社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2005－OS－98 (3)
2005／2／22

ジャーナリングファイルシステムの構造を利用した非同期リモートミラーリング手法

藤田 智成† 矢田 浩二†

†NTT サイバーソリューション研究所

要旨

本稿は，ホストコンピュータがジャーナリングファイルシステムを構築し，データを保存しているストレージシステムと，ネットワークで接続された他のストレージシステムとの非同期リモートミラーリングの高速化手法，TARM を提案する．TARM では，ローカルサイトのストレージシステムが，ファイルシステムのディスク上のデータ構造を利用することで，リモートストレージシステムに保存されたデータの一貫性保証に必要な更新順序制限の有無を検出する．これによって，ホストコンピュータやリモートストレージシステムに，ミラーリングのための専用機能を追加することなく，リモートストレージでの更新順序に制限が生じないデータを検出し，並列に更新することで，高速化を達成する．本稿は，Linux の ext3 ファイルシステムを利用する場合の，アルゴリズムを述べる．

Asynchronous Remote Mirroring with Journaling File Systems

FUJITA Tomonori† YATA Kouji†

†NTT Cyber Solutions Laboratories

Abstract

This paper presents TARM, a block storage system, which transparently and asynchronously replicates data across multiple storage sites. TARM is designed for journaling file systems. It uses the knowledge of the file system such as information about its on-disk data structures to ensure that the replicated data are recoverable regardless of catastrophic site failures. TARM enables remote storage systems to write the received data in a favorable order without requiring any procedures from the host or special features from the remote storage systems. We implement a storage system that adopts the algorithm to the knowledge of a popular journaling file system available for Linux, Ext3 file system as our first case study.

1 Introduction

Many companies have started to realize the importance of protecting data from catastrophic site failures such as flood, fire, or earthquake. A widely deployed solution is *remote mirroring*, which stores data across multiple storage sites.

Compared with the traditional data protection, tape backup, remote mirroring provides faster data recovery and more recent data in the event of a disaster. With remote mirroring, it is possible to quickly resume normal operations by using a spare host and replicated data on the secondary storage system on the remote site.

How closely the copies are kept synchronized is an important design choice for remote mirroring.

Synchronous remote mirroring immediately updates all copies. there is no divergence between two copies. All writes (updates) from the host are blocked until their completion on the secondary storage system. Synchronous remote mirroring incurs large write la-

tency and requires a high bandwidth network.

Asynchronous remote mirroring permits some divergence between two copies. The primary storage system sends the write completion to the host independent of the secondary storage system. Then, the primary storage system sends the writes to the secondary storage system. This provides better write performance and requires a less expensive network, in exchange for the potential loss of recently written data.

With asynchronous remote mirroring, it easily leads to inconsistent, unusable replicated-data unless the write ordering across the entire data set is maintained.

The basic approach to keep replicated-data consistent is that committing all updates on the secondary storage system in the exact order. However, it leads to the poor performance because the secondary storage system can not commit several updates simultaneously.

The common approach, which commercial storage systems use, for ensuring the consistency of replicated data with asynchronous remote mirroring is using specialized functionality installed a host and storage systems.

There are two disadvantages of this approach.

- Companies are reluctant to introduce software changes to operating systems or applications for new functionalities such as asynchronous remote mirroring.
- Storage vendors use own proprietary protocol to control a host and storage systems. Therefore, you cannot replace your storage system with other vendors' storage system easily. Additionally, this makes it difficult to use shared storage systems on a storage service provider (SSP).

In this paper, we describe the TARM (Transparent, Asynchronous Remote Mirroring) storage system. A host accesses TARM by using a block-level storage protocol, such iSCSI [1]. TARM works as a primary storage system and replicates data over multiple remote sites asynchronously.

