DESIGN AND IMPLEMENTATION OF A SCSI TARGET FOR STORAGE AREA NETWORKS

Ashish A. Palekar

Robert D. Russell

TR 01-01

May 2001

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

DESIGN AND IMPLEMENTATION OF A SCSI TARGET FOR STORAGE AREA NETWORKS

by

Ashish A. Palekar
University of New Hampshire, May 2001

The Small Computer Systems Interface (SCSI) has been used to transmit data between applications (Initiators) and storage devices (Targets). One of the major limitations of SCSI has been the length of the SCSI bus. With the evolution of Storage Area Networks (SANs), several protocols have been proposed to extend the length of the SCSI bus e.g., Fibre Channel, SCSI Encapsulation Protocol (SEP), and Internet SCSI (iSCSI). The evaluation of these technologies requires the use of an Initiator and a Target that implement the said protocols. A large portion of what such Initiators or Targets need to do from a SCSI perspective can be isolated into a logical code unit referred to as a mid-level. While there exists in the Linux kernel a generic SCSI Initiator mid-level that drivers written for various Initiators can interface with, no corresponding facility exists for the Target side. This thesis involves the development of a Generic SCSI Target mid-level for Linux along with implementing front-end drivers for Fibre Channel, SEP and iSCSI that can utilize the said Target mid-level. Other uses for the Target Emulator are as a bridge between two protocols and as an interface for SAN Management.

CHAPTER 1

INTRODUCTION

1.1 Motivation and Goals for this Thesis

The past few years have seen a tremendous growth in the amount of traffic on the Internet. An increase in the number of people with access to the Internet has meant that the Internet has become a viable pathway for trade and related commerce activities. Increasingly, mission-critical applications are being put on the Internet. This increase in traffic over the Internet has meant an exponential growth in the data that needs to be stored. Access to this data has become a critical resource and traditional server-based approaches to data access indicate this as a potential bottleneck in the very near future. This has led to the emergence of the concept of Storage Area Networks (SANs) where SCSI (Small Computer System Interface) is used to exchange data between an application (Initiator) and the storage device (Target).

Fibre Channel was one of the first protocols designed with the idea of SANs in mind. Interoperability issues and incompatibility with existing infrastructure has led to the development of several different protocols in place of Fibre Channel with the objective of transmitting SCSI data over existing TCP/IP networks. The evaluation of these protocols requires an Initiator and a Target that can transmit SCSI data using the said low-level protocols. A good way in which this can be achieved is by isolating the common portions of what these Initiators and Targets need to do in terms of a logical unit of code that is responsible for processing SCSI commands, data and responses. To adapt this unit of code to a specific SCSI Transport Protocol, a relatively simpler front-end driver can be written that handles the details of the SCSI Transport Protocol itself. Thus, an Initiator or a Target driver for a SCSI transport protocol would consist of two portions – a common SCSI processing portion which would be common to all SCSI Transport Protocols and a second portion specific to the SCSI Transport Protocol. The common SCSI processing portion is referred to as the SCSI mid-level (for reasons explained in Chapter 5) whereas the latter is referred to as the front-end driver. The Linux kernel has existing support for SCSI Initiators in terms of a SCSI Initiator mid-level (SIML). Such a mid-level does not exist for SCSI Targets. This thesis aims at developing a SCSI Target mid-level (STML) for the Linux kernel. Three front-end Target drivers will be written to interface with this SCSI Target mid-level implementing the SCSI Encapsulation Protocol (SEP), the Internet SCSI (iSCSI) and Fibre Channel SCSI Transport Protocols.

1.2. Resources Used

The major resource used for this project is a Fibre Channel card. For the purposes of this project, QLogic Corporation made two ISP2200 A cards available. The facilities to test the implemented code are available through the InterOperability Lab, University of New Hampshire.

The ISP 2200 card is a 64-bit PCI to Fibre Channel card capable of operating at 33 and 66 MHz PCI. The cards support up to 200 MB/s Fibre Channel data transfer rates in full duplex. It supports the Class 2 and Class 3 service in all Fibre Channel topologies. The card supports SCSI

Initiator and Target operation. A description of the firmware support for the QLogic ISP 2200 A card is described in Appendix C.

The SCSI Initiator driver for the ISP 2200 A card was written by Chris Loveland, University of New Hampshire. This driver was used as a basis for developing the Fibre Channel Target driver. This Target driver interfaces with the SCSI Target Mid-Level. Xyratex Corporation has implemented a Target Emulator using the QLogic 2200 card for the Windows Operating system with the intent of using it as a testing tool. No access is currently available to this testing tool. Matthew Jacob of Feral Inc. (http://www.feral.com), has written an Initiator and Target driver for ISP 2100/2200/2200 A for various flavors of Unix. The Target driver written by him has a section of code that can be isolated as a SCSI Target mid-level. This code has been looked at and it is not functional on Linux. No attempt has been made at debugging this Target driver.

The drafts of the relevant SCSI ([1], [13], [14], [16]) and Fibre Channel ([2], [3], [15]) standards are available at the IOL. In addition, access to the draft standards is provided via the websites of the respective standards bodies (http://t10.org and http://www.t11.org). The relevant SEP draft [9] and the corresponding iSCSI draft [7] are available at the IETF (http://www.ietf.org) website.

1.3 Organization of the Thesis

Chapter 2 explains the SCSI protocol, detailing the issues involved with special emphasis on the operational details necessary from the point of view of a SCSI Target. Chapter 3 of this Thesis explains the concept of Storage Area Networks, the emergence of this concept and the enabling factors. Chapter 4 deals with SCSI Transport Protocols such as Fibre Channel, SEP and iSCSI. This chapter tries to detail the aspects that make these protocols unique and in conjunction with Chapter 2 helps to isolate the needs of a SCSI Target mid-level and any front-end driver. Chapter 5 details the design of a SCSI Target Emulator. It starts with the design of the SIML in the Linux kernel, rationalizes the design of a Target Emulator based on the inferences from the previous chapters and finally presents applications relevant to a SCSI Target Emulator. Chapter 6 deals with the user space implementation of the SCSI Target Emulator whereas Chapter 7 deals with the corresponding kernel space implementation along with the implemented front-ends. Chapter 8 presents basic testing and performance analysis performed on the three implementations and the corresponding front-ends more with an objective to get an “order of magnitude” idea about I/O rates. Chapter 9 summarizes the work done, and the conclusions drawn along with work that can be done in the future.

CHAPTER 2

SMALL COMPUTER SYSTEM INTERFACE

2.1 Introduction

Small Computer System Interface (SCSI) originated from the Selector Channel on IBM-360 computers, and was later scaled down by the Shugart Associates Company to make a universal, intelligent disk drive interface. It was called the Shugart Associates Systems Interface (SASI). SCSI became an ANSI standard in 1986. SCSI is an intelligent, parallel peripheral bus, with medium-to-high performance. SCSI is both a bus hardware specification and a command set to optimize the use of that bus. Over time, the SCSI standards have grown to recognize several devices - magnetic disks drives, tape drives, printers, scanners, processors, communications devices among several others. SCSI speeds now range from 1 MB/s to 160 MB/s.

2.1.1 Reasons for SCSI

Prior to the development of SCSI, for each new peripheral that was added to a given computer system, the computer had to be specially configured to manipulate the hardware in order to accomplish the task of reading and writing data to and from the device. This implied that more often than not, by the time the hardware and software design of the computer was complete, a new generation of peripherals was usually available. Thus, the peripherals attached to a computer were often a generation or more behind the computer itself.

The basic premise of SCSI is to give the computer complete device independence. In other words, all magnetic devices appear identical to the system except for their total capacity. All printers are identical as are all CD-ROMs. With SCSI, the system should not need any modification when replacing a device from one manufacturer with that from another manufacturer. The major implication for the development cycle therefore, is that the developer no longer has to write a new I/O driver for a brand new peripheral. The onus 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.

2.1.2 SCSI Terminology

There are two kinds of devices on the SCSI bus: the SCSI *Initiators* – which start the I/O process and the *Targets* - which respond to a request to perform an I/O process. The traditional 'master' and 'slave' functions switch back and forth between Initiators and Targets. The single-byte (Narrow) SCSI bus supports up to eight devices whereas the 16-bit Wide SCSI bus supports 16 devices. For any given set of interconnected devices, there must be at least one device capable of providing the functionality of an Initiator and at least one other capable of providing the functionality of a Target.

The Initiator starts arbitration and selects a Target. The corresponding Target then requests a command from the Initiator. The Initiator then sends a Command Descriptor Block (CDB) over to the Target. The Target then executes the received CDB and returns the appropriate response.

Each Target device can also be subdivided into several Logical Units (LUNs – explained later). A representation of a SCSI system is shown in Figure 2-1. It is also possible to connect several computers, each with one or more SCSI host adapters, to a shared peripheral, such as a SCSI scanner. The maximum number of SCSI devices (each of these devices having a SCSI_ID) on a single-byte SCSI is eight. In addition to this, each of the devices, except the Initiator, can have up to eight logical units (LUNs). This brings the theoretical maximum number of devices/LUNs on the eight-bit SCSI bus to 57 (1 Initiator + 7 Targets x 8 LUNs).

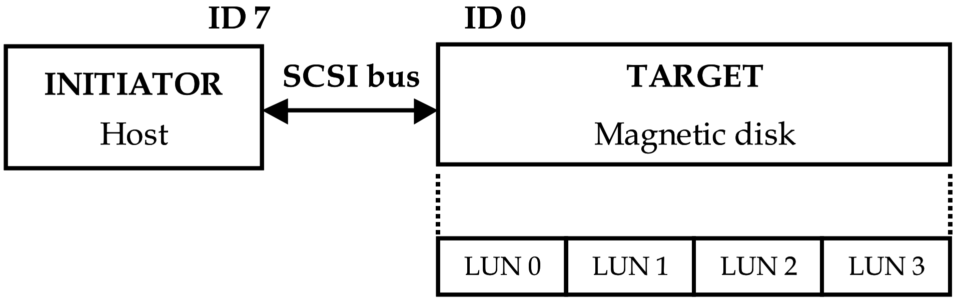

Figure 2-1: Representation of a typical SCSI system

A further level of abstraction is achieved in SCSI with the introduction of the concept of Logical Block Addressing. If a computer needs access to data, it is addressed in terms of a certain Logical Block Address (LBA). The Initiator does not need knowledge about the physical geometry or details about the layout of the drive in question. In this sense, a logical address is very similar to the concept of a virtual address. In other words, an Initiator issues commands directed to a specific set of LBAs. The Target upon receiving commands converts the LBAs into actual information about the track, cylinder head and the sector address. With some loss of efficiency, LBAs provide a uniform means for the Initiator to refer to the Targets. The mechanism of addressing is made device-independent. The Initiator uses the READ CAPACITY command in order to determine the maximum LBA on a magnetic disk as well as the size of one Logical Block.

2.1.3 SCSI Commands and their format

A SCSI command is generated by the Initiator (on the host) and is sent to the Target during the command phase (described later). A command and its parameters are sent as a block several bytes long called the Command Descriptor Block (CDB). SCSI commands can be classified into three types of commands based on the length of their CDBs:

Group	– 0	uses 6-byte CDBs
	- 1, 2	uses 10-byte CDBs
	- 5	uses 12-byte CDBs

Command groups 3 and 4 are reserved, whereas command groups 6 and 7 are vendor-specific.

SCSI was initially designed for magnetic disks which in the late seventies had relatively small capacities compared with those available today. Thus, transfer sizes could be adequately expressed in one byte - as a result, the Group-0 commands were sufficient. The 10-byte and 12-byte CDBs were added for the vastly expanded capacities of present day devices.

An example of a SCSI command is the READ(6) command [14]. The CDB for the READ(6) command is shown in Figure 2-2. The first byte of the CDB is the Operation Code (OP Code). A list of OP Codes and the corresponding command names is shown in the Appendix A. It is followed by the LUN in the upper three bits of the second byte, and by the LBA and transfer length fields (READ and WRITE commands) or other parameters. The last byte of each CDB is the Control byte. This byte contains two important bits, the LINK and the Normal ACA. The LINK bit is used to continue a Task across multiple commands whereas the Normal ACA bit is used to control the rules for handling error conditions created by the failure to execute a command.

Bit **Byte**	**7**	**6**	**5**	**4**	**3**	**2**	**1**	**0**
0	Operation Code (08h)							
1	Logical Unit Number			Logical Block Address (MSB)				
2	Logical Block Address							
3	Logical Block Address							
4	Transfer Length							
5	Vendor Unique		Reserved			NACA	Obsolete	Link

Figure 2-2: The READ(6) CDB

The READ(6) command requests that the Target transfer data to the Initiator. The current data values contained in the addressed logical block on the Target's storage device, shall be returned to the Initiator that transmitted the READ(6) command.

The Logical Block Address field specifies the logical block at which the READ operation shall begin. The transfer length field specifies the number of contiguous logical blocks of data to be transferred. A transfer length of zero indicates that 256 logical blocks shall be transferred. Any other value indicates the number of logical blocks that shall be transferred. The typical size of a logical block is 512 bytes, although it can be any multiple of 512 bytes (This value is obtained from the response to the READ CAPACITY command – described earlier).

The execution of SCSI commands can be thought of in terms of phases. The SCSI architecture includes eight distinct phases:

a. BUS FREE phase:
The BUS FREE phase indicates that there is no current I/O process and that the SCSI bus is available for a connection

b. ARBITRATION phase
The ARBITRATION phase allows one SCSI device to gain control of the SCSI bus so that it can initiate or resume an I/O process. Priority is given in accordance with device IDs. Higher IDs have higher priority. On the WIDE SCSI bus in SCSI-3, the low-byte Ids have higher priority over high-byte IDs. This is to allow the 8-bit devices to be always recognized by all other devices.

c. SELECTION phase
The SELECTION phase allows an Initiator to select a Target to initiate some Target function (e.g., READ or WRITE command).

d. RESELECTION phase

The RESELECTION phase is an optional phase that allows a Target to reconnect to an Initiator for the purpose of continuing some operation that was previously started by the Initiator but was suspended by the Target (i.e, the Target disconnected by allowing a BUS FREE phase to occur before the operation was complete).

e. COMMAND phase

The COMMAND phase allows the Target to request command information from the Initiator.

f. DATA phase

The DATA phase consists of both the DATA IN and the DATA OUT phase. The DATA IN phase allows the Target to request that data be sent to the Initiator from the Target. The DATA OUT phase allows the Target to request that data be sent from the Initiator to the Target.

g. STATUS phase

The STATUS phase allows the Target to request that status information be sent from the Target to the Initiator.

h. MESSAGE phase

The MESSAGE phase consists of both a MESSAGE IN, and a MESSAGE OUT phase. Multiple messages can be sent during either phase. The first byte transferred in either of these phases shall be either a single-byte message or the first byte of a multiple-byte message. Multiple-byte messages shall be wholly contained within a single message phase. The MESSAGE IN phase allows the Target to request that messages be sent from the Target to the Initiator. The MESSAGE OUT phase allows the Target to request that messages be sent from the Initiator to the Target. The Target invokes this phase when the ATN (Attention) condition is invoked by the Initiator.

The last four phases are collectively referred to as the Information Transfer phases. The SCSI bus can never be in more than one phase at any given time. Typically, the Information Transfer phases are implemented in software whereas the first four are implemented in hardware. Furthermore, these hardware phases can be implemented in an interconnect-specific manner. Thus, SCSI leaves open to the protocols used by the low-level interconnect the method for how to select a Target and how to decide if the Target and the Initiator are ready for data transfer.

In addition to these phases, SCSI also defines two conditions: the Attention condition and the Reset condition.

The Attention condition allows an Initiator to inform the Target that the Initiator has a message ready. The Target may get this message by performing a MESSAGE OUT phase.

The Reset condition is used to clear all SCSI devices from the bus. This condition shall take precedence over all other phases and conditions. Any SCSI device may create the reset condition. The BUS FREE phase always follows the Reset condition. The effect of the Reset condition on the I/O processes that have not completed, SCSI device reservations, and the operating mode of SCSI device is determined by whether the SCSI device has implemented the hard reset alternative or the soft reset alternative (one of which shall be implemented). The hard and soft reset alternatives are mutually exclusive within a system.

A SCSI device that implements the hard reset alternative, upon detection of the Reset condition shall:

a. Clear all I/O processes including queued I/O processes.
b. Release all SCSI device reservations.
c. Return any SCSI device operating modes to their appropriate original conditions (similar to those conditions that would have been found after a power-on). MODE SELECT conditions shall be restored to their saved values if there have been saved values. In the absence of saved values, the default values are used.
d. Unit attention condition shall be set.

A SCSI device that implements the soft reset alternative, on the other hand, shall:

a. Attempt to complete any I/O processes that have not completed, and that were fully identified.
b. Preserve all SCSI device reservations.
c. Preserve any SCSI device operating modes (MODE SELECT, PREVENT/ALLOW MEDIUM REMOVAL commands, etc)
d. Preserve all information required to continue normal dispatching of I/O processes queued prior to the reset condition.

The order in which phases are used on the SCSI bus follows a prescribed sequence. The Reset condition can abort any phase. It is always followed by the BUS FREE phase. In addition, any other phase can be followed by the BUS FREE phase but quite a few such transitions are the results of error conditions. The normal sequence of phases as described by the standards is depicted in Figure 2-3.

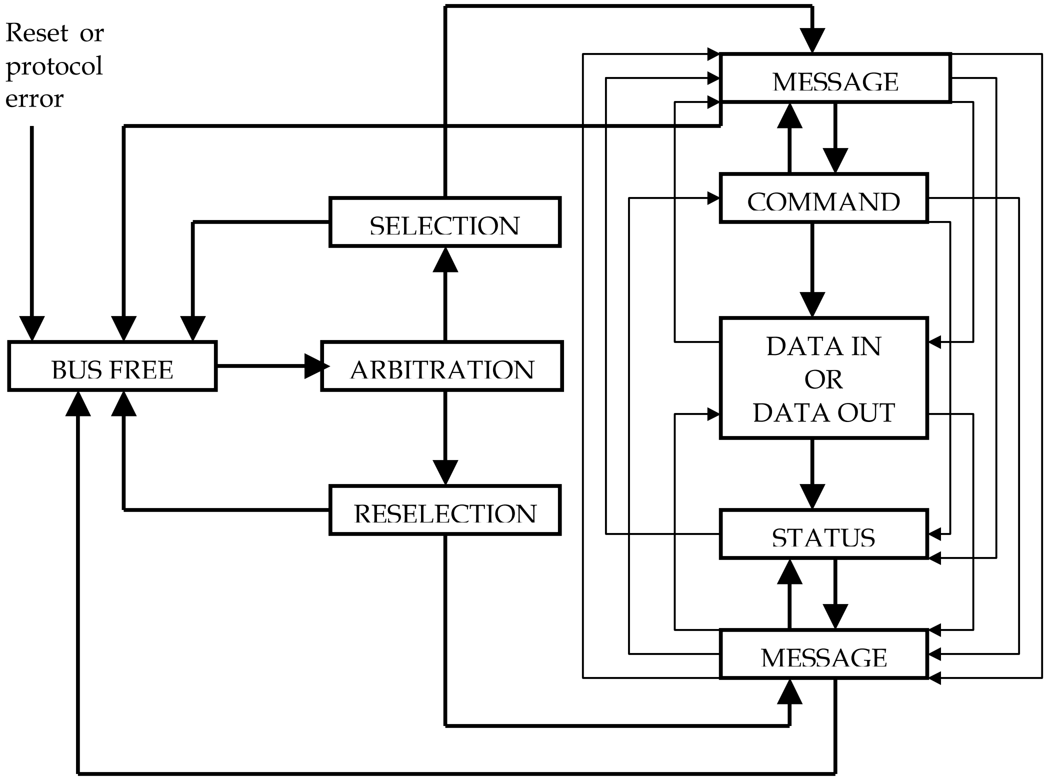

Figure 2-3: SCSI Phase Sequences

2.2 SCSI Target Model

A Target is composed of a Target Identifier, a Task Manager, and one or more Logical Units. A Target Identifier is a field containing up to 64 bits that is a SCSI device identifier for the device. Every Initiator references a Target using the Target Identifier. The process of assignment of a Target Identifier is beyond the scope of SCSI. A Task Manager is a server that controls one or more tasks in response to task management requests (discussed in a following section). There is one Task Manager per SCSI Target device. A basic Logical Unit consists of a Logical Unit Number, a Device Server and one or more Task Sets (see Figure 2-6).

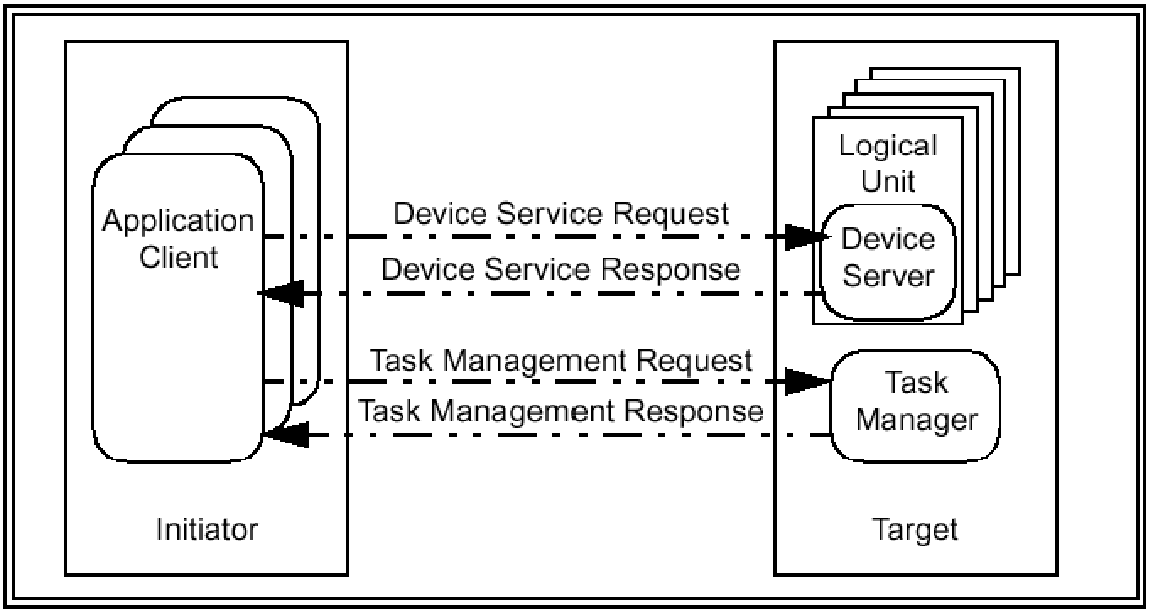

Figure 2-4: SCSI Client-Server Model

The basic structure of a SCSI sub-system is as shown in Figure 2-4. Each SCSI Target device provides two types of services, device services executed by the LUNs under the control of the Device Server and Task Management functions performed by a Task Manager. The abstract model of a SCSI Target is shown in Figure 2-5.

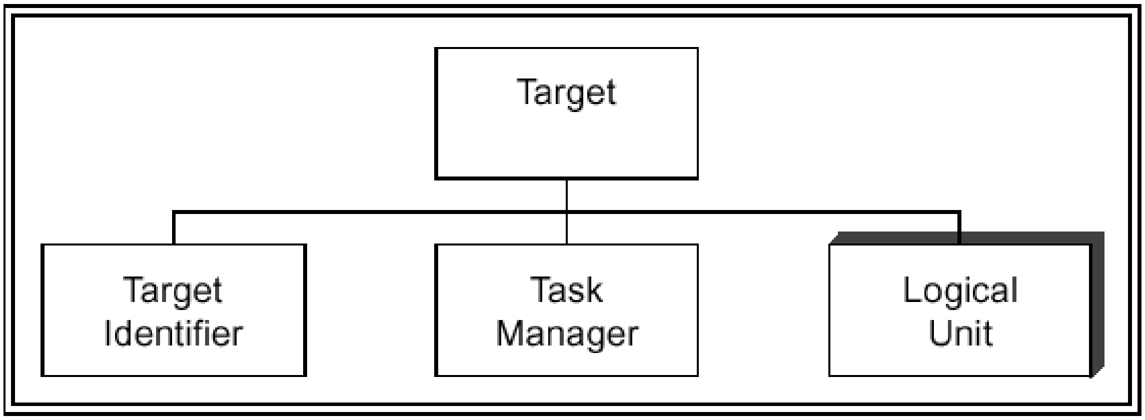

Figure 2-5: SCSI Target Hierarchy

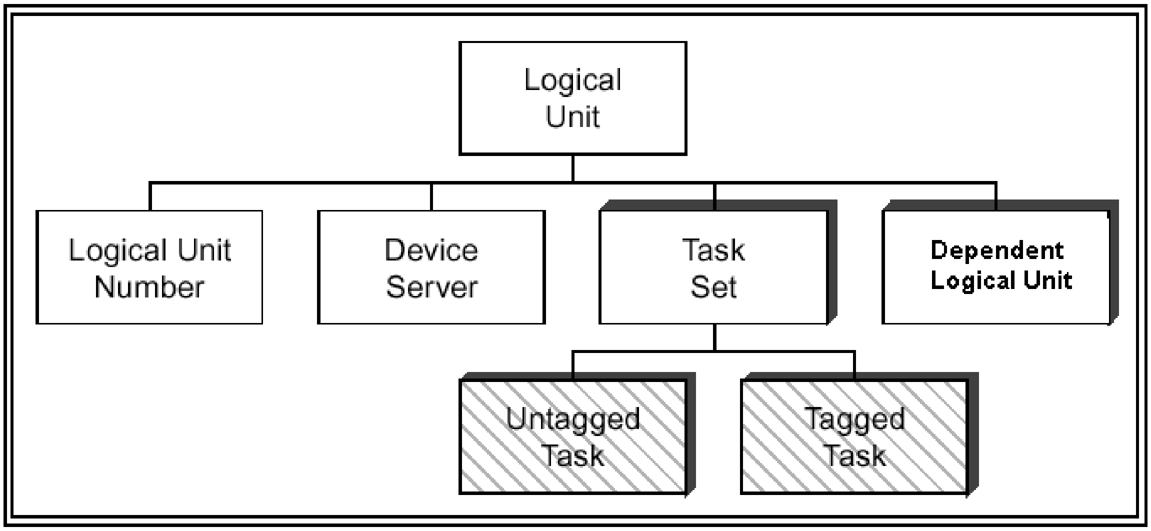

Figure 2-6: Structure of a Logical Unit

The structure of the Logical Unit is depicted in Figure 2-6. A basic Logical Unit consists of a Logical Unit Number, a Device Server and one or more Task Sets. A Logical Unit may contain a dependent Logical Unit (such as the one shown in Figure 2-6). The Logical Unit Number serves as an external identifier used by an Initiator to identify a Logical Unit within the Target. All SCSI Targets shall accept LUN 0 as a valid address to execute SCSI CDBs. The Device Server is responsible for executing the SCSI commands and manages a Task Set according to the rules established by SCSI. A Task Set is composed of at most one Untagged Task or a combination of zero or more Tagged Tasks. A Tag is an identifier (assigned by the Initiator) to uniquely identify a Task within a Task Set. Accordingly, a Task with a Tag assigned to it is referred to as a Tagged Task; otherwise, it is referred to as an Untagged Task. The composition of a Task includes a definition of the work to be performed by the Logical Unit in the form of a command or a group of linked commands. Each Task is uniquely identified by a Task Identifier.

