Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme

Lehrstuhl Systemarchitektur

Studienarbeit

Advanced SCSI Programming Interface over Internet Protocol

Johannes Lieder

Betreuer: Prof. Dr. Frank Bellosa

Stand: 27. Februar 2006

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Hiermit erkläre ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

Karlsruhe, den 27. Februar 2006 Johannes Lieder

Abstract

The development of technologies facilitating spatial separation of storage subsystems and their responsible computational resources is in increasing manner subject to recent commercial and scientific research. While most existing solutions relate to the SCSI (Small Computers System Interface) standard, resulting in a high degree of universality, from an end-user's point of view there is, however, a lack of dedicated support for remote *optical storage*. Although CD/DVD recorders typically behave as standard SCSI multimedia device, specific requirements for this class of devices (e.g., timely arrival of command blocks) have to be taken into account, especially when considering scenarios of distributed deployment; that is, host computer and target device reside in separate locations connected by a common network interconnect. The objective of this work is to remedy missing support by the development of a viable solution under the main premise of an inexpensive application among end-users.

This thesis describes an approach, which allows the transmission of ASPI (Advanced SCSI Programming Interface) requests across an IP-based network. To accomplish transparent usage of remote devices, these request blocks are *tunnelled* over a transport layer connection, which means being serialized and enveloped by an appropriate packet header and trailer. In addition to this functionality, the designed protocol includes supplementary mechanisms in respect of extensibility and future enhancements, turning it into a generic framework for the given task at the same time. The solution's accompanying implementation demonstrates feasibility (proof-of-concept) in general and correct operation of the elaborated protocol design in both *single-threaded* and *multi-threaded* application.

Contents

Chapter 1

Introduction

SCSI (Small Computers System Interface) comprises a common protocol for personal computers, workstations and servers defining communications between hosts and peripherals from a high-level command set down to the physical layer where bits typically travel along a copper wire. Most current operating systems use SCSI internally to abstract from many of the various devices' peculiarities (often accomplished by SCSI miniport drivers). Thus, using a SCSI command set, virtually all types of devices like hard disk drives, scanners, or jukeboxes can be accessed. This even applies to most ATA devices today, due to the fact that the ATA command set has been extended to support *SCSI like* command blocks (ATA Packet Interface [17]). To overcome the distance between computing devices locally (i.e., within or aside the computer's case) often wired bus topologies are applied. Unlike SCSI being constrained to relatively short distances, network topologies (e.g., Ethernet) are much less limited in end-to-end diameter (meters in contrast to centimeters). The broad deployment of Ethernet today makes it an ideal low-cost replacement for local buses, but since utilizing completely different layered architectures, a kind of tunnelling has to be performed where the local high-level SCSI command blocks are encapsulated and passed to the now underlying network stack.

Due to the recent need for separation of computing devices and storage subsystems in large data centers, a variety of standards have been developed facilitating the paradigm of so called SANs (Storage Area Networks). Only hardware solutions do not suffer from performance issues in these high-demand environments, rendering this class of hardware impractical and unaffordable for average PC users. In contrast to SANs – providing block-oriented storage (the smallest accessible unit is a block of data residing on a logical storage device) – NAS (Network Attached Storage) is gaining in popularity among users lately. Thus, more and more of these network-aware hard-disks can also be found among private networks. In case of NAS, storage space is made available in form of network shares (via high-level network protocols like FTP [10] or SMB), where files are the smallest unit of allocation.

The assignment of this thesis is the development of a solution which addresses normal PC users by combining the possibility to access devices remotely and the advantages of utilizing already existing network infrastructure and introducing a simple and inexpensive software layer at virtually no additional cost. Further, design paradigms like reliability, scalability, integrity and security should be realized, albeit likely only up to a level which complies to the scope of a study thesis. However, a working solution, potentially attractive for average PC users, should be implemented at least. Being constrained by latter requirements and the demand for a low-cost software solution in particular, this also constrains the range of suitable standards seeming applicable to the architecture. Advantageous for this task is the fact that a large amount of the stipulated properties is already provided by the most common network transmission protocol (TCP) today and the underlying network stack, respectively. Due to the broad deployment of TCP/IP-based networks and assuming this kind of implementation, a certain platform independence can be guaranteed by building an application layer (ISO/OSI layer 5–7, see [1]) upon this transmission protocol.

Relevance

Today, virtually every newly purchased PC ships with at least one optical storage device. During the advent of optical media even the plain CD-ROM technology (i.e., a medium only capable of being read) was relatively expensive. However, with the introduction of the next generation of devices, for example CD-R or CD-RW, the already established devices became continuously cheaper over time, which may be attributed to the need for and consequently the broad spreading of these devices. A very similar process could be detected with virtually every subsequent drive generation.

Currently, DVD+/-RW (multi-format) recorders can be found in many computers with the new HD-DVD/Blu-ray Disc standards pending to be released. So, at a first glance, it might seem superfluous to share these devices over a network since all computers are fully equipped, but at least two scenarios beyond just making optical storage devices accessible remotely are imaginable. First, for reasons of admission control and second, to lower hardware costs. Especially, in case of the new BD (s.a.) costs are disproportionately high – and even higher, if a larger number of users should have access to this storage technology. Sharing those devices within a corporate network or a pool minimizes asset cost while still having access to state-of-the-art hardware.

Issues/Problems

In a host computer peripheral scenario an issued SCSI command sent to a local device usually never quits the system. So data being written or read only has to be placed in memory prior to the call or has to be copied from memory afterwards. To overcome spatial separation of host computer and peripheral the shared memory between these endpoints has to be kept up-to-date, resulting in a form of *distributed memory coherency*. This coherency might be sustained by a simple communication protocol consisting of the following three steps:

1. Transmit client memory block to server (C-to-S update)
2. Execute SCSI command on the server
3. Send the server's modified memory block back to the client (S-to-C)

Here we still use the standard network notions of client/server, while in the following sections of this work the more adequate notions of initiator and target will be introduced.

The depicted protocol can be used for the entire SCSI command set in general. However, in case of commands being accompanied by large data blocks some optimizations may be applied. For example, after issuing a `WRITE` command (`W`-type command, see [5], p. 16) the memory block holding the data to be written obviously remains unaltered. In other words, a `W`-type command to the medium in turn is a **read-only** operation in memory. Thus, retransmission of the unmodified content (previously copied from the client's address space) is superfluous. The omission of the S-to-C transfer ought to have a significant positive impact on network message delay and finally application-level latency (see chapter 4). The same applies to `READ`s (`R`-type command) in an analogous manner: the C-to-S update of the target buffer to the server is not necessary as the transferred data will always be overwritten completely unappreciated.

Further, as the issued SCSI command blocks contain pointers to local memory locations any implementation has to cope with the task of managing these pointers transparently on the client's computer to ensure an integrative operation of the client process. For the server side, every incoming command is "new" and so the serving process has to manage memory allocations in addition to the the client side's memory copy operations (figure 1.1 shows a block diagram of the basic architecture).

At least, the described functionality has to be implemented by the solution developed throughout this thesis. Further, to ensure flexibility and extensibility and to overcome the problem of complexity the widespread design paradigm of layered software development will be applied.

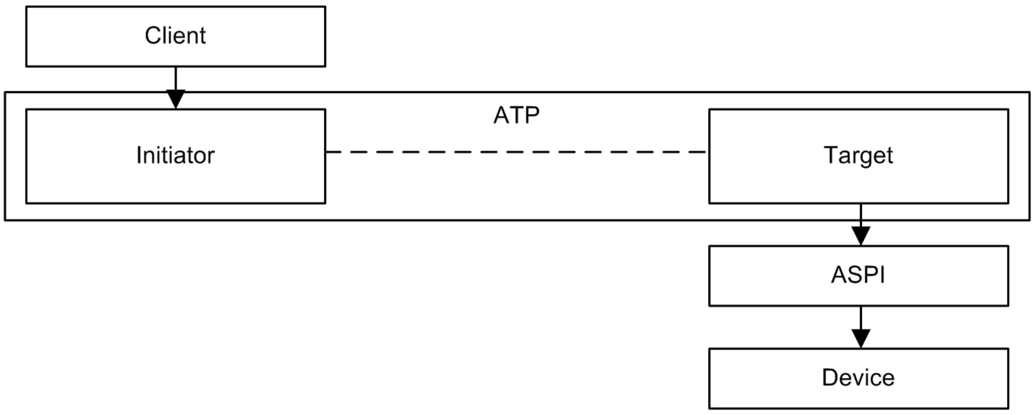


Figure 1.1: Block Diagram

Simple network communication on the data link layer (e.g., Ethernet) is unreliable and provides virtually no guarantees in terms of data integrity, ordering, or arrival. Preferably, the implementation should be lean and so will utilize as many services as possible already provided by the host's OS in form of the TCP/IP stack. Concordantly, requirements like data ordering and integrity are introduced within these lower network layers (ISO/OSI layers 3 and 4). The application-level functionality of transmitting SCSI command blocks and request blocks respectively will be handled by the remaining session, tunnelling and application layers (equivalents to ISO/OSI layer 5 through 7).

Purely software-based implementations often struggle with an increased latency caused by a complex stack of software layers. Similar issues have already been observed with iSCSI solutions [18]. Specialized and expensive hardware has been developed by a small number of companies (Adaptec et al.) to overcome some of the depicted drawbacks. For example, by offloading some of the CPU-intensive tasks to dedicated processing units the load imposed by iSCSI packet generation can be alleviated (to a certain degree). These considerations do apply to an ASPI over IP implementation as well. However, due to the special deployment scenario for this solution – being preliminarily defined (see above, p. 6) – the consequences may be neglected for the course of this work. Again, additional costs for the end-user should be avoided whenever possible. Considering that even with low-cost hardware computational power (i.e., raw CPU cycles) is rather cheap, the still layered ASPI over IP implementation should be of no concern.

Another important reason why iSCSI cannot be applied to the existing scenario is the fact that – although the iSCSI standard does not restrict the type of SCSI device being used – current implementations (e.g., Microsoft's iSCSI initiator) do not support devices other than storage class devices (which is most likely an restriction of the initiator kernel driver emulating a local device of hard-coded type). In case of CD recording the task is heavily dependent on latency of the interconnect medium replacing the typically dedicated internal SCSI bus of the local computer. With Fast Ethernet and Gigabit Ethernet available today, transfer bandwidth appearing to be critical at data rates of several megabytes per second of today's recording devices, however, is of secondary concern. Pure sequences of `READ`s (reading data blocks from the medium) commands are not critical in contrast to sequences of `WRITE` (writing data blocks to the medium) commands especially in case of CD recorders. In the following sections the causes and resulting implications will be examined in more detail.

During the course of this study thesis a solution for the given problem will be presented. Many design decisions have been made with the limited resources and a preferably simple implementation in mind. Finally, the main objectives will be achieved, but obviously with limitations imposed by the respective hardware environment (CPU speed, network bandwidth and latency). However, an outlook to further promising tweaks which may be applied to circumvent unavoidable transmission latencies can be given at the end of this thesis.

Chapter 2

Background and Related Work

2.1 Background

The following chapter is confined to background issues and other work related to this study thesis. Therefore, the following sections will delve into various aspects of SCSI and ASPI [2] standards – later on also other related technologies like SAS (Serial Attached SCSI), FC (Fibre Channel) and its counterpart iSCSI. However, only the most important aspects relevant for this work will be discussed rather superficial.

2.1.1 SCSI Fundamentals

For the first overview a basic knowledge of some of the important SCSI fundamentals is necessary. The entire SCSI standard comprises multiple documents representing different layers and also different levels of standardization. The main starting point is the SCSI Architectural Model (SAM-2) [13] which describes the overall architecture of a SCSI subsystem. In an abstract view SCSI can be regarded as common client/server model, here denoted by the term SCSI distributed service model ([13], fig. 5). This scenario is very similar to the ubiquitous client/server model in network communications. The client (SCSI Initiator Device) requests a service from the server (SCSI Target Device), which sends back a corresponding response. Furthermore, SCSI is based on the layered model (fig. 2.1) for distributed communications which abstracts from a certain *Interconnect* or *Service Delivery Subsystem* ([13], fig. 25). Thus, this interconnect can be of any given type rendering SCSI a very flexible system, which may be easily observed in case of Fibre Channel – a formidable transport for SCSI commands. The SCSI model for distributed communications consists of two layers: the SCSI Application Layer (SAL) and the SCSI Transport Protocol Layer (STPL), stacked on top of an Interconnect Layer or Service Delivery Subsystem and

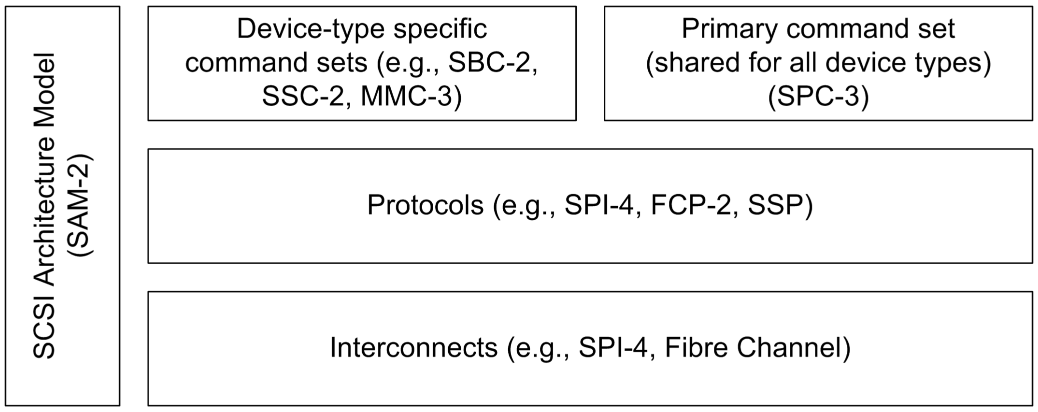

SCSI Architecture Model (SAM-2)
Device-type specific command sets (e.g., SBC-2, SSC-2, MMC-3)
Primary command set (shared for all device types) (SPC-3)
Protocols (e.g., SPI-4, FCP-2, SSP)
Interconnects (e.g., SPI-4, Fibre Channel)

Figure 2.1: SCSI Architecture: Layered Model

finally an appropriate interconnect. As usual two kinds of communication, vertical between adjacent layers and horizontal communication between the two endpoints take place in this layered protocol architecture. Being responsible for I/O operations invoked by the client, the SAL guarantees a certain level of standardization at application level (Command Standards). Now, the STPL provides services and protocols for clients and servers to communicate (SCSI Transport Protocol Standard). With the Interconnect Layer, a signaling system and a service for the physical transfer of data between sender and receiver (an Interconnect Standard) is introduced. To accomplish I/O operations SCSI defines an Execute Command procedure call (or mechanism) in its architectural model which follows the previously mentioned request/response protocol. However, this procedure call has to be dispatched with a number of accompanying parameters (IN). The return data represents the associated response. Assuming the Request-Response transaction model ([13], 4.15, fig. 27) an Execute Command call to the application layer (SAL) implicates an amount of protocol communication in the lower layers. For example, [13] 5.4 the STPL splits the call up into calls to Send SCSI Command, SCSI Command Received, SCSI Transport Protocol Service Response, and Command Complete Received. In the SAM/SAL the procedure call is noted as follows:

```
Service Response = Execute Command(
        IN(I_T_L_Q Nexus, CDB, Task Attribute, [Data-In Buffer Size],
                [Data-Out Buffer], [Data-Out Buffer Size],
                [Command Reference Number], [Task Priority]),
        OUT([Data-In Buffer], [Sense Data], [Sense Data Length], Status)
)
```

The parameters of the Execute Command procedure call will be subject to the following paragraphs, as they heavily influence the interface to a client application utilizing SCSI. These parameters also constitute a great part of the SCSI standard and since the three SCSI layers are relatively thin, they have to be handled and transported in case of the given program design. Depending on the SCSI command to be invoked some of the input and output parameters are optional (denoted by square brackets). So three parameters become mandatory, namely the `I_T_L_Q` Nexus, the Command Descriptor Block (CDB), and a Task Attribute. The SCSI notion of a nexus describes a logical relationship of different levels, for example between an initiator port and a target port (`I_T` Nexus), or more specific between initiator, target, and a target's logical unit (Logical Unit Number, LUN), which would be noted as `I_T_L` Nexus. Furthermore, a defined task (representing a pending command) can be addressed by a so called `I_T_L_Q` Nexus (`Q` stands for Task Tag), actually a numeric value – like in the case of the Execute Command procedure call. Essential for the Execute Command procedure call is the Command Descriptor Block as it defines a unified memory structure to govern SCSI command execution. The CDB's generic layout is also defined by the SAM, viz. the location and exact composition of the `OPERATION CODE` (in the first byte). To proceed it is necessary to study the SCSI Primary Commands (SPC-3) documentation, which describes the CDB format for all common SCSI commands (i.e., SCSI commands any type of SCSI device has to support) with more minutiae [16] 4.3. The CDB usually is of fixed size and contains all additional information required for the execution of a SCSI command except for the optional parameters, like buffer locations and sizes, for example the `LOGICAL BLOCK ADDRESS`, `TRANSFER LENGTH`, and `CONTROL` (allowing optional features in conjunction with the SCSI command being about to be executed). Finally, Task Attribute as the last mandatory parameter to Execute Command enables features like *linked commands* (multiple I/O operations are treated as a single task) and *command reordering*, rendering SCSI an even more powerful peripheral subsystem, in contrast to low-cost subsystems like ATA missing this range and richness of functionality.

