Design, Implementation, and Performance Analysis of Session Layer Protocols for SCSI over TCP/IP

Anshul Chadda

Robert D. Russell

TR 01-06

August 2001

Table of Contents

List of Tables v
List of Figures vi
ABSTRACT viii

CHAPTER PAGE

1. Introduction 1
   1.1 Motivation and Goal for this Thesis 1
   1.2 Resources Used 1
   1.3 Organization of the Thesis 2

2. SCSI, NAS, and SAN 3
   2.1 SCSI (Small Computer System Interface) 3
      2.1.1 Phase sequences in the SCSI Protocol 4
      2.1.2 Tasks 6
      2.1.3 Command Descriptor Block (CDB) 6
      2.1.4 Task Management Functions 7
   2.2 Network Attached Storage (NAS) 8
   2.3 Storage Area Network (SAN) 9
   2.4 SAN Approaches 10
      2.4.1 SCSI on top of SAN on top of Ethernet 10
      2.4.2 SCSI on top of SAN on top of IP 11
      2.4.3 SCSI on top of SAN on top of UDP 11
      2.4.4 SCSI on top of SAN on top of TCP 12
   2.5 SAN Approach for this thesis 12

3. Session Layer Protocols 15
   3.1 SCSI Encapsulation Protocol (SEP) 15
   3.2 Internet SCSI (iSCSI) 17
      3.2.1 Steps involved in iSCSI Protocol to support I/O operation 17
      3.2.2 iSCSI PDU format 20
      3.2.3 Sequence Numbering 20
      3.2.4 Error Recovery 23
      3.2.5 Important Operational Parameters 23

4. Fast Kernel Tracing 25
   4.1 FKT Setup 25
   4.2 Application Programmer Interface 26
   4.3 Kernel Programmer Interface 27
   4.4 Recording and Printing 27

5. SCSI SubSystem in Linux 29
5.1 SCSI Layers in Linux 29
5.1.1 Upper level 29
5.1.2 Middle level 30
5.1.3 Low-level drivers 30
5.2 Important SCSI Data Structures 30
5.2.1 Scsi_Host_Template struct 30
5.2.2 `Scsi_Host` struct 34
5.2.3 Scsi_Cmnd struct 35
5.3 Discovery Process for SCSI Targets 37

6. SEP Initiator Design And Implementation 38
6.1 Overview of SEP Low-Level Driver (LLD) Design 38
6.2 sep_config 40
6.3 Data structures involved in SEP LLD 40
6.4 The SCSI_Host_Template Implementation 42

7. iSCSI Initiator Design And Implementation 46
7.1 Overview of iSCSI Low-Level Driver (LLD) Design 46
7.2 Data structures involved in iSCSI LLD 48
7.2.1 session struct 49
7.2.2 connection struct 51
7.2.3 command struct 51
7.3 The SCSI_Host_Template Implementation 53
7.4 Low-level iSCSI Driver Design 57
7.4.1 Processing a WRITE SCSI Command 57
7.4.2 Processing a READ SCSI Command 61
7.4.3 Processing an ABORT Command 62

8. Performance Analysis 64
8.1 Test Set-Up 64
8.1.1 CPUs 64
8.1.2 Ethernet Technologies 64
8.1.3 Fiber_Channel Technologies 65
8.1.4 Version of Linux Operating System 65
8.2 Performance Metrics 65
8.3 Performance Variables 65
8.4 Accuracy of Data 67
8.4.1 Side Effects of Adding Probes 67
8.4.2 Confidence Level of Data 67
8.5 Ethernet Payload 68
8.6 Performance Results for SEP 68
8.6.1 Effect of Target Domain on Bandwidth for SEP 69

8.6.2 Effect of Target Block Size on Bandwidth for SEP 72
8.6.3 Effect of Initiator Scatter Gather List Size on Bandwidth for SEP 74
8.6.4 Effect of Ethernet Link Speed on Bandwidth for SEP 75
8.6.5 Effect of Ethernet Packet Size on Bandwidth for SEP 76
8.6.6 Effect of Coalescing Interrupt Time Interval on Bandwidth for SEP.............. 77
8.6.7 Effect of Old (Alteon) and New (3-Com) Acenic Cards on Performance....... 78
8.7 Performance Results for iSCSI 78
8.7.1 Effect of LLD Queuing Length on Bandwidth for iSCSI................................ 79
8.7.2 Effect of Max PDU Size on Bandwidth for iSCSI 81

9. Conclusions and Future Work 83
9.1 Conclusions.. 83
9.2 Future Work 84

References.. 85
APPENDIX A 87
APPENDIX B 91
APPENDIX C 92

List of Tables

Table Page

2.1 Device Involvement in Information Transfer Phases. 6
2.2 Task Management Response Values. 7
7.1 Session State Table. 50
7.2 Task Management Response Values. 52
7.3 Task Management Function Values.. 52
8.1 Statistics on Bandwidth with Target in Kernel Mode.. 70
8.2 Analysis Table produced by fkt_print for SEP Low-Level driver............................. 72
8.3 % CPU utilization comparison for Target Domain change. 72
8.4 % CPU utilization comparison for Target Block Size change.................................... 73
8.5 % CPU utilization comparison for Scatter-gather list size change. 74
8.6 % CPU utilization comparison for Ethernet Link Speed Change.............................. 75
8.7 % CPU utilization comparison for Ethernet Packet Size change. 76
8.8 % CPU utilization comparison for CITI change... 77
8.9 Effect of Old (Alteon) and New (3-Com) Acenic Cards on Performance.................. 78
8.10 Analysis Table produced by fkt_print for iSCSI Low-Level driver. 80
8.11 % CPU utilization comparison for CITI change... 81
8.12 % CPU utilization comparison for LLD queuing length. 82

List of Figures

Figure Page

2.1 Representation of a SCSI System. 4
2.2 SCSI Protocol.................... 4
2.3 Information Transfer Phases. 5
2.4 10-Byte CDB. 6
2.5 Network Attached Storage (NAS). 8
2.6 Storage Area Network (SAN). 9
2.7 Fiber Channel Solution to Storage Area Network (SAN). 10
2.8 Our Approach to Storage Area Network (SAN). 12
2.9 Components involved to support SCSI. 13
3.1 SEP Header. 16
3.2 iSCSI Basic Header Segment (iSCSI draft ver 7). 21
3.3 iSCSI Basic Header Segment(BHS) for SCSI Command (iSCSI draft ver 7). 22
3.4 iSCSI Additional Header Segment(AHS) for SCSI Command. 22
5.1 SCSI SubSystem in Linux. 29
5.2 Scsi_Host_Template struct Definition. 31
5.3 API between SCSI Mid-Level and Low-Level driver. 33
5.4 Scsi_Host struct Definition. 34
5.5 Scsi_Cmnd struct Definition. 35
6.1 SEP Low-Level Driver (LLD) Design. 39
6.2 Organization of Data Structures in SEP LLD. 41
6.3 `sep_control_block` struct Definition. 42
6.4 `SCSI_Host_Template` struct Definition. 43
6.5 API between SCSI Mid-Level and SEP Low-Level Driver. 44
7.1 iSCSI Low-Level driver (LLD) Design. 47
7.2 Organization of data structures in iSCSI LLD. 49
7.3 `session` struct Definition. 50
7.4 connection struct Definition. 51
7.5 `command` struct Definition. 52
7.6 Scsi_Host_Template struct Definition. 53
7.7 API between SCSI Mid-Level (SML) and iSCSI Low-Level Driver (LLD). 55
7.8 Processing of SCSI Command. 58
7.9 Data Processing for WRITE Command. 60
7.10 Culmination of SCSI Command. 61
7.11 Data Processing for READ Command. 62
7.12 Processing an ABORT Command. 63
8.1 Ethernet Frame for a TCP. 68
8.2 TCP TimestampOption. 68

8.3 Effect of Target Domain on Bandwidth for SEP. 69
8.4 Effect of Target Block Size on Bandwidth for SEP. 73
8.5 Effect of Initiator Scatter-Gather List Size on Bandwidth for SEP 74
8.6 Effect of Ethernet Link Speed on Bandwidth for SEP 75
8.7 Effect of Ethernet Packet Size on Bandwidth for SEP. 76
8.8 Effect of CITI on Bandwidth for SEP. 77
8.9 Effect of LLD Queuing Length on Bandwidth for iSCSI 79
8.10 Effect of Max Data PDU Size on Bandwidth for iSCSI. 81

ABSTRACT

Design, Implementation, and Performance Analysis of Session Layer Protocols for SCSI over TCP/IP

By

Anshul Chadda
University of New Hampshire, December 2001

In the last decade, there has been a demand in the computer industry to physically separate storage from the main computing system. The motivation for this thesis came from the desire for higher performance and greater scalability in Storage Networking. There are two approaches in the industry to support storage across networks: Network Attached Storage (NAS) and Storage Area Networks (SAN). NAS intercepts the communication between application (host) and storage at a fairly high level, because the application typically accesses the remote storage via a special file system that in turn utilizes a standard network protocol stack. The unit of access is a file managed by the NAS. A SAN intercepts the communication between application (host) and storage at a fairly low level, because the host views the remote storage as a device that is accessed over an I/O channel capable of long-distance transfers. The unit of access is a 'raw' storage block managed by the SAN. Small Computer System Interface (SCSI) is a ubiquitous and popular disk technology that supports block level access. The major limitation of SCSI is the length of the SCSI bus. Several protocols have been proposed to extend the length of the SCSI bus: Fiber Channel (FC), SCSI Encapsulation Protocol (SEP) and Internet SCSI (iSCSI). Although FC fits into the requirement of high performance, it has limited scalability as it needs a specialized network of storage devices. SEP and iSCSI are the two Session Layer Protocols to support SCSI on top of TCP/IP/Ethernet networks and hence enhancing scalability of SANs.

The objective of this thesis is to design, implement, and evaluate SEP and iSCSI. As an overview, the two protocols get the SCSI I/O requests in the kernel on the host (application) and transfer it over the TCP/IP network to the target (device). The SEP and iSCSI low-level drivers are designed and implemented for the initiator to support the latest protocol drafts. An existing target emulator is modified and extended to support the additional features specified in the latest iSCSI draft. The evaluation of the two protocols involves basic performance analysis using software tracing facilities in the Linux kernel. The results suggest the optimum performance parameter values for I/O operation over generic TCP/IP/Ethernet Networks.

Chapter 1

Introduction

1.1 Motivation and Goal for this Thesis

In the last decade, there has been a demand in the computer industry to physically separate storage from the main computing system. The motivation for this thesis came from the desire for higher performance and greater scalability in Storage Networking. There are two approaches in the industry to support storage across networks: Network Attached Storage (NAS) and Storage Area Networks (SAN). NAS intercepts the communication between application (host) and storage at a fairly high level, because the application typically accesses the remote storage via a special file system that in turn utilizes a standard network protocol stack. The unit of access is a file managed by the NAS. A SAN intercepts the communication between application (host) and storage at a fairly low level, because the host views the remote storage as a device that is accessed over an I/O channel capable of long-distance transfers. The unit of access is a 'raw' storage block managed by the SAN. Small Computer System Interface (SCSI) is a ubiquitous and popular disk technology that supports block level access. The major limitation of SCSI is the length of the SCSI bus. Several protocols have been proposed to extend the length of the SCSI bus: Fiber Channel (FC), SCSI Encapsulation Protocol (SEP) and Internet SCSI (iSCSI). Although FC fits into the requirement of high performance, it has limited scalability as it needs a specialized network of storage devices. SEP and iSCSI are the two Session Layer Protocols to support SCSI on top of TCP/IP/Ethernet networks and hence enhancing scalability of SANs.

The objective of this thesis is to design, implement, and evaluate SEP and iSCSI. As an overview, the two protocols get the SCSI I/O requests in the kernel on the host (application) and transfer it over the TCP/IP network to the target (device). The SEP and iSCSI low-level drivers are designed and implemented for the initiator to support the latest protocol drafts. An existing target emulator is modified and extended to support the additional features specified in the latest iSCSI draft. The evaluation of the two protocols involves basic performance analysis using software tracing facilities in the Linux kernel. The results suggest the optimum performance parameter values for I/O operation over generic TCP/IP/Ethernet Networks.

1.2 Resources Used

The facilities for the thesis project are provided by the InterOperability Lab, University of New Hampshire. The resources used for this project are two high speed Linux PCs, one serving as an initiator and other as an target. The initiator system has a Gigabit Ethernet Network Interface Card (Company: Alteon Acenic) to connect to the target. The target

system has a Gigabit Ethernet Network Interface Card (Company: Alteon Acenic card) to connect to the initiator and a Fiber Channel Card (Company: Qlogic Corporation ISP2200 A) to connect to a Fiber Channel Disk.

1.3 Organization of the Thesis

Chapter 2 explains the SCSI protocol features, advantages, and disadvantages. The Storage Area Networks (SAN) and Network Attached Storage (NAS) concepts are also explained. Different SAN approaches over a generic TCP/IP/Ethernet Network and the SAN approach followed for this thesis are discussed. Chapter 3 explains the Session Layer Protocols used to support SCSI over TCP. The two protocols discussed are SCSI Encapsulation Protocol (SEP) and Internet SCSI (iSCSI). Chapter 4 explains the Fast Kernel Tracing (FKT) Software Probes used to do performance analysis of SEP and iSCSI Implementations. Chapter 5 explains the Organization of SCSI Subsystem in Linux. The Interface provided by SCSI to the low-level driver is elaborated. Chapter 6 explains the design and implementation of the SEP Initiator Implementation. Chapter 7 discusses the design and implementation of the iSCSI Initiator Implementation. Chapter 8 goes into the Performance Analysis of SEP and iSCSI using FKT Software Probes. Chapter 9 summarizes the work done and the conclusions drawn along with work that can be done in the future. Appendix A explains the additions and changes made to the target to support the latest iSCSI draft (version 7).

Chapter 2

SCSI, NAS, and SAN

In this section, we discuss the SCSI (Small Computer System Interface) technology and bring out its advantages over other existing disk technologies. The concepts of Network Attached Storage and Storage Area Networks are elaborated next. Finally, the possible approaches to support SAN and the approach followed in this thesis are discussed.

2.1 SCSI (Small Computer System Interface)

SCSI [1] is a universal disk drive interface which is much more advanced than its chief competitor, IDE/ATA [2]. SCSI is a system-level bus, with intelligent controllers on each SCSI device working together to manage the flow of information on the channel. SCSI supports many different types of devices, and is not at all tied to hard disks the way IDE/ATA is—ATAPI supports non-hard-disk IDE devices but it is really a kludge of sorts. SCSI has several advantages over IDE that make it preferable for many situations, usually in higher-end machines. It is far less commonly used than IDE/ATA due to its higher cost.

The basic feature of SCSI is to give the computer complete device independence. With SCSI, the system should not need any modification when replacing a device from one manufacturer with that from another manufacturer. The burden of being able to manipulate the peripheral specific hardware shifts from the host system to the peripheral device. As a result, development cycles are significantly reduced.

The kinds of SCSI devices are numerous: interface cards, hard disks, CD-ROMs, and scanners. All of them fall into two fundamentals categories: initiators and targets (Fig. 2.1). The initiator device is also called the host, and it starts or initiates device-to-device communication. The target device receives the communication from the initiator and responds. For example, when reading a file from a SCSI hard disk, the SCSI interface card (the initiator) requests data from the SCSI hard disk and the hard disk (the target) responds to the request by sending the data. This is the most common initiator-target interaction in a SCSI system.

SCSI systems can have up to eight devices connected in a daisy chain on a 8-bit SCSI bus (16-bit Wide SCSI can have up to sixteen devices). These devices can be any combination of initiators and targets, but at least one must be an initiator and one a target in order to have a useful system.

The SCSI protocol specifies a unique kind of identification called a SCSI ID for each SCSI device on the bus. These IDs, or addresses, identify each device so that the command requests go to the right device. Without this identification, there would be no way to know where to send commands and data along the bus and no way to direct signals to a specific device. A 16-bit Wide SCSI bus allows a maximum of 16 devices, with IDs ranging from 0 to 15; and 32-bit Wide SCSI allows for 32 devices, ranging from 0 to 31.

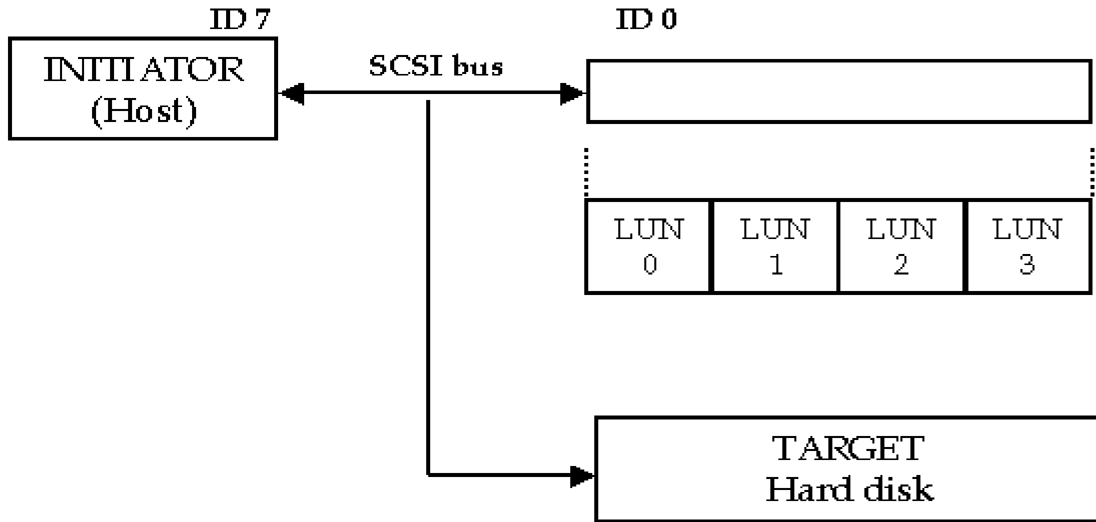


Fig 2.1 Representation of a SCSI System.

Each Target device can also be subdivided into several Logical Units (LUNs). Logical units represent devices within devices and are divisions within IDs. The maximum number of LUNs on a device, except the Initiator, cannot be more than 8. So, the maximum number of LUNs that can be supported on a 8-bit Wide SCSI is 57 (1 Initiator + 7 Targets x 8 LUNs).

2.1.1 Phase sequences in the SCSI Protocol

SCSI uses a method to transfer data between devices on the bus in a circular process that starts and ends in the same layer. The process overview is shown in Fig. 2.2.

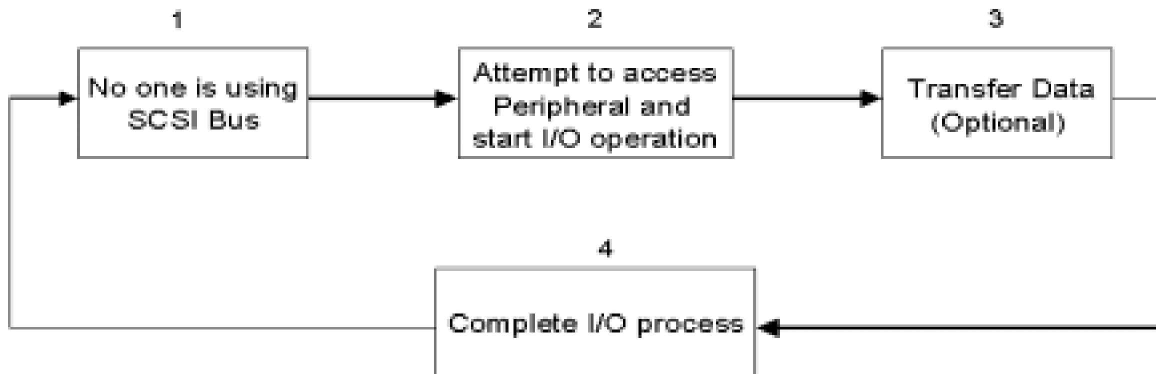


Fig 2.2 SCSI Protocol.

The SCSI architecture includes eight distinct phases for communication between the initiator and the target:
BUS FREE Phase (Step 1 in Fig. 2.2): The SCSI devices (initiators and targets) use this phase to recognize bus availability.
ARBITRATION Phase (Step 2 in Fig. 2.2): The SCSI devices (initiators and targets) use this phase to gain control of the bus. In other words, this phase resolves bus contention in order to access the bus.
SELECTION Phase (Step 2 in Fig. 2.2): The initiator uses this phase to select targets in order to start an I/O process.
RESELECTION Phase (Step 2 in Fig. 2.2): The target uses this phase to continue a previously disconnected I/O process with an initiator.
COMMAND Phase (Step 2 in Fig. 2.2): In this phase, the target requests Commands, in the form of Command Descriptor Blocks, from the initiator. Command Descriptor Block is discussed in detail in Section 2.1.3.
DATA Phase (Step 3 in Fig. 2.2): Depending on the data direction, the phase can be of either two types, DATA IN and DATA OUT. The DATA IN phase allows the target to send data to the initiator. The DATA OUT phase allows the initiator to send data to the target.
STATUS Phase (Step 4 in Fig. 2.2): This phase allows the target to send status information for any Command to the Initiator.
MESSAGE Phase (Step 2 or 4 in Fig. 2.2): This phase is divided into two types, MESSAGE IN and MESSAGE OUT. In MESSAGE OUT phase (Step 2 in Fig. 2.2), the Initiator transmits a message to a target. In MESSAGE IN phase (Step 4 in Fig. 2.2), the Target transmits a message to an initiator.

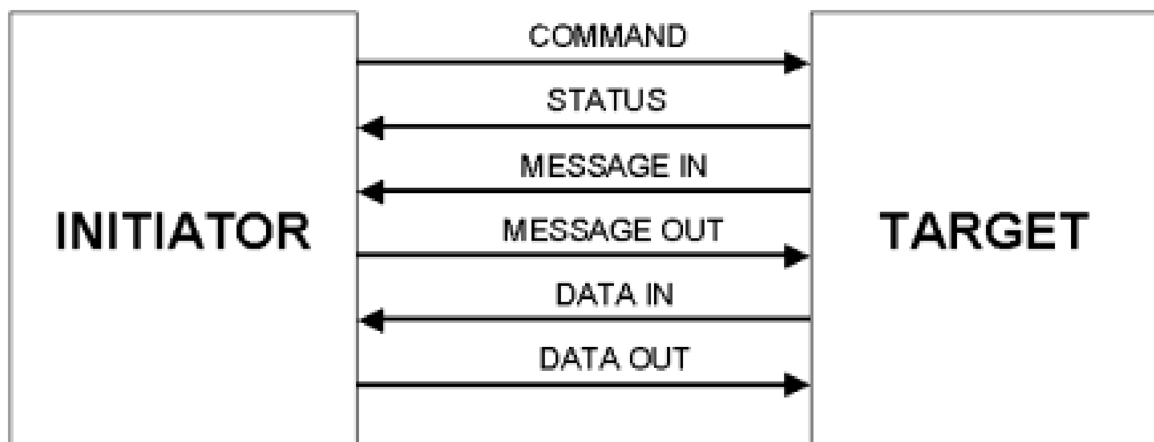


Fig 2.3 Information Transfer Phases.

The SCSI bus can be in any one bus phase at a given time. Each phase has a predetermined set of rules (or protocol) that apply when the bus changes from one phase to another. The BUS FREE, ARBITRATION, SELECTION, and RESELECTION phases form part of 'Initialization Process' that does not involve any information transfer. The COMMAND, DATA, MESSAGE, and STATUS phases are the 'Information Transfer Phases' used to transfer real information across the data bus (Fig. 2.3). The

information content of the data bus and the device (Initiator or Target) responsible for it during the different Information transfer phases is explained in Table 2.1.

Information Transfer Phase	Content of Data Bus	Device that determines information
COMMAND	CDB bytes	Initiator
DATA IN	Data in byte(s)	Target
DATA OUT	Data out byte(s)	Initiator
STATUS	Status byte	Target
MESSAGE IN	Message in byte(s)	Target
MESSAGE OUT	Message out byte(s)	Initiator

Table 2.1 Device Involvement in Information Transfer Phases.

2.1.2 Tasks

The composition of a Task includes a definition of the work to be performed by the Target (or logical unit) in the form of a command or group of linked commands. A Task can be either Tagged or Untagged. A Tagged Task includes a Tag in its Tagged Task Identifier (field present in Command Descriptor Block) that allows many uniquely identified tagged tasks to be present concurrently in a single task set. An Untagged Task does not include a Tag in its Task Identifier, which restricts the number of concurrent untagged tasks in a single task set to one per initiator.

2.1.3 Command Descriptor Block (CDB)

A command is executed when an initiator sends a Command Descriptor Block (CDB) to the target during the COMMAND phase. A CDB gives information about the I/O operation to be performed to the target. It is classified into three types based on its length: 6-byte, 10-byte, and 12-byte. A basic format of a 10-byte CDB is shown in Fig. 2.4 [1].

<table>
<tr><th></th><th>Bit 7</th><th>Bit 6</th><th>Bit 5</th><th>Bit 4</th><th>Bit 3</th><th>Bit 2</th><th>Bit 1</th><th>Bit 0</th></tr>
<tr><td>Byte 0</td><td colspan="8">Operation Code</td></tr>
<tr><td>Byte 1</td><td colspan="3">Logical Unit Number</td><td colspan="5">Reserved</td></tr>
<tr><td>Byte 2</td><td colspan="8" rowspan="4">Logical Block Address (if required)</td></tr>
<tr><td>Byte 3</td></tr>
<tr><td>Byte 4</td></tr>
<tr><td>Byte 5</td></tr>
<tr><td>Byte 6</td><td colspan="8">Reserved</td></tr>
<tr><td>Byte 7</td><td colspan="8" rowspan="2">Transfer length (if required)</td></tr>
<tr><td>Byte 8</td></tr>
<tr><td>Byte 9</td><td colspan="8">Control Byte</td></tr>
</table>

Fig 2.4 10-Byte CDB.

The fields in the CDB are explained as follows:
Operation Code: This field tells the target the length of the CDB and the operation an initiator wants to perform.
Logical Unit Number: This field specifies the Logical Unit Number of the target.
Logical Block Address: This field tells the target where the information is located on the physical medium. Logical blocks start at 0 and are contiguous to the last block location

on the device's medium. The smallest unit of measurement on a device is a block (specified by number of bytes). A typical block size for a hard disk is 512 bytes.
Transfer Length: This field tells the target the data transfer length associated with the CDB, in terms of number of blocks.
Control Byte: This field is used for command linking, and vendor-specific operations.

The following guidelines apply for the format of a CDB:

- the first byte of the CDB is always an operation code.
- the last byte of the CDB is the control byte.
- the format of the operation code and control byte are identical for every SCSI command.

2.1.4 Task Management Functions

Task Management Functions are used by Initiator to control the execution of one or more tasks. The initiator invokes a task management function by means of a procedure call having the format:

```
Service Response = Function name(Nexus between Initiator and Target)
```

The responses returned by the target for any task management function can be one of the following:

Response from Target	Meaning
FUNCTION COMPLETE	Requested function is completed by the Target
FUNCTION_REJECTED	Requested function is not supported by the Target
TARGET FAILURE OR SERVICE DELIVERY	Request is terminated due to service delivery failure or target malfunction

Table 2.2 Task Management Response Values.

Different types of task management functions that an initiator can support are as follows:
Abort Task: The target aborts the task if it exists. If the logical unit supports this function, a response of FUNCTION COMPLETE from the target indicates that the task is aborted or is not in the task set. The target guarantees that no further responses from the task are sent to the initiator. This function is required to be supported by a logical unit that supports tagged tasks and may be supported by a logical unit that does not support tagged tasks.
Abort Task Set: The target aborts all the tasks in the task set that were created by the initiator. This is equivalent to receiving a series of ABORT TASK requests. Tasks from other initiators or in other task sets shall not be aborted. This function is required to be supported by all LUNs.
Clear ACA: The initiator invokes CLEAR ACA to clear an auto contingent allegiance condition from the task set serviced by the logical unit. The function is required to be supported by a logical unit that accepts a NACA bit value of one in the CDB Control byte.

Clear Task Set: All tasks in the appropriate task set are aborted. No status is sent to the initiator for any task affected by this request. This function is required to be supported by all LUNs that support Tagged Tasks and is optional for those that do not.
Logical Unit Reset: The target supporting Logical Unit Reset function has to implement the following functions:

- Abort all the tasks in its task set(s)
- Clear Auto Contingent Allegiance or Contingent Allegiance condition, if present.
- Release all established reservations.
- Set a Unit Attention Condition.
- Initiate a logical unit reset for all dependent logical units.

This function has to be supported by all logical units.
Target Reset: The target supporting Target Reset should issue Logical Unit Resets only to the logical units it is using. There is no requirement by any target to support this function.

2.2 Network Attached Storage (NAS)

NAS is one of the mechanisms to separate storage form the main computing system. It intercepts the communication between application and storage at a fairly high level, because the host typically accesses the remote storage via a special file system that in turn utilizes a standard network protocol stack. The unit of access is a file managed by the NAS. The host file system becomes a client utilizing the network to access a remote server daemon process that in turn accesses the physical storage device.

In this model, the remote server platform has to be relatively sophisticated, usually built from a workstation or mainframe computer with a complete multiprocessing operating system, file system, network stack, and daemon process. The physical storage device itself (i.e., the disk) is physically connected to the remote server via a standard I/O channel, such as SCSI.

