USENIX Association

Proceedings of the 5th Annual Linux Showcase & Conference

Oakland, California, USA
November 5–10, 2001

DESIGN AND IMPLEMENTATION OF A LINUX SCSI TARGET FOR STORAGE AREA NETWORKS

ASHISH PALEKAR
Trebia Networks Inc
33 Nagog Park
Acton, Massachusetts 01720, USA
Email: apalekar@trebia.com

NARENDRAN GANAPATHY
ANSHUL CHADDA
ROBERT D. RUSSELL
InterOperability Laboratory
University of New Hampshire
Durham, New Hampshire 03824, USA
Email: {ng3,achadda,rdr}@iol.unh.edu

ABSTRACT

This paper describes the architecture of a set of kernel components for developing and testing storage area network transport protocols under Linux. This software is intended for several uses: as a general prototype for network transport protocol development; as a reference implementation of the iSCSI protocol currently under development for standardization by IETF; as a basis for conformance testing for iSCSI; and as a testbed for development of interoperability test suites for iSCSI.

Keywords: storage area networks, SCSI, iSCSI.

1 INTRODUCTION

The widespread adoption of the World Wide Web and e-commerce has created a huge demand for vast storage repositories that provide real-time on-line access 24 hours a day, 7 days a week, 52 weeks a year. This demand has overwhelmed traditional storage mechanisms, and has prompted the development of promising new technologies, one of which is the "Storage Area Network" or SAN [1].

The key idea behind SAN development is to replace the traditional data bus between a host computer and its storage devices with a high-speed data network. In a SAN, the storage device is directly attached to that network, and interacts with multiple host computers using standard network protocols, rather than specialized bus protocols. This leverages the substantial installed network base already in place, and opens up the prospect of geographically disperse, enterprise-wide storage. With the advent of gigabit-per-second and higher network transfer rates, the effective transfer rates between hosts and storage increase while costs are decreased. To enable this paradigm, the approach that is being taken is to design new protocols that encapsulate SCSI commands and data for transport over the network. These are collectively referred to as "SCSI Transport Protocols".

The first technology to utilize this idea was Fibre Channel [2], which uses a gigabit-per-second link to carry SCSI commands and data over distances up to 10 KM. This technology requires special hardware adapters on both the host computers and the target storage devices, and installation of new fiber-optic cabling.

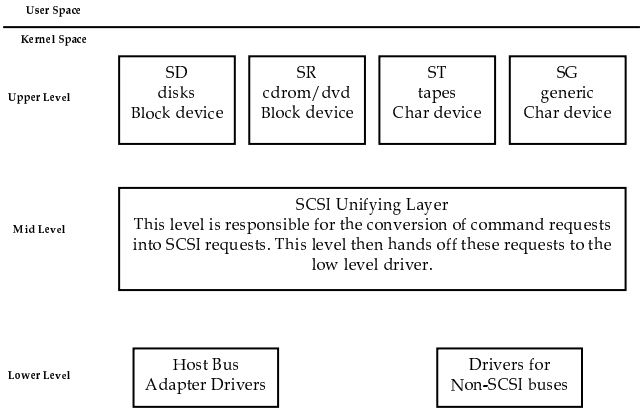

FIGURE 1: Architecture of the Linux SCSI Subsystem

To avoid the special hardware requirements of Fibre Channel, several recent proposals have been made to leverage the huge existing network infrastructure, which is largely built on Ethernet at the host computers and which forms the basis for the global Internet.

The new storage transport protocols of interest to us are those that utilize the existing TCP/IP protocol stack. The two main proposals in this category are the SCSI Encapsulation Protocol (SEP) [3] and the iSCSI Protocol [4], which is currently under development as a Standard by the Internet Engineering Task Force (IETF) [5]. Our initial effort was concentrated on SEP because it is relatively simple compared to iSCSI, and also its specification was more stable. This first effort allowed us to design an architecture and test it quickly. Using this experience as a basis, we then generalized the architecture to encompass both pro-

