iSCSI Initiator Design and Implementation Experience

Kalman Z. Meth
IBM Haifa Research Lab
Haifa, Israel
meth@il.ibm.com
tel : +972-4-829-6341
fax: +972-4-829-6113

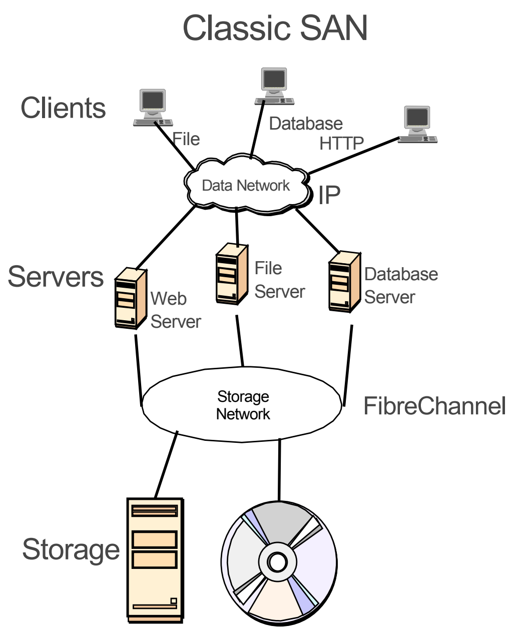
Abstract

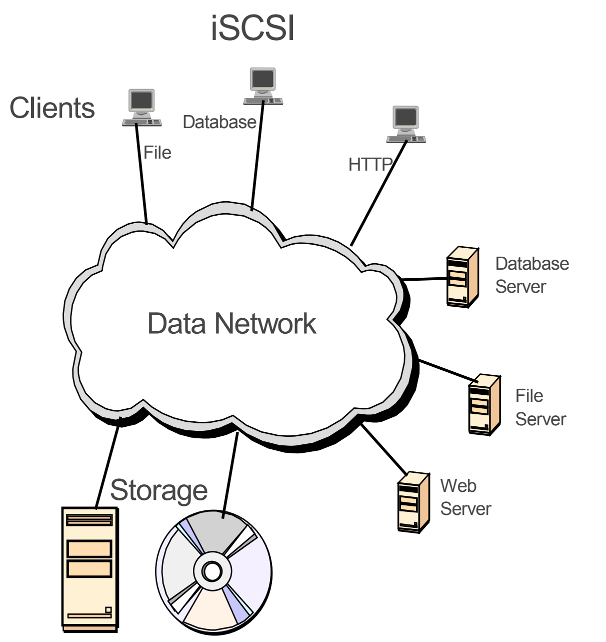
The iSCSI protocol provides access to SCSI devices over a TCP transport. Using the iSCSI protocol enables one to build a Storage Area Network using standard Ethernet infrastructure and standard networking management tools. This paper outlines how we implemented a family of iSCSI initiators on a common core. The initially supported initiators were on the Windows NT and the Linux Operating Systems. Code for a version of the Linux iSCSI initiator has been released as Open Source. Initial testing indicates that iSCSI can provide reasonable performance relative to traditional storage environments.

1. Introduction

1.0 SANs and iSCSI

Storage Area Networks (SANs) provide a way for multiple hosts to access a shared pool of remote storage resources. Traditional SANs are built using FibreChannel technology [1] running the FCP protocol to carry SCSI commands over the FibreChannel network. Two separate network infrastructures are needed in an environment that uses a Local Area Network (LAN) for usual network activity and a SAN for shared remote: an Ethernet (or equivalent) infrastructure (running TCP and similar protocols) for usual LAN activity, and a FibreChannel infrastructure (running FCP protocol) for the SAN activity. iSCSI [2] is a protocol that carries SCSI commands over the TCP protocol [3]. iSCSI enables access to remote storage devices using TCP over standard LAN infrastructures. Using iSCSI dispenses with the need for a separate FibreChannel infrastructure and the need for a separate set of FibreChannel management tools. The difference between a traditional SAN and a possible iSCSI setup is depicted in the following figure.





1.1 iSCSI Overview

The iSCSI protocol [2] is a mapping of the SCSI remote procedure invocation model (see SAM2 [4]) over the TCP protocol [3]. Communication between the initiator and target occurs over one or more TCP connections. The TCP connections carry control messages, SCSI commands, parameters and data within iSCSI Protocol Data Units (iSCSI PDUs). The group of TCP connections that link an initiator with a target form a "session". The SCSI layer builds SCSI CDBs (Command Descriptor Blocks) and relays them with the remaining command execute parameters to the iSCSI layer. The iSCSI layer builds iSCSI PDUs and sends them over one of the session's TCP connections.

