A platform-independent iSCSI Target in Java

Bachelor Thesis

Submitted by Andreas Ergenzinger

for Fulfillment of the Requirements for the Degree of

Bachelor of Science (B. Sc.)

in Information Engineering

Assessor: Prof. Dr. Marcel Waldvogel

Associate Assessor: Prof. Dr. Marc H. Scholl

Supervisor: Sebastian Graf, M. Sc.

Department of Computer and Information Science
University of Konstanz

April, 2013

© Copyright

by

Andreas Ergenzinger

2013

Contents

List of Figures . . . . **ii**

List of Tables . . . . **iii**

List of Abbreviations . . . . **iv**

1 Introduction **1**

2 Background **3**
- 2.1 SCSI 3
- 2.2 iSCSI Basics 5
  - 2.2.1 The iSCSI Protocol Stack 6
  - 2.2.2 iSCSI Nodes and Sessions 7
  - 2.2.3 Protocol Data Units 7
  - 2.2.4 Command and Numbering 9
  - 2.2.5 Connection Stages and Session Types 10
  - 2.2.6 Text Negotiation 11
- 2.3 Block-based I/O using iSCSI 13
  - 2.3.1 PDUs for Reading and Writing 14
  - 2.3.2 Read Operation 15
  - 2.3.3 Write Operation 15
  - 2.3.4 Multipath I/O 16

3 jSCSI Target Implementation **19**
- 3.1 System Architecture and Package Structure 19
- 3.2 Creating, Sending, and Receiving PDUs 20
- 3.3 Sessions, Connections, Phases, and Stages 21
- 3.4 Text Negotiation and Fast Parameter Access 22
- 3.5 Logical Unit Simulation and Data Storage 24
- 3.6 Error Handling 25

4 Performance Evaluation and Discussion **27**
- 4.1 General Testing Procedure 27
- 4.2 Effect of the ImmediateData, InitialR2T, and Block Size 28
- 4.3 Effect of Burst Length 29
- 4.4 Effect of Data Segment Length and TCP Buffers Sizes 30
- 4.5 Comparison of the jSCSI Target and the IET 32
- 4.6 Effect of the Number of Sessions 33

5 Conclusion **37**

Bibliography **39**

List of Figures

1.1 Comparison of Java and Traditional iSCSI Implementations 1

2.1 SCSI Client/Server Model 4
2.2 Operation code field of a Command Descriptor Block 5
2.3 iSCSI Layers 6
2.4 iSCSI PDU Encapsulation 7
2.5 iSCSI Sessions 8
2.6 Basic PDU Structure 8
2.7 General Layout of the Basic Header Segment. 9
2.8 Connection Lifetime Phases and Stages 10
2.9 PDU Types for Reading and Writing Data 14
2.10 Multipath I/O Layouts 17
2.11 Position of the Wedge Driver for Multisession I/O 17

3.1 jSCSI Target Architecture 19
3.2 Settings Package Class Hierarchies 22
3.3 The Settings Class 24
3.4 Hierarchy of Classes Related to Data Storage 24

4.1 Effect of ImmediataData, InitialR2T, and Block Size on I/O Performance . 29
4.2 Effect of Burst Length on I/O Performance 30
4.3 Interaction of MaxRecvDataSegmentLength and TCP Buffer Size 31
4.4 Effect of MaxRecvDataSegmentLength and TCP Buffer Size 32
4.5 jSCSI Target vs. iSCSI Enterprise Target 34
4.6 Effect of the Number of Sessions 35

List of Tables

2.1 Status codes indicating the outcome of a completed SCSI command 6
2.2 PDU Opcodes by Category 9
2.3 Stage Codes of the CSG and NSG fields 10
2.4 Chart of iSCSI Operational Parameters 12

3.1 Execution Methods for Starting the Connection, Phases, and Stages 21
3.2 Text Negotiation Method Call Sequence 23

4.1 Host and Network Details 27
4.2 Parameter Configurations for Comparing Target Implementations 33

List of Abbreviations

ACA	Auto Contingent Allegiance
AHS	Additional Header Segment
BHS	Basic Header Segment
CDB	Command Descriptor Block
CmdSN	Command Sequence Number
CPU	Central Processing Unit
CSG	Current Stage
ExpCmdSN	Expected Command Sequence Number
ExpStatSN	Expected Status Sequence Number
FFP	Full Feature Phase
ICMP	Internet Control Message Protocol
IET	iSCSI Enterprise Target
I/O	Input/Output
IP	Internet Protocol
iSCSI	Internet Small Computer System Interface
JVM	Java Virtual Machine
LBA	Logical Block Addressing
LONS	Login Operational Negotiation Stage
LTDS	Logical Text Data Segment
MaxCmdSN	Maximum Command Sequence Number
MC/S	Multiple Connections per Session
MPIO	Multipath I/O
MTU	Maximum Transmission Unit
NACA	Normal ACA
NIC	Network Interface Card
NOP	No Operation
NSG	Next Stage
OS	Operating System
PDU	iSCSI Protocol Data Unit
RFC	Request For Comments
SAL	SCSI Application Layer
SAM	SCSI Architecture Model
SAN	Storage Area Network
SBC	SCSI Block Commands
SCSI	Small Computer System Interface
SNS	Security Negotiation Stage
SNACK	Sequence Number Acknowledgement
SPC	SCSI Primary Commands
StatSN	Status Sequence Number
RAM	Random Access Memory
RTT	Round Trip Time
R2T	Ready To Transfer

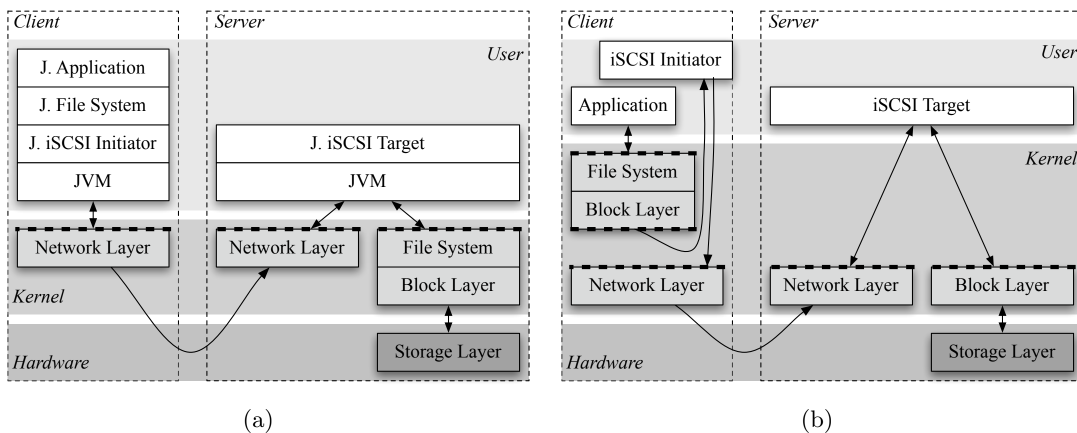
TCP	Transfer Connection Protocol
TSIH	Target Session Identifying Handle
UTF-8	8-Bit Universal Character Set Transformation Format
LUN	Logical Unit Number

Chapter 1

Introduction

In the first chapter of his master thesis [16] from February 2007, Volker Wildi included the following two figures: Figure 1.1b shows how an application can access a block storage device, belonging to a remote host, as if this device is part of the local machine. This is made possible by two programs, an iSCSI initiator running on the client machine and an iSCSI target that runs on the server side, which are connected over a network. Figure 1.1a shows an iSCSI initiator written in Java, that can be included in any Java program to perform raw disk I/O on storage devices accessible through connected iSCSI targets. In his thesis, Wildi describes the development of such an initiator.

Five years have passed since then and several related documents have been published in the meantime, but only now one of them deals with the right-hand part of Subfigure (a), describing the implementation of an iSCSI target written in Java.



(a) (b)

Figure 1.1: Comparison of Java (a) and traditional iSCSI implementations (b)

The Java iSCSI Initiator has to potential to greatly enhance the capabilities of a Java program by offering direct read and write access to block devices. Developing an iSCSI target in Java, on the other hand, is a little bit more difficult to justify. Although numerous target implementations for all kinds of platforms exist, from Linux computers [5] to the iPhone [6], so far nobody has developed a target in Java. There are probably two reasons for this: Reduced performance due to running in the Java Virtual Machine (**JVM**) and a lack of low-level access, which only allows storing data in files. Depending on how big the performance penalty really is, these drawbacks could be outweighed by having an iSCSI target implementation capable of running on any system with an installed JVM.

Chapter 2

Background

This chapter begins with a short introduction to SCSI, followed by a look at the iSCSI protocol and a section on block-based I/O. The goal of this chapter is to familiarize the reader with the background information necessary for understanding the remaining parts of this document.

The information in this chapter is taken from the following SCSI- and iSCSI-related standards documents: SAM-5, SPC-3, SBC-3, and RFC 3720 [12–15]. Other sources are mentioned where this is neccessary.

2.1 SCSI

The Small Computer System Interface (SCSI) is a family of standards specifying commands, transport protocols, and requirements serving the goal of transmitting data between computers and peripheral devices. SCSI is used mostly for interfacing with storage devices, such as disk and tape drives, however SCSI protocol supports a wide range of device types, including card readers, scanners, and printers.

In addition to iSCSI, which is covered in section 2.2, three distinct SCSI standards have been relevant during the development of the jSCSI Target:

- SAM-5 (SCSI Architecture Model) [13]: defines the general structure and the basic behavior shared by all SCSI devices and establishes a layered model allowing for the definition of modular standards.
- SPC-3 (SCSI Primary Commands) [15]: describes universal SCSI commands understood by all SCSI devices.
- SBC-3 (SCSI Block Commands) [14]: describes commands specific to direct access block devices.

Most SCSI devices fall into one of two categories – they are either SCSI initiator or SCSI target devices. A SCSI initiator device originates device service and task management requests to be processed by SCSI target devices and receives the resulting device service and task management responses returned by those targets. Only few devices act as both initiator and target.

A device service requests is a command, i.e. a description of a unit of work to be carried out by a device server. As illustrated by Figure 2.1, a device server object is an element

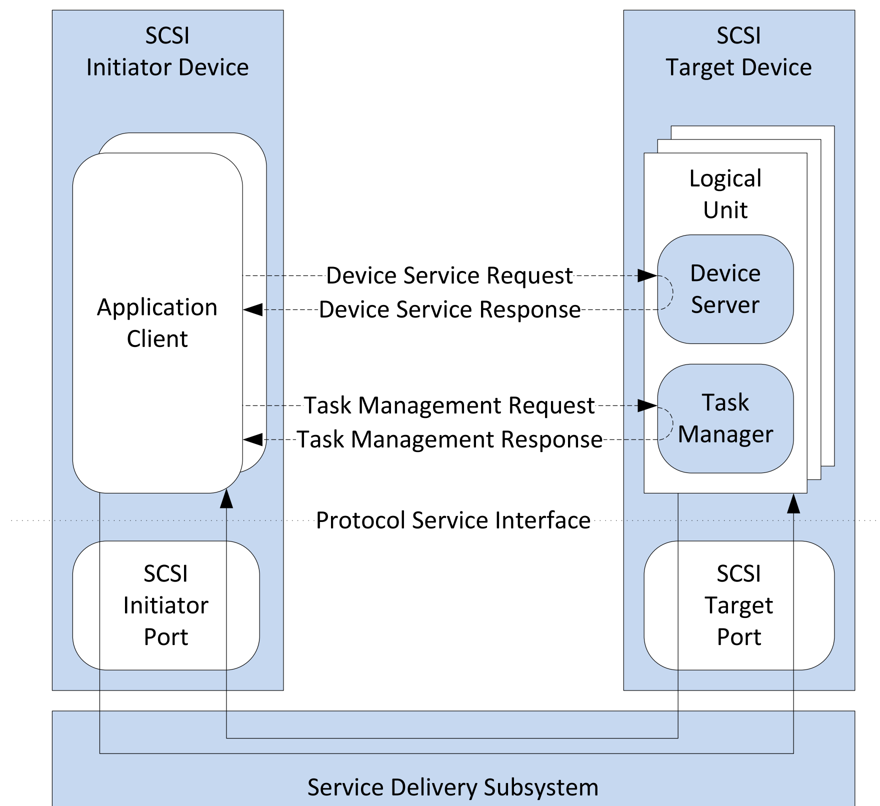


Figure 2.1: SCSI Client/Server Model

of a logical unit, dedicated to processing commands. Task management requests sent to a logical unit are processed by its task manager and affect the processing of one or more commands by the device server. For example, an initiator might send a task management request to abort an ongoing write command.

Each logical unit implements a single device model, this means that it is capable of providing all the functionality required from a specific SCSI device type. Multiple logical units per SCSI target device are possible, but most contain just one. Logical Units are addressed by their Logical Unit Number (LUN), a 64-bit or 16-bit identifier unique across all logical units connected to the same service delivery subsystem.

The logical unit counterparts on the initiator side are application clients. An application client originates commands and task management requests and keeps track of all requests for which it has not received a response, yet.

This architecture corresponds to the client/server model, with the application clients acting as the clients and logical units acting as servers. The logical flow of commands and task management transactions, as it appears to these components, is depicted in Fig. 2.1 using dashed arrows. The actual transaction flow is represented by the solid line arrows. Both client and server rely on the transmission functions provided by the SCSI initiator port and the SCSI target port, respectively. The ports hide the details of the transmission process from the client and server, therefore to them a transaction always looks the same, irrespective of the kind of service delivery subsystem being used.

