An Architecture for Public Internet Disks

Fanny Xu and Robert D. Russell
Computer Science Department
University of New Hampshire
Durham, NH 03824, USA
email: {fx,rdr}@cs.unh.edu

Abstract

Because iSCSI technology enables a local SCSI host to exchange SCSI commands and data with a remote SCSI target device over a TCP/IP network, it is possible for independent hosts to directly access independent storage devices over the global Internet in the same manner as they access locally attached private SCSI disk devices, i.e., without any intervening servers. We call storage devices accessible in this manner "Public Internet Disks" (PIDs).

This paper discusses an architecture to implement PIDs. The design involves a balance between processing that can be done on the hosts independently of each other, and processing that is best done on the storage side of the network. Our approach differs from previous proposals in that it requires no change to either the SCSI or iSCSI standards; rather, it utilizes extensibility features of the existing iSCSI protocol. In addition, we introduce a novel locking mechanism that eliminates the need for communication between hosts when resolving contention.

1 Introduction

SCSI [1] is both a well-established standard protocol and a set of standard buses for communication between a host processor and I/O devices. iSCSI [2] is an encapsulation protocol that replaces the SCSI bus with a TCP/IP network, thereby eliminating distance limitations between the iSCSI host "initiator" and the iSCSI I/O device "target". It also enables two or more hosts to share the same iSCSI target, each host connecting to the target via its own TCP connection.

iSCSI is mechanism for transporting SCSI commands and data – it has no notion of files, directories, etc. It does nothing to manage blocks or the contents of blocks, nor does it control how multiple initiators access the blocks – it simply transports them. Each host views an iSCSI target as if it were a local disk device under total control of that host's operating system. With this view, two or more hosts can share the same target as long as all blocks on the disk are read-only, but chaos will result when one or more of the hosts is allowed to write blocks. For example, each host's operating system will allocate blocks on the disk without regard to allocations performed by the other hosts, resulting in corrupted block allocation tables and potential allocation of the same block multiple times. In addition, local caches on some hosts will quickly become out of sync with the disk device, since iSCSI has no provision for maintaining cache consistency across hosts (it knows nothing about caches).

By itself, iSCSI is incapable of coherently sharing a modifiable target disk between multiple platforms. Several solutions to this problem are possible. The most common one, used in many clusters, is to impose a higher-level synchronization mechanism that requires either separate communication channels between the platforms in the cluster, or communication with an intermediate server platform that synchronizes disk access to avoid conflicts. A combination of both approaches is also possible. An alternative approach is to modify the target disk by providing synchronization and conflict resolution mechanisms that can be used by the file systems on the cluster platforms to synchronize themselves. There have been at least two proposals for modifications to SCSI along these lines: DLOCKs [3], and the recent standard for object storage [4], both of which introduce new SCSI commands.

In this paper we propose a new approach that utilizes provisions in the iSCSI standard plus additional functionality in the iSCSI target to accomplish synchronization and conflict resolution without modifications to SCSI and without additional communication between platforms sharing the disk. The basic idea is similar to that of an object store: to split the work of maintaining a file system so that part of the work traditionally done by the operating system on the initiator is now done on the target. Block allocation, metadata management, and directory management are key functions relegated to a new module on the target side of an iSCSI connection. This moves the need for conflict resolution closer to the conflict source, namely the target disk itself, which permits faster resolution with fewer network exchanges. It also eliminates the need for independent communication between hosts, allowing them to be unaware of each other. Of lesser importance in clusters, this is crucial in the Internet, where hosts may be inaccessible to each other.

The cost of this approach is added processing on the target, which could potentially become a bottleneck that limits scalability. There is also the potential for extra communication between the target and host platforms. To minimize this, we have developed a new locking mechanism that in most cases piggy-backs on the I/O operations, eliminating the need for extra iSCSI exchanges. This is discussed in Section 3. Section 4 indicates future areas of work.

2 Background

Frangipani [5] was one of the first cluster file systems. It uses a token-based distributed lock manager, with one lock per file, directory or symbolic link. Deadlocks are avoided by globally ordering locks and acquiring them in two phases: the first to identify the needed locks, the second to sort the locks by the file identification number. Write-ahead logging of metadata is used for failure recovery.

The Global File System (GFS) [6] is a file system for computer clusters based on SCSI and extensions to SCSI that provide a read-write locking mechanism called DLOCKs [3], with one DLOCK per file. Contention for locks is resolved by communication between the hosts. A lock holder has a lease which must be periodically renewed by the holder or else host failure is assumed and recovery is begun by one of the other hosts using journals.