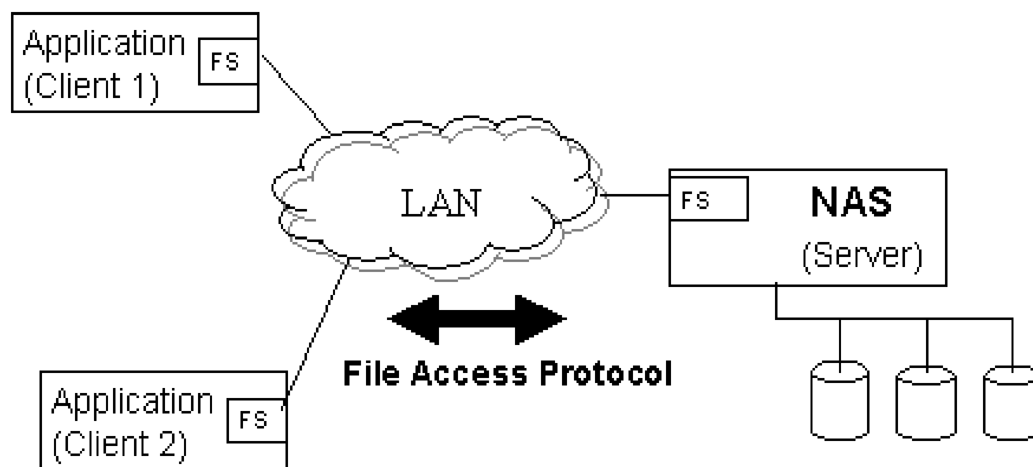


Fig 2.5 Network Attached Storage (NAS).

A NAS must deal with issues of security, consistency, integrity, etc. at the level of a complete file (Fig. 2.5). This includes the metadata, directory information about a file and all the data blocks in the file. The most common example of a NAS is Sun's Network File System (NFS) [3] which is almost universally supported by all manufacturers. In Linux, NFS is one of many file systems accessed by applications through the Virtual File System Switch (VFSS). The NFS implementation on the client host utilizes Sun's Remote Procedure Call (RPC) system to perform synchronous interactions with a daemon process on the remote server that in turn uses the file system on that platform to do the I/O to the physical disk.

Some of the advantages of NFS are that it is easily installable and is affordable. One of the disadvantages is that the storage access is limited through the Server. Direct access to storage devices is not possible. The other disadvantage is that the I/O performance is limited by the speed of a single NAS server's ability to handle I/O requests from different application clients.

2.3 Storage Area Network (SAN)

A Storage Area Network intercepts the communication between application and storage at a fairly low level, because the host views the remote storage as a device that is accessed over an I/O channel capable of long-distance transfers. The unit of access is a 'raw' storage block managed by the SAN (Fig. 2.6). In this model, the remote server system is typically much less sophisticated than for a NAS, because it only has to implement that part of the network protocol stack necessary to communicate with clients. Furthermore, the remote server system has no notion of a filesystem.

A SAN must deal with issues of security, consistency, integrity, etc. at the level of a single block, not a whole file as in a NAS. The most common example of SAN is SCSI over Fiber Channel [4]. In this system, the upper-level SCSI driver on the application

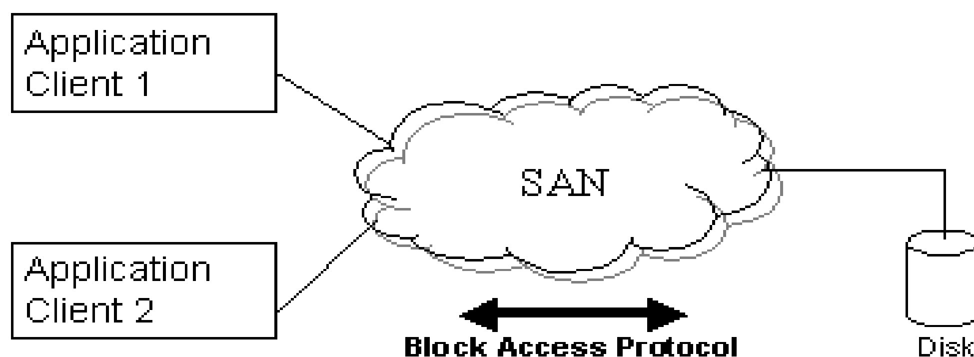


Fig 2.6 Storage Area Network (SAN).

client sees Fiber Channel as just another low-level SCSI device driver, and has no knowledge that a network is involved, since it is only the Fiber channel driver that deals with the network.

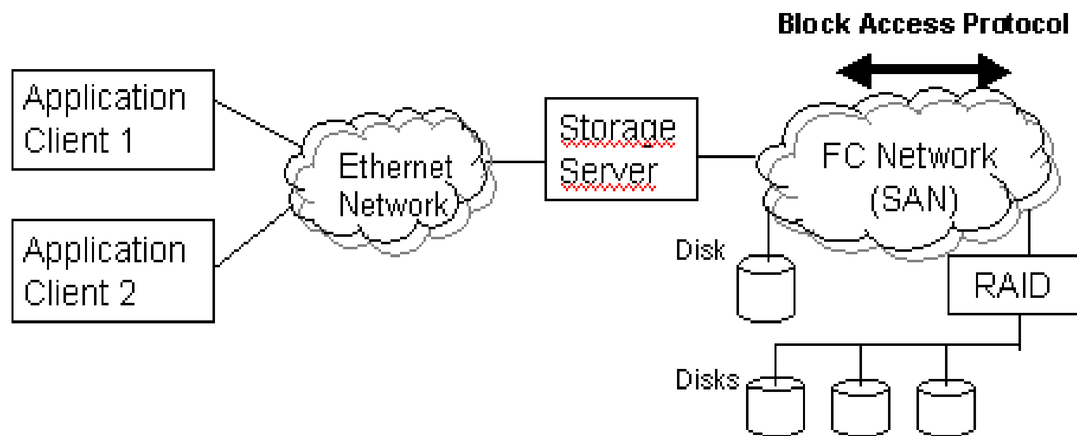


Fig 2.7 Fiber Channel Solution to Storage Area Network (SAN).

The upper levels of the normal network protocol stack are completely bypassed, since the SAN protocol is implemented entirely within the Fiber Channel driver. The remote server system is just a SCSI disk controller modified to accept SCSI commands and to transmit data over a Fiber Channel network rather than over an I/O bus. As shown in Fig. 2.7, one of the drawbacks of the Fiber Channel solution to SAN is that any Application Client, in a generic TCP/IP network, cannot have direct access to any Disk in a Fiber Channel Network.

2.4 SAN Approaches

In past two years, there have been different scenarios considered for integrating SAN into Ethernet technology to enhance the scalability of SAN. These scenarios differ in how the SAN is inserted into the normal network protocol stack, each possibility having its own advantages and disadvantages. Each scenario is elaborated upon in a separate section below.

1. SCSI on top of SAN on top of Ethernet.
2. SCSI on top of SAN on top of IP
3. SCSI on top of SAN on top of UDP
4. SCSI on top of SAN on top of TCP

2.4.1 SCSI on top of SAN on top of Ethernet

This option is similar to the SCSI over Fiber Channel approach, the difference being that the underlying link layer technologies are different. Fiber Channel has been designed to implement the prerequisites required by the SCSI Architecture Model (SAM) [5]. Thus, it provides a logical means for extending the SCSI bus and Fiber Channel provides a reliable interconnect with SCSI serving as the Upper Level Protocol. One difference in the Ethernet environment is that it introduces congestion and contention that can lead to packet loss that is not possible in SCSI over Fiber Channel. The other limitation of Gigabit Ethernet is the limited distance (maximum of 2 km on multi-mode fiber [6]) and

the bus topology that it supports. Fiber Channel, on the other-hand, allows connectivity over several kilometers using several different topologies. This requires the SAN to deal with reliability issues and to add naming conventions so that the geographical reach can be extended beyond a LAN. Some of the models have been thought over at UNH are discussed briefly:

- Ignore the problems by assuming that all hosts and storage devices are connected to a single LAN, and that there is no packet loss due to contention on the Ethernet. Chris Loveland has implemented SCSI over Gigabit Ethernet using the Alteon AceNIC cards. The implementation included modification of the existing driver by adding functions to it that made it look like a low level SCSI driver in Linux. The packet loss is ignored by the driver, so that if an error occurs the low-level SCSI operation times out, forcing the high-level SCSI driver to reattempt the operation. 6-byte Ethernet MAC addresses were used to denote the client and servers which were connected on the same LAN.
- Define a SAN protocol that provides inorder, reliable delivery over unreliable Ethernet. Limit the connectivity to a single LAN. This approach taken by Barry Reinhold in his specification of SANTRAN [7]. He suggests defining a protocol that included an acknowledgement and retransmission on timeout scheme, with fail-over to alternate adapters as a last resort. It was designed with the idea that most of SANTRAN would be implemented in firmware on the NIC. The host system driver would look like a low-level SCSI driver, as in Chris's system. The driver would utilize a SANTRAN packet format that encapsulates SCSI commands along with its own control and status information. Although SANTRAN provides its own addressing, it seems to expect that this address can be mapped directly onto a MAC level address, thereby limiting geographic range to one LAN.
- Define a SAN protocol that provides in-order, reliable delivery over any underlying networking technology, not necessarily limited to a single LAN.

2.4.2 SCSI on top of SAN on top of IP

This approach obviates the problem associated with using just SAN on top of Ethernet of limiting the geographic range to one LAN. The limitation in IP is that it doesn't provide in-order delivery of packets, which has to be handled by the SAN. One problem in this approach is to handle the intermediate copying. The IP specifications require the IP payload to be contiguous on the wire, whereas to achieve zero-copying, the IP payload would almost certainly have to be stored separately from the IP header. One way to achieve zero-copying is by pushing part of the IP implementation on to the NIC. Nishan's Storage over IP(SOIP) has adopted this approach though no details have been provided.

2.4.3 SCSI on top of SAN on top of UDP

This approach uses UDP as the underlying transport layer protocol. The UDP approach allows for error detection (but not correction) through use of checksumming, and port multiplexing. Checksumming is a viable option for a SAN environment if it is done in the hardware. Port multiplexing is not an issue in the SAN environment. The unreliability of IP is not solved by UDP as it is only a thin wrapper around IP. SUN's RPC subsystem [8]

utilizes UDP as their network transport protocol to support the SUN's Network filesystem (NFS). RPC deals with the unreliability of UDP by detecting RPC timeouts and then retrying RPC operations.

2.4.4 SCSI on top of SAN on top of TCP

This is the one of the most promising of the approaches considered. TCP provides the reliable in-order delivery over worldwide networks, unlike IP and UDP, which are connectionless oriented. This reliability is desirable for SAN environment though some issues which have to be considered:

- How to avoid intermediate copying? This involves modifications to TCP and all layers below TCP in the protocol stack. The intermediate copying affects the performance which is very critical to do data transfer in a SAN.
- How much of the complete TCP/IP protocol can or should be implemented on the NIC? The problem is more critical for TCP implementation than for IP or UDP, because TCP maintains a huge amount of state information about each connection. This is because TCP is a connection oriented protocol.

Implementing SCSI over TCP for IP storage leverages existing network hardware, software, and technical know-how. SCSI over TCP will enable the use of Gigabit Ethernet, a proven industry standard, as an infrastructure for storage-area networks (SAN). TCP/IP and Ethernet are dominant technologies in networking, and we see this trend continuing with SANs. This is the approach used for this thesis that involves implementation and evaluation of Session Protocols to support SCSI over TCP/IP network.

2.5 SAN Approach for this thesis

Block level access in a TCP/IP/Gigabit Ethernet network to a target disk would require TCP/IP implementation in the target hardware in addition to implementation of Session

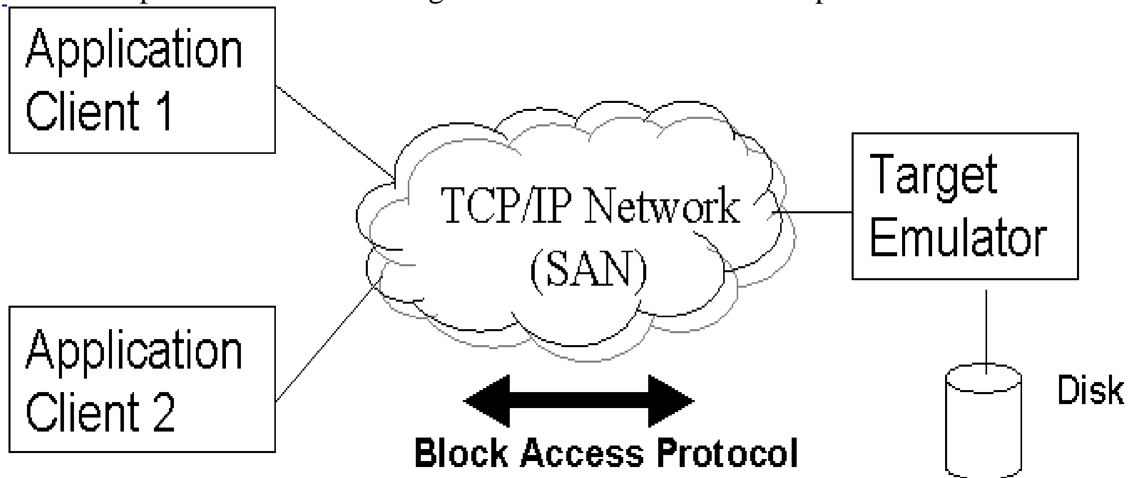
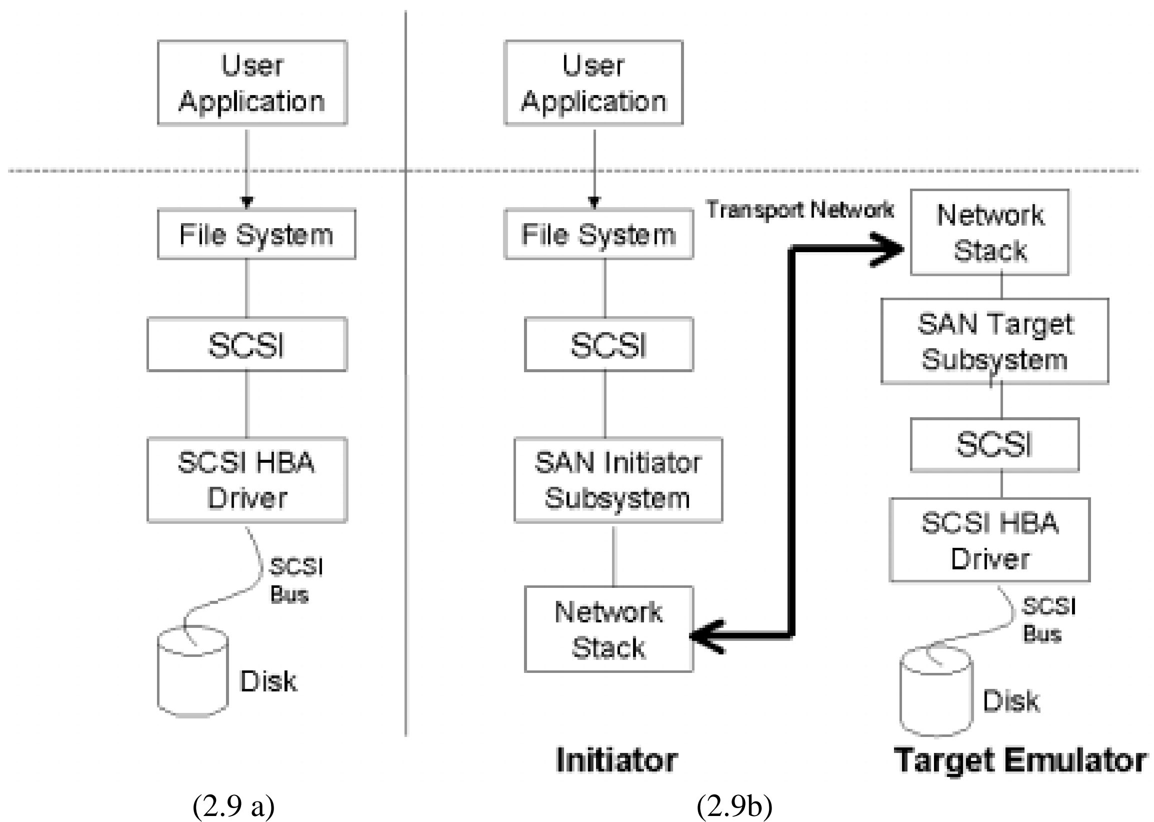


Fig. 2.8 Our Approach to Storage Area Network (SAN).

Protocols. Hardware implementation of Network Protocols like TCP/IP is an intricate process by itself. An alternate approach, followed in this thesis project, is to emulate a target in software on a Remote system as shown in Fig. 2.8.

Efforts to increase scalability of SCSI have increased the number of components involved in the Operating System (Fig. 2.9). The transition from classic SCSI approach to our approach is discussed as follows:

- The classic SCSI approach is to have a SCSI Host Bus Adapter (HBA) accessing a SCSI disk (target) connected by a SCSI Bus(Fig. 2.9a). The READ/WRITE requests generated by the user go through the Filesystem to resolve into SCSI Commands, Messages, and Responses. The SCSI requests are passed to the HBA that accesses the SCSI disk (target).
- In our approach (Fig. 2.9b), the extra components added are as follows:
    - Network Stack to support TCP/IP.
    - SAN subsystem to support the Session Protocols.
    - Transport Network to communicate between the Initiator and the Target Emulator



(2.9 a) (2.9b)

Fig 2.9 Components involved to support SCSI.

The SEP and iSCSI low-level drivers (explained in Chapter 5) are implemented to support the latest specification of the two protocols in this thesis work. The existing target emulator, developed by Ashish Palekar [9], is extended to support additional SEP

and iSCSI protocol features. Also, the target emulator code is modified to support the latest version of the iSCSI draft. The target emulator is discussed in detail in Appendix A.

Chapter 3

Session Layer Protocols

In this chapter, we discuss the two SAN Session Layer Protocols used over TCP/IP. SCSI Encapsulation Protocol (SEP) and Internet SCSI (iSCSI) are the two proposed Internet drafts chosen for this thesis work. The two protocols have been discussed in detail in the next two sub-sections.

3.1 SCSI Encapsulation Protocol (SEP)

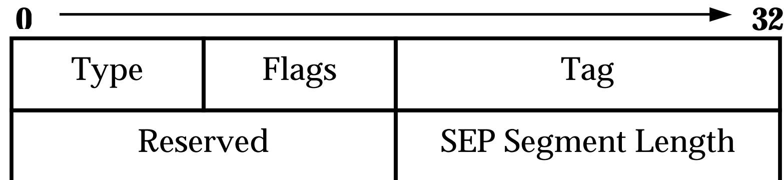
Adaptec Inc. has developed the SCSI Encapsulation Protocol (SEP) as their answer to operate SCSI on top of TCP [10]. The SEP protocol assumes an underlying reliable transport protocol such as TCP/IP. It is a session layer protocol that encapsulates SCSI commands, status, and data for transmission over a transport layer protocol. The SEP architecture focuses on being able to provide a cost and performance competitive solution in those spaces. With typical server applications, CPU utilization by the host is a concern. The CPU utilization is relatively low in traditional storage adapters. To achieve similar CPU utilization metrics with host adapters using SEP, the SEP protocol envisages the processing of the entire TCP/IP protocol on the host adapter itself rather than burning CPU cycles.

Connection establishment, authentication, security, etc. are the features to be handled by TCP and are beyond the scope of SEP. SEP requires that each SCSI LUN be represented by a unique TCP connection in order to enforce the host's ordering between commands to that LUN. So, there is no LUN information included in the SEP header. This obviates keeping additional state information that is required if multiple LUNs are multiplexed on a single TCP connection. In SEP, multiple connections may be used when accessing different LUNs on the same target.

Each SEP message starts with an 8-byte fixed format header followed by SCSI command, status, message or data information. The SEP header contains 1 or 2 byte fields that are aligned on corresponding byte boundaries for easy accessibility, as depicted in Fig. 3.1. The payload can go to a minimum of 1 byte and maximum of 65536 bytes. The data is always padded by trailing zeros to a multiple of 4 bytes to maintain 4 byte boundary alignment. The CRC is optional and is added to the end of the segment, after any required pad bytes. The segment length indicated by the SEP header does not include the pad bytes or the CRC. The Flags field in the SEP header is used to minimize the transmission of status messages to the host.

The steps involved for a host to utilize SEP would be as follows:

- Open a TCP connection to the desired target. The host has to know the IP address and port number of the desired target to make a TCP connection. The mechanism to find the IP address and port number is not specified. It is assumed that the port number will be a "well known port number" for SEP target servers.



Type:	
0x01	Simple Tagged Command
0x02	Head of Queue Tagged Command
0x03	Ordered Tagged Command
0x04	SCSI Data
0x05	SCSI Status
0x06	SCSI Message
0x08	SCSI Data Request
0x09	SCSI Set Data Pointer
0x10	Connect and Negotiate
0x11	Negotiation Response
0x12	Third Party Open
0x13	Third Party Open Response
0x14	Third Party Close
0x15	Third Party Close Response

Fig. 3.1 SEP Header.

- The next step is to establish a connection for a LUN between the Initiator and the target by sending a 'Connect and Negotiate' message (type 0x10). This is the first message sent over a new connection, as it contains the LUN that identifies the SCSI device for all future operations on this TCP connection. This command also contains a number of flags to define options, and two unsigned 16-bit flow control values. The first is MWPS, the maximum number of data bytes sent by the host on a write operation without receiving a "Get SCSI Data" command from the target; and the second MRPR, the maximum number of data bytes sent by the target on a READ operation without receiving a "Get SCSI Data" command from the host.

- The initiator receives a 'Negotiation Response' message (type 0x11) sent by the target. This is the first message sent by the target and indicates that a connection has been established. It contains the target's flag and flow control values as determined by the target from its capabilities and those requested by the host in the previous "Connect and Negotiate" message. These values are valid for all the subsequent communication on this connection.

- The host can then transmit any of the appropriate SCSI commands of type 0x01, 0x02, and 0x03 to send SCSI Command Descriptor Blocks to the target, commands of type
  0x04 to send data, commands of type 0x06 to send a message, commands of type 0x09 to reset the target's current data pointer, commands of type 0x12 to open a third party session with a specified IP address and LUN, and commands of type 0x14 to close a previously opened third party session. The target will respond to these commands with commands of type 0x04 to send data, commands of type 0x05 to send status messages, commands of type 0x13 to respond to a request to open a third party session, and commands of type 0x15 to respond to a request to close a third party session.

- There is no specified way to close a connection from either side. The only presumable way is to terminate the TCP connection. The EOF on READ and EPIPE error on write will be detected as a terminated connection on the other side.

One of the issues that is not considered by SEP is error recovery in case of a dropped connection. This leads to a couple of issues regarding the file-system state on the target side, which is unknown, when a connection is lost.

SEP is the approach used by Adaptec in their Ether-Storage Technology. Their web site states "This new technology enables block-based storage traffic to be efficiently and reliably transferred over existing IP and Ethernet-based networks, and is the result of over two years of research and development at Adaptec on the future storage fabric architectures" [11].

3.2 Internet SCSI (iSCSI)

The Internet SCSI (iSCSI) is another example that is a combined effort of a group of companies including IBM, Cisco Systems, Hewlett-Packard, SANGate, Adaptec, and Quantum. It is currently being developed under the aegis of the Internet Engineering Task Force [12]. The protocol is on its course to become an RFC and is currently an Internet Draft at version 7. The iSCSI protocol, like SEP, is a session layer protocol that assumes the presence of a reliable connection oriented transport layer protocol like TCP. It is a much more elaborate protocol than SEP, and includes considerations for naming, TCP connection establishment, security, and authentication.

Communication between an Initiator and a Target occurs over one or more TCP connections. The TCP connections are used for SCSI commands, Task management commands, data, protocol parameters, and control messages within an iSCSI Protocol Data Unit (iSCSI PDU). All related TCP connections between the same initiator and the target are considered to belong to a session (referenced by a session ID).

3.2.1 Steps involved in iSCSI Protocol to support I/O operation

The steps involved for a host to utilize iSCSI (draft 7) would be as follows:

- Open a TCP connection to the desired target. The host has to know the IP address and port number of the desired target to make the TCP connection. There have been mechanisms described in another Internet Draft [13] for naming and discovery of targets. For this thesis, it is assumed that the IP address and the port number of the target is known to the initiator.

- The next phase is the **Login phase**, where an iSCSI session is established between initiator and target. It sets the iSCSI protocol parameters, security parameters, and authenticates initiator and target to each other.

  The login phase starts with a login request via a "login command" (opcode 0x03) from the initiator to the target (Fig. 3.3). This is the first message sent over a new connection. The login request includes the protocol version supported by the initiator (currently 01), session and connection IDs, security parameters (if security is requested), and protocol parameters.

  The initiator receives a "Login Response" (opcode 0x23) message from the target. This is the first message sent from the target on a new connection after receiving the "Login Command" from the host. The target can answer the initiator in the following ways:
    - Login response with "login reject" to reject the session establishment request.
    - Login response with "login accept" with the session ID, iSCSI parameters, and Final bit 1. In this case, the target does not support any security, authentication mechanism or parameter negotiation and starts the session immediately.
    - Login response with Final bit 0 indicating the start of the authentication/ negotiation sequence. The negotiation is done with the help of "Text" exchange (opcode 0x04 from initiator to target, opcode 0x24 for the response back) which allows for the exchange of information in the form of key=value pairs.

- The host can now send any of the other types appropriate for a SCSI initiator and is said to be in the **Full Feature** phase. There are 8 opcode types that can be sent from the host to the initiator, and 10 opcode types that can be sent back to the host by the initiator.
    - The primary pair of opcode types for accomplishing SCSI data transfers is the "SCSI Command" (opcode 0x01) which encapsulates a SCSI command block from the host to target, and the "SCSI Response" (opcode 0x21) which is used to report the status of a SCSI command from target back to host.
    - A pair of opcodes for "Task Management" (opcode 0x02 from initiator to target, opcode 0x22 for the response back) that allows the initiator to explicitly control the execution of one or more tasks in the target.
    - A pair for "Text" exchange (opcode 0x04 from initiator to target, opcode 0x24 for the response back) which allows for the exchange of information in the form of key=value pairs.

- A pair for "NOP" exchange (opcode 0x00 from initiator to target, opcode 0x20 for the response back) which is used to verify that a control connection is still active.
- "SCSI Data" (opcode 0x05 from initiator to target for WRITE operation, opcode 0x25 from target to initiator for READ operation) which is used to transfer SCSI data between initiator and target.

- There are 3 unpaired "unsolicited responses" which the target can send to the initiator.
  - "Ready to Transfer" (opcode 0x31) which is sent by the target to the host when the target is ready to receive data from the host.
  - "Asynchronous Event" (opcode 0x32) which is sent by the target to the host to indicate special conditions.
  - "Reject" (opcode 0x3f) which is sent by the target upon receiving from the host a message with an opcode that it doesn't recognize.

- There is 1 unpaired "unsolicited responses" which the initiator target can send to the initiator.
  - "SNACK" (opcode 0x10) which is sent by the initiator to request retransmission of numbered- responses, data or R2T PDUs from the target.

- SCSI data is sent over the same TCP connection as the one used to transfer the SCSI command from the initiator.
  - To read data from a target, the host sends a "SCSI command" message with an encapsulated SCSI CDB that describes the data the host wants to read. The target then sends the requested data (opcode 0x25) followed by the status to the initiator on the same TCP connection that was used to deliver the SCSI command.
  - To write data to a target, the host sends a "SCSI command" message on the control connection, receives back the associated "SCSI command response", and then waits to receive the "Ready to Transfer" (R2T) response from the target. This message contains parameters describing that portion of the total amount of data that should now be transferred. The data (opcode 0x05) is then transferred from the initiator to the target on the same TCP connection that was used to deliver the SCSI command.

    There is an exception to this sequence of exchanges for doing WRITE operation. The initiator can send "Immediate Data" which follows the "SCSI Command" as Command Data (Fig. 3.2) going to the target. Also, the initiator can send "Unsolicited Data" as separate Payload Data Units (PDUs) following the SCSI Command without waiting for an R2T from the target. The maximum size of Unsolicited Data and Immediate Data, that can be transferred, is negotiated during Login phase.

- There are two ways that a session can be terminated between the initiator and the target devices:

- Explicit mechanism: The initiator sends a Logout Command when it wants to end the session with the target. The target acknowledges the received Login Command by sending a Login Response. After the exchange of this iSCSI PDUs, the session is terminated.
- Implicit mechanism: The implicit way is to terminate the TCP connection without any warnings. The EOF on READ and EPIPE error on write will be detected as a terminated connection on the other side. This is not recommended but is an option that an initiator or a target might choose to close a session.

3.2.2 iSCSI PDU format

An iSCSI PDU has one or more header segments and, optionally, a data segment. After the entire header segment group there may be a header digest. The first segment, and in many cases the only segment, (Basic Header Segment or BHS) is a fixed-length 48-byte header segment. It may be followed by Additional Header Segment (AHS). So, if we have only a BHS (no data or digests) the size of the iSCSI PDU is 48 bytes. All the PDU segments and digests are padded to an integer number of 4 byte words. The padding bytes should be 0. The different components are defined as follows:

- Basic Header Segment (BHS): The Basic Header Segment is 48 bytes long. The Opcode, TotalAHSLength, and DataSegmentLength fields appear in all iSCSI PDUs. In addition, the Initiator Task Tag, Logical Unit Number, and Flag fields, when used always appear in the same location in the header (Fig. 3.3).
- Header Digest and Data Digest: Optional header and data digests protect the integrity and authenticity of header and data, respectively. The digests, if present, are located, respectively, after the header and PDU-specific data and include the padding bytes. The digest types are negotiated during the login phase.
- Additional Header Segment(AHS): The AHS starts with 4 byte TLV (Type-Length-Value) information. It is followed by the actual AHS, the length of which is specified by Length field in TLV (Fig. 3.4).

3.2.3 Sequence Numbering

The iSCSI protocol supports command, status, and data numbering schemes which are needed during flow control, error handling, and error recovery. Command numbering is session wide and is used for ordered command delivery over multiple connections. It can also be used as a mechanism for command flow control over a session. Status numbering is per connection and is used to enable recovery in case of connection failure. Data numbering is per command and is meant to reduce the amount of memory needed by a target sending unrecoverable data for command retry. The three numbering schemes are

Byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	Opcode								Opcode-specific fields																							
4	TotalAHSLength								Length of the data following the 48 byte header																							
8	Logical Unit Number (LUN) or Opcode-specfic fields																															
16	Initiator Task Tag or Opcode Specific fields																															
20 – 48	Opcode-specific fields																															

Fig. 3.2 iSCSI Basic Header Segment (iSCSI draft ver 7).