In terms of error detection the client application can evaluate Sense Data automatically collected by SCSI if the Status code equals to `CHECK CONDITION`. In case of an error condition, the Sense Data structure provides detailed information . As the objective of this work is control of WORM (Write Once, Read Many) devices, two new SCSI commands `READ` and `WRITE` have to be mentioned not yet defined by SPC-3. Since these commands are not common to all device types they are introduced in the SCSI Multi-Media Commands (MMC-5) [15] document. There are different subtypes of `READ` and `WRITE` commands as the SCSI standard distinguishes between command classes of different sized CDBs accompanying the `READ/WRITE` command. Typical commands for *direct-access block devices* are `READ 6`, `READ`

10, and READ 12 introducing a level of flexibility when addressing large disks by LOGICAL BLOCK ADDRESS. This description of SCSI fundamentals should cover all main aspects of the standard needed to understand the considerations of the subsequent chapters.

2.1.2 ASPI fundamentals

When studying the ASPI (Advanced SCSI Programming Interface) it should be regarded primarily as application-level programming interface. Therefore, the API helps to simplify the programmer's view and access to local SCSI devices.

From the system's perspective multiple SCSI HBAs (Host Bus Adapters) representing different initiators may exist. Although being uniformly accessible by the host's OS through drivers abstracting from the adapter's peculiarities, a mechanism has to be introduced that allows to discover and address devices as well as the responsible host adapter. Combined with the function set needed to expose all main SCSI commands, the result constitutes virtually all the functionality covered by ASPI. With the adapter count already being returned upon initialization (GetASPI32SupportInfo()) an inquiry can be sent to HBAs allowing the detection of all existing devices; that is, the determination of all valid HA:ID tuples. The INQUIRY command is part of the SCSI standard and may be dispatched using ASPI's SendASPI32Command() function call accompanied by a properly initialized SRB structure.

```
typedef struct
{
    BYTE SRB_Cmd;              // ASPI command code
    BYTE SRB_Status;           // ASPI command status byte
    BYTE SRB_HaId;             // ASPI host adapter number
    BYTE SRB_Flags;            // ASPI request flags
    DWORD SRB_Hdr_Rsvd;        // Reserved, MUST = 0
}
SRB_Header;

DWORD GetASPI32SupportInfo(VOID);
DWORD SendASPI32Command(LPSRB);
```

Figure 2.2: ASPI Programming Interface

There are many similarities between the SCSI programming interface (see CDBs) and ASPI with a single main procedure call, where SendASPI32-Command() in conjunction with a parameter structure (SRB, SCSI Request

Block) is analogous to Execute Command and its CDB in case of SCSI. The SRB's first field holds the ASPI command, for example `SC_HA_INQUIRY` (to retrieve adapter-related information) or `SC_EXEC_SCSI_CMD` (to dispatch a SCSI command).

In summary, ASPI unifies all different aspects of accessing SCSI devices in a straight-forward way (see CDB in comparison to SRB). With ASPI simplifying the discovery of SCSI hardware, introducing management for SRBs, and providing a programming interface abstracting the view to multiple host bus adapters (different hardware, various drivers) it encapsulates the SCSI standard and extends it in terms of application-level programmability. However, since the introduction of SPTI (SCSI Pass-Through Interface) with the Windows operating system the importance of Adaptec's ASPI software layer (originally developed and exclusively shipped with the firm's AHA series of host bus adapters) is continuously diminishing.

As illustrated in figure 2.3 with a compartment model, different levels of processing are performed with the passed command structures. Leaving a compartment implies the processing of the responsible entity which (in case of ASPI) is the Windows ASPI layer, the SCSI Miniport/bus driver (SCSI), or the device driver (O/S), eventually. However, the `EXEC_SCSI_CMD` command represents an exception to the remaining ASPI command set (mostly providing an interface to management and discovery functionality) as the accompanying CDB is directly passed to the appropriate SCSI stack and finally the device (see fig. 2.3).

2.1.3 Sockets API

Another broadly used and thoroughly tested API is the classical BSD-style network programming interface, building upon its original *file handle-centric* design paradigm. With Microsoft's WinSock API as its superset these programming interfaces provide a well abstracted and easy-to-use API to the TCP/IP stack and finally to the whole variety of existing networking hardware. Sockets are the logical abstraction of network communication endpoints represented by file handles. Once being established, the programmer can operate on these network connections (between instances of the Transport Layer, ISO/OSI layer 4), typically by use of `send()` and `receive()` primitives. While adhering to a proper client/server design, TCP connections are not built symmetrically, say the client uses another mechanism (`connect()`) than the server which passively accepts (`accept()`) a connection.

Although WinSock supports advanced programming techniques, for example **Overlapped I/O** to overcome limitations imposed by blocking system calls (causing the affected process/thread to enter a waiting state), these features will not be used with the proposed implementation. Instead, parallel execution of multiple worker threads introduces the needed level of asynchronous processing. Therefore, from a programmer's point of view relying on the blocking network API is advantageous, yet even mandatory while a blocked state confines to the scope of a single worker thread resulting in a nearly synchronous implementation design within this domain.

2.2 Related Work

2.2.1 SCSI

During the last couple of years storage access over network topologies has been topic of thorough research and commercial development resulting in a diversity of different technologies utilizing various types of interconnect, specialized hardware, or layered software implementations. As the objective of this thesis is the development of another similar solution, yet focussed on real end-user usability and particularly optimized for transparent access to CD recording devices residing at remote locations (issues that have often been neglected in the past).

Evolving from Fast SCSI (SCSI-2) with a 8-bit wide bus offering transfer speeds up to 10 MiB/s to Ultra 320 SCSI delivering aggregated data rates of up to 320 MiB/s (with a widened 32-bit bus), SCSI has traditionally been a storage subsystem processing transfers over wired, parallel cables. Although transformed into a sequence of serial transfers in the lower SCSI layers (STPL and SDS), from the application's high-level point of view this is done transparently giving an impression of parallelism. In fact, when submitting an Execute Command procedure call first the complete set of parameters and data structures have to be in place. Anyway, whether representing a substitute for a SCSI interconnect, an adaptation technology for the complete SCSI standard, or an application-level transport for SCSI/ASPI command traffic, the protocol *has* to be serialized and de-serialized allowing it to travel along an arbitrary interconnect technology.

2.2.2 Fibre Channel (FC)

The first important standard to mention is Fibre Channel (first draft published 1994), or better the Fibre Channel Protocol for SCSI (FCP) and the Fibre Channel mapping layer for SCSI (FC-4), adapting the versatile SCSI standard to another underlying interconnect. FC's most prominent feature is the optical infrastructure defined by the corresponding Physical and Signaling Interface (FC-PH). However, while being another layered communication technology, Fibre Channel is not limited to the fiber optical domain. For example, there are also wired Fibre Channel devices (hard-disks with a 40-pin SCA-2 connector) sometimes found in workstations where performance is of highest priority.

Through its optical delivery subsystem (including *Classes of Service* functionality) and a single protocol layer mapping SCSI communications directly to the physical frame format, Fibre Channel represents an ideal storage subsystem interconnect for high-performance environments. The underlying physical layer again can be divided into 3 sublayers FC0 – FC2, which, however, might be considered as single layer due to their compactness. The behavior is representative when analyzing similar protocols, therefore a short example should be given. After dispatching `FCP_CMND` to initiate an Execute Command on the target the command is marked by a unique identifier (Fully Qualified Exchange Identifier, `FQXID`). Most protocols are designed to only transmit solicited data (from initiator to target or vice versa depending on the type of command; e.g., `READ` or `WRITE`). Hence, a `FCP_XFER_RDY` command has to be encountered by the initiator first. The other party replies with an according `FCP_DATA` packet followed by a finalizing `FCP_RSP` containing SCSI status and SCSI Request Sense information if applicable (see Request-Response Protocol, chapter 2.1.1, SCSI Fundamentals).

2.2.3 iSCSI

Fibre Channel is a mature technology with many additional features needed for high-demand scenarios. However, due to the costly optical infrastructure and specialized hardware this technology cannot be applied in case of Personal Computers, although representing an important example in terms of protocol communications and SCSI Application Layer (SAL) adaptation.

As counterpart technology and since Fibre Channel has always been relatively expensive (s.a.), IETF [11] has pushed development for the iSCSI protocol standard (finally posted for RFC in 2003). Originally aiming at software implementation and thus building upon TCP as transport protocol, the standard overcomes some limitations (iSCSI is a high-level application layer protocol capable of being routed through the Internet, hence its name "*Internet SCSI*"). Recently, iSCSI is gaining importance although still suffering from various design-inherent drawbacks also inhibiting its popularity. However, when being deployed in an environment with an inexpensive (Gigabit) Ethernet infrastructure, iSCSI is an ideal way to attach a separate SAN (see chapter 1). Because of significant overhead caused by a complex software stack and application layer in particular (see [18]), specialized hardware solutions have been developed over time (see chapter 1), too.

2.2.4 Serial Attached SCSI (SAS)

With the advent of next generation hard-drives allowing high transfer speeds and burst rates (mainly caused by large disk caches) new internal storage attachment standards are about to supersede SCSI's aged parallel cables. Serial point-to-point connections do not suffer from limitations becoming dominant in case of parallel technologies with increasing transfer speeds and decreasing timings (e.g., signal echos impairing signal fidelity). Thus, an initial U640 (Ultra SCSI 640) standard could not prevail in favor of a new serial interconnect technology, as can be observed in case of various other technologies (Serial ATA, USB, IEEE 1394). This Serial Attached SCSI (SAS) standard borrows its lower layers from SATA, while FC - or more accurate FCP - lends its approved SCSI adaptation layer (FCP) to the new standard. In comparison to FCP, the SAS standard utilizes only required fields of the command packet, defining superfluous fields reserved. Hence, Serial Attached SCSI might be regarded as a Fibre Channel implementation where the FCP is tunneled over SATA cables. SAS unites the many advantages of dedicated point-to-point connections (allowing full-speed transfers to each device), SCSI's versatility, and backwards-compatibility to SATA infrastructure, respectively. Actually, SAS hard-disks may be installed among SATA disks as they ship with the same connectors and SAS defines a SAT (SAS ATA Tunneling Protocol). This, however, does *not* allow the reversed scenario of deployment.

2.2.5 HyperSCSI/SCSI RDMA

Further, there are several less important protocols which provide the same services either using different underlying network technologies or trying to overcome some of the already discussed drawbacks by directly building upon low-level network protocols like Ethernet or Internet Protocol (IP).

An example of the latter class is HyperSCSI [7], which is designed to utilize both options for the transmission of SCSI protocol traffic (today, only an Ethernet implementation in software exists). Any Ethernet-based protocol communication is limited to the local network domain because of the inability to be routed. This, however, might be a wanted behavior when data has to be kept confidential and may not leave the local storage network.

A last instance of storage communication protocols is the SCSI RDMA Protocol (SRP), where RDMA is an abbreviation for Remote Direct Memory Access, utilizing an Infiniband network infrastructure. All previously discussed topics also apply to SRP apart from the required RDMA communication service model. This service model introduces RDMA Channels allowing pairs of consumers to communicate via messages and data transfers. The data transfers occur with direct access to the remote computers address space. The combination of message delivery and direct data transfer allows the application of a RDMA service as another delivery subsystem for SCSI commands (the exact realization is subject to the accompanying SRP protocol proposal [14]).

2.2.6 NeroNET

As the primary objective of this thesis is the development of a communication protocol capable of transporting application-level request blocks (SRBs) as well as maintenance of data coherency by payloading data buffers, a comprehensive overview of related protocols and work has been given in the previous sections. There is, however, another product sold for deployment in conjunction with the Nero Burning ROM application suite that takes a completely different approach to provide a service allowing the centralized recording of CD media. Once the extensions for NeroNET are enabled, a new virtual recording device is available for usage from within Nero and the entire Nero API, respectively. Although for the end-user the experience is very similar, NeroNET does not continuously send commands over the network. Instead, the complete CD image is compiled on the client and then transmitted to the server where a lightweight Nero process finally writes the data to the disc. An advantage of this concept is the possibility to implement multi-user operation (e.g., a queuing mechanism). However, real multiplexing is impossible due to the exclusiveness of this type of resource (CD recorder) prior to operation such a device is reserved for a single party, otherwise a SCSI `RESERVATION CONFLICT` condition occurs. On the other hand this approach has the disadvantage of an increased burning process duration. The overall time at least sums up to the transmission time of the CD image plus recording-time, in contrast to the interleaved transmission of data and immediate execution of the accompanying `WRITE` command as is the case with ASPI over IP.

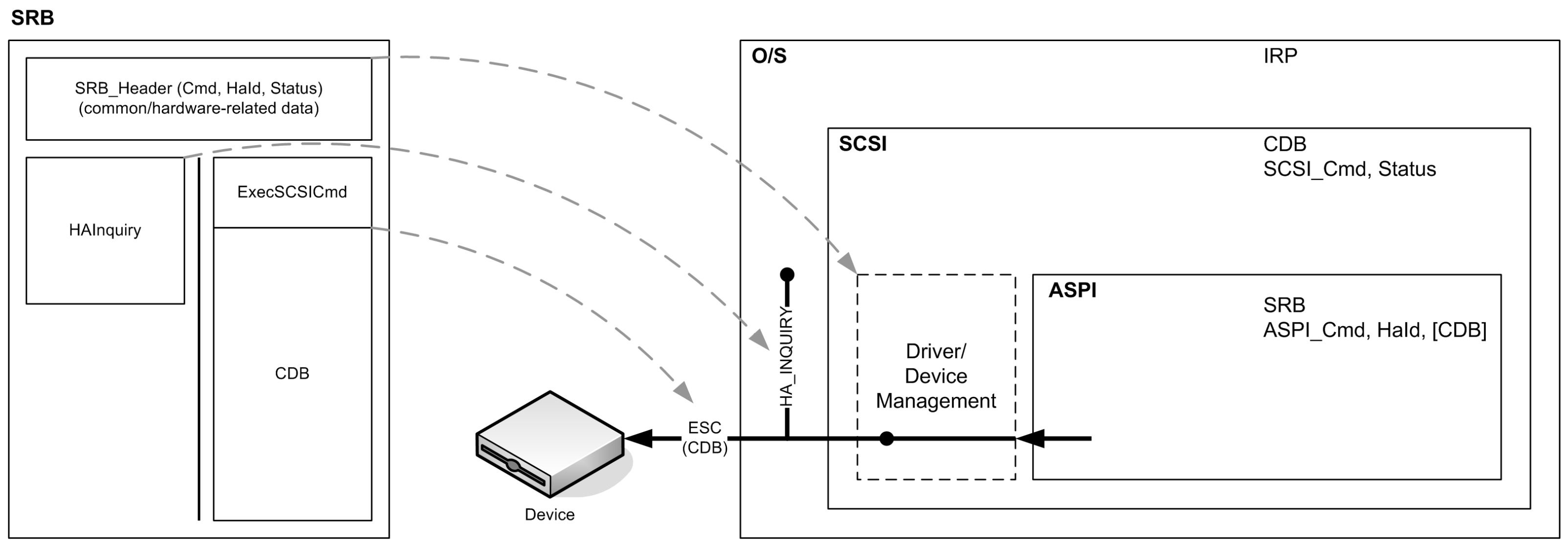


Figure 2.3: Compartment Model (Processing of EXEC_SCSI_CMD and HA_INQUIRY)

Chapter 3

Design Considerations

3.1 Layered Solution

The following chapter covers the gradual design and implementation of a solution to the given problem of this thesis. As discussed in chapter 1 the primary objective to realize a form of ASPI tunnelling over IP. In other words, this is the development of a message and data passing protocol sustaining memory coherency between initiator and target of SCSI transactions. The protocol has to be of bidirectional design to support the underlying Request-Response model and should define different types of messages which initiate a well-defined behavior on the peer machine.

For the remaining part of this chapter a bottom-up design process will be applied. The standard Internet protocol design procedure suggests a layered solution design to overcome complexity, which leads to the ISO/OSI reference model. As discussed earlier in chapter 1 the proposed solution will build on the TCP/IP stack, whereas the lower layers up to the transport layer (layers 1 through 4) are well defined; session, presentation, and application layer make up the residuary part. With the restriction of architecturally similar end-systems (little-endian vs. big-endian) a fully-fledged presentation layer, which might be responsible for an independent data representation can be omitted. In return, a supplementary tunnel layer should be placed upon the session layer for the reasons mentioned above. This again results in a thin application layer (see fig. 3.1).

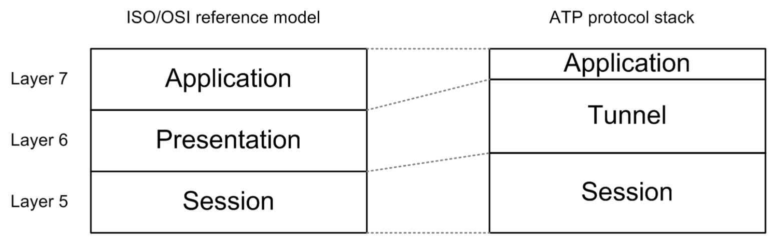


Figure 3.1: Protocol Stack

3.2 Protocol Design

State Machine

Many Internet protocols are plain-text protocols. A fact which pays off in debug situations and whenever resource-constrained systems might be involved. Another example for this mind is IETF's iSCSI protocol (RFC 3270, [11]) – a protocol primarily designed for unstructured data transport – however, it defines an initial text-based negotiation phase (chapter 5, p. 50). As iSCSI represents the principal model for the ASPI Tunnelling Protocol (ATP) this methodology will be adopted, providing space for future extensibility. For the well-defined communication between initiator and target (client and server) a communication protocol has to be defined. It represents a mandatory *operations contract* for the two parties and stipulates the behavior when transiting between different protocol phases. For appropriate visualization a connection-oriented transport state machine pattern ([1], p. 64, fig. 5–4) provides the foundation. This representation of a finite protocol automaton uses a special notion for service primitives and their decomposition. With the initial phase (text-based negotiation part) of the ATP protocol being an element of the session layer the state machine processes session-layer primitives (prefix S) as input and returns session- or transport-layer (prefix T) primitives as output (<input>; <output>). Concatenated with a service and service primitive part an unambiguous notation is given (for example TConReq). To maintain diagram readability and to limit the amount of intermediate protocol states the common request-response sequence (Req $\rightarrow$ Ind, <Intermediate State>, Res $\rightarrow$ Cnf) [fig. 3.2] is abbreviated by the simple notation of the respective layer-service identifier (for example TCon).

