University of Alberta

BANDWIDTH REGULATION AND PERFORMANCE ENHANCEMENTS FOR OPEN-iSCSI NETWORKED STORAGE

by

Yongjian Zhang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Fall 2010
Edmonton, Alberta

Examining Committee

Mike MacGregor, Ph.D., Computing Science

Marek Reformat, Ph.D., Electrical & Computer Engineering

Janelle Harms, Ph.D., Computing Science

Abstract

Virtual machines are gaining a growing importance in modern business IT infrastructure. They facilitate multiple operating system instances on one physical host, which provides more efficient use of the computing power of the physical host but increases the amount of network traffic as well. To avoid potential network congestion and prioritize link resource usage in a virtual machine system, we propose a bandwidth regulation scheme. Extensive evaluation demonstrates that this bandwidth regulation scheme is accurate and effective. In addition, we resolved a drastic performance degradation of the Open-iSCSI initiator. We thoroughly tested the performance of the Open-iSCSI initiator and three modified versions under two methods of setting the TCP send buffer size - statically and dynamically. Based on these results, we propose a performance tuning scheme, which can enable users of Open-iSCSI, especially those using Open-iSCSI over a long fat network, to achieve significant throughput gains.

Acknowledgements

First of all, I would like to thank my supervisor Professor Mike MacGregor for being a great mentor. This thesis would not have been possible without his support, encouragement, help, and guidance.

I would also like to thank Precarn Inc. and TRLabs for their financial support and DataGardens Inc. for the opportunity to work on the HTB project and their assistance in this effort.

I am grateful to my colleague Zhu Pang, a wonderful developer, for his help in coding and debugging. I would also like to show my gratitude to Mike Christie from the Open-iSCSI Google group for his help with the Open-iSCSI source code.

It has been a great pleasure to work with Marcin Misiewicz and Eric Chalmers in TRLabs. Marcin offered tremendous help with my English and cultural questions and the random conversations between the three of us were of great joy.

My special thanks goes to Sapphire (Cailu) Zhao, a great friend, for always being there for me. Special thanks also goes to my friend Calvin Penny for his help with the text, his encouragement, and his faith in me.

I want to express my gratitude to Lan Liu, Ao Zhang, and Jinwen Liu. Adapting to a new life was not 100 percent smooth for me; their help from the other side of the globe certainly made it much easier. Now Lan Liu and her husband are starting their graduate school in the US and I would like to take this opportunity to wish them all the best in their study and research.

Final thanks to my parents, Guixia Yang and Xun Zhang, for their constant unconditional support.

Table of Contents

List of Tables

List of Figures

List of Algorithms

List of Listings

Chapter 1

Introduction

Live migration of virtual machines plays an increasingly crucial role in modern business IT infrastructure. Virtual machines, when not running, live as one or more static files on the physical host's storage subsystem. When started, virtual machines are loaded by the physical host from these files and presented to the user as if they were independent physical hosts. As more than one virtual machine can be loaded at the same time on the host, more than one operating system instance can run concurrently on one physical host, which allows for more efficient use of computing resources.

Also, as virtual machines are represented as data in files, when correctly replicated, virtual machines can be migrated to a different physical host and run in exactly the same manner as they do on the original physical host. The migration can even occur during the time when a virtual machine is running; this is called live migration. One crucial performance indicator of virtual machine live migration is the service down time, which is largely dependent on the migration time of the virtual storage subsystem. Pang presents a technique that facilitates more efficient disk migration [22]. This technique is able to reduce the service down time during virtual machine live migration and therefore improve business continuity.

As well as the benefits virtual machines bring to the IT infrastructure, they extend new challenges, of which an obvious one is the ever higher pressure they put on network resources: when multiple virtual machines are running on one physical host, the amount of network traffic of the physical host is multiplied accordingly; moreover, the migration of virtual machines just adds another significant amount on top of the multiplied traffic. In a distributed IT system, the overwhelming traffic from virtual machines can cause severe competition for link resources and even disastrous network congestion.