Opcode:

0x00: NOP-Out Message
0x01: SCSI Command
0x02: SCSI Task Management Command
0x03: Login Command
0x04: Text Command
0x05: SCSI Data-out(for WRITEs)
0x06: Logout Command
0x10: SNACK Request

Opcode:

0x20: NOP-In message
0x21: SCSI Response
0x22: SCSI Task Management Response
0x23: Login Response
0x24: Text Response
0x25: SCSI Data In(for READs)
0x26: Logout Response
0x31: Ready To Transfer
0x32: Asynchronous Event
0x3f: Reject

Byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	X	I	Opcode (0x01)						F	R	W	0	0	ATTR			Reserved								CRN or Reserved							
4	TotalAHSLength								DataSegmentLength																							
8	Logical Unit Number (LUN)																															
16	Initiator Task Tag																															
20	Expected Data Transfer Length																															
24	CmdSN																															
28	ExpStatSN or ExpDataSN																															
32	SCSI Command Descriptor Block (CDB)																															
48+	Header Digest (if any)																															
	Command Data (if any)																															
	Data Digest (if any)																															

Fig. 3.3 iSCSI Basic Header Segment(BHS) for SCSI Command (iSCSI draft ver 7).

Byte	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
0	AHSType								AHSLength																AHS-Specific							
4+	AHS-Specific																															

Fig. 3.4 iSCSI Additional Header Segment(AHS) for SCSI Command (iSCSI draftr ver 7).

as follows: CmdSN (Command Sequence Number), StatSN (Status Sequence Number), and DataSN (Data Sequence Number).

3.2.4 Error Recovery

The iSCSI protocol, unlike SEP, deals with error recovery in case of protocol errors. This section of the protocol is undergoing a lot of change currently. It is assumed that iSCSI in conjunction with SCSI is able to keep enough information to be able to rebuild the command Protocol Data Unit (PDU), and that outgoing data is available in host memory for retransmission while the command is outstanding. It is also assumed that at a target, iSCSI and specialized TCP implementations are able to recover unacknowledged data from a closing connection or, alternatively, the target has the means to re-read the data from a device server. The transmission of, or absence thereof, status and sense information is used by the initiator to decide which commands have been executed or not.

iSCSI recovery for communication errors involves the following steps:

- abort the offending TCP connection(s) (target and initiator) and recover at the target all unacknowledged read data.
- create one or more new TCP connections (within the same iSCSI session) and associate all the outstanding commands from the failed connection to the new connection(s) created at the initiator and the target.
- the initiator will reissue all outstanding commands with their original Initiator Task Tag and their original Command Sequence Number (CmdSN). The latter is needed only when the commands were not acknowledged. If acknowledged, a new CmdSN needs to be used. The Opcode will be set to indicate that the command is a retry.
- The target then performs the operation either by recovering the old data (if possible) or re-doing the operation.

Other issues concerning configurable options, security, error handling, and recovery are also addressed in the latest iSCSI draft.

3.2.5 Important Operational Parameters

There are operational parameters defined in the iSCSI protocol to support I/O operation between the initiator and the target. The operational parameters are negotiated with the help of 'Text Commands' (going from initiator to target) and 'Text Responses' (going from target to initiator), exchanged during the Login or Full Feature Phase. During operational parameter negotiation, the key in 'key=value' pair (which forms the payload of 'Text exchanges') stands for the operational parameter. Some of the important operational parameter keys defined in the protocol and implemented in the iSCSI low-level driver (explained in Chapter 5) are explained as follows:
<u>InitialR2T:</u> The InitialR2T key is used to specify if the target can/cannot support of Unsolicited Data in Separate Data PDUs. A 'key=value' pair of 'InititialR2T=no' means that the target can support Unsolicited Data in separate Data PDUs. A value of 'yes' means that the target cannot support Unsolicited Data in separate Data PDUs.

Immediate Data: Initiator and Target negotiate support for Immediate Data. If the 'ImmediateData' key is set to 'yes', then the Initiator can send Data as payload in the SCSI "SCSI Command" PDU. Key value of 'no' means that the Initiator cannot send Data as payload in the iSCSI "SCSI Command" PDU.
DataPDULength: Initiator and target negotiate the maximum data payload supported by the iSCSI "SCSI command" or "SCSI Data" PDUs through this key. The value for this key is in units of 512 bytes.
FirstBurstSize: Initiator and target negotiate the maximum length supported for (Immediate Data + Unsolicited Data in Separate Data PDUs) in units of 512 bytes.

The Text parameter negotiation for Login Phase has been designed and implemented by Narendran Ganapathy (Graduate Student, CS Dept., UNH) as part of his thesis.

Chapter 4

Fast Kernel Tracing

Fast Kernel tracing (FKT) is a method for obtaining a precise, time-stamped trace of the dynamic activities of kernel-level code[13]. The basic motivation being that it is extremely difficult to know exactly what an operating system is doing, especially in a networked environment.

The method consists of placing special software "probe" macros at various locations in the kernel source code. Placement of probes is controlled by the kernel programmer, as is the information recorded by the probe. Typically, one probe is placed at the entry to, and another at the exit from, every kernel function of interest to the programmer. The entry probe can record the parameters passed into the function, and the exit probe can record the function's return value also. If any changes are made regarding any probe (added, deleted or relocated), the kernel must be rebuilt and rebooted.

Probes record data and store it into a large buffer in the kernel. The buffer size available is limited and can get filled up before tracing is finished. Currently, probing is suspended whenever the buffer gets filled. However, traces that cover many tens of seconds of intense operating system activity can be obtained by the current version of FKT. After recording has been finished, a user-level program can obtain a copy of the probe data from the kernel buffer and write it to a file for later off-line analysis. There are permanently assigned probes that record every IRQ, exception, system call, and task switch handled by the kernel during the sampling interval. This data, together with that from the programmer assigned codes, can be presented in the form of the actual traces themselves, with time spent in each step of the trace. However, it is easy to obtain more detailed data, such as the minimum, maximum, and average amounts of time spent in each kernel function, the nesting characteristics of the functions, etc.

4.1 FKT Setup

A software probe is a call to a special routine that reads the processor's cycle clock and stores it into a buffer. The pid of the process, which is an identification code passed to the probe, and any additional parameters passed to the probe are recorded in addition to the processor's cycle clock. The format of a probe entry in the buffer is:

1 low-order 32-bits of processor cycle clock
2 pid of current process in low 16-bits, cpu number in high 16-bits
3 code passed to probe
4 Parameter 1 if present

5 Parameter 2 if present
….
3+N Parameter N if present

Each probe stores upto 3+N 32-bit unsigned integers into the buffer, where N is the number of parameters passed in the probe.

A "code" is a unique number assigned by the kernel programmer that is used to identify a particular probe. There are two types of codes:

- Unshifted codes: These codes are never accompanied by parameters. Therefore, the probe entry for these codes is always exactly the first 3 integers shown above. These codes are permanently assigned to system calls, exceptions, and IRQs.
- Shifted codes: These codes may be accompanied by parameters. The kernel programmer chooses the assignment of shifted code values.

The user-level programs actually control the kernel probing via system calls. The kernel's probe recording buffer is allocated dynamically at the start of a measurement session. A set of probes is also enabled at that time, so that whenever control subsequently flows through one of the probe points in this set, the corresponding probe record will be stored in the probe buffer, as explained earlier. The user can selectively change the set of enabled probe points in order to limit tracing to only certain segments of a longer test. The user can also include additional probe records with appropriate parameters. Such probes help the user to correlate kernel activities with actions in the user-level code. After probing has been stopped, the user-level control program obtains a copy of the data from the kernel's recording buffer and writes it to a file for later offline analysis. The measurement session ends with the release of all the pages allocated to the kernel recording buffer.

4.2 Application Programmer Interface

Any recording session is started when the `fkt_setup()` system call is invoked by a user-level program with a 32-bit keymask passed as a parameter. This allocates the kernel's recording buffer and enables those probes already compiled into the kernel whose keymasks contain a 1 that matches a 1 in the corresponding bit in the parameter. Whenever control flows through them these probes will write a record into that buffer.

After the probe buffer has been set up, the user can invoke the `fkt_keychange()` system call to enable and/or disable sets of probes with the appropriate keymasks. This system call also inserts into the recording buffer a probe record having the code FKT_KEYCHANGE_CODE and the new keymask and current system clock as parameters. At any time a user program can insert its own probe records into the kernel's recording buffer by making `fkt_probe0(), fkt_probe1(), or fkt_probe2()` system calls with the appropriate number of parameters. After a recording session has been stopped for the final time, the user invokes the `fkt_getcopy()` system call to copy the kernel's probe buffer into a file in user space for later analysis. After data from a measurement session has been copied out of the kernel buffer and written to a file, the

user can start another session by invoking the `fkt_reset()` system call. `fkt_reset()` takes the same parameter and has the same effect as `fkt_setup()`, except that it reutilizes an existing kernel recording buffer from a prior `fkt_setup()` rather than allocating a new one.

After the probing is done, a user program makes a final system call `sys_endup()`, which will release all the pages allocated to the kernel recording buffer.

4.3 Kernel Programmer Interface

A number of macros have been defined that can be utilized within kernel code to cause probe data to be recorded when control flows through them. Their names are:

FKT_PROBEx(KEYMASK, CODE, …)

where x is a digit in the range 0…5 that indicates the number of parameters whose values replace the '…' in the macro. This gives a programmer the flexibility of passing different numbers of parameters to the probe depending on the need. The above macros are designed for use with shifted codes only.

For unshifted codes, which are never accompanied by parameters, only one macro is necessary:

FKT_PROBE_NOSHIFT(KEYMASK, CODE)

4.4 Recording and Printing

As explained earlier, any program can be instrumented to set up and start probing, run a test, stop probing, and copy the data to a file. A general-purpose tool called `fkt_record` has been developed that will probe any "target" program. The `fkt_record` program starts by calling `fkt_setup()` to set up the probing. It then calls `fork()` to create a child process and that child process calls `exec()` to start running the target program. When the target program completes, `fkt_record` disables all probes, copies the kernel's probe buffer to a file and then calls `fkt_endup()` to deallocate the buffer. The output file is written in binary. `fkt_record` takes two optional command-line switches:

- -k *bitmask*
  The *bitmask* value is a number and determines which probes will be enabled while recording the trace. If this switch is omitted, a value of –1 is used as the initial key set parameter, which will enable all probes.
- -f *outputfile*
  The *outputfile* value, which is a string, denotes the name of the file to which kernel buffer is written. The buffer is written to a file called 'trace_file' if the switch is not given.

A second tool called `fkt_print` has been developed to analyze and print the file recorded by `fkt_record`. `fkt_print` takes the following optional command-line switches to modify the information being printed depending upon the specified switch.

- -f *inputfile*
  The *inputfile* value, which is a string, denotes the kernel buffer file. The buffer is read from a file called 'trace_file' if the switch is not given.
- -p
  If this switch is present, printing is forced for all the items found in the buffer. If it is not given, printing starts with the first item having the pid of the child process spawned by `fkt_record`.
- -c
  If this switch is present, the time printed for each line of the trace is the cumulative time since the time of the first probe record. If it is not given, the time printed on each line is the time since the previous line printed, which is the time elapsed between those two probe points.
- -d
  If this switch is present, debug printout to show stack nesting, etc. is written along with trace output. If not given, no debugging output is printed.
- -a
  If this switch is present, statistics are accumulated for all trace items. If it is not given, statistics are accumulated only between the first and second probe records for the `times()` system call with the pid of the child process spawned by `fkt_record`. This means that the child process should call `times()`to time its own exection.

Chapter 5

SCSI SubSystem in Linux

In this chapter, we first discuss the layered architecture for SCSI support in the Linux Operating System. We then discuss the important data structures involved, followed by a discussion of the relationship between the different structures. Finally, the discovery process for the targets is explained.

5.1 SCSI Layers in Linux

The whole SCSI subsytem in the Linux kernel is divided into three levels: upper, middle, low-level drivers (Fig. 5.1). We discuss the functionalities of each layer as follows:

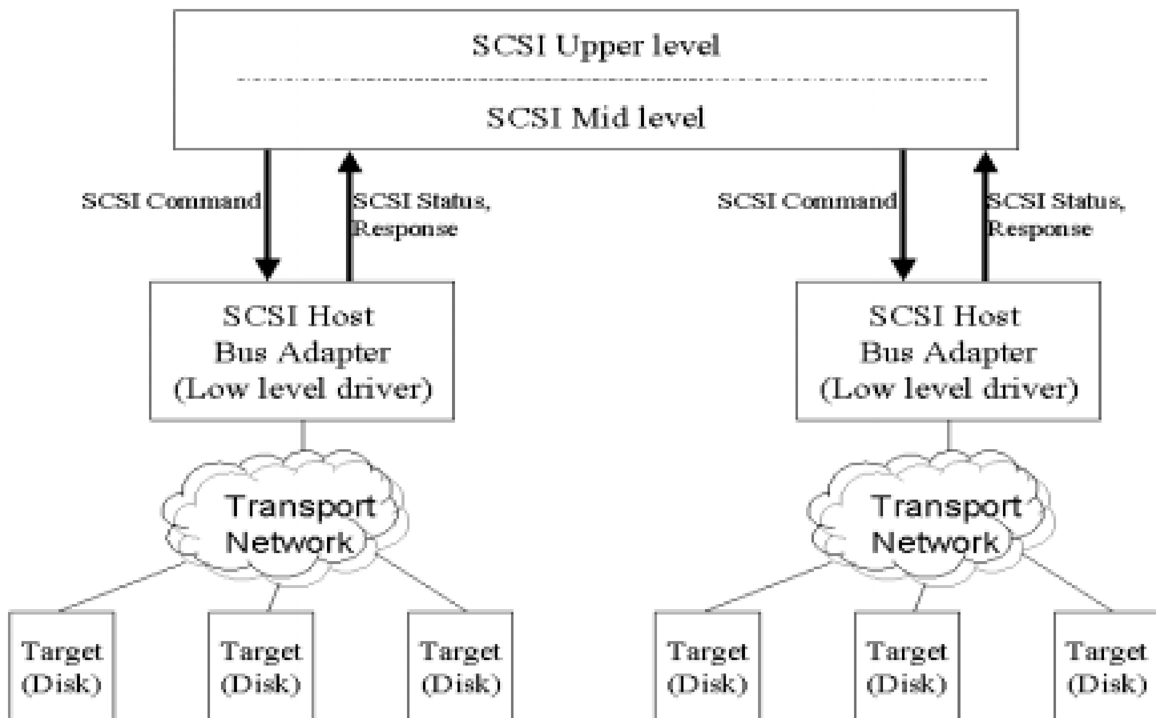


Fig. 5.1 SCSI SubSystem in Linux.

5.1.1 Upper level

The upper level of the SCSI subsystem has the task of taking requests that come from outside of the SCSI subsystem, and turning them into actual SCSI requests. The requests, in turn, are passed down to the middle level. Once the command processing is complete,

the upper level receives the status from the middle level, and in turn the upper level will notify the external layer of the status.

Requests originate from 3 different sources. For block devices, requests originate from the ll_rw_blk layer. For character devices, the requests effectively originate directly from the filesystem code as users attempt to operate on the devices. Finally, the third source of requests is via the ioctl system call - to a large degree this is similar to how character device requests are originated, however ioctls can also be issued to block devices. The basic tasks of the upper layer are:

- Translate incoming requests into SCSI commands
- Create scatter-gather lists for the request
- Track usage counts as file descriptors are opened and closed
- Maintain externally visible arrays for device size and block size
- Finally, there is some amount of common glue that is required to make it possible for an upper level driver to be a module

5.1.2 Middle level

The SCSI middle level (SML) provides various functions that can be described as the following subcomponents: the module-related glue itself (only used when SCSI is used as a module), Proc filesystem support, Bus scan support, Other miscellaneous initializations (both boot time and module related), Error handling, Queuing of commands, Bottom half handler, and Utility functions.

5.1.3 Low-level drivers

The low level drivers (LLD) actually transfer commands, data, status, messages etc. between the initiator and the target. Conventional Fiber Channel, SEP or iSCSI can be used as drivers to carry out SCSI requests from initiator from target.

5.2 Important SCSI Data Structures

There are three important data structures involved in the implementation of the LLDs: `Scsi_Host_Template`, `Scsi_Host`, and `Scsi_Cmnd`. We explain the three data structures in different sub-sections and then discuss the relationship among them and their involvement in communication between the SML and LLD.

5.2.1 Scsi_Host_Template struct

The `Scsi_Host_Template` struct is the most important data structure (Fig. 5.2), as it serves as a direct interface between the SML and the LLD (Fig. 5.3).

```
typedef struct SHT
{
        struct SHT * next;
        struct module * module;
        struct proc_dir_entry *proc_dir;
        int (*proc_info)(char *, char **, off_t, int, int, int);
        const char *name;
        int (* detect)(struct SHT *);
        int (*revoke)(Scsi_Device *);
        int (*release)(struct Scsi_Host *);
        const char *(* info) (struct Scsi_Host *);
        int (*ioctl)(Scsi_Device *dev, int cmd, void *arg);
        int (* command)(Scsi_Cmnd *);
        int (* queuecommand)(Scsi_Cmnd *, void (*done)(Scsi_Cmnd *));
        int (*eh_abort_handler)(Scsi_Cmnd *);
        int (*eh_device_reset_handler)(Scsi_Cmnd *);
        int (*eh_bus_reset_handler)(Scsi_Cmnd *);
        int (*eh_host_reset_handler)(Scsi_Cmnd *);
        int (* abort)(Scsi_Cmnd *);
        int (* reset)(Scsi_Cmnd *, unsigned int);
        int (* bios_param)(Disk *, kdev_t, int []);
        void (*select_queue_depths) struct Scsi_Host *, Scsi_Device *);
        int can_queue;
        int this_id;
        short unsigned int sg_tablesize;
        short cmd_per_lun;
        unsigned char present;
        unsigned unchecked_isa_dma:1;
        unsigned use_clustering:1;
        unsigned use_new_eh_code:1;
        char *proc_name;
} Scsi_Host_Template;
```

Fig. 5.2 Scsi_Host_Template struct Definition.

The important fields for LLD implementation are discussed as follows:

`int (*proc_info)(char *, char **, off_t, int, int, int)`
Can be used to export driver statistics and other information to the world outside the kernel (i.e., userspace) through the `/proc` interface and it also provides an interface to feed information to the host bus adapter (HBA).
`int (* detect)(struct SHT *)`
The detect function returns non-zero on detection of a HBA, indicating the number of HBAs of this particular type were found. It should also initialize all data necessary for this particular HBA. It is passed the host number, so this host knows where the first entry is in the scsi_hosts[] array.
`const char *(* info)(struct Scsi_Host *)`
The info function will return whatever useful information the developer sees fit. If not provided, then the name field will be used instead.
`int (* command)(Scsi_Cmnd *)`
The command function takes a target, a command (this is a SCSI command formatted as per the SCSI specification), a data buffer pointer, and data buffer length pointer. The

return value is a status field which is of integer type. The values of the field can be one of the following:

0 SCSI status code
1 SCSI 1 byte message
2 host error return
3 mid level error return

`int (* queuecommand)(Scsi_Cmnd *, void (*done)(Scsi_Cmnd *))`

The `queuecommand()` function works in a similar manner to the command function. It takes an additional parameter, void (* done)(int host, int code) which is passed the host number and exit result when the command is complete. Host number is the position in the hosts array of this HBA. The `done()` function must only be called after `queuecommand()` has returned.

`int (*eh_abort_handler)(Scsi_Cmnd *)`

Since the SML handles time outs, etc, we want to be able to abort the current command. Abort returns 0 if the abort was successful. If non-zero, the code passed to it will be used as the return code, otherwise DID_ABORT (0) should be returned.

`int (* reset)(Scsi_Cmnd *, unsigned int)`

The reset function will reset the SCSI bus. Any executing commands should fail with a DID_RESET in the host byte. The `Scsi_Cmnd` struct (explained in Section 5.2.3) is passed so that the reset routine can figure out which HBA should be reset, and also which command within the command block was responsible for the reset in the first place.

`int (* bios_param)(Disk *, kdev_t, int [])`

This function determines the bios parameters for a given hard disk. These tend to be numbers that are made up by the HBA. Parameters: size, device number, list (heads, sectors, cylinders).

`int can_queue`

It is set to the maximum number of simultaneous commands a given HBA will accept.

`int this_id`

This is the id of the host and is set to 7 in most cases.

`int sg_tablesize`

This determines the degree to which the HBA is capable of scatter-gather. The scatter-gather list is discussed later.

`unsigned use_clustering`

If true, all the contiguous memory blocks in the scatter-gather list are clustered.

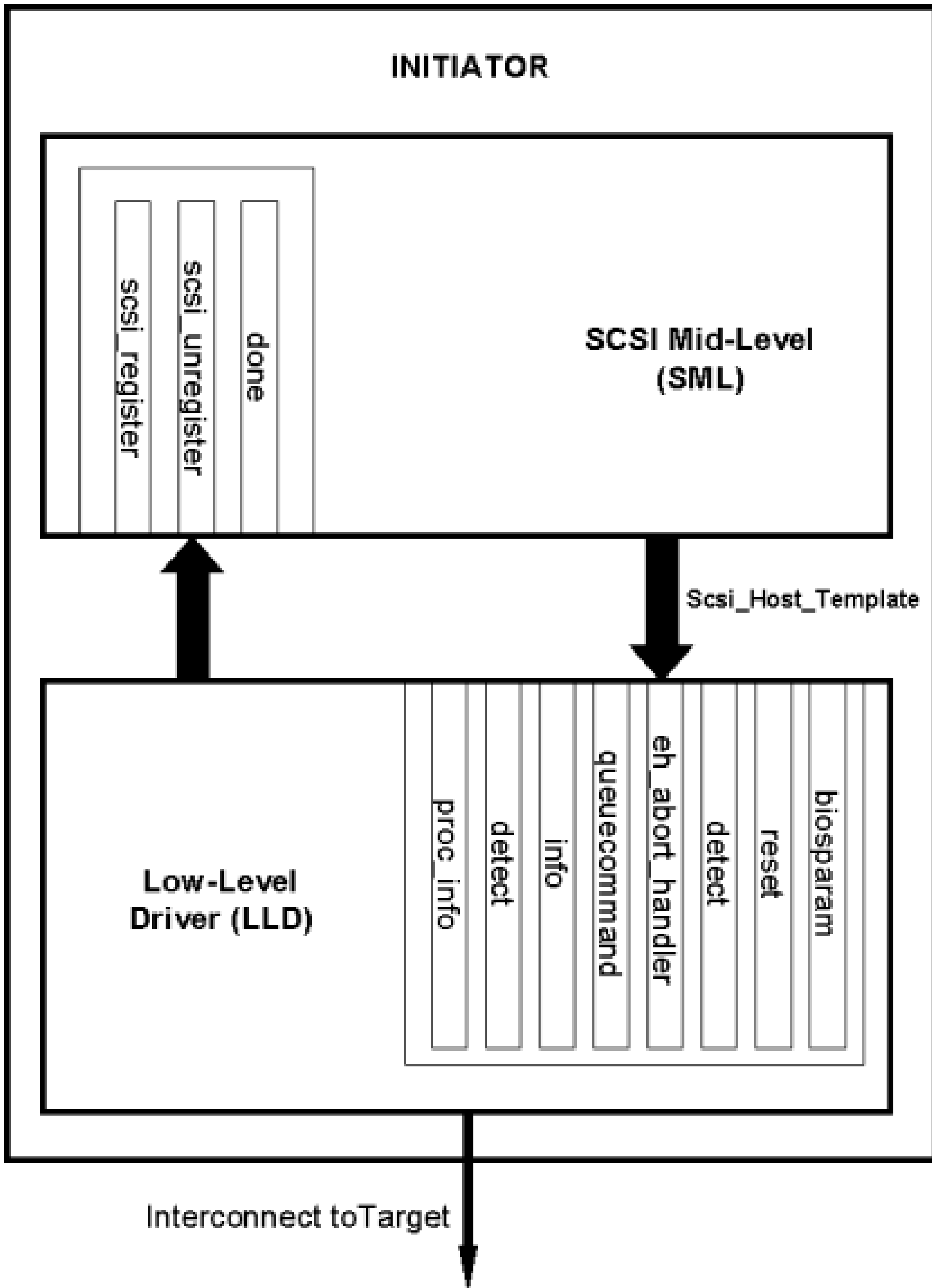


Fig. 5.3 API between SCSI Mid-Level and Low-Level driver.

5.2.2 `Scsi_Host` struct

The `Scsi_Host` (Fig. 5.4) structure is used to describe each instance of a HBA on the system. For example, if there are two Adaptec cards in the system, then there are two instances of `Scsi_Host` structure, one for each. However, there will be only one instance of the `Scsi_Host_Template` struct.

```
struct Scsi_Host
{
      struct Scsi_Host      * next;
      Scsi_Device           * host_queue;
      struct task_struct    * ehandler;
      struct semaphore      * eh_wait;
      struct semaphore      * eh_notify;
      struct semaphore      * eh_action;
      unsigned int            eh_active:1;
      wait_queue_head_t       host_wait;
      Scsi_Host_Template    * hostt;
      unsigned short host_no;
      unsigned long last_reset;
      unsigned int max_id;
      unsigned int max_lun;
      unsigned int max_channel;
      unsigned char dma_channel;
      unsigned int  irq;
      unsigned int unique_id;
      unsigned char max_cmd_len;
      int this_id;
      int can_queue;
      short cmd_per_lun;
      short unsigned int sg_tablesize;
      unsigned use_clustering;
      void (*select_queue_depths)(struct Scsi_Host *, Scsi_Device *);
};
```

Fig. 5.4 Scsi_Host struct Definition.
(Certain fields in the struct have been excluded for the sake of brevity)

The important fields for LLD implementation are discussed as follows:

`unsigned int host_no`
This is the host number for this HBA.
`unsigned int max_id`
This is the maximum SCSI ID for targets/disks accessible through the HBA. The default value is 8.
`unsigned int max_lun`
This is the maximum SCSI LUN for targets/devices accessible through the HBA.
`unsigned int max_channel`
This is the maximum channels for targets/disks accessible through the HBA.
`int this_id, int can_queue, int sg_tablesize, unsigned use_clustering`
All these fields have been discussed in Section 5.2.1

5.2.3 Scsi_Cmnd struct

The `Scsi_Cmnd` struct (Fig. 5.5) contains all the information associated with a SCSI Command. This structure represents a single command that is queued to the LLD. All the context associated with the actual running command is stored in this structure. As the task accomplishment for a command progresses, the state of the command is maintained in this data structure.

```
struct Scsi_Cmnd{
      struct Scsi_Host *host;
      unsigned short state;
      Scsi_Device *device;
      Scsi_Request *sc_request;
      struct scsi_cmnd *next;
      int eh_state;
      void (*done) (struct scsi_cmnd *);
      int retries;
      int allowed;
      int timeout_per_command;
      unsigned volatile char internal_timeout;
      struct scsi_cmnd *bh_next;
      unsigned int target;
      unsigned int lun;
      unsigned int channel;
      unsigned char cmd_len;
      unsigned char sc_data_direction;
      unsigned char cmnd[MAX_COMMAND_SIZE];
      struct timer_list eh_timeout;
      void *request_buffer;
      unsigned char data_cmnd[MAX_COMMAND_SIZE];
      unsigned short use_sg;
      unsigned short sglist_len;
      unsigned short abort_reason;
      unsigned bufflen;
      void *buffer;
      unsigned underflow;
      unsigned transfersize;
      int resid;
      struct request request;
      unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE];
      unsigned flags;
      void (*scsi_done) (struct scsi_cmnd *);
      Scsi_Pointer SCp;
      int result;
};
```

Fig. 5.5 Scsi_Cmnd struct Definition.

(Certain fields in the struct have been excluded for the sake of brevity)

The important fields for LLD driver implementation are discussed as follows:

<u>`struct Scsi_Host *host`</u>

This is a pointer to the `Scsi_Host` associated with the device.

`unsigned short state`

This indicates the current status of this command. It can take the following values: SCSI_ STATE_TIMEOUT, SCSI_STATE_FINISHED, SCSI_STATE_FAILED, SCSI_STATE_QUEUED, SCSI_STATE_UNUSED, SCSI_STATE_DISCONNECTING, SCSI_STATE_INITIALIZING, SCSI_STATE_BHQUEUE, SCSI_STATE_MLQUEUE

`void (*done) (struct scsi_cmnd *)`

This is a pointer to a function for command completion implemented in the SML. This function is called by the LLD, after it receives a SCSI Response from the Target (Disk).

`int retries`

This variable denotes the number of times a command has been retried by the SML.

`int timeout_per_command`

This variable denotes the time (in jiffies), the SML waits before it decides that a command has timed out. If a SCSI Command is not satisfied within this time interval, the SCSI command is aborted by calling the `abort()` function defined in `Scsi_Host_Template` struct.

`unsigned int target`

This is the unique id used to refer to the SCSI target (disk) to which the SCSI Command has to be sent.

`unsigned int lun`

This refers to the logical unit (LUN) within the SCSI target (disk). If the target supports only a single LUN, then this number is 0.

`unsigned char cmd_len`

This variable gives the length of the Command Descriptor Block (CDB). It can be 6, 10, 12 or 16 bytes in length.

`unsigned char sc_data_direction`

This variable is a boolean and is used to denote if the SCSI Command associated is for a READ or WRITE operation.

`unsigned char cmnd[MAX_COMMAND_SIZE]`

This is the actual Command Descriptor Block (CDB) associated with a SCSI Command.

unsigned request_bufflen

This variable specifies the total number of bytes of data transfer involved with the SCSI Command.

`void *request_buffer`

This is the data buffer associated with the SCSI Command. For a READ operation, this buffer has to be filled in by the Target (disk) and for a Write operation, this buffer has to be written to the Target (disk).