TARM is especially designed for journaling file systems, and it has the knowledge of the file system that the host uses on the storage system, such as information about its on-disk data structures. By using this knowledge, TARM ensures the consistency of replicated data without special procedures for remote mirroring between the host and the storage systems or atomic update methods on the secondary storage systems. The host does not need to know anything about

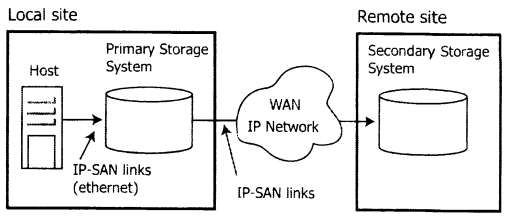


図 1: Typical Remote Mirroring deployment.

remote mirroring and TARM can replicates data on any storage system. TARM permits the secondary storage systems to write transferred data in a favorable order to improve performance. We choose a popular journaling file system, the Linux Ext3 file system, as our first case study, and present how TARM uses its knowledge.

The outline of the rest of this paper is as follows. Section 2 discusses the issues related to asynchronous remote mirroring and data consistency. Section 3 provides the TARM architecture. Section 4 summarizes related work, and Section 5 summarizes the main points.

2 Asynchronous remote mirroring

2.1 System architecture

We concentrate on the two-site configuration here to simplify the explanation.

Figure 1 shows a remote mirroring system in a typical configuration with the iSCSI protocol.

The host and the primary storage system at the local site serve data under normal operating conditions. They communicate by using the iSCSI protocol. The primary storage system and the secondary storage system also communicate by using the iSCSI protocol.

A host views the iSCSI driver as a general SCSI host bus adapter driver managing directly attached disk drives. No difference can be found between an iSCSI storage system and a local hard disk directly connected with the host computer. Therefore, the host can use any local file system or a database that accesses to a local hard disk. TARM is designed for environments in which a host compute uses a file system to access a storage system.

A host issuing SCSI commands is called an initiator in the iSCSI protocol like the SCSI protocol. A

target provides services to initiators. Therefore, in Figure 1, the host and the primary storage system have the initiator-target relationship. In addition, the primary storage system and the secondary storage system also have the initiator-target relationship.

2.2 Data consistency

For file systems, data consistency commonly refers to the consistency of metadata, which is information about the structure of the file system. Some system calls require several metadata changes. In such a case, metadata must be updated in such a way that the file system can avoid file system corruption and restore the file system to an accurate state after a system crash. File systems require some updates to be in a precise order to ensure it.

Traditionally, a file system synchronously updates metadata and scans the file system during the recovery process. This incurs poor performance of metadata operations and a long recovery time. Journaling and Soft Updates [2] are often used by modern file systems to solve these problems [3].

These techniques can update most metadata asynchronously, however they still require precise ordering in some parts. Therefore, replicated-data on the secondary storage system can be inconsistent unless the write ordering across the entire data set is maintained.

Suppose that a file system on the host needs to update sector 1 and sector 2 on the primary storage system in that order. The host updates sector 1. After the host receives the completion of updating on sector 1 from the primary storage system, it requests the primary storage system to update sector 2.

After the above operations, the primary storage system sends both updated data to the secondary storage system. The write ordering of these data on the secondary storage system is undecided.

If catastrophic site failures destroy the whole data on the primary storage system and an unexpected event crashes the secondary storage system during the updates on the secondary storage system, the replicated data can be inconsistent. That is, if sector 1 is not updated and sector 2 is updated, the replicated data is inconsistent. The previous data stored at sector 1 is not stored anywhere, thus the replicated data is not recoverable.

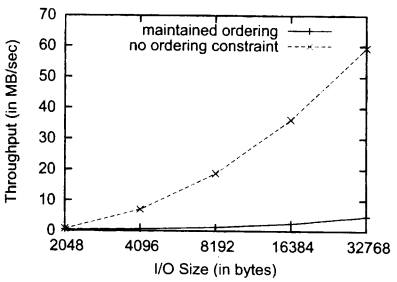


図 2: Effects of write ordering on performance

2.3 Performance drop due to write ordering

The constraint on the write ordering to avoid the inconsistency of the replicated data significantly reduces the mirroring performance. We examined how much the performance falls because of the constraint.