Starting from the *Not Connected* state the initiator may create a connection to the target (NC $\rightarrow$ TCon $\rightarrow$ Login) causing a transition to the *Login/Discovery* state. These state transitions always apply to both parties; that is, the initiator and the target should assume an identical state at all times. Coming from a *Login* state it is possible for the initiator to discover

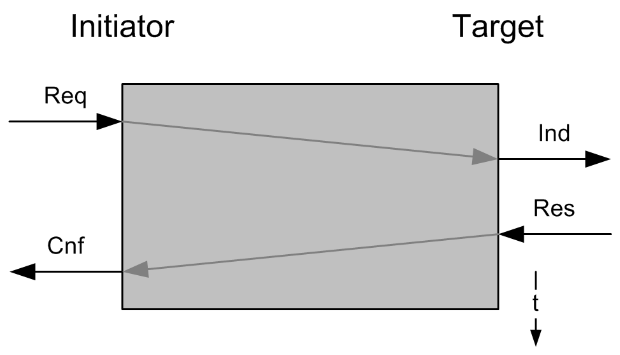


Figure 3.2: Request/Response Model

Layer	Service	Service Primitive
Transport (T)	Connect (Con) Disconnect (Dis)	Request (Req) Indication (Ind) Response (Rsp) Confirmation (Cnf)
Session (S)	Discover Login Close CmdCall CmdReturn Ack Data Leave LeaveForce	

Table 3.1: Service Primitives

the target causing the $S_t Discover$ loop. The notation of S followed by a subscript t stands for text-based session layer communication as d corresponds to binary data communication happening during *Full-Feature* phase. To commence full operation the initiator now can dispatch a $S_t Login$ request (implicating a Request-Response sequence including intermediate state) while at the same time also negotiation may take place. *Full-Feature* phase is sustained by subsequent calls to services of type $S_d CmdCall$, $S_d CmdReturn$, $S_d Ack$, or $S_d Data$. Albeit, the use of the different services may possibly be constrained to only one of the to involved endpoints, as is $S_d CmdCall$ in case of the initiator and $S_d CmdReturn$, $S_d Ack$ in case of the target. $S_d Data$ may be utilized by both service consumers allowing the bidirectional transmission of data premised in the previous section (3.1). The binary communication phase may only be left using the service primitives $S_d LeaveReq$ or $S_d LeaveForceReq$, where Leave exits to the *Login/Discovery* state (with accompanying $S_d LeaveInd$ indication for the far side party) and LeaveForce immediately terminates the underlying transport connection (TDisInd) without further communication resulting in a *Not Connected* state. As the prefix

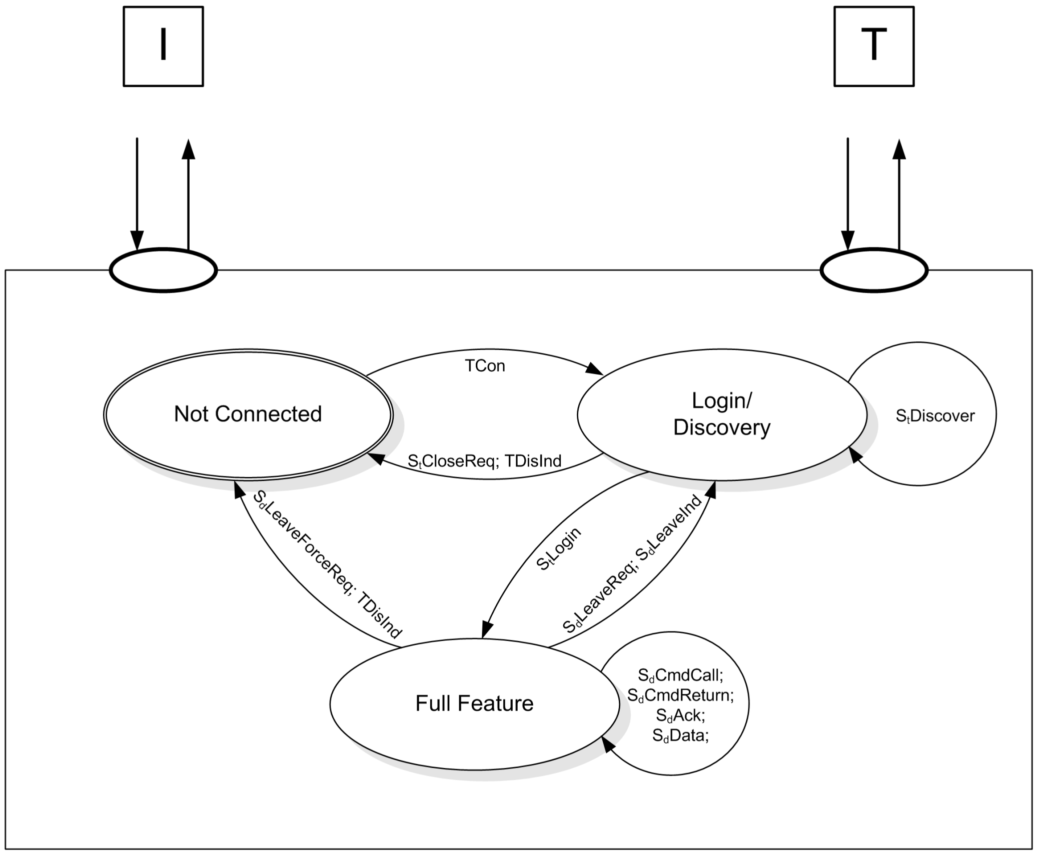

I
T
Not Connected
TCon
Login/ Discovery
S_tCloseReq; TDisInd
S_tDiscover
S_dLeaveForceReq; TDisInd
S_tLogin
S_dLeaveReq; S_dLeaveInd
Full Feature
S_dCmdCall; S_dCmdReturn; S_dAck; S_dData;

Figure 3.3: Protocol State Machine

S_d indicates the Leave service is not communicated by a text-based request in *Full-Feature* phase, but by a binary command packet. A valid session always ends in the *Not Connected* start/end state implying an eventual $S_tCloseReq$ request (in case the *Full-Feature* phase has been left using the $S_dLeaveReq$ service primitive).

Text-Phase Protocol

The text-based protocol messages consist of an input string which may not exceed a length of 4096 bytes while every character is of 8 bits in size with only the lower 7 bits of the ASCII code (0–127) in use. Incompliant characters should be avoided as the protocol behavior when processing this data is not defined. Due to the fact that the input string length is unknown at the time when the initial connection is made the string has to be terminated with a well-known character sequence delimiting the transition from text to binary communication. Posing HTTP (HyperText Transfer Protocol) as example for a standardized Internet protocol the protocol to be designed adopts the empty line delimiter ($\backslash n\backslash n$) also facilitating access to the protocol for a console user (e.g., via telnet). The common format for data exchange over the text-phase protocol is a "`<Property>: <Value>`" sequence of lines (separated by $\backslash n$ characters). The occurrence of at least one property/value tuple determining the service primitive to be invoked is mandatory. Thus, a minimal example for a valid text-phase message turns out to be of the form:

```
Primitive: LoginReq\n
\n
```

Figure 3.4: Minimal Message (example $S_t LoginReq$)

Apart from the ability to transmit other session-specific data this text-based message protocol is also extensible in terms of new functionality which might be introduced in newer versions of this protocol. For example, the property/value format allows negotiation of additional features where the initiator sends a login request, which contains a property enumerating features it is capable to support. The target compares this list with its own capabilities and returns the intersection of the two sets in the subsequent login response.

Initiator

```
Primitive: LoginReq
Initiator: de.jbls.client
Capabilities: multi-path, v2
Isid: 1                         # Initiator Session ID
Device: cdrw, 2:1
User: jlieder                   # User Name
Secret: *****                   # Password Protection (opt)
# Requesting access to device(s) 2:1 and cdrw.
```

Target

```
Primitive: LoginRes
Target: de.jbls.storage
Capabilities: multi-path        # Confirming multi-path cap
Tsid: 1                         # Target Session ID
Usid: 65537                     # Unique Session ID
# Granting access to atp://de.jbls.storage,cdrw.
# Granting access to atp://de.jbls.storage,2:1.
# Switching to Full-Feature mode... you may proceed.
```

Figure 3.5: Sample Negotiation

The previous example (fig. 3.5) shows a sample conversation between initiator and target during login-phase negotiating different parameters for the establishment of a new session. Endpoint names do appear in reversed FQDN (Fully Qualified Domain Name) notation; that is inversed DNS order with TLD (Top Level Domain) to host name from left to right. This is mostly equivalent to the iSCSI convention (see RFC 3720, [11], 3.2.6. iSCSI Names) where the name has to fulfill the requirements for Uniform Resource Names (URN). Hence, the proposal for a general format of ATP-URIs (ASPI Tunnelling Protocol Uniform Resource Identifier) is:

```
atp://<tld.dns.rev>,<alias or remote HA:ID>
```

The reversed DNS notation provides a unique qualifier (at least in the local network) and also allows further indirection by a discovery server or service. Another approach of consolidation could be multi-naming by the DNS naming service which resolves multiple Resource Names (URNs) to the target's IP-Address. The target finally provides the appropriate resource by evaluating a passed "`Target:`" property (comparable to the Virtual Host feature of modern HTTP servers). Nevertheless, in the local network the target name "*workstation*" (no dots) simply remains "*workstation*".

In addition to the already mentioned negotiation of a common set of capabilities a Unique Session ID (USID) is determined (calculated from the initiator's and target's local SID). This ID allows the identification of a session which is necessary when implementing advanced functionality. For instance, the presented software architecture will be prepared for techniques like **Multi-Threading** and **Multi-Path I/O** (which potentially depends on such a SID mechanism) intended to be discussed among other design decisions in the following implementation design chapter. However, actual specification of a full set of allowed session negotiation properties is beyond the scope of this protocol definition. The only specified properties are listed in table 3.2 where "`I`" means the property has to be used in a text-phase *request*. An "`O`" property has an analogous meaning for the according *response*. Thus, referring to the aforementioned example (fig. 3.5) the `Primitive` property appears in both the login request and the subsequent login response. Finally, "`N`" stands for negotiation (ranging across a request/response conversation and implying the previously defined negotiation algorithm) of the corresponding property.

Property	Type
`Primitive`	`I/O`
`Capabilities`	`N`

`I`:	Request Property
`O`:	Response Property
`N`:	Request/Response Negotiable Property

Table 3.2: Property Types

Full-Feature Phase Protocol

Unlike examined in the previous paragraphs the following section elaborates a protocol definition for the opposing full-feature phase binary protocol. Thus, these protocols have to be considered completely independent from a design point of view. Like any other protocol tier the session layer packet format has to encapsulate payload data given in form of SDUs (Service Data Units) from superior tiers. This encapsulation has to be accomplished by initializing in-memory structures comprising preceding header and optional succeeding trailer structures, representing the actual PDU (Protocol Data Unit), which then can be passed to the underlying layer. Actually, the main task for the session layer, namely the correct initiation completed during text-phase and termination of logical sessions completed by a particular type of data packet (containing a `LEAVE` command code), is realized. Apart from this primarily session-specific functionality the session layer also introduces the smallest communication unit in form of a well-defined packet format. In consequence, this can be visualized as a resulting stream of continuous session layer packets travelling along the transport connection. As the transmission of data in binary mode is also of sequential nature similar problems as in case of the text-based protocol in terms of unknown message length emerge. A basic view of the packet header is shown in figure 3.6.

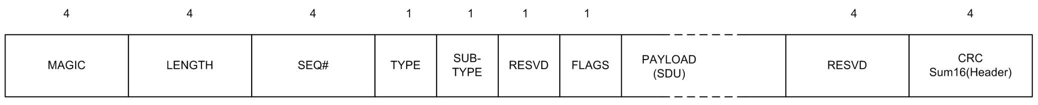


Figure 3.6: Session Layer Packet (Generic Layout)

The session layer packet header comprises several fields including data for protocol-related communication and members increasing protocol robustness. First, a valid packet begins with the magic sequence of 4 bytes containing hex value `0x51` (which is equivalent to the ASCII string `"QQQQ"`). Through the usage of a magic packet header it is likely to detect any protocol mismatch or protocol violation during communication. The next field holds the length of the entire packet (4 bytes, unsigned integer) that is to be processed. As per specification every packet has to have a length multiple of 64 bytes. Thus, the minimum packet size equals to 64 bytes where unused sections in the payload field should be padded with zeros to reach the appropriate size. This restriction, however, involves some advantages for the software implementation as in the situation of processing the first packet it is legal to receive a block of 64 bytes while the actual length of the packet is still unknown. A continuous stream of this length guarantees the presence of the full header allowing magic and packet length field to be extracted for sanity check. The overhead introduced by the "modulo 64" stipulation should neither state a problem for the upper layers (as padding and unpadding happens transparently) nor for the underlying layers where often the physical layer finally has to perform some kind of padding (e.g., Gigabit Ethernet, due to very short timings in conjunction with collision detection). Nevertheless, this might introduce a marginally increased message latency. The third field (again 4 bytes, unsigned integer) contains a sequence number which will be monotonically increased on both the initiator and target, allowing a mechanism to continuously verify packet order (or detect potential packet loss of unreliable transport connections). The last header field comprises four single byte members, where the first two members specify the command `TYPE` and `SUBTYPE` transported by the corresponding packet. All valid combinations of `TYPE` and `SUBTYPE` constants are listed in table 3.3.

TYPE	SUBTYPE	COMMAND
0x00	0x00	INVALID
	0x01	NOP
	0x02	ECHO
0x01	0x00	INVALID
	0x01	CMD_CALL
	0x02	CMD_RETURN
0x02	0x00	DATA
0x03	0x00	ACK
0x04	0x00	RETR
0xFF	0x00	INVALID
	0x01	LEAVE
	0x02	LEAVE_FORCE

Table 3.3: Command Constants

Further, `SUBTYPE` is followed by a reserved field (set to `0x00`) and a `FLAGS` field (each 1 byte in size), which contains contextual information for the session layer in form of two bits `FLAGS_FINAL` (`0x01`) and `FLAGS_CONTD` (`0x02`). These flags are mutually exclusive as `FLAGS_CONTD` indicates that a packet is to be "`CONTINUED`" (unless marked "`FINAL`"); that is, subsequent packets have to be processed to receive all information needed to build a complete upper layer packet. The residual bits of the `FLAGS` field are reserved for future use and should be set to zero.

Apart from a preceding header the session layer packet format also specifies a trailer containing two additional fields. Both, the first reserved field (always equal to zero) and the second CRC field are 4 bytes in size. The latter member should assure the packet header's integrity by calculating a corresponding Cyclic Redundancy Check value. However, a particular CRC or hash function will not be specified by this protocol proposal as long as the algorithm is deterministic in terms of repeatability (viz. the opposing session layer protocol instances have to be able to generate and verify the hash value). Unless necessary (e.g., in case the underlying transport layer guarantees data integrity) an error check might also be omitted by using a constant value verification (for example a bit-field equivalent to `0x8181`). In summary, session layer header and trailer exhibit a combined length (or overhead) of 24 bytes. Therefore, while generally not being constrained in length a *minimal* session layer packet (64 bytes) may carry a payload of 0 through 40 bytes.

Sync-Points

Like iSCSI the ASPI Tunnelling Protocol is designed to support Sync-Points (see [11], RFC 3720, 3.2.8 Fixed Interval Markers (FIM)). These Sync-Points are inserted into the session layer communication stream at fixed intervals where pointer values determine the start of the next packet. The insertion of these markers can be accomplished transparently to the remaining software layers and represents an advanced mechanism to overcome TCP's latency-related drawbacks (in situations of packet loss when out-of-order frames are temporarily stored in buffers at transport layer level). The Sync-Point interval might be negotiated during the login phase, whereas a value of zero means the omission of these markers. Assuming a collision- and routing-free local network, the following solution will not implement sync-points, therefore behaving like an extended ATP implementation with a negotiated sync-point interval of zero (proposed text-phase property: "`Interval: 1000`").

Tunnel Layer

The tunnel layer packet layout is quite similar to the session layer format previously discussed. A magic header field for identification purposes is not necessary so the first field is a tunnel layer related sequence number followed by fields containing the payload length and the payload offset. The latter should always yield a value of `0x08` as only another reserved field (4 bytes again holding a zero value) joins counting from the start of the same field until the end of the reserved field. The offset value should be fixed for all 32-bit and 64-bit architectures (in terms of memory layout) and might serve as check in case of a new protocol version where additional header fields could have been introduced. Thus, by skipping these extensible fields as well as having direct access to the encapsulated payload (by the use of this *pointer* and the corresponding length) a certain version independence also could be realized. The tunnel layer packet format also includes a trailer which is identical to the session layer trailer except for the semantics of the checksum field. Unlike the header, this checksum value references to the handled payload. In other words, it protects the *encapsulated* data. Thus, along with session layer error check mechanism integrity of packet traffic *and* the application-originated payload data may be presumed. In addition to yet another encapsulation the tunnel layer also transparently introduces application-specific semantics by segmenting and reassembling the logical unit of SCSI Request Block (SRB) and its optional data buffer. The tunnel layer decides whether the data buffer has to be transmitted depending on the type of command being dispatched. It might not be necessary to transport the receiving data buffer to the command's target when a Non-Data (`N`) or `READ` command (`R`) is pending to be executed, and vice versa (for `WRITE` commands (`W`)). The abbreviations refer to the common notion used in [5], p. 16.