2.3 Basics of SCSI Target Operation

Between a power-on and the time that it is selected, SCSI Targets should be able to respond with appropriate status and sense data to the TEST UNIT READY, INQUIRY, and REQUEST SENSE commands. All SCSI Targets are required to support, in addition to the above commands, the SEND DIAGNOSTIC command. These commands are used to configure the system, to test devices, and to return important information concerning errors and exception conditions.

The Target model [16] (Section 2-2) tries to minimize the amount of state information resident in a Target. The ideal Target model emphasizes maintaining information with respect to outstanding commands only. The Initiator-Target pair can be thought of in terms of a client-server pair in which the Initiator client makes requests which are responded to by the Target server. There are significant benefits in trying to minimize the amount of information that resides on the server. The most obvious one is that of trying to prevent the Device Server on the SCSI Target from becoming the bottleneck device. Furthermore, maintaining the state information almost entirely on the Initiator enables SCSI Targets to rely on the Initiator to initiate the recovery process when the execution of a SCSI command has failed. Although there are a large number of commands which are a part of SCSI, there is a smaller sub-set which represents commands that are required. This thesis aims at supporting the required set of commands.

The SCSI Architecture Model ([1], [16]) describes the transmission, processing and completion of SCSI commands in terms of remote procedure calls. Thus, for example, an application client invokes the following remote procedure in order to execute a SCSI command:

```
Service response =      Execute Command (Task Address, CDB, [Task
                        Attribute], [Data-Out Buffer], [Command Byte
                        Count], [Autosense Request] || [Data-In
                        Buffer], [Sense Data], Status);
```

where:

INPUT PARAMETERS:

Task Address: A representation of the Initiator port responsible for this command along with the Target id and LUN to which this command is transmitted

CDB: The SCSI command which is to be executed

Task Attribute: Nature of the Task (Simple, Ordered, Head of Queue, ACA)

Data-Out Buffer: Buffer containing command specific information such as data or parameter lists needed to execute the command

Command Byte Count: The maximum number of bytes to be transferred by the command

Autosense Request: Argument requesting the return of automatic sense data when the execution of the SCSI command fails

OUTPUT PARAMETERS:

Data-In Buffer: Buffer containing command specific information returned by the LUN

Sense Data: A buffer containing sense data returned by the LUN by means of an autosense mechanism

Status: A one-byte field containing command completion status

Service Response assumes one of the following values:

TASK COMPLETE: indicating that the command was completed

LINKED COMMAND COMPLETE: LUN response indicating that a Linked Command was successfully completed

SERVICE DELIVERY or TARGET FAILURE: Command execution has been ended because of a device malfunction or a service delivery failure

The SCSI protocol is thus specified in terms of function calls. The specifics of how these functional calls are implemented are not specified. For example, the `Execute Command` function call is specified. However, SCSI does not specify the actual mechanism to get the corresponding command and data (if any) to the Target. This has interesting implications. Foremost amongst these is that SCSI can be easily adapted to any interconnect (referred to as the low-level interconnect) since SCSI leaves it up to the interconnect and the protocol used by the interconnect (referred to as the low-level protocol) to decide how it (the low-level interconnect) provides the functionality requested by the remote procedure. Furthermore, SCSI only specifies the commands and the expected responses to those commands. This implies that the low-level interconnect has a great amount of freedom about how it chooses to get the required information across from the Initiator to the Target and vice versa (depending on the direction of data flow required by the command).

The Client/Server mechanism used by SCSI affects the mode of functioning of Initiators and Targets. When an Initiator (the Client) issues a command, the data buffers required by the command are already allocated (for a READ or for a WRITE – typically by the user space application on the Initiator that is responsible for the READ or WRITE) when the command is transmitted to the Target. As a result, before the Initiator issues a READ to the Target, the buffer to receive the data corresponding to the READ is already allocated. An Initiator thus, does not need flow control mechanisms for READ or WRITE type commands.

A SCSI Target, on the other hand, has to be prepared to receive commands at any point in time. For a SCSI Target, SCSI READs are not a problem. The READ command is received by the Target, the buffers are allocated, the command is executed and the buffers are filled. Since the Initiator already has the buffer space allocated to receive the data for this command, the Target is free to send the buffers across to the Initiator. Thus, a Target does not need any flow control mechanisms for READ type commands (similar to an Initiator). In the case of a SCSI WRITE, the situation is a lot different. The Target receives the WRITE command. The direction of the data flow is from the Initiator to the Target. The Target does not have know the size of the data buffer it should expect to receive from the Initiator until it receives the WRITE command. In other words, the buffers to receive this data are not pre-allocated. This implies that the Initiator cannot automatically send the data associated with the WRITE command until the Target has allocated the necessary buffer space and informed the Initiator about it. Most low-level interconnects recognize this and provide a flow-control mechanism for the transfer of SCSI data from Initiator to the Target. For example, in the Fibre Channel protocol, the Target transmits a XFER_RDY (discussed later) to inform the Initiator about what portion of the data to send (The XFER_RDY frame gives a starting LBA and the number of bytes to be sent). Thus, transmitting a response to a WRITE command involves three steps on the Target:

1. Allocating the data buffers, which is independent of the low-level protocol
2. Informing the Initiator about what data it should send, which is specific to the low-level protocol
3. The execution of the command when the data arrives, which is again independent of the low-level protocol.

This imposed order has to be accounted for in the design of a SCSI Target.

2.4 Task Management Functions

Task Management functions provide an Initiator with a way to explicitly control the execution of one or more tasks. Each Task Management function represents a service requested by the Initiator, typically used to recover a Target from what an Initiator perceives as an error condition with the Target. The SCSI Target returns a response which signifies either that the requested function was completed, or that the function was rejected or that there was a service delivery/Target failure causing the command not to be delivered. Each SCSI protocol standard defines the actual events comprising each of the above service responses. The following are the Task Management functions that have to be provided by SCSI Targets:

1. Abort Task
2. Abort Task Set
3. Clear ACA
4. Clear Task Set
5. Logical Unit Reset
6. Target Reset

These are described below. The symbol '||' in the expressions below implies that the parameter is required only if relevant.

2.4.1 Abort Task

```
Service Response = Abort Task (Task Address ||);
```

This function is required to be supported by a LUN if it supports tagged tasks and is optional for LUNs that do not support tagged tasks. The Task Manager shall abort the specified task if it exists. Previously established conditions (such as Auto Contingent Alliance, reservations etc.) shall not be affected. The Target, if it supports Abort Task, guarantees that no further responses from the Task are sent to the Initiator.

2.4.2 Abort Task Set

Service Response = **Abort Task Set (**Logical Unit Identifier ||**);**

This function is required to be supported by all LUNs. The Task Manager, upon receiving the Abort Task Set, shall terminate all the Tasks in the Task Set created by the Initiator. This is equivalent to performing a series of Abort Task requests. Previously established conditions as well as Task Sets created by other Initiators shall not be affected.

2.4.3 Clear ACA

Service Response = **Clear ACA (**Logical Unit Identifier ||**);**

This function is only to be implemented by those LUNs that accept a Normal ACA (NACA) bit value of 1 in the CDB Control Byte (Refer to Section 2.1.3 and Figure 2-2). The Initiator invokes Clear ACA to clear an auto contingent allegiance condition from the Task Set serviced by the LUN.

2.4.4 Clear Task Set

Service Response = **Clear Task Set (**Logical Unit Identifier ||**);**

This function is required to be supported by all LUNs that support Tagged Tasks and is optional for those that do not. All tasks in the appropriate task shall be aborted. No status shall be sent for any task affected by this request. A Unit Attention command shall be generated for all Initiators with aborted tasks (if any). When reporting the Unit Attention condition, the additional sense code shall be set to "Commands Cleared by Another Initiator".

2.4.5 Logical Unit Reset

Service Response = **Logical Unit Reset (**Logical Unit Identifier ||**);**

This function shall be supported by all LUNs that support hierarchical Logical Units (Refer to Section 2-2) and is optional for non-hierarchical Logical Units. To execute a Logical Unit Reset, the LUN shall:

1. Abort all tasks in its task set(s)
2. Clear an auto contingent allegiance (NACA = 1) or contingent allegiance (NACA = 0) condition, if one is present.
3. Release all reservations established using the reserve/release management method (persistent reservations shall not be affected)
4. Return the operating mode of the device to the appropriate initial conditions, similar to those conditions that would be found following a device power-on.
5. Set a Unit Attention condition
6. Initiate a Logical Unit Reset for all dependent LUNs

2.4.6 Target Reset

Service Response = **Target Reset (**Target Identifier ||**);**

This function shall be supported by all Target devices. Upon receiving a Target Reset Task Management Function, the Target device executes a Target hard reset. The definition of Target Reset events is protocol and interconnect-specific. Each SCSI Transport protocol is required to define the response to a Target Reset and the conditions under which it shall be executed. To execute a hard reset, a Target shall initiate a Logical Unit Reset for all attached LUNs (Refer to Section 2.4.5).

2.5 SCSI Error Reporting

In the event a command completes with a Check Condition status or other error conditions, SCSI requires that a Logical Unit make sense data available to the Initiator. The format, content and conditions under which sense data shall be prepared by a LUN are specified by the SCSI Architecture Model-2 (SAM-2), SCSI Primary Commands-2 (SPC-2), SCSI Block Commands-2 (SBC-2) and applicable SCSI Transport protocol standard.

Sense data may be transferred to an Initiator through one of the following methods:

1. The REQUEST SENSE command
2. An asynchronous event report
3. Autosense delivery

These three methods are discussed below.

2.5.1 The REQUEST SENSE command

The REQUEST SENSE command (shown in Figure 2-7) requests that the device server transfer sense data to the application client. The details of the appropriate response to the sense command are described in SPC-2 and have not been presented here for reasons of brevity.

Bit Byte	7	6	5	4	3	2	1	0
0	Operation Code (03h)							
1	Reserved							
2	Reserved							
3	Reserved							
4	Allocation Length							
5	Control							

Figure 2-7: The REQUEST SENSE command

2.5.2 Asynchronous Event Reporting

Asynchronous Event Reporting is used by a LUN to signal another device that an asynchronous event has occurred. The mechanism automatically returns sense data associated with the event. Each SCSI Transport protocol is required to define a mechanism for Asynchronous Event Reporting, including a procedure whereby an Initiator can selectively enable or disable asynchronous event reports from being sent to it by a specific Target. Support for Asynchronous Event Reporting is optional for a LUN.

Asynchronous Event Reporting is used to signal another device (usually an Initiator) that one of the following events has happened:

a. An error condition has occurred after command completion
b. A newly initialized device is available
c. Some other type of Unit Attention condition has happened
d. An asynchronous event has occurred

Sense data accompanying the Asynchronous Event Report identifies the condition.

2.5.3 Autosense

Autosense is the automatic return of sense data to the application client coincident with the completion of a SCSI command. The return of sense data in this manner is equivalent to an explicit command from the application client requesting sense data immediately after being notified that an ACA condition has occurred. Although inclusion of autosense support in a SCSI Transport protocol is optional, most protocols support it, primarily as it eliminates one additional transaction between an Initiator and a Target. The application client may request autosense service for any SCSI command and provided it is supported by the protocol and the LUN, the device server shall return sense data if the command completes with a status of Check Condition. If autosense is requested and the protocol or the LUN do not support autosense, the device server should indicate that no sense data was returned. The application client may then issue a REQUEST SENSE command to retrieve sense data.

CHAPTER 3

STORAGE AREA NETWORKS

3.1 Introduction

The last decade has seen a change in the way data is perceived. Data is now viewed as a commodity, access to which, in many cases, determines the success or failure of a business. Compounded by changing computing technologies and the globalization of business via the Internet, there has been a tremendous increase in storage requirements. In addition, because of the desired economies of scale achieved by 24 x 7 x 365 businesses, the windows of time available for data backup and recovery have virtually disappeared. In a nutshell, this has led to the need for cost-effective ways to ensure high data availability and reliability.

3.2 Issues with traditional Storage

The numerous manifestations of SCSI (SCSI-2, Wide SCSI, Fast SCSI, and SCSI-3) have long been the interface of choice for high-speed computer-to-storage connectivity for Windows NT and Unix users. A study by International Data Corporation estimates that the cost of managing storage is 10 times the initial cost of the storage device. Furthermore, the challenges facing "data administrators" are:

1. Deploying vital applications across a network
2. Allowing pooled data to be shared simultaneously among a large number of users, who may be widely separated from each other
3. Managing storage distributed across a wide area as effectively and efficiently as possible, without expending large sums of money or manpower
4. Supporting growing number of data-intensive applications

The problems with using traditional SCSI to solve the above problems are:

1. SCSI was designed as a point-to-point, directly attached computer-to-storage device interface. It is, therefore, ill suited for multiple host-to-storage communications.
2. The SCSI maximum of 15 devices is a restriction for companies that want to implement multiple servers to multiple storage devices networking architectures.
3. The maximum point-to-point distance allowed by SCSI is 25 m. For Ultra SCSI, this distance is reduced to 12 m. This distance limitation imposes architectural constraints on how storage is organized and distributed.
4. The point-to-point limitation of SCSI requires backup traffic from server-to-server to travel over the LAN thereby placing additional strain on the LAN.

3.3 Approaches to solving the Storage Bottleneck

The basic approach to solving the problem of the storage bottleneck deals with separating massive direct-access storage devices from the computer systems that access it. There are two mechanisms to achieve this. These are:

1. Network Attached Storage (NAS)
2. Storage Area Networks (SAN)

The NAS approach was the more traditional approach. It intercepts the communication between an application client and a storage device. Communication between the host and the remote storage is via a special file system that uses the traditional network protocol stack. Access to the storage device is controlled by means of the NAS Server. Communication between the NAS server and the storage device is by traditional channel protocols such as SCSI. The communication between the NAS Server and the host (NAS Client) uses a special (typically, messaging) protocol. The unit of access between the NAS Server and the NAS Client is a file. The NAS File System therefore has to deal with such issues as integrity, security, and consistency at the level of a file. An example of a NAS is the Network File System (NFS) from Sun. The problem with this approach is a lack of scalability. In addition, this approach requires a lot of manual intervention in terms of making the connected systems aware of any new storage resources or configuration changes.

3.4 The Storage Area Network Approach

The concept that emerged out of the NAS approach was the need to have a block level protocol controlling access to the data storage units. Another key concept that simultaneously emerged was the creation of a network that was solely dedicated to the task of managing and controlling access to a set of storage devices. The unit of access across this Storage Area Network (SAN) is a block. SCSI provides an interoperable, high-performance block level protocol. Any new block level protocol would have had to deal with many of the same problems that SCSI had already dealt with. Thus, instead of reinventing the wheel, current SANs use the SCSI protocol as the block level access protocol. However, to counter the limitations of SCSI, most notably distance, SCSI is transported over a different low-level protocol that controlled access to the shared medium. This is the concept behind SCSI Transport Protocols. The visualization of the SAN in relation to the traditional LAN has been shown in Figure 3-1.

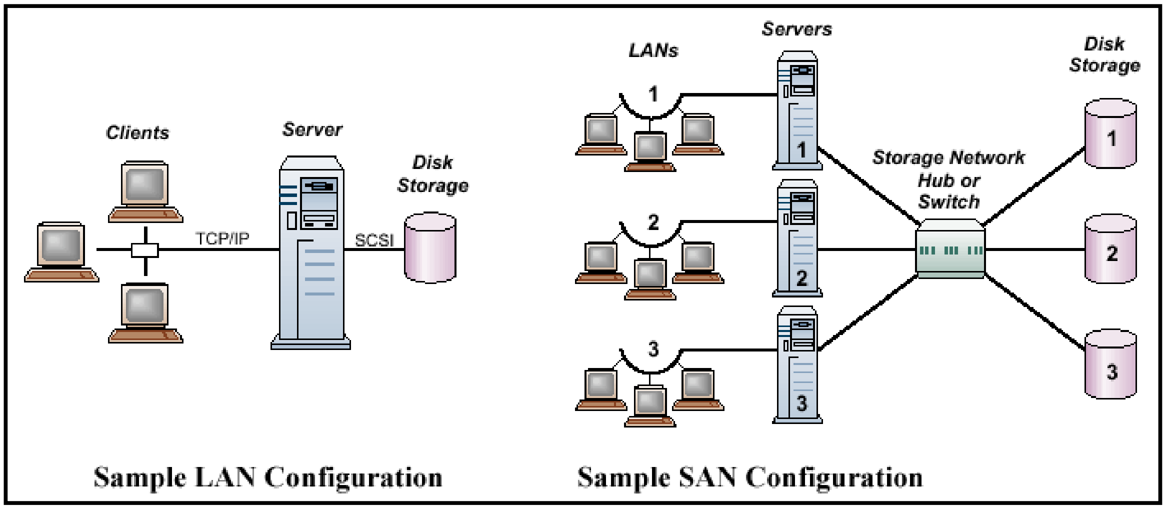

Figure 3-1: Concept of Storage Area Networks

A SAN is thus a dedicated network that connects different kinds of storage – such as tape libraries and RAID systems - to servers. Because stored data does not reside on any network server, server resources can be utilized for other purposes, increasing network capacity. The benefits of such an approach to storage are listed below:

1. *Storage Consolidation*: SANs enable consolidation of storage into a shared, heterogeneous, highly available environment. This is compared to the distributed "islands" of storage that foster unmanaged and unplanned growth.
2. *Improved Management*: The consolidated storage means improved management of storage. SANs help simplify administration and reduce management costs. Separating storage and server functions allows network administrators to view these two functions independently, and to divide bandwidth optimally between them.
3. *Availability*: A SAN can provide the ability to access the same storage over multiple paths. This implies that if one server or an interconnecting path fails, the user can access data through other paths. This is especially important for backup operations that rely heavily on successful operation.
4. *Scalability*: Since there is separation between the servers and storage, all the individual components of a SAN scale well. As additional secondary devices are added to the SAN, they too become accessible from any server within the network. Consequently, an organization can start with the system capacity it currently needs and add storage as and when needed in the future.
5. *Bandwidth*: SANs enable effective use of the bandwidth. Since, storage is tied to a network, and not to a server, the server can operate faster, and at a lower level of utilization. This implies that servers can operate faster and therefore, respond to requests quicker.

An analysis of the different SCSI Transport protocols is presented in Chapter 4.

CHAPTER 4

SCSI TRANSPORT PROTOCOLS

4.1 Introduction

Chapter 3 presented the limitations of SCSI and how the concept of SANs has evolved around SCSI but using a different low-level interconnect to transmit SCSI over greater distances and to a greater number of devices than is allowed by traditional SCSI. The SCSI standards organization (http://t10.org) has defined a set of protocols which allow SCSI to be transmitted over different low-level interconnects. These are collectively called the SCSI Transport protocols. With the growing importance of SANs over the past couple of years, several approaches have been proposed to the IETF and to T10. Examples of such approaches would be:

1. Fibre Channel (FC)
2. Scheduled Transfer Protocol (STP)
3. SCSI Encapsulation Protocol (SEP) – Proposed by Adaptec
4. Internet SCSI (iSCSI) – Proposed by IBM originally – now on the IETF Standards Track
5. Storage Over IP (SoIP) – Proposed by Nishan Systems – now on the IETF Standards Track

The hierarchy of these standards/proposals is shown in Figure 4-1. Some of these options are discussed.

4.2 Fibre Channel

Fibre Channel is a serial, high-speed data channel that provides logical bi-directional service between two ports. Fibre Channel is directed towards unifying LAN and channel communications by defining an architecture with enough flexibility and performance to satisfy both sets of requirements. Fibre Channel presents one solution to achieving most of the requirements of the SAN. It essentially assumes an error free network. Fibre Channel can provide data access at 1 Gbps and 2 Gbps with 10 Gbps expected in the future.

4.2.1 Fibre Channel basics

Individual nodes on a Fibre Channel network are referred to as ‘N_Ports’. Each N_Port resides on a hardware entity. A port capable of providing switching capabilities is referred to as an F_Port. Fibre Channel is structured as a set of hierarchical functions as shown in Figure 4-2. Each of these functions is described as a level.

The Physical interface (FC-0) consists of transmission media, transmitters, and receivers and their interfaces. The Physical interface specifies a variety of media, and associated drivers and receivers capable of operating at various speeds.

The FC-1 level specifies the 8B/10B transmission code that is used to provide DC balance of the transmitted bit stream, to separate transmitted control bytes from data bytes and to simplify bit, byte and word alignment. In addition, the coding provides a mechanism for detection of some transmission and reception errors.

The FC-2 Level is the signaling protocol specifying the rules, and provides mechanisms needed to transfer blocks of data end to end. FC-2 defines functions and facilities available for use by the upper levels.

FC-3 provides a set of services that are common across multiple N_Ports of an FC node. This level is not yet well defined, due to limited necessity for it, but the capability is provided for future expansion of the architecture.

The FC-4 level provides mapping of Fibre Channel capabilities to pre-existing Upper Level Protocols, such as IP, SCSI, ATM, etc.

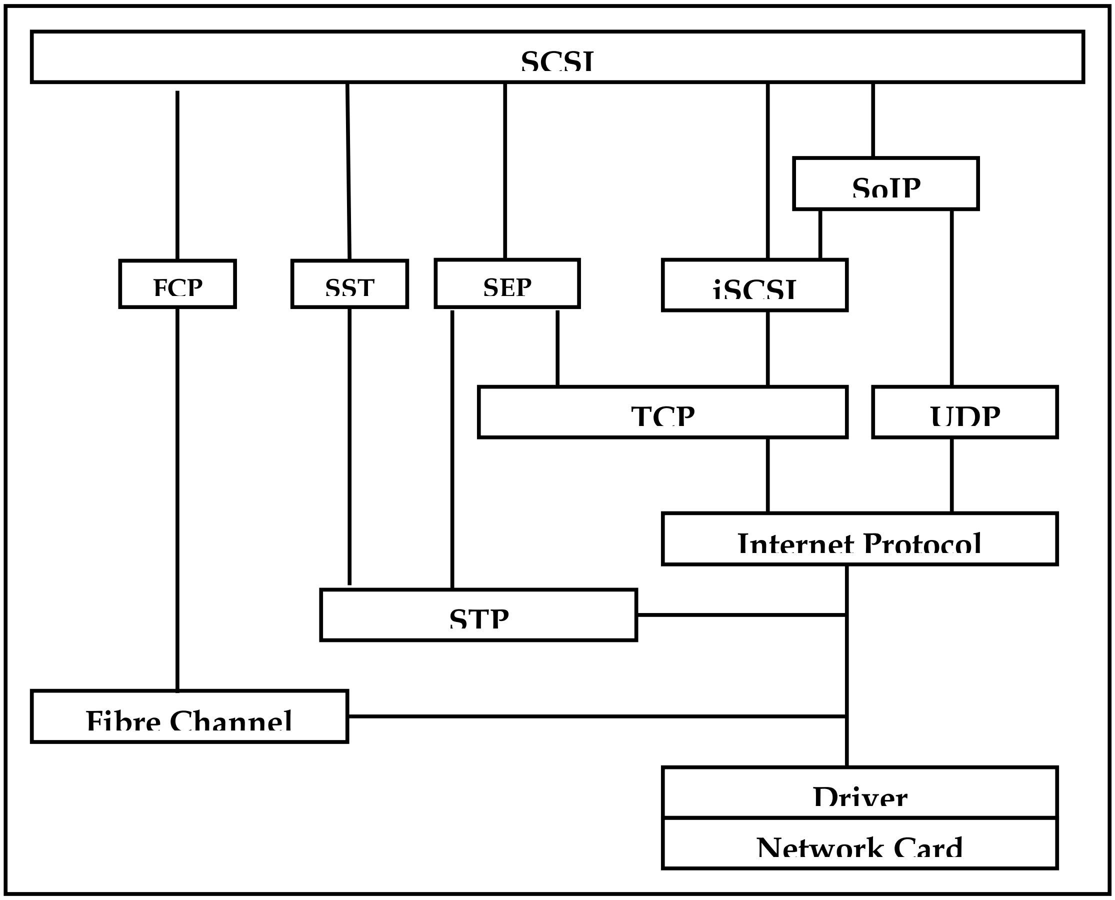

Figure 4-1: Protocols providing a mapping of SCSI over different protocols

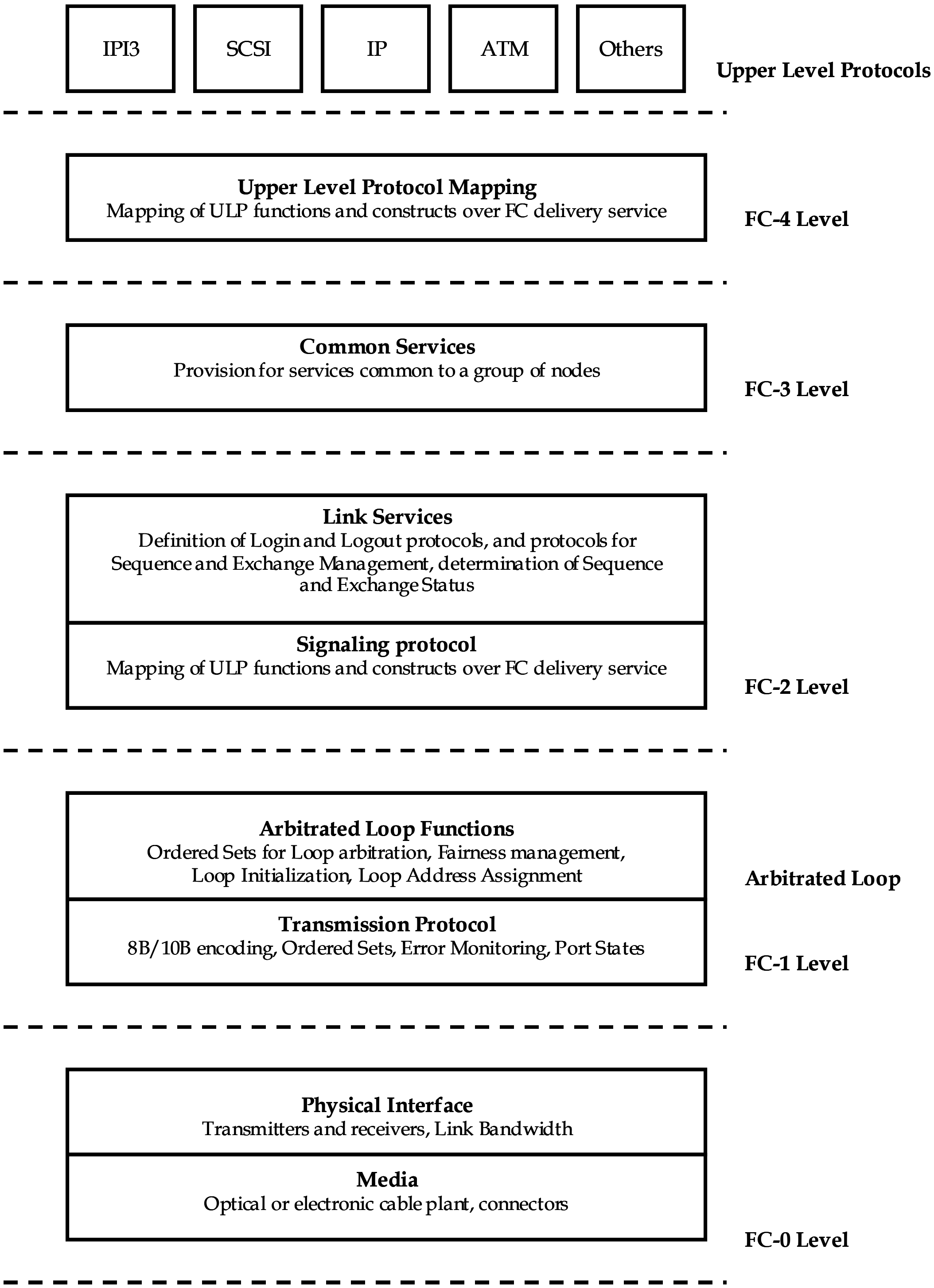
IPI3
SCSI
IP
ATM
Others
Upper Level Protocols
Upper Level Protocol Mapping
Mapping of ULP functions and constructs over FC delivery service
FC-4 Level
Common Services
Provision for services common to a group of nodes
FC-3 Level
Link Services
Definition of Login and Logout protocols, and protocols for Sequence and Exchange Management, determination of Sequence and Exchange Status
Signaling protocol
Mapping of ULP functions and constructs over FC delivery service
FC-2 Level
Arbitrated Loop Functions
Ordered Sets for Loop arbitration, Fairness management, Loop Initialization, Loop Address Assignment
Arbitrated Loop
Transmission Protocol
8B/10B encoding, Ordered Sets, Error Monitoring, Port States
FC-1 Level
Physical Interface
Transmitters and receivers, Link Bandwidth
Media
Optical or electronic cable plant, connectors
FC-0 Level