The iSCSI target host used one 2.8 GHz Xeon processor with 2 GB main memory and ran a modified version of the iSCSI enterprise target [4] and the Linux kernel version 2.4.25. The Maxtor Atlas 10K 36.7 GB 10000 RPM SCSI disks were directly connected to the host via LSI Logic 53C1030 Ultra320 SCSI chip.

The iSCSI initiator host used one 2 GHZ Xeon processor with 1 GB main memory and ran version 4.0.1.1 of a Cisco iSCSI initiator [5] and the Linux kernel version 2.6.4.

Both hosts used an Intel Pro/1000 MT Server Adapter connected to a 66MHz 64-bit PCI slot. They were connected by an Extreme Summit 7i Gigabit Ethernet switch.

Figure 2 shows the results of the microbenchmarks that write sequentially 50,000 times with the I/O sizes ranging from 2 KB bytes to 32 KB. The microbenchmarks were run with two configurations: *maintained ordering*, in which the write ordering across the entire data set is maintained by using the ORDERED attribute of SCSI commands; *no ordering constraint*, in which the target writes the received data in a favorable order.

The results indicate that the write ordering constraint negates the performance advantage that the asynchronous mirroring has.

3 TARM architecture

File systems can update most of data in a favorable order, although they need some of data to be updated in the exact order for the consistency. Storage systems do not know such *dependency information* about the relationship among data, that is, which of the data must be written in a precise order and the write ordering. So even if storage systems can replicate data asynchronously, the secondary storage system must serialize all writes to ensure data consistency.

TARM algorithm is implemented inside a primary storage system. The primary storage system controls the write ordering on the secondary storage system. The primary storage system identifies the write ordering constraint of data by exploiting the knowledge of the file system. Therefore, the secondary storage system can update the data that do not have the write ordering constraint simultaneously. This boosts mirroring performance.

TARM provides *power-failure consistency* [6]. It keeps replicated data as recoverable guarantees that the data can be quickly used as the file system in the event that a disaster has completely destroyed the primary storage system and the host.

A logical volume is used as the entities to mirror. A logical volume may consist of several disk drives to protect data from disk drive failures. We do not explicitly deal with disk failures inside a single storage system.

Due to space limitation, we do not describe the recovery mechanisms. This paper focuses only on how TARM uses the knowledge of a file system to asynchronously replicate data on a remote site to ensure data consistency.

3.1 Choice of file systems

The first design choice concerns what kind of file system makes it practical to infer dependency information from only the knowledge of the file system and the data that the file system updates. That is, without accessing information maintained on the host's memory, TARM must get enough dependency information to ensure the consistency of the replicated data. In addition, the overheads to obtain this dependency information should be low to ensure write performance.

We chose a file system using journaling technique for two reasons.

- File systems used in a production environment need to provide efficient asynchronous metadata updates and fast recovery by using journaling or Soft Updates technique, and almost all of them adopt journaling.

- As described in detail later, a journaling mechanism enables the dependency information to be easily inferred because of the simplicity of the data structures. Soft Updates also provides efficient asynchronous metadata updates and fast recovery, though it uses more complex mechanisms and data structures for such a functionality.

In this paper, we present the detailed algorithm for Ext3, as a case study. The Ext3 file system uses on-disk structures similar to those of the Fast File System (FFS) [7], and it enhances them for journaling functionality. It is probably the most widely used journaling file system in Linux.

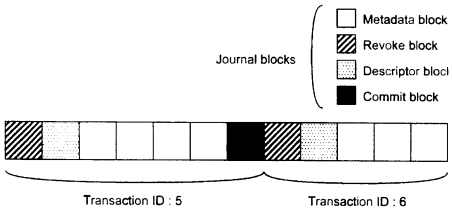
The journaling technique records all metadata operations to a log before the data modified by these operations are written to disk. If the system crashes, operations recorded in the log are replayed to restore the file system to an accurate state.