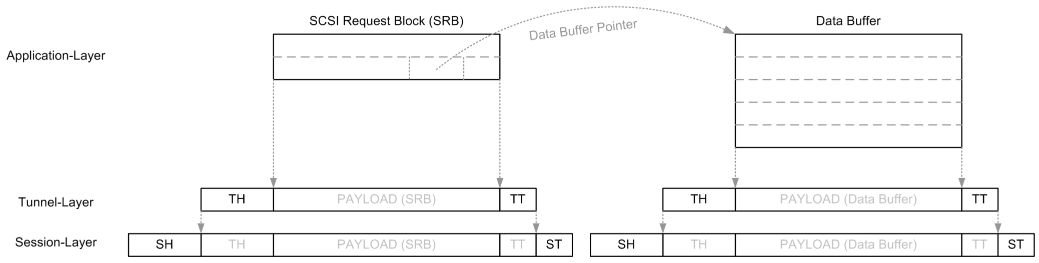


Figure 3.7: Segmentation/Reassembly

In summary, for every command at least one tunnel layer (and therefore one session layer) layer packet is generated. If the transmission of an accompanying data block is required a second tunnel layer packet will be constructed carrying the same tunnel layer sequence number (indicating allegiance to to the same application layer command). These two packets in turn will be handed over to the session layer where again two session layer packets of type `CMD_CALL` (or `CMD_RETURN/ACK`) and `DATA` are generated actually containing the application-level payload.

In contrast to the standards discussed earlier (FC, iSCSI and SAS), all implementing a mechanism (*Ready2Transmit* or equivalent) to avoid transmission of unsolicited data, the ASPI Tunnelling Protocol hands the responsibility to determine the necessity for buffer data transmission to the corresponding initiator. Even though this implies the requirement for large buffers (as stated in [5], p. 28), for ATP, however, there is no unsolicited data. Thus, it is appropriate to avoid an additional R2T mechanism (which actually means transmission of an intermediate notification command) potentially causing increased latency for the eventual command execution. In fact, if indicated by the initiator the buffer is prerequisite for proper execution.

Finally, the thin application layer (see section 3.1) handles additional management tasks (including integrated logic to assert local memory coherency as will be discussed in the subsequent section) and finally pro-

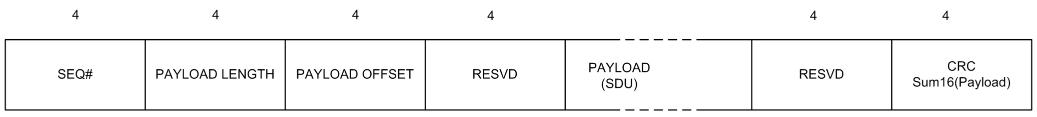


Figure 3.8: Layer 6 Packet (Generic Layout)

vides an adequate programming interface (API) for both the initiator and the target. In case of the initiator this is a function set equivalent to `SendASPI32Command()` and `GetASPI32SupportInfo()`. For the target a callback function, which will be invoked upon command arrival is required.

3.3 Implementation Design

Similar to the protocol design procedure the implementation design will also be developed in a bottom-up approach. Thus, beginning with the lowest software layer means the development of a Network Abstraction Layer (NAL) which can help in the attempt to abstract from various O/S-specific peculiarities. This might also facilitate a cleaner design for the remaining layers building upon this abstraction framework. As prominent example the Microsoft Windows operating system with its superseding WinSock API (see chapter 2.1.3) may be mentioned. While preserving the defined class interface the underlying network stack may be exchanged (for example by defining platform-dependent preprocessor symbols). The same applies to future technologies like IPv6 (see [3]) which should be taken into account right from the early stages of design.

These objectives are achieved by introducing the class hierarchy of `CGenericEndpoint` (as abstract base class) stipulating the generic interface, and `CTransportEndpoint`, which finally implements the corresponding functionality. The term "`Generic`" as element of a class name suggests the abstract nature of this class as can be seen in the case of the given example. The following descriptive texts (particularly in this section) will be aligned with the notions and peculiarities of the C/C++ language since the presented solution has been realized with this programming language, as well as the term layer will be primarily used in the meaning of software layer, unless otherwise noted.

The lowest layer of abstraction, namely `CGenericEndpoint`, is implemented in *Network.cpp* and defined in the corresponding header file *Network.h*. Generally, `CGenericEndpoint` (and the later `CTransportEndpoint`) are designed to provide an encapsulation for socket handles and the function set operating on these handles (e.g., `connect()`, `send()`, and `recv()`). Apart from this set of functions, yet defined purely virtual, this abstract class implements three static functions responsible for name and service resolution (tasks which are independent from particular socket handles). Further, *Network.h* introduces infrastructure needed for Internet Protocol (IP) version independence (for example, the use of `struct in_addr/sockaddr_in` versus `struct in6_addr/sockaddr_in6`). With the definition of a preprocessor symbol `__IPV6__` this software layer can be switched completely to IPv6 support while also remaining compatible to compilation under Unix operating systems. IPv6 support can be determined at runtime by evaluating the static variable `CGenericEndpoint::s_bIpVer6`. For the Windows platform `_WsaTakeCare()` takes care of the timely initialization and finalization of WinSock library (by keeping account of the existing `CGenericEndpoint` objects). The transport layer (*Transport.cpp*) builds on the given infrastructure and implements the residual functionality with `CTransportEndpoint` (namely all common socket functions). The class is independent from a particular transport protocol (TCP/UDP are supported) and allows the registration of a callback handler (`RegisterAcceptHandler()`), which will be invoked on connection request (this only applies to an endpoint operating in *server-mode*). As with connection acceptance the requesting peer's address is also passed, a mechanism for address or network-based connection filtering can be implemented (see the later implementation of the application-level target class).

This also leads to a design issue, yet to be encountered, which actually concerns the application-level objects rather than the current topic of low-level transport endpoints. However, from a software design point of view, unfortunately, initiator and target classes cannot be architected symmetrically in their use of transport protocol endpoints objects. While the high-level initiator can delegate the creation of a transport connection to the lower layers that are actually responsible for this task, in a top-down approach the opposing application-level target object has to create its own listening transport-layer endpoint (the creation takes place within the application layer). On the event of accepting a new connection the newly acquired endpoint object (or socket handle) has to be published to the underlying layers upon construction, resulting in an iterative hand-over of this handle down to the lowest layer.

However, with the implementation of *Network.cpp* and *Transport.cpp* the functionality of ISO/OSI layers 1 through 4 is adapted and available as foundation the remaining software, namely the adjacent session layer, can be built on. This will be subject of the next paragraphs.

The session layer class `CGenericSession` introduces the basic framework for the entire session layer functionality. That is, the assignment of session endpoint roles (session initiator or target), a session state and the corresponding state machine, and an interface for sending and receiving text-

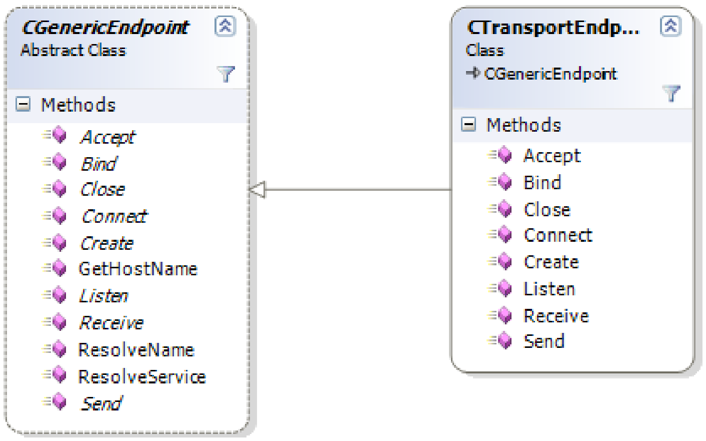


Figure 3.9: Transport Endpoint Classes

phase messages as well as full-feature phase packets. However, due to the abstract nature of this class (see `CGenericEndpoint`) the actual functionality has to be implemented by an inherited class. This, in turn, increases flexibility for the software solution to incorporate or realize different techniques (i.e., various differently powerful classes). Along with these particular considerations one might also anticipate the potential benefit in conjunction with the previously discussed negotiation mechanism. A fact, which will be important for the constitutive architecture as the software should be designed to support Multi-Path I/O (see section 3.2), which is a technique to deploy multiple transport connections at a low level by reasons of increased resilience and throughput. By locating this functionality in a preferably low software layer (obviously the session layer) the upper layers can be insulated from these peculiarities. The first incarnation of the code will only realize a single-pathed nexus yet maintaining an augmented code infrastructure ready for advanced features. Here the notion of a nexus refers to the logical connection between initiator and target, which (in case of an multi-path nexus) might accommodate multiple paths. Thus, a path forms the smallest logical unit of communication (in this case simply a transport layer connection). In summary, the superior session class delegates the eventual task of packet transmission to the associated classes (visualized by the UML association between the involved classes, see fig. 3.10). When there should be used multiple paths the responsible session class (`CMultiPathSession`) creates a nexus capable of managing such a facility (`CMultiPathNexus`). This multi-path nexus finally sets up the required communication paths (likely a primary communication path also responsible for management tasks which then might negotiate the further installation of simple paths (`CSimplePath`)). Nevertheless, this functionality still belongs to (and happens within) the session layer and does not represent a violation of the layered paradigm. From a session's point of view these *late connections* implicate continuing issues when connecting with the target. This fact also influenced some design decisions. As already mentioned in the preceding section a clean software design had to be abandoned due to the need for a central management of the server socket. This now comes into play when considering a situation where multiple sessions have to co-exist while still having to be ready to accept further connection request. The scenario gets even worse for the creation of multiple paths since the corresponding session object has no authority to control the serving connection endpoint. Thus, if the affected session would maintain its own transport endpoint – in addition to the actual service socket – to allow authoritative path management, new *dynamic* service port numbers had to be communicated. Since the set of port numbers for a service should be predictable, this dynamic (and protocol-specific) port behavior renders the service impractical when being deployed in firewalled scenarios. However, the primary argument against this approach is finally the fact that distinct session instances should not be aware of their sibling instances by design. The disadvantage of the proposed solution is the reversed assignment of responsibility for the target class hierarchy where the application-level object becomes the connection principal (in contrast to the initiator where the responsibility for transport connection is assigned to the lowest session layer), which leads to this asymmetrical design. Still the possibility for the application-layer (AL) target to distinguish between the established sessions has to be given, realized by the already introduced Session ID (SID). In other words, the SID defines the session allegiance for a new path connection. Thus, prior to the assignment of a new path the peer has to perform a text-based conversation with the AL-Target which then delegates the current transport endpoint to the responsible session object (by using the submitted SID). The delegation takes place by passing the endpoint object to the constructor of the corresponding child object. Both discussed scenarios should be roughly visualized by figure 3.11.

In case of the session classes (`CGenericSession`/`CSinglePathSession`) there are three types of members functions:

nexus management	`EstablishNexus()`, `CloseNexus()`
text-phase protocol	`SendText()`/`ReceiveText()` and `SendLoginRequest()`/`ReceiveLoginResponse()`
full-feature phase communication	`SendPacket()`/`ReceivePacket()` and `NoOperationCommand()`/`LeaveCommand()`

Most part of the session layer (fig. 3.11) should be considered as framework and abstraction for the support of multiple communication paths, yet the code responsible for tunnelling comprises the most important function-

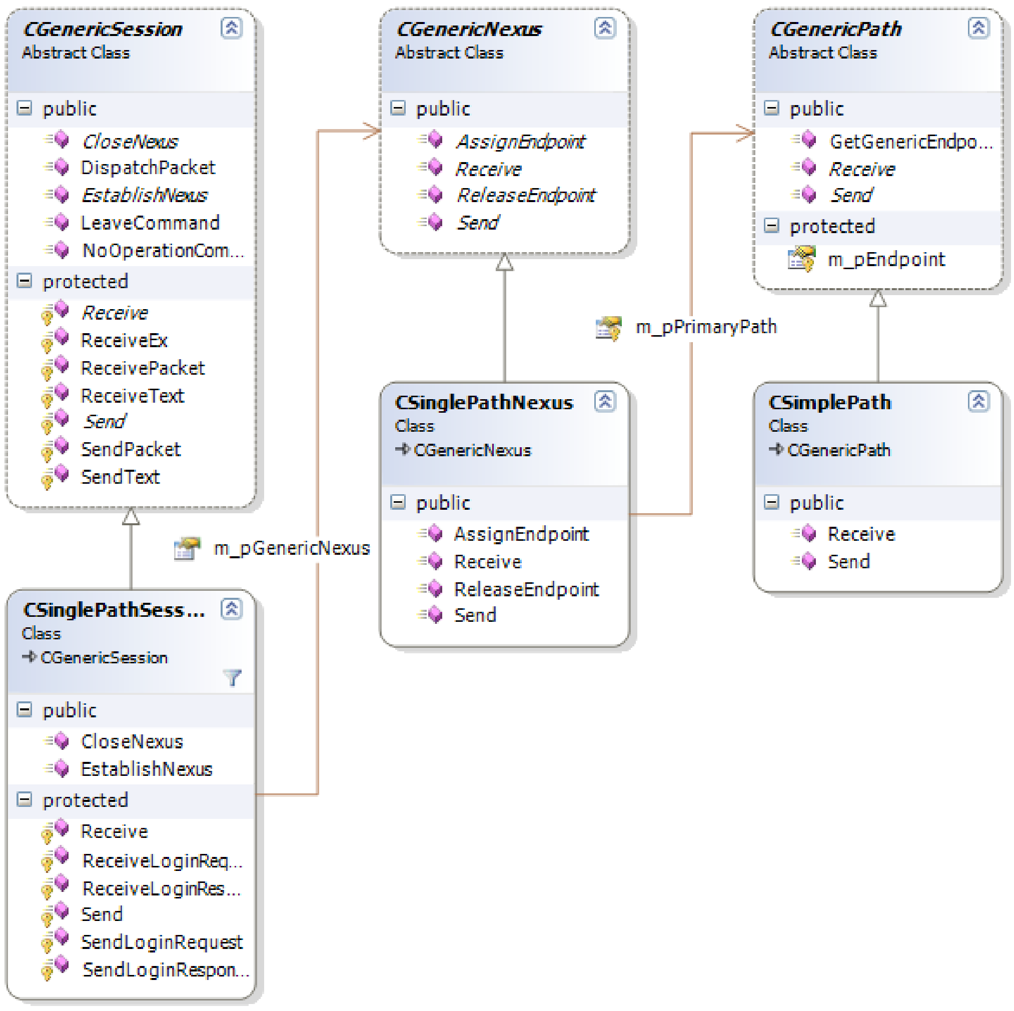


Figure 3.10: Session Layer Class Hierarchy

ality. The SESSION_LAYER_PACKET (fig. 3.12) structure provides all necessary information (Service Data Unit and Interface Control Information) for SendPacket() to assemble a continuous memory block (the actual session layer packet, see section 3.2, Full-Feature Phase Protocol) which eventually can be sent to the underlying transport layer.

With *Session.cpp* and *Nexus.cpp* (implementing nexus & paths) the necessary functionality of layer 5 for session initiation and basic communication is implemented. Now, initiator and target are already able to remotely invoke generic commands and exchange unstructured payload data.

In overall design the next software layer, for example the tunnel layer's packet generation routines, is very similar to the previously presented session

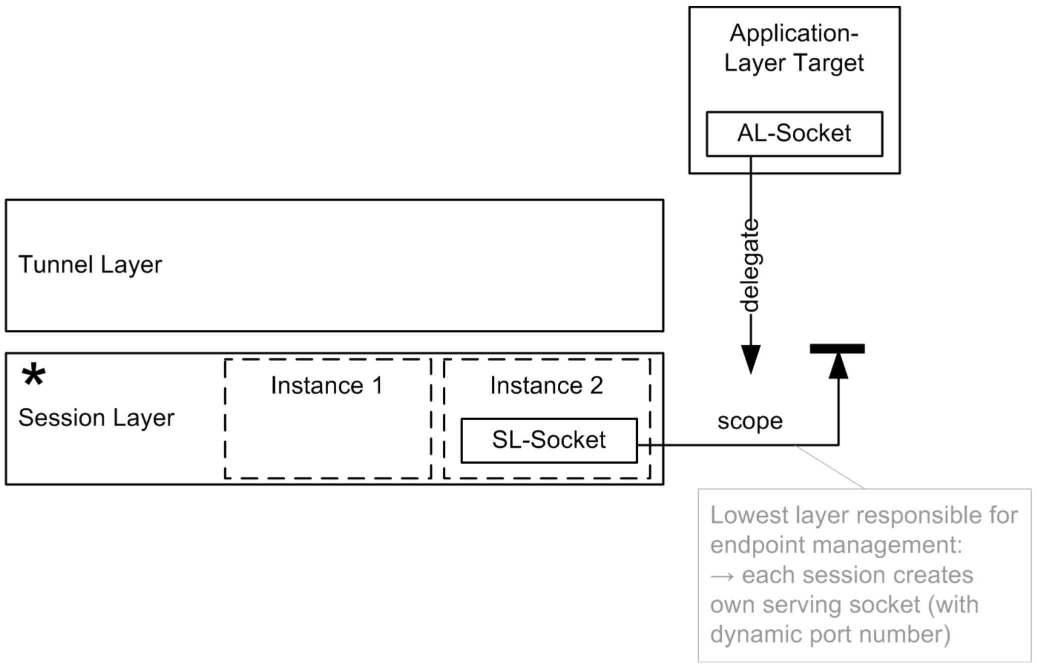

Application-
Layer Target
AL-Socket
delegate
Tunnel Layer
*
Session Layer
Instance 1
Instance 2
SL-Socket
scope
Lowest layer responsible for endpoint management:
→ each session creates own serving socket (with dynamic port number)

Figure 3.11: Design Decision: Server Socket managed by Application-Layer

layer. Its primary objective is to introduce yet another encapsulation layer for further convergence to the application layer, thus different mechanisms to facilitate tunnelling of application-level objects will be added. However, apart from the tunnel layer's deviating packet format, PDU (Protocol Data Unit) generation in general is realized identically.

