to disk. To ensure atomicity, all writes to the journal are implemented as transactions; kjournald executes all scheduled transactions periodically— the default period is 5 seconds.

In the first experiment, we opened a file in the synchronous mode and traced the sequence of kernel events triggered by a write system call. We then repeated the experiment by opening the file in asynchronous mode. Finally, we opened the file in asynchronous mode and observed kernel behavior due to a write followed by an fsync (which flushes the data to disk). Figure 4 illustrates the observed sequence of events in these three experiments. We discuss the salient features of these sequences below:

Synchronous write: As expected, writes in synchronous modes are blocking – the write system call returns only when all data and the journal metadata have been written to disk. In particular, the write call causes the file system to schedule a journal transaction. The file system then writes the data dirtied by the write call to disk. The journal transaction is executed next by kjournald, which flushes the journal to disk (thereby logging the metadata to the disk). kjournald then writes out a commit block indicating the end of the journal transaction. At this point, the write call, which was blocked, returns back to the application, signaling the end of the write. At some later instant, pdflush writes out the dirty meta-data to disk (since the journal already includes this information, there is no need to write out the metadata synchronously; asynchronous metadata writes improve performance).

Asynchronous write: As expected, asynchronous writes are non-blocking. An asynchronous write only schedules a journal transaction and then returns back to the application. At a later instant that is determined by the kjournald commit interval, kjournald actually executes this transactions and writes out the dirty data block as well as the journal block and the commit block. The file meta data is written out by pdflush at a later time.

Asynchronous write followed by fsync: In this case, the write system call returns immediately after scheduling a journal transaction. The fsync, however, is a blocking call that causes all dirty journal and data blocks to be flushed to disk. Linux supports three flavors of `sync`: `fsync`, which flushes all dirty data and metadata of a file to disk, `fdatasync`, which flushes all dirty data but not metadata, and `sync` which flushes all dirty data systemwide.



(a) sync write (b) async write (c) async write + fsync

Figure 4: Event sequence for a synchronous write, an asynchronous write, and asynchronous write followed by a fsync.

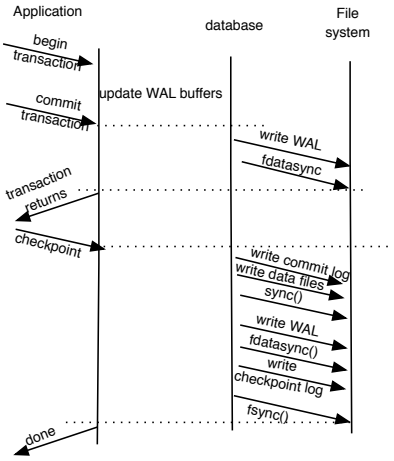


Figure 5: Event sequence for a SQL query in Postgresql.

3.2.2 Behavior of a Database

A typical database can be configured to run on raw disk partitions or over a file system. In the former case, the database is responsible for buffer and storage management—it implements its own buffer cache in application space and writes out dirty data when needed to the raw partition. The advantage of this approach is that it prevents double buffering within the operating system. However, for reasons of convenience, databases also support standard file systems. To understand how this is done, we examined *Postgresql*—an open-source object-relational database. When configured to run over standard file system, Postgresql maintains several files: (1) a write-ahead log (WAL) file, that logs changes to data files (database tables), (2) data files, that hold tables and indices, (3) a commit log file, that contains the commit status of transactions, and (4) a checkpoint log file, that holds the location of recent checkpoints.

By default, Postgresql opens all files in *asynchronous* mode for writing, although it is also possible to configure the database to use synchronous writes. To understand how the database operates on files, we ran an SQL query, wrapped around a database transaction, to insert a few rows into a table. Figure 5 depicts the sequence of events that are triggered by the SQL query. We observe that the query causes the update to be logged to the WAL buffers in user space. When the transaction commits, a write system call is issued to the file system, followed by an fsync which flushes dirty WAL blocks out to disk. The transaction returns only after the flush completes. The data written by the SQL query are held in shared memory in user-space and are written out asynchronously at a later time (either when the buffer fills up or if checkpoint is called). Whenever a database *checkpoint* is invoked by the user (or automatically by the system every so often), a write is issued to the commit log file, and all data in the shared memory buffers are written out, followed by a `sync` to flush these writes to disk. Once the sync completes, a checkpoint record is written out the WAL, followed by a write to the checkpoint log file; both writes are flushed using `fsync`. Thus, the database does not violate correctness (i.e., maintains ACID properties) even though *all writes are in asynchronous mode.* This is ensured by intelligently issuing sync or fsync calls to flush the OS-buffered writes out to disk—the use of asynchronous writes improves performance without violating safety.

4 Semantic Aware Remote Replication

In the previous section we have shown that applications that care about the persistency of written data, intelligently use existing file system mechanisms to ensure that data is written to disk when needed. We have also shown that asynchronous replication is likely to be more efficient than synchronous replication in terms of moving data from a primary to a backup site.