1.2 Design Goals

We designed and implemented a family of iSCSI initiators. Initial testing was performed against an IBM TotalStorage 200i disk controller using a standard 100Mbit Ethernet network connection. A major design goal of our initiators was to allow for multiple Operating Systems to work on the same common code base. Each operating system has its own set of interfaces for the SCSI subsystem and for its TCP transport. However, the implementation of the iSCSI specification should be common to all operating systems. When we upgrade to a different level of the iSCSI specification, only the common core needs to change, while the OS-dependent code should remain more or less intact.

Additional design considerations of our initiators included:

- Allow the initiator to simultaneously use devices from multiple target machines.
- Utilize multiple TCP connections between and iSCSI initiator and target.
- Utilize multiple processors if running on an SMP (Symmetric Multiprocessor).

1.3 Design Assumptions

The common core was designed and written using some basic assumptions about the Operating System (OS) on which it would be run. We assumed that the base Operating System would have the following features:

- Support for multiple threads in the kernel.
- Reading/writing from TCP can be easily abstracted into a single read/write function call.
- Some SCSI commands might be (re-) issued from inside the scsi completion routine, thus possibly running at some priority level for which we may not block.
- Task Management requests may arrive at some priority level, and hence we must provide an implementation that does not block, if requested.

We also had in mind a certain layering of the SCSI subsystem that seems to be prevalent in a number of Operating Systems. In both the Linux and Windows NT operating systems there are 3 layers to the SCSI subsystem. There is one high-level (class) driver for each type of SCSI device: disk, tape, CD, etc. There is a mid-level (port) driver that has common code for all types of devices, which takes care of command timeouts and serialization of commands. The low-level (miniport) driver is specific to an adapter, and must provide a queuecommand() or dispatch() routine that is used by the mid-level driver. This 3-level layering is essentially the model that is presented in the Common Access Method [5].

2. Implementation Details

2.1 Data Type and Function Abstractions

In order to build a common core, we abstracted the basic Operating System dependent data types and services that we would need to use, and defined these individually for each Operating System on which we implemented the iSCSI initiator driver. The basic data types that we defined are described here with their corresponding Linux (2.2) definitions.

```
typedef spinlock_t              iscsiLock_t;    /* basic spin lock */
typedef struct wait_queue*              iscsiEvent_t;   /* sleep event */
typedef uchar                   iscsiIrql_t;    /* interrupt level */
typedef struct scsi_cmnd                SCB_t;          /* SCSI Command Block */
typedef struct {                /* TCP connection descriptor */
        struct sockaddr_in      sin;
        struct socket           *sock;
        iscsiEvent_t            event;
} iscsiSock_t;
```

The basic services that must be provided by each Operating System and their corresponding Linux implementation are:

```
#define iscsiOSmalloc(size)   kmalloc(size, GFP_KERNEL)
#define iscsiOSlock(lock, irql)         spin_lock_irqsave(lock, (*irql))
```

```
#define iscsiOSunlock(lock, irql)       spin_unlock_irqrestore(lock, irql)
#define iscsiOSsleep(event)   sleep_on(event)
#define iscsiOSwakeup(event)          wake_up(event)
```

The common core uses these macros, which enable us to write code that is common to multiple platforms. The common core must also call some functions to perform TCP operations. Their prototypes are given below.

```
s32 iscsiOSreadFromConnection(iscsiSock_t *sock, void *buffer, u32 len, u32 offset);
s32 iscsiOSwriteToConnection(iscsiSock_t *sock, void *header, u32 headerLen, void
*buffer, u32 len, u32 offset);
s32 iscsiOSmakeConnection(u32 addr, u16 portNum, iscsiSock_t *isock);
void iscsiOScompleteCommand(SCB_t *scb, u32 status);
```

The core provides a number of routines that the OS-dependent layer can call. The prototypes of the main core functions are given below.

```
s32 iscsiCoreCreateSession(u32 addr, u16 portNum, u32 nConnections, uchar
*loginParams, u32 loginParamsLen);
s32 iscsiCoreEnterCmdInQ(SCB_t *scb, void *cdb, u32 cdbLen,  u32 sessionhandle,
iscsiLUN_t lun, void *data, u32 datalen, u32 flags);
u32 iscsiCoreResetDevice(u32 sessionHandle, SCB_t *scb, iscsiLUN_t lun);
```

The OS-dependent code calls iscsiCoreEnterCmdInQ() for each command that it wants to send to the target. The core then takes over and processes the command, sending it to the target, receiving a response from the target, and reporting the results back to the OS-dependent code by calling the iscsiOScompleteCommand() function.