Figure 4-2: Fibre Channel Functional Levels

The FC-1, FC-2, and FC-3 levels are typically implemented in hardware whereas the FC-4 level is implemented in software.

A Fibre Channel network can be set-up in three basic topologies:

1. *Point-to-Point topology*: Here two N_Ports are directly connected to each other. It is a non-blocking connection.
2. *Fabric Topology*: A network of multiple N_Ports is connected to a switched network by means of an F_Port. The basis for this non-blocking network is to take advantage of the fact that devices cannot sustain high rates of transfer over long periods. Such a configuration allows for fewer interconnects and makes the Fibre Channel network extensible.
3. *Loop Topology*: The Loop topology consists of a maximum of 127 participating ports on one Loop. There is one link bandwidth that is shared between all ports. The Loop topology provides for a blocking and non-meshed network. Ports participating on the Arbitrated Loop are referred to as L_Ports (NL_Ports and FL_Ports).

An implementation of a Fibre Channel network can consist of a combination of these topologies. This is shown in Figure 4-3. Most devices tend to support multiple topologies. Ports on such devices are referred to as Fx_Ports (if they provide Fabric functionality) or Nx_Ports otherwise.

Fibre Channel supports five Classes of Service. These Classes of service are distinguished primarily by the methodology with which the communication circuit is allocated and retained between the communicating Nx_Ports and the level of delivery integrity required for an application. Classes 1, 2, and 3 are topology independent. If the Fabric is not present, the service is provided as a special case of point-to-point. Classes 4 and 6 require functionality outside of the participating Nx_Ports. Fabrics and Nx_Ports are not required to support all Classes of service.

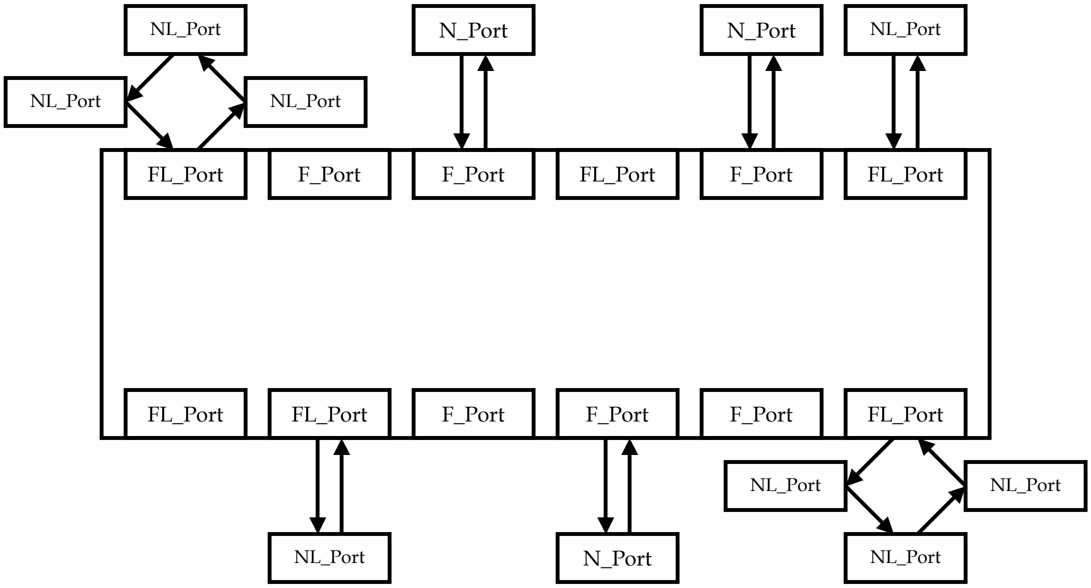

Figure 4-3: A possible implementation of a Fibre Channel Network

1. *Class 1 service - Dedicated Connection:* Class 1 is a service that establishes Dedicated Connections. Once established, a Dedicated Connection is retained and guaranteed by the Fabric. This service guarantees maximum bandwidth available between two N_Ports across the established connection. In Class 1, frames are delivered to the destination N_Port by the Fabric in the same order as they are transmitted by the source N_Port.
2. *Class 2 service - Multiplex:* Class 2 is a connectionless service multiplexing frames at frame boundaries. Multiplexing is supported from a single source to multiple destinations and to a single destination from multiple sources. There are no guarantees for in-order delivery of frames. Furthermore, there is notification of delivery or failure to deliver.
3. *Class 3 service - Datagram:* Class 3 is a connectionless service with unacknowledged delivery. There is no notification of delivery or failure to deliver and any error recovery is performed by the Upper Level Protocol (ULP) level. Any acknowledgement of Class 3 service is left up to and determined by the ULPs. The transmitter transmits Class 3 Data frames in sequential order within a given Sequence. However, there are no guarantees for in-order delivery of frames. In Class 3, the Fabric is expected to make a best effort to deliver the frame to the intended destination and does not issue a busy or a reject frame to the source N_Port if unable to deliver the frame.
4. *Class 4 service - Fractional Bandwidth:* Class 4 is a service that uses a virtual circuit established within a Fabric and between two communicating Nx_Ports to transmit frames to each other using a fabric-managed fractional bandwidth allocation protocol. This service requires a Fabric. The transmitter transmits Class 4 Data frames in a sequential order within a given Sequence. In Class 4, frames are delivered to the destination N_Port by the Fabric in the same order as they are transmitted by the source Nx_Port. The Fabric or destination Nx_Port guarantees notification of delivery or failure to deliver in the absence of link errors. In case of link errors, notification is not guaranteed since the Source_Identifier (S_ID) may not be error free.
5. *Class 6 - Multicast Connection:* Class 6 allows an Nx_Port to establish simultaneous Dedicated Connections with multiple Nx_Ports. Once established, these Dedicated Connections are retained and guaranteed by the Fabric. This service guarantees maximum bandwidth available from the source N_Port to each destination N_Port across the established connections. The effective bandwidth of any Class 6 connection is multiplied by the number of destination Nx_Ports. Class 6 is inherently unidirectional. Data flows only from the source Nx_Port to the destination Nx_Ports. All destination Nx_Ports respond with the appropriate Link_Response frames to a Multicast Server. The Multicast Server collects the Link_Response frames and returns a single Link_Response frame to the source Nx_Port. Frames are delivered to the destination Nx_Ports by the Fabric in the same order as they are transmitted by the source Nx_Port. This service requires a Fabric.

The Login protocol allows devices to communicate the Classes of Service that they support to each other. Class 3 is the most common Class of Service.

4.2.2 SCSI over Fibre Channel

Fibre Channel has been designed to implement the prerequisites required by the SCSI Architecture Model (SAM). Thus, it provides a logical means for extending the SCSI bus. The basic synergy between Fibre Channel and SCSI is that Fibre Channel allows SCSI-3 compliant devices to communicate over a reliable Fibre Channel interface over a greater distance and at a greater throughput than would have been possible by the use of SCSI. Thus, Fibre Channel provides a reliable interconnect with SCSI serving as the Upper Level Protocol.

The mapping of SCSI into Fibre Channel is defined by the Fibre Channel Protocol for SCSI standard (X3.269 – 1995 revision 12) – also referred to as SCSI-FCP. Four kinds of functional management functions are defined by SCSI-FCP:

- Device Management
- Task Management
- Process Login/Logout Management
- Link Management

The FCP device and the task management protocols define the mapping of the SCSI functions defined in SAM to FC-PH. The SCSI-FCP is based on a two-level paradigm. The SCSI I/O Operation is mapped into an Exchange. The Request and Response primitives required by the I/O Operation are mapped into information units each of which may be contained within a Sequence. Link control is performed by the standard FC-PH protocol. This mapping is shown in Figure 4-4.

SCSI function	**FCP equivalent**
I/O Operation	Exchange
Request/Response Primitives	Sequence
Command service request	Unsolicited Command IU (FCP_CMND)
Data delivery request	Data descriptor IU (FCP_XFER_RDY)
Data delivery action	Solicited data IU (FCP_DATA)
Command service response	Command status IU (FCP_RSP)

Figure 4-4: Functional Mapping between SCSI and Fibre Channel

An application client begins an FCP Operation when it provides to the FCP a request for an Execute command service. A single request or a set of linked requests may be presented to the software interface of the FCP. Each request contains all the information necessary for the execution of one SCSI command, including the local storage address and characteristics of the data to be transferred by the command. The FCP then uses the services provided by Fibre Channel in order to execute the command.

The SCSI Initiator for the command starts an exchange by sending an unsolicited command Information Unit (IU) containing the FCP_CMND payload, including some command control flags, addressing information, and the SCSI command descriptor block (CDB). In particular, the FCP_CMND payload is the Execute Command service request and starts the FCP I/O operation. The exchange is identified by its fully qualified exchange identifier which is used exclusively for all IUs associated with the execution of the command request.

Upon receiving a SCSI command, the SCSI Target interprets the command. If a SCSI WRITE is requested, the SCSI Target determines the amount of data transfer required and allocates the necessary buffers to receive the data. It then transmits a data descriptor IU containing the FCP_XFER_RDY payload to the Initiator to indicate which portion of the data is to be transferred. The SCSI Initiator then transmits a solicited data IU to the Target containing the FCP_DATA payload requested by the FCP_XFER_RDY payload. If, on the other hand, the SCSI command received described a SCSI READ operation, the SCSI Target determines the amount of data transfer requested and allocates the necessary buffers. The data is then transferred using a solicited FCP_DATA IU to the Initiator. Data delivery requests continue until all data described by the SCSI command is transferred in either case. Thus, FCP_XFER_RDY are used to transfer data during SCSI WRITE operations but not during SCSI READ operations.

After all the data has been transferred, the device server transmits the Execute command service response by requesting the transfer of an IU containing the FCP_RSP payload. That payload contains the SCSI status and if an unusual status has been detected, the SCSI REQUEST SENSE information and the FCP response information describing the condition. The command status IU terminates the command. The SCSI logical unit determines if additional commands will be performed in the FCP I/O Operation. Upon determining that the command executed is the last or the only one to be executed in the FCP I/O Operation, the FCP I/O Operation and the exchange are terminated.

When the command is completed, returned information is used to prepare and return the Execute command service confirmation to the software that requested the operation. The returned status indicates whether the command was successful. The successful completion of the command indicates that the SCSI device performed the desired operations with the transferred data and that the information was successfully transferred to or from the SCSI Initiator. If on the other hand, command execution was unsuccessful, then the required error information can be provided according to a defined protocol.

If the command is linked to another command, then the FCP_RSP contains the proper status indicating that another command will be executed. The Target presents the FCP_RSP in an IU that allows command linking. The Initiator continues the same exchange with an FCP_CMND IU, beginning the next SCSI command.

FCP allows full use of Fibre Channel and the Classes of Service provided by it as well as the different topologies allowed by it.

4.3 SCSI over Ethernet

The development of Fibre Channel led to the birth of the concept of Storage Area Networks. The concept of SANs extends beyond the use of Fibre Channel as a lower level interconnect. The next logical step is to try and rationalize existing network infrastructure to serve the storage needs of organizations. Towards this end, there are several proposals to try to transmit SCSI over the existing Ethernet infrastructure. Most of these protocols use the TCP/IP protocol. Thus, these protocols are extensible to all link-level technologies that support TCP/IP.

The intrinsic issues that network-based technologies need to solve are those of reliability and Target identification. The latter translates into routing. This is solved by using the IP layer for routing. One can visualize a network cloud where Target devices are identified by an IP address. The reliable delivery is solved by having a reliable connection-oriented protocol on top of IP such as that provided by TCP. The advantages of this approach are manifold. The first issue is the ability to use existing network protocol stacks for transmission of data. The second advantage is that these protocols are well understood. The effects and implications of time-outs and window sizes have been studied in detail. Thus, a vast knowledge base is already available when considering the implementation of any protocol on top of the TCP/IP protocol suite. The use of Ethernet also keeps the costs low as it is the most widely deployed LAN technology and Ethernet components enjoy the benefits that come with the high volume. The following sections discuss the proposed protocols which transmit SCSI over Ethernet.

4.3.1 SCSI Encapsulation Protocol

The SCSI Encapsulation Protocol (SEP) was developed by Adaptec Inc [9]. This protocol was primarily intended as a proof of concept demonstration to be able to use network technology to create a high performance storage subsystem. The SEP Protocol assumes an underlying reliable session protocol such as TCP/IP. The protocol uses TCP/IP to allow the sharing of targets across various hosts. (Refer to Figure 4-5)

The SEP architecture is designed to address the needs of the desktop, small server, campus wide and larger storage area networks. The 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. In order to achieve similar CPU utilization metrics with host adapters using the SEP protocol, the SEP protocol envisages the processing of the entire TCP/IP protocol stack on the host adapter itself rather than using CPU cycles.

SEP operates at the session layer of the network protocol stack, just above the TCP/IP layer. It relies on the transport layer for correct delivery, and concentrates on multiplexing SCSI command, data, and status information. Because TCP/IP can be used on any Link and Physical layer, IP-storage (IPS) implemented with SEP can provide SCSI service over any network which has adequate performance.

Traditional SCSI (using the parallel bus interconnect) identifies the type of information (phase) on the data lines with a set of encoded control lines. The mapping of this to a serial bus can be efficiently done by sending a type code at the beginning of the new phase, which applies until the next phase. This type code is part of a header that is appended to the beginning of data from each phase. This header also contains the SCSI tag, so that data and status segments can be matched to the correct command. To separate SEP packets from each other, a packet length placed in the header is used to read in the bytes until the next header.

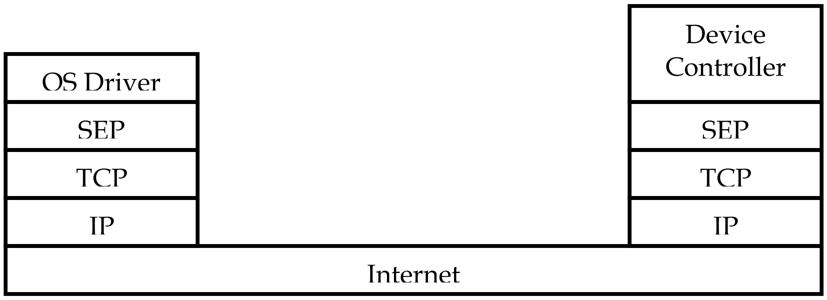

Figure 4-5: Relationship between SEP and the TCP/IP Protocol

The underlying transport layer provides minimal addressing and flow control for each of its instances. This may require separate connections for each command. However, there is an overhead associated with establishing the connection. Furthermore, SCSI places an ordering requirement on each command going out to a specific LUN. SEP uses this as a logical point of separation. Different commands to the same LUN travel over the same TCP connection. Multiple connections may be used when accessing different LUNs on the same Target. The requirement for using different connections for different LUNs results in SCSI frames being transmitted without LUN information in the SEP header. The TCP connection can be established and maintained either continuously (for devices accessed continuously such as disk drives) or on-a per use basis (for devices accessed sporadically such as a printer).

SEP encapsulates SCSI command, status, message and data information as SEP segments with a SEP header at the beginning. The SEP header is a fixed length of eight bytes as depicted by Figure 4-6.

All SEP segments are padded with zeroes to four byte boundaries. 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 SEP protocol involves establishing a connection with an IP address by transmitting a 'Connect and Negotiate' message. This message is specific to a LUN. The 'Negotiation Response' indicates that a connection has been established. The host then transmits various SCSI commands using the 'Simple Tagged Command' SEP header to the Target. The Target then responds with either 'SCSI Data', 'SCSI Status' or a 'SCSI Message'. Flow control is managed by using the SCSI Data Request and in this sense is very similar to the way Fibre Channel implements flow control. The Target controls the flow of data from the host. The Target originally allows the host to transmit a limited amount of data to it without requiring checking if the Target has buffers available. This is decided by the 'Negotiation Response' transmitted by the Target. The host then has to wait until it receives a 'SCSI Data Request' from the Target before it can transmit any more data.

SEP does not deal with error recovery in the case of a dropped connection. SEP assumes that error conditions are dealt with by SCSI and by TCP/IP. This leads to a couple of issues with regard to the state of the file system on the Target when a connection is lost.

0 → **31**

Word 0	Type	Flags	Tag
Word 1	Reserved		SEP Segment Length

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

Flags:

0x80:	Command Complete
0x40:	Good Status

Figure 4-6: SCSI Encapsulation Protocol header

4.3.2 Internet SCSI

The Internet SCSI (iSCSI) protocol [7] was proposed by a group of companies including IBM, Cisco Systems, Hewlett-Packard, SANGate, Adaptec, and Quantum. The standard is currently being developed under the aegis of the Internet Engineering Task Force (http://www.ietf.org). The iSCSI protocol, similar to SEP, is designed primarily with TCP as the underlying level.

Communication between an Initiator and a Target occurs over one or more TCP connections. The TCP connections are used for sending control messages, SCSI commands, parameters and data within iSCSI Protocol Data Units (iSCSI PDU).

The iSCSI protocol requires a login in order to enable a TCP connection for iSCSI use. This login can be used for authentication and authorization. Connections from an Initiator to a given Target are part of an iSCSI session. There can be multiple iSCSI sessions between an Initiator and a Target. The targets listen on a well-known TCP port for incoming connections. The Initiator begins the login process by connecting to that well-known TCP port and transmitting a "login" message. After authorizing and authenticating the Initiator, the Target transmits an "accept login". The login establishes a session ID. Other parameters may be negotiated using the highly extensible Text Command message that allows arbitrary key:value pairs to be passed. Once this exchange has been completed, the iSCSI session is said to be in the iSCSI full feature phase.

In the iSCSI full feature phase, the Initiator may send SCSI commands and data to various LUNs on the Target by wrapping them in iSCSI messages that go over the established iSCSI connections. For SCSI commands that require data and/or parameter transfer, the (optional) data and the status for a command must be sent over the same TCP connection that was used to deliver the SCSI command. The Initiator and Target may interleave unrelated SCSI commands, their SCSI Data and responses, over the session. Outgoing SCSI data (Initiator to Target – user data or command parameters) is sent as either unsolicited data or solicited data.

The iSCSI protocol identifies targets using a URL type name of the format:

scsi://<domain-name>[/modifier]

The name used to connect will be optionally included in the login in order to enable the Target to present different views. This is the Target Acquired Name (TAN). The domain names can follow the IPv4 or the IPv6 naming convention. The iSCSI message header used in draft 3 of the proposed IETF standard to encapsulate SCSI commands and data is shown in Figure 4-7.

The iSCSI protocol makes some effort to deal with 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 means to re-read the data from a device server. In other words, the iSCSI protocol makes the assumption that in spite of a protocol error on the iSCSI side, the Target has some means to access the data. (It must be pointed out that one potential mode of operation anticipated by iSCSI is where access to the disk (i.e., device server) will be through a "gateway" or a "bridge" that will return the data to the Initiator after converting it to the iSCSI data format.) The transmission or absence thereof, of 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 with 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 Reference Number (CmdRN). The latter is needed only when the commands were not acknowledged. If acknowledged, a new CmdRN 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.

Several issues concerning configurable options, security, error handling and recovery are currently being discussed by the iSCSI IETF Working Group.

Word	Byte 3	Byte 2	Byte 1	Byte 0
0	Opcode	Opcode-specific fields		
1	Length of the data following the 48 byte header			
2 – 3	Logical Unit Number (LUN) or Opcode-specific fields			
4	Initiator Task Tag			
5 – 12	Opcode-specific fields			

Opcode:

0x80:NOP-In message
0x81:SCSI Response
0x82:SCSI Task Management Response
0x83:Login Response
0x84:Text Response
0x85:SCSI Data (for READs)
0x90:Ready To Transfer
0x91:Asynchronous Event
0xEF:Reject

Opcode:

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

Figure 4-7: Generic iSCSI Message Header (from iSCSI Draft 3)

4.3.3 Storage over IP (SoIP)

Nishan Systems (http://www.nishansystems.com) proposed the concept of Storage over IP (SoIP). This paradigm is similar to and in some sense, parallel to the concept of iSCSI. While SoIP defines the concept, the actual protocol proposed by Nishan to implement this paradigm is Metro FCP (mFCP) [8]. Although this protocol has not been implemented as a part of this thesis, a description of the concept behind the protocol is presented below.

mFCP as the name suggests is a protocol designed to transport Fibre Channel Protocol for SCSI (FCP) over metro- and local-scale IP networks. mFCP tries to use the existent networking protocols and in that sense combines IP and FCP. It uses IP as the addressing and routing layer and uses the FCP layer to transmit SCSI. The mFCP protocol essentially defines an encapsulation of FCP over IP. All FCP mechanisms are transported natively over IP between Fibre Channel and SCSI storage devices. mFCP uses the UDP transport protocol to facilitate high performance, and assumes that reliability and flow control will be handled by the encapsulated Fibre Channel protocol. mFCP's primary objective is to allow interconnection and networking of existing Fibre Channel devices over an IP network.

mFCP achieves this objective by leveraging FCP mechanisms already in use in storage products, and statelessly mapping these to UDP/IP. Existing FCP-based Fibre Channel products can now use mFCP to internetwork over an IP-based network. mFCP achieves high performance by forwarding FCP information units directly between FCP end nodes without the delays introduced by conventional storage routers.

The details of the protocol have not been made completely public. As a result, the details of the protocol have not been presented.

CHAPTER 5

SCSI TARGET EMULATOR

5.1 Introduction

As seen in the previous chapters, there are several proposals to extend the length of the SCSI bus. In the evaluation of any networking technology that attempts to supplant SCSI, the resources needed are two-fold. The first of these is something that can perform the functionality of a SCSI Initiator over the desired interconnects. The pieces for this are already in place in terms of the SCSI mid-layer in the Linux kernel. The driver needed to interface with the hardware has to provide the functionality requested by the SCSI mid-layer. The driver in this case, is responsible for translating the function required by the SCSI Initiator mid-level in terms of the protocol used by the low-level interconnect. The functions provided by the Initiator use the protocol required by the interconnect to convert the received SCSI command into a correctly formatted frame or packet that can be transmitted over the interconnect.

The other necessary resource is something that can provide the necessary functionality of a Target. As Targets tend to be mostly devices like hard drives or tape drives with their micro-code being resident on the device itself, the SCSI mid-layer, as it presently exists in the Linux kernel, does not cater to the needs of a Target and the way it is required to handle instructions in accordance with the relevant SCSI standards. From this point of view, it is highly desirable to have a generic way in which a driver written to perform Target functionality can interface with the Linux kernel. This functionality is what can be potentially provided by a SCSI Target Emulator. When implemented in the Linux kernel, this concept parallels that of the SIML and can be thought as the SCSI Target mid-level (STML).