An application client issues a command to a device server by sending it a Command Descriptor Block (CDB), a byte structure that identifyes the requested operation and contains required command parameters. A CDB typically has a length of 6, 10, 12, or 16

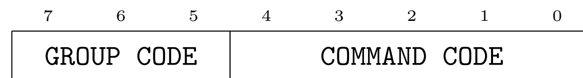


Figure 2.2: Operation code field of a Command Descriptor Block

bytes total. A variable-length format allows CDBs of up to 260 bytes. The first byte of a CDB is the `OPERATION CODE` field, idenfifying the requested operation. The `OPERATION CODE` field is split into the `GROUP CODE` field and the `COMMAND CODE` field, as shown in Fig. 2.2. The former determines the CDBs format, the latter field's value usually cannot be interpreted on its own. Including vendor-specific operation code values, a total of 256 possible operation codes exists. Some operation codes specify only the general category of the requested operation, relying on the `SERVICE ACTION` field, provided for by all but the 6-byte CDB format, for disambiguation.

Some commands require data to be sent to the logical unit, which cannot be included in the CDB, e.g. parameter lists or write data. That data is made available to the logical unit in the application client's data-out buffer[1]. Data returned as part of the command will be copied to the application client's data-in buffer.

When the device server has finished processing the command, it signals the application client by returning a status code. The possible values are listed in Table 2.1. GOOD status means that the command completed without errors, all other codes indicate command failure. If the status is CHECK CONDITION, then the device server will return sense data in the sense data buffer, giving a more detailed description of what caused the error. If, additionally, the logical unit supports Auto Contingent Allegiance (ACA) conditions and the CDB's Normal ACA (NACA) bit is set to one, then an ACA condition is established. While this ACA condition is in effect, the processing of all commands from the initiator that originated the command is paused. The initiator can clear the ACA condition by sending an appropriate task management request.

The sizes of the data-in, the data-out, and the sense data buffer are communicated as necessary.

After initiating the command delivery process by the service delivery subsystem, the application client waits for the delivery system's service response. Two response values are possible: COMMAND COMPLETE indicates that the device server has returned status and that all data and parameters have been transferred successfully. SERVICE DELIVERY OR TARGET FAILURE means that the delivery process has failed and that all data and parameters returned by the device server are invalid.

2.2 iSCSI Basics

iSCSI protocol is a transport protocol for SCSI Application Layer (SAL) transactions, that uses a TCP/IP network as the interconnect. The iSCSI standard is defined in RFC 3720 [12], updated by RFCs 3980, 4850, and 5048 [1, 8, 17]. A historical account of the protocol's development is given by Hufferd [7].

[1] Transfer directions are defined from the initiator's viewpoint, i.e. 'incoming' or 'in' refers to data transmitted from the target to the initiator, 'outgoing' or 'out' refers to data transmitted in the opposite direction.

Status	Code
GOOD	0x00
CHECK CONDITION	0x02
CONDITION MET	0x04
BUSY	0x08
Obsolete	0x10
Obsolete	0x14
RESERVATION CONFLICT	0x18
Obsolete	0x22
TASK SET FULL	0x28
ACA ACTIVE	0x30
TASK ABORTED	0x40

Table 2.1: Status codes indicating the outcome of a completed SCSI command

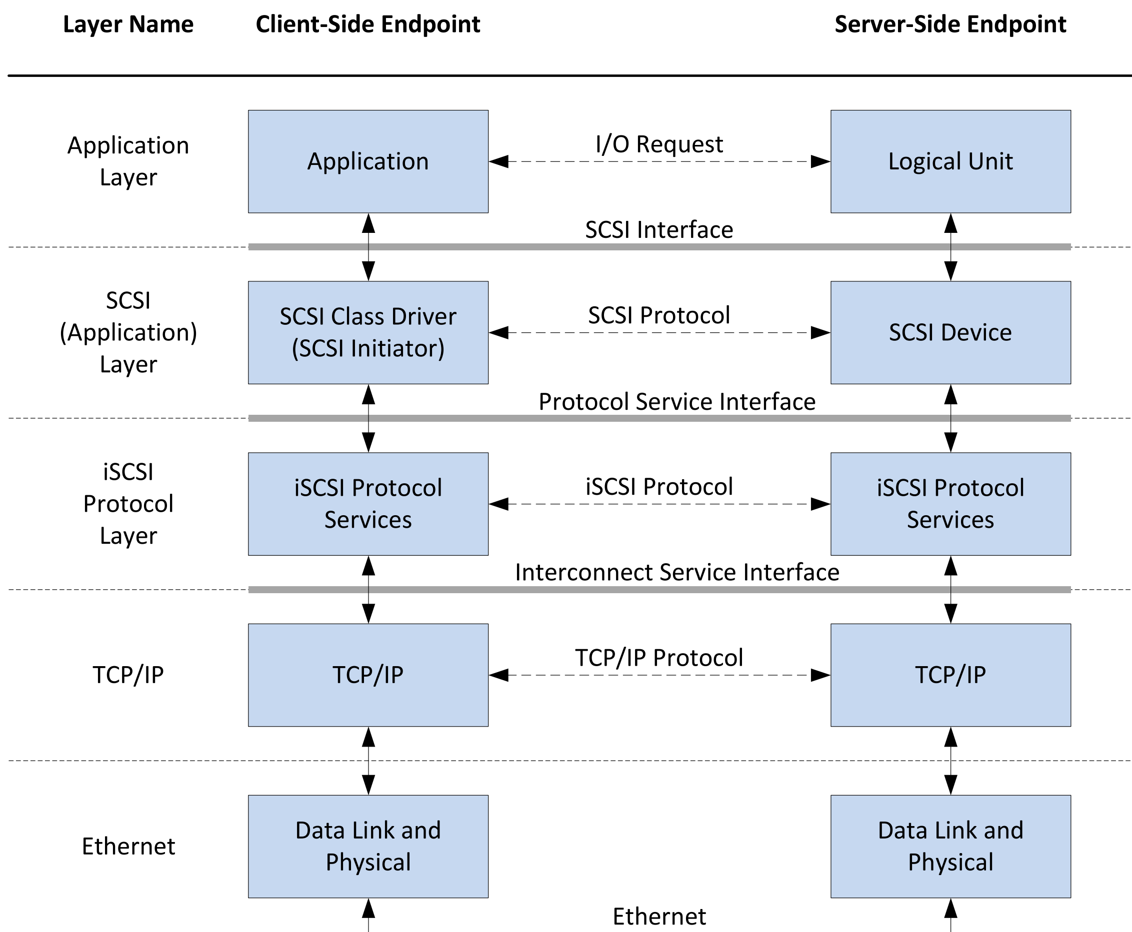


Figure 2.3: iSCSI Layers. Source: [7].

2.2.1 The iSCSI Protocol Stack

The communication layers involved in an application accessing a remote logical unit via iSCSI protocol is shown in Figure 2.3. Each of the four upper layers relies on the services of the layer directly below for data transport, indicated by solid-line arrows. Ethernet is depicted as the link layer standard, however other comparable standards may be used.

Communications in the iSCSI protocol layer are split into discrete messages referred to as (iSCSI) Protocol Data Units (PDU)[2]. Several PDU types exist, some are used for transporting SCSI requests and responses, others serve control purposes, such as setting up and checking the connection between two iSCSI layer endpoints. PDUs are covered more deeply in Section 2.2.3.

Figure 2.4 shows how a PDU is encapsulated by headers and trailers used by lower-layer protocols during transport over the Ethernet layer. Since PDU length may exceed the MTUs of the lower layers, large PDUs may be split across several Ethernet frames during delivery.

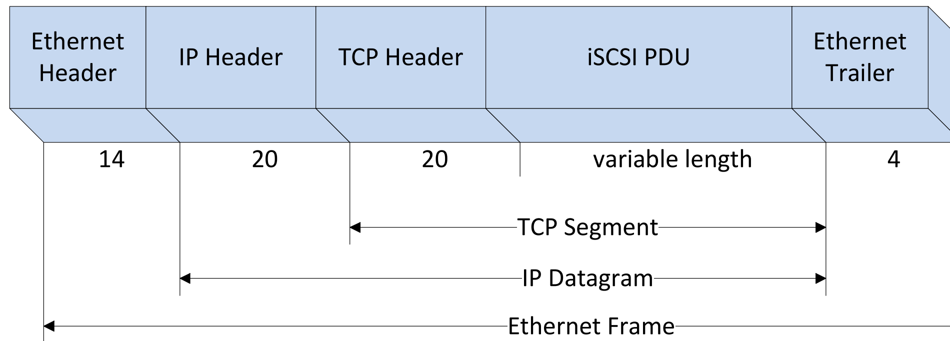


Figure 2.4: Encapsulation of an iSCSI PDU by lower-layer protocol data during transport over the Ethernet layer, with the size in bytes of headers and trailers. Source: [7].

2.2.2 iSCSI Nodes and Sessions

The iSCSI layer entities originating, receiving, and processing PDUs are called iSCSI nodes. Since the client/server architecture of the SCSI standard remains valid for the iSCSI protocol, two types of iSCSI nodes can be distinguished: The client-side iSCSI node is called the iSCSI initiator, the corresponding server-side node is called the iSCSI target.

The exchange of PDUs takes place over connections, which in turn use regular TCP connections for sending and receiving messages. An iSCSI node must be able to have at least one open connection, but if both sides support it, multiple connections are possible. All PDUs belonging to the same iSCSI transaction must be sent over the same connection, a limitation referred to as command connection allegiance.

The TCP connections between iSCSI nodes are logically grouped together in sessions. Sessions with more than one connection are called MC/S (Multiple Connections/Session) sessions. A target assigns a Target Session Identifying Handle (TSIH), a unique 16-bit tag, to each of its sessions, which must be communicated to the initiator. If the initiator wants to add another connection to an existing session, it must include the specified TSIH in the first PDU sent over the newly-established connection.

2.2.3 Protocol Data Units

Figure 2.6 shows the basic PDU structure. All PDUs have a Basic Header Segment (BHS) at the beginning, which may be followed by a number of Additional Header Segments

[2] Traditionally, the term ‘Protocol Data Unit’ is used in layered protocol systems to describe messages of any protocol layer. In this document, however, ‘PDU’ always means ‘iSCSI PDU’.

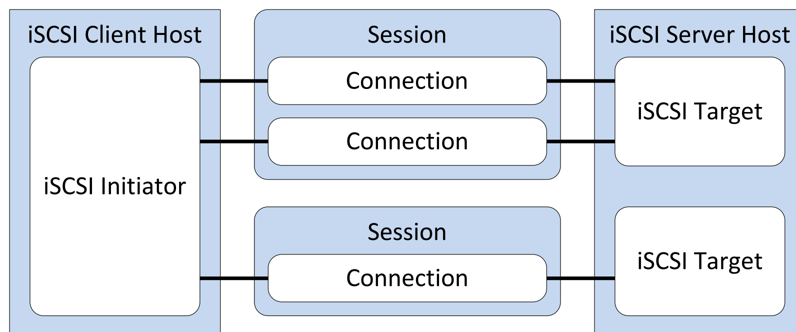


Figure 2.5: iSCSI Sessions

(AHS). Together, these segments form the PDU's header. The integrity of the header part and of the optional data segment may be independently protected through the use of checksums included in the header digest field and the data digest field, respectively.

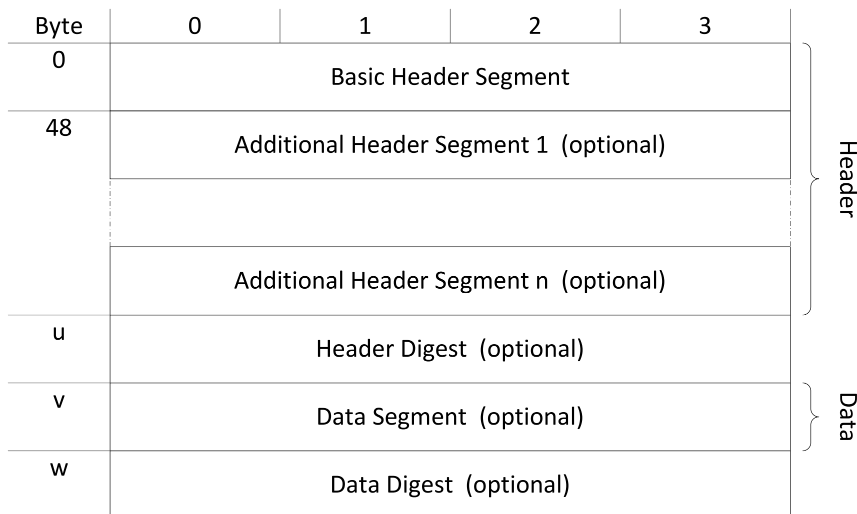


Figure 2.6: Basic PDU Structure

The BHS has a fixed length of 48 bytes and adheres to the general layout shown in Fig. 2.7. The first byte of the BHS contains the `Opcode` field, which determines the PDU type and the specific structure of the BHS. There are two opcode categories: initiator-opcodes used exclusively in PDUs sent by the initiator and target opcodes, used exclusively in PDUs sent by the target. Table 2.2 lists all opcodes by iSCSI node type.

The total length of all additional header segments, measured in four-byte words, is specified in the `Total AHS Length` field of the BHS. Two AHS types exist, both of which may only be used in SCSI Command PDUs. One is used in case the length of a SCSI CDB exceeds the 16-byte limit of the SCSI Command BHS's CDB field. The other one is used for bidirectional SCSI commands – commands that both read and write data. However, these cases are only of theoretical importance.

Also part of the BHS is the `Data Segment Length` field, which specifies the length in bytes of the PDU's optional data segment. The data segment is used for transferring iSCSI task parameters or data from the SCSI layer, such as in, out, and sense data. The maximum data segment length is determined by the `MaxRecvDataSegmentLength` parameter of the receiving node. The default is 8192 bytes, but both initiator and target may independetly declare different values.