To avoid potential network congestion and prioritize link resource usage in a distributed virtual machine system, Sheng proposes an effective dynamic network resource allocation algorithm to calculate the optimal maximum allowed bandwidth for the physical hosts in the network[23].

Based on this calculated maximum throughput of the physical hosts, we realize the actual traffic throttling. We discuss the use of the native Linux traffic control tools, such as `tc` and HTB (Hierarchical Token Bucket) to regulate the network traffic rate and examine their practicality and

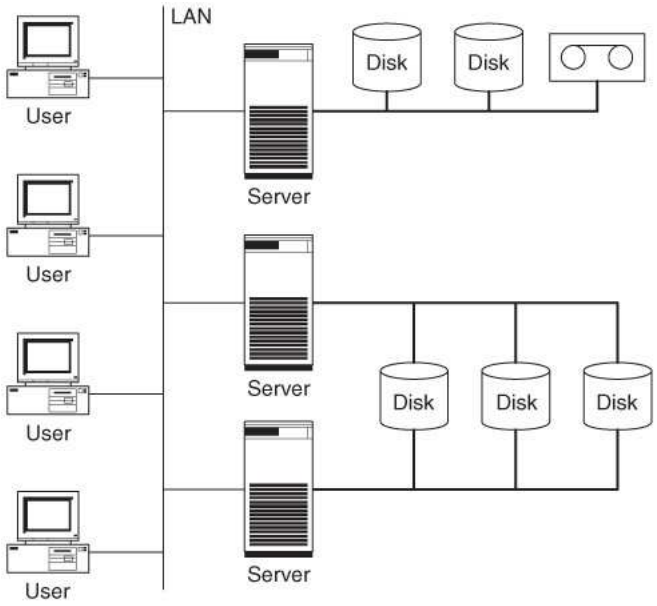


Figure 1.1: Server-Centric Architecture. [26]

effectiveness. In addition, we focus more specifically on the regulation of the iSCSI traffic in order to provide differentiated storage service for virtual machines. Chapter 3 documents this work. The positive results demonstrate that with careful configuration, native Linux traffic control tools combined with the Open-iSCSI initiator can effectively regulate the traffic rate of a physical host and provide differentiated quality of service for the disk traffic of virtual machines within the physical host.

1.1 iSCSI Overview

In this new era of information, a tough challenge has been extended for our conventional storage systems: According to Troppens et al. [26], the amount of data a normal company owns grows exponentially each year; a company with 1 TB of data will have to deal with 32 TB in five years. In addition to the constant generation of new data, the backup windows continue to shrink. These trends undoubtedly bring a strong challenge for the expansion of conventional server storage systems based on the SCSI bus, due to their limitation on cable length and the number of devices allowed on a daisy chain.

Moreover, data storage tends to be increasingly distributed geographically, yet the need for sharing data is ever growing. The decentralization of data storage and the converse demand of data sharing surely challenge the architecture of conventional server-centric IT systems, as depicted in Figure 1.1, in two ways [26]: Firstly, a multi-server daisy chain does not work effectively; therefore, it is difficult to share data efficiently between multiple servers through the SCSI bus; secondly, one server cannot use the free space on the storage attached to another server if they are not on the same

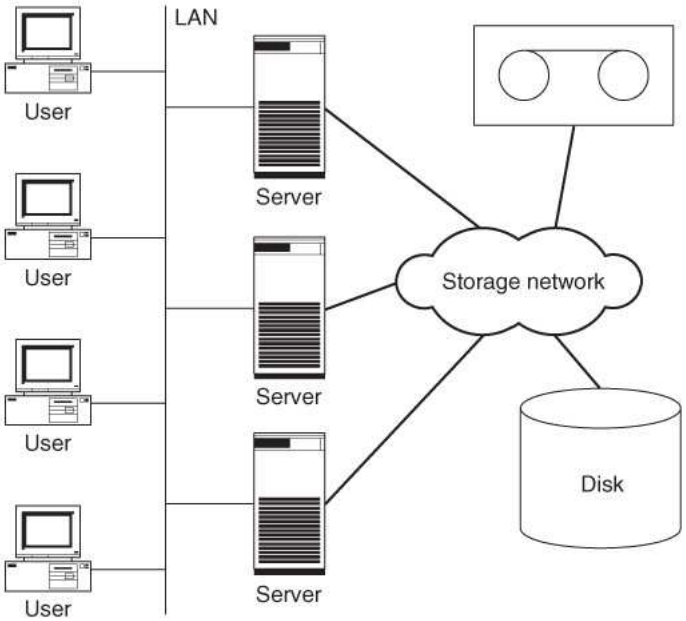


Figure 1.2: Storage-Centric Architecture. [26]

SCSI daisy chain, which causes inefficient use of storage resources.