2.2 Common Core General Structure

For each session established by the initiator (i.e. for each target), the initiator maintains a queue of items that must be sent to the target. We call this queue the command queue. The initiator also maintains a dedicated command queue handler thread to read items from this queue and to send them to the target.

The initiator maintains state of each command that has been sent to a target. This state information is saved in a table, indexed by an Initiator Task Tag. The target may send status or R2T (Ready to Transfer) PDUs to the initiator relating to a particular command. The Initiator Task Tag is used to easily look up the relevant information in the table.

For each TCP connection (even if we have multiple connections for a single session) the initiator maintains a dedicated thread to read from that TCP connection. The use of separate threads to read from each TCP connection and to send out messages allows us to better exploit the CPUs while waiting for data to arrive or be sent over a network connection. A read thread posts a read request on its TCP connection to receive an iSCSI header. The thread waits until data has arrived and has been placed in its buffer. The read thread parses the iSCSI header to determine how much additional data follows the header. The read thread then posts a read request for the data of specified length, providing an appropriate buffer into which the data is to be placed. The read thread then performs whatever processing is needed to handle the PDU.

2.3 Implementation Lessons

In this section we briefly discuss some problems we encountered and lessons we learned in our implementation.

Windows NT expects command completion to occur from within an interrupt handler. Since we did not have any real hardware of our own, and all of our internal threads ran at regular priority, we broke a basic assumption of the Windows NT SCSI subsystem. We had to artificially create an interrupt in order to get the Windows NT SCSI subsystem to complete the processing of commands that were handled by our driver.

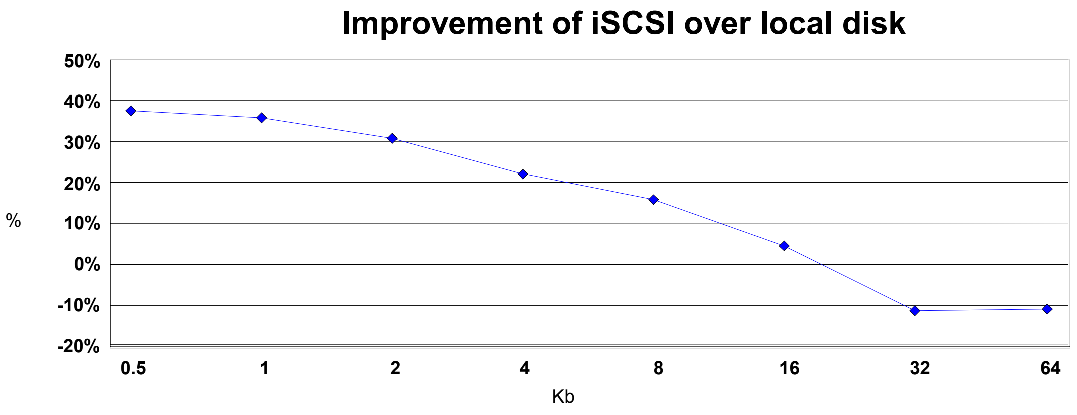
Linux also expects commands to be completed in a type of interrupt handler. A certain lock must be obtained and interrupts must be blocked when calling the Linux SCSI command completion handler.

In Linux, SCSI commands might be issued from within an interrupt handler. The call to iscsiCoreEnterCmdInQ() might therefore be called from within an interrupt handler, and any locks that we obtain in that routine must be safe to obtain and contend with an interrupt handler. It is therefore necessary to block interrupts whenever we obtain locks that may also be obtained at interrupt level inside the iscsiCoreEnterCmdInQ() function.

There is an inherent problem in mounting and unmounting iSCSI disks automatically upon reboot. In general, when the system first tries to mount its file systems, the network isn't yet up, thus preventing us from reaching our iSCSI disks. Also, the disk cannot automatically unmount cleanly during shutdown since by the time the system tries to sync its disks the network may already be gone.

2.4 Performance

We performed some preliminary measurements of iSCSI performance versus a directly attached IDE disk. We ran the IOTEST benchmark [7] on Linux for different sized block transfers and compared the results. The following graph shows the relative number of read operations between the iSCSI and local IDE disk.