Byte	0			1		2	3
	0 1 2 3 4 5 6 7			0 1 2 3 4 5 6 7		0 1 2 3 4 5 6 7	0 1 2 3 4 5 6 7
0	0	I	Opcode	F	Opcode-specific Fields		
4	Total AHS Length			Data Segment Length			
8 12	LUN or Opcode-specific Fields						
16	Initiator Task Tag						
20	Opcode-Specific Fields						
48							

Figure 2.7: General Layout of the Basic Header Segment.

PDU Type	Opcode
NOP-Out	0x00
SCSI Command	0x01
SCSI Task Management Function Request	0x02
Login Request	0x03
Text Request	0x04
SCSI Data-Out	0x05
Logout Request	0x06
SNACK Request	0x10
Vendor-specific Codes	0x1c-0x1e

(a) Opcodes of Initiator PDUs

PDU Type	Opcode
NOP-In	0x20
SCSI Response	0x21
SCSI Task Management Function Response	0x22
Login Response	0x23
Text Response	0x24
SCSI Data-In	0x25
Logout Response	0x26
Ready To Transfer (R2T)	0x31
Asynchronous Message	0x32
Vendor-specific Codes	0x3c-0x3e
Reject	0x3f

(b) Opcodes of Target PDUs

Table 2.2: PDU Opcodes by Category

SCSI Command PDUs and other initiator PDUs

If the PDU is transmitted as part of an iSCSI transaction started by the initiator, then the `Initiator Task Tag` field of the BHS contains an initiator-supplied task identifier, which allows identifying all PDUs sent as part of the same task.

2.2.4 Command and Numbering

The initiator uses a session-wide counter called the *Command Sequence Number* for numbering all commands sent to the target. The number is included in the `CmdSN` field which is part of all initiator PDU types that may be sent at the beginning of a new task. The sequence number specifies the order in which commands must be executed by the target, irrespective of the order in which the commands actually arrive at the target. The *Command Sequence Number* is also used in a windowing scheme that prevents the initiator from overwhelming the target with too many commands. The boundaries of the current command window are communicated to the initiator in the final last target PDU of each command, in the `ExpCmdSN` and `MaxCmdSN` fields.

The following equation shall hold for all command PDUs sent by the target:

$$ExpCmdSN \leq CmdSN \leq MaxCmdSN$$

Stage	Code
Security Negotiation	0
Login Operational Negotiation	1
Full Feature Phase	3

Table 2.3: Stage Codes of the CSG and NSG fields

If the CmdSN of a PDU violates this equation, then the message will be ignored by the target. The command window size may be dynamically adjusted, e.g. if the target is running out of buffer space.

2.2.5 Connection Stages and Session Types

The lifetime of a connection can be divided into two consecutive phases, the Login Phase and the Full Feature Phase (FFP), listed in order. The Login Phase consists of up to two stages, the optional Security Negotiation Stage (SNS) and the mandatory Login Operational Negotiation Stage (LONS). Figure 2.8 shows the allowed transitions between these stages and phases.

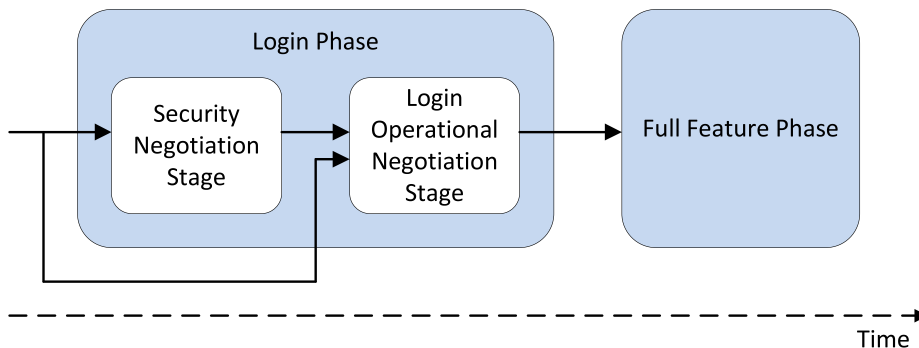


Figure 2.8: Connection Lifetime Phases and Stages

While the connection is in the Login Phase, the initiator may only send Login Request PDUs, each of which must be answered with a Login Response PDU from the target. Both Login PDU types have a CSG (Current Stage Group) and a NSG (Next Stage Group) field in their respective BHS, which can take the values listed in Table 2.3. The CSG field of the initiator's initial Login Request PDU, the first PDU sent over the connection, determines the connection's starting stage. After that, the CSG value must always match the stage code of the current stage. While the initiator wants to remain in the current stage, the values of the NSG and the CSG fields must match. Once it is ready to transition to the next stage, it must set value of the NSG field accordingly. The target signals a successful transition by returning a response PDU with the requested NSG field value.

Throughout the login phase, the connected iSCSI nodes use a text-based method for information exchange that is described in Section 2.2.6. During the SNS it is used for mutual authentication of the initiator and target nodes, during the LONS the method is used to declare and negotiate operational parameters of the connection. The negotiated changes only take effect after the Login Phase has been cleared. Until then the default parameter values are used.

In case of an error during the SNS or LONS, or if the initiator has skipped the SNS but the target considers authentication to be mandatory, the node which first notices the error

has to close the connection. In case the target discovered the the problem it has to send a Login PDU with an according error message, first. This means that the Login Phase serves as the gateway to the FFP, the only phase during which the initiator can request the target to perform actual work. Barring serious errors, the connection will remain in the FFP until the initiator sends a Logout Request PDU and the target responds with a Logout Response.

The selection of PDUs the initiator is allowed to send during the FFP of a connection is determined by the `SessionType` parameter of the enclosing session. This parameter can take two values: "Normal" and "Discovery". Which one is determined by the initiator in the initial Login PDU.

A discovery session is limited to one connection and the initiator is only allowed to use Text Request and Logout Request PDUs during the FFP. This is enough, however, to retrieve a list of target names and associated target addresses, which is the purpose of the discovery session.

If the initiator wants to open a normal session connection to a target from that list, then that target's name and a part of its target address must be included in the initial Login PDU. But after a successful Login Phase, the initiator may send all PDU types from Table 2.2a (except for Login Request PDUs, of course) and access the target's logical units.

2.2.6 Text Negotiation

iSCSI uses a versatile, easily extensible, text-based method called text negotiation that allows the initiator and target to communicate and negotiate shared operational parameters. The method relies on the exchange of key-value pairs for transmitting information. A key-value pair is a composite string with a key part at the beginning and a value part at the end, separated by an equals sign. The whole string is case-sensitive and encoded in a subset of UTF-8. The key part consists of a single key indentifying an operational parameter. A string representation of one or more relating values is contained in the value part. All operational parameters defined in the iSCSI standard can be addressed with keys that match their respective name, however, for backward compatibility reasons, some alternative keys exist.

The initiator and target nodes exchange key-value pairs by including them in the data segment of a sent Login or Text PDU. Each pair is followed by a null byte, which allows multiple key-value pairs to be included in the same PDU. If the total length of the resulting text data segment exceeds the receiver's limit (8192 bytes durin login) then it is called a Logical Text Data Segment(LTDS). An LTDS is transmitted in a sequence of several PDUs, sliced into chunks of acceptable sizes.

Table 2.4 lists all operational parameters that may be negotiated during the LONS, along with their most important properties. Operational parameters are either declared or negotiated. The value of a declared parameter is determined by just one of the iSCSI nodes. The chosen value must be communicated to the other node with a single key-value pair that has exactly one value. This value must be silently accepted by the receiver. Negotiated parameters, on the other hand, are selected in a colaborative process, in which both sides have a say. Negotiating a parameter always requires two key-value pairs: the first one specifies the values supported by the sender, the second one, sent in response to the first, describes the outcome of the negotiation. The value part of the response consists of a single value which is either one of those specified by the initial sender or one of the following: "Reject", if there is no overlap between the values supported by the initiator

Result Function Symbols	
Boolean Result Functions	
∧	AND
∨	OR
Num. Result Functions	
+	Maximum
-	Minimum

		Parameter / Key																								
		DataDigest	DataPDUInOrder	DataSequenceInOrder	(Default)Time2Retain	(Default)Time2Wait	ErrorRecoveryLevel	FirstBurstLength	HeaderDigest	IFMarker	IFMarkInt	ImmediateData	InitialR2T	InitiatorAlias	InitiatorName	MaxBurstLength	MaxConnections	MaxOutstandingR2T	MaxRecvDataSegmentLength	OFMarker	OFMarkInt	SessionType	TargetAddress	TargetAlias	TargetName	TargetPortalGroupTag
Data Type	Boolean		∨	∨						∧		∧	∨							∧						
	Numerical (Single Value)				-	+	-	-								-	-	-	•							•
	Numerical (Interval)										•										•					
	String	•							•					•	•							•	•	•	•	
	Declared (Initiator)													•	•				•			•		•	•	
	Declared (Target)																		•				•			•
	Negotiated	•	•	•	•	•	•	•	•	•	•	•	•			•	•	•		•	•					
Scope	Connection-only	•							•	•	•								•	•	•					
	Session-wide		•	•	•	•	•	•				•	•	•	•	•	•	•				•	•	•	•	•
Use	SNS													•	•							•	•	•	•	•
	LONS	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•	•
	FFP													•					•				•			
	Leading Only		•	•	•	•	•	•				•	•			•	•	•				•				
	Default Value	None	Yes	Yes	20	2	0	65536	None	No	2048	Yes	Yes			262144	1	1	8192	No	2048	Normal				

Table 2.4: Chart of iSCSI Operational Parameters

and the target, “Irrelevant”, if the parameter no longer matters due to the values of the other parameters, or “Unknown”, if the receiver does not recognize the key. A response value of “Reject” represents a critical error, resulting in an early termination of the login phase and closing of the connection. Both initiator and target may start the negotiation of a corresponding parameter and both sides must accept the result. Redeclaration and renegotiation of parameters is not allowed during the login phase.

Parameters can also be distinguished by their data type. Three major types exist: boolean, numerical, and string.

Boolean parameters can take values of “Yes” (i.e. `true`) and “No” (i.e. `false`). Key-value pairs relating to a boolean parameter may only carry one of these. Different outcomes for negotiated boolean parameters are produced using boolean result functions. The result functions `AND` and `OR` combine the desired values of the initiator and target using the boolean logic operator of the same respective name.

The values of numerical parameters are integers. For negotiated numerical parameters two separate classes can be distinguished: parameters that allow only a single integer in the initial key-value pair and parameters that allow the explicit specification of an integer interval (the boundary values are separated by a tilde character). The negotiation outcome for the former class is determined based on numerical result functions: the `Minimum` function selects the smaller number from the integer sent in the initial key-value pair and the value desired by the receiver, the `Maximum` function selects the larget one.

String parameters have the least requirements regarding the format of values, any combination of nonrestricted characters is allowed. Multiple string values can be contained in a key-value pair as a comma-separated list, ordered from highest to lowest preference.

The scope of an operational parameter describes to how many connections it applies, either to a single connection (connection-only), or to all connections of a session (session-wide).

A parameter's use property refers to the circumstances under which it may be declared or negotiated. Use is determined by a combination of partially interdependent modifiers[3]. Parameters marked with LONS may be declared or negotiated during Login Operational Negotiation Stage. A couple of parameters must or can only be declared right at the beginning of a connection's lifetime, in the first Login Request and Response PDU, respectively. These parameters are marked with SNS to indicate that their keys may also be used during the Security Negotiation Stage. The FFP use modifier identifies those operational parameters, that can be changed even after the connection has reached the Full Feature Phase. Some session-wide parameters can only be negotiated or declared over the first connection of a session. These parameters are marked with Login Only.

Except for InitiatorAlias and TargetAlias, all parameters without a default value must be negotiated or declared during login. For the remaining parameters this step is optional, their respective default setting will be used if neccessary.

Some parameters have special rules regarding the meaning and processing of proposed values. For example it must hold that MaxRecvDataSegmentLength $\leq$ FirstBurstLength $\leq$ MaxBurstLength and, since key-value pairs may be transmitted in arbitrary order, the receiver of a (logical) text data segment must wait for the arrival of the last key-value pair before it may begin processing any of them.

2.3 Block-based I/O using iSCSI

Most data storage devices are block-based. That means the storage medium is partitioned into sections of equal size called blocks, which can only be read and written as a whole. Typical block sizes range from 512 to 4096 bytes. SCSI uses the LBA scheme for referencing the blocks of a block-based storage device in which blocks are consecutively numbered, beginning with 0.

The SCSI Block Commands standard [14] defines a wide range of SCSI commands for reading from and writing to block devices. However, only four of them, READ (6), WRITE (6), READ (10), and WRITE (10), are considered mandatory and the first two of these are deprecated, because the lengths of the `LOGICAL BLOCK ADDRESS` and `TRANSFER LENGTH` fields (21 and 8 bits, respectively) of the associated 6-byte CDBs are deemed too short for today's requirements. In the 10-byte CDBs of the READ (10) and WRITE (10) commands, these limits are raised to 32 bits and 16 bits, respectively.

The transfer length specifies the number of consecutive blocks to be read or written by the command, starting with the block at the position given in the logical block address field.

The initiator can inquire a logical unit's blocks size and number of blocks by sending a SCSI Command PDU carrying either a READ CAPACITY (10) or READ CAPACITY (16) SCSI command.

[3] Note that, for clarity's sake, this document does not utilise the original use modifier labels from [12].

2.3.1 PDUs for Reading and Writing

Five PDU types from Table 2.2 play a role in transmitting SCSI read and write commands and all accompanying data over iSCSI: the SCSI Command and SCSI Data-Out PDU sent by the initiator, as well as the SCSI Response, SCSI Data-In, and R2TPDUs sent by the target. These PDUs and their most important fields, listed in Fig. 2.9 shall be presented in this section.