The Global Parallel File System (GPFS) [7] is a parallel file system for supercomputers based on computer clusters. Data locking is controlled by distributed locking that consists of a centralized global lock manager, elected from the set of hosts in the cluster, that coordinates the actions of local lock managers, one on each host. Each block in a file can potentially have its own lock, which permits multiple writers of the same file, provided they are writing different parts of it. For each file, metadata locking and management is controlled by dynamically electing one of the nodes accessing that file as the "metanode" for that file. This metanode merges all metadata changes from separate hosts and updates that file's inode in an atomic fashion. Host failure recovery is accomplished by one of the remaining hosts that reruns the failed host's recovery log.

Storage Tank [8] is a distributed file system for arbitrary collections of hosts. It has two separate paths to storage – hosts must go through special server nodes for file metadata management, but can directly access file data over a separate storage area network. There are three different data locking protocols, depending on the application's needs. Locks can be pre-empted (but a request for pre-emption can be refused by a lock holder), and are designed to maximize sharing local to one host rather than between hosts. Host failure is detected by a lease mechanism, and recovery is accomplished by one of the remaining hosts which polls the others in order to reconstruct the pre-failure state of the failed host.

Antara [9] is a prototype storage system based on the concept of an "object store" [10]. The key idea of an object store is to change the abstraction presented by a storage device from an array of unrelated, fixed-size blocks to a collection of unrelated objects of arbitrary size. This is done by introducing an "object manager" which locates objects and synchronizes access to them in a secure fashion. Hosts must go through the object manager to gain access rights to an object, but data and metadata is dealt with directly between the host and the storage device via operations on objects (for example, when transferring data, the host specifies the amount of data and the offset to the data within the object, but it does not deal with allocation or mapping of logical blocks within the object onto the physical blocks in the store). Management of the metadata for the object is incorporated into the storage device. A standard set of SCSI operations for object-based storage devices has been developed [4].

3 Design

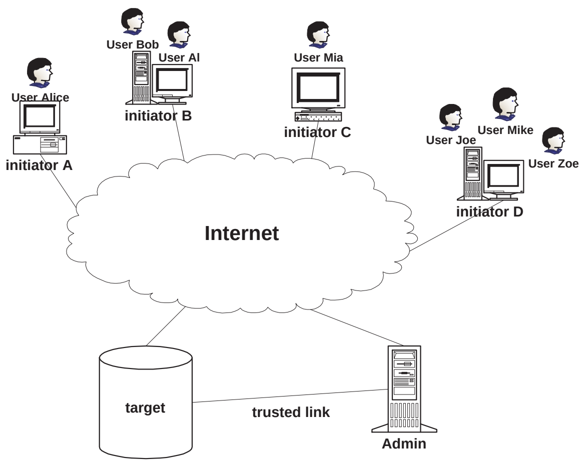

Figure 1: General overview of a Public Internet Disk

Figure 1 shows a picture of one Public Internet Disk. There are three fundamental components, all connected to the public Internet. The "target" represents the storage device; each "initiator" represents a host platform with one or more users; and the "admin" represents a security access control module. This module may be co-located at the target, but is more likely to run on a separate trusted host platform connected to the target by a secure link.

Our design for PIDs has been most heavily influenced by the object store paradigm. The main difference is that rather than requiring a new set of SCSI commands that require new target disk controllers to implement them, we envision using the existing SCSI block storage commands in conjunction with a small set of extensions to iSCSI. The modifications are located in the iSCSI (software) interfaces on the initiator and target, not on the disks or disk controllers. As when using object store, the file system on the initiator must also be modified – see Figure 2.

3.1 Communications Environment

Because the global Internet is a hostile environment, our design requires that there be no communication between hosts using a PID – all communication for accessing or managing the PID occurs only between a host and the target. For security purposes, communication between a host and a trusted admin (and between that admin and the target) is also required.

The iSCSI protocol is an ideal transport mechanism for PIDs because it already exists as an IETF standard [2], because it works over the global Internet, because it already has mechanisms for transport security (IPSec) and access authorization (CHAP), because it is becoming an integral part of most operating systems, and because it contains extension mechanisms by which it can handle the special requirements of PID communication in a standard manner. It enables us to define new keys for negotiating parameter and security information and to define additional header fields that can accompany data transfer and control operations in order to carry extra security and contention resolution information. It also provides message types for sending additional status and control information in both directions.

3.2 Security

There are three aspects of PID security: network security, storage security and access security.

Network security is handled by IPSec, which is utilized by iSCSI. The integrity of data in transit can be ensured by use of an iSCSI CRC32C.