The application-related functionality is defined by the `CGenericTunnel` class also preparing interface functions for the later initiator and target-based roles. As before, this class is the abstract ancestor for further inherited software tiers. Only infrastructure for SRB-related tasks and callback handler management for command arrival is provided. These are the common tasks finally required in the inheriting child classes (`CTunnelEndpoint`, `CTunnelInitiator`, and `CTunnelTarget`).

The next level of functionality is introduced by the former class `CTunnel-Endpoint` which comprises tunnel-layer packet generation (including segmentation and reassembly) and procedures allowing R/R-based (see section 2.1.1) communication of entire SRBs and their logically linked data blocks. Apart from only transmitting the payload data, this set of functions (`TunnelCommand()`, `ReturnCommand()` and `AcknowledgeCommand()`) also transports semantics (see section 3.2, session layer and the defined command types). The type of command influences the kind of behavior which is invoked. Since in the direction from initiator to target (request) there is

```
enum SESSION_LAYER_PCI_TYPE
{
        slInvalid,
        slNoOperation,
        slCommand,
        slDataSegment,
        slAcknowledge,
        slRetransmit,
        slLeave
};

struct SESSION_LAYER_PCI
{
        SESSION_LAYER_PCI_TYPE spciType;

        BYTE bySubType;

        BOOL bContinued;
        BOOL bFinal;
};

struct SESSION_LAYER_PACKET
{
        UPPER_LAYER_PACKET *pBottomPdu;

        SESSION_LAYER_PCI spciControlInfo;
        UPPER_LAYER_PACKET ulTunnelPdu;
};
```

Figure 3.12: Session Layer Protocol Interface

only the `CMD_CALL` primitive (dispatched by `TunnelCommand()`), as response the two primitives `CMD_RETURN/ACK` (dispatched by `ReturnCommand()` and `AcknowledgeCommand()`, respecitvely) are valid. The exact communication model for the different types of implementation will be discussed in the subsequent sections. Nevertheless, the tunnel endpoint class is also responsible for transparent transmission of SRBs; that, is coherency between the SRB's data buffer pointer and the actual location of its buffer (the target uses local temporary buffers) has to be maintained.

As already mentioned in section 3.2 probably two session layer PDUs have to be generated for a single SRB to be tunnelled. Nevertheless, since the session layer builds on transport layer functionality, there is no maximum transfer size limitation and thus no need for any kind of fragmentation. This also applies to the generation of tunnel layer PDUs and the corresponding SDUs (namely, application layer SRB and data buffers). Finally, this means the responsible tunnel layer may simply generate two packets and pass them to the underlying layer, if the layer's remote instance guarantees reassembly of both PDUs to a valid SRB. This behavior requires additional functionality in the `CTunnelEndpoint` class which is implemented by the `UpdateSrb()` function taking care of correct updates to a SRB memory structure. These updates to in-memory SRBs represent a critical task as the contained pointers are only valid in the local domain of the executing machine. Generally, a SRB and a data buffer are unrelated, unless the SRB points to this data buffer prior to execution. On a remote computer the virtual memory layout may look completely different requiring the correction of all data buffer pointers. However, the additional effort confines to the processing of `EXEC_SCSI_CMD` request blocks (since only these SRBs contain pointers). Another high level task for the tunnel layer is the maintenance of sequence numbers which depend on the client application's dispatch order. Especially for a multi-threaded implementation sequence numbers are important as ordered queues have to be processed and acknowledgements may arrive out of order.

The next level of inheritance is realized by the classes `CTunnelInitiator` and `CTunnelTarget`, which customize the generic behavior (of `CTunnelEndpoint`) for the initiator and target roles of the ATP protocol. By using inheritance it is possible for the child classes to override the introduced common functionality. For example `DispatchCommands()` has been implemented in `CTunnelEndpoint` realizing a dispatch loop for arriving tunnel layer packets by continuously calling `DispatchCommand()` (until a session layer `LEAVE` command is encountered). By overriding `DispatchCommand()` within `CTunnelInitiator` and `CTunnelTarget`, the target is able to adequately process `CMD_CALL` packets and the initiator may react on `CMD_RETURN` and `ACK` messages. The exact implementation of these two classes determines whether the later ATP initiator or target will act synchronously or asynchronously. In the next two sections this will be discussed in more detail.

The final software layer is the application layer. As announced in section 3.1 this layer will be relatively thin. Actually, the involved classes (in Application.cpp) do not participate in the previous class hierarchy, but they do rely on the accumulated functionality of the ATP protocol stack by creating instances of `CTunnelInitiator` and `CTunnelTarget`, respectively.

The main assignment for an application layer initiator class is the provision of a standard ASPI programming interface (`SendASPI32Command()` and `GetASPI32SupportInfo()`) which then can be used in client applications to transparently include ASPI services or to export via DLL (Dynamic Link Library). Since the `CTunnelInitiator` class already implements most of the required functionality the `CAtpInitiator` class remains relatively compact.

While not being included in the current implementation the class should also be responsible for providing a virtual Host Bus Adapter (which allows access to a set of ATP tunnels leading to different targets/target devices) and accordingly loading an adequate configuration on initialization.

In contrast to the initiator `CAtpTarget` maintains connection management. On the event when a client connects to the server socket the ATP target class hands over the new connection to a dynamically created `CTunnelTarget` object. Before application layer communications may commence `CAtpTarget` also registers a default SRB handler function with the tunnel target class. This callback function (`DefaultSrbHandler`) is a static member function of `CAtpTarget` and acts as default implementation which simply dispatches the incoming SRB requests to the target's local WinASPI layer for actual execution.

For a comprehensive overview one may find a complete UML diagram in Appendix A including the entire ATP class hierarchy. It might help in surveying the solution's principal structure and design outlined in the past chapter. A valuable hint for easier understanding might be the recognition of associations as links between protocol layers/software sub-layers.

3.4 Synchronous Implementation

A synchronous software implementation is characterized by its procedural design. At runtime it is possible to exactly determine the algorithm's next internal state if the previous state was also known. Hence, both endpoints behave as deterministic automatons realizing a strict horizontal communication protocol which under no circumstances may be violated (in contrast to the later asynchronous implementation where these commitments will be relaxed by a certain degree). The actual communication sequence of the synchronous implementation is shown in figure 3.13. Referring to previous considerations (see chapter 1) a basic R/R-model has to be realized resulting in two distinct states *receiving* and *sending*, an endpoint may assume. Due to the asynchronous nature of some ASPI function calls (deferred completion notification) this fact has to be regarded when designing the protocol communication sequence. In the first step **(1)** the initiator tunnels a SRB by invoking `TunnelCommand()` (which maps to a `CMD_CALL` packet on session layer) while the target is in *receiving* mode. The target knows (by specification) that a single tunnel layer PDU arrives and executes the ASPI command after the corresponding SRB has been placed properly into memory **(2)**. When regaining program control from the WinASPI layer the target changes to *sending* mode **(3)** and dispatches a `ReturnCommand()` to notify the *receiving* initiator about this event as soon as possible (since `ReturnCommand()` also submits the target's updated SRB memory structure the initiator may evaluate the return status of `SendASPI32Command()`). If the issued ASPI command behaves asynchronously (see *Tunnel.cpp*, `CGenericTunnel::IsAsyncSrb(LPSRB)`) the target has to check for SRB completion which depends on notification method previously passed in the `SRB_Flags` field. However, some of these flags will be masked out to determine a common notification mechanism on the target side (preferably the Windows Event Notification API (`SRB_EVENT_NOTIFY`) for low CPU overhead, see [2], p. 31). If the SRB status already is not equal to `SS_PENDING` (see [2], p. 34) the command is completed or has failed. Otherwise the process ceases operation cooperatively until completion by calling the Windows API `WaitForSingleObject()`. Although rendering this implementation suboptimal an `AcknowledgeCommand` PDU is sent **(4)** even if a synchronous SRB has been transmitted. The initiator in turn waits for both messages and modifies the local memory facilitating a transparent ASPI layer emulation for the client application. The client may (legally) poll the SRB's status field for completion or error code. The downside of this approach is the inability to process multiple SRBs in parallel which might be desirable for increased-demand scenarios. Additionally, supplemental CPU resources in form of multiple system cores cannot be used since all activity takes place in a single thread within a process.

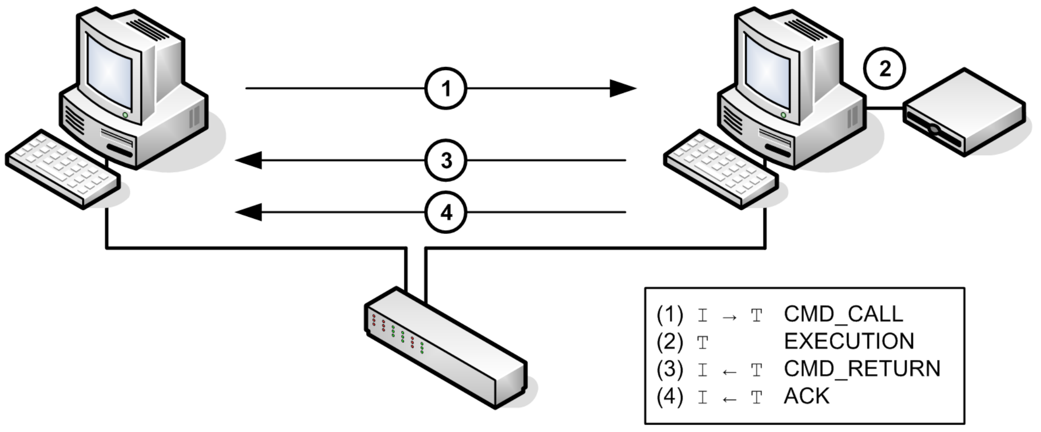


Figure 3.13: Synchronous Implementation: Sequence Diagram

The source of a synchronous implementation is not provided with this solution as the approach only marks an intermediate state of the work. Various modifications had to be applied to the code to allow multi-threaded operation and improvements have only been incorporated into the later code base, thus both implementations might be recognized as independent branches of the solution. However, finally it should be easy to build an advanced synchronous implementation drawing benefits from the results of this work and building on the presented ASPI Tunnelling Protocol stack and infrastructure.

3.5 Asynchronous Implementation

For the intention to implement asynchronous protocol instances additional code infrastructure has to be at the programmer's disposal since constructs to protect critical code sections are essential in multi-threaded programming. Furthermore, different kinds of queues acting as input and output mechanisms between dependent threads will be needed. The most prominent types are the FIFO (first-in first-out) queue and especially in communications software the priority heap for tasks like packet reordering. Most of these ADTs (Abstract Data Types) provided by standard programming libraries (e.g., Standard Template Library, STL) are not re-entrant and thus not applicable to multi-threaded programs. As the presented solution will utilize the validated STL classes `queue` and `priority_queue` to avoid errors being potentially introduced by custom code, the ATDs have to be enveloped by classes responsible for adequate locking. Functions like `Enqueue()` and `Dequeue()` have to be of atomic nature. This might also apply to operations on numeric values implying the need for interlocked variable classes. The source file *Queue.cpp* and its header aggregate this functionality by implementing `CGenericQueue`, `CSimpleQueue` (encapsulates `std::queue`), and `CSyncQueue`. The priority heap is implemented by `CPriorityQueue` (using an instance of `std::priority_queue`) and synchronized by `CSyncPrioQueue`. Finally, the last type of ADT is realized by `CBoundedBuffer<TYPE>` (inherits from `CTypedPtrArray<CPtrArray, TYPE>`, provided by Microsoft Foundation Classes (MFC)). The bounded buffer provides a number of slots in which incoming acknowledgements from the target can be arranged. Like previously visualized by template notation, all classes are templated and thus type-independent (as usually demanded from ADTs). Another important class is `CSignal` which facilitates signaling of events between threads by using the *Windows Event Notification* API. Hence, instances of `CSignal` (containing a Windows event handle) can also be used for convenience on the initiator to notify the client application of command completion (ASPI layer emulation) and completion tasks on the target (see previous section).

Multi-threaded design primarily affects the lowest sub-tier of the tunnel layer, namely the classes `CTunnelInitiator` and `CTunnelTarget`. In the former class two queues (`m_psqSendQueue`, `m_psqSentQueue`) and a bounded buffer (`m_psbPendingBuffer`) are introduced. With a client application dispatching a `SendASPI32Command()` function call the accompanying SRB is enqueued in the *Send Queue*. There are also three types of threads completing different tasks. The *Transmit Thread* continuously dequeues elements from the *Send Queue* for transmission over the ATP tunnel. After tunnelling the SRB is categorized as *sent* by adding the address of the SRB to the *Sent Queue*. However, prior to the blocking transmission the thread has to check whether the *Sent Queue* may accept another element. Otherwise it is blocked until an element gets freed (by using three interleaved critical section locks as internal signalling mechanism, see *producer/consumer* problem). This is the sequential behavior referred to in section 2.1.3. The blocking ensures that when the SRB should be inserted into the *Bounded Buffer* simultaneously (the buffer is accessed by increasing sequence numbers) already used slots will not be overridden. The bounded buffer is necessary for rapid access to pending SRBs when an acknowledgement comes in (the queue does not and must not provide random access) and since the buffer does not possess further access control facilities this might cause loss of data. It is just the *Receive Thread* which is responsible for the access to pending SRBs and their update (according to the received data). Finally, completed SRBs have to be disposed sequentially by the *Dispatch Thread*; that is, only the head of the *Sent Queue* is examined. By peeking the queue's first element (the thread is blocked if the queue is empty) the current sequence number is determined. This sequence number defines the slot in the bounded buffer where the actual state information is stored. A state value of `qssGarbage` indicates that signalling and freeing of memory structures have occurred. This behavior induces the existence of potential head-of-queue issues during the process of disposal, however, due to the fixed maximum number of pending SRBs this behavior is explicitly desired. Nevertheless, disposal is independent from signaling (which happens in the *Receiver Thread*), thus from the client's perspective all ASPI tasks are still completed asynchronously (as indicated by the target).

For an asynchronously working target different considerations have to be made. As the initiator may employ more than just one *Transmit Thread*, theoretically consecutively enqueued SRBs might arrive at the target out of their order. Further, the target also is not restricted in its number of *Receive Threads*. To ensure correct re-ordering of the arriving SRBs the target implements a priority queue (`m_pqSrbQueue` of type `CSyncPrioQueue<CPrio-QueuedSrb *>`). All incoming SRBs are enqueued while the element with the highest priority, that is the earliest SRB, resides on top of the heap. The priority heap does not deal with the omission of sequence numbers, thus the target additionally maintains a sequence number for the next command pending to be issued. As long as the next command is not received (the top element of the heap does not carry the pending sequence number) all *Worker Threads* are blocked. The blocking is realized by a signalling mechanism which reacts to the arrival of new elements. As previously hinted multiple *Worker Threads* may exist while the number of threads determines the amount of concurrently executing ASPI commands. Each thread processes an ASPI command and is responsible for its further observation; that is, returning the return result back to the initiator, waiting for the execution to be completed, and the eventual acknowledgement of the command (see figure 3.14). Subsequent to this process the thread may assume responsibility for a new ASPI command.

Keeping advanced functionality like Multi-Path I/O in mind, the existence of multiple *Transmit* and *Receive Threads* matches exquisitely to the concept of multiple communication paths. Here, each thread is associated with a previously established path: upon creation each threads receives a pointer to its associated path which the thread in turn uses as ICI (Interface Control Information) for the function call when dispatching a tunnel layer packet. The information is passed by each layer until the nexus finally delivers the session layer packet to the responsible path object. Parts of this behavior (see *Session.h*, `CMultiPathSession::Send(PCBUFFER pData, int nLength, SESSION_PCI pPci)`) are already implemented yet redirecting all requests to the (single) primary path of a multi path nexus. In summary, for the initiator this means for each path there must exist a transmitting and a receiving thread ($2n + 1$, whereas n is the number of paths). The target has to employ a receiving thread for each utilized path and an arbitrary number of worker threads ($n + m$, whereas m is the number of concurrent ASPI tasks). For a single path nexus these requirements are optional; that is, the target may keep the 5 threads especially for use with SMP (Symmetric Multi-Processing) systems.

The existing code base is written in C++ for compilation with the Microsoft product line of C++ compilers. Initial development was started with the MS Visual C++ 6.0 (SP6) IDE and therefore should be still compatible with this environment. However, development was shifted to MS Visual Studio .NET 2003 and finally MS Visual Studio 2005 during the overall process. This is why the focus did not remain on backwards compatibility to VC6 and it might be necessary to modify some portions of the code to recover compatibility. Further, the presented code should be considered experimental with the primary objective to demonstrate feasibility of the given specifications, yet an application among end-user systems nearly might be imaginable. Especially multi-threaded programming is error-prone and several bugs are likely to be eliminated yet.