The ext3 file system does not implement the journaling functionality for itself but uses the *Journaling Block Device* (JBD), which provides the common functionality necessary for journaling file systems in the Linux kernel. The ext3 file system allocates some blocks exclusively for the JBD log. We call such a block *a journal block*. The ext3 file system consists of file systems block and journal blocks. The size of a file system block is equal to that of a journal block.

We explain here the default behavior of the ext3 file system, which requires the JBD to record metadata changes in its log asynchronously.

An important property of the JBD for TARM is that it records the entire copy of the blocks that have been modified instead of recording the individual metadata operations. That is, the JBD has no knowledge about how system calls modify metadata. It just records modified blocks due to system calls. This simplifies the JBD's design.

The ext3 file system merges metadata changes due to several system calls into a single transaction. That is, when several system calls change the same metadata within a certain period of time, the ext3 file system overwrites them and then the JBD records the last state of the metadata in the log. This means the ext3 file



☒ 3: Ext3 file system log example

system cannot isolate the changes of each system call in a single transaction.

Figure 3 shows an example of the JBD log. Suppose that the JBD restores the ext3 file system to an accurate state after a system failure by using the log. A square in the figure represents a single journal block.

Besides a metadata block, JBD use three types of journal blocks internally: *revoke*, *descriptor*, and *commit*. Thus, journal blocks are divided into the following four types.

metadata block a metadata block is a modified metadata block during the transaction. It can be replayed during recovery, that is, written to its actual location on disk.

revoke block A revoke block is used to prevent deleted metadata blocks from being replayed during recovery.

descriptor block A descriptor block tells the JBD where the following metadata blocks are written.

commit block A commit block represents the end of a transaction.

The log in Figure 3 has one finished transaction and one unfinished transaction. The fifth transaction, which has the revoke block and four metadata blocks between the descriptor and commit blocks, is finished. Therefore, the JBD writes these metadata to disk unless the revoke block tells it not to do so. By contrast, the sixth transaction, which does not have the commit block, is unfinished. The system crashes before recording all metadata modified during the sixth transaction. Therefore, these metadata in the log are ignored.

3.2 TARM Algorithm

To simplify the explanation, we start from a situation in which logical volumes on primary and secondary storage systems are synchronized.

3.2.1 Initialization

When the host mounts a journaling file system stored on a logical volume, the primary storage system needs to identify journal blocks by using the knowledge such as the on-disk data structures. That is, the primary storage system must know their actual locations on disk.

The JBD log is stored as a file, called *journal file* (it is not visible or accessible to users). Identifying journal blocks consists of two phases.

Firstly, the primary storage system reads the superblock in the fixed location on disk to know the inode number of the journal file.

Secondly, the primary storage system reads the inode table, which includes the journal file inode, and identify journal blocks by using the knowledge of the disk inode structure.

The JBD log is stored inside the file system itself by default, however the JBD can be configured to store journal blocks in an external block device. In this case, the external block device must be one of logical volumes that the primary storage system provides.

3.2.2 Accessing the primary storage

The primary storage system needs two logical volumes. One is offered to the host, and another is used as the TARM log. the TARM log stores updated data that is sent to the secondary storage system later.

The primary storage system performs the following operations upon receiving a write request from the initiator host.

1. The primary storage system updates the block on the logical volume and records it, its block number, its length in the TARM log.
2. The primary storage system waits for both I/O completions.
3. After the completions, the primary storage system notifies the host that the write request has finished.

When the primary storage system receives a read request, it simply sends some of data on its volume to the initiator. There is no need to access the TARM log.

3.2.3 Transferring updates to the secondary storage

All writes go to the tail of the TARM log. A background process transfers the writes to the secondary

storage system and remove them. This allows the TARM log to wrap around.

To ensure that the replicated data are recoverable, before a commit block in a transaction is written to the secondary storage system, all changed between the first descriptor block in the same transaction and itself must be written.

Firstly, needs a way to find the descriptor and commit blocks among journal blocks.