Storage security is handled by having the host file system encrypt and decrypt its data, so that the target does not know the encryption keys and methods.

Access security is handled by storing access authentication and authorization information in a separate security module, shown as the "admin" in Figure 1. Every operation sent to the target is is accompanied by a credential issued by this admin. This credential is based on a cryptographic one-way hash function [11], that permits the target to efficiently verify the requested operation. Keys are issued via a secure link between target and admin.

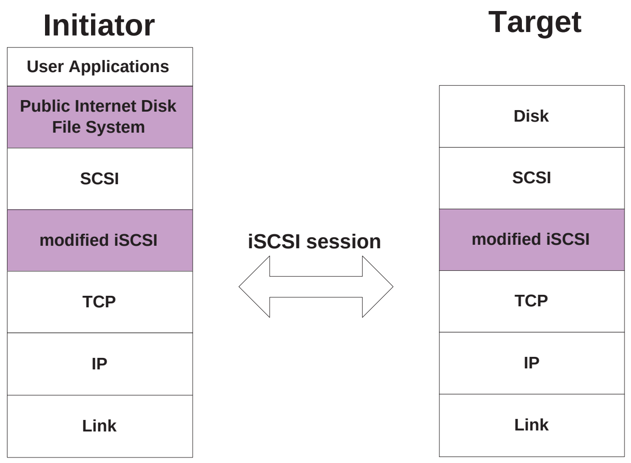

Figure 2: General architecture for implementing Public Internet Disks using iSCSI

3.3 Overview

There are two types of blocks stored on a target disk: data blocks and metadata blocks. The target's modified iSCSI layer controls all block allocation.

Sets of data blocks are read and written directly by host initiators using SCSI read and write commands supplemented with additional iSCSI control information. Data blocks are always part of some file, and are referenced by relative block numbers within that file. The target's modified iSCSI layer maps these onto absolute disk block numbers.

Manipulation of metadata blocks is done exclusively by the target, in response to commands from initiators that reference metadata blocks by absolute disk block numbers. Directories are considered to be metadata blocks, so the target maintains them.

Each file or directory is defined by one metadata block (analogous to an inode in Unix). A metadata block is partitioned into "operational" and "administrative" components. Operational metadata includes information needed by a target to map blocks in the file onto disk data blocks, to map names in a directory onto a metadata block, etc. The target implicitly changes this information as part of many operations, such as create, read, etc. Administrative metadata includes ownership and other information to control access to the file. It is changed only by explicit commands from initiators with appropriate privileges.

To resolve access contention, each command from an initiator provides the target with one or more block references, each accompanied by access "preconditions" that must be satisfied before the command is passed on to the target's SCSI layer (if any pre-condition is not met, the command is enqueued until it can be met), and access "postconditions" that indicate what type of access the target should retain between this initiator and this block after completing the command. The values for preconditions, in order of decreasing "strength", are: "exclusive", "shared", "unneeded". The values for post-conditions are: "unchanged", "demote", "release". Stronger access can never be acquired by a post-condition, so that establishing a post-condition can never cause a blocking situation. References to sets of data blocks require one pre- and one postcondition per set. Metadata blocks require separate pre- and post-conditions for both the operational and administrative components.

A normal read command from an initiator will contain a reference to a set of data blocks within a file, a pre-condition of "shared" for that set and a post-condition of either "unchanged" if the initiator will cache the blocks after transfer from the target, or "release" if it will not. The command must also specify the number of the file's metadata block, with an operational pre-condition of "shared", an operational post-condition of "unchanged", an administrative pre-condition of "unneeded", and an administrative post-condition of "unchanged".

If a read command is part of an update transaction, the read's data block pre-condition is "exclusive" and the post-condition is "unchanged". The subsequent write command would also have a data block precondition of "exclusive", with a post-condition of "unchanged" if the initiator plans further reads or writes on this block, "demote" if the initiator will cache the block, or "release" if not. All commands in the update have the same two metadata pre- and post-conditions as for a normal read. A write command needs "exclusive" access to operational metadata only when it changes a file's length.

The POSIX API for I/O operations does not allow a user to explicitly specify pre- and post-conditions. Therefore, the PID file system must infer the values from the operation and its context (i.e., how a file was opened, etc.). For example, the file system can easily distinguish between writes that update existing blocks and those that append to the end of a file, in order to set the operational metadata pre-condition to "shared" for updating, "exclusive" for appending.