To efficiently tame the drastic data growth and effectively meet the need to share data, modern IT architecture has gradually moved from the server-centric structure to a storage-centric structure, as shown in Figure 1.2. In a storage-centric IT system, the storage devices and servers are connected via network links, which are generally dedicated for storage access.

This network of storage devices is called *Storage Area Network* (SAN). SAN inherits the SCSI command set and replaces the SCSI bus by a network. As a result, the storage devices do not have to be directly attached to the servers, which facilitates flexible expansion of existing storage volumes. Also, multiple servers can now connect to the same storage devices, which enables storage consolidation and seamless data sharing. With the network properly built, storage sharing between geographically dispersed servers is made available as well.

Unfortunately, everything comes with a cost and it is no exception for SAN. SANs were first built upon the expensive Fibre Channel technology, which still dominates the majority of the SAN market at the time of writing. The protocol stack of a Fibre Channel SAN is shown in 1.3. Fibre Channel technology is fundamentally different than the widely used IP technology; therefore, to deploy Fibre Channel SAN, an entirely new Fibre Channel network infrastructure is required. While establishing the Fibre Channel network, the cost of necessary training and maintenance needs to be budgeted as well. The collective cost of the network infrastructure, personnel training as well as network maintenance inevitably sets a financial barrier for the deployment of Fibre Channel SAN in average companies and lower-end users as well.

To tackle the financial barrier of Fibre Channel SAN, Internet SCSI (iSCSI) was brought into

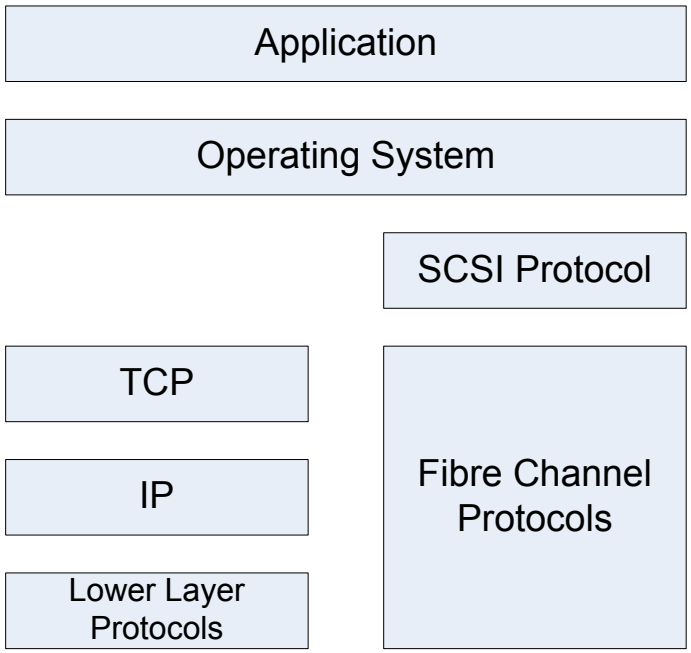


Figure 1.3: Fibre Channel Protocol and TCP/IP.

the picture. As shown in Figure 1.4, iSCSI replaces the expensive Fibre Channel network with an IP network, which is the existing network infrastructure for almost every company. This infrastructure is usually based on conventional Ethernet controllers (cards) and cables, which are much more affordable than their Fibre Channel equivalents and yet still meet the average and even higher end performance requirements. In addition, the staff training and network maintenance cost of IP networks is minimal compared to Fibre Channel networks. As a result, the total cost of ownership for iSCSI is much lower than for a Fibre Channel SAN. Furthermore, iSCSI is inherently TCP/IP based; hence it has no distance limits and can fit well in WAN based distributed systems, for which Fibre Channel SAN requires extra special hardware enhancements to function properly, adding even more cost to the entire IT system.