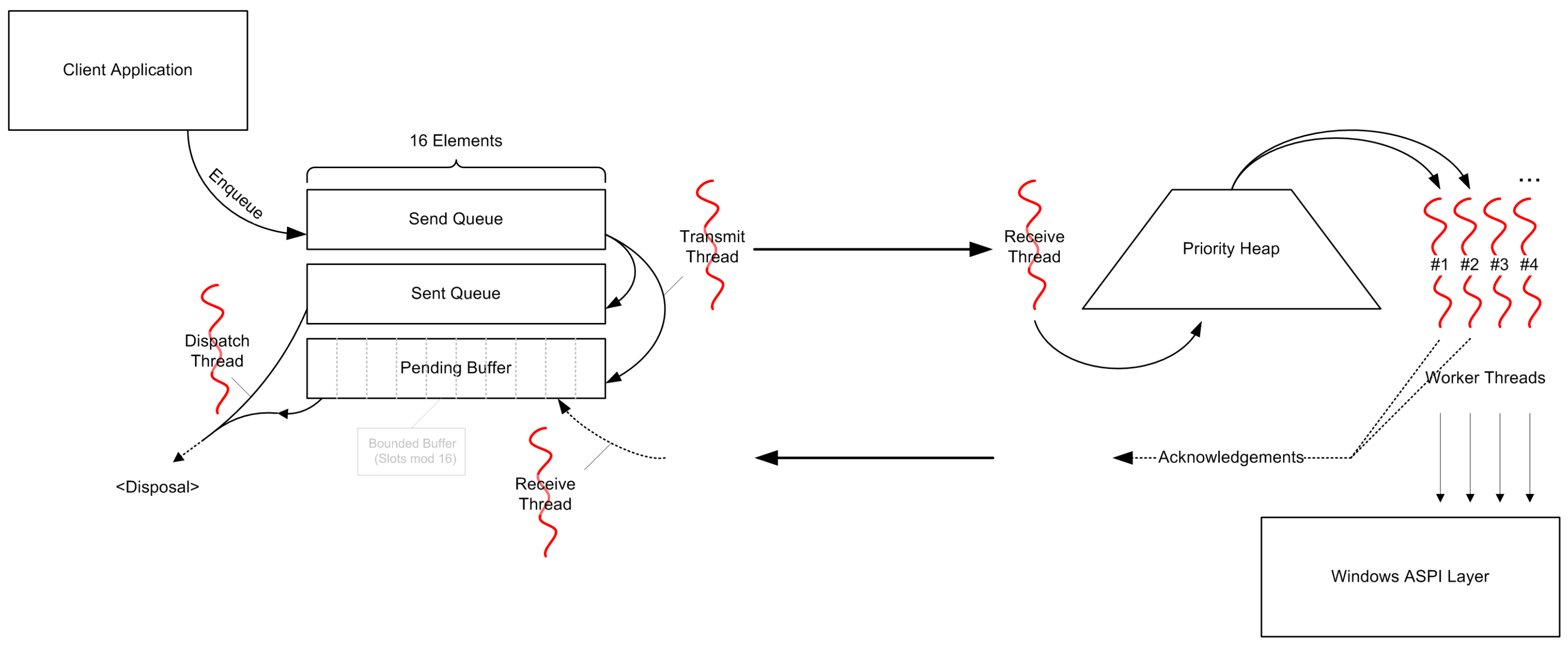


Figure 3.14: Asynchronous Implementation: Comprehensive Design Diagram

Chapter 4

Evaluation

4.1 Synchronous Implementation

The initial synchronous implementation has not been extensively analyzed during development. While these implementations already worked (all important types of SRBs were transmitted correctly), they caused high CPU loads due to the utilization of polling (busy waiting) for command completion. Debugging was conducted with a client and a server application (*SockClient* and *SockServer*) acting as simple initiator and target. Here, SockClient comprises test code which performs a device discovery resulting in a listing of devices found on the remote system (see figure 4.1). On the other hand, *SockServer* realizes a simple target allowing the connection to a single initiator. Incoming SRBs are forwarded to the local Windows ASPI layer for execution. Later versions of SockServer implemented Event Notification which has a diminishing effect on the excessive usage of resources. To allow further testing the current code base was incorporated into a new project which should yield a Windows ASPI layer (Wnaspi32.dll) replacement library. However, this implementation remained synchronous. The results indicated relatively moderate performance, yet the first endeavor of erasing and burning a CD-RW medium over network connection succeeded. Further, *Nero 6 Reloaded* performed as ASPI client application.

Despite working properly, throughout this thesis performance will be benchmarked by regarding the absence of Buffer Underruns (BURN) at different data rates (i.e., recording speed). Variable buffer fill levels are strictly considered unacceptable, even though todays recording devices are able to avoid corruption of media during these underrun situations. However, it is not desirable to tolerate BURNs, as a frequent buffer refill generally prolongs the burning process (with rising BURN count elapsing time increases exceedingly).

```
Date: Wed Dec 14 21:58:52 2005
Target Name: chief
Target Port: 8181
Target Alias: alias
Host adapter inquiry succeeded.

Manager: <NERO ASPI32>
Identifier: <atapi>
Device: NEC CD-ROM DRIVE:463
Device: MEGARAID  LD 0 RAID 0
```

Figure 4.1: Console Output: Device Listing (SockClient)

Hardware Configuration
Pentium 4 1.6 GHz
256 MB RIMM
100 MBit/s Realtek NIC
Pentium III 600 MHz
320 MB DIMM
Intel PRO/100 NIC
O/S: Windows XP
Cross-Connected CAT5

Experimental Results		
Rec. Speed	Data Rate	Device Buffer
4x: 600 KiB/s	1.788 MiB/s	80%
8x: 1200 KiB/s	1.311-2.980 MiB/s	< 60%
10x: 1500 KiB/s	peak 4.410 MiB/s	not stable

Table 4.1: First Performance Results

Figure 4.1 shows the impact of the initial implementation's triple payload transmission. Thus, the collected measurements range between two and three times the actual recording speed. Stable results (relatively constant buffer fill level of 80%) were only observed with 4x (600 KiB/s) recording speed, although an optimization has been deployed causing the omission of `CMD_RETURN` messages (the initiator waits until the target sends the final command completion acknowledgement). Speeds of 8x or even 10x yielded no stable results, hence the imprecise data rates.

4.2 Asynchronous Implementation

The SockClient source code could virtually remain unmodified, since the ATP interface for initiators (with the usual ASPI interface function set of `SendASPI32Command()` and `GetASPI32SupportInfo()`) was not touched. However, the actual implementation of the ATP protocol stack changed dramatically (see section 3.5). Also new code dispatching multiple SRBs in parallel to test the new implementation's capabilities was introduced. Here it is important that the initiator side of protocol correctly signals completion with event notifications. Further, various problems with thread synchronization occurred and compatibility to Nero had to be reconstituted. The *SocketServer*, on the other hand, was now able to accept multiple client connections accomplished by a dedicated server socket *Accept Thread*. Each ATP tunnel also employs multiple *Receiver* and *Worker Threads* again permitting asynchronous operation at this level. For evaluation purposes the target's `DefaultSrbHandler()` dumps the current state of the internal SRB queue as shown in figure 4.2.

```
[106038585] Queue Size 1, Async SRBs pending 0
```

Figure 4.2: Console Output: Queue Size (SockServer)

Having *SockClient* and *SockServer* approved to work correctly with the new implementation, it is of great interest whether the ASPI wrapper (Wnaspi32.dll) can also be suited to the new code and more importantly the yields over the former realization that might be expected. Unfortunately, the status printed by *SockServer* (figure 4.2) while operating together with *Nero Burning Rom* as initiator showed a low utilization of available queue capacity. To be more accurate, the maximum queue size was a single pending element (as shown in figure 4.2). This means, albeit Nero is programmed as a multi-threaded application, the main thread responsible for dispatching commands to the ASPI layer does not fully use its asynchronous feature set. Thus, Nero does also not make use of the queued services provided by ATP-MT (the multi-threaded implementation of the ASPI Tunnelling Protocol). That means even during a burning process always only a data block of 64 KiB is in flight. Nevertheless, eventually data is buffered by the target's recording device providing reliability for the overall process.

To achieve the prescribed objective of actual user-friendliness and easy deployment, another project is included with the solution. *AtpTarget* is a Windows Service which comprises the ATP implementation and additional infrastructure for service management (e.g., command line options to install and uninstall the service on a Windows machine; sufficient administrative rights presumed). A Windows Service can be run with SYSTEM account permissions without needing a user to be logged onto the machine. In the following section this *ATP Target Service* is consulted for all further measurements (see Appendix D).

4.3 Experimental Results

The asynchronous implementation of ASPI Tunnelling Protocol initiator and target has been thoroughly tested in the following environment (see fig. 4.2). As previously hinted, for the target role a properly installed *AtpTarget* Windows Service in conjunction with the Nero ASPI Layer (version 2.0.1.74) has been employed. *Nero Burning Rom 6 Reloaded* (version 6.6.0.18) acted as client application for the latest Wnaspi32.dll ATP initiator.

The environment comprised two dedicated target and initiator machines (hardware configuration see below) running a copy of Windows Server 2003 SP1 and Windows XP SP2, respectively.

Target	Initiator	Network
AMD Duron 700 MHz 512 MB Intel GbE PRO/1000MT Plextor PX-W4012A	AMD Athlon 1333 MHz 512 MB Intel GbE PRO/1000MT	Gigabit Ethernet Switch 16 Gbps Backplane 1.4881 Mpps per Port
Windows Server 2003 EE Service Pack 1, Build 3790	Windows XP Service Pack 2	

Table 4.2: Hardware Setup

To determine network performance a network latency measurement has been conducted by using the `traceroute` utility. Traceroute can be instructed to either use ICMP `ECHO` or UDP packets for operation and with the `-q` option a certain number of packets can be sent. With the determination of average round-trip time (RTTs) network performance can be estimated. For network layer `ECHO` packets (38 bytes) the average RTT is 0.171 ms (171 μs) and for 128 byte transport layer UDP packets the average delay is 0.365 ms (365 μs). To be able to comprehend these values the console output is listed in figure 4.3.

```
$ traceroute stef.jl.pa23.net -I -q 20
traceroute to stef.jl.pa23.net (192.168.8.3), 30 hops max,
38 byte packets
1  stef.jl.pa23.net (192.168.8.3)
0.386 ms  0.269 ms  0.153 ms  0.152 ms  0.179 ms  0.170 ms  0.164 ms
0.151 ms  0.152 ms  0.152 ms  0.148 ms  0.150 ms  0.151 ms  0.152 ms
0.145 ms  0.150 ms  0.149 ms  0.147 ms  0.147 ms  0.150 ms
```

Figure 4.3: Latency Measurements

With this configuration the highest possible (yet stable) speed could be ascertained to 2,400 KiB/s. Since the applied hardware environment can be regarded as average equipment, generally, viable burning speed with the current ATP implementation was ultimately limited to 16x CD.

To further analyze the performance of the implementation in conjunction with the hardware environment *Ethereal* as Network Protocol Analyzer has been deployed. Prior to use possible TCP Offload Engines (TOEs) should be deactivated through the driver software, as this causes the analyzer software to mark most frames (Ethernet and TCP segments) due to invalid CRC values. These calculations are offloaded from the TCP stack and not made until the NIC dispatches the Ethernet frame. The analyzer running on the initiator machine captured an excerpt from a ATP session being busy of recording a CD. With the recorded dump the exact timing of the transmitted Ethernet frames and the corresponding TCP segments could be determined. Here, communication showed large delays in message response that were caused unlikely by the processing on the target computer. The conclusion should be that the receiver is not able to process the data immediately. This adverse timing just coincided with the TCP segments which were not marked with the TCP `PUSH` flag. The `PUSH` flag is located in the TCP header of each segment and instructs the remote TCP stack to immediately pass the incoming data to the corresponding application process. However, the occasional absence of `PUSH` flags should be traced back to Nagle's Algorithm (1984) [9] described in RFC 896. At that time, grave problems arose due to limited bandwidth and the excessive transmission of small packets (for example caused by usage of `telnet`) causing congestion collapse. Nagle's proposal was to retain small packets on the sender's side if acknowledgements are still outstanding. This behavior matches well with the previously observed timing issues. From a programmer's point of view it is not possible to influence TCP flag generation directly, but the appliance of Nagle's Algorithm (which is deployed in virtually every modern TCP/IP implementation) can be easily deactivated. This is done by applying a `setsockopt()` function call to the corresponding socket handle. Socket API documentation disadvises the usage of the `TCP_NODELAY` socket option with the constraint the option yet might be applicable to code which has to send many small messages. For this implementation the `TCP_NODELAY` option means the emulation of a datagram-like service (implicating small delay) over an established stream-oriented TCP connection. In other words, when sending a large chunk of data or a small message on the application layer, the last TCP segment (carrying the `PSH` flag) is sent *immediately* and thus cannot cause a delay any more. To realize the modified behavior the existing source files can easily be patched, whereas in *Nexus.cpp* the member function `CSinglePathNexus::AssignEndpoint(CGenericEndpoint *pEndpoint)` is located allowing the change to be applied to the initiator, target, and even all involved path endpoints. The additional code is shown in figure 4.4. Just in proximity to this modification the final code of Nexus.cpp also adjusts the TCP receive and send windows (`SO_RCVBUF` and `SO_SNDBUF` options) to further optimize the stack behavior. These new options, however, are not of transport layer level but of socket level (modifying the socket operation).

When reviewing the Linux iSCSI implementation (hosted on SourceForge.net) it is not long until stumbling upon the code section where the `TCP_NODELAY` option is activated there (*linux-iscsi/daemon/iscsi-io.c*, line 79).

```
int flag = TRUE;

CTransportEndpoint *pTranspEndp =
        dynamic_cast<CTransportEndpoint *>(pEndpoint);

_ASSERTE(pTranspEndp);
BOOL bSuccess = pTranspEndp->SetSockOpt(
                TCP_NODELAY, &flag, sizeof(flag), IPPROTO_TCP
        );
```

Figure 4.4: Application of the TCP_NODELAY Option

In other words, the `TCP_NODELAY` option is an indispensable prerequisite for communication protocols like iSCSI and ATP (as of equal nature). A short Ethereal dump of this *datagram* communication between `TCP_NODELAY` initiator and target on the data link layer (Ethernet) is shown in Appendix C. In addition to this dump Appendix C also includes a listing of the ATP connection establishment phase on transport level (hex dump of conversation).

With the new code the former issue is resolved and in the given configuration ATP now is able to operate with data rates up to 40x CD speed and yet no indication for symptoms of too high demand. Further, this performance seems to be relatively insensible against additional traffic (i.e., a file copy process on a network share), since switched networks should be ideally collision-free (no retransmits). Indeed, data rates of 6,000 KiB/s are quite near the theoretically possible bandwidth of Fast Ethernet (100 MBit/s) networks.

The diagram in figure 4.5 shows the reached data rates and the user time CPU utilization (pure user level overhead) of the ATP target during a 40x CD (6,000 KiB/s) speed test run. The attached Plextor recording device practices ZoneCLV with distinct zones of constant burning speed (the drive's motor is adjusted to maintain speed while otherwise the actual write speed would continuously increase), which can be easily observed in the diagram. The user time (in percent) alone is utilized by the *AtpTarget* process on the machine. With increasing write speed the moderate CPU utilization is slightly rising. When only accounting active periods the average user time is about 6%. This can also be put into relation with the average data rate of the corresponding zone which results in a value of 1.50% user time CPU utilization per processed MiB/s. While being acceptable for average computers, certainly, this amount of overhead can still be optimized.

With the knowledge about feasibility of various burning speeds in the given environment the observed behavior can be mathematically analyzed. The following formulas will calculate bandwidth and latency requirements

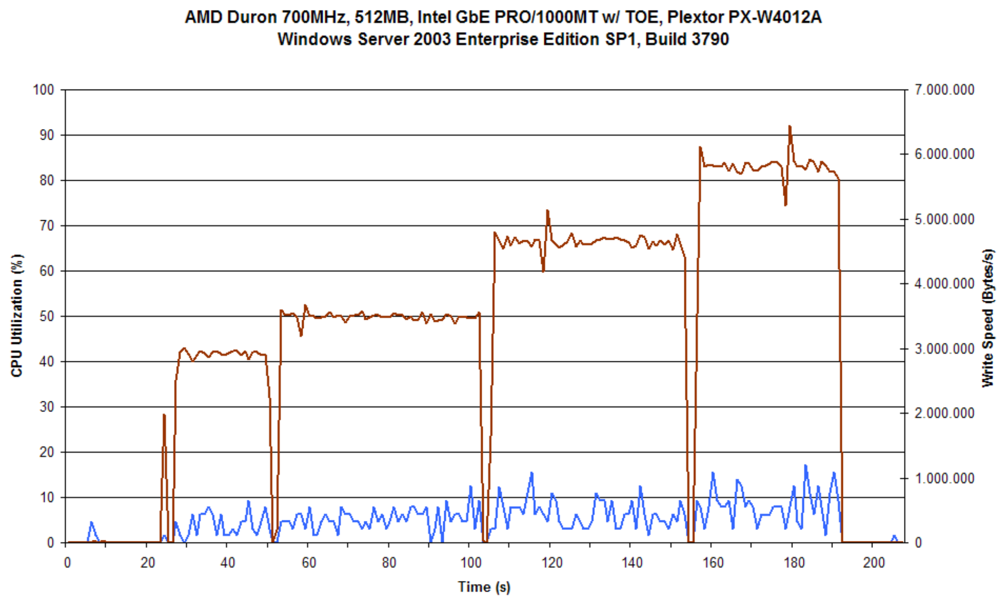


Figure 4.5: AtpTarget Process Analysis: CPU Utilization and Data Rate at 40x ZoneCLV

for various scenarios and thus can give a hint whether the deployment of the ATP protocol at certain data rates seems possible. However, the results are not claimed to be applicable in every situation. For example, positivlely rated parameters must no imply flawless operation in the actual environment. An important parameter for the calculations is the recording speed v (in times the single CD speed of 150 KiB/s) and the available network bandwidth B (in Mbit/s) together with the network latency t_{lat}. The model is based on the assumption of transmission of whole data blocks of 64 KiB. Since the basic unit for a recording device is a sector of 2048 bytes, a block comprises 32 real sectors. As defined by the CD standard (for data CDs) single CD speed corresponds to 75 sectors per second. Hence, the number of written blocks per second (at a given recording speed) is approximated by formula (4.1). Further, (4.2) calculates the expected time required to write such a block to the medium. The same applies to (4.4), but with the actually available network infrastructure accounted, as (4.3) evaluates to the number of blocks that can be transferred over the medium. Finally, (4.5) calculates the overall time needed for transmission which now allows the verification of the resultant values against one another.