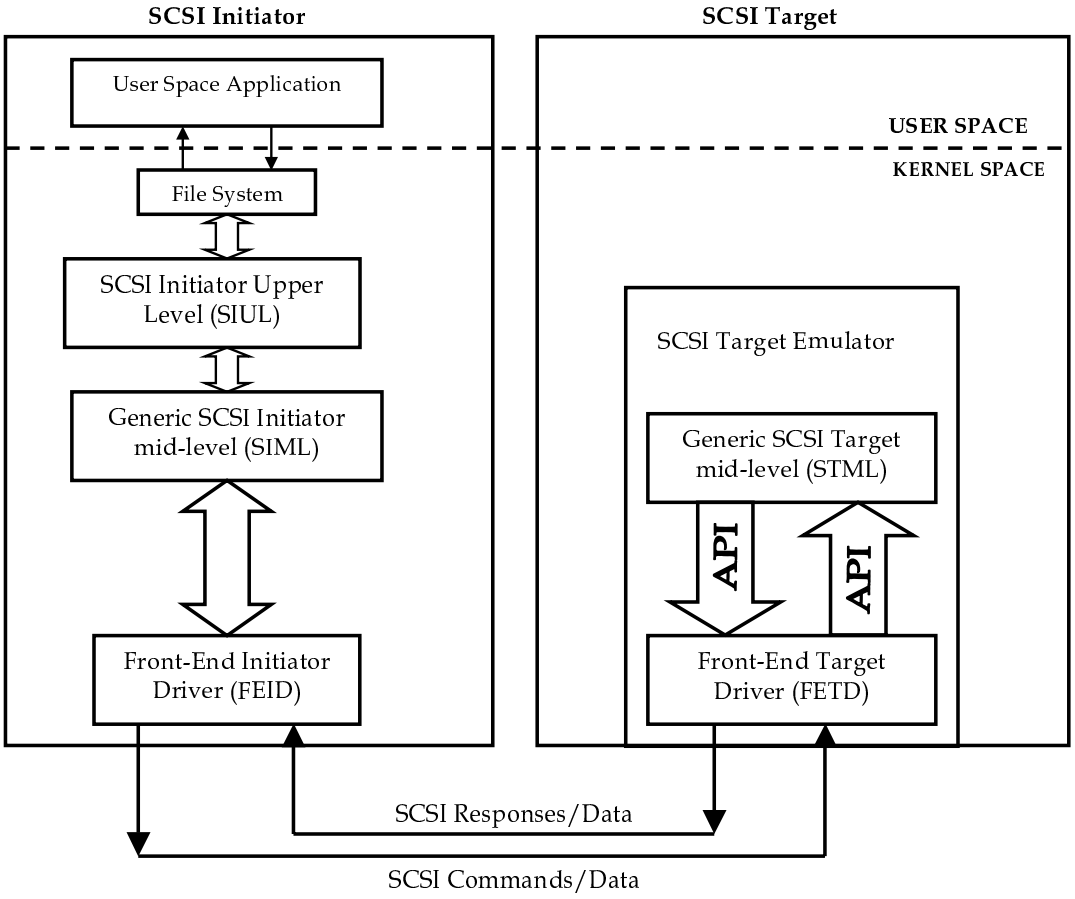

FIGURE 2: Architecture of the Linux SCSI Initiator and SCSI Target Emulator

tocols, as well as allowing for any future protocol proposals. We then utilized this new architecture to implement a prototype system of both SEP and draft 6 of the iSCSI proposal, and that is the work which is reported on in this paper. Complete details are available in the thesis [6].

2 DESIGN

The design of our kernel-level SCSI Target Emulator was based on two general guidelines: (1) be as general as possible, so that different SCSI transport protocols could be incorporated and could coexist simultaneously; (2) utilize as much as possible of the existing Linux kernel mechanisms.

Since the purpose of any SCSI transport protocol is to interface with the SCSI subsystem, our architecture was heavily influenced by the general organization of SCSI subsystems as defined by the SCSI Architectural Model (SAM-2) [7] and its corresponding Initiator-side architecture in the Linux kernel [8].