`unsigned short use_sg`

This variable is used to denote the number of scatter-gather buffers associated with the SCSI Command. `use_sg=0` denotes presence of a single char buffer.

`unsigned short abort_reason`

This variable is set when a SCSI Command has to be aborted by the SML. The reason is specified by this variable.

`unsigned char sense_buffer[SCSI_SENSE_BUFFERSIZE]`

This buffer is used for sense data by the LLD. The sense data is received by the target and is filled before making a call to the `done()` function.

`int result`

The LLDs fill in this field with the status of the command prior to calling the SML completion routine. The status consists of status_byte, the msg_byte, and the driver_byte.

- The status byte is the status that is returned from the device.
- The driver_byte is the status returned by the LLD.
- The msg_byte is the message byte that comes back from the device.

5.3 Discovery Process for SCSI Targets

The LLD in the SCSI subsystem is software code and drives the HBA, the physical entity present on the system to access the targets (Fig. 5.1). When an LLD is loaded, the corresponding HBA is registered with the system. The registration happens through the `register()` function implemented in the LLD. The `register()` function has to specify the number of targets and LUNs per target available through the particular HBA. The SCSI Upper-Level will give target IDs to all the targets available through the HBA. The SML will then try to get basic information about the different targets (disks) by sending SCSI Commands through the `queuecommand()` function implemented in the LLD. The SCSI Commands sent to the target are of types INQUIRY, TEST_UNIT_READY and DISK_CAPACITY (different SCSI Command types/opcodes are explained in Appendix B). It will then try to READ block 0 which is the superblock for the SCSI device. After successful completion of these commands, the Operating system assigns device names /dev/sdx (compliant with SCSI naming convention for the Linux Operating system) to the discovered SCSI targets/disks. The discovered SCSI targets/disks can be accessed like any normal disk for I/O operation, making partitions, making a filesystem, etc.

Chapter 6

SEP Initiator Design And Implementation

In this chapter, we first give an overview of the SEP Low-Level Driver design. The configuration tool 'sep_config', which is used to make a TCP connection and break the TCP connection with the target, is explained next. The important data structures for book-keeping with the SEP Low-Level Driver are also discussed. Finally, we discuss the `Scsi_Host_Template` Interface Implementation in the SEP Low-Level Driver.

6.1 Overview of SEP Low-Level Driver (LLD) Design

When the SEP low-level driver (LLD) is loaded, it registers itself with the SCSI Mid-level (SML). The SEP LLD provides to the SML, through the `Scsi_Host_Template` struct, a list of functions that enable the SML to pass commands to the SEP LLD. As soon as the SML is registered, it tries to get basic information about the targets(disks) by passing SCSI commands INQUIRY, TEST_UNIT_READY, DISK_CAPACITY. The SML then attempts to send a READ Command (to read superblock of the target/disk) to the SEP LLD. The TCP connection to the target is not made at the time of registration of SEP LLD, so the SML is not able to get this information.

The 'sep_config' configuration tool, explained in Section 6.2, is used to make a connection to the target through the `/proc` filesystem interface. After a TCP connection is made, the LLD enters the Login Phase and does Text Parameter negotiation. After a socket is created, a kernel thread `sep_thread` (Fig. 6.1) is generated which is responsible for communication with the target. The SEP LLD supports multiple targets and for each target, a separate `sep_thread` is created for an established TCP connection. The `sep_thread` sends and receives SEP packets to/from a particular target (disk). As soon as a thread is spawned, it sends a 'Connect and Negotiate' message to the target which is responded to by a 'Negotiation Response' message from the target. After this sequence of communication messages between the initiator and target has been exchanged, the LLD notifies the SML that it is ready to send SCSI commands to the target. The current implementation of the SEP LLD does not negotiate the flow parameter values (specified by the SEP protocol) with the target. The SEP LLD then waits on a queue to get commands from the SML.

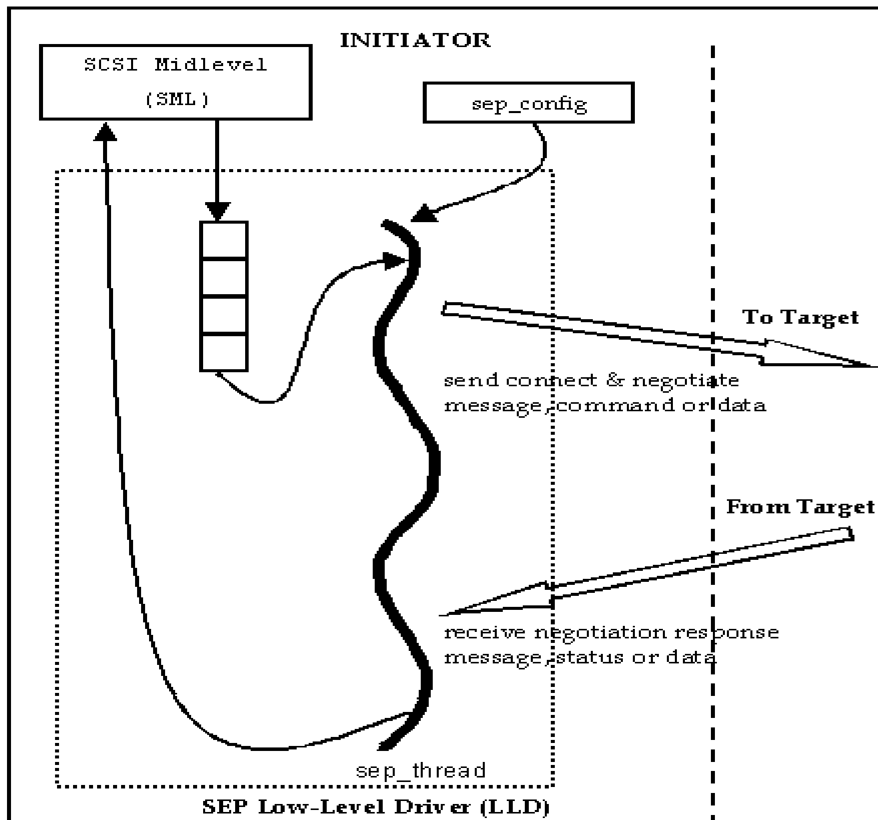


Fig. 6.1 SEP Low-Level Driver (LLD) Design.

Each WRITE request by the SML results in the `sep_thread` sending two SEP Payload Data Units (PDUs) to the target: the first is the SEP 'Simple Tagged Command' PDU (opcode 0x01), which contains the SCSI CDB as the payload; the second is the SEP 'SCSI Data' PDU (opcode 0x04), which is the WRITE Data itself. The `sep_thread` then waits for a SEP 'SCSI STATUS' PDU (opcode 0x05) from the target to know if the data was written correctly or not. After this write request, the `sep_thread` again waits on a wait queue to get the next SCSI command from the SML.

Each READ request, on the other hand, will just result in the `sep_thread` sending to the target a single SEP 'Simple Tagged Command' PDU (opcode 0x01), which contains the SCSI CDB. The `sep_thread` then waits for the data from the target to be read into the data buffers provided by the SML. After this, the `sep_thread` again sits on a wait queue to get the next SCSI command from the SML.

As required by the SEP protocol, fragmentation of SEP packets has been implemented, which means that if the amount of data is greater than 65535 bytes, then more than one SEP packet is used to send it. The other side (initiator or target) receives all the SEP data packets till it gets the last one, which is denoted by the 0x80 flag in the SEP header.

When there is a huge amount of data to be written or read, the SML passes an array of data buffer pointers (instead of single data buffer pointer) to the SEP LLD, which is referred to as 'scatter-gather list'. The scatter-gather list is used when the requested

amount of memory is not available contiguously to the SML. The SEP LLD can handle a scatter-gather list, that is, if it gets an array of data pointers, it will pass all the pointers to the TCP/IP socket functions appropriately.

6.2 sep_config

sep_config is a user-level program application which brings the SEP protocol support up/down in the initiator. The program can be used to bring up the initiator only if the scsi_sep module is loaded. The sep_config program can be invoked by the following syntax from the shell command prompt:

```
sep_config up/down [ip_address or hostname] [host number]
                   [lun number]
```

up/down	-	bring up/down the SEP initiator protocol support in the Linux O/S.
ip_address	-	ip_address or hostname of the target.
host number	-	SCSI host number which is entered in the `/proc` filesystem. It specifies the SCSI Host Bus Adapter number assigned by the kernel.
lun number	-	specifies the LUN on a specific target device.

When the sep_config interface is brought up, the sep_config initiates a TCP connection by calling SEP LLD functions and writes the ip_address and LUN number of the target in the /proc/scsi/scsi_sep/host_number file. To cause a disconnect, one brings down the sep_config interface which, as a result, breaks the TCP connection with the target. Connection/Disconnection is done by specifying ‘up’/‘down’ as the first parameter when invoking the sep_config program. sep_config emulates the ifconfig up/down interface provided for Ethernet drivers on Linux.

6.3 Data structures involved in SEP LLD

There is one data structure involved in keeping state information with the LLD: `sep_control_block` (Fig. 6.2). The `sep_control_block` struct keeps specific information for a single active target.

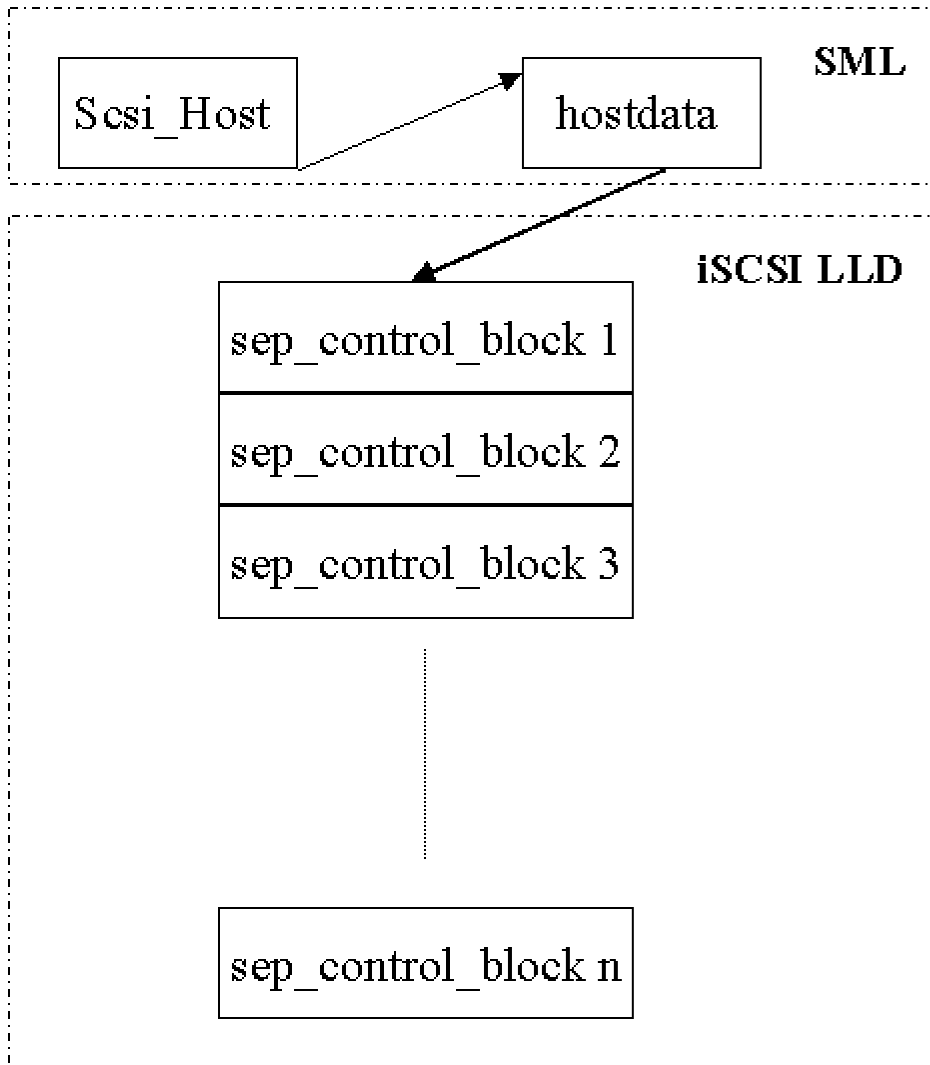


Fig. 6.2 Organization of Data Structures in SEP LLD.

The fields in `sep_control_block` struct (Fig. 6.3) are explained as follows:

`struct socket * sock;`
This variable stores the socket information about a TCP connection.
`sruct task_struct * sep_thread;`
This thread is used to transmit/receive SEP PDUs from/to the target.
`Scsi_Cmnd *Cmnd;`
This is a pointer to the `Scsi_Cmnd` struct maintained by the SML.
`struct iovec iov_tx[SGLIST+2];`
This struct stores the buffer pointers and is used by the TCP routines to send data through the TCP socket.
`struct msghdr tx_msghdr;`
This struct is passed as a parameter to the TCP function, `sock_sendmsg()`. It contains information about the TCP buffers that are involved in sending data through the TCP socket.

```
struct sep_control_block {
  struct socket * sock;
  struct semaphore rx_sem;
  struct semaphore tx_sem;
  struct task_struct * sep_thread;
  Scsi_Cmnd *Cmnd;
  struct msghdr msg_tx;
  struct msghdr msg_rx;
  struct iovec iov_tx[SG_LIST+2];
  struct iovec iov_rx[SG_LIST+1];
  struct sep_header sephdr_tx;
  struct sep_header sephdr_rx;
  void   (*global_done) ( Scsi_Cmnd *);
  wait_queue_head_t global_wait_queue;
  int scsi_sep_ip_address;
}
```

Fig. 6.3 `sep_control_block` struct Definition.

`struct iovec iov_rx[SGLIST+1];`

This struct stores the buffer pointers and is used by the TCP routines to receive data through the TCP socket.

`struct msghdr rx_msghdr;`

This struct is passed as a parameter to the TCP function, `sock_recvmsg()`. It contains information about the TCP buffers that are involved in receiving data through the TCP socket.

`struct sep_header sephdr_tx;`

This struct stores the SEP header that is to be sent to the target.

`struct sep_header sephdr_rx;`

This struct stores the SEP header that is received from the target.

`wait_queue_head_t global_wait_queue;`

This is the wait queue where the SML queues up SCSI Commands for the `sep_thread` to process them.

`void   (*global_done) ( Scsi_Cmnd *);`

This is a function pointer to the `done()` function implemented by the SML, and is called by the SEP LLD whenever a SCSI Command has been processed.

`int scsi_sep_ip_address;`

This variable stores the IP address of the target and is used when making a TCP connection to the target.

6.4 The SCSI_Host_Template Implementation

The SEP initiator code which forms the LLD in the SCSI Initiator subsystem, has implemented the functions described in the `SCSI_Host_Template` struct in Chapter 5. The following function definitions and field values are defined in the jump table present in scsi_sep.h (Fig. 6.4).

```
#define SCSI_SEP {

detect:            scsi_sep_detect,
release:           scsi_sep_release,
proc_info:         scsi_sep_proc_info,
info:              scsi_sep_info,
ioctl:             scsi_sep_ioctl,
queuecommand:      scsi_sep_queuecommand,
eh_abort_handler: scsi_sep_abort,
reset:             scsi_sep_reset,
bios_param:        scsi_sep_bios_param,
can_queue:         1,
this_id:           7,
sg_tablesize:      64,
cmd_per_lun:       1,
unchecked_isa_dma:0,
use_clustering:    ENABLE_CLUSTERING,
use_new_eh_code:   1
}
```

Fig. 6.4 `SCSI_Host_Template` struct Definition.

The API functions implemented in the SEP LLD (Fig. 6.5) are described as follows:

`int scsi_sep_detect(Scsi_Host_Template * tmpt)`

This function is called when the SEP LLD is loaded. This function calls the `scsi_register` function, which is implemented in the SML, in order to register the SEP initiator. The `LUN` and `target` fields in the `tmpt` struct specify the number of targets and number of LUNs in each target that are accessible through this initiator. The `hostdata` struct (containing SEP LLD specific information) is registered with the SML that keeps the state information about all the targets. The pointer to the `global_hostdata` struct is saved as a field in `Scsi_Host` struct (returned when SML `scsi_register()` function is called).

`int scsi_sep_release(Scsi_Host_Template * tmpt)`

This function is called when the sep initiator driver module is unloaded. This function unregisters the initiator by calling the `scsi_unregister` function implemented in the SCSI Mid-level. The socket connection with the target is released and the `sep_thread` is terminated.

`int scsi_sep_proc_info(char *buffer, char **start, off_t offset, int length, int inode, int inout)`

This function is called when the user level configuration tool 'sep_config' (explained in Section 6.2) is executed. It can also be called from the `/proc` interface.

`const char *scsi_sep_info(struct Scsi_Host *host)`

This function is called whenever the SEP initiator driver module is loaded. This function returns a buffer denoting the SEP initiator driver name in string format.

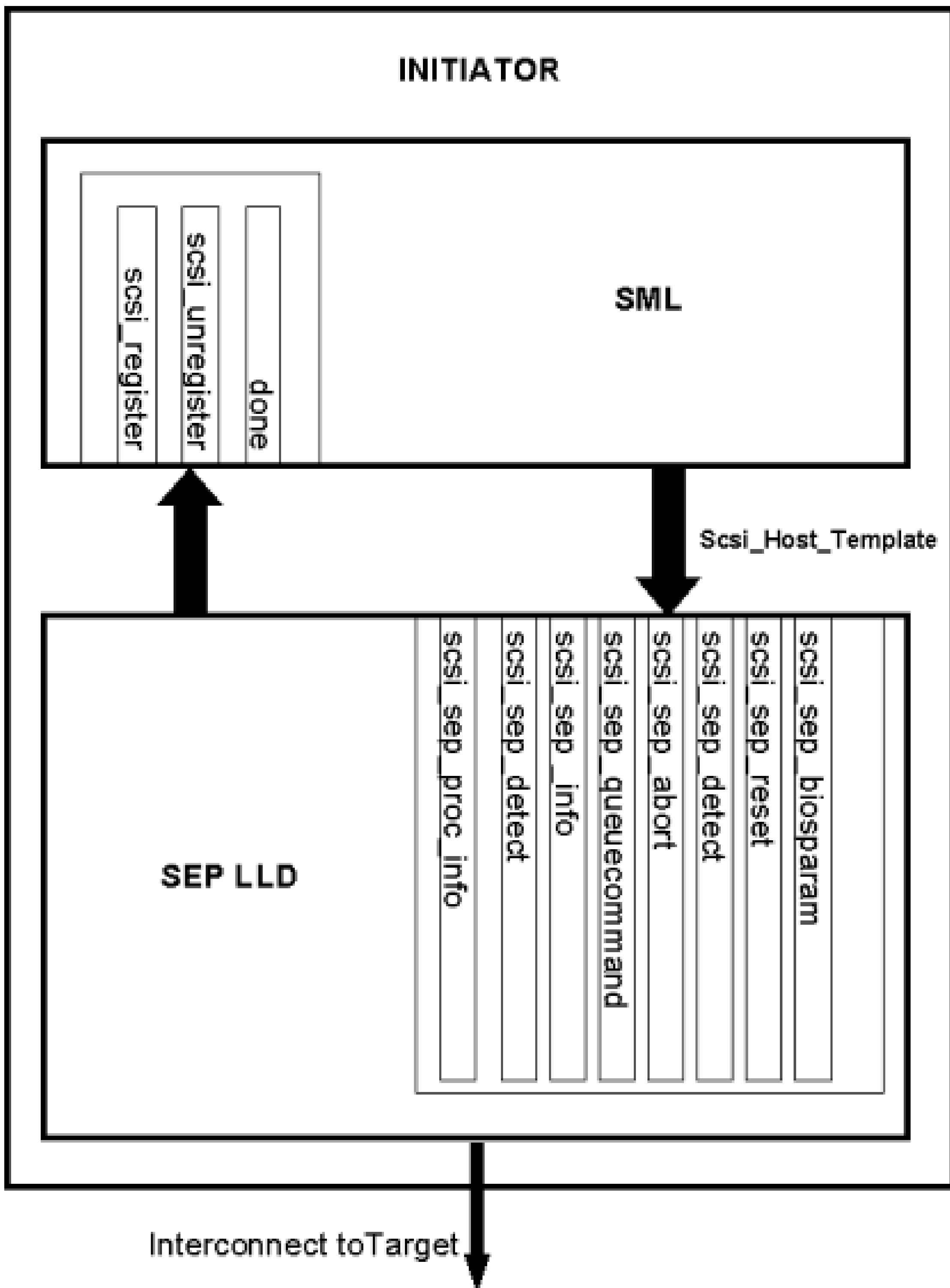


Fig. 6.5 API between SCSI Mid-Level and SEP Low-Level Driver.

`int scsi_sep_queuecommand(Scsi_Cmnd * Cmnd, void (*done) (Scsi_Cmnd *))`

This function is called whenever the SCSI Initiator Mid-level has to pass any SCSI Command to the SEP initiator driver. This function is called in the context of a global lock called `io_request_lock`, so there is a requirement that we limit processing that is to be done in this function. This function calls the `do_command()` function implemented in the SEP Initiator driver which queues up the SCSI command for the `sep_thread`. The `sep_thread` is responsible for transmitting the SCSI command encapsulated in the 8 byte SEP header to the target.

`int scsi_sep_abort(Scsi_Cmnd * Cmnd)`

This function is called whenever the SML has to abort any SCSI Command after a time-out is reached. This function is also called in the context of a global lock called `io_request_lock`, so there is a requirement that we that we limit processing that is to be done in this function. There are no error recovery mechanisms in the SEP implementation, so we just break the socket connection and kill the `sep_thread`.

`int scsi_sep_reset(Scsi_Cmnd * Cmnd)`

This function is exactly the same as the function `scsi_sep_abort`.

Chapter 7

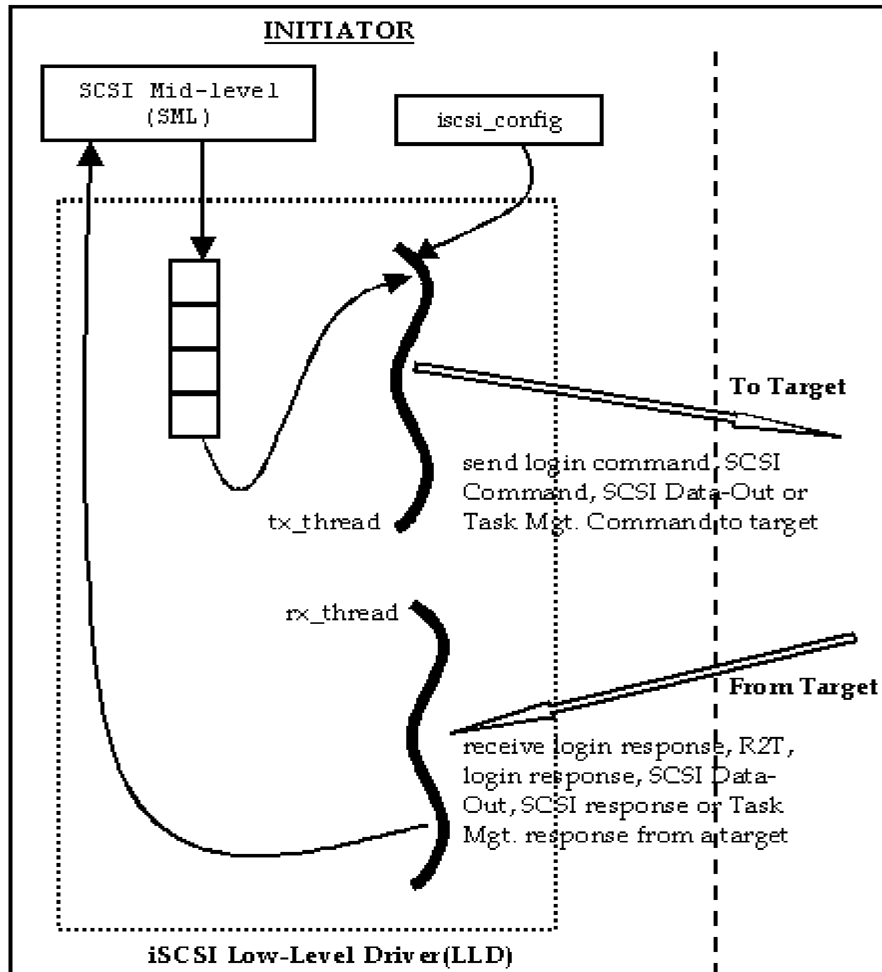
iSCSI Initiator Design And Implementation

In this chapter, we first give an overview of the iSCSI Low-Level Driver design. The important data structures for book-keeping with the iSCSI Low-Level Driver are also discussed. The `Scsi_Host_Template` Interface Implementation in the iSCSI Low-Level Driver is explained next. Finally, step-by-step procedures to handle READ, WRITE, and TASK MANAGEMENT Requests are described.

7.1 Overview of iSCSI Low-Level Driver (LLD) Design

When the iSCSI low-level driver (LLD) is loaded, it registers itself with the SCSI Mid-level (SML). The iSCSI LLD provides to the SML, through the `Scsi_Host_Template` struct, a list of functions that enable the SML to pass commands to the iSCSI LLD. As soon as the SML is registered, it tries to get basic information about the targets(disks) by passing SCSI commands INQUIRY, TEST_UNIT_READY, DISK_CAPACITY. The SML then attempts to send a READ Command (to read superblock of the target/disk) to the iSCSI LLD. The TCP connection to the target is not made at the time of registration of iSCSI LLD, so the SML is not able to get this information.

The 'iscsi_config' configuration tool, developed by Narendran Ganapathy (Appendix C), is used to make a connection to the target through the `/proc` filesystem interface. After a TCP connection is made, the LLD enters the Login Phase and does Text Parameter negotiation. After successful Login Phase Parameter Negotiation, the iSCSI LLD spawns two threads. A transmit thread (`tx_thread`) is started that can transmit iSCSI PDUs of type 'SCSI Command' (opcode 0x01), 'Task Management Command' (opcode 0x02) and 'SCSI DataOut' (opcode 0x02) to the target. A receive thread (`rx_thread`) is started that can receive iSCSI PDUs of type 'SCSI Response' (opcode 0x21), 'Task Management Response' (opcode 0x22), 'SCSI Data-In' (opcode 0x25), 'Ready to Transfer' (opcode 0x31), and 'Reject' (opcode 0x3f) from the target.



--- TCP Connection between initiator & target

Fig 7.1 iSCSI Low-Level driver (LLD) Design.

After starting the two threads (Fig. 7.1), the iSCSI LLD enters the Full Feature Phase and notifies the SML that a target with a particular target number is accessible for I/O operations. The SML again passes the SCSI commands INQUIRY, TEST_UNIT_READY, DISK_CAPACITY, and READ (to read the superblock of the target/disk) to the iSCSI LLD. The iSCSI LLD is able to get all the relevant information from the Target as it has an established TCP connection and is in the Full Feature Phase.

When a new command is passed to the LLD through the `Scsi_Host_template` API, the LLD sets up the TCP buffers to send the iSCSI PDU of type 'SCSI Command' (opcode 0x01). The `tx_thread` is woken up to send the PDU to the target. Data handling for any SCSI Command by the LLD depends on the operation involved: READ or WRITE. We discuss data handling by the iSCSI LLD in two separate cases:

- For a READ request, the LLD receives the 'SCSI Data-In' PDUs through the `rx_thread`. It fills the data buffers provided by the SML.

- For a WRITE request, the LLD sends the 'SCSI Data-Out' PDUs as 'Unsolicited Data' or in response to 'Ready to Transfer' PDUs received by the `rx_thread` from the target.

After all the Data has been transferred between the initiator and the target, the iSCSI LLD waits for a iSCSI 'SCSI Response' PDU (0x02) to be received from the target. On receipt of iSCSI 'SCSI Response' PDU in the `rx_thread`, the LLD delivers status and sense data, if present, to the SML. The LLD then frees up the resources allocated for the particular SCSI command.

When a SCSI command passed to the iSCSI LLD has to be aborted, the LLD checks if the command has been sent to the target or not. These two cases are discussed separately:

- If the SCSI command to be aborted was sent to the target, the LLD sets up the TCP buffer with the iSCSI 'Task Management Command' (opcode 0x02) PDU and wakes up the `tx_thread`. The `tx_thread` sends the PDU to the target. The LLD waits for a iSCSI 'Task Management Response' PDU (opcode 0x22) to be received from the target. After receiving the iSCSI 'Task Management Response' PDU, the SML is notified of the abort command's success or failure. In either case, the LLD frees up the resources allocated for the abort command and the command that had to be aborted.
- If the command to be aborted was not sent to the target, the LLD just frees up the resources allocated for the command that had to be aborted and returns.

The iSCSI LLD design supports multiple sessions and multiple connections within a session as specified by the iSCSI protocol. A separate `rx_thread` receives iSCSI PDUs from each connection in a session from the target. A separate `tx_thread` is used to send iSCSI PDUs on each connection in a session to the target.

When there is a huge amount of data to be written or read, the SML passes an array of data buffer pointers (instead of single data buffer pointer) to the iSCSI LLD, which is referred to as 'scatter-gather list'. The scatter-gather list is used when the requested amount of memory is not available contiguously to the SML. The iSCSI LLD can handle a scatter-gather list, that is, if it gets an array of data pointers, it will pass all the pointers to the TCP/IP socket functions appropriately.