Clearly, iSCSI is in a more favorable position financially. The old saying, however, makes no exception for iSCSI either: The financial advantage comes with a cost, which, in this case, is in the form of potential performance concerns. iSCSI is based on TCP/IP and hence inherits the two most noticeable concerns in TCP/IP networks - packet loss and long delay over the network links.

To closely investigate how these two issues affect the performance of iSCSI, we started our work on the performance measurement of a software implementation of the Open-iSCSI initiator over lossy links and long delay links. Our results show that iSCSI is able to reach the same maximum performance as a general TCP application does on a lossy link. On a long delay link, however, the performance of Open-iSCSI initiator degrades drastically as the delay on the link increases, while the general TCP application is able to maintain its throughput. We then performed a thorough

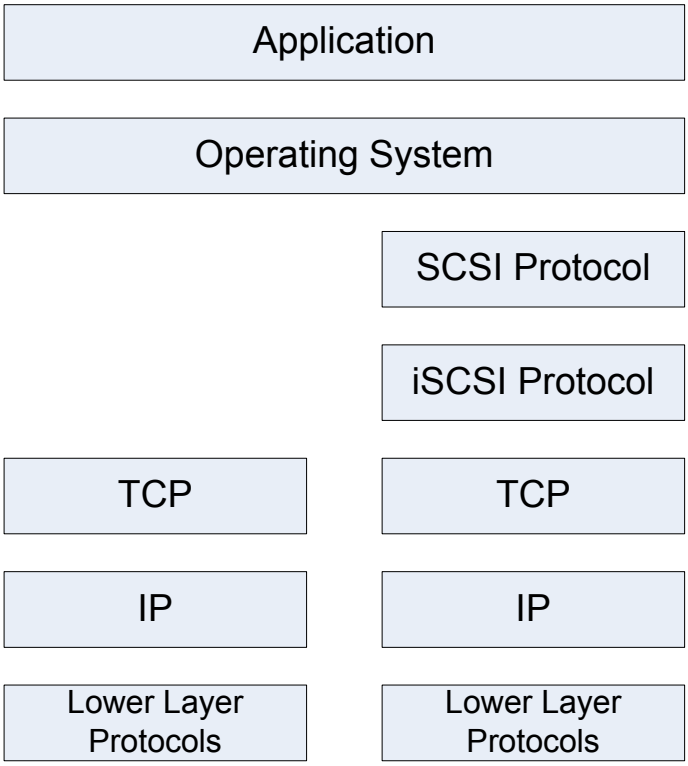


Figure 1.4: iSCSI and TCP/IP

examination of this problem and discovered two resolutions that gain substantial improvement for Open-iSCSI throughput. Chapter 4 documents this work and presents our resolutions to the iSCSI performance degradation problem. By the time of writing, we have not been able to find any other published solutions to this problem.

Chapter 2

Background and Previous Work

2.1 A Few Terms

For the purpose of clarification, we introduce here some terms used in the following sections.

Round-Trip Time (RTT) : The length of time it takes for a TCP segment to be sent plus the length of time it takes for an acknowledgment of that segment to be received. It is twice the end-to-end delay of the link.

Bandwidth Delay Product (BDP) : A link's BDP is calculated as the product of its bandwidth and RTT[1]. This product is the amount of data required to be on the wire to fill the pipe in order to fully utilize the capacity of the link; therefore, the BDP of a network link implies the proper size of the send and receive buffer of the applications communicating over the link.

Long Fat Pipe and Long Fat Network (LFN) : If the BDP of a link path significantly exceeds 10^5 bits, we call it a *long fat pipe*. And we call a network with a long fat pipe a *Long Fat Network (LFN)*[15].