Figure 1 shows the three-layered architecture of the Linux Initiator SCSI Subsystem. The upper-level layer provides a generic “read/write” interface that is traditionally used by a file system. The mid-level converts the reads and writes into SCSI command and data transfer sequences. The lower-level consists of the drivers for the Host Bus Adapters (HBAs) of the specific SCSI devices. These HBAs control the delivery of the SCSI commands and data over the SCSI bus to a particular SCSI device. It is therefore natural for our kernel-level Front-End Initiator Driver (FEID) to be designed as a special HBA that encapsulates the SCSI commands and data into the appropriate transport protocol and then delivers them over a network rather than over a SCSI bus. This is also the way Fibre Channel Initiator interfaces are built.

Traditionally a SCSI Target is a device, such as a disk drive or tape drive, so that its internal organization does not necessarily follow any particular software architecture. Also, “traditional” SCSI Targets are not designed to work over multiple SCSI transport protocols. Therefore, in accordance with our first design guideline, an important objective of our architecture was to keep information specific to a particular SCSI transport protocol separate from information that SCSI needs to process a command.

The general architecture of our SCSI Target Emulator software and its relationship with the architecture of the Initiator is shown in Figure 2. The Target Emulator con-

```
register_target_template(struct Scsi_Target_Template *)
deregister_target_template(struct Scsi_Target_Template *)
register_target_front_end(struct Scsi_Target_Template *)
deregister_target_front_end(struct Scsi_Target_Device *)
rx_cmnd(struct Scsi_Target_Device *, __u64, char *, int)
scsi_rx_data(struct Target_Scsi_Cmnd *)
scsi_target_done(struct Target_Scsi_Cmnd *)
scsi_release(struct Target_Scsi_Cmnd *)
rx_task_mgmt_fn(struct Scsi_Target_Device *, int, void *)
```

Table 1: Set of API functions by which an FETD calls the STML

```
struct  Scsi_Target_Template
    {
  int   (*detect)(struct Scsi_Target_Template *);
  int   (*release)(struct Scsi_Target_Device *);
  int   (*xmit_response)(struct Target_Scsi_Cmnd *);
  int   (*rdy_to_xfer)(struct Target_Scsi_Cmnd *);
  int   (*task_mgmt_fn_done)(struct Scsi_Target_Message *);
 void   (*report_aen)(int, __u64);
    };
```

Table 2: Template of API functions by which the STML "calls back" an FETD

sists of two components: a generic SCSI Target Mid-level (STML) and a Front-End Target Driver (FETD). There is one FETD for each SCSI transport protocol (i.e., SEP and iSCSI), and all details of this transport protocol on the Target are handled completely within this component.

The STML is therefore totally independent of the SCSI transport protocol used in the FETD. Its role is to process SCSI commands and data received from a FETD "on behalf of the Initiator". This means processing the received SCSI commands and data, and handing off the generated responses and data back to the FETD for delivery back to the Initiator. The STML is also responsible for SCSI error handling, and for maintaining SCSI state information.

To accomplish its tasks, our kernel-level version of the STML is designed to utilize the existing local SCSI subsystem on the Target platform, in accordance with our second design guideline. The STML is organized as two threads: the SCSI Target Thread (STT), which is the main vehicle for processing SCSI commands, and the SCSI Signal Processing Thread (SSPT), which is used to deal with the asynchronous arrival of SIGIO signals, as explained in the next section.

An important part of the design of the Target Emulator was specification of an "API" between the STML and the FETD. This API consists of a set of data structures and two sets of functions – one for use by the FETD to hand off

```
struct  Scsi_Target_Message
    {
struct  Scsi_Target_Message *next;
struct  Scsi_Target_Message *prev;
  int   message;
struct  Scsi_Target_Device device;
 void   *value;
    };

struct  Target_Scsi_Cmnd
    {
  int   state;
  int   id;
__u64   dev_id;
struct  Scsi_Target_Template *dev_template;
__u64   target_id;
__u64   lun;
uchar   cmd[MAX_COMMAND_SIZE];
  int   len;
struct  Target_Scsi_Cmnd *next;
struct  Scsi_Request *req;
    };

struct  Scsi_Target_Device
    {
__u64   id;
struct  Scsi_Target_Device *next;
struct  Scsi_Target_Template *template;
    };
```

Table 3: Additional API data structures

incoming SCSI commands and data to the STML, and the other for use by the STML to hand back the generated SCSI responses and data to the FETD.