Combining these observations, the essence of our approach to remote replication is to perform replication asynchronously by default, and to automatically switch to synchronous replication semantics only when prompted to do so by the application. Note that switching between asynchronous and synchronous replication in our system is not

on a system wide basis, but on-demand based on a per application/file basis. With synchronous semantics the relevant data will be guaranteed to have been replicated to the remote system when the system call returns. Specifically our system operates as follows:

For files opened in synchronous mode (i.e., using `O_SYNC`), our system performs synchronous remote replication.

For files opened in asynchronous mode (i.e., using `O_ASYNC`), our system performs asynchronous replication, unless the application issues one of the following calls:

- `fsync`: All dirty data and meta data of the opened file that reside in the filesystem or the volume manager are replicated to the remote site.
- `fdatasync`: Dirty data associated with the opened file in either the file system or the volume manager is replicated to the remote site.
- `sync`: All dirty data in the file system or the volume manager is replicated to the remote site.

Thus, the replication system needs to be *sync-aware*—a sync from the application causes file system buffers to be flushed to volume manager, and requires a sync to be issued to the volume manager to flush all asynchronously queued up writes to the remote site. As a result, when the sync call returns, not only are all dirty blocks written out to the local disk, they are also guaranteed to have been flushed to the remote site. While a complete system design is clearly beyond the scope of this paper, there are several issues that will need to be considered.

While a system-wide sync call is easy to implement within the replication system (by simply flushing all queued up data), the implementation of fsync and fdatasync are more interesting. It requires the replication system to be aware of the block to file mapping or be *file-aware*, so that a fsync or fdatasync only flushes those queued up blocks that belong to this file.

Write ordering has important implications on consistency. For example, consider the same file that is opened in synchronous mode by one application and in asynchronous mode by another. If an asynchronous write is issued first, followed by a synchronous write to the same block, then the latter write may be sent out first and get overwritten by the older write, resulting in an inconsistency. The replication system needs to perform write coalescing at the volume manager level to avoid such inconsistencies. The implications of out-of-order writes also merits careful attention [4].

Finally, the system can be made *network-aware* to better utilize the available network capacity. Buffering of writes enables the system to smooth out the load on the network. A further optimization may be to enable the system to inform the filesystem so as to opportunistically use network capacity by flushing its dirty buffers. Such optimizations can reduce overall network bandwidth needs for replication. The advantages obtained through write coalescing and judicious use of network resources will need to be carefully considered.

5 Conclusion

We have proposed a Semantic Aware Remote Replication system to meet the contrasting needs of both system efficiency and safe remote replication with tight recovery-point and recovery-time objectives. Our system exploits the fact that applications already intelligently deal with the weak persistency semantics offered by modern file systems. In this paper we have motivated our approach and presented a high level design of how such a system would be realized. Our future work is in evaluating the potential benefit of our approach for various real-world workloads. Further, while we mentioned some of the potential pitfalls, the detailed system design and implementation is work in progress.

References

[1] M. Abd-El-Malek, W. V. Courtright II, C. Cranor, G. R. Ganger, J. Hendricks, A. J. Klosterman, M. Mesnier, M. Prasad, B. Salmon, R. R. Sambasivan, S. Sinnamohideen, J. D. Strunk, E. Thereska, M. Wachs, and J. J. Wylie. Ursa minor: versatile cluster-based storage. USENIX Conference on File and Storage Technologies, December 2005.

[2] K. Brown, J. Katcher, R. Walters, and A. Watson. Snapmirror and snaprestore: Advances in snapshot technology. Network Appliance Technical Report TR3043. `www.netapp.com/tech_library/3043.html`.

[3] Disaster Recovery Journal. Business continuity glossary. `http://www.drj.com/glossary/drjglossary.html`.

[4] M. Ji, A. Veitch, and J. Wilkes. Seneca: Remote mirroring done write. USENIX 2003 Annual Technical Conference, June 2003.

[5] M. Rajagopal, E. Rodriguez, and R. Weber. Fibre channel over tcp/ip (fcip). IETF RFC 3821, July 2004.

[6] C. Ruemmler and J. Wilkes. Unix disk access patterns. Proceedings of Winter 1993 USENIX, Jan 1993.

[7] M. I. Seltzer, G. R. Granger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. USENIX Anual Conference, June 2000.

[8] Symantec Corporation. *Veritas Volume Replicator Administrator's Guide*. `http://ftp.support.veritas.com/pub/support/products/Volume_Replicator/2%83842.pdf`, 5.0 edition, 2006.

[9] U.S. Securities and Exchange Commission. Draft interagency white paper on sound practices to strengthen the resilience of the u. s. financial system. Available from:`http://www.sec.gov/rules/concept/34-46432.htm`, Aug 2002.