An initiator wishing to cache a block after completion of a command must specify the proper postcondition, which implies that the target must maintain state information. Whenever an initiator flushes a block from its cache, it must inform the target. Similarly, before an initiator can modify a block, the target must send iSCSI asynchronous notification messages to all initiators that have cached that block, instructing them to flush that buffer from their cache.

3.4 Contention Resolution

In order to keep track of the access state of data and metadata blocks, the target instantiates locks associated with a set of blocks whenever a command on that set is queued or in progress, or whenever state is retained due to a previous post-condition. A set of data blocks has one associated lock, a metadata block has two. The lock states are those of a classic read-write lock: “exclusive”, “shared”, “unlocked”.

Each pre- and post-condition effects one lock. To avoid deadlocks, circular dependencies between locks are removed by the following rules. For each operand in a command, locks are ordered such that any operational metadata lock is processed first, then any administrative metadata lock, then any data block lock. If a command has several metadata operands (for example, when a file is moved from one directory to another), the metadata blocks are ordered by their absolute block numbers. A command is delivered to the target’s SCSI layer only after all locks for all operands are in the desired state. A blocked command is queued on exactly one lock. Post-conditions are applied in the reverse order of pre-condition application. As part of acquiring a lock for an initiator, the target may send notification messages to other initiators and may change it’s retained strength between that lock and those initiators.

The decision sequence for processing a pre-condition (other than “unneeded”, which needs no locking) of command C from initiator X at strength s (“exclusive” or “shared”) for lock L is given next (processing completes as soon as one of these actions is taken). C is considered “in progress” for lock L if its pre-condition for L has been satisfied but its post-condition for L has not been processed. During this time, X’s retained strength for L equals L’s state.

1. If L does not exist, create it, set its state to s, no queuing.

2. If L’s state is “unlocked”, set its state to s, no queuing.

3. If L is held for “exclusive” access by X, and no command holding L is currently in progress, and either s is “exclusive” or L’s queue is empty, set L’s state to s, no queuing.

4. If L’s queue is not empty, append C to end of L’s queue.

5. If L’s state is “shared” and s is “shared”, no change to L’s state, no queuing.

6. If another command holding this lock is currently in progress, append C to end of L’s queue.

7. If L is held for “exclusive” access by another initiator Y, append C to end of L’s queue, send notification to Y to “demote” (if s is “shared”) or “release” (if s is “exclusive”) this block.

8. If L is held for “shared” access, send “release” notifications to each other initiator Y currently sharing L, set L’s state to “exclusive”, no queuing.

There are two situations when a target sends a notification to another initiator Y during pre-condition processing, both of which occur when no in-progress command currently holds L and L’s queue is empty: when Y has “exclusive” access; or when Y has “shared” access and s is “exclusive”.

When notified to “release” a “shared” block, an initiator must flush this block from its buffer cache, but it sends no reply to the target because the target changed that initiator’s retained strength to “unlocked” when sending the notification. When notified to “demote” or “release” an “exclusive” block, an initiator must respond with a new command on this block having a pre-condition of “exclusive” and an appropriate post-condition, and must also flush a released block from it’s cache. This command will jump ahead of commands from any other initiator in the queue for this lock (see step 3 above). An initiator asked to “demote” or “release” an “exclusive” block is given a deadline by which to rely – if this is deadline is not met, the target considers the initiator to be dead and forcibly releases all locks held by it.

In SCSI and iSCSI, completion of every command C requires the target to send a reply to the initiator X, after which it uses the following decision sequence to process post-condition p for each lock L held by X with strength s while C was in progress, and to set the retained strength r between X and L.

1. If s is “exclusive” and the first queued command D is from X, set r and L’s state to D’s pre-condition, remove D from queue and continue D;

2. If s is “exclusive” and p is “unchanged”, then: (a) if L’s queue contains a command D from X with pre-condition “exclusive”, remove the first such D from queue and continue D; (b) else if the first queued command requires “exclusive” access, send “release” notification to X; (c) else if the first queued command requires “shared” access, send “demote” notification to X.

3. If s is “exclusive” and p is “demote”, set r and L’s state to “shared”, then: (a) if the first queued command D requires “exclusive” access, send “release” notification to X, set r to “unlocked”, set L’s state to “exclusive”, remove D from queue and continue D; (b) else, while the first queued command D requires “shared” access, remove D from queue and continue D.

4. If s is “exclusive” and p is “release”, set r and L’s state to “unlocked”, then: (a) if the first queued command D requires “exclusive” access, set L’s state to “exclusive”, remove D from queue and continue D; (b) else, while the first queued command D requires “shared” access, set L’s state to “shared”, remove D from queue and continue D.