Table 1 lists the nine API functions available for use by an FETD in order to pass SCSI commands and data to the STML. The six API functions required by the STML in order to "call back" the FETD are provided in the **Scsi_Target_Template** structure shown in Table 2. There are three other data structures used by these functions, as summarized in Table 3.

A quick explanation of how the FETD's interact with the STML using the API functions follows.

When an FETD module is dynamically loaded into the Linux kernel using the **insmod** command, the FETD calls **register_target_template()** to register itself with the STML. (Later, when this module is removed, the FETD must call **deregister_target_template().**) The parameter to both these functions is the "jump table" of "call back" functions shown in the **Scsi_Target_Template** structure (Table 2). As part of processing the **register_target_template()** function, the STML will call back the FETD's **detect()** function in order to detect all "de-

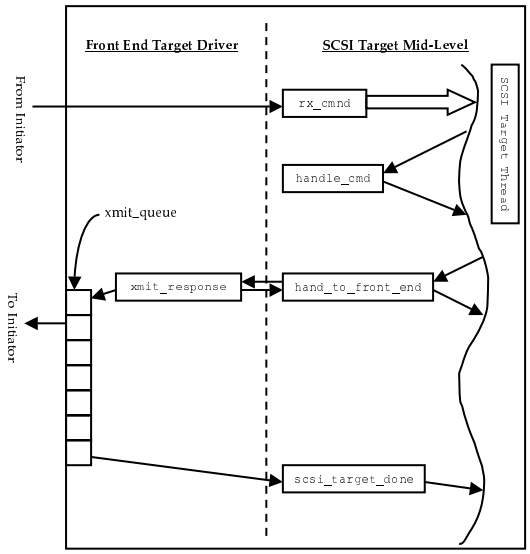

FIGURE 3: Processing a READ-type command on the Target Emulator

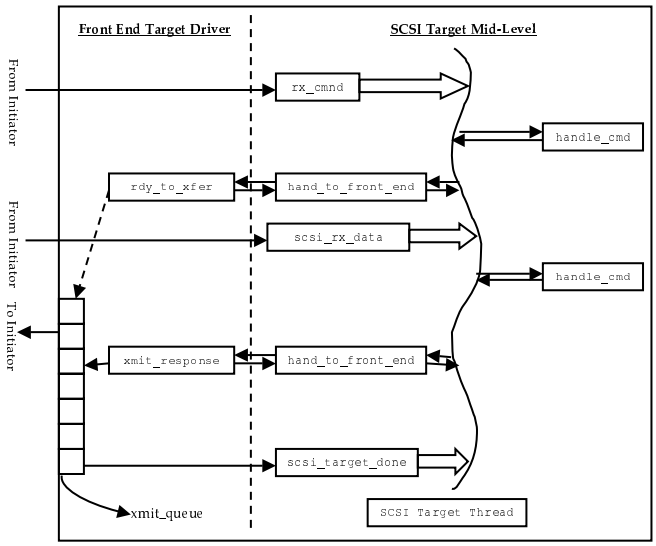

FIGURE 4: Processing a WRITE-type command on the Target Emulator

vices" being handled by this FETD. The FETD's **detect()** function in turn registers each device with the STML by calling **register_target_front_end()**.

Once registered, an FETD operates asynchronously with respect to the STT. An FETD device interrupt handler calls the STML's **rx_cmnd()** function whenever it receives a new SCSI command from the network. This command will be enqueued for processing by the STT thread. If the command is a READ-type command, where data is being sent from the Target to the Initiator, the STT thread allocates buffers and passes the request to the SCSI mid-level subsystem on the Target using the SCSI generic interface. When this subsystem informs the STT that the data is in the buffers, the STT calls back the **xmit_response()** function of the FETD, which is responsible for encapsulating the data and asynchronously sending it over the transport network to the Initiator. Once this transmission is complete, the FETD notifies the STT by calling the **scsi_target_done()** function of the STML. This sequence of events is illustrated schematically in Figure 3.