7.2 Data structures involved in iSCSI LLD

There are three data structures involved in keeping state information about sessions, connections and pending commands with the iSCSI LLD: `session, connection` and

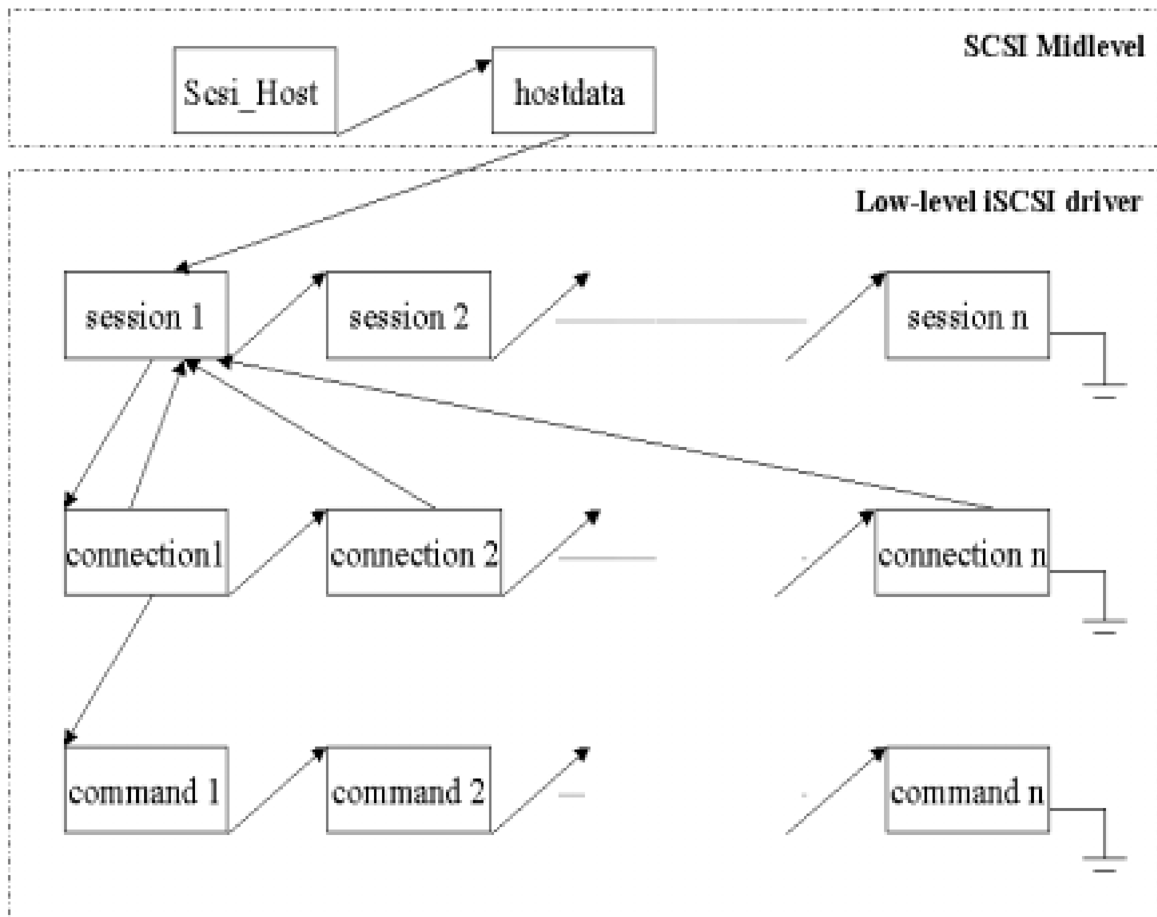


Fig. 7.2 Organization of data structures in iSCSI LLD.

`command` struct (Fig. 7.2). The `session` struct maintains session specific information for each active target. For each connection in a session, a unique `connection` struct is required for bookkeeping of connection specific information. The pending commands on each connection are maintained in the `command` struct. The `hostdata` struct registered with the SML has a field pointing to the head of the `session` struct linked list.

7.2.1 session struct

The fields in `session` struct (Fig. 7.3) are explained as follows:

`__u8 *tx_buf;`
This pointer points to the buffer that is to be sent to the target.
`__u32 scsi_target_id;`
This is the SCSI ID assigned by the SML.
`__u32 cur_cmd_sn;`
This variable maintains the command sequence numbering for each session and is updated whenever a new iSCSI PDU (SCSI Command, SCSI Task Management Command, Login Command, Text Command or Logout Command) is sent to the target.

```
struct session
    {
      __u8    *tx_buf;
      __u32   scsi_target_id;
      __u32   session_state;
      __u32   cur_cmd_sn;
      __u32   max_cmd_sn;
      struct connection *connection_head;
      sruct session *next;
      sruct task_struct * tx_thread;
      struct semaphore tx_sem;
      struct semaphore task_mgt_sem;
      unsigned int text_param_len;
      void *text_param;
      struct parameter_type (*session_params)[MAX_CONFIG_PARAMS];
    };
```

Fig. 7.3 `session` struct Definition.

`__u32 session_state;`

The `session_state` values (Table 7.1) are explained in the following table:

STATE	MEANING
STATUS_NOT_PRESENT	Session has not entered the Login Phase
STATUS_CONNECTED	Session is in the Login Phase
STATUS_LOGGED_IN	Session is in the Full Feature Phase

Table 7.1 Session State Table.

`__u32 max_cmd_sn;`

This variable stores the maximum value of command sequence number that any iSCSI PDU can have. It is updated from the iSCSI PDUs received from the target.

`struct connection *connection_head;`

This pointer points to the head of the `connection` struct linked list.

`sruct task_struct * tx_thread;`

This thread is used to transmit iSCSI PDUs to the target.

`struct semaphore tx_sem;`

The `tx_thread` waits on this semaphore and if there is any iSCSI PDU to be sent from the initiator to the target, it is woken up from this semaphore.

`struct semaphore task_mgt_sem;`

The `abort()` function, defined in `Scsi_Host_Template` structure and called by the SML, waits on this semaphore after sending iSCSI 'Task Management Command' PDU to the target. When the Task Management Response is received from the target, the `rx_thread` wakes up the `abort()` function waiting on this semaphore.

`struct parameter_type (*session_params)[MAX_CONFIG_PARAMS];`

This pointer points to the session specific operational parameters.

7.2.2 connection struct

The fields in `connection` struct (Fig. 7.4) are explained as follows:

```
struct connection
      {
      __u8        *rx_buf;
      __u32   target_address;
      __u32   target_port;
      __u32   exp_stat_sn;
      struct socket *sock;
      struct task_struct * rx_thread;
      struct semaphore rx_sem;
      struct session *my_session;
      struct command *pending_commands;
      struct connection *next;
      struct parameter_type (*connection_params)[MAX_CONFIG_PARAMS];
      };
```

Fig. 7.4 connection struct Definition.

`__u8 *rx_buf;`

This pointer points to the buffer in which we receive data from the target.

`__u32   target_address;`

This variable stores the IP address of the target and is used whenever a new TCP connection is to be made to the target.

`__u32   target_port;`

This variable stores the TCP Port Number of the target and is used whenever a new TCP connection is to be made to the target.

`__u32   exp_stat_sn;`

This variable stores the Expected Status Number for an iSCSI PDU received on a connection. Status numbering is unique for each connection.

`struct socket *sock;`

This variable stores the socket information about a TCP connection.

`struct task_struct * rx_thread;`

This thread is used to receive iSCSI PDUs from the target.

`struct session *my_session;`

This pointer points to the session struct to which this connection belongs.

`struct command *pending_commands;`

This pointer points to the head of the `command` struct linked list for this particular connection. This list stores the pending commands on this connection.

`struct parameter_type (*connection_params)[MAX_CONFIG_PARAMS];`

This pointer points to the connection specific operational parameter struct.

7.2.3 command struct

The fields in `command` struct (Fig. 7.5) are explained as follows:

`__u32 target_xfer_tag;`

This variable has a unique number and is assigned by the target.

`__u32 init_task_tag;`

This variable has a unique number and is assigned by the iSCSI LLD. It is used for searching commands through the `command` struct linked list whenever an iSCSI PDU is received from the target.

```
struct command
      {
      __u8    task_mgt_response;
      __u8    task_mgt_function;
      __u32   target_xfer_tag;
      __u32   init_task_tag;
      __u32   data_offset;
      __u32   r2t_xfer_length;
      __u32   tx_size;
      Scsi_Cmnd *SCpnt;
      struct iscsi_init_scsi_data_out data;
      struct iovec iov[66];
      struct msghdr tx_msghdr;
      struct iscsi_init_scsi_cmnd iscsi_cmd;
      struct command * next;
      };
```

Fig. 7.5 `command` struct Definition.

`__u8 task_mgt_response;`

This field is used only for `command` struct maintained for task management functions. It stores the Task Management Response received from the target and it can take the following values (Table 7.2):

Value	Meaning
0	Function Complete
1	TASK was not in the task set
2	LUN does not exist
255	Function Rejected

Table 7.2 Task Management Response Values.

`__u8 task_mgt_function;`

This field is used only for `command` struct maintained for task management functions. It stores the Task Management Function value (Table 7.3) depending on the functionality required by SML:

Function	Value
ABORT_TASK	1
ABORT_TASK_SET	2
CLEAR_ACA	3
CLEAR_TASK_SET	4
LOGICAL_UNIT_RESET	5
TARGET_WARM_RESET	6
TARGET_COLD_RESET	7

Table 7.3 Task Management Function Values.

`__u32 r2t_xfer_length;`

When an R2T is received for any command, the transfer length for it is stored by this variable.

`__u32 tx_size;`

This variable stores the total number of bytes that has to be transmitted for a single iSCSI PDU.

`Scsi_Cmnd *SCpnt;`

This is a pointer to the command's corresponding `Scsi_Cmnd` struct maintained by the SML. Every `command` struct, maintained by the LLD, has a unique `Scsi_Cmnd` struct.

`struct iscsi_init_scsi_data_out data;`

The iSCSI 'SCSI Data_Out' header is stored in this struct. Whenever a iSCSI 'SCSI Data_Out' PDU has to be sent to the target, this struct is filled and the `iovec` buffer pointer is pointed to this struct.

`struct iovec iov[66];`

The `iovec` struct store the buffer pointers and is used by the TCP routines to send/receive data from the TCP sockets.

`struct msghdr tx_msghdr;`

This struct is passed as a parameter to the TCP functions, `sock_recvmsg()` and `sock_sendmsg()`. It contains information about the TCP buffers that are involved in sending/receiving data from the TCP socket.

`struct iscsi_init_scsi_cmnd iscsi_cmd;`

The iSCSI 'SCSI Command' header is stored in this struct. Whenever a iSCSI 'SCSI Command' PDU has to be sent to the target, this struct is filled and the `iovec` buffer pointer is pointed to this struct.

7.3 The SCSI_Host_Template Implementation

The iSCSI initiator code, which forms the LLD in the SCSI Initiator subsystem, has the implemented functions that are described in the `SCSI_Host_Template` struct in Chapter 5. The following function definitions and field values (Fig. 7.6) are defined in the jump table present in iscsi_initiator.h (Refer to README in Appendix C).

```
#define ISCSI_INITIATOR {\
        proc_info:            iscsi_initiator_proc_initiator, \
        detect:               iscsi_initiator_detect,\
        release:              iscsi_initiator_release,\
        info:                 iscsi_initiator_info,\
        queuecommand:         iscsi_initiator_queuecommand,\
        eh_abort_handler:     iscsi_initiator_abort,\
        reset:                iscsi_initiator_reset,\
        bios_param:           iscsi_initiator_biosparam,\
        can_queue:            8,\
        this_id:              -1,\
        sg_tablesize:         64,\
        cmd_per_lun:          8,\
        present:              0,\
        unchecked_isa_dma:    0,\
        use_clustering:       ENABLE_CLUSTERING\
}
```

Fig. 7.6 Scsi_Host_Template struct Definition.

The API functions (Fig. 7.7) implemented in the iSCSI LLD are described as follows:

`int iscsi_initiator_detect(Scsi_Host_Template * tmpt)`

This function is called when the iSCSI LLD is loaded. The `scsi_register()` function, which is implemented as SML code, is called to register the iSCSI LLD. The `LUN` and `target` fields in the `tmpt` struct are set to number of targets and number of LUNs per target respectively that are accessible through the LLD or HBA. The `hostdata` struct (Fig. 7.2) is registered with the SML that maintains the iSCSI LLD specific information. Then `init_initiator()` is called that initializes the members `session_head` pointer, `cur_task_tag, param_table` array in `hostdata` struct. The last step is to initialize the `jumbo_sem` semaphore that is used to lock iSCSI specific data structures for mutual exclusion.

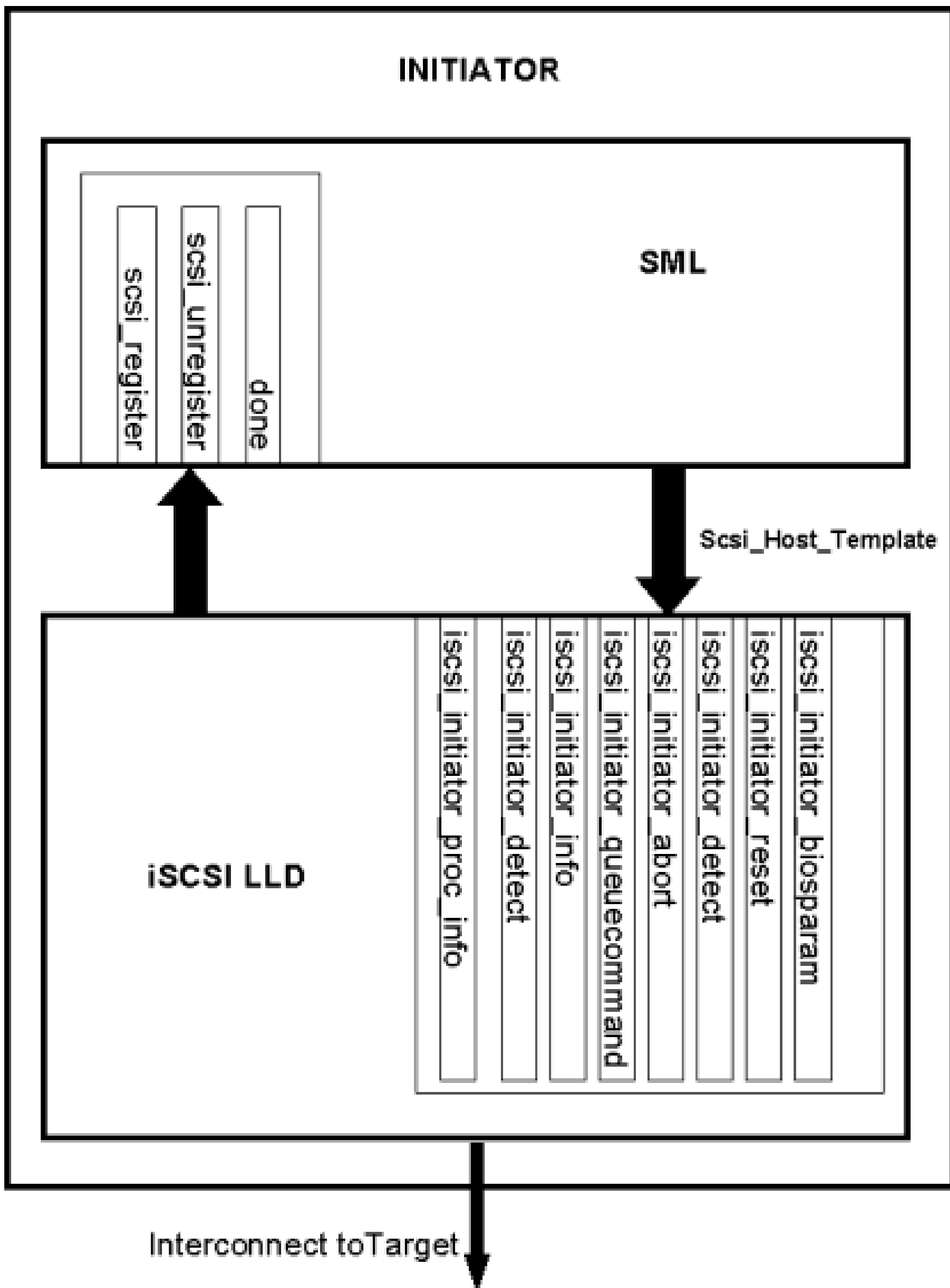


Fig. 7.7 API between SCSI Mid-Level (SML) and iSCSI Low-Level Driver (LLD).

`int iscsi_initiator_release(struct Scsi_Host *host)`

This function is called when the iSCSI LLD is unloaded. The primary goal of this function is to unregister the iSCSI LLD and do cleanup of data structures and threads involved. The tasks accomplished by this function are as follows:

- `close_session()` function, implemented in the LLD, is called to clean-up the `session` data structure. The call to different functions involved is explained here:
    - `close_connection()` function, implemented in the LLD, is called which kills the `rx_thread`, releases the socket `sock` and clears all the pending commands `command` struct linked list.
    - `tx_thread` is killed and the `oper_param` data structure is freed.
- `scsi_unregister()` function, implemented in the SML, is called which unregisters the iSCSI LLD.

`int scsi_sep_proc_info(char *buffer, char **start, off_t offset, int length, int inode, int inout)`

This function is called when the user level configuration tool 'iscsi_config' is executed. The 'iscsi_config' is explained in detail in Appendix C.

`const char *scsi_sep_info(struct Scsi_Host *host)`

This function is called whenever the iSCSI LLD module is loaded. It returns a buffer containing the iSCSI LLD name in string format.

`iscsi_initiator_queuecommand(Scsi_Cmnd *Cmnd, void(*done)(Scsi_Cmnd*))`

This function is called whenever the SML has to pass a SCSI Command to the iSCSI LLD. This function is called in context of a global lock called `io_request_lock`, so there is a requirement that processing in this function is limited. The steps taken in this function are as follows:

- `session` struct having the same target_id as the `Cmnd`'s target_id for the target is searched through the `session` struct linked list. The `session` struct linked list is accessed through `host->hostdata` struct (Fig. 7.2) which is one of the fields in `Cmnd` struct.
- Memory for a new `command struct` is allocated and initialized with a unique initiator task tag.
- iSCSI 'SCSI Command' header (opcode 0x01) is filled. The datasegmentlength field in the iSCSI 'SCSI Command' header is set to appropriate length looking at the `DataPDULength`, `FirstBurstSize` and `ImmediateData` fields in the `oper_param` struct for a WRITE Command. The CDB from the `Cmnd` struct is copied into the iSCSI 'SCSI Command' header. The F_BIT in the iSCSI header is set if no Unsolicited iSCSI 'Data-Out' PDUs is to be sent. This depends on the negotiated `InitialR2T` Operational Parameter value.
- The TCP `iovec` buffers are set to send the iSCSI 'SCSI Command' PDU which includes the iSCI header and Immediate Data, if present.
- The `tx_thread` waiting on `tx_sem` is woken up to send the set TCP `iovec` buffers.
- The `new command` struct created is added to the linked list of all the pending commands for the connection.

`int iscsi_initiator_abort(Scsi_Cmnd * Cmnd)`

This function is called whenever the SML has to abort any SCSI Command after a time-out or due to some error. This function is called in context of a global lock called `io_request_lock`, so there is a requirement that processing in this function is limited. The steps taken in this function are as follows:

- `session` struct having the same target_id as the `Cmnd`'s target_id for the target is searched through the linked list of `session` struct. The `session` struct linked list is accessed through `host->hostdata` struct which is one of the fields in `Cmnd` struct.
- Memory for a new `command struct` is allocated and initialized with a unique initiator task tag.
- iSCSI Task Management Command header (opcode 0x02) is filled.
- The TCP `iovec` buffers are set to send the iSCSI 'Task Management Command' PDU.
- The `tx_thread` waiting on `tx_sem` is woken up to send the set TCP `iovec` buffers.
- The `new command` struct created is added to the linked list of all the pending commands for the connection.

`int iscsi_initiator_reset(Scsi_Cmnd * Cmnd)`

This function is not implemented in the current implementation.

7.4 Low-level iSCSI Driver Design

This section presents a more detailed view of the functionality of the STML using the functions that have been defined above. The interaction between various code pieces and how they interact with each is dealt with.

7.4.1 Processing a WRITE SCSI Command

After the Login Phase is complete, the SML passes SCSI Commands to the iSCSI LLD through `queuecommand()` (Steps 1 & 2 in Fig. 7.8) as defined in the `SCSI_Host_Template` implementation. The value of `sc_data_direction` helps determine if the SCSI Command is a READ or WRITE request. The `session` struct having the same target_id as the `Cmnd`'s target_id for the target is found in the `session` struct linked list. The `session` struct linked list is accessed through the `host->hostdata` struct which is one of the fields in `SCSI_Cmnd` struct. Memory for a new `command` struct is allocated and initialized with a unique initiator task tag. The iSCSI 'SCSI Command' header (opcode 0x01) is then filled. The `datasegmentlength` field in the iSCSI 'SCSI Command' header is set to the appropriate length by looking at the `DataPDULength`, `FirstBurstSize`, and `ImmediateData` fields in the `oper_param` struct for the WRITE Command. The CDB from the `Cmnd` struct is copied into the iSCSI SCSI Command header. The F_BIT in the iSCSI 'SCSI Command' header is set if no Unsolicited iSCSI Data PDUs is to be sent. This depends on the negotiated `InitialR2T` Operational Parameter value. The TCP `iovec` buffers are set to send the iSCSI 'SCSI Command PDU' that includes the header and Immediate Data, if present. The `tx_size` field in `command` struct is set to the iSCSI PDU Length to be sent. The `tx_thread` waiting on

tx_sem is woken up to send the set TCP iovec buffers. The new command struct created is added to the linked list of all the pending commands for the connection (Step 3 in Fig. 7.8).

As mentioned in Section 7.1, there is a unique tx_thread for every connection in a session. When the tx_thread is woken up, it searches through the linked list of command struct for the involved connection struct (Step 4 in Fig. 7.8). The tx_size > 0 field in the command struct determines that there is iSCSI 'SCSI Command' PDU to be sent to the target. The socksendmsg(), TCP routine, is called to send the iSCSI 'SCSI Command' PDU to the target (Step 5 in Fig. 7.8).

The send_subsequent_PDU() function is called by the tx_thread if the F_BIT in the iSCSI 'SCSI Command' PDU is not set (Step 6 in Fig.7.8), which means there is unsolicited data to be sent to the target. The send_subsequent_PDU() function will set up the iSCSI 'Data-Out' PDUs based on the FirstBurstSize and DataPDULength fields in the parameter_type struct. The F_BIT is set if this is the last iSCSI 'Data-Out' PDU to be sent to the target. It will set up the TCP buffers, iov and tx_msghdr fields in the command struct, for the tx_thread to send the iSCSI Data PDU (Step 7 in Fig. 7.8). The send_subsequent_PDU() function is called by the tx_thread till the data_offset field is less than the FirstBurstSize field in the oper_param struct. After the last Data-Out PDU (with F_BIT set) is sent to the target, the tx_thread waits on the tx_sem.

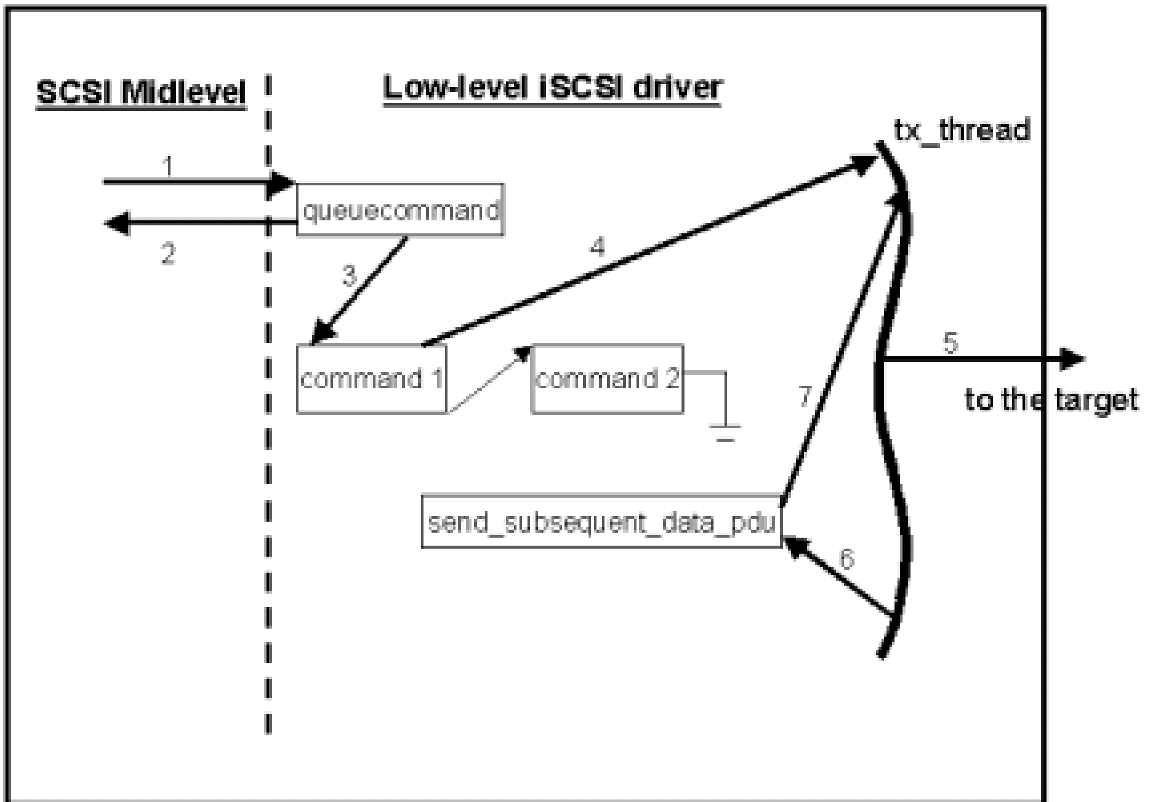


Fig. 7.8 Processing of SCSI Command.

If there is more WRITE data to be sent involved with the SCSI Command, the LLD expects iSCSI 'Ready To Transfer' (R2T) PDU from the target. The `rx_thread` receives the iSCSI 'R2T' PDU (Step 1 in Fig. 7.9) and then calls the `rx_r2t()` function (Step 2 in Fig.7.9). The `rx_r2t` function finds the relevant `command` struct from the linked list by comparing the initiator task tag of the received iSCSI 'R2T' PDU with that of the `command` struct (Step 3 in Fig. 7.9). It will set up the iSCSI 'Data-Out' PDU based on the `DataPDULength` field in operational parameters. The F_BIT is set if this is the last iSCSI 'Data-Out' PDU to be sent in response to the iSCSI R2T PDU. The function will then update the `data_offset, r2t_xfer_length` fields in the `command` struct that is used by the `setup_subsequent_PDU()` function. The `rx_r2t()` function will set up the TCP buffers, iov `and tx_msghdr` fields in `command` struct, for the `tx_thread` to send the iSCSI Data PDU. The `tx_thread` is then woken up from the `tx_sem`.

The `tx_thread,` after waking up, will search the `command` struct linked list (Step 5 in Fig. 7.9) and send the iSCSI 'Data_Out' PDU for the command whose field `tx_size >0` (Step 6 in Fig.7.9). The `tx_thread` will call `setup_subsequent_PDU()` function for setting up the subsequent iSCSI 'Data-Out' PDUs to be sent to the target (Step 7 in Fig.7.9). The `setup_subsequent_PDU()` function will update the `data_offset` and `r2t_xfer_length` fields in the `command` struct. The `send_subsequent_PDU()` function is called by the `tx_thread` till the `r2t_xfer_length` field is greater than 0. After the last iSCSI 'Data-Out' PDU (with F_BIT set) is sent to the target, the `tx_thread` waits on the `tx_sem`.

After sending the WRITE data to the target, the LLD expects the iSCSI 'SCSI Response' PDU from the target. The `rx_thread` receives the iSCSI 'SCSI Response' PDU (Step 1 in Fig. 7.10) and then calls the `rx_rsp()` function (Step 2 in Fig. 7.10). The `rx_thread` finds the relevant `command` struct from the linked list. The `rx_rsp()` function will get the global `io_request` lock for mutual exclusion from the `abort()` and `queuecommand()` functions for accessing the `command` struct linked list. The `done()` function, implemented by the SCSI Mid-level, is called (Step 3 and 4 in Fig. 7.10) and the relevant `command` struct is removed from the `command` struct linked list (Step 5 in Fig. 7.10). The processing of this WRITE SCSI Command is finished.

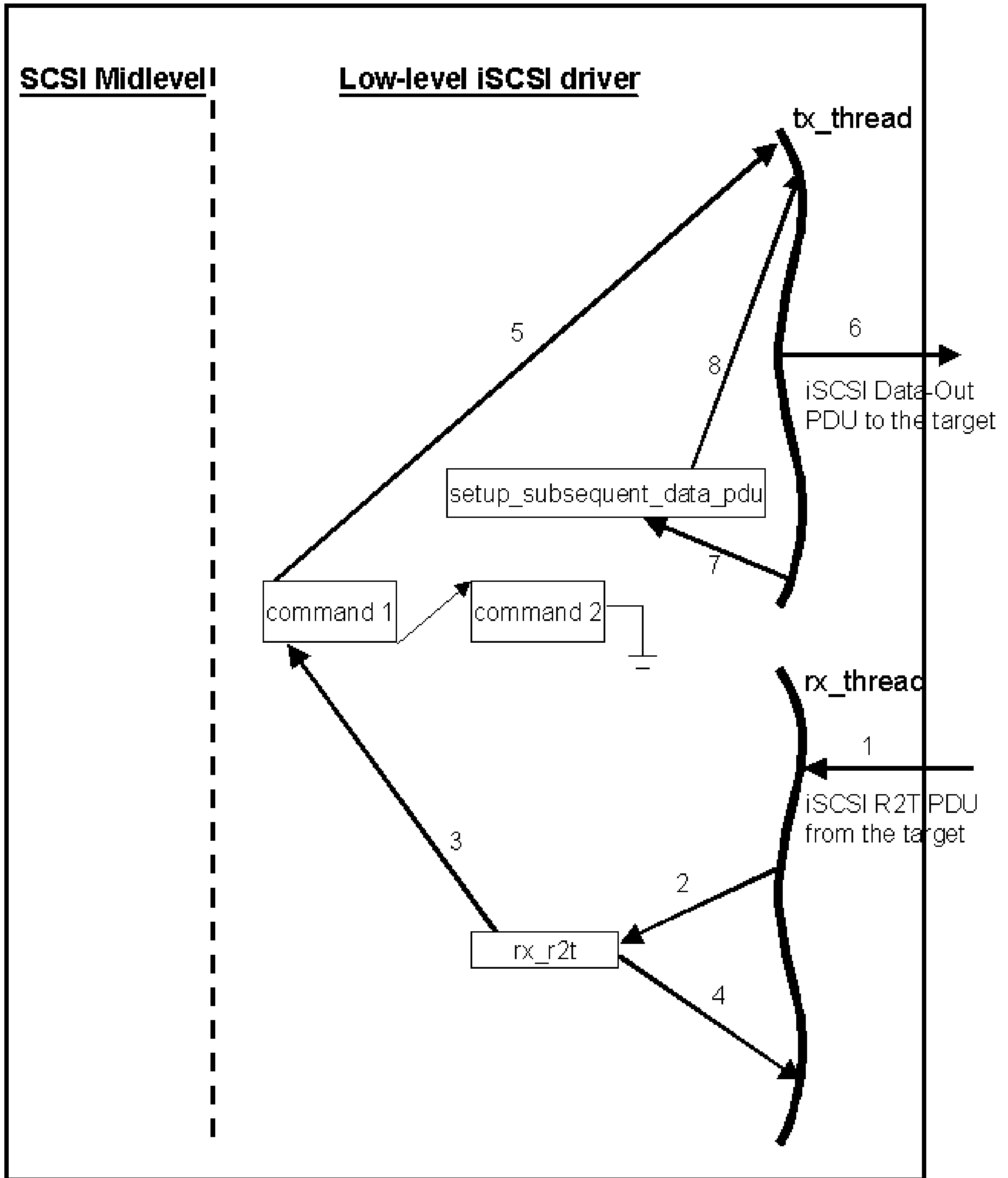


Fig. 7.9 Data Processing for WRITE Command.