First, the desired recording speed has to be checked against the available network bandwidth (constraint **A** (4.6), an actual network efficiency of 70 percent is assumed). If this condition is met the overall timing can be verified with constraint **B** (4.7), which demands an excess of 20 percent for the calculated timing requirements.

v = recording speed [x CD]
B = network bandwidth [MBit/s]
t_{lat} = network latency [ms]

1x CD = 150 KiB/s
1 Block = 64 KiB

Formulae

$$a = \left\lceil \frac{75}{32 \cdot v} \right\rceil \tag{4.1}$$

$$t_{req} = \frac{1}{a \cdot 1000} \tag{4.2}$$

$$b = \left\lfloor \frac{B \cdot 1000 \cdot 1000}{8 \cdot 64 \cdot 1024} \right\rfloor \tag{4.3}$$

$$t_{trans} = \frac{1}{b \cdot 1000} \tag{4.4}$$

$$t = 2 \cdot t_{trans} + t_{lat} \tag{4.5}$$

Constraints

$$B > \frac{v \cdot 150 \cdot 1024}{70\% \cdot 125000} \tag{4.6}$$

$$t < 80\% \cdot t_{req} \tag{4.7}$$

For the given experimental setup ($v = 40$, $B = 1000$, $t_{lat} = 1$) the result is an overall block transmission time of $t = 2.52$ milliseconds. Apart from the fact that the Gigabit Ethernet definitively is capable of sustaining the assigned data rate, this is well below the required timing of $t_{req} = 10.64$ ms (which corresponds to 94 Blocks per second). Even with a Fast Ethernet network as interconnect ($B = 100$) formula (4.5) returns a feasible value of

$t = 7.26$ ms.

Of different concern is the implicated CPU utilization during operation on the target (and also the initiator). With the previously determined value of 1.50 percent utilization per MiB/s data rate the impact can be extrapolated for other CPU speeds (same architecture and linear scaling assumed).

$$U = 1.5 \cdot \frac{v \cdot 150}{1024} \cdot \frac{700}{f} \tag{4.8}$$

This represents a very imprecise estimation where U means the resultant utilization and f the actual processor speed in megahertz. User time utilization also implies consumption of privileged time (geater or equal, at least). Hence, 50 percent CPU utilization should be considered as saturation for the processor. Nevertheless, while being potentially inaccurate the estimation might give a hint to minimum CPU requirements at a defined recording speed v. For the present 40x (6,000 KiB/s) the lower limit seems to be reached at some 133 MHz.

Similar tests have been conducted with the same target but a different initiator machine (Intel Pentium III 600 MHz, 320 MB RAM, 100 MBit NIC). By experiment, a maximum recording speed of 24x was identified. According to the previous calculations (p. 52) the timing requirements are met, but continuous buffer underrun conditions occurred. Due to the observation of relatively high CPU utilization during the test it seems likely to be the reason of failure. Probably additional aspects remained unconsidered in the presented set of formulas.

Finally, it might be interesting to analyze the limits of the solution of this thesis by exchanging the underlying physical network with a wireless technology. Here, latency is the predominant limitation as air is a shared medium which has to be arbitrated. Unfortunately, for testing only 802.11b WLAN equipment was available, which is not even capable of providing the required (transport layer) data rate for the lowest possible recording speed of 4x (600 KiB/s). In a real world scenario the WLAN link (11 MBit/s) yields a data rate of approximately 350 KiB/s, a fact that inevitably leads to buffer underruns when serving as network infrastructure for an ATP tunnel. However, despite suffering inherently from latency issues, a Wireless LAN with adequate bandwidth seems to be capable, as buffer underruns were moderate during the tests. Experiments generally acknowledging feasibility of *ATP over WLAN* might be of further interest. Even tests over a WAN (Wide Area Network) connection were successful. While the process of erasing a rewritable medium (CD-RW) only produces few and infrequent commands, this experiment confirmed the possibility to even use an Internet connection with the ATP protocol.

Chapter 5

Analysis

The following chapter will analyze the collected results and discuss various other aspects concerning the so far presented solution of this thesis. Apart from the many advantages and downsides associated with the given solution, it may be referred to as correct. In addition to the careful design process the actual proof of correctness is done by the observation of overall correct operation in normal and experimental application.

However, correctness is not sufficient in real-world scenarios as for the end-user the primary concern is reliable operation and performance of the solution. Depending on the given network infrastructure this objective is accomplished, viz. data rate and latency requirements at the application level are met. Last but not least, this is significantly facilitated by the usage of the `TCP_NODELAY` option.

5.1 ASPI over IP

As demonstrated up to this point, with the developed solution it is possible to tunnel application-level ASPI SRBs over a network interconnect. The ASPI Tunnelling Protocol (ATP) has been designed just for this purpose and realizes all specified requirements, as it actually facilitates the tunnelling of ASPI commands. Tasks like maintaining *distributed memory coherency* (chapter 1) and many others being associated with the primary objective require careful design (with several accompanying decisions of importance) and a rather complex framework of supporting functionality.

Beginning with a layered solution paradigm and after having previously studied SCSI, ASPI, and network fundamentals the design was being elaborated by a bottom-up approach. The protocol design included the definition of an appropriate state machine and was followed by the formulation of a text-phase and finally a full-feature phase protocol. This part of the protocol demanded the definition of a session layer and a tunnel layer packet format. With the *Implementation Design* part and in particular the more detailed sections *Synchronous Implementation* and *Asynchronous Implementation* the actually realized design decisions were outlined (Appendix B.1 includes a detailed diagram of the layered overall architecture, p. 65). The subsequent chapter delved into the experimental analysis of the developed *ASPI Tunnelling Protocol* and shows the limitations of the solution imposed by external conditions. Nevertheless, the experimental analysis facilitated the revelation of a software-related issue which could be identified and coped with.

However, with the experiences gained over the course of this thesis it has to be noticed that even while ASPI provides the functionality to leverage multi-threaded operation, it is barely utilized by client applications (as learned with the example of Nero). From a system architectural point of view the multi-threaded design of an appropriate solution seems to be worthwhile, yet for future work it might be reasonable to rather focus on a less sophisticated single-threaded solution based on the insights of this thesis and the provided implementation. Since that would be sufficient, at least for the initiator part, inherent sources of bugs in the residual code could be eliminated. Some of the potential possibilities will be elaborated in the subsequent section.

5.2 Advantages/Drawbacks (ATP)

The software architecture for the entire ASPI Tunnelling Protocol stack has been thoroughly designed. The goal of a good design should finally be the realization of a straightforward and elegant implementation, however, it is impossible to prevent a project from growing and thus with the experience gained during implementation, it certainly would have been possible to correct some of the formerly committed programming flaws. Some of the most urging issues will be discussed in the following *Outstanding Issues* section. Multi-Threaded programming and especially debugging is complex, a fact that actually limited the amount of revision the code could obtain in the course of this study thesis. In any case, the result can be regarded as proof-of-concept.

Further, a strictly scientific design process sometimes involves an enlarged or even bloated OOP (Object Oriented Programming) hierarchy. Often the same functionality might be accomplished with procedural programming and fewer resources. A prominent example can be found with the class compound of `CGenericSession`, `CGenericNexus`, and `CGenericPath`. Inheritance is the appropriate procedure to aggregate functionality, however, the same inheritance complicates debugging due to large call stacks and causes overhead at runtime. Certainly the ATP protocol could be redesigned to flatten the class hierarchy with a low-level implementation of a single software layer. Probably two software layers (session and tunnel layer) might not be necessary. Nevertheless, in case of features like Multi-Path I/O (MPIO) the clean software design of the existing class hierarchy still can ease programmability and despite the previous considerations a great deal of work and careful thought went into the layout of particular classes and functions.

Obviously, the advanced features mentioned throughout this work like **Multi-Threading** and **Multi-Path I/O** are aimed at the operation with *Nero Burning Rom* (in increased-demand scenarios). With multi-threading finally realized and a class hierarchy prepared for operation in multi-path mode, it is unfortunate that these features can only be utilized by software which by itself is of multi-threaded nature and prepared to asynchronously wait for multiple commands to complete. The multi-pathing feature readies the ATP protocol for resilience and the benefit of multiple links between initiator and target allowing the utilization of additional bandwidth. Otherwise throughput and latency would be limited by a single network adapter in the single-threaded case. Anyway, systems with multiple cores (together with SMP) seem to become more common in future, even for the average home user and thus multi-threading still remains an interesting aspect for the overall system architecture. In summary, it is hardly possible to anticipate Nero's low requirements in terms of *multi-thread awareness* prior to having realized an actually multi-threaded implementation.

5.3 Outstanding Issues

As previously announced in 5.1 this section will discuss some possible improvements for the existing code base towards an optimized and reliable software solution which can be deployed among a production environment. First, a technique called Scatter/Gather I/O (SGIO) should be applied in the code; that is, a vector of blocks (comprising pointer and the block's length) is passed to a *vectorized version* of a function. SGIO is used in several places of an operating system, however, for this solution the vectorized versions of Socket API functions, namely `sendv()` and `recvv()`, are important. Similar functions are also provided by the WinSock API. While optimizing program efficiency SGIO can also simplify packet assembly as scattered data does not have to be copied into a contiguous chunk of memory. For example, header part, payload data, and packet trailer may reside at different locations which might **(1)** avoid the allocation of memory (`new`) and the **(2)** accompanying copy operations (`memcpy()`).

```
PBUFFER pBuffer = new BUFFER[ulPacketLength];
TUNNEL_HEADER *pHeader = (TUNNEL_HEADER *) pBuffer;

PBUFFER pPayloadBuffer = (PBUFFER) (pHeader + 1);

memcpy(pPayloadBuffer, pcUpperLayerPacketBuffer,
        ulUpperLayerPacketLength);
```

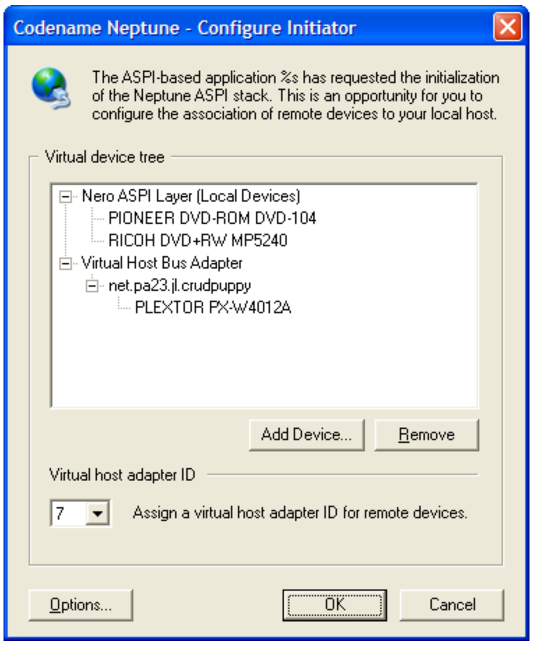
Figure 5.1: Packet Generation Code (*Session.cpp*)

Together with the generation of packet checksums these tasks cause the most processing overhead of the protocol stack. However, the well separated structure of Network Abstraction Layer (NAL) and its adjacent software layers should constrain the effort needed for the integration of a SGIO-enabled interface (i.e., verctorized function signatures). The realization already would decrease required resources, but a methodology for passing an ordered list of packet fragments between protocol layers has to be introduced. Such an abstracted data structure is a common obstacle in layered protocol programming hence several approaches can be found among internal operating system designs. An elegant way to overcome the problem is given with Windows' paradigm of I/O Request Packets (IRPs) applied within the Windows driver stack [8]. The IRP provides a common structure enabling each driver located in the stack to access driver-specific (or layer-specific) information. This design could also be applied to the existing ATP implementation and facilitate (vertical) inter-layer protocol communication. Another important issue in conjunction with internal stack communication is freeing of previously allocated memory which could also be simplified by the use of a data structure equivalent to IRPs. The problem is caused by the asymmetric behavior of the sending and receiving protocol instances. In the former case the memory allocation originates in the highest protocol layer (a `SendPacket()` request) and deallocation has to happen after the PDU has actually been dispatched on the lowest layer. In the contrary case of the opposing protocol instance, the initial packet originates in the lowest protocol layer where memory has to be allocated on packet arrival. The required memory cannot be freed until the PDU has been processed by the entire protocol stack demanding a separate packet processing behavior for the two occurrences of protocol endpoints. Again, a common data structure facilitating internal protocol communication and carrying all important information for the separate protocol layers could ease the task of keeping account of allocated memory blocks (memory leaks).

With today's network hardware (high bandwidth, low latency) it is possible to realize the presented solution of the ATP protocol. Even the serial network technology is able to provide latencies which are adequate for demanding tasks like the transmission of storage subsystem traffic, at least for high level (application layer) protocols like ATP or iSCSI. This traffic cannot be compared to the raw SCSI communication over an interconnect bus or point-to-point connection. Here, with these specialized interconnect technologies the demands to the timing behavior are much more restrictive. The performance gains accomplished by the `TCP_NODELAY` modifications provide some headroom for the increasing data rates reached with the current generation of CD/DVD recorders. However, due to restrictions caused by network or computer hardware with limited resources, it might be necessary to discover new possibilities to decrease latency requirements or further optimizations of the ASPI Tunnelling Protocol. A potential tweak to diminish relevancy of protocol message delay might be the *premature* acknowledgement of pending requests to the client software. This application then may dispatch further `WRITE` commands although the former SRBs are still about to be processed on the target. This, in turn, demands an mechanism to detect unexpected conditions in case of an error during execution. However, the SCSI standard already defines two mechanisms in conjunction with this behavior. It is *explicitly* legal to acknowledge `WRITE` commands with a `GOOD` status while the request is buffered in the drive's cache unless the so called Force Unit Access (FUA) bit is set ([15], p. 131). In this case, subsequent commands are obligated to indicate a Deferred Error Condition ([16], 4.5.5, p. 39). Fortunately, Nero omits the FUA bit to leverage the drives' buffering features, which also supports the possible application of this tweak in future implementations of the ASPI Tunnelling Protocol. In other words, the immediate acknowledgement is likely to provide a mechanism to become virtually independent from latency issues while the single limitation of sufficient bandwidth between initiator and target remains.

Apart from this set of *internal* optimizations some *external* improvements could also be made. The following considerations suggest completion of a purely multi-threaded implementation or an advanced single-threaded incarnation of the protocol initiator. Certainly, the MT version would be more versatile, however, due to the lack of support ST would be sufficient even for the use with Nero Burning Rom. Since the advanced ST solution only comprises a single main program thread it would be rendered almost unable to implement features like Multi-Path I/O. As discussed above, packet generation overhead could be easily diminished by the presented mechanisms, yet the single-threaded initiator should allow the application client to use the Windows event signaling system to further optimize the usage of CPU resources. Anyway, sources for bugs could be eliminated with the less complex single-thread solution, which might be the primary objective for eventual deployment among end user computers. In either case, the target's implementation should remain multi-threaded, as for the server side asynchronous execution might be of foremost concern.

In summary, the core of the presented ATP protocol stack may remain to provide a reliable framework for future software utilizing the ASPI Tunnelling Protocol. Nevertheless, some additional proposals for these implementations should be mentioned. First, the concept of a Virtual Host Adapter (VHA) could be enforced by the protocol initiator. This VHA then could realize an abstraction for the management of multiple sessions to different target computers. Here, a mapping between virtual devices and session has to be established. Since the transmission of SRBs is transparent between initiator and target the common device addresses (HA:ID) are derived from the target's local device map. This means all initiators have to support a device address mapping from virtual addresses to target addresses, while the target controls access to unlocked devices for the corresponding session. Prior to initialization of the local ATP ASPI Layer the user should be able to modify the association of Virtual Devices (VD) (to the VHA) which then can be used in the following Nero session. If configuration changes are not likely to happen frequently, it is also imaginable to provide access to the configuration in form of a Windows Control Panel applet. In figure 5.2 a suggestion for an appropriate UI is visualized.


Codename Neptune - Configure Initiator

The ASPI-based application %s has requested the initialization of the Neptune ASPI stack. This is an opportunity for you to configure the association of remote devices to your local host.

Virtual device tree

- Nero ASPI Layer (Local Devices)
  - PIONEER DVD-ROM DVD-104
  - RICOH DVD+RW MP5240
- Virtual Host Bus Adapter
  - net.pa23.jl.crudpuppy
    - PLEXTOR PX-W4012A

Virtual host adapter ID: 7

Assign a virtual host adapter ID for remote devices.

Figure 5.2: Association Dialog UI Proposal

Chapter 6

Conclusion

To conclude the work of this study thesis presented in the course of the previous five chapters, the following paragraphs will finally give a repetitive overview. The impetus for the given topic, ASPI over IP, was the observation of an apparent lack of support for optical devices as rather important form of mass storage in the case of already existing solutions facilitating separation of storage devices from the responsible host system. It is obvious that this concept in general is applicable to various scenarios and thus has been subject of recent research and commercial interest. A fact also outlined by the sheer number of different standards (amongst others, there are for example the most important occurrences of Fibre Channel and its sibling iSCSI), which were considered throughout the first part of this thesis.

The existing technologies allow the transparent transmission of SCSI command traffic across almost any type of interconnect to achieve the objective of a distributed storage network and independence from local computer buses at the same time. By choosing this generic approach, namely *tunnelling SCSI over serial network transports*, a completely new level of storage virtualization is achieved. However, this type of virtualization has not yet been carried to the domain of optical storage, its peculiar requirements, and the respective application software (at least in case of the Windows family of operating systems). Hence, an appropriate and equivalent solution (that means taking a very similar approach to existing solutions) for the ASPI interface, which represents an important standard for this type of application software, should be designed. While several methodologies and notions could be adopted from the standards earlier studied, a new protocol with text-based and full-feature phase had to be developed to facilitate the semantically correct transmission of SRBs. Having a functional ASPI tunnelling protocol stack at hand, in the subsequent course of design two distinct realizations (*single-threaded/synchronous* and *multi-threaded/asynchronous*) were identified and also elaborated. An actual implementation in software finally demonstrated the correctness of the presented solution also in its multi-threaded incarnation (initiator and target). Subsequently, various tests with the final code base have been conducted in real-world scenarios (target computer equipped with recording device and initiator running *Nero Burning Rom* as client application). By identifying software-related issues obviously impeding network message timing, it became possible to further improve overall performance up to a level, which may well satisfy the requirements for most optical storage recording tasks.