Processing a WRITE-type command is somewhat more complicated, because the data from the Initiator will asynchronously follow the SCSI command itself. The necessary sequence of events is illustrated in Figure 4. As in the case of a READ-type command, the STT thread processes a WRITE by first allocating buffers, but it obviously cannot fill these buffers locally as was done for a READ. Instead, it passes these buffers back to the FETD by calling the **rdy_to_xfer()** function. This function uses flow control mechanisms unique to the particular SCSI transport protocol to initiate the transfer of data from the Initiator over the transport network. When this data arrives, the FETD passes it to the STML by calling the **scsi_rx_data()** function, which in turn will pass this data to the SCSI mid-level subsystem on the Target using the SCSI generic interface. When this subsystem informs the STT that the data transfer to the local SCSI device has been completed, the STT calls back the **xmit_response()** function of the FETD, which must send the SCSI status response back to the Initiator. Once this transmission is complete, the FETD notifies the STT by calling the **scsi_target_done()** function of the STML.

3 IMPLEMENTATION

All our coding was done in C for the Linux 2.4 kernel. It is freely available from the UNH IOL iSCSI Consortium at http://www.iol.unh.edu/consortiums/iscsi/ under the terms of the GNU Copyleft agreement.

We have implemented two Initiator-side drivers, and have tested three drivers for storage transport protocols. The drivers we implemented, for SEP and iSCSI draft 6, utilize the existing Linux software TCP/IP stack to communicate over an Alteon ACENIC 1000 BaseT Ethernet card. The existing driver was for the Fibre Channel protocol, and utilizes a QLogic Fibre Channel HBA communicating over a Gigabit Fibre Channel link.

On the Target side we have two implementations of the Target Emulator: one that operates entirely in user space, called the User Space Target Emulator (USTE), and one that operates entirely in kernel space, called the Kernel Space Target Emulator (KSTE). Both are organized into the two-part structure described in the previous section: a protocol-dependent FETD, and a protocol-independent STML. For the USTE the only FETD we have implemented is for SEP, and that software is not discussed further in this paper. We put most of our development effort into the KSTE so that we would have a software reference implementation with reasonable performance.