The JBD puts a magic number in the first four bytes of the revoke, descriptor, and commit blocks. It puts the number for identifying the types of journal blocks in next four bytes. In addition, the JBD guarantees that the magic number never appears in the first four bytes of the metadata blocks stored in the journal blocks.

As explained before, the primary storage system identifies which sectors are used as journal blocks at startup. Thus, the primary storage system can easily find commit journal blocks.

The primary storage system uses task attribute feature in the iSCSI protocol to update commit journal blocks in a precise order on the secondary storage. The iSCSI task attribute is identical to the SCSI task attribute to be used to control the write ordering. Thus, TARM does not need any specialized procedures on the host or the secondary storage system.

Propagation of changes in the primary volume to the secondary volume can be triggered by various policies, such as the amount of changes in the primary volume, the elapsed time from the last synchronization of two copies, or the space in the TARM log.

4 Related work

Some of remote mirroring products available on the market support asynchronous remote mirroring [8][9][10]. However, we are unaware of a remote mirroring system that transparently and asynchronously replicates data in a consistent state and enables secondary storage systems to write data in a favorable order by using the knowledge of a general journaling file system.

SnapMirror [11] is an asynchronous remote mirroring technology. It periodically generates snapshots of the data on the primary storage system, and then asynchronously sends them to a secondary storage system. It exploits the characteristics of the WAFL file system [12], which is updated atomically by using tree-structured on-disk data structures, to ensure the consistency of the replicated data. By contrast, TARM is designed for a general journaling file systems, which do not provide such a functionality.

Note that SnapMirror and TARM are designed for different storage architectures. Whereas SnapMirror is an asynchronous solution implemented by the file system, TARM is one implemented at a storage system. That is, while SnapMirror is a storage system for Network Attached Storage (NAS), TARM is one for a Storage Area Network (SAN). A host uses a remote file system protocol such as NFS to access a storage system for NAS. On the other hand, a host uses any local file systems to access a storage system for a SAN with a block level protocol.

The iSCSI protocol encapsulates the SCSI protocol into the TCP/IP protocol, and it carries packets over IP networks. Whereas the widely used SAN protocol, Fibre Channel, uses specialized networking hardware, the iSCSI protocol uses commodity IP networks. Therefore, it can be used as an inexpensive SAN protocol and also as a wide area storage protocol that moves block data over IP networks.

Ji *et al.* provide an analysis of the various design choices for remote mirroring [13]. They also provide the design of an asynchronous remote mirroring protocol, Seneca, which ensures the consistency of replicated data. Their protocol is general, that is, useful for various applications such as file systems and databases. It assumes that applications use special procedures for remote mirroring to ensure the consistency of the replicated data. Unlike TARM, it does not provide transparent replication feature.

Starfish [14] transparently replicates data over multiple storage sites. It optionally uses asynchronous updates to increase data availability and write performance. However, it does not address the consistency of replicated data with asynchronous updates.

One aspect of our design that we would like to improve further is data placement. Myriad[15] achieves the same level of disaster tolerance as a typical single mirrored solution, but uses considerably fewer physical resources by employing cross-site checksums instead of direct replication.

The Semantically-Smart Disk System (SDS) [16] uses the detailed knowledge of how the file system is using a SDS storage system to improve performance. It mainly addresses how to discover the file system's structure automatically. By contrast, we assume that TARM knows the file system that the host uses on the storage system.

TARM algorithm is designed for journaling file systems. It would be hard to extend and apply the algorithm to other file systems. However, journaling technology is used by most of the successful file systems in the production environment, such as the Linux ext3 file system, the Solaris Unix File System, SGI's XFS, IBM's Journaling File System (JFS), Microsoft's NTFS, and Apple's Mac OS Extended file system (HFS+). We believe the TARM algorithm is generally applicable.

5 Conclusion

Block storage systems lack the knowledge about the relationship among blocks. Thus, ensuring the consistency of replicated data with asynchronous remote mirroring needs a host to use specialized procedures with a storage system or a storage system to maintain the write ordering across the entire data set. Our preliminary experiments show that mirroring performance significantly suffers from the constraint on the write ordering. They indicate that getting rid of the write ordering constraint is essential to achieve a high level of performance.