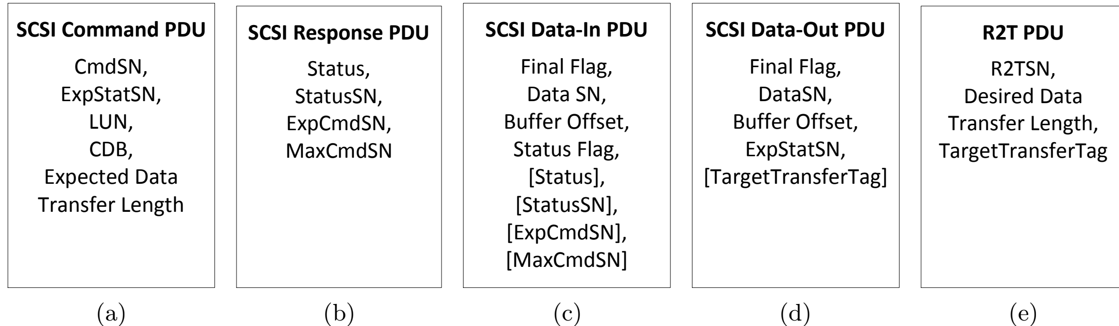


Figure 2.9: PDU types for reading and writing data, including their major type-specific fields. Fields in brackets may contain reserved values.

The SCSI Command PDU is used for transmitting a SCSI CDB to the target and the PDU's arrival marks the beginning of a new iSCSI task. The `CmdSN` field carries the current Command Sequence Number of the connection and the `ExpStatSN` contains the Status Sequence Number the initiator expects to receive with the last PDU from the target sent during the course of the new task. The value in the `InitiatorTaskTag` field carries a unique initiator-supplied task identifier, that has to be included in the `InitiatorTaskTag` fields of all PDUs belonging to the current task. This allows multiple ongoing tasks at the same time over the same connection. The `CDB` field contains the command descriptor block of the SCSI command to be processed by the logical unit specified by the logical unit number in the `LUN` field. The `Expected Data Transfer Length` field declares the total length in bytes of the data that is to be read from or written to the logical unit. For read and write commands, this value must match the logical unit's block size times the transfer length value of the CDB. If the command is accompanied by write data, then the initiator may be allowed to include some (or all) of it in the SCSI Command PDU's data segment as immediate data.

SCSI Data-Out PDUs serve as the primary means of transport of write data from the initiator to the target. This data is included in the PDU's data segment, up to the `MaxRecvDataSegmentLength` limit in bytes that has been declared by the target during the login phase. Due to this limitation, the write data may have to be spread accross several Data-Out PDUs. The `Buffer Offset` field specifies the offset in bytes, from the memory address given in the CDB, at which the contained chunk of data shall be written. A Data-Out PDU is always part of a sequence – a collection of succeeding PDUs of the same type that are sent without interruptions. The Data-Out PDU's location whithin a sequence is given in the `DataSN` field. Additionally, the last PDU of a sequence is marked with a `F` (Final) flag set to one.

For the iSCSI target a task started by the initiator ends with the sending of status (see Table 2.1) which describes the outcome of the SCSI command. The default method for transporting status is incuding it in the `Status` field of a sent SCSI Response PDU. The `StatusSN` field contains the current value of the associated status sequence number. The

values in the ExpCmdSN and MaxCmdSN fields specify the command window for subsequent commands, as described in section 2.2.4.

The target's counterpart to the initiator's SCSI Data-Out PDU is the SCSI Data-In PDU. This message type is used for carrying read data from the target to the initiator and it may also be sent in sequences. Under certain circumstances, the target may decide to forego sending the SCSI Response PDU at the end of the task and include status, the expected as well as the maximum command sequence number in the last Data-Out PDU instead. This so-called phase- collapse is indicated by setting the Status Flag of the last PDU to one.

To avoid targets from being overwhelmed by sequence after sequence of Data-Out PDU's, R2T (Ready To Transfer) PDUs are used to regulate the sending behavior of initiators. Usually, each Data-Out sequence must be preceeded by an R2T from the target, signalling its readiness to receive a burst of data. Which part of the data and how much of it the target is willing to receive is specified by the values in the Buffer Offset and Desired Transfer Length fields. The R2Ts of a task are numbered, starting at zero. This R2T sequence number is included in the R2TSN field. The target may enclose a transfer identifier in the TargetTransferTag field of each R2T, which must then be included in the associated Data-Out PDUs.

2.3.2 Read Operation

This section describes the basic iSCSI read operation for sessions with ErrorRecovery-Level=0. The initiator triggers a read task by sending a SCSI Command PDU with a read command CDB. The target responds with a sequence of SCSI Data-In PDU which carry the content of the specified memory regions. The size of the Data-In PDUs' data segments is limited to the MaxRecvDataSegmentLength parameter declared by the initiator and the total number of transferred read data may not exceed the negotiated MaxBurstLength value. Iff the DataPDUInOrder parameter is set to Yes, then the buffer offset of the data PDUs must continually increase and the forwarded read data chunks must not overlap.

In case the iSCSI nodes have agreed on accepting immediate data (i.e. Immediate-Data=Yes), the target may send status in the last Data-In PDU. Otherwise the target finishes the task by sending a SCSI Response PDU.

If the error recovery level is greater than 0, then the target may request positive acknowledgement from the initiator for the received data by setting the A (Acknowledge) flag of the final SCSI Data-In PDU to one. The initiator is forced to respond with a SNACK Request PDU that informs the target about the number of successfully transmitted Data-In PDUs. If holes in the data are detected, the target is supposed to repeat the failed data transmission sequence.

2.3.3 Write Operation

The initiator starts a write task by sending a SCSI Command PDU with a write command CDB. In a session that allows sending immediate data, the initiator may include the first chunk of write data with buffer offset zero in the Command PDU's data segment. This write data, if any, is considered to be part of the first burst, during which the combined length of all data segments may not exceed the value of the FirstBurstLength parameter. If InitialR2T=Yes, then the first burst ends with the SCSI Command PDU. Otherwise,

the initiator may send a single sequence of Data-Out PDUs directly afterwards, which is also part of the first burst.

While there remains unwritten data for the command, additional bursts follow. Each burst begins with an R2T PDU sent by the target, detailing which part of the data it wants to receive next. The initiator complies with a sequence of Data-Out PDUs carrying the requested data. The total length of write data transmitted during a follow-up burst is limited to `MaxBurstLength` in bytes. As during the read operation, the requirements regarding the order and overlap of the delivered data chunks is determined by the `DataPDUInOrder` parameter. The session's `DataSequenceInOrder` parameter plays a comparable role – if its value is `Yes`, then consecutive R2Ts may only request continuous non-overlapping ranges of the data to be transmitted by the following Data-Out PDU sequence. In the simple case the target only sends one R2T PDU and waits for the arrival of the desired data before sending the next R2T. However, the target may have multiple outstanding Data-Out sequences in a connection at the same time, up to the negotiated value of the `MaxOutstandingR2T` parameter.

In case of an error during the transmission of the Data-Out PDU sequence and if the `ErrorRecoveryLevel` is greater than zero, the target is allowed to send a recovery R2T PDU, that requests a retransmission of the missing data.

Once the target has received and written all announced data to the logical unit, it will finish the task by sending a SCSI Response PDU.

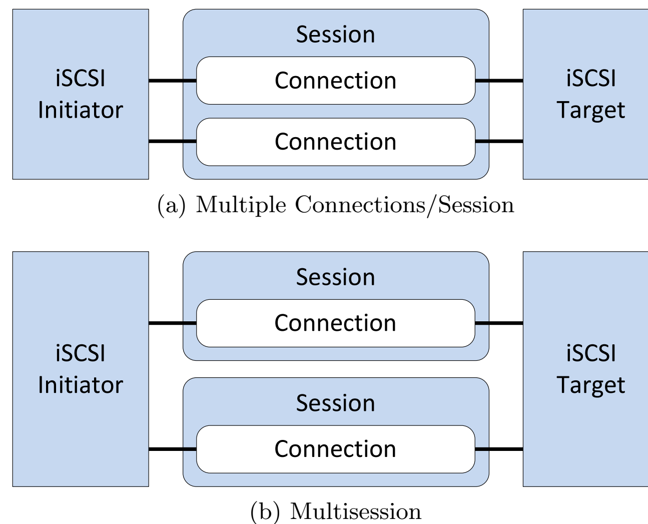
2.3.4 Multipath I/O

Multipath I/O (MPIO) refers to the use of more than one physical or logical connection for reading and writing data. With regards to iSCSI, the term describes a setup in which an iSCSI initiator and an iSCSI target are connected by more than one connection. Although such an arrangement means an increase in complexity, this is usually outweighed by the potential benefits of higher throughput and fast connection failover. The multipath connections of an iSCSI node may all originate at the same Network Interface Card (NIC). Ideally, however, each connection uses a different NIC and an exclusive route not shared by the other connections. This both increases the available bandwidth and minimizes the effect of network component failure, since a malfunction along one of the routes will only affect a single connection.

Two major multipath solutions for iSCSI can be distinguished: Multiple Connections/Session (MC/S) and multisession. Figure 2.10 illustrates the structural differences between these two.

MC/S is part of the iSCSI standard and therefore a protocol level multipath feature. Ordered delivery of (SCSI) commands across connections is guaranteed by the session-wide command sequence number included in all SCSI Command PDUs.

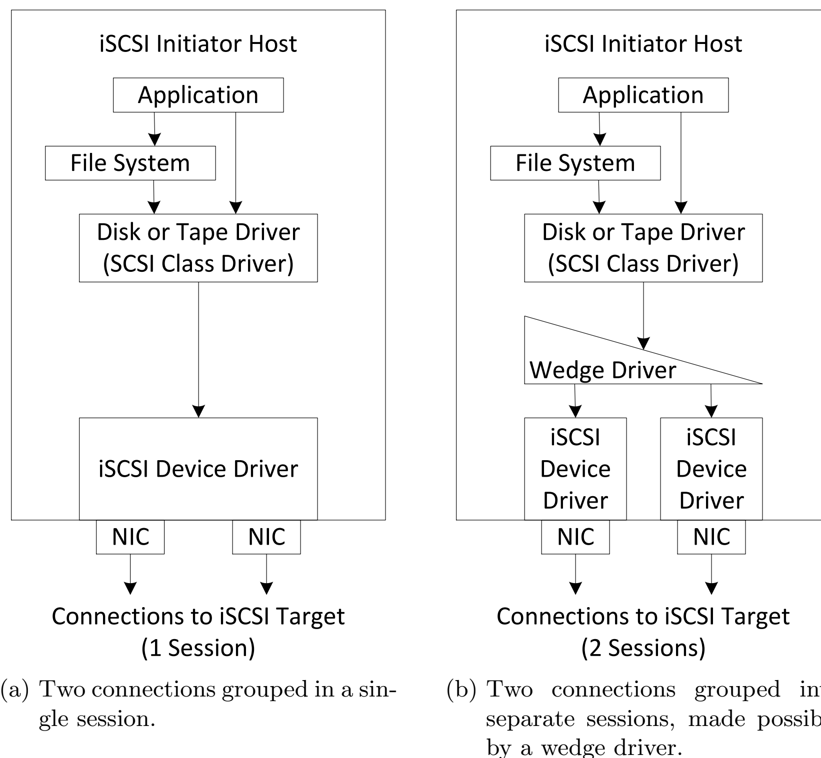
In a multisession setup, multiple sessions between the initiator and target exist, each of them having just one connection. Multisession is not part of the iSCSI protocol, though. This means that the order in which SCSI commands, sent over different sessions, are delivered to and executed by a logical unit, is uncertain, unless additional actions are taken. The vendors of SCSI target devices address this issue with so-called initiator wedge drivers – vendor-specific drivers that assign each command to one of the available sessions. This obviously allows load balancing across sessions, but it also enables the logical unit



(a) Multiple Connections/Session

(b) Multisession

Figure 2.10: Multipath I/O layout examples with two connections.



(a) Two connections grouped in a single session.

(b) Two connections grouped into separate sessions, made possible by a wedge driver.

Figure 2.11: Position of the wedge driver for multisession I/O. Source: [7].

to figure out the original order of incomming SCSI commands. Figure 2.11 shows the position of a wedge driver in a multisession iSCSI environment.

Since there is no common standard for wedge drivers, a client in a diverse SAN with a variety of storage devices might need several different wedge driver implementations for universal multisession access. This represents a potential drawback of the multisession solution compared to MC/S, which may only need to be implemented once for each iSCSI initiator and target software. However, since wedge drivers function independently from the SCSI transport protocol, their usefulness is not limited to just iSCSI storage environments.

Although device-specific wedge drivers might be able to use multiple sessions more efficiently, it is possible to develop universal drivers for this task, based on operation conflicts. Two operations (read or write) are said to be in conflict if different orders of execution lead to different outcomes. Hence, a conflict can only occur if at least one of the operations is a write operation and both access the same data. A wedge driver can assign nonconflicting incomming SCSI commands to any session without risks to data integrity. Only if an incoming command is detected to be in conflict with an ongoing or queued SCSI operation, special care must be taken. Conflicting commands must either be assigned to the same session in order of arrival, exploiting ordered delivery within sessions, or be postponed until all older conflicting operations have finished.

Chapter 3

jSCSI Target Implementation

This chapter offers a description of the jSCSI Target's final implementation and provides some of the rationale that went into the design. The first section of this chapter gives a short overview of the software architecture and identifies the major functional categories. The remaining sections explore these categories more deeply.