A SCSI Target device receives the SCSI commands from an Initiator. The process of how the command delivery occurs and how data is transmitted between the Initiator and the Target is implemented in an interconnect-dependent manner. This command is handled by an interconnect-specific low-level front-end Target driver (FETD). It will strip off headers introduced by the protocol used on the interconnect and hand off the SCSI command and data (if any) to the 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 must also be able to respond to the error handling facilities provided by SCSI. This abstract overview is presented in the Figure 5-1. A description of the nature of the interaction and a discussion of the API needed to handle these interactions is provided in the following sections.

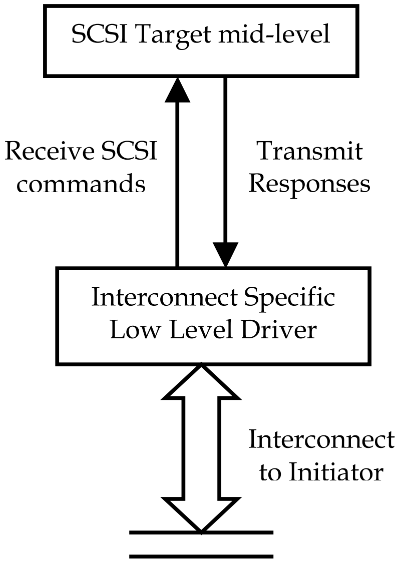
SCSI Target mid-level
Receive SCSI commands
Transmit Responses
Interconnect Specific Low Level Driver
Interconnect to Initiator

Figure 5-1: Abstract Overview of the functionality of a Target Emulator

5.2 Preliminary Design Issues

There are several options about how exactly the Target Emulator can be implemented. These options are discussed in this section. Clearly, the most obvious mode of functioning is when the Target Emulator functions exactly like a driver. It could potentially function entirely internal to the kernel. The issue then is of defining what processing a command actually means. The most simplistic implementation would involve being able to transmit frivolous data when READs are requested and accepting data for WRITEs without actually doing anything with that data. The second potential implementation is where the READs and WRITEs can be directed to a specific device (e.g., a SCSI or an IDE drive in the system). Another potential implementation is when the Target Emulator functions in a manner similar to the TCP/IP stack. Data is either transmitted to the user space or requested from the user space. The idea is to give the user flexibility over what to do with the actual data. The level of granularity can be varied depending upon the needs of the user. As a project objective, it is desirable to implement all of these modes of operation.

The second aspect of this project involves actually using the Target Emulator implementation. Although the actual Target needs to be independent of the underlying interconnect, in order to demonstrate the functionality of the Target Emulator, it is proposed that a couple of low level drivers be implemented with the necessary functionality. The low-level interconnects for the purposes of this project are selected to be Fibre Channel, SEP and iSCSI.

Apart from creating a generic interface for Target drivers, the Target Emulator has some other potential applications. One potential use is in testing. Depending upon the level of granularity (i.e., does the driver have frame/packet level control over the card or data level control) that the low-level driver has over the SCSI operation, it may become possible to transmit incorrect frames, or frames out of order. This functionality is desirable in a tool to test the SCSI functionality implemented on the Initiator. Another potential use is in being able to provide an interface to share files off a disk. The Target Emulator could function as a front-end for a large number of physical disks. This makes it an ideal location to implement policies about sharing partitions, volumes and files. It could also potentially be used to set policies about mirroring, path-redundancy and data access. The Target Emulator could be used to as an interface to a file sharing application (see Figure 5-2).

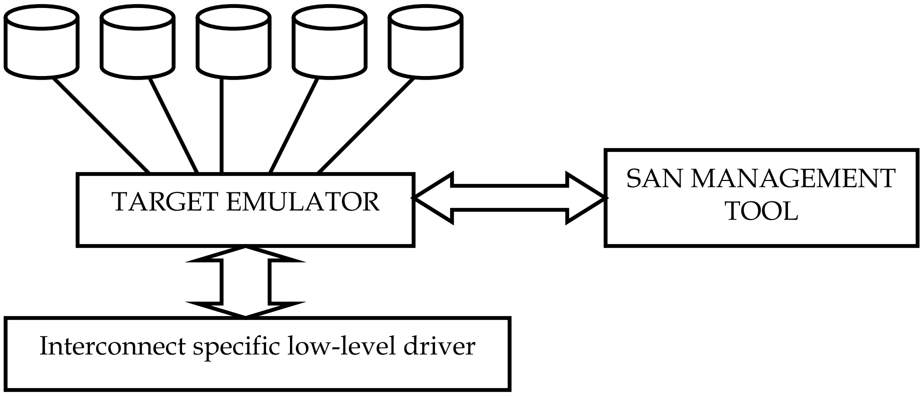

Figure 5-2: Functionality provided by the Target Emulator

5.3 Description of API needed

This section gives an approach to designing the STML. Since the functionality will need to mirror that offered by the SIML, this section also takes a look at the SIML design. The details of how the Target Emulator is actually implemented in user space and in kernel space are discussed in Chapter 6 and Chapter 7, respectively.

The Linux SCSI subsystem can be classified into three levels. The three levels are depicted in Figure 5-3. In addition to this, the Linux SCSI subsystem also consists of a bottom-half handler that in effect is a thread with an infinite loop responsible for processing the responses received for SCSI commands. The key to the design lies in the fact that the low-level Initiator driver has to provide certain defined functionality. This functionality is defined by the struct `Scsi_Host_Template`. Typical functions that low-level Initiator drivers need to provide are a `queuecommand()` function (which is called when the SIML needs to line up SCSI CDBs for the low-level Initiator driver to transmit to the Target(s) connected to it), an abort function (called when there are errors in SCSI execution), etc. The SIML is completely independent of the low-level Initiator driver. Furthermore, a common SIML enables rationalizing SCSI requests issued to Initiator drivers for different protocols. The SIML serves a two-pronged function. It provides an interface to enable abstraction on the user side, and on the interface to the lower level Initiator driver, it provides generic features that do not take away any necessary SCSI functionality.

A SCSI Initiator can thus be viewed as the functionality provided by the Upper Level, the SIML and the low-level driver. The operation of the SCSI Initiator involves identification of the SCSI resources accessible to the Initiator. This is done as part of the set-up procedure when the low level Initiator driver registers with the SIML. When a `read()` or a `write()` is generated from the user level for an identified SCSI Target, the request is converted into SCSI CDB(s) by the SIML. The SIML then calls the `queuecommand()` for the low-level Initiator driver. The low-level Initiator driver, through its `queuecommand()`, then executes the command, i.e., transmits the SCSI command to the Target. When the response to the SCSI command is received, the low-level Initiator driver informs the SIML about the received response and/or data. The SCSI bottom-half handler then recognizes that a response has been received and if the "good" SCSI status is received, the corresponding command is said to be complete.

A general visualization of the nature of the operation required by a SCSI Target follows from the example of a SCSI Command being executed, as discussed in Chapter 2 and Chapter 4. In the typical operating scenario, a Fibre Channel card such as the QLogic ISP 2200 A, acting as a target, would generate a hardware interrupt for the low-level interconnect front-end Target driver (FETD) upon receiving a frame from an Initiator. The interrupt handler of the FETD would be called upon to take care of the interrupt. The interrupt handler then retrieves the frame. Upon deciphering that the frame is a valid SCSI frame, the interrupt handler would then hand it off to the STML. The driver then goes off to wait for further commands to be received. The STML then checks for any SCSI commands received. It has to decide whether this frame is a part of an entirely new command or a continuation of a command previously handled. Accordingly, the STML will need to assign an ID to the received command. The STML, depending on its mode of operation, will appropriately process the command and return the relevant response to the FETD. (If the command received is a READ, the STML will directly return the relevant data. If, on the other hand, the received command is a WRITE, the STML needs to be able to send XFER_RDY, which allocates the necessary buffer for the Initiator to write the requisite data. It then has to await the reception of the SCSI data – see section 2.3). The STML then has to be able to generate a RESPONSE indicating how the command was executed. If the execution of the SCSI command proceeded without any exceptional conditions, this transmission of a RESPONSE indicates a termination of the command execution sequence. If, on the other hand, exceptional conditions are reported, the STML may receive a few more frames inquiring about the exceptional condition.

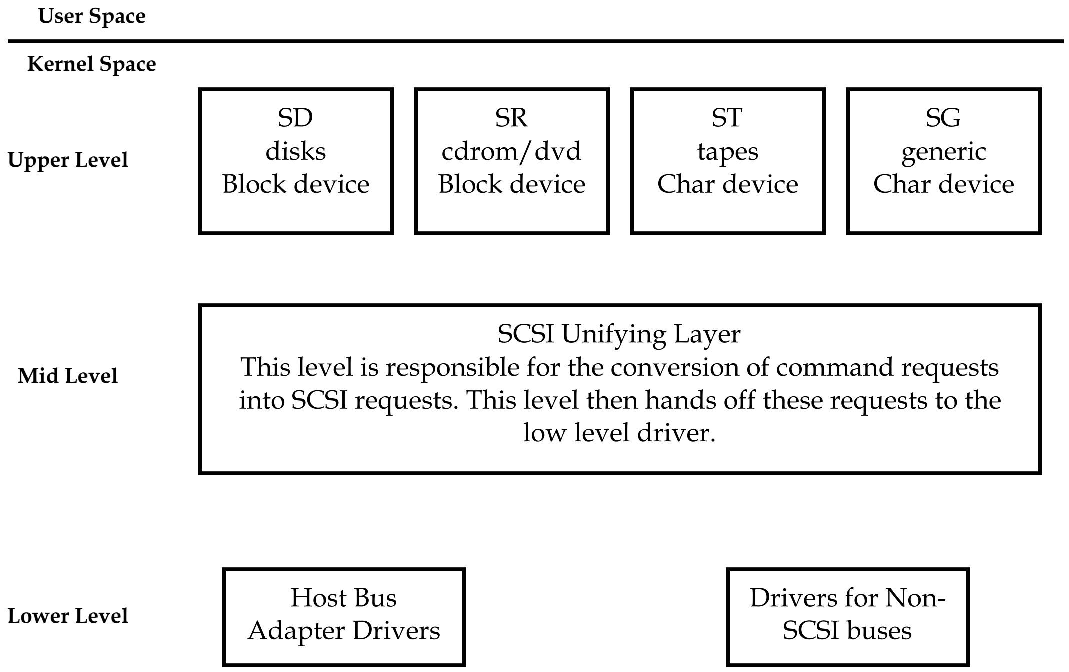
User Space
Kernel Space
Upper Level
SD
disks
Block device
SR
cdrom/dvd
Block device
ST
tapes
Char device
SG
generic
Char device
Mid Level
SCSI Unifying Layer
This level is responsible for the conversion of command requests into SCSI requests. This level then hands off these requests to the low level driver.
Lower Level
Host Bus Adapter Drivers
Drivers for Non-SCSI buses

Figure 5-3: Organization of SCSI code in the Linux kernel

From the discussion of the organization of the SIML and the mode of execution required by the STML, we now discuss the kernel level API that is needed on the Target side. The STML defines the functionality that an FETD has to provide. This is implemented in a manner similar to the way it is currently being done for an HBA driver. A struct `Scsi_Target_Template` (see Figure 7-6) defines the functionality needed. An FETD registers with the STML indicating that it is capable of functioning as a Target. The nature of these functions is defined by the operations defined by the previous paragraph. Essentially, the idea is to be able to separate SCSI functionality from the requirements of the low-level interconnect.

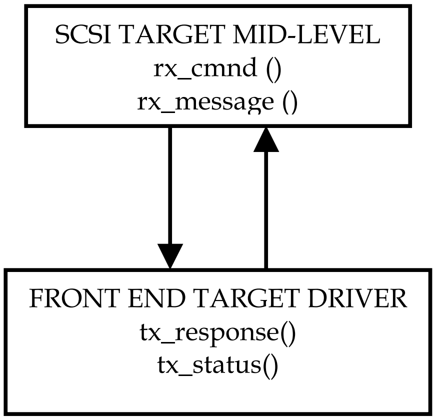

Figure 5-4: API between the generic SCSI Target Mid-Level and the Front End Target Driver

The functions required by the STML (in pseudo-code) would be:

tx_response	(scsi_data, tag)
tx_status	(scsi_cmnd, status_message, tag)

The functions provided by the STML (in pseudo-code) would be:

rx_cmnd	(scsi_cmnd, data, tag)
rx_message	(scsi_message, tag)

The other issue is that of being able to provide some functionality to the user. One of the intended uses of the SCSI Target Mid-Level being for testing, there needs to be some ability to transmit SCSI commands on to the user. This is also important for debugging purposes. The other interface that needs to be available is for configuration. The SCSI Target Mid-Level can also be envisioned as a “gateway” to managing a Storage Area Network (SAN). This gateway needs to be able to define what SAN resources are visible and in what manner. All such issues fall under the gamut of being able to provide a User Level API. While this was a proposed goal of the thesis, this has not been implemented because of a paucity of time.

CHAPTER 6

USER-SPACE TARGET EMULATOR

6.1 Overview

The objective behind the User Space Target Emulator (USTE) is to create an entity that can process SCSI commands on a Target device, and provide functions that are common to all front ends using this USTE. The primary purpose behind the design of the USTE was to gain some experience about what a Kernel Space Target Emulator (KSTE) would need to provide in terms of an interface and also provide some basic understanding about what are good design features. Furthermore, the user space is a better place to debug code as compared to starting work directly in the kernel space, and working in the user space helps USTE development become largely independent of changes in the kernel version. Figure 6-1 shows an abstract overview of what functionality the USTE is expected to provide.

The USTE consists of two logical pieces of code, one of them is responsible for handling the details of the SCSI transport protocol and the other is responsible for the processing of SCSI commands. If designed correctly, the section of code dealing with the processing of SCSI commands should function completely independently of the SCSI transport protocol. In other words, it should extract all the common functionality needed by the section of code dealing with SCSI transport protocol.

6.2 Basic Structure of the User Space Target Emulator

The USTE is a group of functions contained in `scsi_interface.c`. The functions are available to any given front-end for the processing of SCSI commands:

```
int    open_SCSI_device         (__u64 scsi_id, __u64 scsi_lun);
int    close_SCSI_device        (__u64 scsi_id, __u64 scsi_lun);
__u32  get_allocation_length    (unsigned int cmd_len, unsigned char*
                                 scsi_cdb);
int    handle_SCSI_cmd          (unsigned int cmd_len, unsigned int
                                 in_size, unsigned char* i_buff, unsigned
                                 int out_size, unsigned char* o_buff);
```

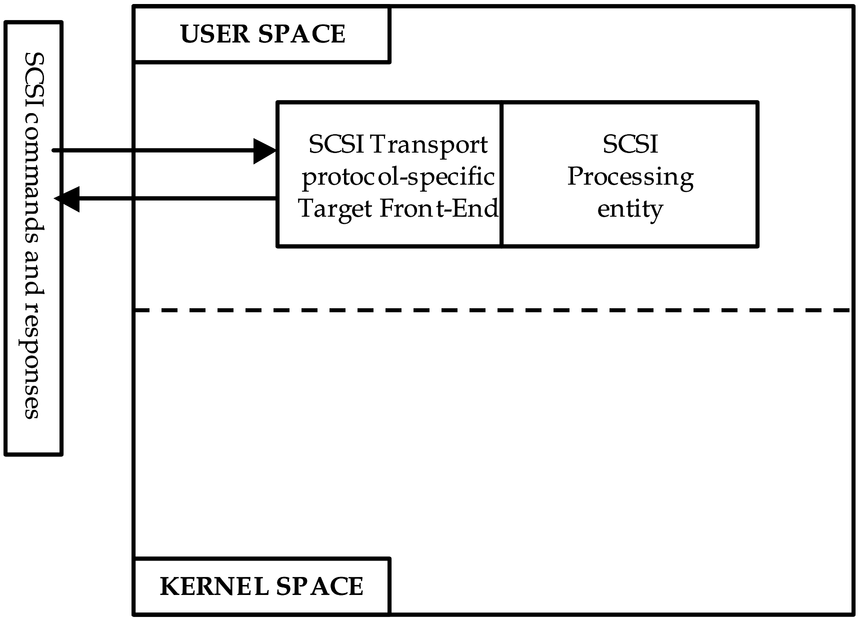
USER SPACE
SCSI commands and responses
SCSI Transport protocol-specific Target Front-End
SCSI Processing entity
KERNEL SPACE

Figure 6-1: Functional Overview of the User Space Target Emulator

In addition to these, a global struct (`struct disk_properties` *`disk_drive`* – Line 17 `scsi_interface.c`) is common between the USTE and the Target front-end.

The use of these functions is discussed below. The USTE can currently work in three modes:

1. I/O to and from a real disk drive
2. I/O to and from a file that contains the logical blocks
3. I/O to and from memory

Only one of these modes can be selected at a given time, since the decision is made when the USTE is compiled. The modes can be selected depending upon hash-defines selected in the `scsi_interface.c` file. These three modes of functioning are shown in Figure 6-2.

6.2.1 I/O to and from a real disk drive

The basic mode of operation (selected when BASIC is uncommented - in `scsi_interface.h`) is to read and write to a real disk drive. For this mode of operation to work, there needs to be some means of transmitting the received SCSI CDBs to a SCSI disk from the user-space as well as some means to recover the response. This function is served by the SCSI generic driver in the Linux kernel. The interface to this driver (discussed in detail later) is by means of traditional `read` and `write` system calls. This enables a user space "driver" to transmit SCSI commands onto a SCSI entity that has been recognized by the Linux kernel. In other words, to use this feature of the USTE, there needs to be on the platform running the USTE, a SCSI-capable host bus adapter (HBA) and a disk drive attached to this HBA. For purposes of testing, a QLogic ISP 2200 card was used as the SCSI HBA. This card accepts SCSI commands and transmits them onto a Fibre Channel interconnect. The driver used for this SCSI Initiator was written by Chris Loveland and is a part of the Linux kernel. Connected to this QLogic card was a 36 GB Seagate Fibre Channel drive (Refer to Figure 6-2).

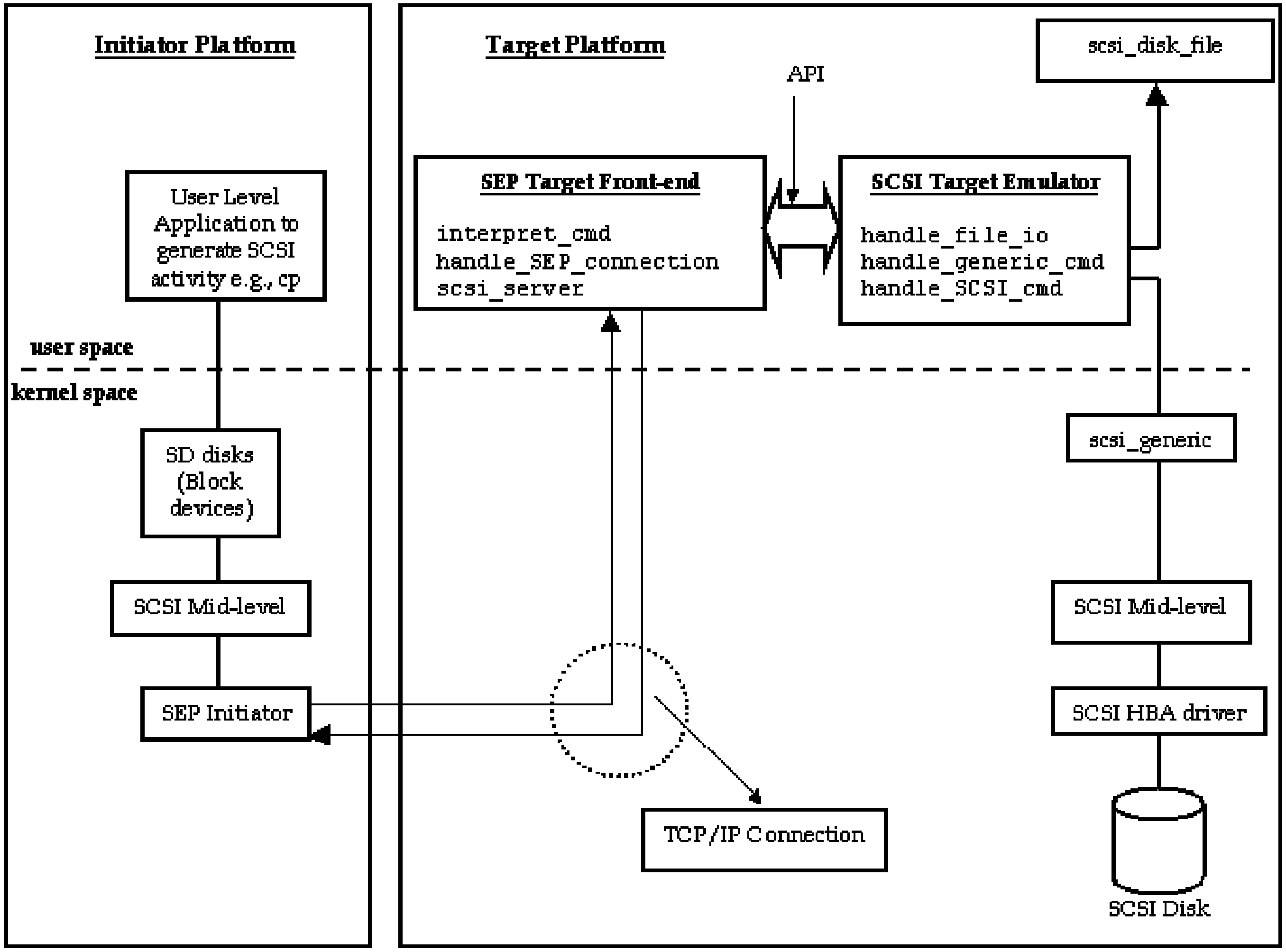

Figure 6-2: Implementation of the User Space Target Emulator

To understand how this feature works, it is important to have an overview of the SCSI generic driver in the Linux kernel. As shown in Chapter 5, the SCSI upper level consists of the SCSI generic functionality as well. This provides an API to the user level to transmit SCSI commands directly into the SCSI mid-level. Whenever a SCSI device is detected by the kernel, the device gets mapped onto a corresponding SCSI generic device. In other words, in our case, when the QLogic driver detected the Seagate disk, it was mapped by the mid-level into /dev/sda (if no other SCSI disk was already present). Furthermore, if the SCSI generic upper-level driver was enabled, the disk drive also gets mapped as /dev/sga. The disk drive is then able to accept commands using the SCSI generic API.

The SCSI generic API has two versions that are currently supported. While this USTE was being developed, the API was changed (changes first noticed in the kernel version 2.4.0-test9). These two versions are henceforth referred to in this document as the SCSI Generic 1 (SG-1) and the SCSI Generic 2 (SG-2) respectively. However, since the original focus of the USTE was to gain an understanding of the issues involved, there was no change made to the code to reflect the changed API. The API discussed below is the SG-1 interface which continues to be supported for reasons of backward compatibility.

The SCSI generic interface, developed by Lawrence Foard (`/usr/src/linux/include/scsi/sg.h`), provides a general way of exchanging data with the generic driver. Before a command can be sent to a SCSI generic device, the device needs to be "`open`"ed. The file descriptor returned by the `open` system call allows the user space application to reference the device in all subsequent system calls. More importantly, it allows the SCSI generic driver to distinguish between different user-space applications trying to send SCSI commands to the same SCSI generic device.

To send a command to an opened SCSI generic device, a data buffer containing:

`struct sg_header`
SCSI command
Space for data to be sent with this command

is sent to the device using the `write`() command.

To obtain the result of a command, a `read`() command is executed with a data buffer containing space allocated for a similar structure:

`struct sg_header`
Space for data to be read with this command

Thus, executing SCSI commands is equivalent to executing a `write`() command in which a command is "written" to a device followed by a `read`() command which gets the required response to the executed commands. The `struct sg_header` presented above is defined as shown in Figure 6-3. The individual terms in the struct are explained in the comments. The `sense_buffer` array gets used only when the command for which the `read`() is being executed has failed.

There are some clear limitations to this mode of functioning. The most obvious is that the data is passed and received as a single logical buffer. The trouble with this occurs when the data reaches the SCSI Generic Upper Level driver. In order make use of the data, the kernel needs either to transfer the data from the user space to the kernel space (in the case of a WRITE-type command) or to copy the data to the user space from the kernel space (in the case of a READ-type command). This forced `memcpy` in the kernel causes some inefficiency. Internally, the kernel expects the data to be in the form of a vector (Figure 6-4). Typically, this is achieved using the `readv` and the `writev` system calls. The `struct iovec` shown in Figure 6-4 is a typical data structure the kernel uses. The data is structured in the form of an array of `struct iovec`, each entry consisting of an address (`iov_base`) and a length (`iov_len`) – defining the number of bytes of data pointed to by `iov_base`.

```
struct sg_header {
int   pack_len; /* length of incoming packet (including header) */
int   reply_len; /* maximum length of expected reply */
int   pack_id; /* id number of packet */
int   result; /* 0==ok, otherwise refer to errno codes */
unsigned int twelve_byte:1; /* Force 12 byte command length for
group 6 & 7 commands */
unsigned int other_flags:31; /* for future use */
unsigned char sense_buffer[16]; /* used only by reads */
/* command follows then data for command */
};
```

Figure 6-3: Definition of `struct sg_header`

```
int readv   (int fd, const struct iovec *vector, int count);
int writev  (int fd, const struct iovec *vector, int count);
struct iovec {
      __ptr_t iov_base; /* Starting address.  */
      size_t iov_len;   /* Length in bytes.  */
};
```

Figure 6-4: I/O using Vectors

There is also an inherent but slightly less obvious limitation to the way SCSI generic works. The maximum size of a buffer that can be passed while executing a `write()` or while receiving from a `read`() is limited by SG_BIG_BUFF which is a compile time constant within the kernel (defined in `include/scsi/sg.h` - Line 238). This constant has been hard coded to be 32 KB since the 2.2 version of the Linux kernel. SCSI commands on a well-written Initiator tend to execute commands with a data buffer of the order of ~128 kB. Using a smaller buffer causes a tremendous amount of inefficiency. For instance, when an Initiator issues a SCSI WRITE(10) command requesting that 128 kB of data be written to the disk, it sends the data to the Target as one contiguous buffer. Upon receiving this 128 kB block of data from the Target front-end, the USTE has to break it up into four corresponding SCSI WRITE(10) commands (accounting for the correct LBAs) and pass the data to SCSI generic appropriately. There are two ramifications because of this. The first is that when a SCSI READ command requesting greater than 32 KB of data is received by the USTE, there is a `memcpy` imposed on the USTE when copying the data from a bunch of smaller buffers to a larger buffer. Secondly, this means that UNIX/POSIX asynchronous I/O cannot be used with this interface, since the smaller READs/WRITEs will have to be completed in sequence.