2.2 Background on iSCSI

The iSCSI related protocol stack is illustrated in Figure 2.1. iSCSI works between the SCSI layer and the TCP layer. The I/O requests from applications first go through the SCSI layer, where the SCSI driver builds SCSI Command Description Blocks (CDBs) based on the I/O requests. The SCSI CDBs then move to the iSCSI layer and the iSCSI driver assembles Protocol Data Units (PDUs) and then hands them over to the TCP/IP layer. Thereafter, the PDUs travel across the network just like any other TCP payload. At the other end of the network, the iSCSI driver disassembles the received PDUs and passes the encapsulated CDBs to the SCSI layer. The SCSI driver then disassembles the CDBs and the I/O requests are eventually performed on the SCSI Logical Unit(s).

[1] Sometimes it is also calculated as the product of the link's bandwidth and end-to-end time. In this thesis, when referring to BDP, we always use round-trip time instead of end-to-end time.

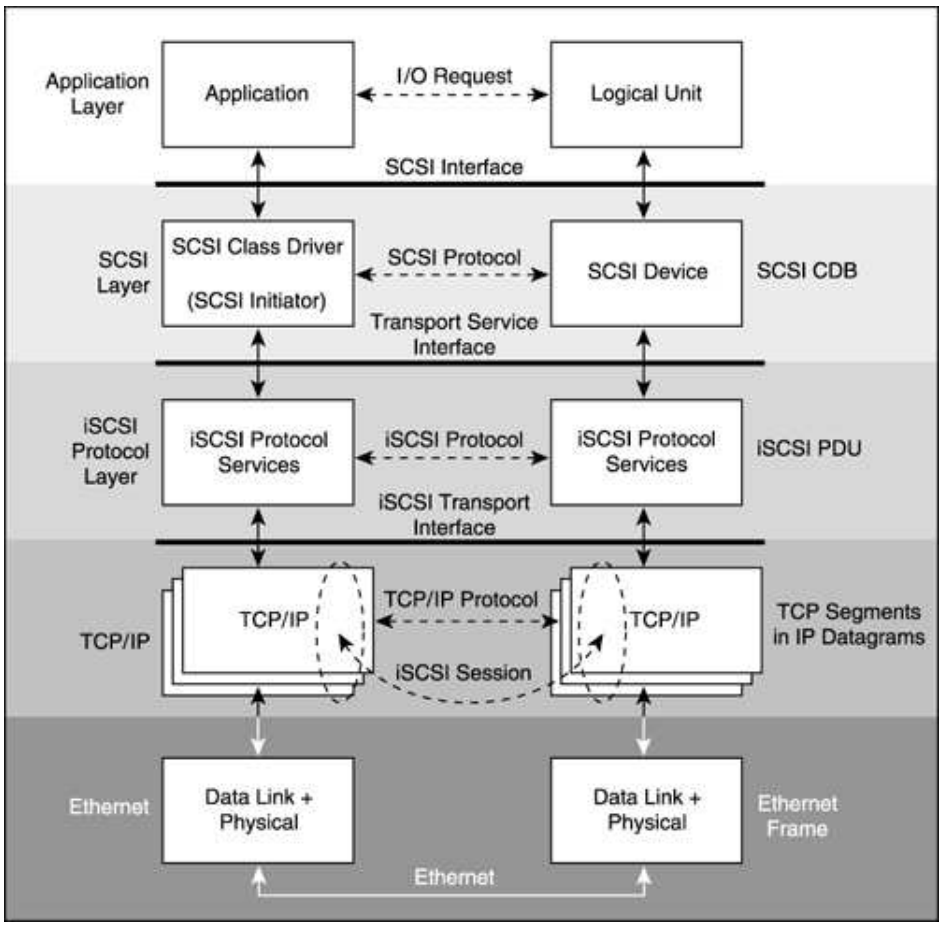


Figure 2.1: The iSCSI Protocol Stack. [14]

With TCP/IP, iSCSI delivers a ubiquitous interconnection to storage that the market has been long waiting for [14]. This interconnection undoubtedly provides new opportunities for data backup and storage consolidation. At the same time, iSCSI manages to keep the Total Cost of Ownership (TCO) low because the entire existing network infrastructure can be directly utilized by iSCSI.