5. If s is “shared” and p is “unchanged”: (a) if commands holding L are in progress or if L’s queue is empty, do nothing further; (b) else, if the first queued command D requires “exclusive” access, send “release” notification to all initiators holding L, set L’s state to “exclusive”, remove D from queue and continue D; (c) else, while the first queued command D requires “shared” access, remove D from queue and continue D.

6. If s is “shared” and p is “release”, set r to “unlocked”, then: (a) if commands holding L are in progress, do nothing further; (b) else, if L’s queue is empty and no other initiators hold L, set L’s state to “unlocked”; (c) else, if the first queued command D requires “exclusive” access, send “release” notification to all other initiators holding L, set L’s state to “exclusive”, remove D from queue and continue D; (d) else, while the first queued command D requires “shared” access, set L’s state to “shared”, remove D from queue and continue D.

3.5 Performance Considerations

The purpose of having separate block-level pre- and post-conditions for both data and two types of metadata is to increase opportunities for fine-grained parallel access by different host initiators, to try to minimize the situations where contention may arise, and to enable initiator-side caching without requiring initiator-to-initiator communication for maintaining cache consistency. This enables different hosts to simultaneously read and/or write disjoint sets of blocks in the same file, and enables one host to cache data blocks as long as no other host attempts to write those blocks.

To reduce the time and space required for iSCSI target processing, block sizes should be large, 32K or more, and directories are limited to a single block. It would also be beneficial to define more powerful POSIX-level commands, for example, a “copy” that would not require data to cross the network but could be performed entirely on the iSCSI target.

The advantages of initiator caching must be weighed against the cost of bookkeeping and cache-flush notification imposed on both the initiator and target. Caching can be controlled dynamically by both initiator and target – the initiator through the use of appropriate post-conditions, the target through asynchronous notifications to initiators. Experimentation will be necessary to determine effective strategies, but a simple one would be to limit caching to directory metadata blocks, since they are used repeatedly during path name resolution, while data block caching is often more speculative.

4 Summary and Future Work

Public Internet Disks are a new way of making independent storage devices directly available to independent hosts via the Internet without intervening file servers. This paper describes an architecture for PIDs that partitions a traditional file system's functionality between the host and the storage device, and uses extension mechanisms in the iSCSI protocol to carry additional information along with traditional SCSI commands. A novel target locking mechanism based on pre- and post-conditions eliminates the need for host to host communication.

Much work remains to be done to implement this architecture and to evaluate and tune its performance under various operating conditions. It would also be desirable to formally specify the set of file system operations along with their pre- and post-conditions, in order to be able to verify its completeness and correctness, and possibly to generate tools for interoperability testing and performance monitoring.

References

[1] T10 Technical Committee of the NCITS. SAM2, SCSI architecture model – 2. Technical Report T10 Project 1157-D Revision 24, National Committee for Information Technology Standards, September 2002.

[2] J. Satran et al. Internet small computer systems interface (iSCSI). Technical Report RFC3720, Internet Engineering Task Force, April 2004.

[3] Andrew Barry et al. An overview of version 0.9.5 proposed scsi device locks. In *Proceedings of the 17th IEEE/8th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 243–251, March 2000.

[4] T10 Technical Committee of the NCITS. Object-based storage device commands (OSD). Technical Report T10 Project 1355-D Revision 10, National Committee for Information Technology Standards, July 2004.

[5] Chandramohan Thekkath et al. Frangipani: A scalable distributed file system. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 224–237, October 1997.

[6] Kenneth Preslan et al. Implementing journaling in a linux shared disk file system. In *Proceedings of the 17th IEEE/8th NASA Goddard Conference on Mass Storage Systems and Technologies*, March 2000.

[7] Frank Schmuck and Roger Haskin. GPFS: a shared-disk file system for large computing clusters. In *Proceedings of the Conference on File and Storage Technologies (FAST'02)*, January 2002.

[8] Randal Burns. *Data Management in a Distributed File System for Storage Area Networks*. PhD thesis, University of California, Santa Cruz, March 2000.

[9] Alain Azagury et al. Towards an object store. In *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 165–176, April 2003.

[10] Thomas Ruwart. OSD: a tutorial on object storage devices. In *Proceedings of the 19th IEEE/10th NASA Goddard Conference on Mass Storage Systems and Technologies*, pages 21–34, April 2002.

[11] Hugo Krawczyk et al. HMAC: keyed-hashing for message authentication. Technical Report RFC2104, Internet Engineering Task Force, April 2004.