For small data transfers iSCSI outperformed the local IDE disk by about 30%, despite the network overhead. This is due to the higher performance SCSI disks on the TotalStorage 200i target. Using the TotalStorage 200i SCSI disk locally outperformed iSCSI by about 3% for small transfers. For large data transfers, the network overhead started to take on a larger and larger impact, causing the iSCSI performance to be up to 12% worse than the local IDE disk. This is apparently due to the numerous interrupts needed for the packaging and processing of many small TCP packets for a large data transfer. This phenomenon should be alleviated when using Network Interface Cards (NICs) that offload the TCP processing, thereby reducing the number of interrupts that must be handled by the host.

3. Related Work

Network Storage is becoming more and more common, allowing remote hosts to more easily access remote and shared data. A number of studies have been performed that show that IP attached network storage can provide reasonable performance relative to FibreChannel and other storage environments. See, for example, [8] [9] [10].

Other early iSCSI initiator drivers have been made available as Open Source [6]. Some of these early implementations support a fixed target with a single TCP connection. Some of these implementations were written and tested for uniprocessors, and could not take advantage of the multiple processors in an SMP. Our implementation has the distinction of allowing multiple targets, multiple TCP connections within each session to a target, and the ability to fully exploit SMP machines. The performance achieved on a uniprocessor by the other software iSCSI initiator implementations that we tested (against the same target) were essentially the same as for our initiator.

4. Future Work

A version of our Linux initiator has been released as Open Source [11]. We continue to revise our driver to keep up with changes in the iSCSI specification as it evolves. Over time, we are adding additional features that are defined in the specification. We intend to perform comprehensive performance measurements and adjust our driver accordingly.

5. Conclusion

We implemented a family of iSCSI initiators utilizing a common core. We outlined our basic design objectives and how we implemented our initiators. When we upgraded to a newer version of the iSCSI specification, we were able to perform the necessary changes to the common core to correspond to the new iSCSI specification, while the OS-dependent parts of the code remained essentially unchanged. Our implementation allows multiple targets, multiple TCP connections within each session to a target, and the ability to exploit SMP machines. Using the IBM TotalStorage 200i target, iSCSI significantly outperforms the local IDE disk for small data transfers, but lags behind IDE for large data transfers, apparently due to the extra overhead of network interrupts.

Acknowledgement
The author has had the benefit of interacting and drawing on the experience, ideas, and help of many people that were involved in the iSCSI project. The author would especially like to thank Zvi Dubitzky, Eliot Salant and Liran Schour for their contributions.

References

[1] Benner, A. *Fibre Channel: Gigabit Communications and I/O for Computer Networks*, McGraw Hill, New York, 1996.

[2] Julian Satran, et al, iSCSI (Internet SCSI), *IETF draft-ietf-ips-iscsi-10.txt* (Jan 20, 2002); see www.ece.cmu.edu/~ips or www.haifa.il.ibm.com/satran/ips

[3] RFC793, Transmission Control Protocol, DARPA Internet Program, Protocol Specification, Sep 1981.

[4] SAM2, SCSI Architecture Model – 2, T10 Technical Committee NCITS (National Committee for Information Technology Standards), T10, Project 1157-D, Revision 20, 19 Sep 2001.

[5] CAM, Common Access Method – 3, draft of American National Standard of Accredited Standards Committee X3, X3T10, Project 990D, Revision 3, 16 Mar 1998.

[6] See www.ece.cmu.edu/~ips/IPS_Projects/ips_projects.html.

[7] See www.soliddata.com/products/iotest.html.

[8] Rodney Van Meter, Greg Finn, and Steve Hotz, "VISA: Netstation's Virtual Internet SCSI Adapter," in *Proceedings of the ACM 8th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (San Jose, Calif., Oct.), ACM Press, New York, 1998, 71-80. see also www.isi.edu/netstation.

[9] Wee Teck Ng, Hao Sun, Bruce Hillyer, Elizabeth Shriver, Eran Gabber, and Banu Ozden, "Obtaining High Performance for Storage Outsourcing", *FAST 2002, Conference on File and Storage Technologies*, Jan 2002.

[10] Garth A. Gibson and Rodney Van Meter, "Network Attached Storage Architecture", *Communications of the ACM*, Nov 2000, vol 43, no. 11.

[11] See oss.software.ibm.com/developerworks/projects/naslib.