Despite its preliminary status, the code base demonstrates feasibility of the approach and proper operation of the ATP communication protocol. At this point, together with some of the clues for further software design improvements and the outlook to user interface optimizations identified throughout the final part of this work, the conclusion of a viable solution to the objective of this thesis can be drawn.

Appendix A

Class Hierarchy

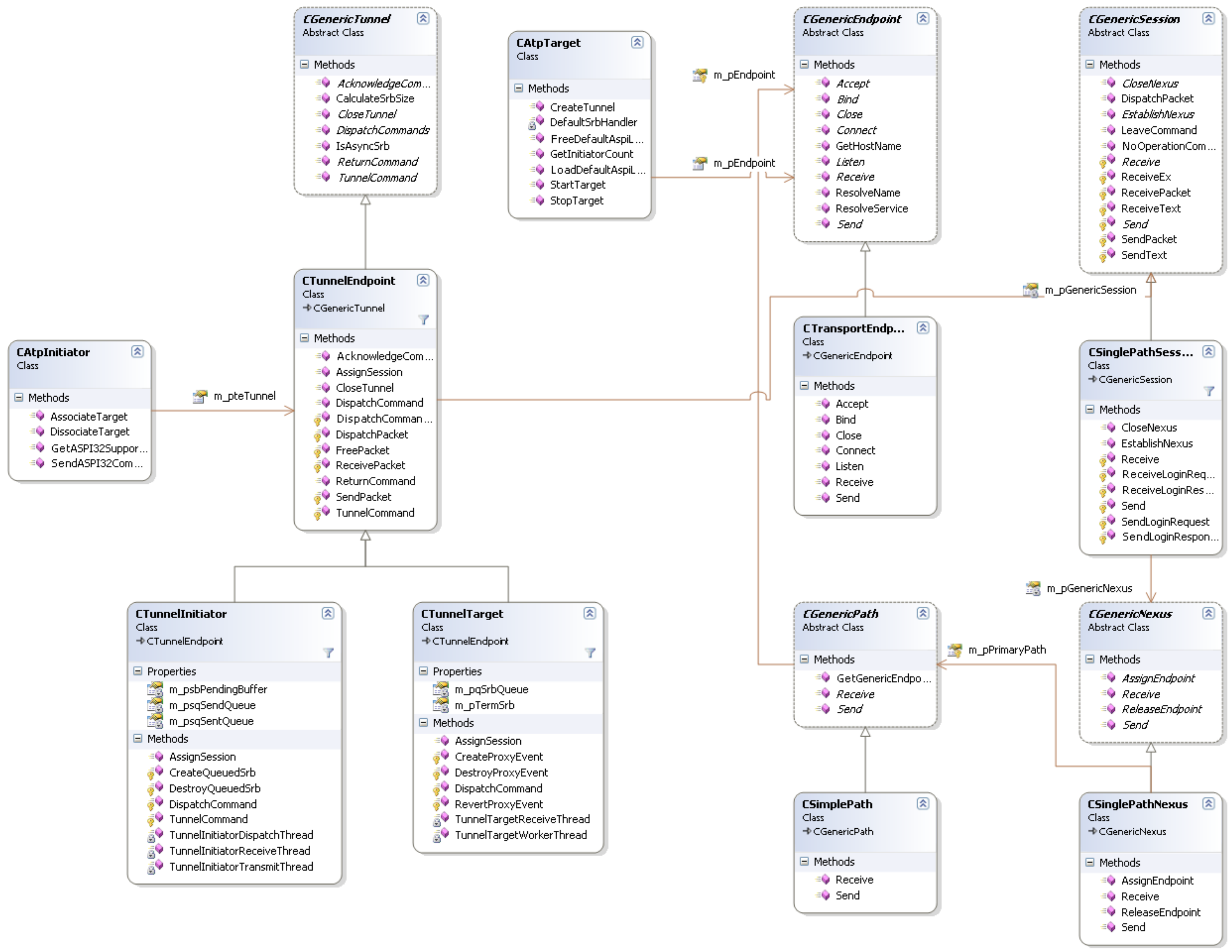


Figure A.1: Comprehensive UML Diagram of the ATP Class Hierarchy

Appendix B

Layered Architecture

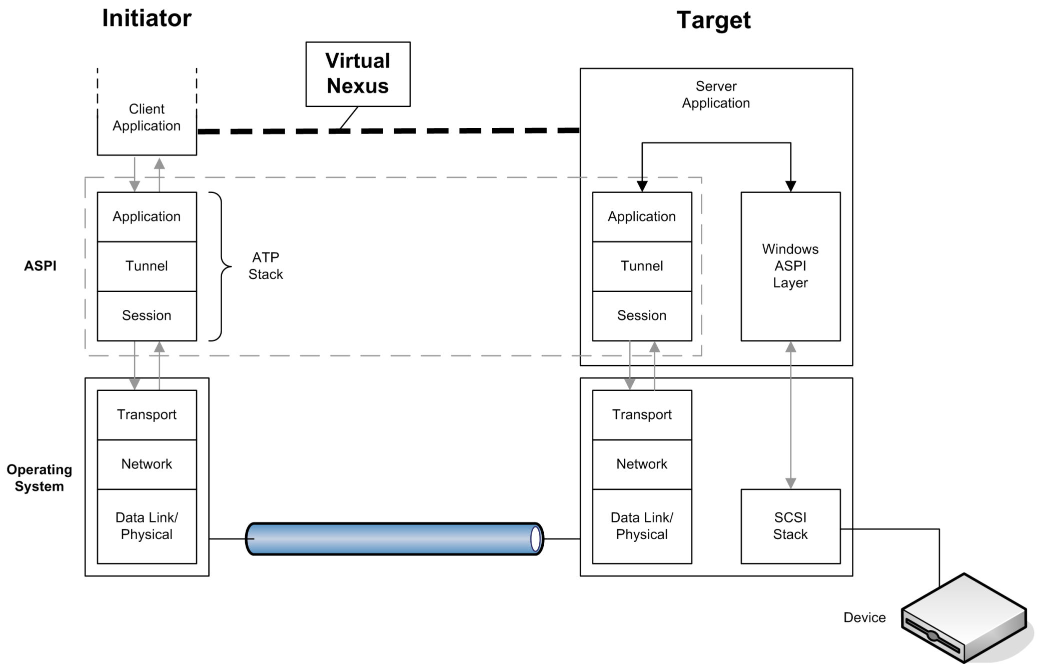


Figure B.1: Detailed Diagram of the layered ATP Architecture

Appendix C

ASPI Tunnelling Protocol

```
 1 0.000000 192.168.8.11 192.168.8.3  2227>8181 [SYN] Seq=0 Ack=0 Win=65535 Len=0 MSS=1460
 2 0.000113 192.168.8.3  192.168.8.11 8181>2227 [SYN, ACK] Seq=0 Ack=1 Win=16384 Len=0 MSS=1460
 3 0.000034 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=1 Ack=1 Win=65535 Len=0
 4 0.000198 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1 Ack=1 Win=65535 Len=21
 5 0.000467 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=1 Ack=22 Win=65514 Len=21
 6 0.191807 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=22 Ack=22 Win=65514 Len=0
 7 0.128133 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=22 Ack=22 Win=65514 Len=128
 8 0.000570 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=22 Ack=150 Win=65386 Len=128
 9 0.011079 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=150 Ack=150 Win=65386 Len=128
10 0.000435 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=150 Ack=278 Win=65258 Len=128
11 0.013139 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=278 Ack=278 Win=65258 Len=128
12 0.000287 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=278 Ack=406 Win=65130 Len=128
13 0.027535 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=406 Ack=406 Win=65130 Len=192
14 0.169253 192.168.8.3  192.168.8.11 8181>2227 [ACK] Seq=406 Ack=598 Win=64938 Len=0
15 0.000090 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=598 Ack=406 Win=65130 Len=128
16 0.000452 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=406 Ack=726 Win=64810 Len=192
17 0.149751 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=726 Ack=598 Win=64938 Len=0
18 0.000260 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=598 Ack=726 Win=64810 Len=448
19 0.008737 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=726 Ack=1046 Win=64490 Len=192
20 0.168798 192.168.8.3  192.168.8.11 8181>2227 [ACK] Seq=1046 Ack=918 Win=64618 Len=0
21 0.000095 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=918 Ack=1046 Win=64490 Len=128
22 0.000438 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=1046 Ack=1046 Win=64490 Len=192
23 0.122146 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=1046 Ack=1238 Win=64298 Len=0
24 0.000247 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=1238 Ack=1046 Win=64490 Len=448
25 0.002900 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1046 Ack=1686 Win=65535 Len=192
26 0.202279 192.168.8.3  192.168.8.11 8181>2227 [ACK] Seq=1686 Ack=1238 Win=64298 Len=0
27 0.000085 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1238 Ack=1686 Win=65535 Len=128
28 0.000446 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=1686 Ack=1366 Win=64170 Len=192
29 0.194568 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=1366 Ack=1878 Win=65343 Len=0
30 0.000261 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=1878 Ack=1366 Win=64170 Len=448
31 0.003057 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1366 Ack=2326 Win=64895 Len=192
32 0.129667 192.168.8.3  192.168.8.11 8181>2227 [ACK] Seq=2326 Ack=1558 Win=65535 Len=0
33 0.000095 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1558 Ack=2326 Win=64895 Len=128
34 0.000419 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=2326 Ack=1686 Win=65407 Len=192
35 0.166930 192.168.8.11 192.168.8.3  2227>8181 [ACK] Seq=1686 Ack=2518 Win=64703 Len=0
36 0.000265 192.168.8.3  192.168.8.11 8181>2227 [PSH, ACK] Seq=2518 Ack=1686 Win=65407 Len=448
37 0.002378 192.168.8.11 192.168.8.3  2227>8181 [PSH, ACK] Seq=1686 Ack=2966 Win=64255 Len=192
38 0.158045 192.168.8.3  192.168.8.11 8181>2227 [ACK] Seq=2966 Ack=1878 Win=65215 Len=0
```

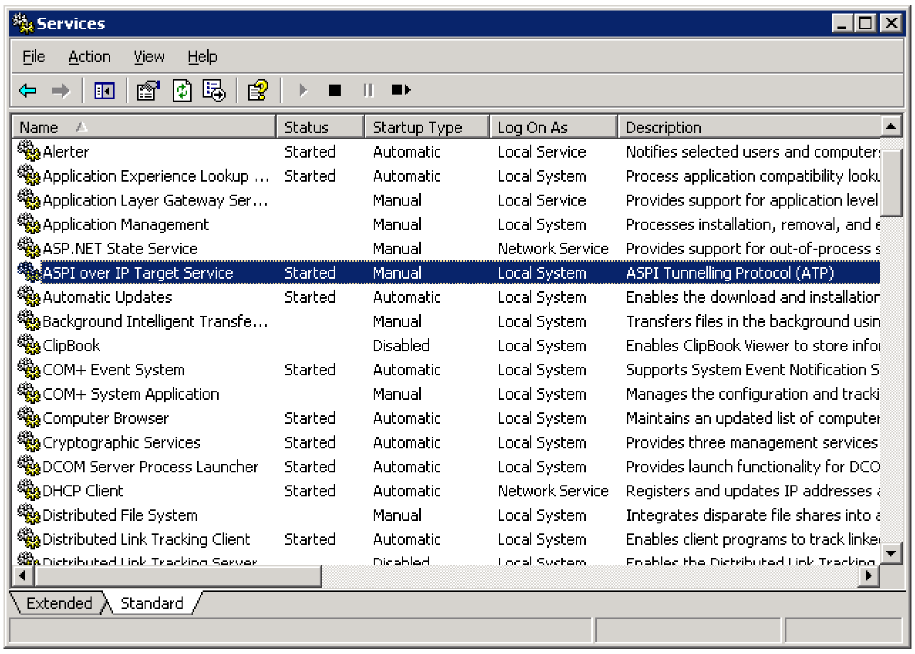
Figure C.1: Sample ATP Conversation (Ethereal Dump)

```
> 00000000  50 72 69 6d 69 74 69 76  65 3a 20 4c 6f 67 69 6e Primitiv e: Login
> 00000010  52 65 71 0a 0a                                   Req..

< 00000000  50 72 69 6d 69 74 69 76  65 3a 20 4c 6f 67 69 6e Primitiv e: Login
< 00000010  52 65 73 0a 0a                                   Res..

> 00000015  51 51 51 51 80 00 00 00  64 00 00 00 01 01 00 01 QQQQ.... d.......
> 00000025  e8 03 00 00 3c 00 00 00  08 00 00 00 00 00 00 00 ....<... ........
> 00000035  00 00 01 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 00000045  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 00000055  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 00000065  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 00000075  30 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00 0....... ........
> 00000085  00 00 00 00 00 00 00 00  00 00 00 00 2b 02 00 00 ........ ....+...

< 00000015  51 51 51 51 80 00 00 00  65 00 00 00 01 02 00 01 QQQQ.... e.......
< 00000025  e8 03 00 00 3c 00 00 00  08 00 00 00 00 00 00 00 ....<... ........
< 00000035  00 01 01 00 00 00 00 00  03 ff 4e 45 52 4f 20 41 ........ ..NERO A
< 00000045  53 50 49 33 32 00 00 00  00 00 61 74 61 70 69 00 SPI32... ..atapi.
< 00000055  00 00 00 00 00 00 00 00  00 00 00 00 02 08 00 00 ........ ........
< 00000065  02 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
< 00000075  c8 03 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
< 00000085  00 00 00 00 00 00 00 00  00 00 00 00 2d 02 00 00 ........ ....-...

> 00000095  51 51 51 51 80 00 00 00  66 00 00 00 01 01 00 01 QQQQ.... f.......
> 000000A5  e9 03 00 00 3c 00 00 00  08 00 00 00 00 00 00 00 ....<... ........
> 000000B5  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 000000C5  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 000000D5  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 000000E5  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00 ........ ........
> 000000F5  30 01 00 00 00 00 00 00  00 00 00 00 00 00 00 00 0....... ........
> 00000105  00 00 00 00 00 00 00 00  00 00 00 00 2d 02 00 00 ........ ....-...
```

Figure C.2: Sample ATP Conversation (Transport Layer)

Appendix D

ATP Target Service



Name	Status	Startup Type	Log On As	Description
Alerter	Started	Automatic	Local Service	Notifies selected users and computer
Application Experience Lookup ...	Started	Automatic	Local System	Process application compatibility look
Application Layer Gateway Ser...		Manual	Local Service	Provides support for application level
Application Management		Manual	Local System	Processes installation, removal, and
ASP.NET State Service		Manual	Network Service	Provides support for out-of-process
ASPI over IP Target Service	Started	Manual	Local System	ASPI Tunnelling Protocol (ATP)
Automatic Updates	Started	Automatic	Local System	Enables the download and installation
Background Intelligent Transfe...		Manual	Local System	Transfers files in the background usin
ClipBook		Disabled	Local System	Enables ClipBook Viewer to store info
COM+ Event System	Started	Automatic	Local System	Supports System Event Notification S
COM+ System Application		Manual	Local System	Manages the configuration and tracki
Computer Browser	Started	Automatic	Local System	Maintains an updated list of computer
Cryptographic Services	Started	Automatic	Local System	Provides three management services
DCOM Server Process Launcher	Started	Automatic	Local System	Provides launch functionality for DCO
DHCP Client	Started	Automatic	Network Service	Registers and updates IP addresses
Distributed File System		Manual	Local System	Integrates disparate file shares into
Distributed Link Tracking Client	Started	Automatic	Local System	Enables client programs to track linke
Distributed Link Tracking Server		Disabled	Local System	Enables the Distributed Link Tracking

Figure D.1: ATP Target implemented as native Windows Service

Bibliography

[1] Sebastian Abeck, Peter C. Lockemann, Jochen Schiller, and Jochen Seitz. *Verteilte Informationssysteme.* dpunkt Verlag, 2002.

[2] Adaptec. Advanced SCSI Programming Interface (ASPI). Technical reference (SDK), 1998.

[3] S. Deering and R. Hinden. RFC 2460, Internet Protocol, version 6 (IPv6) specification. Request for comment, 1998.

[4] Rob Elliott. SAS comparison to Fibre Channel with FCP. Presentation slides, Hewlett-Packard, 2003.

[5] Rob Elliott. SAS SCSI upper layers. Presentation slides, Hewlett-Packard, 2003.

[6] S. Hopkins and B. Coile. The ATA over Ethernet protocol. Protocol specification, Coraid, http://www.coraid.com/documents/AoEr8.txt, 2004.

[7] Patrick Beng T. KHOO and Wilson Yong H. WANG. Introducing a flexible data transport protocol for network storage applications (HyperSCSI). Protocol specification, MCSA Group, NST Division - Data Storage Institute, Affiliated to the National University of Singapore, 2003.

[8] Microsoft. Handling IRPs: What every driver writer needs to know, 2003.

[9] John Nagle. RFC 896, congestion control in IP/TCP internetworks. Request for comment, 1984.

[10] J. Postel and J. Reynolds. RFC 959, File Transfer Protocol (FTP). Request for comment, 1985.

[11] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. RFC 3720, Internet Small Computers System Interface (iSCSI), 2004.

[12] INCITS T10. Small Computer Systems Interface – 2 (SCSI-2), 1994.

[13] INCITS T10. SCSI Architecture Model – 2 (SAM-2). Standards draft, 2002.

[14] INCITS T10. SCSI RDMA Protocol (SRP). Standards draft, 2002.

[15] INCITS T10. Multimedia Commands – 5 (MMC-5). Standards draft, 2005.

[16] INCITS T10. SCSI Primary Commands – 3 (SPC-3). Standards draft, 2005.

[17] INCITS T13. AT Attachment – 5 with Packet Interface Draft (ATAPI-5), 2000.

[18] Fujita Tomonori and Ogawara Masanori. Performance of optimized software implementation of the iSCSI protocol. Technical report, NTT Network Innovation Laboratories, 2003.