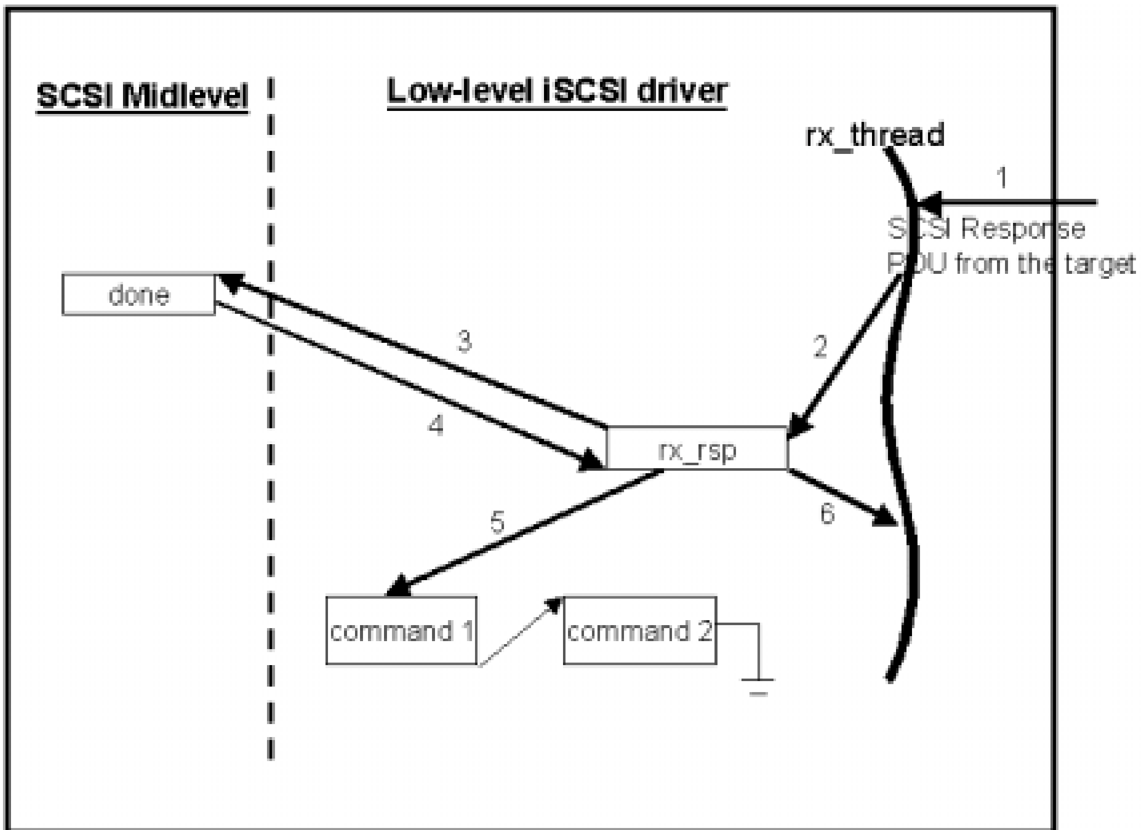


Fig. 7.10 Culmination of SCSI Command.

7.4.2 Processing a READ SCSI Command

The SCSI Command Processing is the same as described in Section 7.4.1. After the iSCSI 'SCSI Command' PDU has been sent to the target, the iSCSI LLD expects iSCSI 'Data-In' PDUs from the target for this command. The `rx_thread` receives the iSCSI 'Data_In' PDU (Step 1 in Fig. 7.11) and then calls the `rx_data()` function (Step 2 in Fig. 7.11). The `rx_data` function finds the relevant `command` struct from the linked list by comparing the initiator task tag of the received iSCSI 'Data-In' PDU with that of the `command` struct (Step 3 in Fig.7.11). The payload of the iSCSI 'Data-In' PDU is copied to the SCSI Mid-level data buffers using the `data_offset` and `datasegmentlength` fields in the received iSCSI 'Data-In' header. As the iSCSI 'Data-In' PDUs are received, the `data_offset` field is updated in the `command` struct to keep track of the amount of data read from the target. The `rx_thread` receives all the subsequent iSCSI 'Data-In' PDUs from the target and the last PDU is expected to have the F_BIT set.

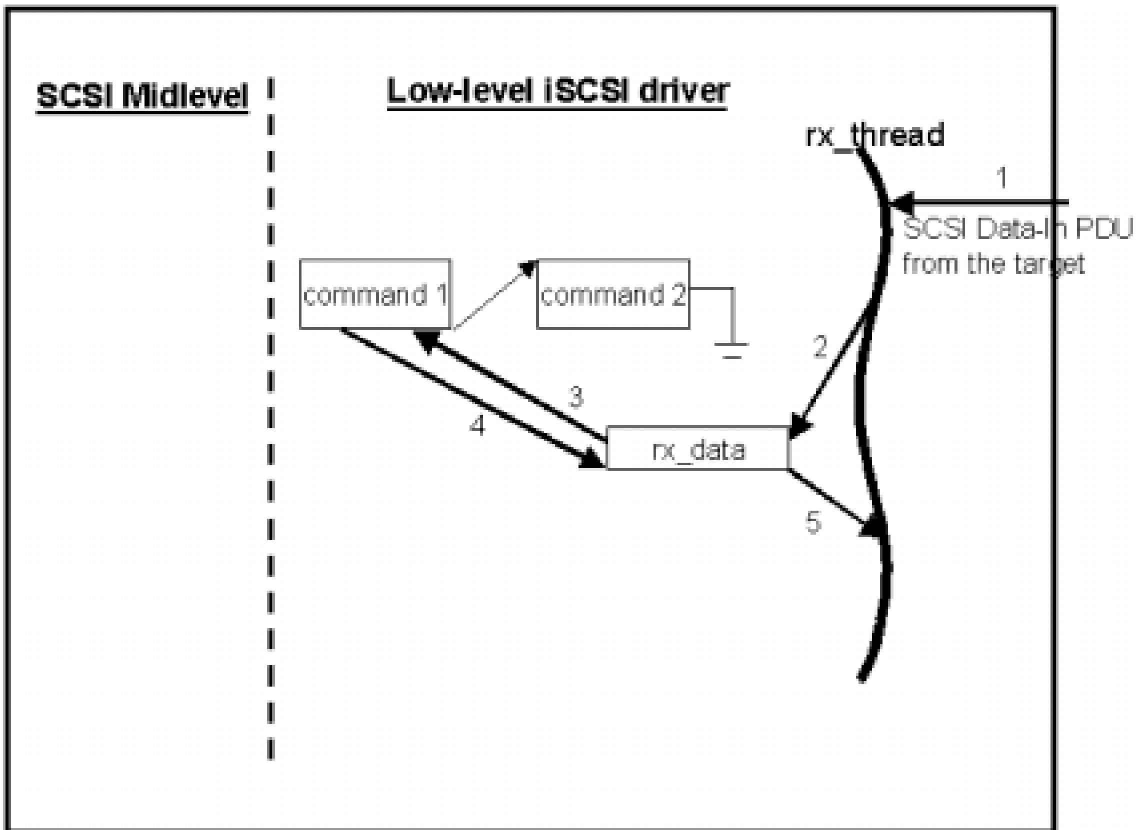


Fig. 7.11 Data Processing for READ Command.

After receiving the READ data from the target, the LLD expects an iSCSI 'SCSI Response' PDU from the target. The `rx_thread` receives the iSCSI 'SCSI Response' PDU and then calls the `rx_rsp()` function. The iSCSI 'SCSI Response' PDU processing is the same as for WRITE SCSI Command explained in the previous section (Fig. 7.10).

7.4.3 Processing an ABORT Command

The SML can abort any SCSI Command passed earlier to the iSCSI LLD through the `abort`() function (Step 1 in Fig. 7.12) as defined in the `SCSI_Host_Template` implementation. The `session` struct having the same target_id as the `Cmnd`'s target_id for the target found in the linked list of `session` struct. The `command` struct linked list is searched to find out the initiator task tag of the command that has to be aborted. If the command to be aborted is found, then the `state` field of the command struct is checked to see if the command has been sent to the target or not. There are two possibilities for the command state:

- COMMAND_NOT_SENT: The `command` struct entry for the command to be aborted is freed from the linked list, and the `abort`() function returns (Step 10 in Fig. 7.12).
- COMMAND_SENT: A new `command struct` is allocated and initialized with a unique initiator task tag. The iSCSI 'Task Management Command' PDU (opcode 0x02) is filled in and the TCP `iovec` buffers are set to send the iSCSI PDU. The `tx_thread` waiting on tx_sem is woken up to send the set TCP `iovec` buffers (Step 3

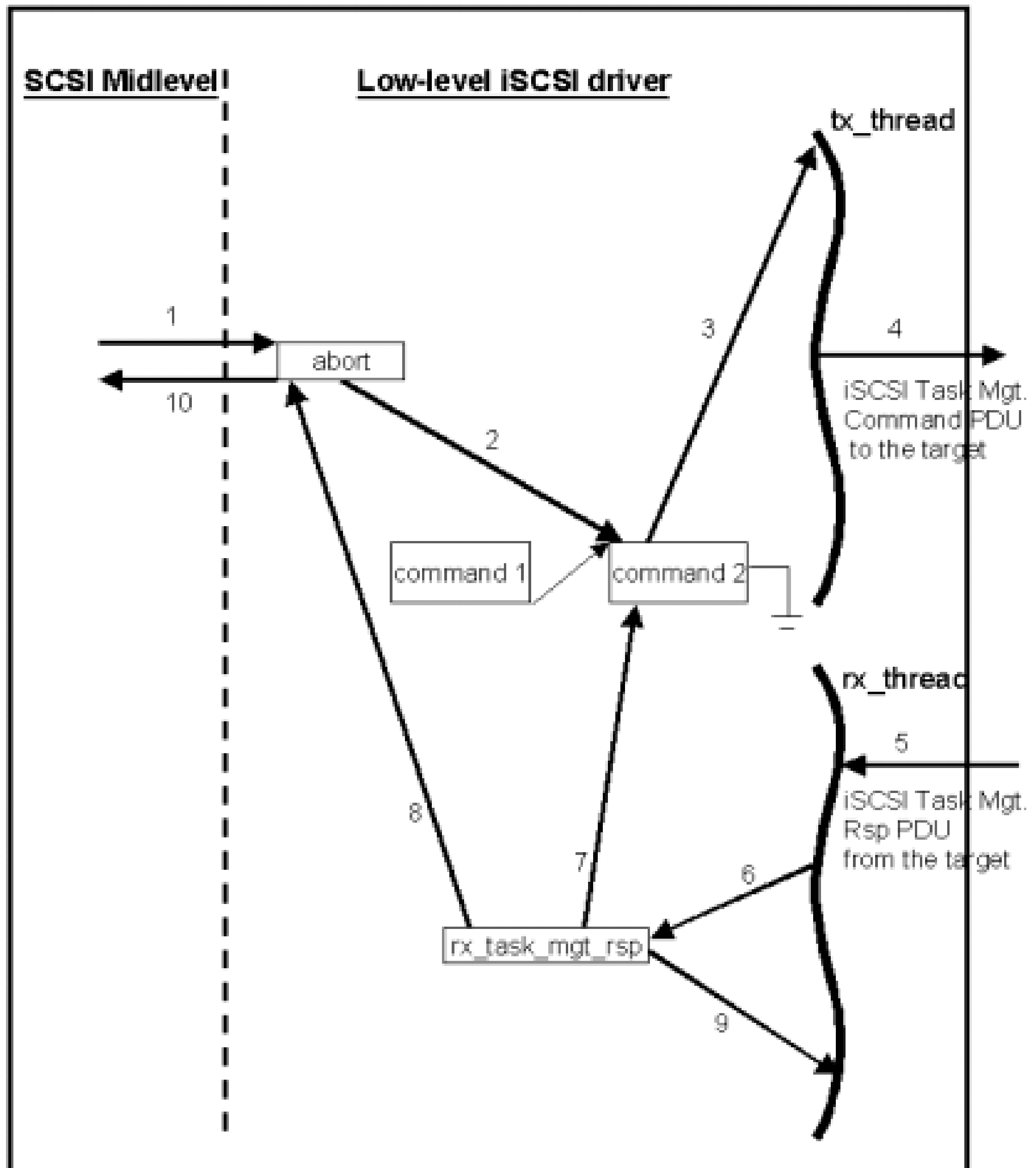


Fig. 7.12 Processing an ABORT Command.

- & 4 in Fig.7.12). After waking up the `tx_thread`, the `abort()` function waits on a `tx_sem` semaphore for a finite time. If a iSCSI 'Task Management Response' PDU (opcode 0x22) is received (through the `rx_thread`) within that finite time (Step 5 in Fig. 7.12), the `rx_task_mgt_rsp()` function processes the received iSCSI PDU and wakes up the `abort()` function waiting on the `tx_sem` semaphore (Step 7 & 8 in Fig. 7.12). The two `command` struct entries (one for abort command and the other for command to be aborted) are freed before the `abort()` function returns (Step 10 in Fig. 7.12). The `Response` field from the received iSCSI 'Task Management Response' PDU decides the return value for the `abort()` function.

Chapter 8

Performance Analysis

This chapter describes the Test Set-Up Details for doing performance analysis. The Performance Metrics (variables to quantify performance) and Variables (parameters that affect performance) are discussed next. The Performance Analysis for SEP and iSCSI Protocol is discussed in the final two subsections.

8.1 Test Set-Up

The test set-up involves two computer systems: an initiator and a target. The two computer systems are high speed PCs running the Linux operating system. The initiator and target emulator code are loaded as kernel modules. The details for initiator and target modules installation are explained in Appendix B.

8.1.1 CPUs

- Initiator System
  Intel Pentium III
  455 MHz Processor.
  128 Mbytes RAM.
- Target System
  Intel Pentium III
  667 MHz Processor.
  128 Mbytes RAM

8.1.2 Ethernet Technologies

The tests were run on Fast Ethernet and Gigabit Ethernet as two Link Layer Technologies. The following are the NICs that are used for each technology.

- Fast Ethernet
  Card: 3 Com Vortex
  Driver: `linux/drivers/net/3c59x.c`
  V1.102.2.38H 9/02/00 Donald Becker and others
- Gigabit Ethernet
  Card: 3 Com Acenic
  Driver: `linux/drivers/net/acenic.c`
  V0.33a 08/16/99 Jes Sorensen

8.1.3 Fiber_Channel Technologies

The Fiber Channel driver for the Host Bus Adapter (HBA) is used to access a Fiber Channel on the target emulator system. The details of the Fiber Channel driver are as follows:

Card: Qlogic Corporation ISP2200 A
Driver: `linux/drivers/net/qlogicfc.c`

8.1.4 Version of Linux Operating System

The Linux kernel version used is 2.4.0-test9

8.2 Performance Metrics

The Performance metrics give a measure of how well or how poorly a system is behaving. The common performance Metrics are as follows:

<u>Bandwidth</u>
The bandwidth measurement gives the data transfer rate between the initiator and the target during Disk I/O operation. This metric is measured for the SEP and iSCSI Protocol, as discussed in Section 8.6 and 8.7.

<u>CPU utilization</u>
CPU utilization is an important parameter that should be monitored on the initiator and the target side during data transfer. The measurement unit gives the % CPU used by different routines in the Operating System while performing Disk I/O operation. This metric is measured for the SEP and iSCSI Protocol, as discussed in Section 8.6 and 8.7.

<u>Latency</u>
The latency measurement gives the delay (in secs) to perform any request given by the initiator. In other words, it gives the time required to perform any I/O operation requested by the initiator. This metric is not measured in this thesis.

For each of the performance parameters, measurements produce a distribution, a minimum, a maximum, an average, a standard deviation and confidence intervals.

8.3 Performance Variables

The performance variables are the parameters that affect the performance metrics. This section lists the performance parameters and elaborates on the effect of each parameter on the performance metrics.

Target Domain
Ashish Palekar [9], in an effort to develop a target emulator for linux platforms, has implemented versions of the SEP front-end for the user space and the kernel space. The kernel space and the user space target can affect all the performance metrics considered. This test cannot be performed on iSCSI Target Emulator as there is no user space implementation.

Block size
The block size is the portion, or sector, of a disk that stores a group of bytes that must all be read or written together. The target specifies the block size for data storage at the time of inquiry by the initiator. Disk I/O rate will be affected by the number of blocks transferred per unit time which in turn depends on the block size. The block size in the target emulator, which is used in testing the SEP/iSCSI Implementations, is a constant and can be changed at compile time.

Scatter-gather list size on Initiator
The scatter-gather list, as explained in Chapter 5, is an array of data buffer pointers passed on by the SCSI Mid-level to the low-level SCSI disk drivers/session layer protocols. The size of the list is controlled by the low-level driver's session layer protocols and is a variable that can be changed to see performance variation.

Link Layer Protocol
The underlying Link Layer technology used is Ethernet. The performance tests can be performed on the different Ethernet interfaces: Fast Ethernet (100 Mbps) and Gigabit Ethernet (1 Gbps). The performance metrics should be affected by the link layer interconnect used.

Ethernet Driver Tuning parameters
Coalescing interrupts: The Ethernet driver generates interrupts during packet transmission using a manufacturer-specific strategy. The Alteon Acenic Gigabit Ethernet driver uses one of the following two criteria to generate interrupts: when the rings are close to getting full; or after a fixed time interval. The default criteria for the Acenic driver has been set to generate interrupts after a fixed time interval, which is defined by the constants DEF_TX_COAL for transmitting data and DEF_RX_COAL for receiving data. Coalescing Interrupts produced by the Ethernet driver will affect performance, as interrupt servicing involves a lot of CPU overhead. It is a tuning parameter to test for performance.

Ethernet Packet size: The standard packet size for Ethernet, Fast Ethernet, and Gigabit Ethernet is 1500 bytes. However, there is an option of increasing the packet size to 9000 bytes when the Gigabit Ethernet interface is brought up on Linux O/S. The Ethernet packet size is changed to analyze the effect on Performance metrics.

I/O on Target Side
The following possibilities are considered for I/O on the target side:

- I/O to and from memory directly – this will performance-test the session layer protocol (SEP or iSCSI) under test. This is the target mode used for most of the performance tests in this Chapter.
- I/O to and from a file – this will test the performance of SCSI over the interconnect in question.
- I/O to and from a disk without going through the file system– this will test the performance of the protocol in a real world system.

Queuing Length of commands to low-level driver on Initiator

The queuing length of commands to the low-level driver is specified by the field can_queue in the `Scsi_Host_Template` struct. This value can be changed to observe the effect of queuing length on the performance metrics. This performance test is valid for the iSCSI initiator only, as it has a multi-threaded design which can support multiple commands at a given time. The SEP initiator cannot support multiple commands at a time.

Size of the Data PDU in the Session Protocol

When data is exchanged between the Initiator and the Target for the SEP and iSCSI Protocols, the actual READ/WRITE Data is transferred in SEP/iSCSI Data PDUs. The iSCSI Protocol provides an option of changing the Maximum Data PDU Length for any I/O operation. A change in Maximum Data PDU Length for the iSCSI Protocol can affect the Performance Metrics, as discussed in Section 8.2. This variable cannot be used to test the SEP protocol because SEP does not provide any option to change the Maximum Data PDU Length.

8.4 Accuracy of Data

8.4.1 Side Effects of Adding Probes

The probe recording routines mentioned in Chapter 4 have been optimized so that they require minimum time for recording the probes. However, it does take a few dozen cycles to record the probe data. It was observed that a probe requires an average of 80 cycles for recording data. This makes an overhead of 160 cycles for every function being probed because two probes, one at entry and the other at exit, are required to find out the number of cycles utilized.

8.4.2 Confidence Level of Data

It is necessary to have a small confidence interval to have value in the observations made. In order to calculate the confidence interval, each SEP and iSCSI test was run 10 times [14]. The average number of cycles required by READ/WRITE requests were noted and the confidence interval for a 95 % confidence level were calculated for each function. The following formulae were used.

Standard Deviation (σ) = $$\frac{\sqrt{n\sum x^2 - (\sum x)^2}}{n(n-1)}$$ Equation 8.1

where n = number of samples (10 in our case), x = value of sample

$$\text{Confidence Interval} = 1.96 * \frac{\sigma}{\sqrt{n}} \qquad \text{Equation 8.2}$$

Error Margin = 100 *(Confidence Interval / Average) Equation 8.3

The constant 1.96 in Equation 8.2 is fixed for a confidence level of 95 % which can be looked up in a mathematical table called the normal distribution table.

8.5 Ethernet Payload

It is known that the maximum payload of an Ethernet packet is 1500 bytes, i.e., it can carry a maximum of 1500 bytes of data. Out of those 1500 bytes, if the TCP/IP stack is involved, 52 bytes are used by the TCP (32 bytes) and IP (20 bytes) headers. Effectively, the packet contains only 1448 bytes of actual data. Fig. 8.1 illustrates a typical Ethernet frame.

Preamble (8 bytes)	Frame Header (14 bytes)	IP Header (20 bytes)	TCP Header (32 bytes)	User Data (0-1448 bytes)	Checksum (4 bytes)

Fig. 8.1 Ethernet Frame for a TCP.

Standard TCP uses a fixed 20 byte header which would allow for 1460 bytes of actual data. However, running a trace on SEP and iSCSI tests, it was observed that the SEP or iSCSI header started at the 53rd byte in the Ethernet payload. The TCP header, found in Linux, uses an extra 12 bytes to convey the timestamp option.

Kind (8 bytes)	Length (10 bytes)	TS value (TSval)	TS Echo Reply (Tsecr)

Fig. 8.2 TCP TimestampOption.

The timestamp option (Fig. 8.2) is used to measure the roundtrip time (RTT) and also for protection against wrapped sequence numbers (PAWS). Accurate and current RTT estimates are necessary to adapt to changing traffic conditions and instabilities in a busy network. The timestamp option carries two four-byte timestamp fields. The PAWS mechanism uses timestamp values to reject old duplicate segments that can corrupt an open TCP connection. The TimeStamp value field (TSval) contains the current value of the timestamp clock of the TCP stack sending the option. The Timestamp Echo Reply field (Tsecr) is only valid if the ACK bit is set in the TCP header. If it is valid, the timestamp value was sent by the remote TCP in the TSval field of a Timestamp option. If it is invalid, its value must be zero.

8.6 Performance Results for SEP

The performance tests are performed for WRITE requests between the initiator and the target. The WRITE operation requested by the user application on the initiator are raw, which means that the filesystem on the initiator system is bypassed. This is done to avoid

buffering, caching effect and possible asynchronous I/O operation due to filesystem presence on the initiator system. In order to measure the speed of the transport network between the initiator and the target, the data is written/read out of memory on the target side. This section discusses the performance results for the SEP implementation.

8.6.1 Effect of Target Domain on Bandwidth for SEP

In Fig. 8.3, the WRITE Request size plotted on the X-axis represents the amount of WRITE Data (in MB) requested by a user application on the initiator to write from the initiator to the target emulator. The WRITE Data Requests range from 10 MB to 1500 MB. For each WRITE Request size, the Data Rate (in MB/s) recorded is plotted on the Y-axis. Each bandwidth value plotted on the Y-axis is the average value for 10 sample runs.

The target domain does not significantly affect the bandwidth (~21 MB/s) when doing WRITE operation to the target. It might be expected that the User Mode Target Emulator should be slower than the Kernel Mode Target Emulator, as it involves extra switching from kernel space to user space during I/O operation. However, the indifference to bandwidth in switching domains means that the switching overhead is too small compared to other contributions.

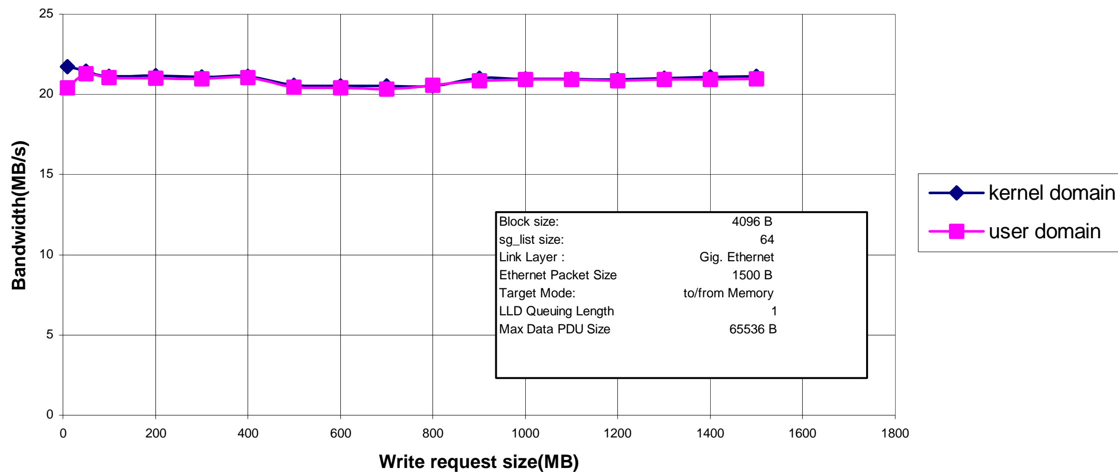


Fig. 8.3 Effect of Target Domain on Bandwidth for SEP.

The above chart is obtained when the Nagle Algorithm [15] is turned OFF on both the initiator and the target emulator. The Nagle Algorithm has the following set of rules to decide when to send data:

- If a packet is equal to or larger than the segment size (or MTU), and the TCP window is not full, send an MTU size buffer immediately.
- If the interface is idle, or the TCP_NODELAY flag is set, and the TCP window is not full, send the buffer immediately.

- If there is less than 1/2 of the TCP window in outstanding data, send the buffer immediately.
- If the amount to be sent is less than a segment size buffer, and if more than 1/2 the window is outstanding, and if TCP_NODELAY is not set, wait up to 200 msec for more data before sending the buffer.

When the Nagle Algorithm is turned ON, the bandwidth for the above operation is observed to be only 3 MB/s. The delivery of small iSCSI PDUs is delayed because TCP will wait for the window to be at least half-full. The PDUs are delivered only after a 200 msec time-out. This delay in delivery of small iSCSI PDUs is avoided by turning OFF the Nagle Algorithm when doing I/O operation. Hence, the bandwidth for the discussed WRITE operation increased from 3 to 21 MB/s.

Statistics on Bandwidth with Target in Kernel Mode

The Standard Deviation, Confidence Interval and Error Margin values are calculated using Equation 8.1, 8.2 and 8.3 respectively, for Target in Kernel Mode (shown in Fig. 8.3). Table 8.1 lists the statistical values for different WRITE requests (from 10 to 1500 MB). 10 sample runs are used to perform calculation for each WRITE request. It is observed that the Standard Deviation, Confidence Interval and Error Margin values are the highest for 10 MB WRITE request.

Write Request Size (MB)	Average Bandwidth (MB/s)	Standard Deviation (σ)	Confidence Interval (C.I.)	Error Margin
10	20.884	0.173	0.108	0.516
50	21.441	0.004	0.003	0.011
100	21.191	0.011	0.007	0.033
200	21.281	0.007	0.005	0.022
300	21.203	0.008	0.005	0.023
400	21.222	0.011	0.007	0.033
500	21.086	0.028	0.017	0.083
600	20.981	0.037	0.023	0.110
700	21.029	0.028	0.018	0.084
800	21.126	0.024	0.015	0.070
900	21.071	0.022	0.014	0.064
1000	21.068	0.024	0.015	0.070
1100	21.140	0.001	0.006	0.029
1200	21.137	0.010	0.006	0.003
1300	21.119	0.015	0.009	0.044
1400	20.856	0.105	0.065	0.313
1500	20.928	0.071	0.044	0.212

Table 8.1 Statistics on Bandwidth with Target in Kernel Mode.

CPU Utilization for Initiator system with Target in Kernel Mode

The FKT probes discussed in Chapter 4 are used to find CPU utilization on the initiator while recording bandwidth for the above experiment. The `fkt_print` program produces the following analysis on the trace data recorded by the `fkt_record` program. It precisely attributes every cycle that elapsed during the trace to a specific system call, IRQ, kernel function or user-level process. The analysis for the SEP initiator system is shown in Table 8.2.