As mentioned in Section 1.1, iSCSI suffers from an imperfect network environment, such as WAN. To achieve a better understanding of these issues and improve the iSCSI performance over imperfect networks, we chose a well maintained open source software iSCSI initiator-target pair, the Open-iSCSI initiator and the iSCSI Enterprise Target (IET), and extensively examined the iSCSI initiator throughput over links with emulated WAN-like RTT and packet loss. The results (Chapter 4) show that the Open-iSCSI implementation reacts well to pack loss but suffers severely from long delay over the links. Based on our analysis, we propose a possible explanation for this performance degradation.

2.3 Previous Work on iSCSI Performance

Lu et al.[18] have done a thorough simulation study of a iSCSI-based storage system with the network simulator ns-2[2]. Their study covers the performance variance of iSCSI in a simulation setting while varying PDU size, TCP Maximum Segment Size (MSS)[3], and TCP window size. The simulation model they built facilitates the exploration of the iSCSI performance with adjustable parameters and even different scheduling algorithms. Our results on iSCSI performance degradation over long delay links (presented in Chapter 4) conform to the results of their simulation with varying TCP window sizes.

Gauger et al. [12] also performed a simulation study on iSCSI performance. Their study analyzes the iSCSI throughput and total request write times over varying link RTTs, link loss probabilities, process delays in the iSCSI layer, as well as I/O request characteristics. One major contribution of their paper is that their simulations are based on statistically realistic network and I/O request models. Although the maximum RTT studied in their model is only 10 ms, it is sufficiently descriptive as they set the link bandwidth to 1 Gbps. As for the results, our observation of iSCSI performance degradation over long delay links (presented in Chapter 4) also conforms to their results in the scenario with a single iSCSI session over long RTT links.

As the work of both Lu et al. and Gauger et al. is performed with simulation, they did not consider practical factors that may affect actual performance, such as the buffering effect of file systems. Also, their study is observation oriented; they did not provide any practical approach to address the observed iSCSI performance degradation.

Aiken et al. [3] performed real measurements of iSCSI performance. They first measured the performance of two commercial iSCSI initiators, a software one and a hardware one, against a hardware iSCSI target. Then they compared these performance results against a storage subsystem based on Fibre Channel, yielding the conclusion that the commercial software iSCSI initiator outperforms the hardware initiator and the software initiator compares quite "favorably" to Fibre Channel. Next, they measured the performance of Intel open source iSCSI initiator[4] and target pair over a Storage Area Network (SAN) setting and a Wide Area Network (WAN) setting (emulated by an external router) as well. Our results in Chapter 4 conform to their WAN results and we both noticed the importance of network tuning for iSCSI to achieve best performance over WAN. However, Aiken et al. did not attempt to explore any of the actual network tuning options in their paper.

Bianco et al [7] used the same iSCSI implementation as we do in Chapter 4. They performed tests including RAID[5] 0 with iSCSI over SAN and iSCSI without RAID over WAN. Their results demonstrate that iSCSI can utilize the data striping feature of RAID 0 to multiply its performance

[2] Please refer to `http://www.isi.edu/nsnam/ns/` for more details about the network simulator ns-2.

[3] Please refer to `http://en.wikipedia.org/wiki/Maximum_segment_size` for more details on TCP maximum segment size.

[4] Please note that this is *not* the Open-iSCSI initiator we used in Chapter 4.

[5] Redundant Array of Independent Disks. Please refer to `http://en.wikipedia.org/wiki/RAID` for more details.

in the SAN setting. The results in the WAN setting, not surprisingly, show the identical limitation as we observed in our study - the 256 KB send buffer size (Please refer to Section 4.2.2 for more details.). In their paper, however, they did not further investigate the cause of this performance issue.