3.1 System Architecture and Package Structure

With only little prior experience in server programming in general and even less knowledge about the specific requirements of developing an iSCSI target, it was deciced to keep the class structure of the jSCSI Target as simple as possible. The major components of the final architecture and their structural relationships are shown in Fig. 3.1. Please note that most component labels in the figure do not match actual class names and that components may be implemented by a combination of multiple coequal classes.

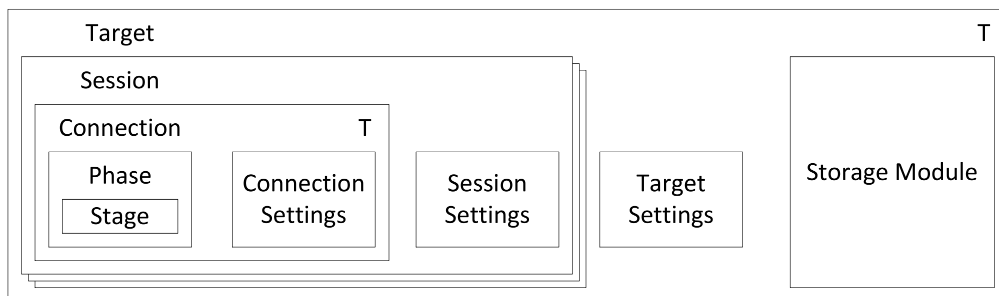


Figure 3.1: Logical architecture of the jSCSI Target. Components with their own associated thread are marked with a T in the upper right-hand corner.

The architecture of the jSCSI Target is largely determined by the basic iSCSI concepts presented in Chapter 2. The target has a single thread that accepts new (TCP) connections and assigns them to sessions. To keep complexity low, the number of connections a session can have is fixed to one, i.e. for each connection a new session is created. This would have allowed to ignore the difference between operational parameters with session-wide and connection-only scope, however, in order to allow MC/S sessions in a future implementation, the distinction was preserved. Consequently, three different settings categories exist: connection, session, and target settings, which manage and provide parameters of corresponding scope.

Each connection has an associated thread that receives, processes, creates, and sends PDUs. Blocking I/O is used for sending and receiving. This means that a connection behaves like a finite state machine that is controlled by the PDU's from the initiator. To describe the state of the connection and determine the allowed transitions, phases and stages are used. A connection's phase gives a broad description of its current state (login phase or full feature phase). At a finer granularity, a phase's stage represents an ongoing iSCSI task.

This means that a connection is either in a specific stage, or waiting for the next one to begin. Processing multiple tasks at the same time is not possible, as is having the target send asynchronous messages, which is why the command window size must be held constant at one.

The jSCSI target offers access to a single virtual direct-access block device. The read and write stages use the corresponding methods of the storage module to access a file that holds the logical unit's memory blocks.

The components of the target's architecture can be grouped into three categories: storage-related, settings-related, and structural or task-related. These categories correspond to three of the following five base packages: connection, scsi, settings, storage, and util. The connection package contains the connection, session, phase, and state classes. The settings package contains everything relating to text negotiation and operational parameters. The classes of the storage package pertain to persistent data storage. The scsi package contains classes and interfaces required for processing SCSI commands and mimicking the communication behavior of a simple SCSI block storage device. Lastly, the util package contains classes with static methods for bit manipulation and accessing integer fields of different size as well as a couple of miscellaneous classes that do not fit elsewhere.

Although technically not a package of the jSCSI Target, the PDU-related core and parser packages of the jSCSI Initiator [16, Sections 3.3 and 3.4] shall also be mentioned here. The classes from these packages are used for instantiating, serializing and deserializing objects that represent iSCSI PDUs.

3.2 Creating, Sending, and Receiving PDUs

PDUs are represented by a nested hierarchy consisting of a ProtocolDataUnit, a BasicHeaderSegment, and an opcode-specific Parser object. Therefore, accessing a specific field of a PDU might require traversing several layers of that hierarchy, first. In order to facilitate PDU creation, the jSCSI Target uses a factory class with a static build method for each target PDU type. That way a PDU can be created and all of its neccessary fields directly set with just one method call.

Not all PDU fields are set this way, however. The ExpCmdSN, MaxCmdSN, and StatSN fields, if present, are set just prior to the serialization of the ProtocolDataUnit object, by the method responsible for sending PDUs.

An additional method exists for receiving PDUs from the initiator. This method first creates a blank ProtocolDataUnit object and then deserializing the PDU off the TCP connection. As part of the deserialization process, basic integrity checks are performed. If the PDU carries a SCSI command, then the method will also check if the value of the CmdSN field falls into the command window. If not this will be treated as an iSCSI error (see Section 3.6).

Class Name	Execution Method Signature
TargetConnection	Void call()
TargetLoginPhase	boolean execute(ProtocolDataUnit pdu)
TargetFullFeaturePhase	void execute()
TargetStage (abstract class)	void execute(ProtocolDataUnit pdu)

Table 3.1: Execution methods for starting the the connection, phases, and stages.

Since blocking IO is used for receiving and sending messages, the target must only call the method for receiving, when it is the initiator's turn to send the next PDU, lest the thread is blocked indefinitely.

3.3 Sessions, Connections, Phases, and Stages

Each TCP connection accepted by the target is immediately associated with a TargetConnection object. The TargetConnection class implements the `java.util.concurrent.Callable` interface, which means its *call()* method can be executed in a separate thread. This happens in a thread pool of fixed size, which makes it easy to control the maximum number of threads, but also limits the number of active connections the target can have.

The login phase and the full feature phase are represented by instances of the TargetLoginPhase and TargetFullFeaturePhase classes, respectively. For each specific stage there exists a subclass of the abstract TargetStage class. The connection's thread enters a phase or stage by calling its respective execution method listed in Table 3.1.

A new TargetConnection object first enters the login phase by calling the execution method of a TargetLoginPhase instance. The method's return value indicates the outcome of the login phase and determines whether the connection must enter the full feature phase afterwards or terminate prematurely.

Once in the full feature phase, the connection's thread enters a loop that lasts until either the initiator sends a Logout Request PDU or an iSCSI error occurs. During each iteration of the loop, the initial PDU of the next stage is received and a corresponding TargetStage object is created and its execution method is called.

Each iteration of the loop consists of receiving an initial PDU, inspecting the ProtocolDataUnit object to identify the next stage, creating a corresponding TargetStage object, and running its execution method. Since the initial PDU may contains additional information and data, it must be passed during the method call.

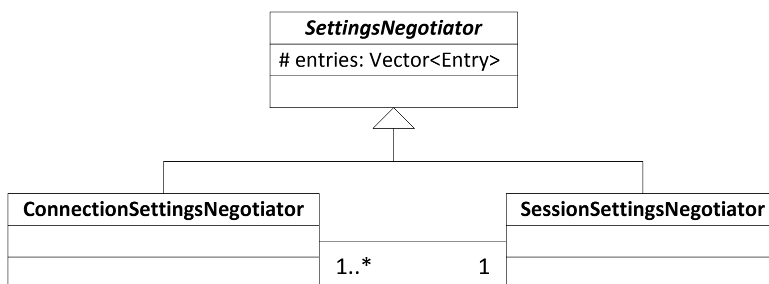
The sessions of the jSCSI Target are represented by instances of the TargetSession class. Normal sessions are distinguished from discovery sessions by means of an enum variable which is set after the connection has cleared the login phase, based on the declared SessionType parameter. At first glance, using different classes for the two session types might seem preferable. However, this would mean that the initial Login PDU's data segment would have to be parsed twice. The first time just in order to select the right session class and the second time when extracting all key-value pairs for text negotiation.

Since sessions are by definition mutually independent, implementing multi-session support required little additional effort, compared to an architecture with just one single-connection session. The only point of contact where the multiple sessions could interfere

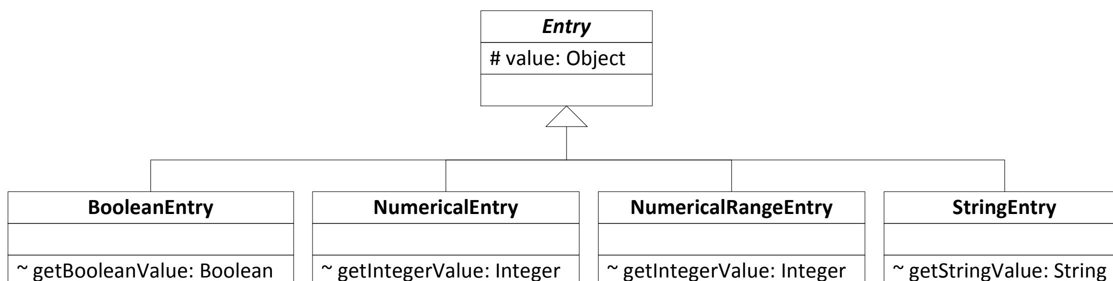
with each other is during reading from or writing data to the storage module. How this concurrency issue was addressed is explained in Section 3.5.

3.4 Text Negotiation and Fast Parameter Access

Connection-specific and session-wide operational parameters are managed by subclass instances of of the abstract SettingsNegotiator class. As shown in Fig. 3.2a, two child classes exist. Each connection has one ConnectionSettingsNegotiator object and the enclosing session has one instance of the SessionSettingsNegotiator class. The former SettingsNegotiator is in charge of all parameters with connection-only scope, the latter is responsible for session-wide parameters. Each operational parameter is managed by a single Entry object. Figure 3.2b shows the four Entry subclasses that are used for parameters of different data type. An Entry object stores the current value of a single operational parameter and performs all text-negotiation related processing for that variable, including parsing of corresponding key-value pairs, negotiation, use and value checks, and, if required, formulation of response key-value pairs.



(a) Class hierarchy of the SettingsNegotiator class.



(b) Class hierarchy of the Entry class.

Figure 3.2: Class hierarchies of the settings package.

Individual Entry objects and the session's SessionSettingsNegotiator can not be accessed directly by classes that do not belong to the settings package. Access is only possible via the public methods of by the ConnectionSettingsNegotiator object. Although this reduces the number of possible errors due to concurrent access by different connections, in a future implementation with MC/S session support errors would still be possible. Therefore, a stage performing text negotiation must adhere to the sequence described in Table 3.2 when accessing the ConnectionSettingsNegotiator's text negotiation methods.

Operational parameters must frequently be accessed during time-critical operations such as reading and writing data. Hence, an implementation is required that can quickly provide the required parameter values, even while another connection belonging to the same session might be involved in text negotiation. To avoid having to search and access a specific Entry object each time a parameter's value is required, container classes capable

Step No.	Action of accessing stage	Responsibilities of CSN and SSN
1	Call *beginNegotiation()* until it returns `true`.	If SSN is already locked, then block until the current lock holder has finished step 4. Then back up current parameter values, grant SSN lock to caller and return `true`.
2	Call *negotiate(...)* passing use data, initiator-supplied key-value pairs, and a container for key-value pairs to be sent by the target. Repeat this step for each PDU sequence with key-value pairs sent by the initiator.	Let each key-value pair be processed by the corresponding Entry object. Add mandatory response and declarative key-value pairs to specified collection. Return `false` in case of error (use violation, missing declarations, value errors).
3	Call *checkConstraints()* to check constraints involving multiple parameters.	Return `false` iff combination of negotiated and declared parameter values violates iSCSI protocol.
4	Call *finishNegotiation(boolean)* with `true` parameter to commit or with `false` parameters to roll back changes.	Keep updated operational parameter values or roll back changes. Release SSN lock.

Table 3.2: Mandatory sequence of steps when accessing the text negotiation methods of a ConnectionSettingsNegotiator object. (CSN: ConnectionSettingsNegotiator, SSN: SessionSettingsNegotiator)

of holding a snapshot of the current parameter values are used. As shown in Fig. 3.3, ConnectionSettingsBuilderComponent and SessionSettingsBuilderComponent objects contain connection-specific and session-wide operational parameters, respectively. These objects are replaced by up-to-date instances every time the represented parameters may have changed.

In order to further simplify parameter access, the Connection- and SessionSettingsBuilderComponent objects are not made available outside of the settings package, instead they are used as constructor arguments of the Settings class, whose instances contain a snapshot of all (connection-only and session-wide) operational parameters that apply to a specific connection. A current Settings object can be retrieved from the ConnectionSettingsNegotiator using the *getSettings()* method. This method always makes sure that the returned object is up-to-date, by comparing its *settingsId* with the *currentSettingsId* of the associated SessionSettingsNegotiator. The latter variable is incremented each time the SessionSettingsNegotiator is involved in a successful text negotiation, so *settingsId* ¡ *currentSettingsId* means that the Settings object must be replaced.

The current Settings object is retrieved once at the beginning of each stage, during the instantiation of the TargetStage object, and is accessed each time the value of an operational parameter is required. Hence, the stage is not affected by any parameter changes resulting from concurrent text negotiations and is guaranteed to use the correct parameter values, as long as TargetStage obects are created in the order dictated by the command sequence number.