Table 8.2 gives the name of the various IRQs, system calls and routines that are called during WRITE Data test explained in Section 8.6.1. The Performance Parameter values corresponding to Table 8.2 are as follows:

Target Domain:	Kernel
Block Size:	4096 B
sg_list size:	64
Link Layer:	Gig. Ethernet
Ethernet MTU Size:	1500 B
Coal. Interrupt Interval:	400 clock ticks
Target Mode:	to/from Memory
LLD queuing length	1
Max PDU size	65536 B

The column ‘code’ gives the probe identification code for the probes present in those routines. The ‘Cycles’ column lists the total number of cycles that are spent in each routine. ‘Count’ gives the number of times the routines are called. The average number of cycles (Cycles/Count) that are spent in each routine are shown in the column ‘Average’. Finally, the last column shows the percentage of time spent in each routine.

Table 8.2 indicates that most of the cycles are spent in the idle process (62.44 %). The idle process time can be attributed to the time spent in NIC processing [15] (on both initiator and target emulator [9]) and in I/O request processing by the Operating System in the Target Emulator. The Target Emulator, when processing the I/O request from the initiator, involves the network stack including TCP, IP, and Gigabit Ethernet driver in the Linux Operating System. The other components involved are SCSI Target Mid-Level (STML) and iSCSI Front-End Target Driver (FETD).

The other major component that contributes to CPU utilization is the `sep_thread` (29.31%). The `sep_thread` code involves calls to TCP, IP, and Gigabit Ethernet Driver routines on the initiator before the WRITE request is passed to the Gigabit Ethernet Firmware.

```
*******************************************************************************
                         Name  Code       Cycles   Count     Average   Percent
*******************************************************************************
sys_call            sys_write  0004      3633201       9   403689.00     1.77%
sys_call            sys_times  002b         1130       1     1130.00     0.00%
sys_call           sys_select  008e         9185       1     9185.00     0.00%
     IRQ                timer     0       443360      45     9852.44     0.22%
     IRQ             keyboard     1         9851       1     9851.00     0.00%
     IRQ AceNIC Gigabit Ethernet  10    12778812    1534     8330.39     6.23%
     IRQ                 ide0    14        12746       1    12746.00     0.01%
   other        other process   319        11663                         0.01%
    idle         idle process     0    128049490                        62.44%
   other           sep_thread   866     60103164                        29.31%
    user         user process   870        20139                         0.01%
   Total         Total cycles          205072741                       100.00%
```

Table 8.2 Analysis Table produced by fkt_print for SEP Low-Level driver.

The `sys_write()` system call takes 1.77 % of the total time. This is the time taken by the SCSI Upper-Level and Mid-level routines to process the WRITE request. It is comparitively less than the time taken by the `sep_thread` (29.31%).

Processing of the Gigabit Ethernet card interrupts takes 6.23 % and other processes running on the initiator take 0.25 % of the time during WRITE operation.

The total % CPU utilization on the initiator system is calculated from the FKT Analysis:
= 100 – (idle process % CPU utilization) Equation 8.4
= 100 – 62.44%
= 37.56 %

CPU utilization comparison
The total % CPU utilization on the initiator system is compared for Target Domain change using Equation 8.4:

Target Domain	Total % CPU utilization
Kernel	37.56
User	37.53

Table 8.3 % CPU utilization comparison for Target Domain change.

As shown in Table 8.3, the % CPU utilization on the initiator system is not affected by the Target Domain change.

8.6.2 Effect of Target Block Size on Bandwidth for SEP

The bandwidth increases from 18.5 MB/s for 512 KB block size to 21 MB/s for 4096 KB block size (Fig. 8.4). Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs. The improvement in bandwidth can be attributed to the fact that the block size increment increases the number of bytes per I/O request passed to the SEP Low-

Level Driver (LLD). This decreases the number of I/O requests that SCSI Mid-Level (SML) has to pass to the SEP LLD. It can be inferred that the number of I/O requests overhead is a major contributor to the cost.

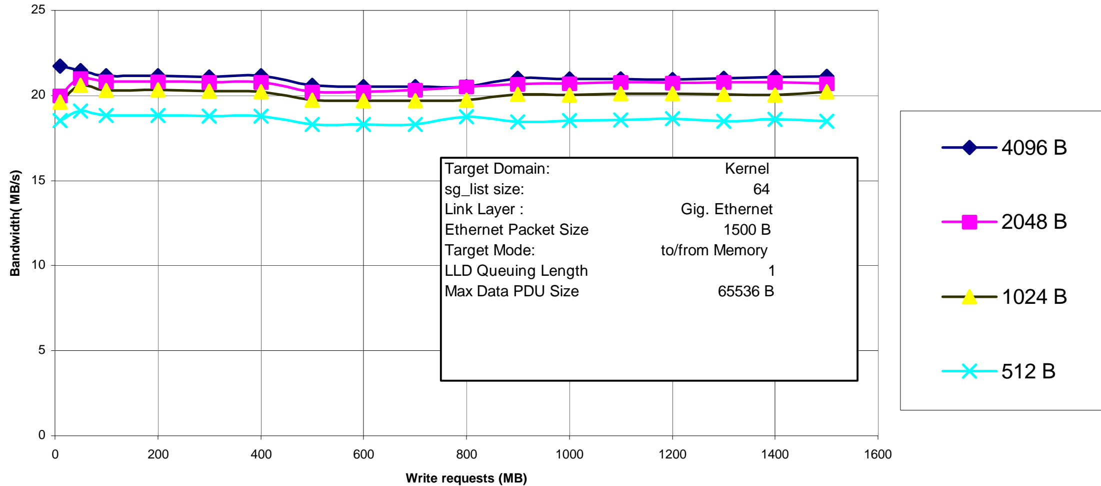


Fig. 8.4 Effect of Target Block Size on Bandwidth for SEP.

An interesting observation made is that the scatter-gather list entry size is equal to the block size of the data that is used for storage on the target. This implies that the SCSI Mid-level is using the Target block size information in generating scatter-gather list entries.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for Target Block Size change using Equation 8.4:

Target Block Size	Total % CPU utilization
512 B	44.62
1024 B	40.61
2048 B	39.03
4096 B	37.56

Table 8.4 % CPU utilization comparison for Target Block Size change.

As shown in Table 8.4, the % CPU utilization on the initiator system decreases with increase in Target Block size. The initiator allocates the SCSI scatter-gather buffers with each buffer size same as the Target Block Size. The initiator has to allocate more buffers if the Target Block Size is small (for writing constant amount of data to the target). This, in turn, utilizes more CPU cycles on the initiator. It is due to this reason that the % CPU utilization decreases from 44.62 % to 37.56 % when the Target Block Size increases from 512 B to 4096 B.

8.6.3 Effect of Initiator Scatter Gather List Size on Bandwidth for SEP

The Initiator Scatter-Gather (sg) list size change does not affect the bandwidth when increasing the scatter-gather list size from 16 to 64 on the initiator (Fig. 8.5). Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs.

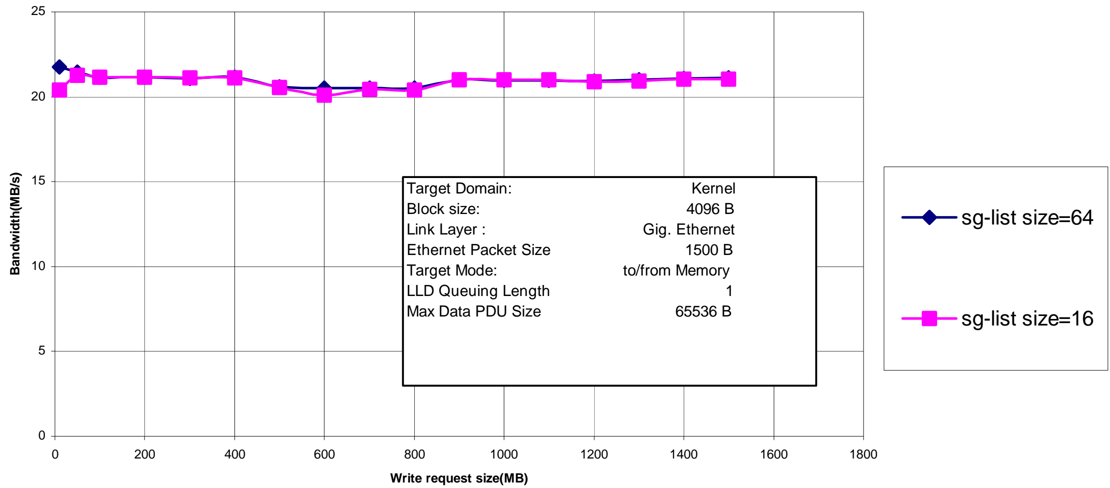


Fig. 8.5 Effect of Initiator Scatter-Gather List Size on Bandwidth for SEP.

The constant `sg_tablesize`, a field defined in the `Scsi_Host_Template` struct (in file iscsi_initiator.h), denotes the sg list size. The indifference to observed bandwidth by a sg list size change suggests that the bottleneck is in some other involved component, maybe in the network stack on the initiator and the target system.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for Scatter-gather list Size change on the initiator using Equation 8.4:

Scatter-Gather List Size	Total % CPU utilization
16	37.39
64	37.56

Table 8.5 % CPU utilization comparison for Scatter-gather list size change.

As shown in Table 8.5, the % CPU utilization on the initiator system does not change with Scatter-gather list change on the initiator.

8.6.4 Effect of Ethernet Link Speed on Bandwidth for SEP

As shown in Fig. 8.6, on a Fast Ethernet Link, the bandwidth for a WRITE operation is 10.2 MB/s, using 86 % of the maximum bandwidth possible at 11.9 MB/s. On a Gigabit Ethernet link, the bandwidth for a WRITE operation is 21 MB/s, using only 19 % of the maximum bandwidth possible at 119 MB/s. Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs.

It can be concluded that for I/O operations, the Link Speed in Fast Ethernet Connection is a limiting factor, because increasing the link speed results in an absolute increase in bandwidth. However, this is not the case with Gigabit Ethernet link - the bottleneck in this case must be some other involved components, since the absolute increase in available bandwidth is actually accompanied by a significant decrease in link speed utilization.

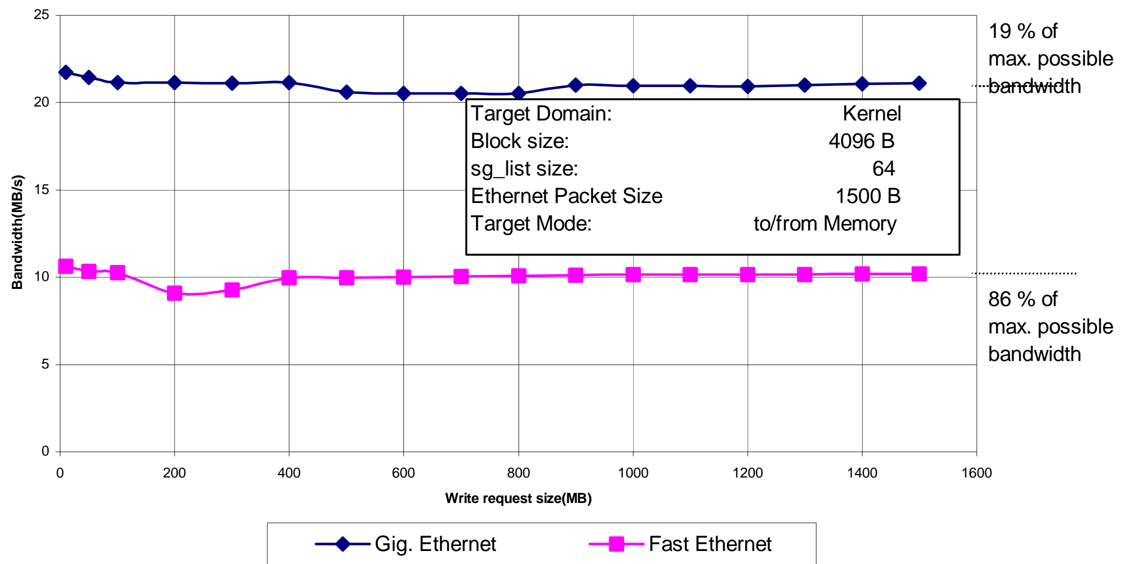


Fig. 8.6 Effect of Ethernet Link Speed on Bandwidth for SEP.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for Ethernet Link Speed change using Equation 8.4:

Ethernet Technology	Total % CPU utilization
Fast (100 Mbps)	27.77
Gigabit (1000 Mbps)	37.56

Table 8.6 % CPU utilization comparison for Ethernet Link Speed Change.

As shown in Table 8.6, the % CPU utilization on the initiator system for Fast Ethernet is comparatively less than that for Gigabit Ethernet. The reason for this can be that the available bandwidth on Fast Ethernet link is utilized completely (86 % of maximum theoretical bandwidth), so the initiator CPU is idle for more time compared to that for Gigabit Ethernet Link. The Link speed in Fast Ethernet is the bottle-neck in doing WRITE I/O operation.

8.6.5 Effect of Ethernet Packet Size on Bandwidth for SEP

As shown in Fig. 8.7, the Bandwidth increases from 21 to 30 MB/s when the Ethernet Packet size increases from 1500 to 9000 bytes on both the initiator and the target systems. Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs.

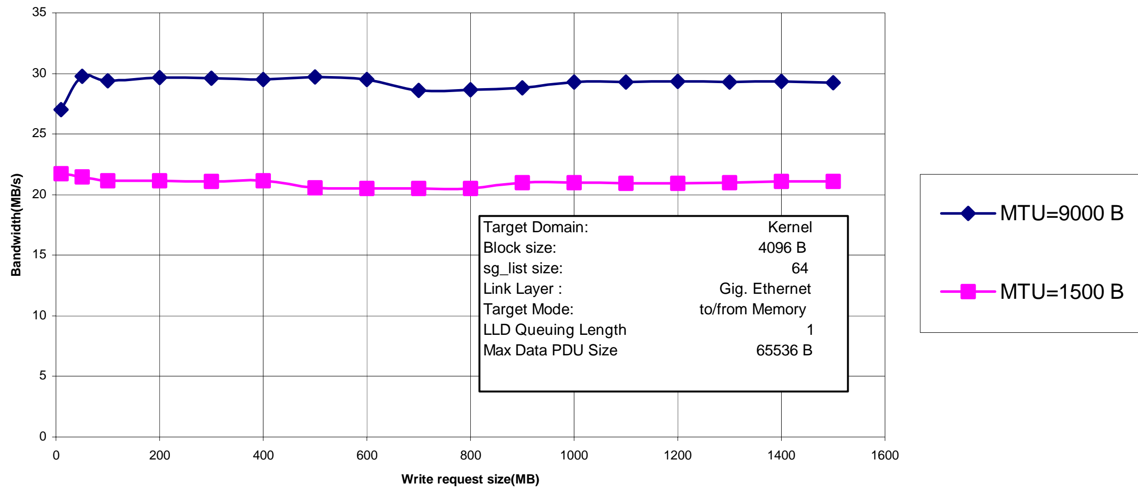


Fig. 8.7 Effect of Ethernet Packet Size on Bandwidth for SEP.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for Ethernet Packet Size change using Equation 8.4:

Ethernet Packet Size	Total % CPU utilization
1500 bytes	37.56
9000 bytes	35.67

Table 8.7 % CPU utilization comparison for Ethernet Packet Size change.

TCP allocates buffers whose maximum size is equivalent to the Ethernet Maximum Payload Size. The number of TCP buffer allocated, for a given amount of data, decreases when the Ethernet Packet size increases (from 1500 to 9000 bytes). The TCP processing on the initiator system is less for 9000 bytes TCP packets than for 1500 bytes TCP

packets, justifying the decrease in % CPU utilization when using 9000 byte Ethernet Frames.

8.6.6 Effect of Coalescing Interrupt Time Interval on Bandwidth for SEP

The default value for Coalescing Interrupt Time Interval (CITI) in the Alteon Acenic driver is 400 (constants DEF_TX_COAL and DEF_RX_COAL in acenic.c). The Bandwidth (~21 MB/s) does not change when the CITI is decreased from 400 to 30 clock ticks (for both the initiator and target system). Also, when the CITI is increased from 400 to 3000 clock ticks (for both the initiator and the target systems), the Bandwidth remains unchanged. (Fig. 8.8). The CITI constant value does not seem to have any effect on coalescing interrupts in the Firmware. Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs.

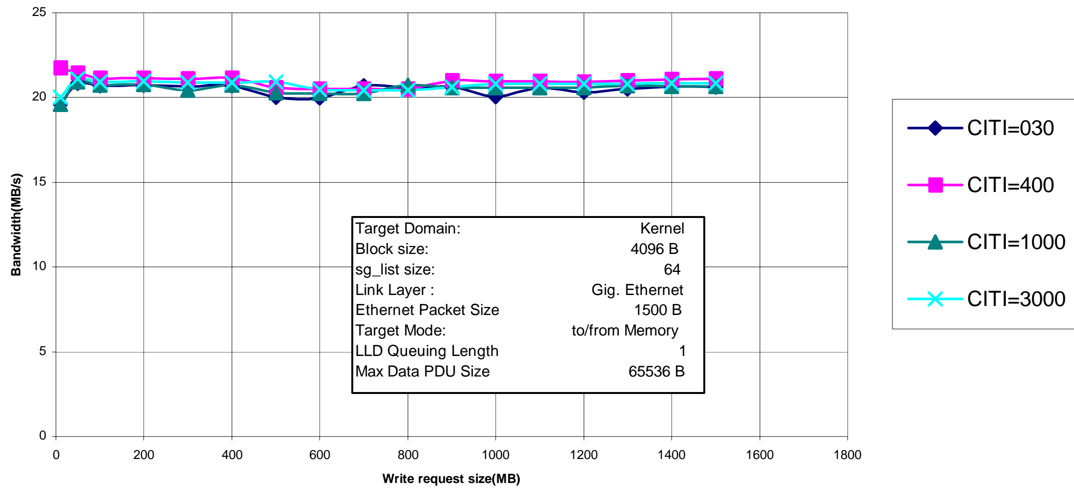


Fig. 8.8 Effect of CITI on Bandwidth for SEP.

<u>CPU utilization comparison</u>
The total % CPU utilization on the initiator system is compared for CITI change using Equation 8.4:

CITI	Total % CPU utilization
30	37.00
400	37.56
1000	36.79
3000	36.49

Table 8.8 % CPU utilization comparison for CITI change.

As shown in Table 8.8, the % CPU utilization on the initiator system does not change with CITI change on the initiator.

8.6.7 Effect of Old (Alteon) and New (3-Com) Acenic Cards on Performance

The results presented in Section 8.6 are for New (3 Com) Acenic Card used on the initiator and the target emulator (mentioned in Section 8.6.2). The performance analysis was done on an old Acenic Card (From Alteon) for SEP. The results from the old Acenic Card are compared with that for the new Acenic Card. The Performance Parameter values corresponding to Table 8.9 are as follows:

Target Domain:	Kernel
Block Size:	4096 B
sg_list size:	64
Link Layer:	Gig. Ethernet
Ethernet MTU Size:	1500 B
Coal. Interrupt Interval:	400 clock ticks
Target Mode:	to/from Memory
WRITE Request Size:	1700 MB
LLD queuing length	1
Max PDU size	65536 B

Performance Metric	New Acenic Gig. Ethernet NIC (3 Com)	Old Acenic Gig. Ethernet NIC (Alteon)
Bandwidth	21 MB/s	24 MB/s
sep_thread, % CPU util	29.31 %	33.12 %
Total, % CPU util.	37.56 %	41.63 %

Table 8.9 Effect of Old (Alteon) and New (3-Com) Acenic Cards on Performance.

As shown in Table 8.9, the Bandwidth attained with the old card (~24 MB/s) is higher than that for the new card (~21 MB/s). Also, the total % CPU utilization when using the old card (~41.63 %) is more than that for the new card (~37.56 %). The possible reason for this behavior can be that the NIC processing overhead in the Old Card might be less than that for the new card, thus resulting in higher bandwidth. The less NIC processing overhead in the old card also results in less % CPU utilization for the idle process (= 100 – 41.63 = 58.37 %) compared to that for the new card (=100 – 37.56 = 62.44 %). The NIC processing is included in the idle process for initiator system as shown in Table 8.2.

8.7 Performance Results for iSCSI

The performance tests are performed for WRITE requests between the initiator and the target. The WRITE operation requested by the user application on the initiator are raw, which means that the filesystem on the initiator system is bypassed. This is done to avoid buffering, caching effect and possible asynchronous I/O operation due to filesystem presence on the initiator system. In order to measure the speed of the transport network between the initiator and the target, the data is written/read out of memory on the target side. This section discusses the performance results for the iSCSI implementation. The tests performed are exactly the same as for the SEP implementation.

The performance parameters, analyzed for SEP, are also tested for iSCSI implementation. The effect of parameters on the Bandwidth during WRITE operation for SEP implementation is the same as that for iSCSI implementation. Also the Bandwidth attained by the iSCSI implementation is the same as that for SEP implementation. The results are summarized as follows:

Parameters actually affecting Bandwidth

- Block Size
- Ethernet Link Speed
- Ethernet Packet Size

Parameters not affecting Bandwidth

- Target Domain
- Scatter-Gather List Size
- Coalescing Interrupt Time Interval

The other performance parameters tested on the iSCSI implementation is discussed in Section 8.7.1 and 8.7.2.

8.7.1 Effect of LLD Queuing Length on Bandwidth for iSCSI

As shown in Fig. 8.9, increase in the queuing length for the iSCSI LLD from 1 to 8 does not affect the bandwidth for the WRITE operations. Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs. The queuing length increase increases the number of pending commands that the Low-level Driver can handle. However, the target emulator cannot process multiple commands in parallel. Due to this reason, the bandwidth remains unchanged when the queuing length of commands on the initiator is increased.

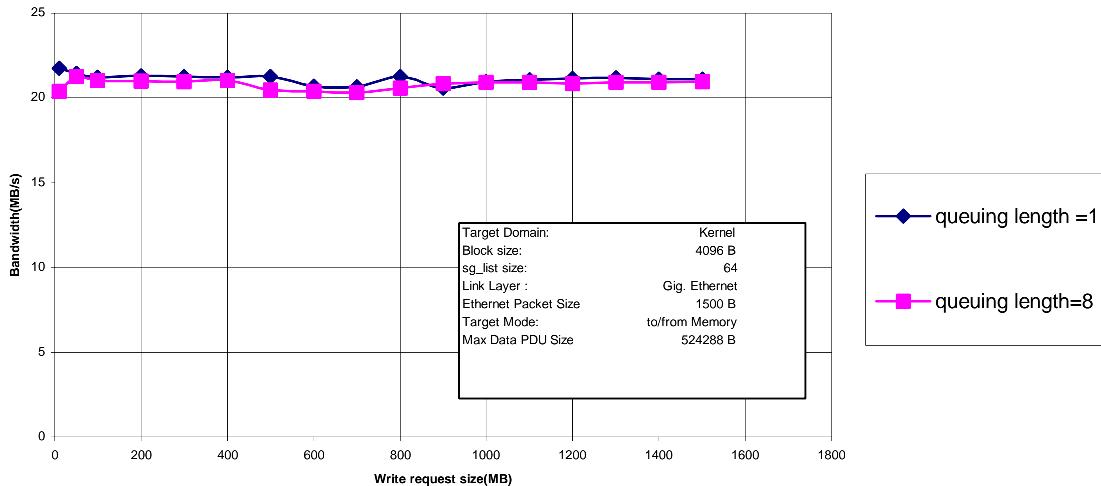


Fig. 8.9 Effect of LLD Queuing Length on Bandwidth for iSCSI.

CPU utilization for LLD Queuing Length=1

The FKT software probes are used to calculate the % CPU utilization on the initiator system. Table 8.10 gives the name of the various IRQs, system calls and routines that are called during WRITE Data test explained in Section 8.6.1. The Performance Parameter values corresponding to Table 8.10 are as follows:

Target Domain:	Kernel
Block size:	4096 B
sg_list size:	64
Link Layer :	Gig. Ethernet
Ethernet Packet Size	1500 B
Target Mode:	to/from Memory
LLD Queuing Length:	1
Max PDU Size:	524288 bytes

As shown in Table 8.10, the idle process takes 63.62 % of the total time to do the WRITE operation. The idle process % CPU utilization for iSCSI initiator is almost the same as for that SEP initiator (4.32 %).

The % CPU utilization for the `tx_thread` (24.35 %) is more than that for the `rx_thread` (4.32 %) because for a WRITE operation, data is transferred from the initiator to the target which involves call to the `tx_thread`. The `rx_thread` involvement during WRITE operation is very less.

Processing of the Gigabit Ethernet card interrupts takes 6.23 %. The `sys_write()` system call takes 1.85 % of the CPU. Other processes running on the initiator take 0.25 % of the time during WRITE operation. These % values are similar to that for the SEP initiator.

The total % CPU utilization on the initiator system is calculated to be = 100 – 63.62%
= 36.38 %

	Name	Code	Cycles	Count	Average	Percent
sys_call	sys_write	0004	3714345	9	412705.00	1.85%
sys_call	sys_times	002b	1199	1	1199.00	0.00%
sys_call	sys_select	008e	8733	1	8733.00	0.00%
IRQ	timer	0	448768	45	9972.62	0.22%
IRQ	keyboard	1	8806	1	8806.00	0.00%
IRQ	AceNIC Gigabit Ethernet	10	11245407	1291	8710.62	5.61%
IRQ	ide0	14	13596	1	13596.00	0.01%
other	other process	319	11996			0.01%
other	rx_thread	857	8661692			4.32%
idle	idle process	0	127618013			63.62%
other	tx_thread	858	48838371			24.35%
user	user process	862	20966			0.01%
Total	Total cycles		200591892			100.00%

Table 8.10 Analysis Table produced by fkt_print for iSCSI Low-Level driver.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for LLD queuing length change using Equation 8.4:

LLD Queuing Length	Total % CPU utilization
1	36.58
8	36.36

Table 8.11 % CPU utilization comparison for CITI change.

As shown in Table 8.11, the % CPU utilization on the initiator system does not change with LLD queuing length on the initiator.

8.7.2 Effect of Max PDU Size on Bandwidth for iSCSI

The Maximum PDU size for the iSCSI Protocol is changed to see the effect on Bandwidth for WRITE operation. When the Maximum PDU size is decreased from 524288 bytes to 1024 bytes, the bandwidth decreases from 21 MB/s to 19 MB/s. Each bandwidth value plotted on the Y-axis is the average value for 3 sample runs. When the maximum Data PDU size is decreased, the number of iSCSI PDUs required to transfer same amount of data increases. It is due to this reason that the bandwidth decreases when the Maximum PDU size decreases.

As a note, the Maximum iSCSI PDU size of 524288 bytes is chosen so that all the WRITE data involved with a SCSI Command fits into a single iSCSI PDU.

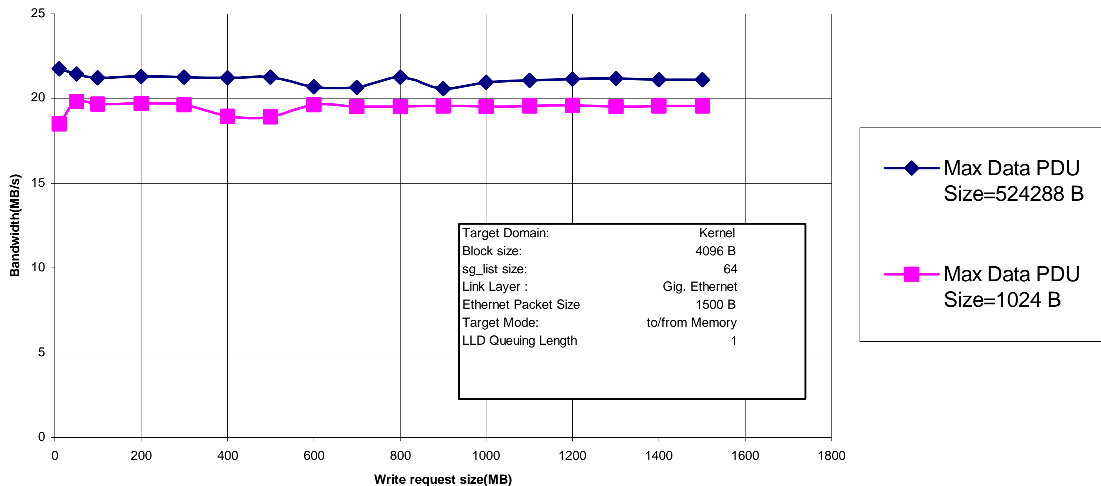


Fig. 8.10 Effect of Max Data PDU Size on Bandwidth for iSCSI.

CPU utilization comparison

The total % CPU utilization on the initiator system is compared for LLD queuing length change using Equation 8.4.

Max PDU Size	Total % CPU utilization
1024 bytes	44.26
524288 bytes	36.58

Table 8.12 % CPU utilization comparison for LLD queuing length.

As shown in Table 8.12, the % CPU utilization on the initiator system increases with a decrease in Max PDU size. When the Maximum PDU size is 1024 bytes, the initiator makes more iSCSI PDUs than when the Max PDU size is 524288 bytes. So, more CPU cycles are consumed when the initiator has Maximum PDU size set to 1024 bytes.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Design and Implementation

This thesis presents a general architecture for implementing Session Layer Protocols on the initiator complaint with their latest draft versions. The existing target emulator is modified and extended to support the additional features specified in the latest iSCSI draft.

The SEP initiator follows a synchronous model where the Low-Level Driver (LLD) can handle a single command at a given time. The SEP LLD processes the command completely before accepting the next command. A single thread is used for sending and receiving SEP PDUs. The iSCSI initiator follows an asynchronous model. Unlike the SEP LLD, the iSCSI LLD can handle multiple commands at a given time. Two threads, one for receiving and the other for transmitting, are used to communicate with the target.