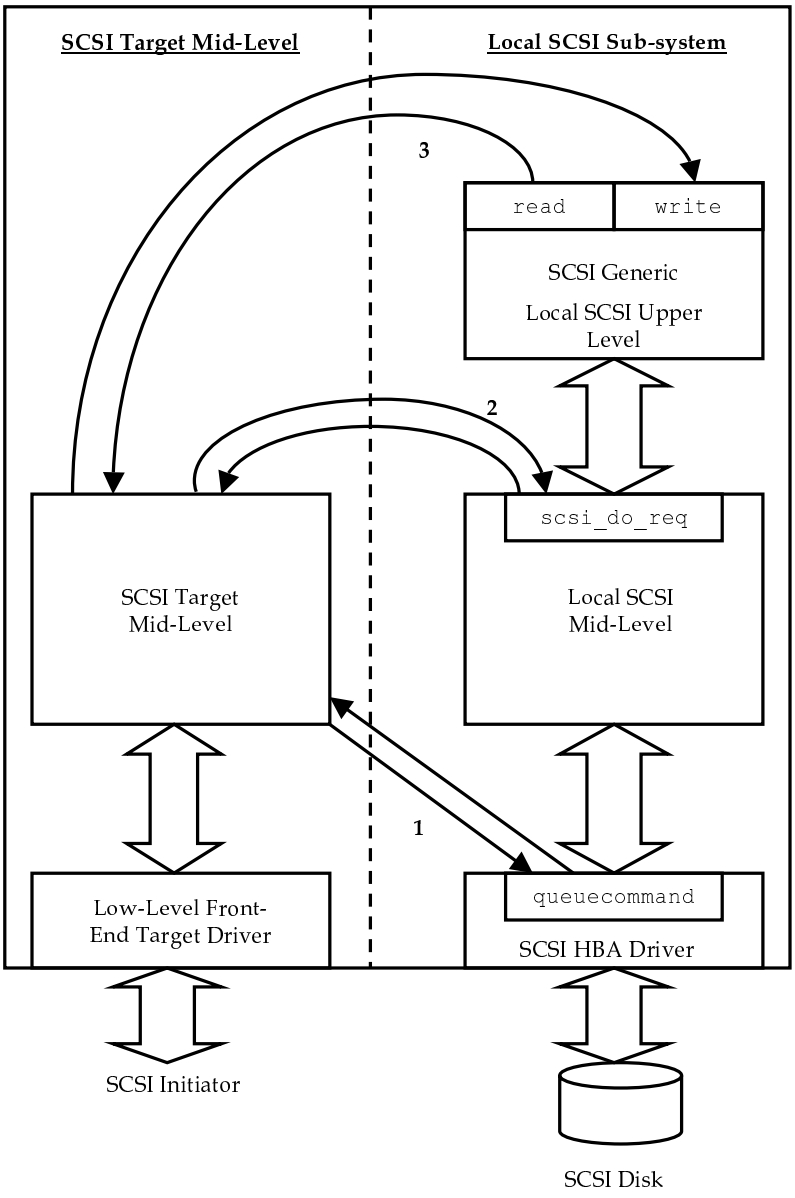

FIGURE 5: Options available for interfacing the STML to the local SCSI subsystem

For the KSTE we have implemented three FETDs – one for SEP, one for iSCSI draft 6, and one for Fibre Channel. It is the KSTE that is discussed in the rest of this paper.

Because the Linux kernel SCSI subsystem is organized into three layers, as already shown in Figure 1, there are three points at which our STML could interact with this subsystem, as shown in Figure 5. The first option would be to use the **queuecommand()** interface provided by every SCSI HBA. This option would have the lowest overhead of the three options, but would require that the STML essentially duplicate most of the functionality already provided by the SCSI mid-level of the local SCSI subsystem. This clearly violates our second design guideline and was rejected.

The second option would be to use the **scsi_do_req()** interface provided by the SCSI mid-level. This is the interface utilized by all SCSI upper-level components, and is the preferred interface on Linux systems. However, it is not yet well-documented. As a result, we have implemented a version using this option but it is not yet fully debugged. Thus we currently utilize the third option, the SCSI generic upper-level, via its new version 3 interface [9]. Besides a slightly higher overhead than option two, this option also introduces an additional complication because, when used for asynchronous processing, the Linux SCSI generic interface sends a SIGIO signal when it completes command processing. The STT cannot receive signals when utilizing certain sections of the SCSI generic code. Therefore, to handle this signal required the introduction of a second kernel

thread, called the SCSI Signal Processing Thread (SSPT). This thread simply catches the SIGIO and enqueues a command for the STT, thereby avoiding re-entrancy problems in the STT. A detailed picture of how these threads interact with each other and the FETDs via the APIs is shown in Figure 6.

The iSCSI FETDs are implemented as two kernel threads per session, one for receiving data from the Initiator and one to send data to the Initiator. At the present time an FETD is limited to only one TCP connection per iSCSI session, but future plans call for allowing multiple TCP connections per iSCSI session, in accordance with the Standard. When this is done, there will be two kernel threads per connection. This will permit a great deal of simultaneous I/O activity within a single iSCSI session, since different connections can go over different networks between the same Initiator and Target.

4 CONFIGURATION

Although the SEP protocol is a simple encapsulation protocol, iSCSI is not – it involves a rather complex login procedure which requires negotiation of security and operational parameters between Initiator and Target. These negotiations permit a dynamic choice between several security schemes, as well as a choice of optional digests to protect iSCSI headers and/or data. Furthermore, a single session can involve multiple TCP connections between the same Initiator and Target, with implications for flow control and organizational complexity on both sides. iSCSI sessions are expected to last for a very long time, in many cases for the entire time the kernel is running. It is therefore not possible to freeze the configuration of these parameters either at the time the software is compiled or when the modules are loaded.

It is clearly important to be able to dynamically manage this software in a convenient manner. To do this we have designed it from the beginning to utilize the Linux "/proc" interface as the means of communication of management information between the kernel components and the system administrator. The /proc interface is a natural means for displaying information about the internal state of a kernel component, but it has also become an increasingly important tool in Linux that allows a system administrator to have direct input into a running kernel.

This design conforms to both our guidelines: the /proc interface is very general, and can be used by any HBA or FETD; and the mechanism is already part of Linux, so we do not have to invent another way to do this.