Furthermore, it is not possible to maintain state information about the SCSI command using the above struct `sg_header`. The absence of this feature also means that UNIX/POSIX asynchronous I/O is not possible with the USTE using the available SCSI generic interface.

The SCSI generic interface has changed considerably in the latest Linux-2.4 kernels. The SG-2 interface [11], discussed in the next chapter, solves most of the problems documented above. The USTE implemented has not been converted to the new interface because of time constraints. However, it should not be a lot of work to convert the USTE to the SG-2 interface to do synchronous I/O. Converting it to UNIX/POSIX asynchronous I/O requires some more work in terms of preparing the USTE to deal with SIGIO.

6.2.2 I/O to and from a file that contains the logical blocks

This is the mode of operation (available when both NON_BASIC and FILE_IO are uncommented - in `scsi_interface.h`) that can be used by systems which do not have a SCSI HBA and/or a SCSI disk attached to the platform that is running the USTE. In this mode of operation, the USTE opens up a file on the local disk drive. The USTE responds to the received SCSI commands as a real SCSI disk drive would. The data from a SCSI READ is obtained by reading the specified number of bytes from the file whereas for a SCSI WRITE, the blocks are written to the file. As far as the remote system is concerned, there is no difference between the file and a real disk. One of the reasons this mode of operation was developed was because it facilitated testing several features of the USTE. This mode of operation does not need the SCSI generic driver in the Linux kernel. The major disadvantage of this mode of operation is that the range of commands that can be used in a SCSI operation is restricted by the number of SCSI commands that are supported by the USTE. In the case where the commands were transmitted to a real disk, the USTE was restricted by the number of commands that are supported by the real disk. The USTE also makes some default assumptions about the block size (set to 512 bytes - defined by BLOCK_SIZE in `scsi_interface.h`) and the size of the disk (set to 600 MB - defined by FILE_SIZE in `scsi_interface.h`) which are relevant when responding to the received SCSI commands and in defining how the received SCSI READs and WRITEs get interpreted.

In this mode of operation, corresponding to each SCSI Identifier (id) and Logical Unit Number (LUN), a file is opened with the name "*`scsi_disk_file_x_y`*", where x = SCSI id and y = SCSI LUN. Thus, all Target front-ends attempting to transmit SCSI commands to a given (id, LUN) pair will end up executing commands to the same file. Consequently, just like on a real shared disk, multiple users attempting to write to the same (id, LUN) will end up corrupting the file system on this simulated disk unless they somehow communicate between themselves to avoid it. The USTE does not attempt to rectify this behavior as it tries to simulate the behavior of a real disk as closely as possible. In block transmission protocols such as SCSI, issues related to maintaining the integrity of a file system are left up to the file system that utilizes SCSI.

6.2.3 I/O to and from memory

This mode of operation (selected when only NON_BASIC is uncommented – in `scsi_interface.h`) is primarily useful for the performance analysis of the front-end protocol implemented for this USTE. In this mode of operation, the USTE, upon receiving a SCSI WRITE command, does not do anything useful with the data and returns a STATUS_GOOD message as required by SCSI. Upon receiving a SCSI READ command, the USTE returns the data buffer "as-is".

6.3 Description of the functions available to a Target front-end

As stated in the previous section, four functions are available to any Target front-end implemented for the USTE. The steps normally involved in executing SCSI commands using the USTE are:

1. Opening the device
2. Get buffer requirements
3. Execute the command
4. Closing the device

Steps 2 and 3 are repeated for the life of the front-end driver. The translation of these steps to the defined USTE is as follows:

6.3.1 Opening the SCSI device using `open_SCSI_device()`

The function expects the SCSI id and the SCSI LUN (both of these are defined by the SCSI Architecture Model for SCSI-3 standard as being 64 bit entities) as inputs. These values are normally a part of the protocol login or the protocol handshake. This must be done when the Target front-end decides that it is ready to receive SCSI commands from an Initiator for that id and LUN. The actual device or file that will be opened will depend upon the mode of operation. For I/O to and from memory, this is a no-op. For I/O to and from a file, the file opened will depend upon the LUN "a" and id "b". The file name will be of the form "*scsi_disk_b_a*". For I/O to and from a real disk, the current design opens SCSI disks on the basis of LUNs. Thus, for LUN 0, `/dev/sga` is opened, for LUN 1, `/dev/sgb` is opened. The id is not used in this mode of operation. It is expected that this mapping of generic devices to IDs and LUNs is the responsibility of an external configuration utility which is a management function not part of the USTE itself.

6.3.2 Finding the buffer requirements using `get_allocation_length()`

Upon receiving a SCSI command, the front-end driver should pass the SCSI Command Descriptor Block (CDB) along with the length of the command to the function `get_allocation_length`(). The function extracts the buffer length in bytes from the CDB and then returns it. This is the length of the buffer that the front-end driver needs to allocate for the data alone. Whether the front-end driver receives the actual data to fill into the buffer from the Initiator or from the USTE depends on the command. For a SCSI READ, the data would be received from the USTE. For a SCSI WRITE, the data would be received over the network from the Initiator that issued the WRITE command. Normally, the data transfer direction is included as part of the header information in the underlying protocol implementing SCSI. While this is the external behavior of the function, internally, it prepares the USTE to receive the command. It makes error checks on the received CDB and copies the CDB over to *`disk_drive`* (of type struct `disk_properties` – see Figure 6-5).

```
struct disk_properties {
        int     fd;             /* file descriptor */
        int     max_buff;     /* how much can I read/write once */

        /*  to help write greater than the amount allowed by
            scsi generic */
        int     write_flag;     /* write received = 1 */
        __u32   to_write;       /* how many bytes to write */
        __u32   written;        /* Bytes already written */
        __u32   lba_to_write;   /* which logical block to write */
        __u32   blocksize;      /* Size of Blocks */
        __u16   max_write;      /* How many to write */
        __u16   max_read;       /* How many to read */

        /*  to help read greater than the amount allowed by
            scsi generic */
        __u32   to_read;        /* how many bytes to read */
        __u32   read;           /* bytes already read */
        __u32   lba_to_read;    /* which logical block to read */

        unsigned char write10_cdb[BYTE+2]; /* 10 Byte write cmd */
        unsigned char read10_cdb[BYTE+2];  /* 10 Byte read cdb */
};
```

Figure 6-5: Definition of `struct disk_properties`

6.3.3 Calling `handle_SCSI_cmd()` to execute the command

After receiving the necessary buffer size from `get_allocation_length()`, the Target front-end needs to create the buffers necessary to deal with the data. The size of struct `sg_header` is defined by the constant SCSI_OFF (Line 58 in `scsi_interface.h`). Thus, for a SCSI WRITE(10) command, the buffer that would have to be allocated would be SCSI_OFF + 10 (size of the WRITE CDB) + size of the buffer obtained from `get_allocation_length`(). For a SCSI READ(10), on the other hand, the buffer that would have to be allocated would be SCSI_OFF + size of the buffer obtained from `get_allocation_length`().

The `handle_SCSI_cmd` function requires the following parameters as inputs when a READ command is received:

a. `cmd_len`: Length of the command (in bytes) being transmitted - For a SCSI READ(10), this is 10.

b. in_size: Length of the input data size (in bytes) - For a SCSI READ(10), this is 0 since the direction of the data transfer is from the USTE to the Front End Target driver.
c. i_buff: A pointer to the CDB in the input buffer allocated as described above.
d. out_size: Length of the output data size (in bytes) - For a SCSI READ(10), this is the value returned by get_allocation_length() + SCSI_OFF.
e. o_buff: A pointer to the output buffer allocated as described above.

The handle_SCSI_cmd function requires the following parameters as inputs when a WRITE command is received:

a. cmd_len: Length of the command (in bytes) being transmitted - For a SCSI WRITE(10), this is 10.
b. in_size: Length of the input data size (in bytes) - For a SCSI WRITE(10), this is the value returned by get_allocation_length(), since the direction of the data transfer is from the Initiator to the Front End Target driver.
c. i_buff: A pointer to the CDB in the input buffer allocated as described above.
d. out_size: Length of the output data size (in bytes) – For a SCSI WRITE(10), this is 0.
e. o_buff: A pointer to the output buffer allocated as described above.

Internally, the handle_SCSI_cmd() is responsible for handing off the command to specific functions depending upon the mode of operation selected by the user. If a NON_BASIC mode of operation (I/O to and from a file, I/O to and from memory) is selected, then the command gets handed off to handle_file_io(). This function executes the command and returns the status along with the data (if required). If the basic mode of operation is selected, then handle_SCSI_cmd() determines if the data buffer associated with the command is greater than SG_BIG_BUFF (defined to be 32 kB by the Linux kernel). If so, executing the command requires executing multiple SCSI commands so that the total transfer size spread across these multiple SCSI commands is equivalent to the transfer size requested by the received SCSI CDB. Thus, a SCSI WRITE(10) requesting that 64 kB be written to a disk drive with a block size of 512 bytes starting at Logical Block Address (LBA) 0 is mapped into two SCSI WRITE(10) commands, each one requesting that 32 kB be written to the disk, the first one starting with LBA 0, and the second one starting with LBA 64 (= 32 x 1024 / 512 + 0). After deciding if multiple commands are needed to execute the received CDB, the function then hands off the command(s) to handle_generic_io which executes the commands and returns the status along with data depending upon the command.

6.3.4 Closing the SCSI device using close_SCSI_cmd()

When the front-end is ready to close a given id and LUN, the front-end should call the function close_SCSI_cmd(). This will prevent this front-end from receiving any more SCSI commands at the given id and LUN unless reopened again. Note that other front-ends may continue to be able to issue SCSI commands to the same id and LUN.

6.4 Design of the SEP User-Space Front-End

For the purposes of testing, the SEP protocol, described in Chapter 4, was implemented as a front-end to the USTE described above. The SEP Initiator was implemented by Anshul Chadda, so that it interfaced with the SCSI Initiator mid-level of the Linux kernel. In addition to the SEP Initiator itself, a configuration tool ("sep_config") was also implemented.

The SEP Initiator interfaces with the SCSI Initiator mid-level using the struct `Scsi_Host_Template`. The `Scsi_Host_Template` used by SEP is shown in Figure 6-6. The SEP Initiator, upon start-up, registers itself with the SCSI Initiator mid-level. The SEP Initiator then spawns a thread which is the main processing thread. This thread is responsible for the transmission of SCSI commands using the SEP protocol and receiving the corresponding responses and handing them back to the SCSI Initiator mid-level. The `sep_config` utility is used to tell the SEP Initiator about what SCSI Target and LUN it must connect to. This is done as follows:

```
sep_config up ip=x lun=y
```

where x = IP address of the Target (either dotted decimal notation or the DNS registered name) and y = SCSI LUN on Target x to which the connection should be established. All communication to LUN y happens on this established connection. When the connection needs to be removed, the `sep_config` utility is used again as follows:

```
sep_config down lun=y
```

The `sep_config` utility has been designed to mirror the `ifconfig` utility for IP interfaces. Linux maps SCSI Targets to device files of the type /dev/sda, /dev/sdb, etc. This mapping happens automatically for every SCSI Target discovered when the SEP Initiator is “`detect`”ed. However, after an Initiator is detected and operational within the Linux kernel, if new Targets are discovered, there isn’t a mechanism from within the SCSI mid-level to map the SCSI Targets to the device files. The `sep_config` utility handles this problem by using the `proc` file interface of the Linux kernel. The `sep_config` utility forces a mapping of the device by “writing” to the proc file-system. This activates the SCSI mid-level to search for new devices corresponding to those that the Initiator had written to the `proc` file system.

```
#     define      SCSI_SEP {                        \
proc_info:             scsi_sep_proc_info,      \
detect:                scsi_sep_detect,         \
release:               scsi_sep_release,        \
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:          SG_LIST,                 \
cmd_per_lun:           1,                       \
unchecked_isa_dma:     0,                       \
use_clustering:        ENABLE_CLUSTERING,       \
use_new_eh_code:       1                        \
}
```

Figure 6-6: `Scsi_Host_Template` used by the SEP Initiator

On the Target side, the SEP front-end (Figure 6-2) was designed to be able to use the USTE functions as described in Section 6.3. The SEP front-end is a simple server application (in `scsi_server.c`) that creates a TCP socket, waits on a well-known port (chosen to be 4000) and spawns off a child process for each accepted connection. The child (in `sep_connection.c`) is then responsible for handling the new connection. The child begins execution at the `handle_SEP_connection`() function (Line 275 in `sep_connection.c`). The basic functionality provided is an infinite loop that receives commands from the SEP Initiator using a TCP connection, determines if the received SEP frame is SEP-specific or something that the USTE needs to handle. If the received frame is SEP-specific, the SEP front-end generates a response to the received SEP packet. In case the received frame is a SCSI command or SCSI data, the SEP front-end strips off the SEP header and hands it off to the USTE. The mode of functioning is in the manner required by the USTE interface and described in the preceding section. Upon receiving a response from the USTE, the SEP front-end adds a SEP header and then transmits the response to the corresponding SEP Initiator via the same TCP connection. The SEP front-end is a very limited implementation of the SEP protocol. It is primarily designed to work synchronously (i.e., at most one command is outstanding at any given moment). Certain command types allowed by the SEP protocol are not implemented (e.g., third-party commands). Flow control between the Initiator and the Target is not used. Also, commands executed to different LUNs are mapped to different SCSI disk drives attached to the system instead of going to multiple LUNs on the same drive. This modification was made because the test environment lacked disk drives with multiple LUNs.

The SEP Target responds to a received CONNECT-NEGOTIATE message from the SEP Initiator with a valid NEGOTIATION response. The corresponding SCSI device is then opened. The SEP Initiator can then transmit SCSI commands to the SEP Target. The mode of operation can be selected as described in the preceding sections.

6.5 Lessons learned from the User Space Target Emulator

The general principle learned was to have the front-end do the bare minimum possible. While this may seem obvious, it implies that the USTE has to do things in a roundabout manner for this principle to be maintained. The following were the design lessons that were learned by implementing the USTE:

1. The allocation of buffers needs to be done by the USTE, not the front-end. This helps clean up code on the front-end considerably.
2. The USTE needs to be a separate logical piece of code. The USTE consists of functions provided to execute SCSI commands. This is compiled with the front-end code which implies that Target stubs for different front-end protocols run as completely separate processes. This implies that sharing of resources between different front-end protocols is not possible. Also, it is desirable to provide a generic entity responsible for handling all SCSI commands received from any number of front-ends.
3. There were some unavoidable copies of SCSI data necessitated by the SCSI generic requirements. These can be avoided in the kernel-space implementation of the Target Emulator. This is explained in Chapter 7 dealing with the kernel space emulator.

Furthermore, many of the design choices made in the user-level implementation of the USTE were influenced by the fact that the SEP protocol was to be implemented synchronously. This allowed sharing of data structures between the USTE and the Target front end. This needs to be avoided in the kernel space Target front end.

CHAPTER 7

DESIGN AND IMPLEMENTATION OF THE KERNEL SPACE TARGET EMULATOR

7.1 Introduction

The Kernel Space Target Emulator (KSTE) refers to an entity that presents itself as a SCSI Direct Access disk while running within the Linux Kernel space. Actually, the term "emulator" when used for the kernel space entity is a misnomer. In the user space implementation, the entity providing the SCSI functionality and the entity responsible for transmitting SCSI over a given protocol form one logical piece of code. However, in the kernel space implementation, the entity responsible for handling SCSI commands has an existence independent of the low-level front-end Target driver (FETD) that is responsible for the transmission of SCSI in a device-specific manner. Thus, in terms of visualization, it may be better to think of the kernel space emulator as consisting of two distinct entities - the generic SCSI Target mid-level (STML) and the low-level front-end protocol-specific Target driver (FETD). This is shown in Figure 7-1.

The SCSI Target mid-level runs parallel to and sometimes, in conjunction with, the SCSI Initiator mid-level and is concerned only with the processing of SCSI commands. This chapter presents the issues involved in the design of the SCSI Target mid-level, the eventual design in terms of the interface it provides to and requests from any FETD that needs to use it, and the actual implementation of the SEP front-end Target driver (FETD-SEP) and the Fibre Channel front-end Target driver (FETD-FC) for this mid-level. It also discusses code organization and the specifics of the design.

7.2 Issues involved in the Design of the SCSI Target Mid-Level

The USTE implemented with a SEP front-end was very valuable in designing the STML. The choices made in the design of the STML have evolved from the design of the USTE.

One of the main lessons learned was that the STML needed to have an existence separate from that of the FETD instead of being compiled with the FETD itself. This is important to make the same STML accessible to multiple FETDs implementing different protocols. The way in which the USTE was designed allowed each protocol to have its own Target Emulator. Thus, it would not have been possible to present the SCSI resources available to the STML with any cohesion were different FETDs to run simultaneously.

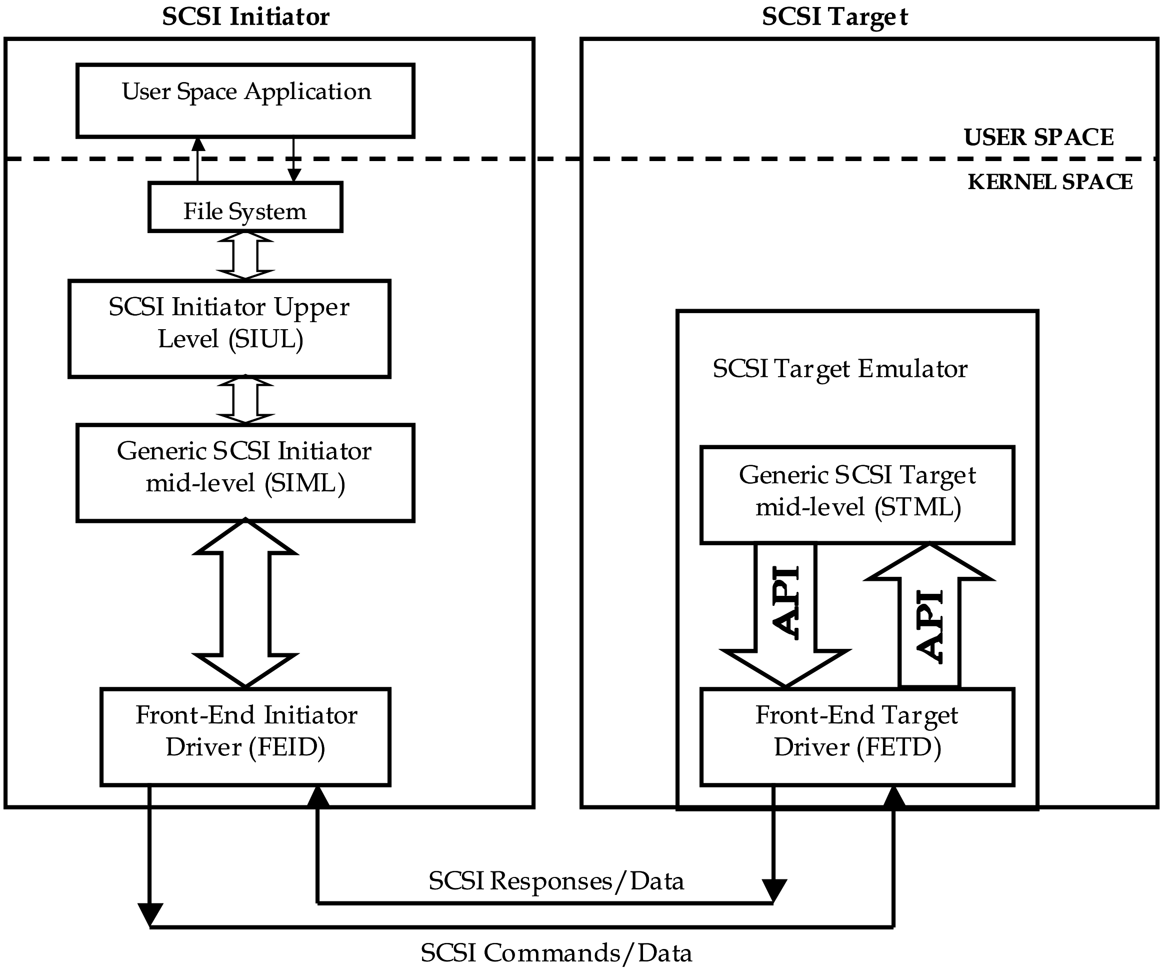
SCSI Initiator
SCSI Target
User Space Application
USER SPACE
KERNEL SPACE
File System
SCSI Initiator Upper Level (SIUL)
Generic SCSI Initiator mid-level (SIML)
Front-End Initiator Driver (FEID)
SCSI Target Emulator
Generic SCSI Target mid-level (STML)
API
API
Front-End Target Driver (FETD)
SCSI Responses/Data
SCSI Commands/Data

Figure 7-1: A Block View of SCSI Initiator and Target Sub-Systems

Unlike the SCSI Initiator mid-level (SIML), during normal operation, the STML is called by the FETD from the context of an interrupt-handler. Running within the interrupt-context severely limits the type of operations that are allowed (the prime restriction being the requirement of not going to sleep while in interrupt context). In addition, the SIML is always called from within the context of some user-level application (except in the case of SCSI error handling). This process context is not available to the STML. In the normal mode of operation, an FETD will receive a SCSI command (in an interrupt handler) and will pass it onto the STML. Thus, in order to process this received command, the STML will need some execution context. This context can be provided by an STML kernel thread. Thus, the interface to be provided to the FETD will have to be two-fold. The STML will have to provide a set of entry points that an FETD can call from within the context of an interrupt handler. The major function of these entry points into the STML is to queue the received information for the STML thread so that the information can be processed, and to awaken the STML thread. The second half of the interface is the set of entry points that the FETD will need to provide so that the STML thread can communicate with the Initiator. The major function of these entry points is to receive the processed SCSI commands and then transmit the appropriate response over the specific protocol for which the FETD was written. This model is clearly extensible to multiple protocols and enables the STML to be generic. The major difference between the SCSI Target stack and the SCSI Initiator stack is the absence of an upper level on the SCSI Target side. The specifics of the interface are discussed in a subsequent section.

In the case of the USTE, the requirements of the older SCSI Generic interface had imposed a memory copy (described in Chapter 6) when using the Target Emulator for I/O to and from a real disk. This interface was changed in the Linux development kernels as of linux-2.4.0-test9. This newer interface is referred to henceforth as the SCSI Generic 2 (SG-2) interface. The SG-2 interface (discussed later) is used in the kernel space implementation. It rationalized the manner in which data could be handed off to the SIML. This helped avoid the unnecessary memory copy and led to some performance gains.

7.3 Overview of the Operation of the SCSI Target Sub-System

Any discussion of the details of the block view presented in Figure 7-1 would be difficult to understand in the absence of an operational overview. This section tries to present precisely this. The details of the entire process are presented in subsequent sections.

Upon start-up, the STML spawns a kernel thread (SCSI Target Thread). The SCSI Target Thread (STT) represents the major processing entity within the STML. The STML has three modes of operation, each independent of each other, which decide how SCSI commands are processed. These modes parallel those available in the USTE namely, (a) I/O to and from memory, (b) I/O to and from a file that contains the logical blocks and (c) I/O to and from a real disk. These modes of operation can be selected at compile time and thus, exist independently of each other. If the selected mode of operation is (c), then the STML spawns a second kernel thread (SCSI Signal Processing Thread). This thread is required to allow asynchronous processing of SCSI commands (explained later).

When a new FETD is introduced to the kernel, it registers itself with the STML. It provides to the STML by means of a jump table, a list of functions that enable the STML to transmit responses to the received SCSI commands. In turn, the STML provides the FETD with a set of functions that the FETD can use to inform the STML about received commands and data, and awaken the STT. These two interfaces represent the front-end to mid-level and mid-level to front-end APIs respectively. The FETD, after it has registered with the STML, is ready to receive SCSI commands. Upon receiving a SCSI command, the FETD hands it over to the STML and in the process awakens the STT. The STT then allocates the space needed to process the command. If the command needs data from the Initiator in order to process it further, the command is then handed back to the FETD in order to get the required data from the Initiator in question and the STT goes back to sleep. The FETD, upon receiving the required data, informs the STML that the command is ready to be processed and awakens the STT.

Depending upon the mode of operation selected for the STML, the command is processed. When processing is completed, the STT informs the FETD that the command is processed. The FETD is then responsible for transmitting the appropriate response and then informing the STML when it is done transmitting the response and awakens the STT. The STT finally frees up the resources allocated to process the command which completes the processing of a command.

7.4 SCSI Target Mid-Level to Front-End Target Driver API

The FETD needs to have some way of notifying the STML about a received command or data. The functions in the STML that provide these operations provide entry-points into the mid-level code and are also responsible for initiating the processing of the received commands. This API consists of the following functions:

- *register_target_template()*
- *deregister_target_template()*
- *register_target_front_end()*
- *deregister_target_front_end()*
- *rx_cmnd()*
- *scsi_rx_data()*
- *scsi_target_done()*
- *scsi_release()*
- *rx_task_mgmt_fn()*

These functions are described below.

`int` **`register_target_template`** **`(`**`Scsi_Target_Template *`*`the_template`***`);`**
The FETD needs to register with the STML before it can request the processing of SCSI commands. This registration and deregistration process has been intentionally kept similar to that required by the SIML. Since this process will be common to all the front-end drivers, the functionality required to do this has been provided in a single file (`scsi_target_module.c`). All front-ends will have to include this file and provide a template (of type `Scsi_Target_Template` – see Figure 7-6) assigned to the local variable *`my_template`*. This template (discussed in the next section) is the interface that must be provided by the FETD. The STML creates a linked list of templates available globally. When an FETD starts up, the function `scsi_target_module_init` (Line 17 - `scsi_target_module.c`) calls the STML function `register_target_template` with *`my_template`* being passed as the parameter.