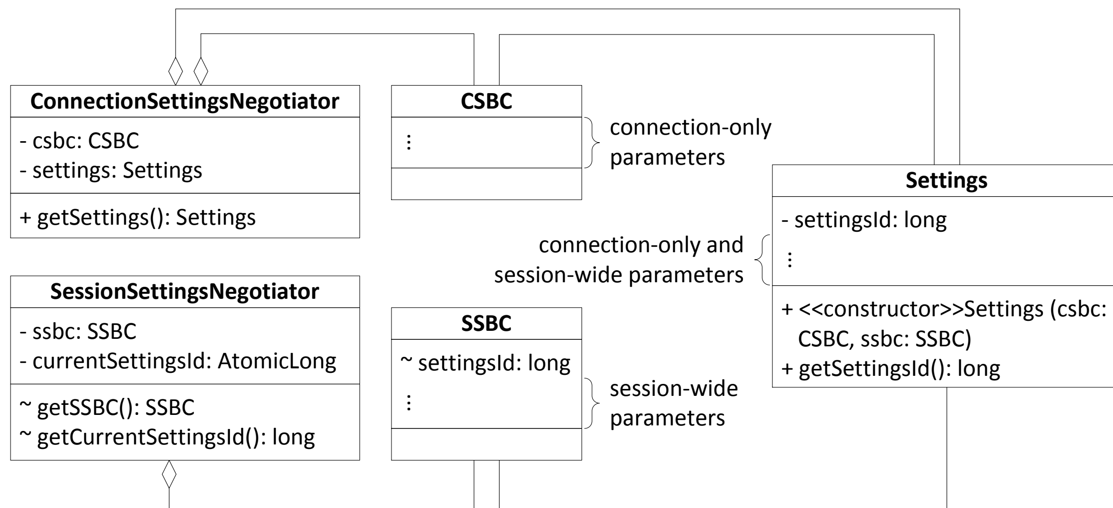


Figure 3.3: The Settings class with related classes and their most relevant fields. (CSBC: ConnectionSettingsBuilderComponent, SSBC: SessionSettingsBuilderComponent)

3.5 Logical Unit Simulation and Data Storage

Due to the goals of keeping the program architecture simple and individual stages as independent as possible, there exists no dedicated class for representing logical units. Instead, the mandatory functionality of a SCSI direct-access block device is distributed accross several stages. This has, in a few occasions, led redundant storage of logical unit parameters, with different copies of a SCSI parameter being accessed by different stages. Additionally, the decision to keep stages separate like this precludes the target from processing SCSI commands that might create changes that last longer than the current stage. Consequently, the jSCSI Target does not support the establishment of ACA conditions due to SCSI errors and rejects SCSI CDBs requesting this feature.

Obviously, the limitation not to accept SCSI commands that produce long-lasting changes does not apply to read and write commands. In order to allow different implementations of how data is persistently stored, while leaving the methods use by stages that read and write data unaffected, the classes for data storage are embedded in the system architecture in a modular fashion. Figure 3.4 shows the inheritance hierarchy of the pertinent classes, including their most important fields and methods.

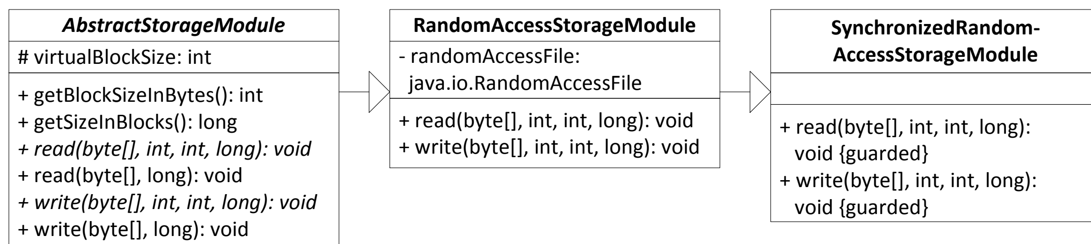


Figure 3.4: Class hierarchy of the storage package classes.

The (abstract) AbstractStorageModule class defines a set of methods to be used by stages performing data I/O. An object of a non-abstract subclass provides both read and write access to a memory area of fixed size. The defined write methods copy some or all bytes of a byte array to a consecutive region of the memory area. The read methods copy bytes in the opposite direction, from persistent storage to a byte array. The number of bytes that

can be copied by calling one of these methods is not linked to the AbstractStorageModule's block size of 512 bytes, i.e. transfers of arbitrary length are possible.

The RandomAccessStorageModule is a non-abstract subclass of the AbstractStorageModule class, that uses a `java.io.RandomAccessFile` for reading and persistently writing data to a specified file. Since the RandomAccessFile class is not declared thread-safe and uses a single file pointer for keeping track of read and write locations, concurrent reads and writes might access the wrong parts of the file.

This issue is resolved by the SynchronizedRandomAccessStorageModule class, which extends the RandomAccessStorageModule class and makes the read and write methods synchronized. This implementation prevents concurrent reads and writes by multiple sessions from interfering with each other. Therefore, the jSCSI Target uses an object of this class to provide data read and write capabilities to those stages responsible for data storage.

3.6 Error Handling

The jSCSI Target distinguishes between two kinds of errors: SCSI errors and iSCSI errors. A SCSI error occurs when a the initiator sends a SCSI command descriptor block that requests an action not supported by the target. The target handles SCSI errors by returning appropriate sense data to the initiator and then continues processing iSCSI tasks as before. An iSCSI error occurs when the target detects an iSCSI protocol error, for example due to receiving an out-of-order PDU or a PDU of an unexpected type. The stage or phase detecting the error throws an exception, which results in the closing of the associated connection.

Chapter 4

Performance Evaluation and Discussion

This chapter describes various kinds of performance measurements and tries to explain the results. The effect of select operational parameters on read and write speed is explored, followed by a performance comparison of the jSCSI Target and the iSCSI Enterprise Target (**IET** [5]). Finally, the potential benefits of multisession I/O are investigated and discussed.

4.1 General Testing Procedure

Table 4.1 lists the hardware and software configuration of the client and server host, respectively, as well as the properties of the physical connection between the two. Both hosts share the same technical configuration and are connected over a Gigabit Ethernet connection with low delay. The average ping round trip time (**RTT**) – the time between sending an Internet Control Message Protocol (**ICMP**) echo request and receiving the ICMP echo response from the remote host – was 0.102ms.

A 1300GiB file was used for data storage, in turn by both targets. In order to prevent read and write performance from being adversely affected by a potential disk I/O bottleneck, the file was stored on a RAM disk of 1.5GiB. Hence, measured read and write speeds should primarily be the result of PDU transmission and processing times.

The virtual logical unit was made available to the client over one single-connection iSCSI session, established by an Open-iSCSI initiator [11]. Open-iSCSI is a fast, popular and mature iSCSI initiator software for Linux systems, written in C. The initiator included the logical unit as a SCSI disk device in the client's local file system and thereby made it accessible to the `dd` command [2].

CPU	Dual Core AMD Opteron Processor 270 (2×2.0GHz)
RAM	8GiB (Target server uses 1.5GiB of that for the RAM-Disk)
Ethernet Controller	Broadcom Corporation NetXtreme BCM5704 Gigabit Ethernet
RTT	(0.102ms, 0.006ms) (Mean, Standard Deviation)
OS	Debian 5.09

Table 4.1: Details of the client, server and network used for performance testing.

`dd` is a Unix command line tool that can be used for low-level copying of raw data to or from raw devices and regular files. Special files such as `/dev/zero` [4] and `/dev/null` [3] can be used for faster input and output, respectively, than would be possible with normal files. The following two invocation examples illustrate the necessary command arguments:

- read: `dd if=/dev/sde of=/dev/zero bs=8M count=100`
- write: `dd if=/dev/zero of=/dev/sde oflag=direct bs=8M count=100`

The parameter prefixes `if="` and `of="` precede the input and output file, respectively. The `bs=` argument declares the block size[1], i.e. the number of consecutive bytes to copy, and the integer following `count=` determines how many of these blocks shall be consecutively copied. The copying of each block may be translated into several iSCSI read and write tasks by the initiator, depending on the transfer length limit of each task and other factors the initiator deems important. By including the `oflag=direct` argument in the write call, the buffer cache is avoided and the data is copied directly to (persistent) memory, instead. The input file is never retrieved from cache.

The combination of Open-iSCSI and `dd` has been used for all measurements except for the multisession-related experiments described in Section 4.6. The client-side multisession setup is explained there.

In the following sections the influence of five operational parameters on I/O performance is explored: ImmediateData, InitialR2T, MaxRecvDataSegmentLength, FirstBurstLength, and MaxBurstLength. All other parameters that might also influence I/O times are fixed and therefore not not subject to investigation, since the jSCSI Target does not support alternative value for those parameters.

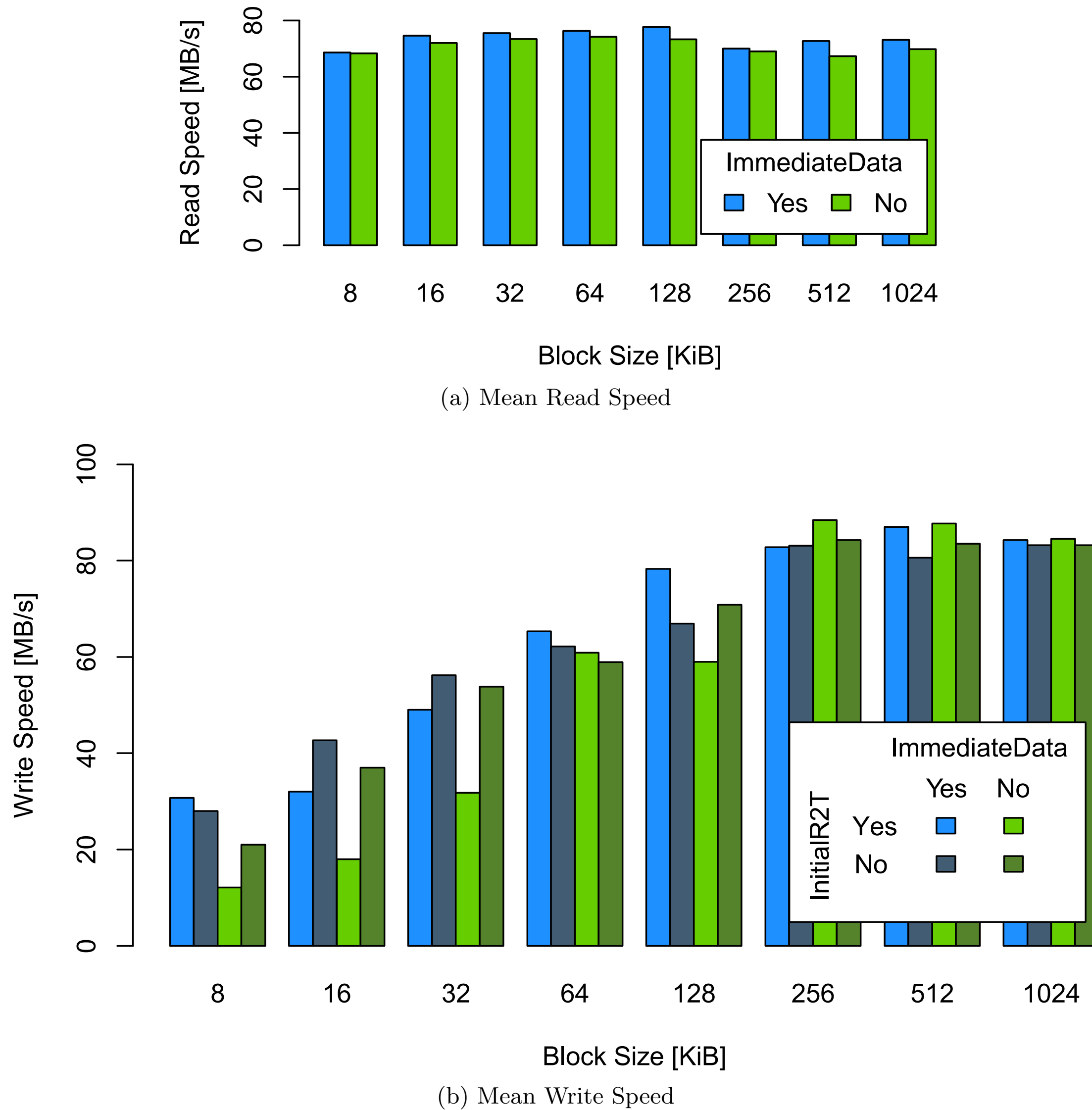
The study of potentially performance-affecting factors is not limited to the iSCSI layer – the effect of the TCP send buffer and receive buffer sizes are analysed as well. These buffers are used by the target for buffering the incoming and outgoing data sent over a TCP connection. The impact of using Nagle's algorithm, defined in RFC 896 [10], was not examined, because its detrimental influence on data-intensive network applications are well-known [9].

4.2 Effect of the ImmediateData, InitialR2T, and Block Size

Figure 4.1 shows the effects of the ImmediateData and InitialR2T parameters, if applicable, on the I/O speeds of read and write commands with different block sizes. Both subfigures, show an advantage of sending immediate data and keeping the total number of PDUs low. This advantage is small but consistent for read commands of all block sizes. Short write commands display the same effect, however much more noticeable. The advantage of the initiator not having to wait for the initial R2T PDU is also clearly visible. The influence of both parameters on write performance evens out as block size increases and the throughput benefits resulting from larger burst sizes begin to dominate any small delays at the beginning of the iSCSI write tasks. All of the effects mentioned so far are to be expected.

Surprisingly, however, block size has only little influence on read speed and the peak write speed is 10.7MB/s faster than data transfers in the other direction. This is in contrast to the usual pattern of read operations being faster than write operations.

[1] This has nothing to do with the block or segment size reported by the logical unit. In this chapter "block size" always refers to the number of consecutive bytes copied by the `dd` command in one go.