There is a separate interface for Initiators and for Targets. On the Initiator side, when the iSCSI Initiator is loaded it creates a "/proc/scsi/iscsi_initiator" directory that contains one file for each of the sessions that this initiator is capable of handling. (Loading the SEP Initiator creates a similar directory called "/proc/scsi/sep_initiator".) A user-space tool, modeled on the "ifconfig" software for network configuration, utilizes these files to bring the indicated session "up" or "down", and to configure it with the target IP address, the target TCP port number, and any other desired information, such as the login negotiation parameters (see below).

On the Target side, there is a file called "/proc/scsi_target/scsi_target" which lists the targets being emulated on that platform. When the iSCSI Target module is loaded it creates a directory called "/proc/scsi_target/iscsi_target". This directory contains one file for each of the sessions that this target is capable of handling, plus a file called "iscsi_config" that contains the parameters to be used when accepting a new session from an Initiator. During the login process with that Initiator these parameters are subject to negotiation, so the individual file for that session contains the applicable values for the session. Writing to the "iscsi_config" file will change the values used when creating the next session; writing to the individual file for a particular session will change the values currently in use by that existing session.

In iSCSI, a big part of the input to both the Initiator HBA and the Target FETD is the setting of parameter values needed for negotiation during the login process. These settings are incorporated into the iSCSI protocol as UTF-8 text having the form: "key=value" where "value" can be either a single numeric value, a single UTF-8 character string, or a list of UTF-8 character strings. Our /proc interface permits input of text having these same forms, but with added information that indicates how and when these keys are to be utilized for both negotiation and operation. Additional configuration information includes management parameters outside the context of the iSCSI protocol, such as the IP addresses and TCP port numbers to use for both Initiators and Targets, etc.

5 CONCLUSION

In this paper we have described the architecture of our software for developing and testing storage area network transport protocols. This software is designed to run in the Linux 2.4 kernel and to take advantage of the facilities offered by the Linux kernel – in particular, the existing TCP/IP network stack, the existing SCSI subsystem, the /proc interface and the loadable module facility.

Our software for the Linux kernel, which is freely available in source form, is beginning to be used for a number of purposes: as the basis for general prototyping for network transport protocols, as a means of "proofing" the iSCSI protocol as it undergoes the IETF standardization process, as a basis for developing tests that check conformance to that standard, and as a basis for developing interoperability tests for Initiators and Targets that utilize iSCSI. While the software contains many "hooks" to facilitate its use in the testing environment, it also defines a "reference implementation" of iSCSI that can be used by small installations where

simply having access to iSCSI without the need for any special hardware is more important than the high performance that is expected to obtained when hardware specifically designed to utilize iSCSI becomes available on the market.

6 ACKNOWLEDGEMENTS

This work was supported in part by SUN Corporation and EMC Corporation.

REFERENCES

[1] Ravi K. Khattar, Mark S. Murphy, Giulio J. Tarella, and Kyell E. Nystrom. *Introduction to Storage Area Network, SAN*. http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245470.pdf, August 1999.

[2] ANSI X2.269-1995. *Information Technology, 'dpANS Fibre Channel Protocol for SCSI (SCSI-FCP)'*, 1995.

[3] Andrew Wilson. *Internet Draft, 'The SCSI Encapsulation Protocol' (SEP)*. http://www.ietf.org/internet-drafts/draft-wilson-sep-00.txt, May 2000.

[4] Julian Satran and et al. *Internet Draft, 'iSCSI (Internet SCSI)'*. http://www.haifa.il.ibm.com/satran/ips/draft-ietf-ips-iSCSI-06.txt, January 2001.

[5] IETF. *The Internet Engineering Task Force: Home Page*. http://www.ietf.org/.

[6] Ashish A. Palekar and Robert D. Russell. *Technical Report 01-01: 'Design and Implementation of a SCSI Target for Storage Area Networks'*. Computer Science Department, University of New Hampshire, Durham, NH 03824, May 2001.

[7] dpANS T10/Project 1157-D/Rev 18. *Information Technology, 'The SCSI Architecture Model-2 (SAM-2)'*, May 2001.

[8] Douglas Gilbert. *The Linux SCSI subsystem in 2.4*. http://www.torque.net/scsi/linux_scsi_24/index.html, 2001.