`int` **`deregister_target_template`** **`(`**`Scsi_Target_Template *`*`the_template`***`);`**
The FETD calls this function with the variable *`my_template`* (used in the registration process) when the module defining the FETD is removed from the kernel. This function is called from the `scsi_target_module_cleanup` function (`scsi_target_module.c` - Line 27) in the FETD. This function is responsible for cleaning up all resources associated with the corresponding module (including the template, any unexecuted commands, etc). Successful execution of this function allows the removal of the module containing the FETD associated with the template.

`Scsi_Target_Device*` **`register_target_front_end`** **`(`**`Scsi_Target_Template *`*`tmpt`***`);`**
During the processing of a call to the `register_target_template` function, the STML makes a call back to the `detect` function associated with that template. This `detect` function is provided by the FETD and is called by the STML to find out the number of devices that will receive SCSI commands using the protocol for which the FETD was written (The registration process is summarized in Figure 7-9). If there are any such devices, then the FETD will need to register these devices with the STML in order to make the STML aware of the existence of these devices. The `register_target_front_end` function needs to be called by the `detect` function for each new device detected. For each new device, the STML creates an entry of the type `Scsi_Target_Device`, shown in Figure 7-2.

The STML assigns a device id for each call to the `register_target_front_end` function. The device id is stored in the *`id`* field and uniquely identifies the corresponding device. The STML stores a list of all devices as a linked list available globally within the STML (discussed in Figure 7-8). The FETD needs to use the assigned device id when indicating to the STML that it has received a SCSI command.

```
typedef struct STD {
/* device id */
__u64                   id;
/* next one in the list */
struct STD              *next;
/* ptr to available functions */
Scsi_Target_Template    *template;
} Scsi_Target_Device;
```

Figure 7-2: Definition of `Scsi_Target_Device`

`int` **`deregister_target_front_end`** **`(`**`Scsi_Target_Device` *`*the_device`***`);`**

When an FETD is removed from the kernel (by calling `rmmod`), the FETD executes the `scsi_target_module_cleanup` function. Similar to the `scsi_target_module_init` function, this function is common to all FETDs and is therefore provided as part of the design (`scsi_target_module.c` - Line 17). The `scsi_target_module_cleanup` function makes a call to the `deregister_target_template` function (The deregistration process is summarized in Figure 7-10). This function cycles through the global list of devices (i.e., FETDs that have previously registered with the STML) looking for those that use *`the_device->template`*. As the STML finds each such device, it calls `deregister_target_front_end`. The `deregister_target_front_end` function then calls the `release` function in the FETD which allows the FETD to free up the resources associated with that device. The STML then dequeues any unprocessed commands associated with that device and deregisters the device by removing it from the global device queue (see Figure 7-8). The FETD can explicitly call the `deregister_target_front_end` function when it has determined that the entity associated with *`the_device`* is no longer functional. Typically, this happens when an exceptional condition has been detected and *`the_device`* is left in a non-functional state. Calling this function will stop the execution of any commands that may currently be pending with the STML. This function provides a safe way for the continued execution of the STML while allowing the removal of a device associated with a FETD.

`Target_Scsi_Cmnd *`**`rx_cmnd`** **`(`**`SCSI_Target_Device` *`*device`*`,` `__u64` *`target_id`*`,` `__u64` *`lun`*`, unsigned char` *`*scsi_cdb`*`, int` *`len`***`);`**

This is the basic function that is called by the FETD when it receives a SCSI command. The command is expected to be stored in a buffer (*`scsi_cdb`*) of length *`len`*. The FETD informs the STML what *`device`* received the command. Since, the SCSI CDB does not indicate what SCSI id (*`target_id`*) and SCSI LUN (*`lun`*) was supposed to receive this command, the SCSI protocol allows this information to be transmitted in an interconnect-specific manner. Hence, this information needs to be explicitly conveyed to the STML as parameters to the `rx_cmnd` function. The `rx_cmnd` function allocates a `Target_Scsi_Cmnd` struct (See Figure 7-3). `rx_cmnd` then fills up the corresponding entries in the allocated struct and returns a pointer to it to the FETD. The FETD should not have to manipulate any of the fields in the returned `Target_Scsi_Command` except to transmit or receive data which is done through *`req`* (of type `Scsi_Request` struct - defined in `/usr/src/linux/drivers/scsi/scsi.h` – shown in Figure 7-4). The fields that are relevant to the FETD are *`sr_data_direction`* (used to indicate where the FETD should expect data from), *`sr_buffer`* (which is a scatter-gather list of length *`sr_sglist_len`*) containing data of length *`sr_bufflen`*. The status of the command can be checked from *`sr_result`*. The conventions followed are those used by the SIML (all of which are noted in the file /usr/src/linux/drivers/scsi/scsi.h). The only fields that should be changed by any FETD should be the *sr_buffer* field while filling up the data to be received by a WRITE command for example. The rx_cmnd function can be called from within an interrupt context.

int **scsi_rx_data (**Target_Scsi_Cmnd **the_command*);
The FETD uses this function to provide notification to the STML that data has been received for *the_command*. This function allows WRITE-type commands to begin processing (Figure 7-12).

```
typedef struct SC {
/*
 *  state: This will take different values depending on what the
 *  present condition of the command is
 */
int                 state;
/* id: id used to refer to the command */
int                 id;
/* dev_id: device id - front end id that received the command */
__u64               dev_id;
/* dev_template: device template to be used for this command */
struct STT          *dev_template;
/* target_id: scsi id that received this command */
__u64               target_id;
/* lun: which LUN was supposed to get this command */
__u64               lun;
/* cmd: array for command until req is allocated */
unsigned char       cmd[MAX_COMMAND_SIZE];
/* len: length of the command received */
int                 len;
/* queue of Scsi commands */
/* next: pointer to the next command in the queue */
struct SC           *next;
/* req: this is the SCSI request for the Scsi Command */
Scsi_Request        *req;
} Target_Scsi_Cmnd;
```

Figure 7-3: Definition of Target_Scsi_Cmnd
(Note: Certain fields in the struct have been excluded for the purposes of brevity)

```
struct scsi_request {
int                 sr_magic;
int                 sr_result;
/*
 * Status code from lower level driver
 * Obtained by REQUEST SENSE when CHECK CONDITION is received on
 * original command (auto-sense)
 */
unsigned char       sr_sense_buffer[SCSI_SENSE_BUFFERSIZE];
struct Scsi_Host    *sr_host;
Scsi_Device         *sr_device;
Scsi_Cmnd           *sr_command;
/* A copy of the command we are working on */
struct request      sr_request;
unsigned            sr_bufflen; /* Size of data buffer */
void                *sr_buffer; /* Data buffer */
int                 sr_allowed;
unsigned char       sr_data_direction;
unsigned char       sr_cmd_len;
unsigned char       sr_cmnd[MAX_COMMAND_SIZE];
/* Mid-level done function */
void                (*sr_done)(struct scsi_cmnd *);
int                 sr_timeout_per_command;
unsigned short      sr_use_sg; /* scatter-gather */
/* size of malloc'd scatter-gather list */
unsigned short      sr_sglist_len;
/* Return error if less than this amount is transferred */
unsigned            sr_underflow;
};
```

Figure 7-4: Definition of `struct scsi_request`

Thus, in the normal mode of operation, when a WRITE(10) command is received, the FETD calls `rx_cmnd`. After the looking at the command, the STML decides that it needs data from the Initiator in order to execute the command. The STML then allocates the buffer space needed (*`sr_buffer`*) and notifies the FETD that it can receive the necessary data by using the `rdy_to_xfer` function in the `Scsi_Target_Template` corresponding to the FETD. The FETD can then receive the required data in an interconnect-specific manner. Upon receiving the data, the FETD then calls `scsi_rx_data` to notify the STML that the expected data has been received. The `scsi_rx_data` can be called from within the context of an interrupt handler. It changes the state of *`the_command`* and wakes up the mid-level thread so that the STT can then process *`the_command`*.

`int` **`scsi_target_done`** **(**`Target_Scsi_Cmnd *`*`the_command`***);**

This function is called by the FETD to signify that it has completed the execution of *`the_command`*. This function changes the state of *`the_command`* and awakens the STT so that *`the_command`* can be dequeued. Upon return from this function, all information about *`the_command`* has been deleted from the STML. The semantics of when this function is called depends upon the protocol used by the low-level interconnect. The normal rule for calling this function is to call it when the FETD has done everything required by the low-level protocol to allow the Initiator to consider the execution of *`the_command`* as being completed. This function can also be called from an interrupt context.

`int` **`scsi_release`** **(**`Target_Scsi_Cmnd *`*`the_command`***);**

This function can be called by the FETD when it wants to abort a command without having to send a response to the command that is being aborted. This function is needed to account for implicit aborts to commands defined in SCSI. The reasons as to why these implicit logouts happen depend on the semantics of the low-level interconnect used. An example of this scenario is when the Loop Address of the Target changes in Fibre Channel.

`Target_Scsi_Message*` **`rx_task_mgmt_fn`** **(**`SCSI_Target_Device *`*`device`*`, int` *`fn`*`, void*` *`value`***);**

This function is called by the FETD to indicate to the STML that it has received a SCSI Task Management function corresponding to a command or a group of commands depending upon the nature of the Task Management function (indicated by *`fn`*). This is most commonly used when the FETD has received an ABORT for a given command. In that case, *`value`* points to a `Target_Scsi_Cmnd`. *`value`* changes meaning depending upon the Task Management function that is received. The present implementation of the STML does not implement the ABORT TASK SET, CLEAR ACA and the CLEAR TASK SET Task Management functions. This is partly because of the needs of this design implementation and partly because of the lack of a unique identifier by which a SCSI Target can reference a SCSI Initiator. All SCSI Transport protocols have the concept of a Login whereby they allow an Initiator to transmit SCSI Commands to the Target. The nature of this Login is specific to the low-level interconnect. As a result, each SCSI Transport Protocol defines an identifier construct by which a Target can keep track of the Initiators that are currently logged in to it. An example of this would be the Fibre Channel Port Name / Node Name construct. However, the STML cannot treat this identifier as an opaque object by means of which it can reference the Initiator, because the semantics of how a Login is invalidated change from interconnect-to-interconnect. As a result, when an ABORT TASK SET is received, the STML has no means by which it can identify which commands belong to which Initiator. Thus, these Task Management functions cannot be implemented using this design model at the STML. The way around this problem is by having the FETDs do the work in determining which commands are affected by the received Task Management function. The FETD can then call the `rx_task_mgmt_fn` in the STML to ABORT those commands. The `rx_task_mgmt_fn` then allocates a `Target_Scsi_Message` struct (Refer to Figure 7-5) and returns it to the FETD. The STML then executes the received Task Management function. The FETD gets informed about the execution of the Task Management function when the `task_mgmt_fn_done` function is called. This process is shown in Figure 7-13.

```
typedef struct SM {
/* next: pointer to the next message */
struct SM         *next;
/* prev: pointer to the previous message */
struct SM         *prev;
/* message: Task Management function received */
int               message;
/* device: device that received the Task Management function */
struct STD        device;
/* value: if relevant to the function */
void              *value;
} Target_Scsi_Message;
```

Figure 7-5: Definition of `Target_Scsi_Message`

7.5 Front-End Target Driver to SCSI Target Mid-Level API

The STML needs to have some way of being able to transmit responses to the FETD in a consistent manner. In the Linux Operating System, this is achieved by using jump tables which define the functionality required. This jump table is the Target template to be assigned to the FETD variable *`my_template`* (the use of this variable is discussed in the previous section). The FETD variable *`my_template`* is of type `Scsi_Target_Template`. This struct is defined in Figure 7-6. A description of the use of the fields in the `Scsi_Target_Template` follows.

```
typedef struct STT {

const char        name[BYTE];

int   (*detect)(struct STT*);

int   (*release)(struct STD*);

int   (*xmit_response)(struct SC*);

int   (*rdy_to_xfer)(struct SC*);

int   (*task_mgmt_fn_done)(struct SM*);

void  (*report_aen) (int, __u64);

} Scsi_Target_Template;
```

Figure 7-6: Definition of `Scsi_Target_Template`

`const char *name*[BYTE];`
This field represents the name of the template. It must be unique (at least in the name space for the templates that have been registered with the STML) so as to help identify the template. The length of `name` has been restricted to eight characters which should be sufficient for most purposes.

`int` **`detect`** `(struct STT *the_template);`
This function is used by the STML to detect the number of devices that use the given front-end protocol (implemented by the functions of *`the_template`*) to receive and process SCSI commands and data. The FETD `detect` function (functionality shown in Figure 7-9) is required to call the `register_front_end` function provided by the STML to make the STML aware of the front-end Target device. The FETD `detect` function should allocate the resources required by the FETD for each device detected to ready the device to receive SCSI commands and data.

`int` **`release`** `(struct STD *the_device);`
This function should free up the resources allocated to *`the_device`* (struct STD defined as shown in Figure 7-2). The `release` function (functionality shown in Figure 7-10) is called by the STML when the module corresponding to the given device is removed from the kernel. Normally, all front-end devices corresponding to a FETD are all removed from the global device list by repeated calls to the `release` function (made by `deregister_target_front_end`). After the function is called and executed, the STML is not available to process any SCSI commands for the given front-end device. Thus, a device is valid between calls to the `detect` and the `release` function.

`int` **`xmit_response`** `(struct SC *the_command);`
This function is the Target equivalent of the SIML `queuecommand` function. The FETD should transmit the response buffer (*`the_command->req->sr_buffer`*) and the status (*`the_command->req->sr_status`*). The expectation is that executing this function is non-blocking. In other words, the `xmit_response` function should serve the function of allowing the STML to inform the FETD that the response for *`the_command`* is ready. The actual transmission of the appropriate response to *`the_command`* needs to happen outside of the context of the `xmit_response` function. The reason for this is that this function is called by the STT which is responsible for processing SCSI commands. There may be multiple commands queued up with the STML (either by the same FETD or from different FETDs). If the `xmit_response` function were written so that the transmission of a response happens within the context of this function, this would imply that the STT would block while the response to *`the_command`* is being transmitted. This, in effect, would slow down the processing of other commands in the STML. After the response is actually transmitted (or more precisely, the FETD has met all the requirements of the interconnect to allow the Initiator in question to consider the execution of *`the_command`* complete), the FETD should call the `scsi_target_done` function provided by the STML which will allow the STML to free up the resources associated with *`the_command`*. The functionality of the `xmit_response` function is detailed in Figure 7-11 and Figure 7-12.

`int` **`rdy_to_xfer`** `(struct SC *the_command);`
The STML uses this function to inform the FETD that data buffers corresponding to *`the_command`* have now been allocated and it is okay to receive data for *`the_command`*. This function is necessary because a SCSI Target does not have any control over the commands it receives. Thus, in the case of a WRITE operation, a Target can receive data for a command only after the STML has called the `rdy_to_xfer` function (functionality summarized in Figure 7-12) corresponding to *`the_command`*. In addition, most SCSI transport protocols have a

corresponding Information Unit / Protocol Data Unit which informs the SCSI Initiator that buffers have been allocated e.g., XFER_RDY in Fibre Channel. As with the `xmit_response` function, it is expected that such packet transmission, if required, is not done within the context of the `rdy_to_xfer` function (i.e., `rdy_to_xfer` is non-blocking). Likewise, data reception should be done outside the scope of the `rdy_to_xfer` function. After the data is actually received, the FETD needs to call the `scsi_rx_data` function provided by the STML in order to continue processing *`the_command`*.

`int` **`task_mgmt_fn_done (`**`struct SM *`*`the_message`*`);`
The STML uses the `task_mgmt_fn_done` function to inform the FETD that a received Task Management function has been executed. This is typically used to complete the processing of ABORTs. *`the_message`* defines the message (Refer to Figure 7-5) which has been executed by the STML. The function `task_mgmt_fn_done`, like the `xmit_response` and the `rdy_to_xfer` functions, is expected to be non-blocking. Furthermore, the STML frees up the resources associated with the commands affected by the Task Management function once the `task_mgmt_fn_done` function returns. The details of the SCSI Task Management functions (issued normally to correct error conditions) are extremely complex. Although the SCSI semantics of how the Task Management functions need to be executed are defined, how they translate to different low-level protocols is highly protocol-specific. The present structure is designed to be simple and flexible.

`int` **`report_aen (`**`int *`*`fn`*`, __u64` *`lun`*`);`
When a Task Management function is received, depending on the nature of the function, the execution may affect the commands queued up by other Initiators. SCSI has thus devised a mechanism whereby Initiators can be notified of such events (referred to as Asynchronous Event Notification – Refer to Section 2.5.2). The `report_aen` function is called to allow the FETD to notify Initiators that have logged in with the Target about the Task Management function *`fn`* which may have affected any commands they may have queued up for LUN *`lun`* (The value of LUN may or may not be relevant depending on the *`fn`*).

7.6 SCSI Target Mid-Level Design

This section presents a more detailed view of the functioning of the STML using the functions that have been defined above. A detailed view of the FETD and STML is presented in Figure 7-7. The interaction between various code pieces and how they interact with each other is dealt with in subsequently.

On start-up, the STML creates a global list of devices (*`st_device_list`* – See Figure 7-8) that have registered with it. The STML then spawns a thread which is responsible for most of the processing done by the STML (`scsi_target_process_thread` – SCSI Target Thread - STT). This is, in real terms, the SCSI mid-level. The STT sets up so that it is ready to receive and process SCSI commands. If the mode of operation used is I/O to and from a real disk, the STML spawns a second thread (`signal_process_thread` – SCSI Signal Processing Thread - SSPT). The sole function of this thread is to detect and process the SIGIO signal received by the thread if the execution of a SCSI CDB transmitted to a disk has been completed.

The STML then sets up a global struct (of type `Target_Emulator` shown in Figure 7-8). This global struct contains fields for the list of devices (*`st_device_list`*) and for the list of commands (*`cmd_queue_start`* and *`cmd_queue_end`*). The remaining fields have been discussed in the comments.

One of the primary considerations involved in this design was whether to create a thread to process every registered device or to let a single thread deal with all the devices. Designing the STML so that each registered device has a thread for itself implies that for a large number of FETDs operating simultaneously with multiple front-end devices, a large amount of system resources will get tied up in maintaining the threads and their states. Whereas in the general case, threads have the effect of simplifying matters, in this particular case, there is a lot of interaction necessary between the threads to gain access to common data structures (using semaphores or spin-locks). This makes it very difficult to track errors and also makes deadlocks extremely likely. On the other hand, this method does have potential benefits in speeding up processing of commands and parallelizing the flow in case of a multi-processor system. Furthermore, having a single processing thread for all registered devices represents a single point of failure, while using one thread per device could help recovery in case a front-end device fails/encounters an exceptional condition. In the interests of simplicity, the single thread design was implemented. However, it is possible to convert the implementation so that there is a processing thread per registered device with some manipulation of the data structures.

7.6.1 Registration and Deregistration with the mid-level

A new FETD will have to be registered with the STML before it can begin to hand over SCSI commands to the STML for processing. An FETD is introduced into the operating system either dynamically by inserting the module corresponding to it (using `insmod`), or at system startup, when the driver is compiled directly into the kernel. The 2.4 series of kernel has rationalized the way in which modules and compiled drivers begin operation. In either case, the `scsi_target_module_init` function in the FETD gets called. This function calls the `register_target_template` function in the STML. The `register_target_template` function adds the template to a global linked list (*`st_target_template`*) maintained by the STML. The STML then proceeds to call the `detect` function in the FETD via the template. The `detect` function is expected to probe the hardware to see if there are devices that use the low-level protocol implemented by the FETD to receive SCSI commands and to transmit SCSI responses and data. If the `detect` function finds any such device(s), then the STML calls the `register_target_front_end` function for each such device. The STML then allocates a struct of type `Scsi_Target_Device` (See Figure 7-2), fills it up, adds it to a global linked list of devices (*`st_device_list`*) in the STML, and returns a pointer to the struct to the FETD. This completes the registration of that device with the STML. This is shown in Figure 7-9.

The deregistration process is initiated when a module corresponding to a FETD is removed from the Operating System (using `rmmod`) or during system shutdown. In either case, the `scsi_target_module_cleanup` function in the FETD module is called. This function calls the STML function `deregister_target_template`. The `deregister_target_template` goes through the global list of devices (*`st_device_list`*) to see if there are any that correspond to the given template. If there are devices present, then the `deregister_target_front_end` function in the STML is called, once for each device implementing the template. This function calls the `release` function in the FETD via the device template. The `release` function frees up any resources allocated by the FETD. The `deregister_target_front_end` then removes the device from the global device list (*`st_device_list`*), any commands that may be pending are discarded and any resources that may be tied up for this device are freed up for the kernel to reuse. The FETD needs some mechanism whereby it can selectively deregister a front-end device. This typically happens when the corresponding front-end device encounters an exceptional circumstance which prevents the device from working normally and can, therefore, not receive SCSI commands or transmit appropriate responses.

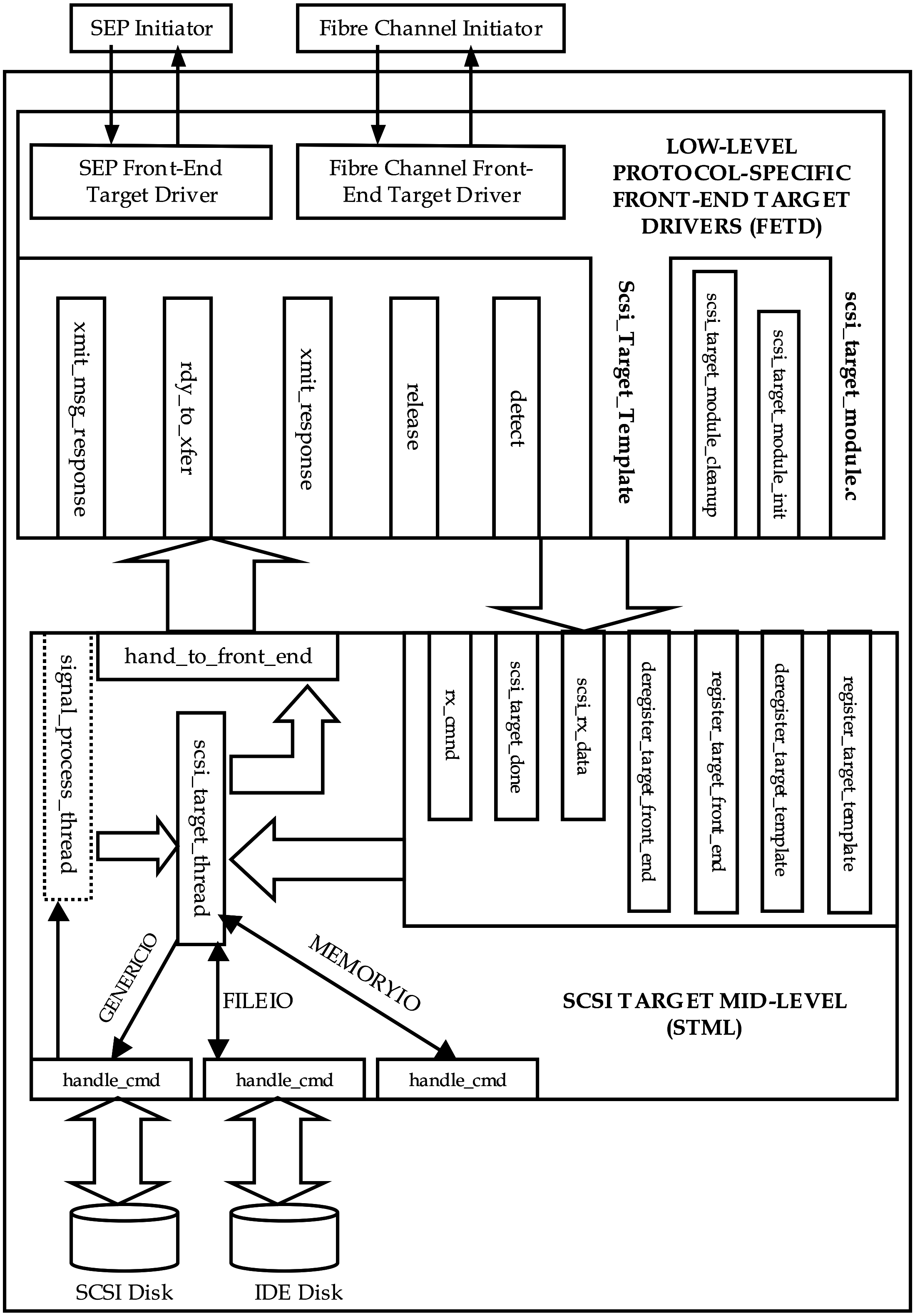

Figure 7-7: A detailed view of the SCSI Target Emulator Implementation

```
typedef struct GTE {
/* command_id: id assigned to a command that gets queued */
int                     command_id;
/* thread_sem: semaphore to control the killing of the STT */
struct semaphore        thread_sem;
# ifdef GENERICIO
/* signal_sem: semaphore to control the killing of the SSPT */
struct semaphore        signal_sem;
/* sig_thr_sem: the SSPT blocks on this semaphore */
struct semaphore        sig_thr_sem;
/* signal_id: pointer to the task struct corresponding to the SSPT */
struct task_struct      *signal_id;
# endif
/* target_sem: the STT blocks on this semaphore */
struct semaphore        target_sem;
/* thread_id: pointer to the task struct corresponding to the STT */
struct task_struct      *thread_id;
/* st_device_list: list of devices registered with the STML */
Scsi_Target_Device      *st_device_list;
/* st_target_template: templates registered with the STML */
Scsi_Target_Template    *st_target_template;
/* add_delete: spinlock to be acquired when a command is added or
 * removed from the STML queue. Necessary because the command is
 * added from the interrupt context.
 */
spinlock_t              add_delete;
/* queue_sem: semaphore to acquire when elements are added or removed
 * from the queue.
 */
struct semaphore        queue_sem;
/* cmd_queue_start: pointer to the start of the command queue */
Target_Scsi_Cmnd        *cmd_queue_start;
/* cmd_queue_end: pointer to the end of the command queue */
Target_Scsi_Cmnd        *cmd_queue_end;
} Target_Emulator;
```