(a) Mean Read Speed

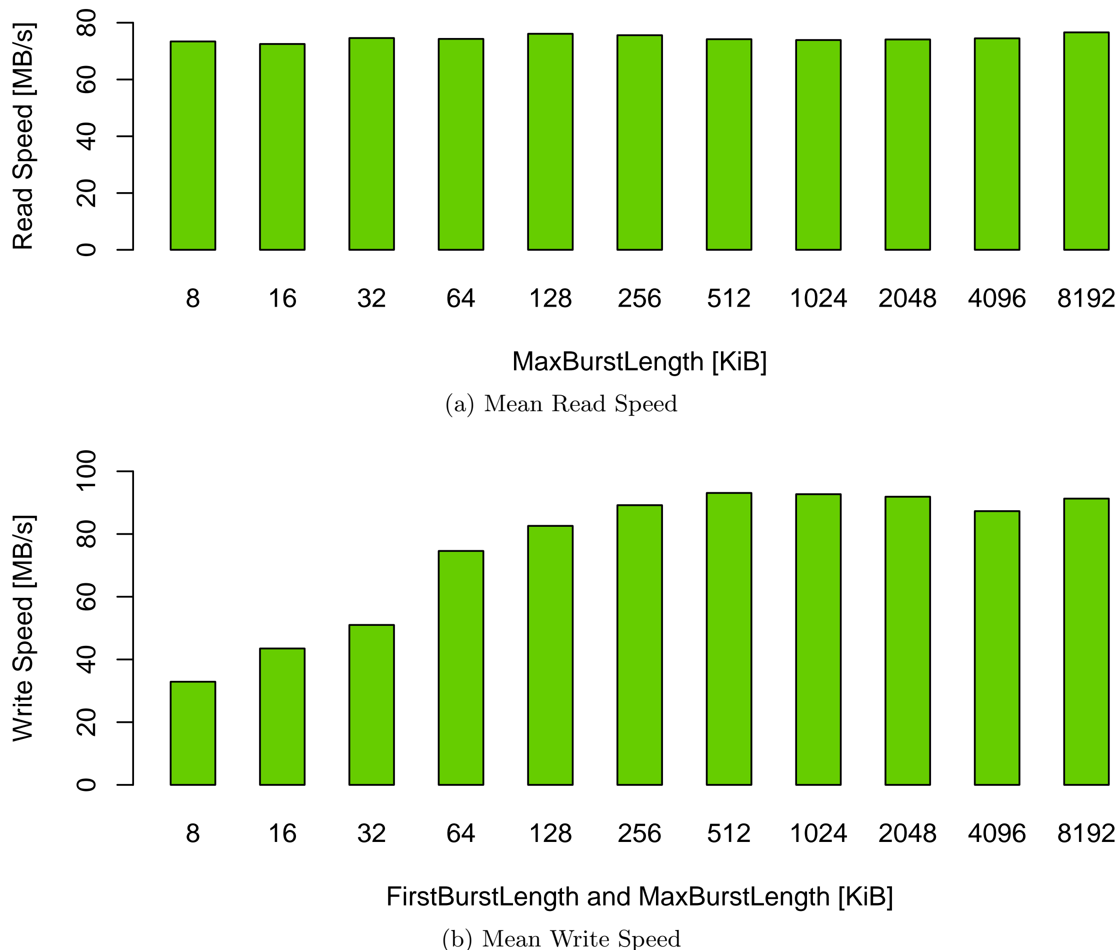
(b) Mean Write Speed

Figure 4.1: Effect of the ImmediataData and InitialR2T parameters as well as block size on read (a) and write (b) performance. FirstBurstLength=65536, MaxBurstLength=262144, MaxRecvDataSegmentLength=8192, send buffer size: 8192B, receive buffer size: 43690B, count per block size: 1000.

4.3 Effect of Burst Length

Since the expected benefits of the parameter settings ImmediateData=Yes and InitialR2T=No had been confirmed in the first experiment, these values were kept for the remaining tests. In order to study the effect that varying the FirstBurstLength and MaxBurstLength parameters might have on performance, the block size was held constant and the two burst length parameters were linked, so that both parameters were always equal. Figure 4.2 shows the results of these comparisons.

Again, mean read speed seemed not to be affected by varying the chosen variables. Write speeds, on the other hand, clearly increased as FirstBurstLength and MaxBurstLength were raised to 512KiB. The jSCSI Target seems to be capable of processing incoming Data-Out PDUs relatively quickly. Longer burst sizes mean longer Data-Out PDU sequences and lower ratios of SCSI Command, SCSI Response, and R2T PDUs per write task.



(a) Mean Read Speed

(b) Mean Write Speed

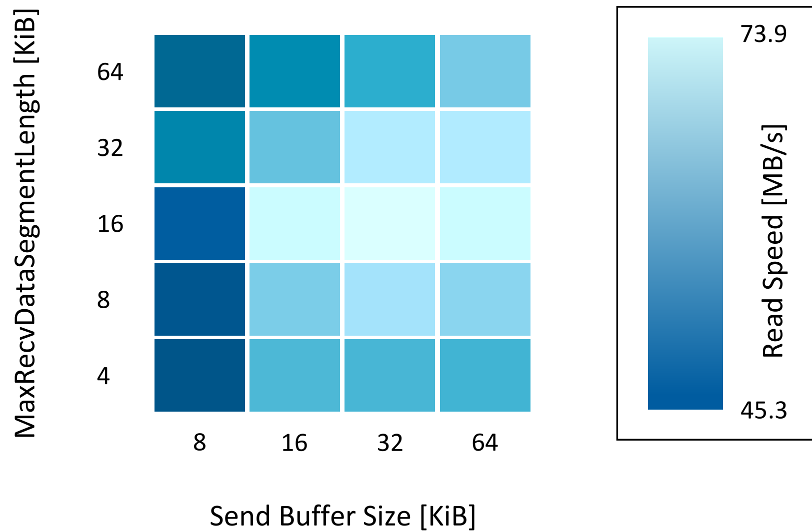
Figure 4.2: Effect of burst length on I/O performance. ImmediateData=Yes, InitialR2T=No, MaxRecvDataSegmentLength=8192, send buffer size: 8192B, receive buffer size: 43690B, block size: 8MiB, count per burst length category: 100.

Due to the beneficial effects on write performance and no negative impact on read speeds, the 512KiB setting for the FirstBurstLength and MaxBurstLength parameters was adopted for most of the the remaining experiments.

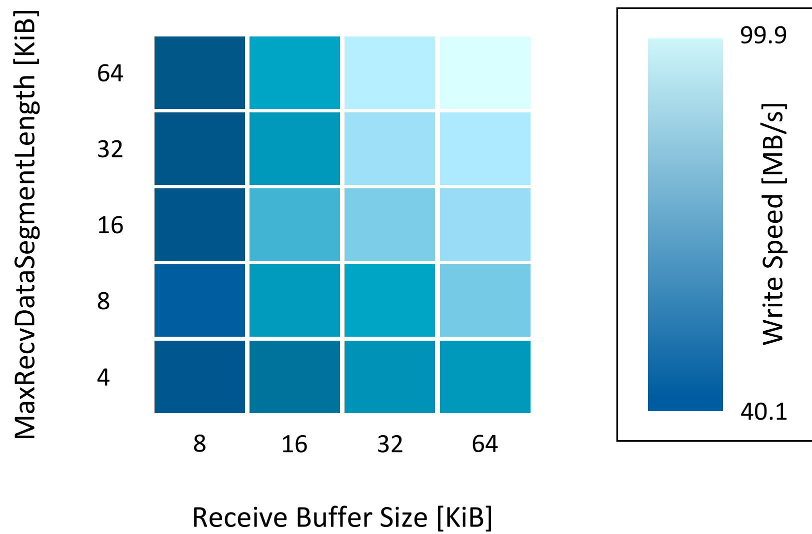
4.4 Effect of Data Segment Length and TCP Buffers Sizes

The MaxRecvDataSegmentLength parameters (one for each direction) have a great influence on the number of Data-In and Data-Out PDUs, respectively, that must be sent during read and write tasks. Although an iSCSI node may send PDUs with data segment lengths smaller than the value declared by the receiver, for performance reasons usually full advantage is taken of the specified limit. Figure 4.3 shows what effect the independent variation of the maximum data segment length and of the respective TCP buffer's size has on I/O speed.

Both subfigures show that, at least for large burst lengths, TCP buffer sizes of 8KiB or less should be avoided, as should be small data segment lengths. As shown by Figure 4.3b, the major interaction effect between the two factors with regards to write speed is quite straightforward: larger values are generally better and best performance is achieved if both values are set to 64KiB.



(a) Mean Read Speed

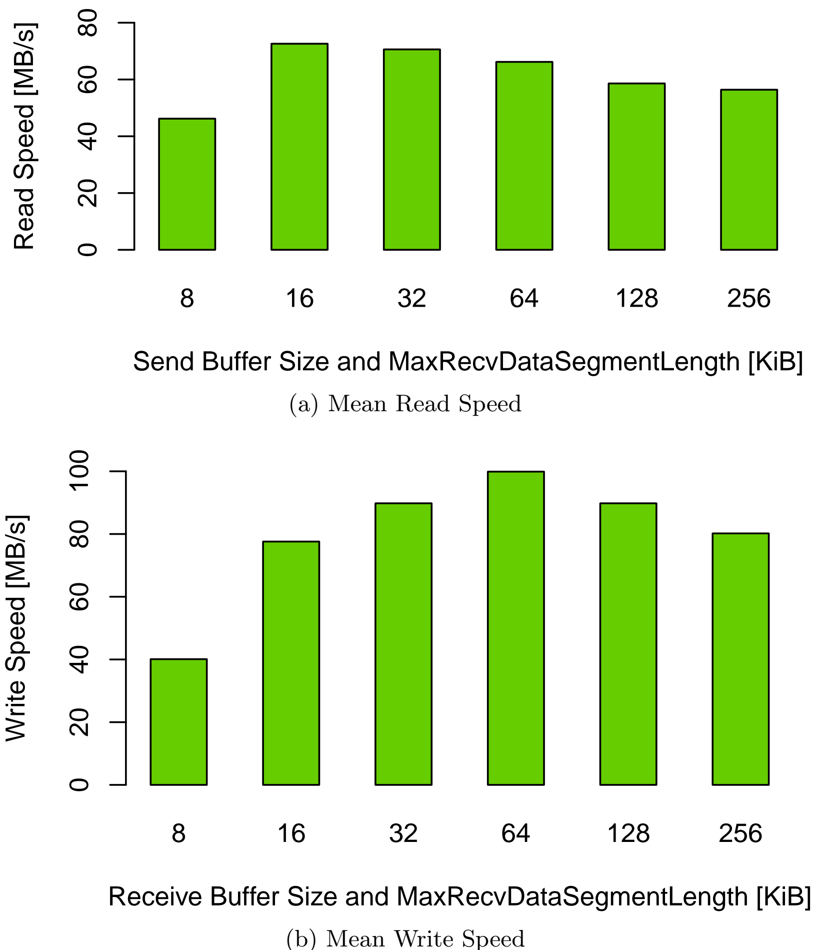


(b) Mean Write Speed

Figure 4.3: Interaction of MaxRecvDataSegmentLength and TCP buffer size. ImmediateData=Yes, InitialR2T=No, FirstBurstLength=512KiB, MaxBurstLength=512KiB, block size: 8MiB, count per matrix element: 100.

For read operations the result is more complicated. According to Figure 4.3a the fastest read times are reached when the data segment size is 16KiB and the send buffer sizes ranges from 16 to 64KiB in size. This more complex behavior is probably due to the TCP receive buffer size of the Open-iSCSI initiator. As the size of the target's PDUs increases, more efficient use is made of the initiator's receive buffer, but as soon as the buffer's length is exceeded, the initiator is overrun and the congestion window size is reduced, resulting in an overall performance drop.

Figure 4.4 explores how far the increase of the MaxRecvDataSegmentLength and buffer sizes can be carried on before negative effects ensue. Subfigure (a) continues the downward pattern indicated int Figure 4.3a. Subfigure (b) shows that probably little if any improvement can be expected beyond the best combination from Figure 4.3b. In fact, further increases in data segment and receive buffer size beyond 64KiB result in worse write speeds. Why this happens is not perfectly clear. Theoretically, a larger receive buffer size



(a) Mean Read Speed

(b) Mean Write Speed

Figure 4.4: Continued Exploration of the impact of MaxRecvDataSegmentLength and TCP buffer size. ImmediateData=Yes, InitialR2T=No, FirstBurstLength=512KiB, MaxBurstLength=512KiB, block size: 8MiB, count per category: 100.

could lead to a larger TCP Receive Window, possibly increasing the number of packets the initiator has to resend in case of packet loss.

4.5 Comparison of the jSCSI Target and the IET

In order to find out how the jSCSI Target compares to a native code implementation, the performance of the Java program was compared to that of the iSCSI Enterprise Target. Read and write speeds were compared under two different conditions, once using the default iSCSI parameter values and once using a parameter configuration optimized for fast writing of data. The parameter values of both configurations are listed in Table 4.2. The results of these comparisons are shown in Figure 4.5.

Immediately apparent from the two subfigures is the performance superiority of the IET in all cases. However, in the write speed comparison, these differences are always well below 20% and if the write-optimized parameter configuration is used, then the difference becomes even smaller.

Differences between the two implementations become much more apparent, when focussing on read performance. As in the previous tests, the jSCSI Target's throughput does not

Parameter	Default Value	Write-optimized Value
ImmediateData	Yes	Yes
InitialR2T	Yes	No
MaxRecvDataSegmentLength	8KiB	64KiB
FirstBurstLength	64KiB	512KiB
MaxBurstLength	256KiB	512KiB

Table 4.2: Operational parameter configurations for comparing the jSCSI Target and the iSCSI Enterprise Target.

increase with block size and hovers between 60 and 70MB/s, whereas the figures for the IET rise up to almost 115MB/s. The read speeds of the jSCSI Target even drop when switching from the default to the write-optimized parameters, although this is in line with the results displayed in Figure 4.3a. Clearly the slow read performance of the Java target cannot be attributed to the initiator, otherwise the IET would not have been able to achieve read speeds close to the theoretical maximum for a single Gigabit Ethernet connection. Consequently, the fault must lie with the jSCSI Target.