The TARM storage system is designed for journaling file systems and transparently and asynchronously replicates data across multiple storage sites. It does not require any procedures or special features from the host and the storage systems. TARM ensures that the replicated data are recoverable after catastrophic failures and to permit the secondary storage to write received data in a favorable order.

TARM uses cross-layer optimizations to ensure the consistency of replicated data. It acquires enough information to achieve it by using the knowledge of the file system. We described how TARM achieves such features by taking a popular journaling file system, Ext3, as examples.

References

[1] Satran, J., Meth, K., Sapuntzakis, C., Chadalapaka, M. and Zeidner, E.: Internet Small Computer Systems Interface (iSCSI) (2004). RFC 3720.

[2] Ganger, G. R., Mckusick, M. K., Soules, G. A. N. and Patt, Y. N.: Soft Updates: A Solution to the Metadata Update Problem in File Systems, *ACM Transactions on Computer Systems*, Vol. 18, No. 2, pp. 127–153 (2000).

[3] Seltzer, M., Ganger, G., McKusick, M. K., Smith, K., Soules, C. and Stein, C.: Journaling Versus Soft Updates: Asynchronous Meta-data Protection in File Systems, *the USENIX Annual Technical Conference*, San Diego, CA, pp. 71–84 (2000).

[4] iSCSI Enterprise Target: (2004). `http://iscsitarget.sourceforge.net/`.

[5] Cisco iSCSI initiator Implementation: (2001). `http://linux-iscsi.sourceforge.net/`.

[6] Azagury, A. C., Factor, M. E. and Micka, W. F.: Advanced functions for storage subsystems: Supporting continuous availability, *IBM Systems journal*, Vol. 42, No. 2, pp. 268–279 (2003).

[7] McKusick, M. K., Joy, W. N., Leffler, S. J. and Fabry, R. S.: A Fast File System for UNIX, *ACM Transactions on Computer Systems*, Vol. 2, No. 3, pp. 181–197 (1984).

[8] Veritas Software Corp.: *VERITAS Volume Replicator 3.5: Administrator's Guide (Solaris)* (2002).

[9] IBM Corporation: Total Storage Enterprise Storage Server PPRC Extended Distance (2002). SG24-6568-00.

[10] EMC Corporation: Using EMC SnapView and MirrorView for Remote Backup (2002). Engineering White Paper.

[11] Patterson, H., Manley, S., Federwisch, M., Hitz, D., Kleiman, S. and Owara, S.: SnapMirror: File System Based Asynchronous Mirroring for Disaster Recovery, *the Conference on File and Storage Technologies (FAST 03)*, Monterey, CA, USENIX, pp. 117–129 (2002).

[12] Hitz, D., Lau, J. and Malcolm, M.: File System Design for an NFS File Server Appliance, *the USENIX Winter 1994 Technical Conference*, San Francisco, CA, pp. 235–245 (1994).

[13] Ji, M., Veitch, A. and Wikes, J.: Seneca: remote mirroring done write, *the USENIX Annual Technical Conference*, San Antonio, TX, pp. 253–267 (2003).

[14] Gabber, E., Fellin, J., Flaster, M., Gu, F., Hillyer, B., Ng, W. T., Özden, B. and Shriver, E.: StarFish: Highly-Available Block Storage, *the USENIX Annual Technical Conference*, San Antonio, TX, pp. 151–164 (2003).

[15] Chang, F., Ji, M., Leung, S.-T. A., MacCormick, J., Perl, S. E. and Zhang, L.: Myriad: Cost-effective Disaster Tolerance, *the USENIX Conference on File and Storage Technologies*, Monterey, CA, pp. 103–116 (2002).

[16] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H.: Semantically-Smart Disk Systems, *the USENIX Conference on File and Storage Technologies*, San Francisco, CA, pp. 73–88 (2003).