Figure 7-8: Global Data Structure used by the STML

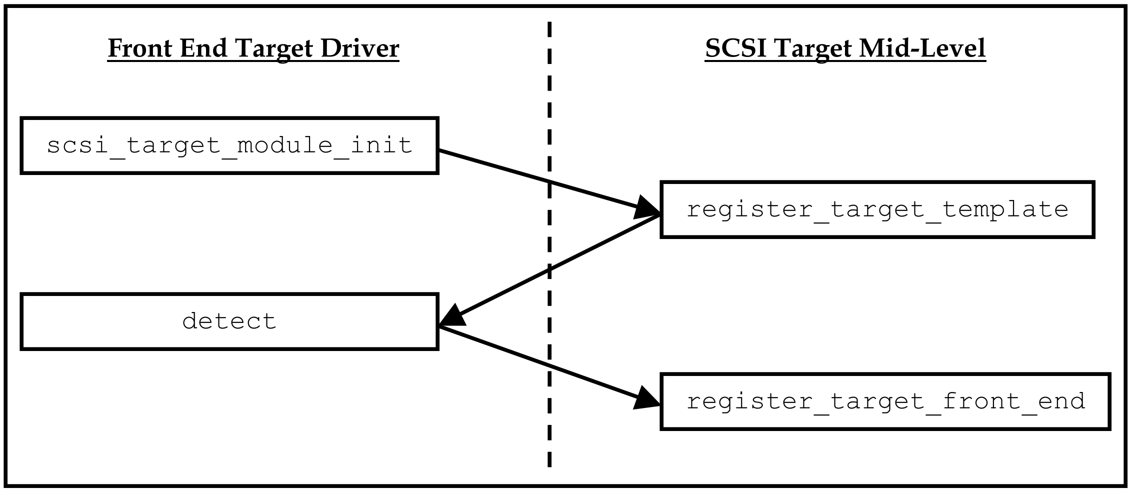

Figure 7-9: Registration of a Device with the SCSI Target Mid-Level

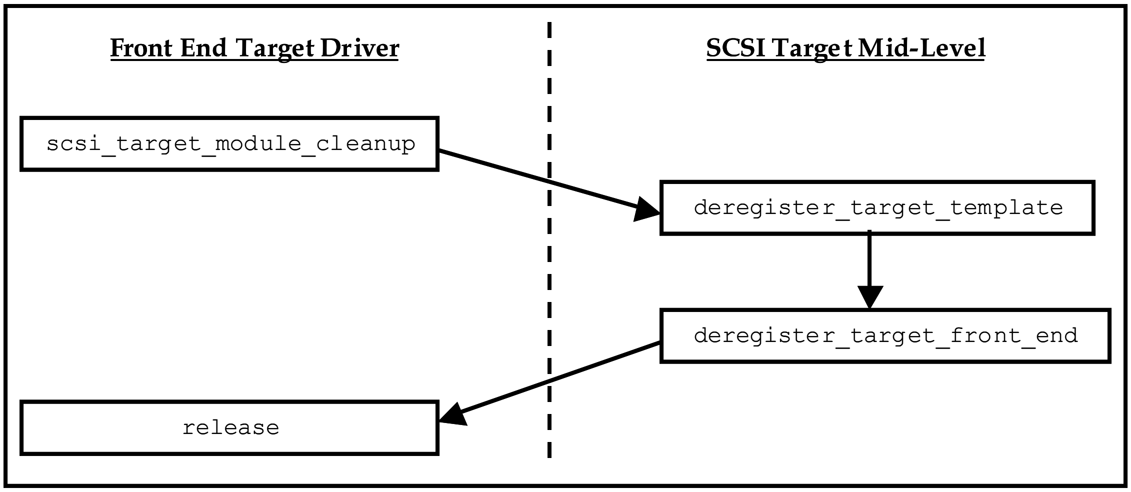
Front End Target Driver
SCSI Target Mid-Level
scsi_target_module_cleanup
deregister_target_template
deregister_target_front_end
release

Figure 7-10: Deregistration of Device(s) from the SCSI Target Mid-Level

As the result, the deregistration process has been designed to be asymmetric. The `deregister_target_front_end` function serves precisely this function and allows the FETD to selectively deregister a front-end device while allowing the mid-level and other front-end devices to continue normal functioning.

7.6.2 Processing of a READ-type command

Once an FETD has registered with the STML and it has detected and registered device(s) with the STML, each front-end device is ready to receive SCSI commands. Upon receiving a SCSI command from an Initiator, the FETD calls the `rx_cmnd` function in the STML. This function creates a `Target_Scsi_Cmnd` struct corresponding to the received SCSI CDB, fills up the struct, adds the command to a global queue of commands (*`cmd_queue_start`* and *`cmd_queue_end`*) and then wakes up the STT. The FETD has access to the allocated struct by means of the return value of the `rx_cmnd` function. The STT processes the command by handing it off to the internal `handle_cmd` function. If the command received is a READ-type command (i.e., where data direction is from a Target to an Initiator or where only a status message needs to be sent), the `handle_cmd` function allocates the necessary buffer space.

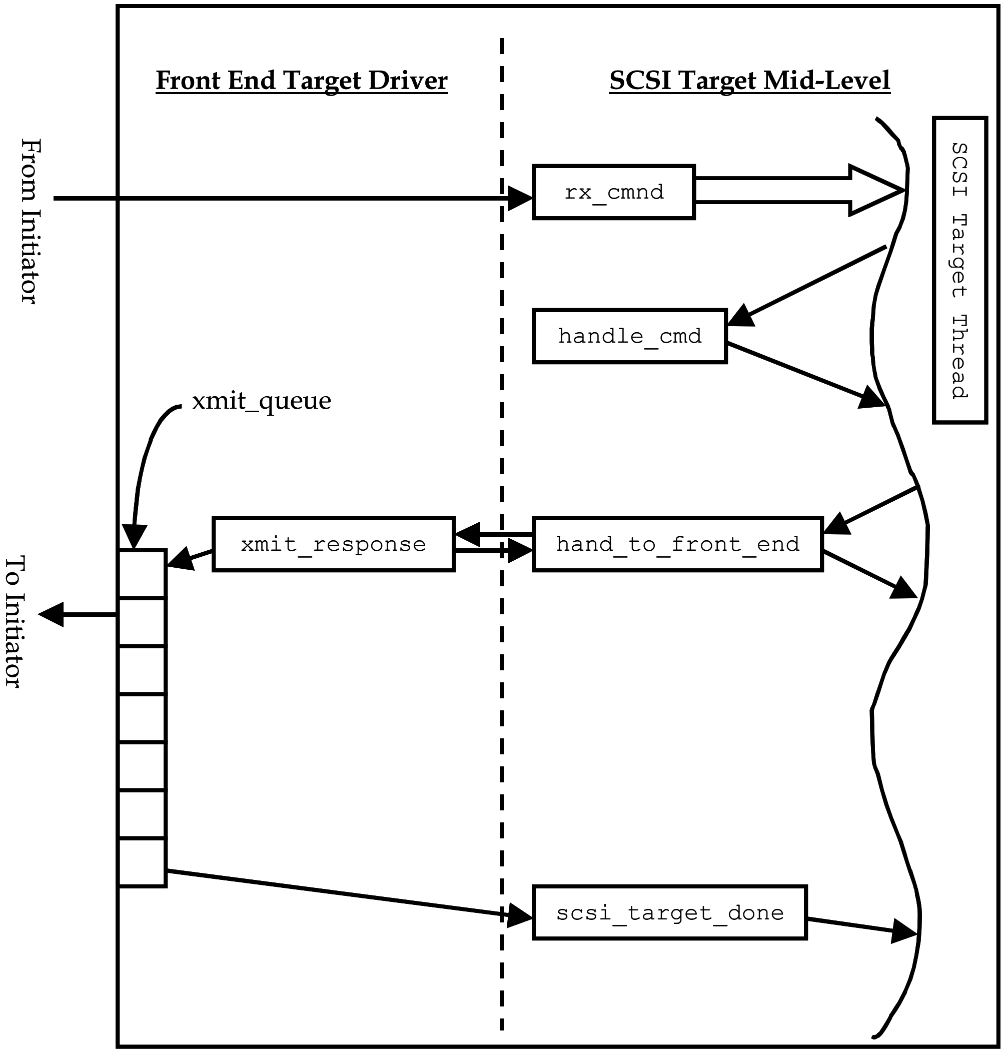

Figure 7-11: Processing of a READ-type command

This buffer space is contiguous so that devices having DMA capabilities can access the memory. Depending on the mode of operation selected, the command is handed off for processing appropriately. When processing is completed, the STT calls the internal `hand_to_front_end` function. This function makes appropriate error checks such as if the command was ABORTED in the intermediate time, or if the FETD was removed etc., and then calls the `xmit_response` function in the template corresponding to the FETD that received the command. The `xmit_response` function should queue up the response to be transmitted to the Initiator and then return. When the response is actually transmitted and the low-level protocol declares the execution of a SCSI command to be complete, the FETD can call the `scsi_target_done` function in the STML. This function allows the STML to free up resources that may be allocated for the command. These steps are shown in Figure 7-11.

7.6.3 Processing of a WRITE-type command

The basic steps when a WRITE-type command (i.e., a command where the direction of data transfer is from the Initiator to the Target) is received are the same as those required by a READ-type command until the STML hands the command off to its internal handle_cmd function. handle_cmd, upon deciding that it is a WRITE-type command, allocates the necessary buffers and changes the state of the command which causes the STT to call the internal hand_to_front_end function. The hand_to_front_end function interprets this command as waiting for data. The hand_to_front_end function then calls the rdy_to_xfer function in the FETD that received the command. A call to the rdy_to_xfer function means that the buffers required for the execution of the command have been allocated. The FETD then transmits any Information Units corresponding to the flow control mechanism relevant to the protocol used by the low-level interconnect (see Chapter 2 and Chapter 4). Since, the rdy_to_xfer function is expected to be non-blocking, the FETD needs to receive the data required for the command outside the context of the rdy_to_xfer function. Once data has been received, the FETD then calls the scsi_rx_data function in the STML. The scsi_rx_data function changes the state of the command and awakens the STT. The STT then processes the received command by handing it off to handle_cmd once more. This time handle_cmd processes the command according to the requirements of the mode of operation relevant to the STML (see Section 7.7). Once the command is processed, the STT calls the hand_to_front_end function again. The hand_to_front_end function this time calls the xmit_response function in the FETD. As in the case of the READ-type commands, once the FETD is done transmitting the actual response, it calls the scsi_target_done function provided by the STML. The STML then frees up the resources allocated for the execution of the command. These steps are shown in Figure 7-12.

7.6.4 Processing a Task Management function

When an FETD receives a Task Management function, it needs to inform the STML about the received Task Management function. As described in Section 7.4, the STML has not implemented the CLEAR ACA Task Management function. Furthermore, the execution of the ABORT TASK SET and the CLEAR TASK SET functions requires the FETD to determine what commands need to be aborted. When the FETD has performed this function, it informs the STML of the received Task Management function using the rx_task_mgmt_fn in the STML. rx_task_mgmt_fn creates a Target_Scsi_Message struct (Refer to Figure 7-5) and queues this entry up in the message queue of the STML. It then awakens the STT. The STT, upon detecting that it has received a Task Management function, executes the Task Management function. It then informs the FETD about the successful execution of the Task Management function by calling the task_mgmt_fn_done. Various SCSI Transport protocols have different interconnect-specific Information Units to inform the Initiator about the successful execution of the Task Management function. The FETD can transmit the required Information Units after the task_mgmt_fn_done function is called. If the Task Management function affects commands queued up by other FETDs than the one that received the Task Management function (i.e., LUN RESET and TARGET RESET), then the report_aen function is called for all the FETDs. This allows the FETDs to inform Initiators that have logged into them about the Unit Attention condition that has been created at the Target. These steps are shown in Figure 7-13.

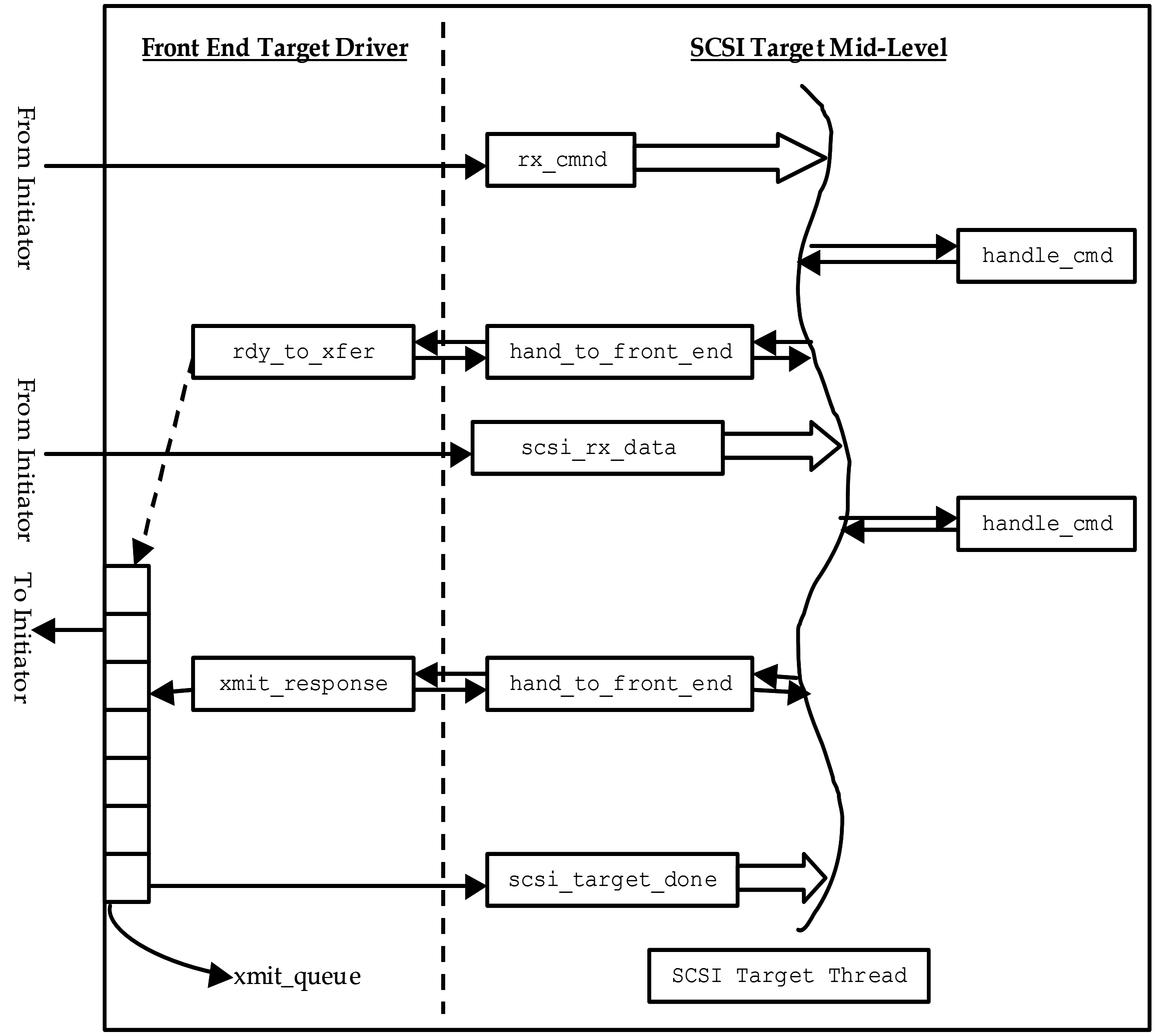

Figure 7-12: Processing of a WRITE-type command

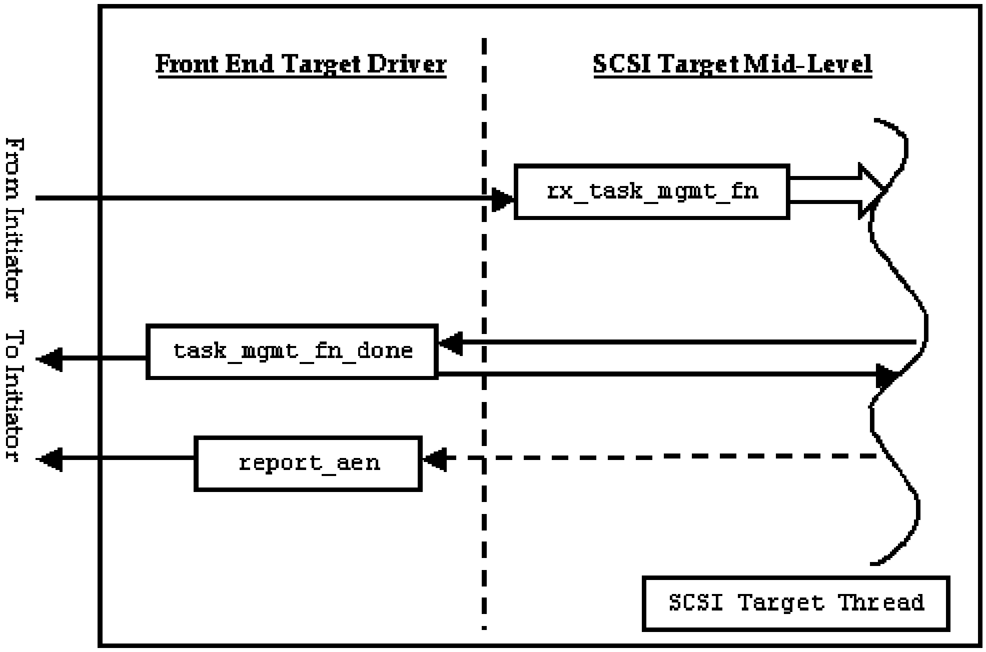

Figure 7-13: Processing of a Task Management function

7.6.5 Other design considerations

Another issue that had to be decided while designing the STML was whether to have a single queue of commands each in a different stage of processing or to have different queues depending upon the stage of processing. The gain by selecting the latter option is primarily to reduce the amount of searching that may have to be done for command processing. The former option on the other hand, requires just one semaphore/spin-lock in order to resolve issues involving multiple accesses to the command queue. In addition, the advantage with the second option is that all commands on the command queue can be processed in one pass of the command queue search. For the option involving multiple queues for commands in different stages of processing, there is a lot of overhead in dequeuing commands from one queue and adding them on another and maintaining the associated state information. Thus, the option of using multiple queues against one queue provided little benefit for added complexity. As a result, the single queue option was implemented. The execution state of the command on the command queue is available to the STML through the `state` variable in the `Target_Scsi_Cmnd` struct (this was not shown in Figure 7-3 as the FETD does not need access to this variable).

7.7 SCSI Target Mid-Level: Modes of Operation

The SCSI Target mid-level, like the USTE, is designed to work in three different modes of operation. The three modes of operation are independent of each other. Only one of these can be selected at compile time. These three modes of operations are:

1. I/O to and from memory
2. I/O to and from a file that contains the logical blocks
3. I/O to and from a real disk

Selecting a mode of operation is primarily responsible for deciding the functionality that will be provided by the `handle_cmd` function in the STML.

7.7.1 I/O to and from memory

This mode of operation (selected by uncommenting MEMORYIO in `scsi_target.h` - Line 47) is primarily designed to be used for performance analysis of a SCSI transport protocol. This mode of operation does minimal processing on the received commands so that the time spent in the STML is the bare minimum. In this manner, the SCSI front-end protocol is the limiting entity and therefore, forms a good comparative base for the SCSI protocol in question. Although the FETD can queue up multiple commands for the STML, the processing of these commands is synchronous - that is, these commands are processed within one cycle of the main loop in the STT. When a READ(10) command is received, the data buffers are allocated and returned to the Initiator. For a WRITE(10) command, the data is discarded.

7.7.2 I/O to and from a file that contains the logical blocks

This mode of operation (selected by uncommenting FILEIO in `scsi_target.h` - Line 49) enables the system acting as a SCSI Target to create and maintain logical data blocks in a regular file that is stored on any type of random access media. When the Target Emulator starts up, it opens up, from within the kernel space, file(s) with the name "*scsi_disk_file_x_y*", where x = SCSI id and y = SCSI LUN. The code is set up to respond to id = 0 and the number of LUNs can be altered through the variable MAX_LUNS (`scsi_target.h` - Line 69). Presently, the code responds to two LUNs (0 and 1). The size of the disk represented by each file can be adjusted by changing the value of FILESIZE (`scsi_target.h` - Line 57) and is presently set to 1.7 GB. The STML responds to all commands except SCSI READs and WRITEs. For the READs and WRITEs, the STML is responsible for converting the Logical Block Addresses (LBAs) in the received Command Descriptor Blocks (CDBs) into block requests that can be executed by the file system. The processing of these commands, like I/O to and from memory, is synchronous. The disadvantages of this approach have been documented in the corresponding section in the Chapter 6, dealing with the USTE.

7.7.3 I/O to and from a real SCSI disk

In moving from the user space to the kernel space, there is an increase in the number of options that are available to transmit SCSI commands onto a SCSI disk connected to the system. These options are a direct result of the way in which the SCSI Initiator mid-level is organized. These three levels differ in their closeness to the lower level interconnect. Corresponding to these three levels, there are three options, each option having its pros and cons. These options are depicted in Figure 7-14.

7.7.3.1 I/O to and from a SCSI disk using the `queuecommand` interface

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

The first option is where the STML uses the `queuecommand` function of the SCSI HBA in the system. For the platform running the SCSI Target Emulator, the list of SCSI HBA drivers in the SIML is available by means of a global queue provided by the SIML and each host registers its host template with the SIML. Each host is required to have a `queuecommand` function which allows the SIML to transmit SCSI commands and data. Using this `queuecommand` interface, the STML can directly transmit SCSI commands and data onto any SCSI disk connected to the corresponding HBA without interfering with the SIML queueing code. The functional overhead is also the lowest of the three options available.

The disadvantages of doing this are that the STML has to deal will any errors that may occur in the execution of the SCSI command. In other words, there is a lot of potential duplication of the SIML code in the STML if this option is implemented. Also, the code is not generic enough in that the STML needs to be aware of all the SCSI hosts on the system and needs to be able to separate SCSI direct access disks from other SCSI Targets that may be connected to the system. In addition, the SIML provides for command retries and timeouts. By interfacing directly with the SCSI HBA driver, the STML does not have access to these features of the SIML. This option has not been implemented in the STML due to these disadvantages.

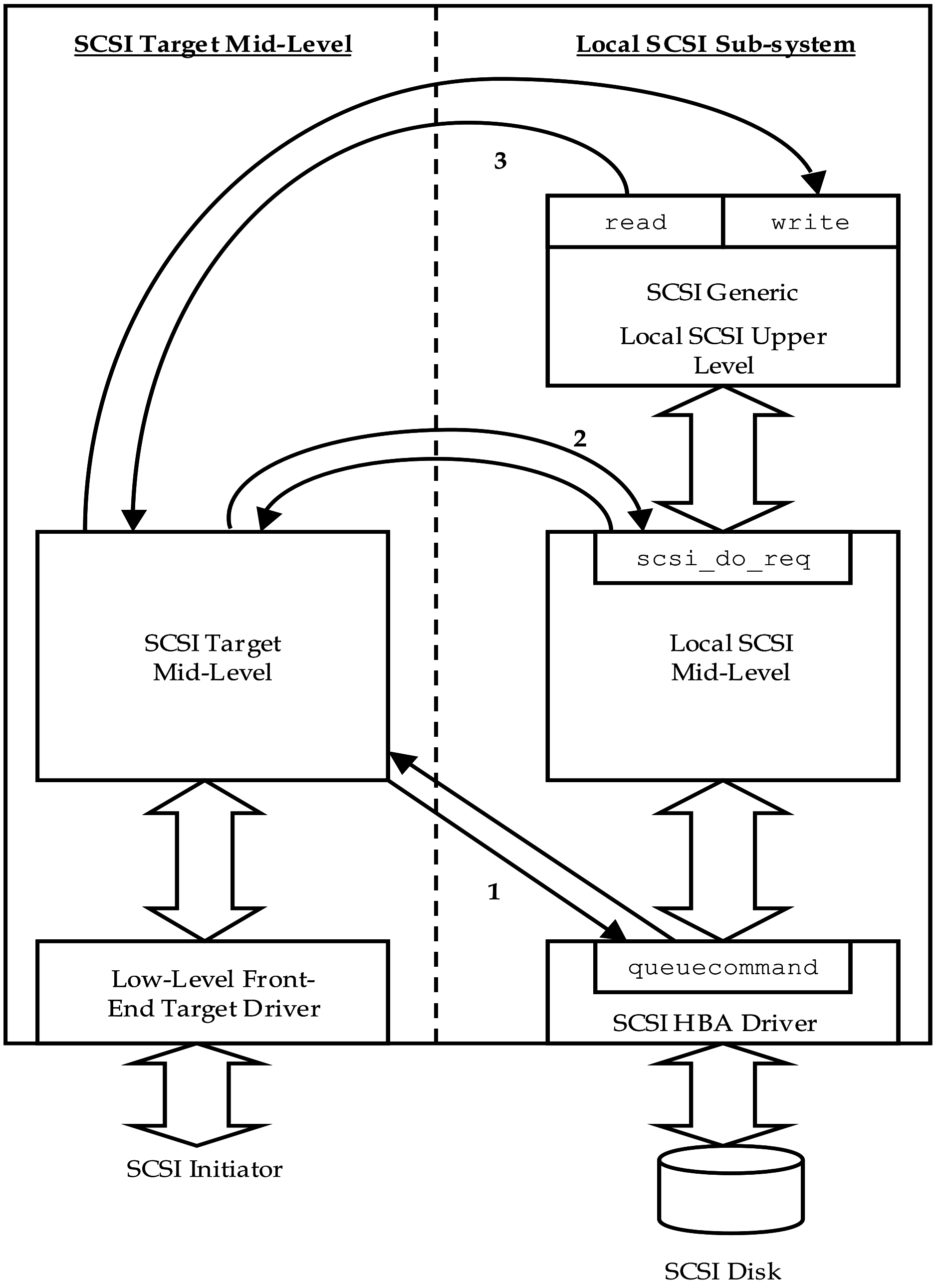

Figure 7-14: Options for I/O to and from a SCSI Disk

7.7.3.2 I/O to and from a SCSI disk using the `scsi_do_req` interface

```
void scsi_do_req  (Scsi_Request *req, const void *cmnd, void *buffer,
                   unsigned int bufflen, void (*done) (Scsi_Cmnd*), int
                   timeout, int retries);
```

The second option consists of the STML cutting in just above the SIML. This is the interface that all the upper-level drivers (`sd`, `st`, `sr` and `sg`) use to transmit SCSI commands to the SIML. The interface was previously provided by means of the `scsi_do_cmnd` function in the SIML. Recent versions of the kernel have changed this interface over to the `scsi_do_req` function. This change has been made to try to differentiate between a request and a command, a request being a SCSI CDB that is put into a queue and a command being a request that is at the head of the same queue. Although not completely implemented, the `scsi_do_req` is the preferred interface. The advantage of this interface is that the SIML handles some of the basic error handling details. Also, the interface is a little more generic, in that, the SIML transmits the SCSI commands received using the `scsi_do_req` function over the correct interface given the correct SCSI id and SCSI LUN. The disadvantage of this interface is that it is not documented very well. As a result, it is relatively non-trivial to debug the code. In terms of functional overhead, this is slightly more overhead as compared to the previous option. This option has been implemented partially and can be utilized by uncommenting DISKIO (`scsi_interface.h` - Line 48). Though implemented, this option has not been completely debugged. More information is needed about the interface before the code can be made error-free.