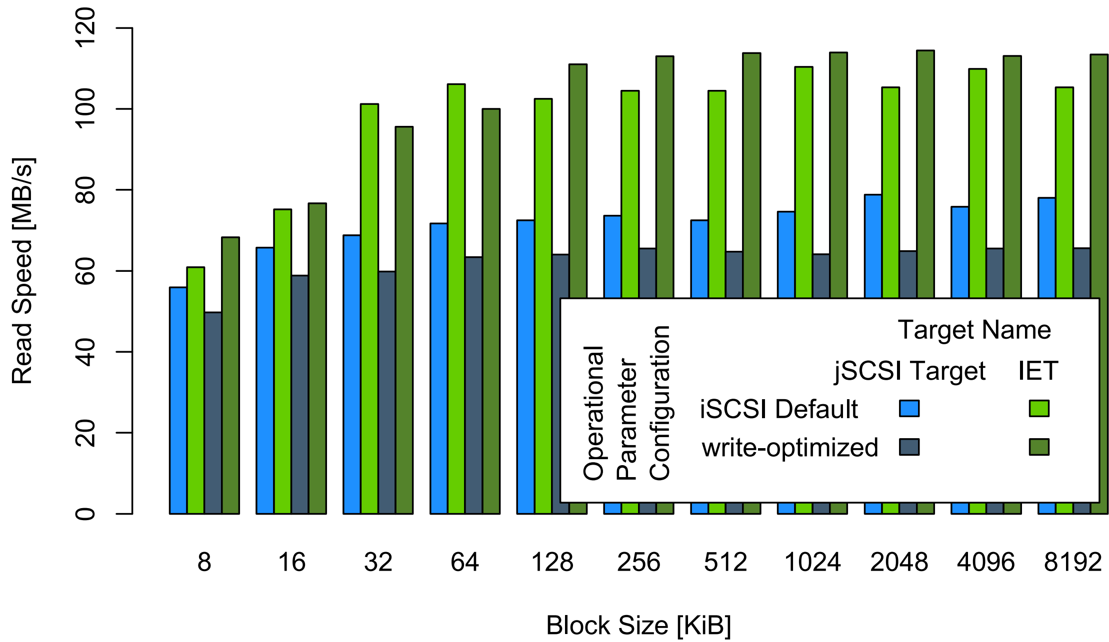
Since the read performance does not increase with burst length, which increases the comparative amount of Data-In PDUs sent during the read task, there seems to be a high processing cost associated with copying data to the buffer of a Data-In PDU and sending that message to the initiator. Determining which of these steps represents the bottleneck, would require a deeper investigation, however.

4.6 Effect of the Number of Sessions

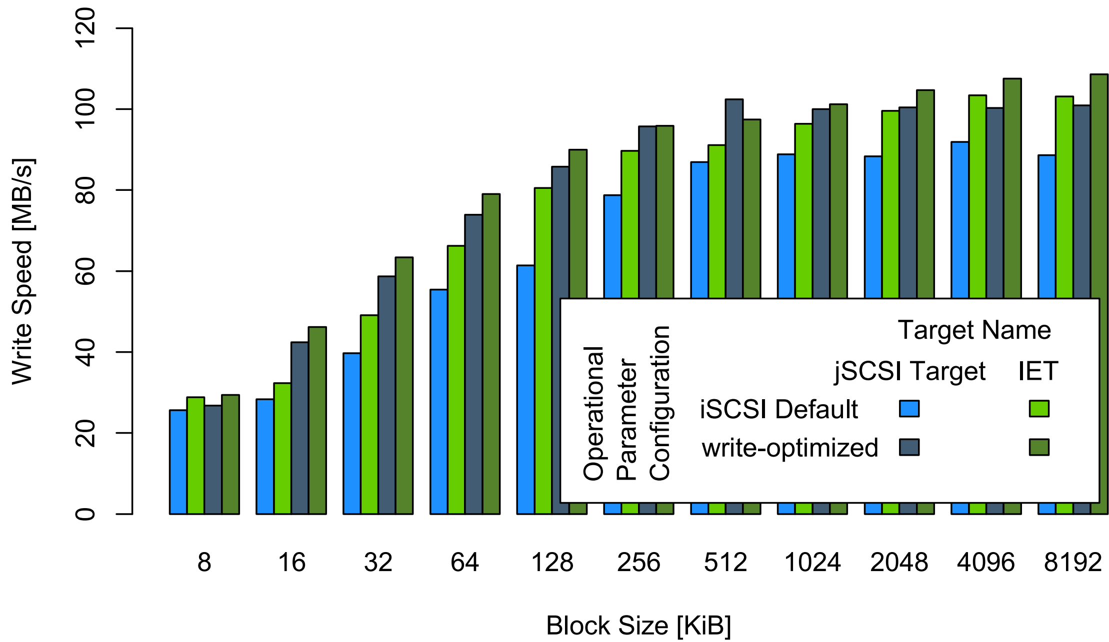
In order to measure the effect of multiple sessions, a different client setup was needed. Instead of using Open-iSCSI and `dd` together with a suitable multipath driver, a special Java initiator based on jSCSI 2.0 was used[2]. Up to 8 single-connection sessions were established. I/O operations were split up into chunks of mostly equal size, assigning one chunk to each session. The performance results are shown in Figure 4.6.

The results show that increasing the number of sessions can improve performance up to a certain point. But this is certainly not a universal solution, since the available processing power might impose a limit on the number of threads and consequently the number of sessions that can be used concurrently.

[2]That way, measured performance is guaranteed to reflect the capabilities of the target and not the limitations of the wedge driver.

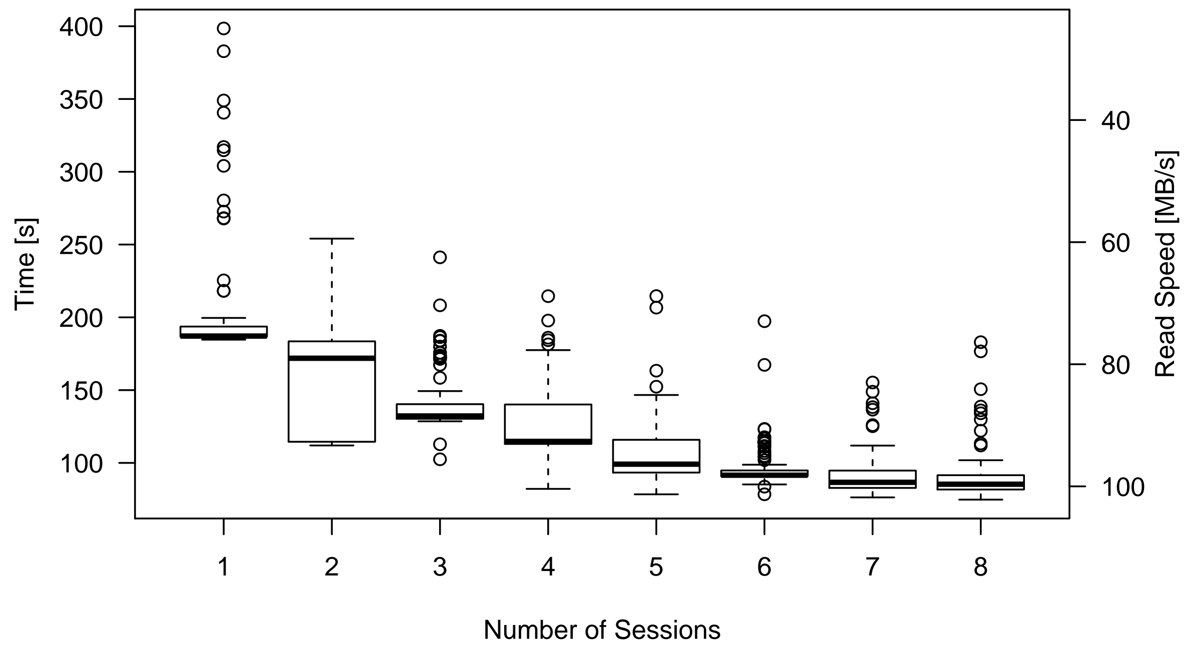


(a) Read Speed Comparison

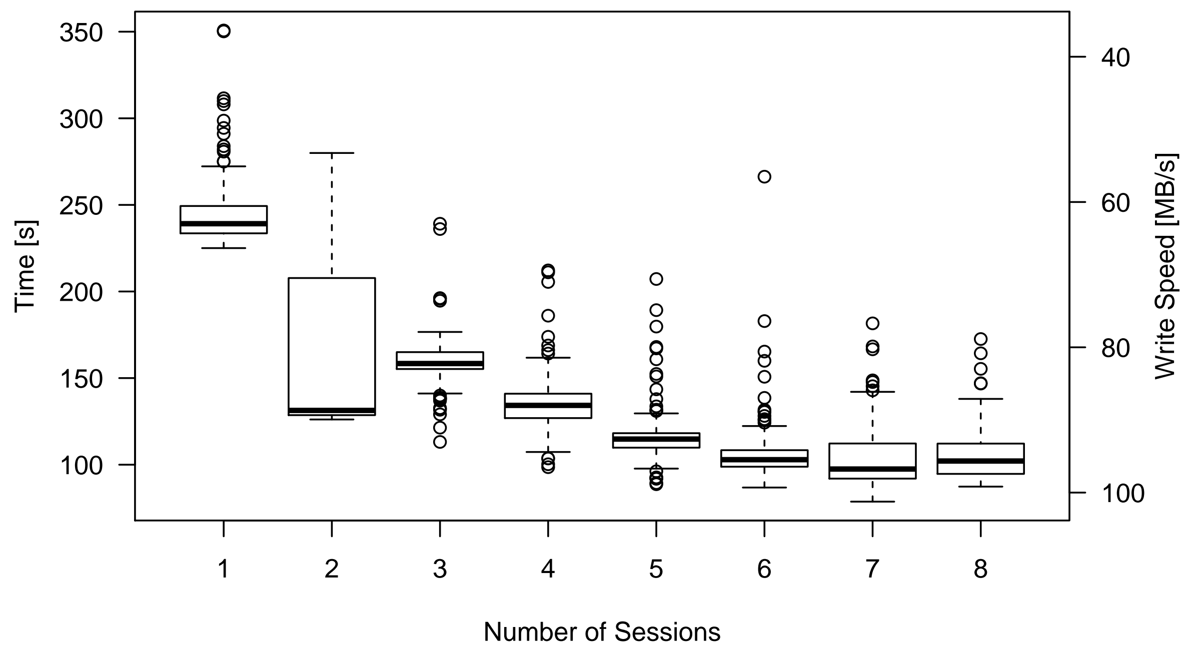


(b) Write Speed Comparison

Figure 4.5: Performance comparison of the jSCSI Target and the iSCSI Enterprise Target (IET) using the configurations from Table 4.2. Send buffer size: 64KiB, receive buffer size: 64KiB, count per block size category: 100.



(a) Read I/O



(b) Write I/O

Figure 4.6: Effect of the number of sessions on I/O times and speed, using the default parameter values from Table 4.2. Block size: 8MiB, count per block size: 100.

Chapter 5

Conclusion

Although minor performance issues exist, the jSCSI Target is proof that an iSCSI Target implementation in Java is feasible. Write speeds are comparable to those of native code implementations and even though read performance is clearly suffering from an as-yet-unexplained bottleneck, it should be fast enough for most real-world applications. Since the storage file will usually not be stored on a RAM-disk, disk I/O times will be the true limiting factor, anyway.

Many of the jSCSI Target's limitations are due to the simple architecture. A future implementation should be based on non-blocking I/O, with a single thread in charge of receiving, sending, and queueing of PDUs. Ideally, this would be combined with *Callable* tasks, that are executed in a thread pool shared by all sessions and connections.

My personal experiences developing the target have been quite frustrating at times. One the one hand I enjoyed diving into the depths of of SCSI hardware, on the other hand the final extent of the project kept growing and growing, resulting in a suboptimal architecture. I doubt, however, that there is a better way to learn about the intricacies of iSCSI than by finding out hands-on what does and what does not work.

Bibliography

[1] M. Chadalapaka. Internet Small Computer System Interface (iSCSI) Corrections and Clarifications. RFC 5048 (Proposed Standard), Oct. 2007.

[2] dd. http://pubs.opengroup.org/onlinepubs/9699919799/utilities/dd.html.

[3] /dev/null. http://en.wikipedia.org/wiki//dev/null.

[4] /dev/zero. http://en.wikipedia.org/wiki//dev/zero.

[5] iSCSI Enterprise Target. http://iscsitarget.sourceforge.net.

[6] iSCSI Target for the iPhone. http://itunes.apple.com/us/app/iscsi-target/id384389954?mt=8.

[7] J. L. Hufferd. *iSCSI: The Universal Storage Connection*. Addison-Wesley Professional, 2003.

[8] M. Krueger, M. Chadalapaka, and R. Elliott. T11 Network Address Authority (NAA) Naming Format for iSCSI Node Names. RFC 3980 (Proposed Standard), Feb. 2005.

[9] J. C. Mogul and G. Minshall. Rethinking the tcp nagle algorithm. *SIGCOMM Comput. Commun. Rev.*, 31(1):6–20, Jan. 2001.

[10] J. Nagle. Congestion Control in IP/TCP Internetworks. RFC 896, Jan. 1984.

[11] Open-iSCSI. http://www.open-iscsi.org.

[12] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. Internet Small Computer Systems Interface (iSCSI). RFC 3720 (Proposed Standard), Apr. 2004. Updated by RFCs 3980, 4850, 5048.

[13] SCSI Architecture Model 5 (SAM-5).

[14] Scsi block commands 3 (sbc-3).

[15] Scsi primary commands 3 (spc-3).

[16] V. Wildi. Java iSCSI Initiator. 2007.

[17] D. Wysochanski. Declarative Public Extension Key for Internet Small Computer Systems Interface (iSCSI) Node Architecture. RFC 4850 (Proposed Standard), Apr. 2007.