[9] Douglas Gilbert. *The Linux SCSI Generic Interface*. http://www.torque.net/sg/p/scsi-generic_v3.txt, 2001.

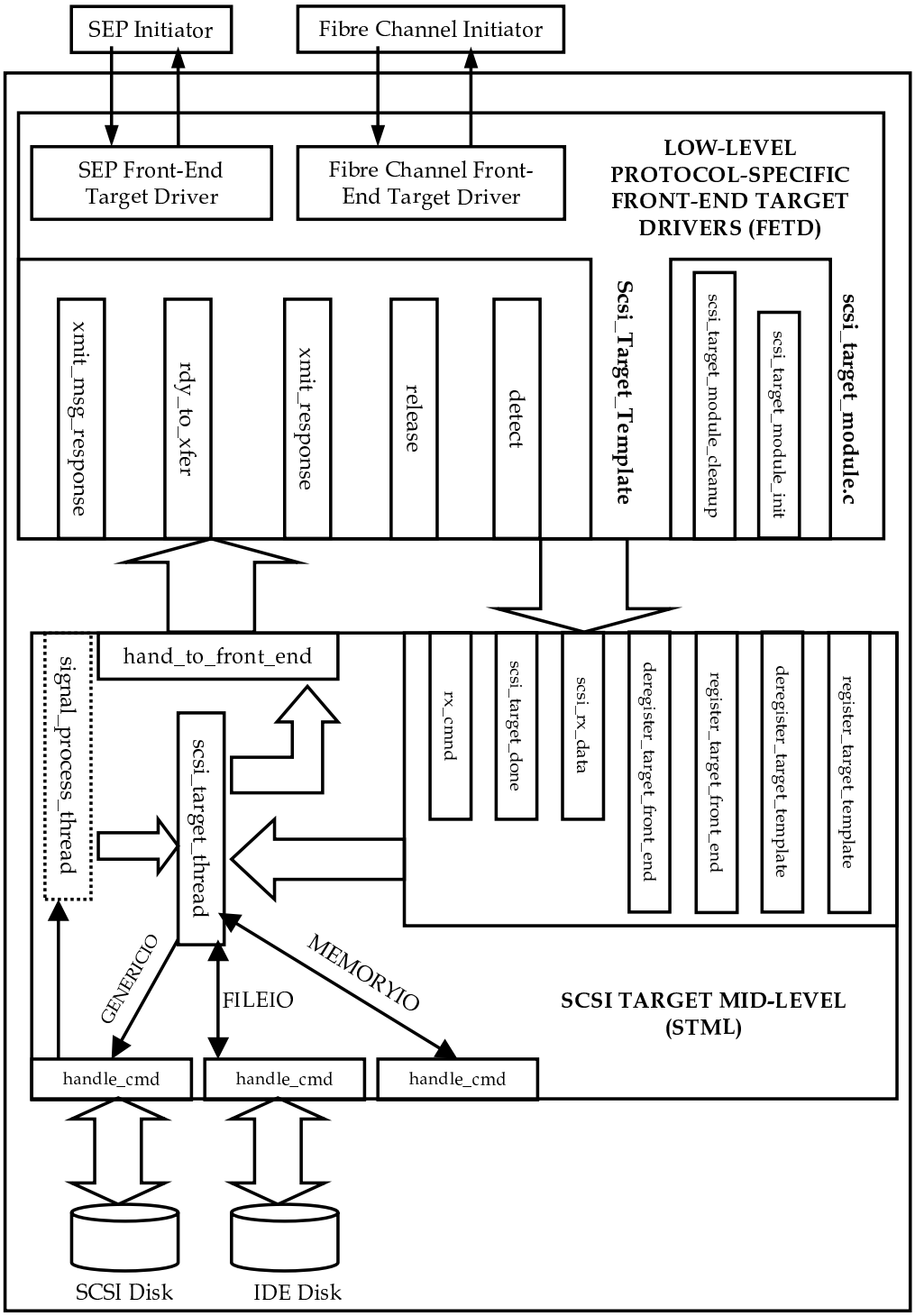

FIGURE 6: Details of the Linux Target Emulator Organization