7.7.3.3 I/O to and from a SCSI disk using the SCSI Generic interface

The third option consists of the SCSI Target mid-level using the SCSI Generic upper level. This is similar to the manner in which the USTE worked. For the kernel space implementation, the newer SG-2 interface was used. The SG-2 interface enables Asynchronous I/O between the application using the SCSI generic interface and the SCSI disk. The SG-2 interface is based on the `sg_io_hdr struct` depicted in Figure 7-15.

The organization of the struct is similar to the manner in which the `Scsi_Cmnd` struct is organized (see Figure 7-16). The SG-2 interface also enables the use of scatter-gather buffers to transmit and receive data. Asynchronous I/O is supported by using the SIGIO signal in conjunction with non-blocking reads and writes. The mechanism used to execute SCSI commands using the SG-2 interface is similar to the mechanism used by the USTE (described in Chapter 6) using the older interface. An `sg_io_hdr_t` struct is filled appropriately by the STML and is handed to a previously "`open`"ed SCSI device using a `write` command. When asynchronous I/O is used, this `write` needs to be non-blocking. When the command has been completed, SCSI Generic issues a SIGIO signal. The STML then needs to execute a `read` command so that the result of the execution of the SCSI command can be retrieved. Using this `read-write` interface in the kernel space involves some amount of manipulation of the traditional system call interface (`sys_read` and `sys_write`). The advantage of this technique is that the interface is well defined with good documentation. In addition, the SCSI generic interface allows application-specific definitions of retries and timeouts. This gives the STML a high enough interface in the SCSI processing stack where it does not have to deal with error handling on the SCSI Initiator side. The disadvantage remains a higher functional overhead as compared to the other two options.

There are some issues with the asynchronous processing of commands. This means that a STML thread is not blocked while a command is being processed. The SG-2 interface then transmits an SIGIO signal when the processing of the command is completed. The STML thread then needs to retrieve the response using the SG-2 interface. While this is not a problem when there is a single FETD operating with an Initiator that does not allow command queuing, in the general case, this proves to be a problem. That is because there are certain sections of the SCSI generic code that do not expect to receive signals. As a result, the STT (in the context of which the SCSI Generic functions are called) cannot receive signals. A separate entity is needed to process signals. When this mode of operation is used, the STML has to spawn a separate thread (SCSI Signal Processing Thread). The SCSI Signal Processing Thread (SSPT) is responsible for processing the received SIGIO and awakening the STT so that it can receive the response to the processed command using the SG-2 interface described previously. This option has been implemented and can be used by uncommenting GENERICIO (`scsi_interface.h` - Line 50).

7.8 Implementation of the SEP front-end in Kernel Space

The SEP front-end Target driver (FETD-SEP) in the kernel space was designed to be similar to the SEP front-end implementation in the USTE in structure. The FETD-SEP registers with the STML and "`detect`"s one device which represents a server (`sep_server_thread`) that is spawned by the `detect` function. The `sep_server_thread` awaits connections on the TCP port 4000. When a connection is accepted, it then spawns one thread to receive SCSI commands (represented by `sep_rx_thread)` and another to transmit SCSI responses and status (represented by `sep_tx_thread`). The semantics of the operation are similar to those used in the USTE. ABORTs and certain command types supported by SEP (e.g., third-party commands) are not supported by the implementation. Flow control is not used between the Initiator and the Target. Since SEP does not use the SCSI id, all LUNs are represented on id 0.

7.9 Implementation of the Fibre Channel front-end in Kernel Space

The Fibre Channel front-end Target driver (FETD-FC) was implemented using the Target Mode operation of the QLogic ISP 2200A Card. The driver for this was based on the HBA driver written by Chris Loveland (University of New Hampshire). The FETD-FC registers with the STML and then "`detects`" any ISP 2200A device that may be connected to the PCI bus. Each such device is registered with the STML. The firmware on each device is then loaded, configured and executed. An interrupt-handler is registered with the kernel. It was also realized that a considerable portion of what the FETD needed to do would have to be done in interrupt-context. To remedy this situation, each registered device spawns a `qlgc_process_thread`. This is the primary processing entity within the FETD-FC. The QLogic firmware interface (briefly described in Appendix C) consists of creating a receive and transmit queue within the kernel and then exchanging I/O Control Blocks (IOCBs) with the ISP 2200A. The IOCBs are 64-byte instructions that can be issued to the ISP 2200A firmware. The FETD-FC is informed about an update to the receive queue (i.e., it has received an IOCB) and about the successful execution of an IOCB in interrupt context. If IOCBs are received, the interrupt handler copies the received IOCBs to an IOCB queue which is maintained by the FETD-FC. If, on the other hand, a received interrupt indicates that an IOCB has completed execution, the interrupt handler prepares this IOCB for being dequeued from the IOCB queue. The `qlgc_process_thread` is then awakened and the required actions are taken. The ISP 2200A firmware deals with most of the specifics of the SCSI-FCP so that the FETD-FC is left free to deal with the SCSI commands themselves. Flow-control and link-level details are also dealt with by the ISP 2200A firmware.

7.10 Implementation of the iSCSI front-end in Kernel Space

The iSCSI front-end Target driver (FETD-iSCSI) was implemented to the iSCSI Draft version 3. This version was chosen because there were some questions about the latest changes in iSCSI Draft version 5 and continuing into future drafts. The FETD-iSCSI was implemented with a structure similar to the FETD-SEP. The FETD-iSCSI registers with the STML and "detect"s one device which represents a server (iscsi_server_thread) that is spawned by the detect function. The iscsi_server_thread awaits connections on the TCP port 4002. When a connection is accepted, it then spawns one thread to receive SCSI commands (represented by iscsi_rx_thread) and another to transmit SCSI responses and status (represented by iscsi_tx_thread). Each connection is part of a session that is established between an iSCSI Initiator and an iSCSI Target. Task Management functions implemented by the STML have also been implemented by the FETD-iSCSI.

```
typedef struct sg_io_hdr {
int            interface_id; /* [i] 'S' for SCSI generic (reqd) */
int            dxfer_direction; /* [i] data transfer direction */
unsigned char  cmd_len; /* [i] SCSI command length ( <= 16 bytes) */
unsigned char  mx_sb_len; /* [i] max length to write to sbp */
unsigned short iovec_count; /* [i] 0 implies no scatter gather */
unsigned int   dxfer_len; /* [i] byte count of data transfer */
void           *dxferp; /* [i], [*io] points to data transfer memory
or scatter gather list */
unsigned char  *cmdp; /* [i], [*i] points to command to perform */
unsigned char  *sbp; /* [i], [*o] points to sense_buffer memory */
unsigned int   timeout; /* [i] MAX_UINT->no timeout (unit: msec) */
unsigned int   flags; /* [i] 0 -> default, see SG_FLAG... */
int            pack_id; /* [i->o] unused internally (normally) */
void           *usr_ptr; /* [i->o] unused internally */
unsigned char  status; /* [o] scsi status */
unsigned char  masked_status; /* [o] shifted, masked scsi status */
unsigned char  msg_status; /* [o] messaging level data (optional) */
unsigned char  sb_len_wr; /* [o] bytes actually written to sbp */
unsigned short host_status; /* [o] errors from host adapter */
unsigned short driver_status; /* [o] errors from software driver */
int            resid; /* [o] dxfer_len - actual_transferred */
unsigned int   duration; /* [o] time taken by cmd (unit: millisec) */
unsigned int   info; /* [o] auxiliary information */
} sg_io_hdr_t; /* 64 bytes long (on i386) */
```

Figure 7-15: Definition of the sg_io_hdr_t struct

```
typdef struct scsi_cmnd {
unsigned int          target;
unsigned int          lun;
unsigned int          channel;
unsigned char         cmd_len;
unsigned char         old_cmd_len;
unsigned char         sc_data_direction;
unsigned char         sc_old_data_direction;
unsigned char         cmnd[MAX_COMMAND_SIZE];
unsigned              request_bufflen; /* Actual request size */
struct timer_list     eh_timeout; /* Used to time out the command. */
void                  *request_buffer; /* Actual requested buffer */
unsigned char         data_cmnd[MAX_COMMAND_SIZE];
unsigned short        old_use_sg;
unsigned short        use_sg;  /* Number of pieces of scatter-gather */
unsigned short        sglist_len; /* size of malloc'd scatter-gather list */
unsigned short        abort_reason;
unsigned              bufflen; /* Size of data buffer */
void                  *buffer; /* Data buffer */
unsigned              underflow;
unsigned              old_underflow;
unsigned              transfersize;
int                   resid;
struct request        request;
unsigned char         sense_buffer[SCSI_SENSE_BUFFERSIZE];
unsigned              flags;
unsigned              done_late:1;
void                  (*scsi_done) (struct scsi_cmnd *);
Scsi_Pointer          SCp; /* Scratchpad used by some host adapters */
unsigned char         *host_scribble;
int                   result; /* Status code from lower level driver */
unsigned char         tag; /* SCSI-II queued command tag */
unsigned long         pid; /* Process ID, starts at 0 */
} Scsi_Cmnd;
```

Figure 7-16: Definition of `Scsi_Cmnd struct`

CHAPTER 8

TESTING AND PERFORMANCE ANALYSIS

8.1 Overview

This thesis has implemented the STML along with the FETDs for the SEP, Fibre Channel and the iSCSI Protocols. The SEP and the iSCSI implementations were based on software implementations of the TCP/IP protocols. The Fibre Channel driver was implemented for the Target Mode operation of the QLogic ISP 2200A card. This Chapter deals with the issues of testing and performance analysis of these implementations.

8.2 Approaches to Testing

Testing a product involves several different facets. Due to the vast scope of the design phase in this thesis, the extent of testing for this thesis is limited. This section outlines the testing that needs to be performed along with a description of those aspects that have been performed as part of this thesis.

8.2.1 Conformance Testing

This is the most basic form of testing which involves deciding whether a product has implemented the specifications of a standard to the fullest aspect. For this project, there are four different standards families which can be referenced: the SCSI family (SAM-2, SPC-2, SBC-2), the SEP draft, the Fibre Channel family (FC-PH, FC-AL-2, FC-FLA, FC-PLDA, SCSI-FCP), and the iSCSI draft. Clearly, the wide scope of standards families across which this thesis works renders it almost impossible to test in any complete manner.

From the standpoint of the STML, the threadbare SCSI commands required for normal operation have been implemented. SCSI Mode Page definitions have not been implemented. Error recovery mechanisms have been implemented but not all the semantics of error recovery have been implemented. As far as the SEP implementation is concerned, flow control and Task Management functions have not been implemented. For the Fibre Channel implementation, the recovery procedures allowed by the QLogic firmware interface have been implemented. Not all the semantics of SCSI-FCP-2 can be implemented. In addition, Class 2 service has not been implemented (Class 2 Service is intended for use with Sequential Access devices like tape drives, whereas the present implementation is for Direct Access devices). For the iSCSI implementation, several negotiable features such as immediate data, header and data digests, have not been implemented.

8.2.2 Stress Testing

The idea behind this type of testing is to create conditions of heavy load and observe the behavior of the entities involved. The primary objective behind such testing is to test the implementation at its limits (for example: for the iSCSI limitations, when the number of connections that the FETD-iSCSI can handle reaches it maximum) and to see that the device handles error conditions caused by resource unavailability gracefully. This type of testing has not been performed on any of the implementations.

8.2.3 Interoperability Testing

This testing checks to see if a given implementation can work successfully with others. The STML has been tested for interoperability with a wide range of SCSI Initiators. The Fibre Channel implementation has been extensively tested for interoperability with different HBAs and Fibre Channel switches from different vendors to see if they recognize the Target. The SEP and iSCSI implementations could not be tested similarly due to a lack of access to devices that implement the SEP and iSCSI protocols.

8.2.4 Operability Testing

This testing checks to see if a given implementation does indeed do what is expected of it. In the case of this thesis, the expectation was to create a SCSI Target device which is able to store data without corruption. Extensive testing has been done for this aspect of testing. The "disk" represented by the STML has been used as a valid file system that can be accessed by different hosts. Consistency tests have also been run on such a "disk". This kind of testing tests the basic path of data through a given implementation. For scenarios involving errors and error recovery, not much testing has been done.

8.3 Performance Analysis

A detailed performance analysis of these and several other protocols will be performed by Anshul Chaddha as a part of his thesis. This thesis references some of the basic data that has been gathered from his initial work.

The mode of operation involving I/O operations to and from memory (# define MEMORYIO uncommented) was used as a comparative basis for the different SCSI Transport Protocols.

No.	Target Emulator	Data Rate (MB/s)
1	USTE/SEP	19
2	KSTE/SEP	21
3	KSTE/FC	45
4	KSTE/iSCSI	19

Figure 8-1: Data Rates with different implementations

In addition, the values of CPU utilization were observed for the above performance tests using the "top" utility provided with the Linux Operating system. The maximum values observed for Fibre Channel (< 5%) were significantly lower than those observed for iSCSI and SEP (~25 %).

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

9.1 Conclusions

This thesis presents a general architecture for implementing SCSI Targets that utilize SCSI Transport Protocols. This thesis documents the design and implementation of:

1. An interface for a SCSI Target in the user-space.
2. A SEP front-end for the above user-space interface.
3. A SCSI Target Mid-Level (STML) for the Linux Operating System kernels
4. SEP, iSCSI and Fibre Channel front-end Target drivers for the implemented STML.

The STML has been implemented in three separate modes of operation with I/O going to and from a real SCSI disk connected to a system, I/O going to and from a file on a local disk that contains the logical blocks, and I/O to and from memory.

Some limited performance analyses were conducted on the different implementations. The SEP and iSCSI protocol implementations yielded similar values for data rates (19 – 21 MB/s) independent of whether the implementation was in the user-space or the kernel-space (if applicable). The Fibre Channel protocol implementation yielded a data rate of 45 MB/s.

In addition, CPU utilization values observed for Fibre Channel (< 5%) were significantly lower than those observed for iSCSI or SEP (~25%).

The above two conclusions together imply that in order to get viable iSCSI or SEP implementations, TCP/IP would have to be implemented in hardware.

Implementation of the above protocols yielded some interesting conclusions about the protocols themselves:

1. For SCSI implementations which deal with multiple SCSI Transport Protocols, a given SCSI Target does not have a common mechanism to reference an Initiator. As a result, not all the semantics of the Task Management functions provided by SCSI can be implemented. This problem is currently being addressed by T10.
2. While attempting to implement the iSCSI protocol using draft 5, a number of issues were observed with the suggested iSCSI PDU header implementation. These problems were conveyed back to the IETF. A new header format has since been suggested and is expected to be implemented in version 6 of the iSCSI draft standard. It was because of these issues that the present implementation is based on version 3 of the iSCSI draft standard.

9.2 Future Work

Future work from this thesis can take several different directions. These are discussed below.

9.2.1 Performance Analysis

This thesis has not looked at performance issues and the effect they can have on the design choices. The choices made in the design of the STML were made based on intuition and program complexity rather than considering actual performance. Examples of such choices would be having separate threads for each FETD instead of having a single STT. Another improvement that could be made is to have separate queues for commands from each separate FETD. This design could also significantly simplify FETD design since a significant portion of the implementation of the FETDs involves dealing with the command queues for the individual FETDs. In addition, this could reduce search time for the STML and thus, affect performance. Another consequence of studying performance analysis would be to reveal possible bottlenecks in the STML design. For the TCP/IP based SCSI Transport Protocols, the present TCP/IP implementations are in software. It is expected that gradually TCP/IP will be implemented in hardware. Performance analysis of the present implementations could yield an idea of the performance improvements to be gained by a hardware implementation of TCP/IP.

9.2.2 Development of Testing Tools and Test Suites

The STML along with the corresponding drivers provides the ability to test the different SCSI Transport Protocols. Several tests in the Private Loop Direct Attach (PLDA) Test Suite written by the Fibre Channel Consortium (University of New Hampshire) could be implemented using the KSTE. Tools for testing the iSCSI protocol could also be implemented using the FETD-iSCSI.

9.2.3 Protocol Development

Another aspect that needs a good deal of future work is protocol development. An extremely limited set of SCSI commands has been implemented. This set needs to be extended. It may also be possible to extend the design to include Sequential Access devices in addition to Direct Access devices. The SCSI Error Recovery protocol has been implemented minimally without all of the semantics. In addition, the SCSI Mode Page definitions have not been implemented. These are also possible projects.

In addition, more FETDs for different protocols need to be implemented to guarantee the generic nature of the design. Examples of such implementations would be for the mFCP protocol and the upcoming Infiniband protocol. In addition, it is expected that software implementations will give way to hardware implementations. The FETDs for SEP and iSCSI will then have to be re-written to work with such implementations. In addition, there are several Fibre Channel cards with Target Mode operation. Drivers for these cards need to be written in the interest of providing wider support for the designed STML.

9.2.4 Kernel Design Projects

One of the interesting implications of the KSTE is that it provides a common interface behind which a large number of storage resources can potentially reside. It thus provides an interface for managing these resources. The functionality associated with this management interface can be isolated into a SAN Layer which could deal with the issues pertaining to redundancy, mirroring, fail-over and also, naming and discovery of the storage resources.

REFERENCES

1. ANSI X3T10-994D, 'Information Technology: The SCSI Architecture Model – 2 (SAM-2)', Revision 15, November 1995.

2. ANSI X3.230-1994, 'Information Technology - Fibre Channel Physical and Signaling Interface', (FC-PH), 1994.

3. ANSI X2.269-1995, 'Information Technology – dpANS Fibre Channel Protocol for SCSI', (SCSI-FCP), 1995.

4. Benner, Alan F., 'Fibre Channel: Gigabit Communications and I/O for Computer Networks', McGraw-Hill, 1995.

5. Dedek, Jan, 'Basics of SCSI', Ancot Corporation, 1992.

6. Eissfeldt, Heiko, 'The Linux SCSI Programming HOW-TO', http://www.ibiblio.org/pub/Linux/docs/HOWTO/other-formats/html_single/SCSI-Programming-HOWTO.html, May 1996.

7. Internet Draft, 'iSCSI (Internet SCSI)', Satran, J., Smith, D., Meth, K., Sapuntzakis, C., Wakeley, M., Von Stamwitz, P., Haagens, R., Zeidner, E., Ore, L., and Klein, Y., http://www.ietf.org/internet-drafts/draft-ietf-ips-iSCSI-03.txt, January 2001.

8. Internet Draft, 'mFCP – Metro FCP protocol for IP Networking', Mullendore, R., Monia, C., and Tseng, J., http://www.ietf.org/internet-draft/draft-monia-ips-mfcp-00.txt, November 2000.

9. Internet Draft, 'The SCSI Encapsulation Protocol', (SEP), Wilson, A., http://www.ietf.org/internet-drafts/draft-wilson-sep-00.txt, 2000.

10. Gilbert, Douglas, http://www.torque.net/scsi, 'The Linux SCSI subsystem in 2.4', 2000.

11. Gilbert, Douglas, 'The Linux SCSI Generic Interface', http://www.torque.net/sg/p/scsi-generic_v3.txt, 2001.

12. Khattar, R. K., Murphy, M. S., Tarella, G. J., and Nystrom, K. E., 'Introduction to Storage Area Network', Redbooks Publications (IBM), 1999.

13. NCITS T10/995D (Revision 11a), 'SCSI-3 Primary Commands (SPC)', 1997.

14. NCITS T10/996D, 'SCSI-3 Block Commands (SBC)', 1997.

15. NCITS T10/1144D, ‘Fibre Channel Protocol for SCSI, Second Version (FCP-2)’, Revision 5, November 2000.

16. NCITS T10/1157D, ‘SCSI Architecture Model – 2 (SAM-2)’, Revision 15, November 2000.

17. NCITS T10/1236D, ‘SCSI Primary Commands – 2 (SPC-2)’, Revision 18, May 2000.

18. NCITS T10/1417D, ‘SCSI Block Commands – 2 (SBC-2)’, Revision 2, October 2000.

19. Stai, Jeffrey D., ‘The SCSI Bench Reference’, Endl Publications, 1996.

20. Stai, Jeffrey D., ‘The Fibre Channel Bench Reference’, Endl Publications, 1996.

21. Waxman, Janet, and McArthur, John, ‘Storage Area Networks: Opportunity for the Indirect Channel’, IDC, January 2000.

APPENDIX A

SCSI COMMANDS FOR DIRECT ACCESS DEVICES

The following is a table of SCSI commands that are relevant to the Direct Access SCSI devices.

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	SYCHRONIZE 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 ATTACHED

APPENDIX B

INSTALLING AND EXECUTING THE KSTE AND THE USTE

1. The files required to install the USTE and the KSTE can be accessed from the IOL website at http://www.iol.unh.edu/consortiums/fc/fc_linux.html
2. The USTE files are located in the `user_space_emulator` directory. The files can be compiled by running "`make`". The executable is `scsi_server` which listens for SEP CONNECT-NEGOTIATE message on TCP Port 4000.
3. The KSTE files are located in the `kernel_space_emulator` directory. The files can be compiled by running "`make`". The modules for the STML, the FETD-SEP, the FETD-FC, and the FETD-iSCSI are scsi_target.o, sep_target.o, qlogicfct.o, and iscsi_target.o, respectively.

APPENDIX C

QLOGIC ISP 2200A FIRMWARE INTERFACE

The ISP2200/ISP2200A firmware supports two interfaces to the host system software: mailbox commands and host memory queues. The mailbox commands configure the ISP2200/ISP2200A firmware and hardware and issue priority IOCBs. The host memory queue interface is the operational interface. This interface has two queues in the host memory: the request queue and the response queue. The request queue is where the host system places IOCBs to be processed by the ISP2200/ISP2200A firmware. The response queue is where the ISP2200/ISP2200A firmware places IOCBs to be processed by the host system.

The SCSI-FCP host adapter firmware for the ISP2200/ISP2200A supports Initiator and Target modes of operation. These modes transport SCSI command descriptor blocks (CDBs) to the desired FC-AL SCSI target device and transfer data associated with the command between the host memory and the FC-AL SCSI device. The ISP2200/ISP2200A firmware does not interpret the CDB nor the data. The host adapter firmware incorporates the following features:

- A user interface that is consistent within the QLogic ISP family of products
- Fibre Channel link level support
- SCSI initiator mode and target mode support

The ISP 2200A is first reset and the firmware is then loaded, verified, configured and executed. This part is achieved by using Mailbox commands. The interrupts are disabled in this process. Interrupts are then enabled. The host system and the firmware then communicate with each other using IOCBs on both the response and the request queue.

APPENDIX D

ACRONYMS USED

The following is a list of the acronyms that have been used in this thesis:

ACA	-	Auto Contingent Allegiance
API	-	Application Programming Interface
CDB	-	Command Descriptor Block
FC	-	Fibre Channel
FEID	-	Front End Initiator Driver
FETD	-	Front End Target Driver
FETD-SEP	-	SEP Front End Target Driver
FETD-FC	-	Fibre Channel Front End Target Driver
FETD-iSCSI	-	iSCSI Front End Target Driver
HBA	-	Host Bus Adapter
IOCB	-	I/O Control Block
IP	-	Internet Protocol
IU	-	Information Unit
iSCSI	-	Internet SCSI
KSTE	-	Kernel Space Target Emulator
LBA	-	Logical Block Address
LUN	-	Logical Unit Number
mFCP	-	Metro Fibre Channel Protocol
NACA	-	Normal ACA
NAS	-	Network Attached Storage
PDU	-	Protocol Data Unit
PLDA	-	Private Loop Direct Attach
SAM	-	SCSI Architecture Model
SAN	-	Storage Area Networks
SASI	-	Shugart Associates Systems Interface
SBC	-	SCSI Block Commands
SCSI	-	Small Computer Systems Interface
SCSI-FCP	-	Fibre Channel Protocol for SCSI
SEP	-	SCSI Encapsulation Protocol
SG-1	-	SCSI Generic 1 (refers to the older interface available in all Linux kernels)
SG-2	-	SCSI Generic 2 (refers to the new SCSI Generic interface available in all Linux kernels subsequent to 2.4.0-test9)
SIUL	-	SCSI Initiator Upper Level
SIML	-	SCSI Initiator Mid-Level

SoIP	-	Storage over IP
SPC	-	SCSI Primary Commands
SSPT	-	SCSI Signal Processing Thread
STML	-	SCSI Target Mid-Level
STT	-	SCSI Target Thread
STP	-	Scheduled Transfer Protocol
TAN	-	Target Acquired Name
TCP	-	Transmission Control Protocol
ULP	-	Upper Level Protocol
USTE	-	User Space Target Emulator