Performance Analysis

The Performance Parameters that affect the bandwidth for WRITE operations are:

- Block Size
- Ethernet Link Speed
- Ethernet Packet Size
- Maximum PDU size for iSCSI Protocol

The Performance Parameters that do not affect the bandwidth for WRITE operations are:

- Target Domain
- Scatter-Gather List Size
- Coalescing Interrupts
- Queuing Length for Initiator Low-Level Driver

On a Fast Ethernet Link, the recorded bandwidth for WRITE operation is 10 MB/s, using 86 % of the maximum bandwidth possible at 11.2 MB/s. On a Gigabit Ethernet link, the absolute bandwidth increased to 21 MB/s but the percentage bandwidth utilization is only 19 %, of the maximum possible bandwidth at 112 MB/s.

The Nagle Algorithm should be turned OFF when doing READ/WRITE operations in order to gain high bandwidth and low latency.

The Performance comparison between the old (Alteon) Gigabit Ethernet Card and the new (3 Com) Acenic Gigabit Ethernet Card suggests that the NIC processing overhead is high. The high NIC processing overhead results in higher idle process % CPU utilization on the initiator system. The idle process takes 62.44 % of the total CPU cycles involved during WRITE operation.

9.2 Future Work

The future work involves support of additional features for the iSCSI Protocol on the initiator and the target emulator. The features to be added in the current implementations are as follows:

- Test and Support Multiple Connections in a Session
- Test Multiple Sessions on the initiator and the target emulator.
- Header and Data Digests
- Security
- TCP Markers

Also, the iSCSI Protocol support should be extended to the latest IETF draft versions as they are made available.

The performance analysis is done only on the initiator system as part of this thesis work. The target system should also be analyzed with the software probes to find out the bottlenecks affecting the performance metrics. Also, a similar analysis should be done for READ operations (on both the initiator and the target emulator).

A detailed analysis is required to be done for each network stack component involved in the I/O operations. The latency measurements, not performed in this thesis, should be done for each component to find the actual bottlenecks affecting performance.

From the performance analysis section, it is observed that the % CPU utilization on the initiator is low. Also, the bandwidth utilization on Gigabit Ethernet link is just 19 %. It is suggested to run multiple simultaneous applications on the initiator system in an attempt to increase the % CPU utilization of the systems and the available bandwidth on the wire. This is to get a better test of how well we can use the available bandwidth.

The performance analysis should also be done on Gigabit Ethernet NICs from different manufacturers to compare NIC processing time on each NIC.

Finally, hardware implementation for TCP/IP and iSCSI, if available, should be tested and compared with Fibre Channel technology for bandwidth, latency and CPU utilization.

.

References

1. G. Field and P. Ridge, *The Book of SCSI*, 2nd ed., No Starch Press, March 2000.
2. J. Dedek, *Basics of SCSI*, 3rd ed., Ancot Corporation, March 1992.
3. S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "NFS version 4 Protocol", *IETF RFC 3010*, December 2000.
4. R. Snively, "Information Technology – dpANS Fiber Channel Protocol for SCSI (SCSI-FCP)", *Proposed Draft (ANSI X2.269-1995, Revision 12)*, December 1995.
5. R.O.Weber, "Information Technology - SCSI Architecture Model –2 (SAM-2)", *Working Draft (T10 Project 1157D, Revision 15)*, November 2000.
6. M. Reardon, "Blaze Breaks Gig Ethernet Distance Barrier", *Technical Article*, Data Communications, 3 May, 1999.
7. R.D. Russell, B.B. Reinhold, Chris Loveland, "Options for Storage Area Networks", *Preliminary Report*, InterOperability Lab, University of New Hampshire, 21 June, 2000.
8. Sun Microsystems Inc., "RPC Remote Procedure Call Protocol specification", *IETF RFC 1057*, April 1988.
9. A. Palekar, "Design and Implementation of the SCSI Target Mid-level for the Linux Operating System", *M.S. Thesis*, Dept. of Computer Science, University of New Hampshire, May 2001.
10. "The SCSI Encapsulation Protocol (SEP)", *IETF Internet* Draft¸ May 2000; http://www.ietf.org/internet-drafts/draft-wilson-sep-00.txt.
11. Borison, Adaptec, Inc., February 2001; http://www.adaptec.com/worldwide/company/pressrelease.html.
12. J. Satran et al., "Internet SCSI (iSCSI)", *IETF Internet Draft*, July 2000; http://www.haifa.il.ibm.com/satran/ips/draft-ietf-ips-iSCSI-07.txt.
13. M. Bakke et al., "iSCSI Naming and Discovery", *IETF Internet Draft*, August 2001; http://www.haifa.il.ibm.com/satran/ips/draft-ietf-ips-iscsi-name-disc-02.txt.
14. R.D. Russell, "FKT: Fast Kernel Tracing", *Technical Report 00-02*, Dept.of Computer Science, University of New Hampshire, March 2000.
15. M. Chavan, "Performance Analysis of Network Protocol Stacks Using Software Probes", *M.S. Thesis*, Dept. of Computer Science, University of New Hampshire, September 2000.
16. J. Nagle, "Congestion Control in IP/TCP Internetworks", *IETF RFC 896*, http://www.ietf.org/rfc/rfc0896.txt?number=896, January 1984.

APPENDICES

APPENDIX A

MODIFICATIONS TO THE TARGET EMULATOR

This Chapter explains the existing target emulator functionality in the first two sub-sections. In the final sub-section, the modifications made to the Target Emulator are discussed.

A.1. Target Emulator Overview

The Target Emulator provides the software Target functionality to be accessed by the Initiator. It is implemented in the Linux kernel, and its design parallels that of the SCSI Mid-level (SML) and Low-Level Driver (LLD) in the initiator. The Target Emulator is designed and implemented by Ashish [9].

The SCSI Target Emulator receives SCSI Commands from an Initiator. Command, Data, and Response transfer between the Initiator and the Target is handled by the interconnect-specific low-level front-end Target driver (FETD). The FETD strips off headers introduced by the protocol (SEP or iSCSI) used on the interconnect and hand off the SCSI command and data to the SCSI Target Mid-Level (STML). The STML processes and executes this command and hands back the responses (data and/or status) back to the FETD so that it can transmit them back to the Initiator. The STML is also able to respond to the error handling facilities provided by SCSI. The overview is presented in Fig. A.1

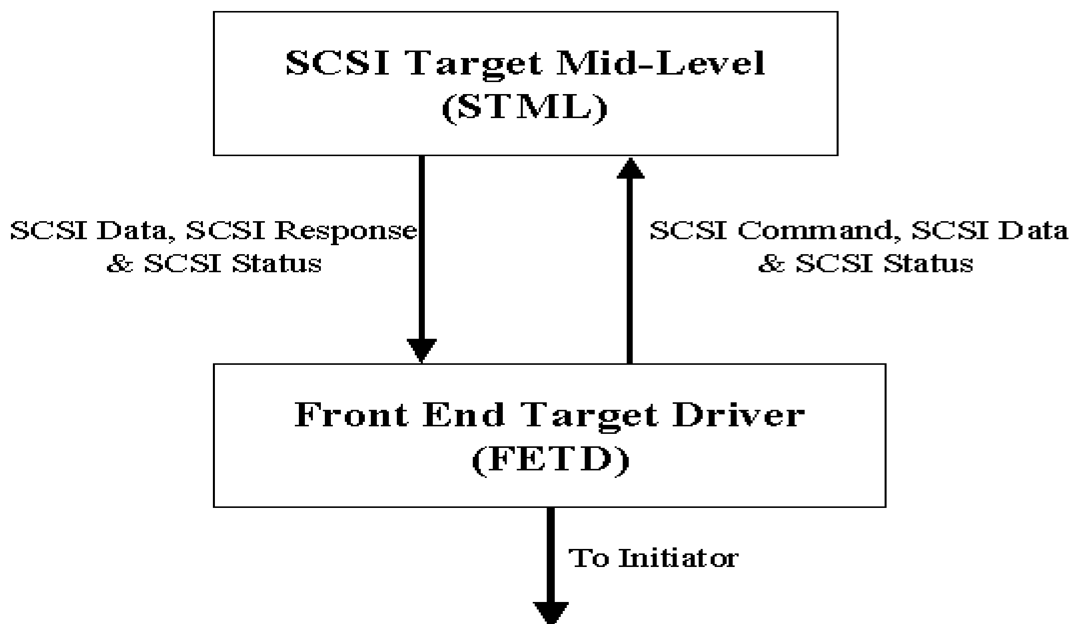


Fig A.1 Overview of API between STML and FETD.

A.2. Design of Target Emulator

This section provides a more detailed view of the functioning of the STML using the functions that are shown in Fig. A.2. The organization of the Data structures in the FETD is the same as shown in Fig. 7.for LLD in initiator. The FETD has two threads, one for transmitting and the other for receiving, to communicate with the initiator. This design is similar to that of LLD on the initiator system.

A.2.1. Processing a READ-type Command

The FETD registers itself with the STML by calling the function `register_target_front_end()` in the STML (Fig. A.2). Upon receiving a SCSI command from an Initiator in the form of iSCSI 'SCSI Command' PDU, the FETD calls the `rx_cmnd` function in the STML. The STML executes the command and if it is READ command fills the READ buffers. The STML then calls the `xmit_response` function in the `Scsi_Target_Template` (Fig. A.2), implemented by the FETD. The `xmit_response` function sends the iSCSI 'Data-In' PDU to the initiator followed by the iSCSI 'SCSI Response' PDU. When the Response is actually transmitted, the FETD calls the `scsi_target_done` function in the STML. This function allows the STML to free up resources that are allocated for a SCSI command.

A.2.2. Processing a WRITE-type Command

Upon receiving a SCSI command from an Initiator in the form of an iSCSI 'SCSI Command' PDU, the FETD calls the `rx_cmnd` function in the STML. The STML processes the command and if it is WRITE command, allocates the necessary buffers. The STML then calls the `rdy_to_transfer` function (Fig. A.2) in the FETD. A call to the `rdy_to_transfer` function means that the buffers required for the execution of the SCSI command have been allocated. The `rdy_to_transfer` function transmits the iSCSI 'Ready to Transfer' PDU to the initiator indicating that it can send the WRITE data. Once data has been received from the initiator, the FETD calls the `scsi_rx_data` function (Fig. A.2) in the STML. The STML calls the `xmit_response` function after SCSI command processing is complete. When the Response is actually transmitted, the FETD calls the `scsi_target_done` function in the STML. This function allows the STML to free up resources that are allocated for a SCSI command.

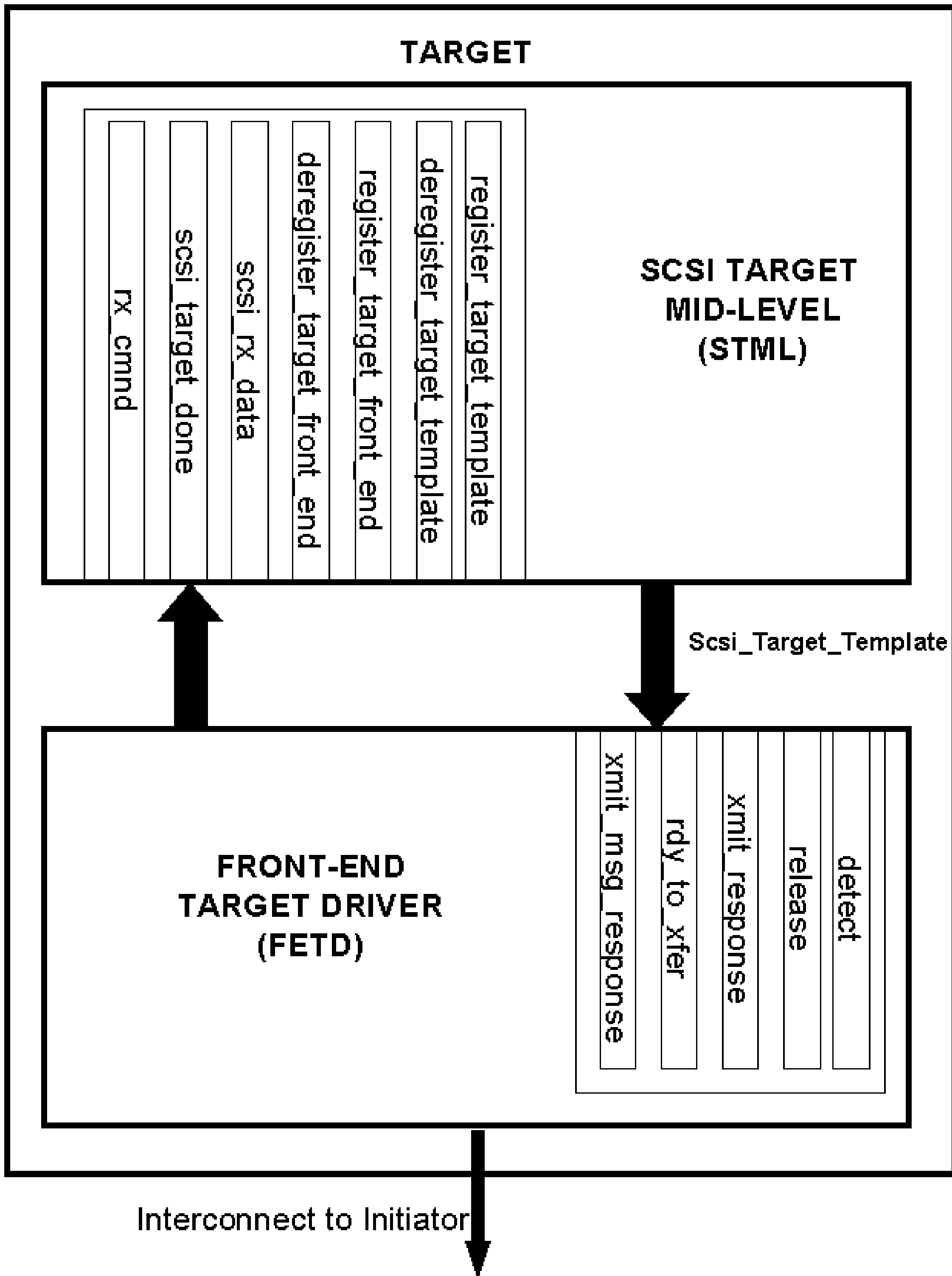


Fig A.2 API between STML and FETD.

A.3. Modifications Made to the Target Emulator

A.3.1. Support to Receive Immediate Data PDUS and Unsolicited Data-Out PDUs

Upon receiving a SCSI command from an Initiator, the FETD calls the `rx_cmnd` function in the STML. If there is Immediate Data present with the iSCSI 'SCSI Command' PDU, the `rx_thread` waits on a `unsolicited_data_sem` semaphore. After processing the command, the STML calls the `rdy_to_transfer` function (Fig. A.2) in the FETD. The `rdy_to_transfer` function will wake up the `rx_thread` waiting on the `unsolicited_data_sem` semaphore. The `rx_thread` will receive the Immediate Data by calling `handle_immediate_data` function. After receiving the Immediate Data, this function will wake up the `tx_thread` to send an iSCSI 'R2T' PDU to the initiator if more WRITE data is to be transferred from the initiator.

When Unsolicited Data PDU is received from the initiator, the `rx_thread` waits on the `unsolicited_data_sem` semaphore. The `rdy_to_transfer` function, called by STML, will wake up the `rx_thread` waiting on the `unsolicited_data_sem` semaphore. The `rx_thread` will then receive all the Unsolicited iSCSI PDUs. After receiving the last Unsolicited Data PDU, the `rx_thread` checks if more WRITE data has to be transferred from the initiator. If more data is required, the `rx_thread` wakes up the `tx_thread` to send an iSCSI 'R2T' PDU.

A.3.2. Implementation of Task Management function in the iSCSI FETD

When an FETD receives a Task Management function in form of a iSCSI 'Task Management Command' PDU, it informs the STML about the received Task Management function by calling the `rx_task_mgmt_fn` function in the STML. The STML aborts the relevant SCSI command and calls the `task_mgmt_fn_done` function in the FETD. The `task_mgmt_fn_done` function sends the iSCSI 'Task Management Response' PDU to the initiator.

A.3.3. Support of Multiple Data-In PDUs in the iSCSI FETD

As mentioned before, the Data Structure Organization in the iSCSI FETD is the same as LLD in the initiator (Fig.7.2). The **`parameter_type`** struct, mentioned in Fig. 7.3, is added to the `session` struct in the FETD to support use of operational parameters in the iSCSI FETD. The 'DataPDULength' key in the **`parameter_type`** struct is used by the `xmit_response` function for sending multiple iSCSI 'Data-In' PDUs to the initiator.

APPENDIX B

SCSI COMMAND OPCODES FOR DIRECT ACCESS DEVICES

The SCSI Command opcodes can be one of the following:

OP Code	Command Name	OP Code	Command Name
00h	TEST UNIT READY	32h	SEARCH DATA LOW
01h	REZERO UNIT	33h	SET LIMITS
03h	REQUEST SENSE	34h	PRE-FETCH
04h	FORMAT UNIT	35h	SYNCHRONIZE CACHE
07h	REASSIGN BLOCKS	36h	LOCK/UNLOCK CACHE
08h	READ (6)	37h	READ DEFECT DATA
0Ah	WRITE (6)	39h	COMPARE
0Bh	SEEK (6)	3Ah	COPY AND VERIFY
12h	INQUIRY	3Bh	WRITE BUFFER
15h	MODE SELECT (6)	3Ch	READ BUFFER
16h	RESERVE (6)	3Eh	READ LONG
17h	RELEASE (6)	3Fh	WRITE LONG
18h	COPY	40h	CHANGE DEFINITION
1Ah	MODE SENSE (6)	41h	WRITE SAME
1Ch	RECEIVE DIAGNOSTIC RESULTS	4Ch	LOG SELECT
1Dh	SEND DIAGNOSTIC	4Dh	LOG SENSE
1Eh	PREVENT/ALLOW MEDIUM REMOVAL	55h	MODE SELECT (10)
25h	READ CAPACITY	56h	RESERVE (10)
28h	READ (10)	57h	RELEASE (10)
2Ah	WRITE (10)	5Ah	MODE SENSE (10)
2Bh	SEEK (10)	5Eh	PERSISTENT RESERVE IN
2Eh	WRITE AND VERIFY	5Fh	PERSISTENT RESERVE OUT
2Fh	VERIFY	A0h	REPORT LUNS
30h	SEARCH DATA HIGH	A7h	MOVE MEDIUM ATTACHED
31h	SEARCH DATA EQUAL	B4h	READ ELEMENT STATUS

APPENDIX C

README for iSCSI Initiator & Target Implementations

The reference iSCSI initiator and target are implemented as dynamically loadable modules in the linux 2.4.0-test9 kernel. The iSCSI initiator and target emulator implementations are available from the iSCSI Consortium, IOL web-page (http://www.iol.unh.edu/consortiums/iscsi) under "iSCSI Implementations". The README for the implementations are organized into following sub-sections.

1. Draft Version Compliance
2. Operational Parameters Used in the Full Feature Phase
3. Known Limitations
4. Testing done on the Reference Implementations
5. PDU support
6. How to Install
7. OS support
8. Default Mode of Reference Initiator and Target
9. Use of ISCSI_CONFIG Tool
10. Use of ISCSI_MANAGE tool

1. Draft Version Compliance

The reference implementations comply to iSCSI draft 7 (dated July 20,2001) which is available at http://www.ietf.org/internet-drafts/draft-ietf-ips-iscsi-07.txt.

2. Operational Parameters Used in the Full Feature Phase

All the operational parameters are negotiable in the login phase. However, in the full feature phase operation, the reference initiator and target make use of the following negotiated operational parameters only.

- DataPDULength
- FirstBurstSize
- InitialR2T
- ImmediateData

The other negotiated security/operational parameters values are ignored (or not used) by the reference initiator or target during Full Feature phase. Such parameters are explained here:

- MaxConnections value is ignored. The initiator and the reference target support only single connection for a single session.
- `FMarker`, `RFMarkInt`, `SFMarkInt`, and `IFMarkInt` are not supported and hence, the negotiated operational parameters are not used.

- BidiInitialR2T is not supported and hence its value is not used.
- `MaxOutstandingR2T` is not supported. The initiator always has one outstanding R2T at any instant. Hence, the `MaxOutstandingR2T` value is not used.
- DataOrder value is ignored as the reference initiator and target expect incoming data to be in order and send the outgoing data in order.
- Header and Data Digests are not supported.
- Security is not supported.

The use/unuse of the iSCSI parameters have been elaborated through the following examples.
<u>Example 1:</u> For any Write SCSI Command, the initiator will look in the negotiated operational parameter table for 'ImmediateData' value. It will then decide whether to send 'ImmediateData' or not based on that value.
<u>Example 2:</u> When a new session is started, there is only 1 connection supported for that session. The initiator will just make one connection to the target even if the 'MaxConnections' value in the negotiated operational parameter table is different than 1.

3. Known Limitations

- The Reference Initiator and Target implementations just support ABORT TASK functionality in the set of Task Management Commands. The other Task Management Commands ABORT_TASK_SET, CLEAR ACA, CLEAR TASK SET, LOGICAL UNIT RESET, TARGET WARM RESET, and TARGET COLD RESET are not supported.
- The Reference Target always sends a separate SCSI Response PDU when a READ/WRITE request sent by the initiator is satisfied. The Piggybacking of Status in the Last DataIn PDU on the initiator has not been tested.
- The Target emulator expects the commands from the initiator in CmdSN order. The Data Sequence Numbering for Data PDUs is not supported.
- No error recovery mechanisms are implemented.
- Parameter negotiation is only restricted to the Login Phase.

4. Testing done on the Initiator and Target Implementations

The initiator and the target have been tested fairly well for the following functionalities:

- Single connection for a single session between the reference initiator and the target.
- Unsolicited Data(as Immediate Data as well as separate Data PDUs)
- Support of multiple pending commands in the reference initiator and target. The maximum queuing length (i.e., number of pending commands) for which the initiator and the target code has been tested is 8.
- Number of scatter gather list entries =16, 32, and 64, on the reference initiator.

5. PDU support

The reference initiator and target only understand the following iSCSI PDUs:

0x01	SCSI Command (encapsulates a SCSI Command Descriptor Block)
0x02	SCSI Task Management Command
0x03	Login Command
0x04	Text Command
0x05	SCSI Data-out (for WRITE operations)
0x21	SCSI Response (contains SCSI status and possibly sense information or another response information)
0x22	SCSI Task Management Response
0x23	Login Response
0x24	Text Response
0x25	SCSI Data-in (for READ operations)
0x31	Ready To Transfer (R2T - sent by target to initiator when it is ready to receive data from initiator)
0x3f	Reject

The reference initiator and the target will not understand the following iSCSI PDUs:

0x00	NOP-Out (from initiator to target)
0x06	Logout Command
0x10	SNACK Request
0x1c-0x1e	Vendor specific Codes
0x20	NOP-In (from target to initiator)
0x26	Logout Response
0x32	Asynchronous Message
0x3c-0x3e	Vendor specific Codes

6. How to Install

This section gives the location of all the files and directories. It also gives information on how to run the initiator and target implementations. The iSCSI implementation is divided into the following three directories:

<u>iscsi_initiator_ver_6.08</u>

This directory contains the initiator-specific files. The initiator can be compiled by running Makefile in this directory. The initiator can be installed by running the script 'iscsi_initiator_install' on shell prompt.

The script will first insert the iscsi initiator driver as a module by executing the following command:

```
insmod iscsi_initiator.o
```

The script will then bring up the iscsi interface by executing the following command:

```
./iscsi_config up ip=<hostname> port=<port number> target=0
```

The use of iscsi_config tool is explained in Appendix A.

iscsi_target_emulator_ver_6.08
This directory contains the target specific files. The target can be compiled by running Makefile in this directory.

The target can be installed by running the script 'iscsi_target_install' residing in this directory. The script will first insert the SCSI target mid-level by writing the following command:

```
insmod scsi_target.o
```

The script will then insert the iSCSI Target front-end by writing the following command:

```
insmod iscsi_target.o
```

The target listens at port 4000, which can be changed at compile time. The value is specified by the constant ISCSI_PORT in file common/iscsi_common.h.

common
This directory contains the common files used by both the initiator and the target.

7. OS used

Linux 2.4.0-test9 on Intel platform.

8. Default Mode of Operation of Initiator and Target

Default Mode is the mode in which the reference initiator and target modules operate when the modules are compiled and loaded. The default mode can be changed with the help of management tools called ‘iscsi_manage’ and ‘iscsi_config’ explained in Appendix A and B. The default mode has the following features:

- Well known port that the target listens on is 4000.
- Reference Initiator and Target don’t do parameter negotiation (except the parameters mentioned in 2) and accept the default operational parameters values as specified in Appendix E of iSCSI draft 7.
- The KERNEL_DIR constant defined in Makefile (present in directories iscsi_initiator_ver_6.08, iscsi_target_emulator_ver_6.08 and common) are defined as /home/iol/ng3/linux-2.4.0-t9. This can be changed to the referred linux 2.4.0-test9 source tree on the system where the loadable modules are compiled.

- The Target mode operates in FILEIO mode. This is a constant defined in file iscsi_target_emulator_ver_6.08/scsi_target.h. Different Target modes are explained in Appendix C.
- The debug messages are turned ON because constant CONFIG_ISCSI_DEBUG is defined during compile time in the initiator and target Makefiles. The debug messages can be turned OFF if we don't define the constant CONFIG_ISCSI_DEBUG during compile time.

9. ISCSI_CONFIG Tool

iscsi_config is a tool, developed by Narendran Ganapathy, to specify iSCSI targets to the kernel initiator and initiate iSCSI protocol operations.

iscsi_config is invoked with at least one mandatory argument and five optional arguments as given below.

```
iscsi_config     <what-to-do>     [host=number]     [ip=address_or_name]
[port=number] [target=number] [lun=number] |
```

what-to-do
what-to-do determines whether an iSCSI session has to be established between an iSCSI initiator and an iSCSI target or whether an existing iSCSI session needs to be closed.
It can be one of the following:
"up" an iSCSI session needs to be established.
"down" an iSCSI session needs to be closed.

host=number
This is an optional field. If omitted assumes a value of 0. Format is

```
host=<number>
```

This specifies the SCSI Adapter number assigned by the kernel to the iSCSI initiator.

ip=address_or_name
This is an optional field that specifies the IP address of the iSCSI target to which a connection has to be established. If omitted assumes the local IP address. Format is

```
ip=name  (or) ip=address
```

where
"name" is the hostname of the machine in which the iSCSI target is waiting for requests from the iSCSI initiator.
"address" is the IP address in dotted decimal notation.

port=number
This is an optional field that specifies the port on which the iSCSI target is listening. If omitted assumes the port number of 4000.

target=number
This is an optional field that specifies the target number(in other words, ID in SCSI addressing scheme). When the iSCSI initiator module 'iscsi_initiator.o' is loaded, iscsi_initiator_detect function is called which specifies the number of targets a particular initiator(host) has access to. All these targets are given numbers by the kernel which corresponds to Target ID in [Host, Channel, ID, LUN] SCSI addressing scheme.

10. ISCSI_MANAGE tool

iscsi_manage is a tool, developed by Narenedran Ganapathy, to configure iSCSI parameters on the iSCSI initiator and iSCSI target implementations. The iSCSI initiator and iSCSI target are implemented as dynamically loadable modules in the linux 2.4.0-test9 kernel.

iscsi_manage uses the table of parameters that has been configured with default values on the iSCSI initiator and iSCSI target modules at compile time. iscsi_manage can change the default values associated with parameters and can set the type of negotiation allowed for the parameter. Each parameter falls into exactly one of the following categories for negotiation.

Category	Description
1	Parameter that has to be negotiated and whose value can be changed on negotiation.
2	Parameter that has to be negotiated and whose value can not be changed on negotiation.
3	Parameter whose default value has to be changed but not to be negotiated.

It is invoked with at least three mandatory arguments and an optional argument from the command line as given below.

```
iscsi_manage <role> <what-to-do> <parameter-value-list>
             <optional-host>
```

role
role determines the device that is being configured. It can be one of the following:
"init" the device that is configured is an iSCSI initiator
"target" the device that is configured is an iSCSI target

what-to-do
what-to-do specifies the operation to be performed on the parameter. It can be one of the following:

“set” sets the parameter's type to category 1.
“setr” sets the parameter's type to category 2.
“setp” sets the parameter's type to category 3.
“restore” restores the type of all parameters with the default set of values (set of values the parameters took when the module was first installed)

parameter-value-list

parameter-value-list specifies the parameter that is configured with the value-list. It is of the form

```
<Parameter>=<value_list>
```

where <Parameter> can take the following values: "MaxConnections", "TargetName", "InitiatorName", "TargetAlias", "InitiatorAlias", "TargetAddress", "SendTargets", "AccessID", "FMarker", "RFMarkInt", "SFMarkInt", "IFMarkInt", "InitialR2T", "BidiInitialR2T", "ImmediateData", "DataPDULength", "FirstBurstSize", "LogoutLoginMinTime", "LogoutLoginMaxTime", "EnableACA", "MaxOutstandingR2T", "DataOrder", "BootSession", "HeaderDigest", "DataDigest", "AuthMethod", "SecurityContextComplete", and <value_list> is an UTF-8 string.

<Parameter> and <value_list> are separated by a '=' character.

optional-host

This is an optional field. If omitted assumes a value of 0. Format is

host=<number>

If the role is initiator this specifies the SCSI Adapter number assigned by the kernel to the iSCSI initiator. If the role is target this specifies the target number assigned by the SCSI Target Mid-Level to the iSCSI target.

Examples

1. If the target can accept just a single connection it has to set the MaxConnections parameter's value to 1.

```
iscsi_manage target setr MaxConnections=1
```

setr is used because the target cannot support value other than 1 for MaxConnections parameter.

2. If the initiator supports both the options (yes/no) but likes the no value as the default for InitialR2T, then we might want to configure the initiator like this

```
iscsi_manage init set InitialR2T=no
```

3. If the initiator just wants to change the default behavior of the key InitialR2T (setting InitialR2T=no) but did not want to negotiate for it, then

```
iscsi_manage init setp InitialR2T=no
```

4. If the initiator wants to restore the default values of all the parameters to its original state (state when the module was initially loaded),

```
iscsi_manage init restore
```