2.4 Previous Work on TCP Tuning

Lawrence Berkeley National Laboratory [17], Mathis et al. from PSU [20] and Tierney [25] all clearly point out that it is necessary to tune the TCP setting if the network link is considered as an LFN (Long Fat Network, please refer to Section 2.1 for more details.), as the current (2010, up to Linux kernel 2.6.32) Linux operating system does not set the default values for relevant parameters to best perform on LFN and the default values may result in poor throughput over LFN. All three works provide similar recipes of tuning due to the same reason behind the poor throughput over LFN - insufficient TCP buffer size. This was the starting point for our examination of the differences between static and dynamic tuning (Please refer to Chapter 4 for more details). We believe of the three sources, Lawrence Berkeley National Laboratory provides the most up-to-date information, which is still continuously updated. In our research, we found that both the static and dynamic methods, if applied properly, can help tune TCP to better carry iSCSI traffic. An Open-iSCSI session with the buffer size statically set outperforms one with TCP autotuning when the RTT is less than or equal to 40 ms. TCP autotuning starts to show its advantage as the RTT further increases. However, on links with very long RTTs, we found that TCP autotuning needs extra tuning itself to achieve the desired performance.

2.5 Traffic Scheduling with Round-Robin Algorithms

Shreedhar et al. [24] propose an efficient fair queueing algorithm named Deficit Round Robin (DRR), which is used in the HTB implementation [10]. The DRR algorithm first assigns newly-arrived packets of different flows to corresponding queues and then services these queues in a round robin manner to dequeue packets from each queue up to the size of the quantum the queue has for the current round.

Although DRR achieves fair queueing, it cannot satisfy the delay requirement of latency-critical flows. Shreedhar et al. also propose a revised version of DRR, named DRR+, to deal with this kind of flows. DRR+ maintains a different set of queues for latency critical flows to guarantee their delay bound. At the same time, DRR+ signs a contract with the latency-critical flows about the amount of data they can send in one service round. Whenever a flow fails to respect its contract, it will no longer be treated as latency-critical.

However, DRR+ still does not service latency-critical flows adequately due to the bursty feature of real-life network traffic, as pointed out by MacGregor et al. [19]. As an improvement to DRR+, they propose another scheduling algorithm, named DRR++.

DRR++ improves DRR+ by not removing a latency-critical flow from its service class when the flow violates its contract. Instead, DRR++ only stops dequeuing from that flow for the current service round and resumes the flow's latency-critical service in the next service round. According to the test results of Zhang et al. [28], DRR++ significantly improves DRR+ in terms of the delay of latency-critical traffic while still preserving the fairness provided by DRR+.

In this thesis, we documented the implementation of HTB in Chapter 3, which is based on DRR. In the future work, we would like to explore the possibility of implementing HTB using DRR++.

Chapter 3

Linux Traffic Control and HTB

In this chapter, we give the details of the mechanism and implementation of the hierarchical token bucket (HTB) method of traffic control. This is the method we used in our work, and while we found it very effective, we also found that documentation of the implementation was somewhat incomplete and opaque. This chapter is an attempt to provide solid documentation for others working in the area, as well as to explain how we used HTB in our work.

3.1 Linux Traffic Control

Linux traffic control has been provided as part of Quality of Service (QoS) support since Linux 2.2 kernel. It fits into the big picture of network data processing as illustrated in Figure 3.1 .

When a packet is enqueued to a Network Interface Controller (NIC) or the kernel decides to dequeue a packet from a NIC (`dev_queue_xmit` in `net/core/dev.c`), the traffic control component, if applied to that device, is invoked and carries out the specified network traffic regulation task. The traffic control subsystem is comprised of the following four main conceptual components:

- queuing disciplines
- classes (within a queuing discipline)
- filters
- policing

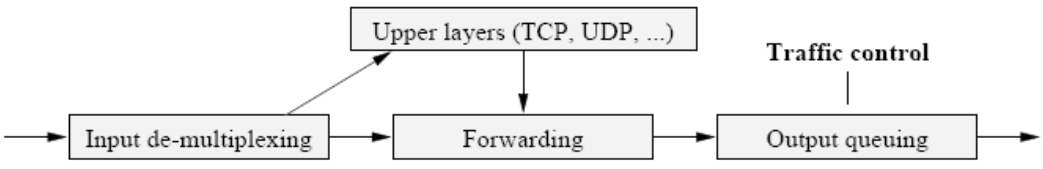


Figure 3.1: The Processing of Network Data in Linux. [5]

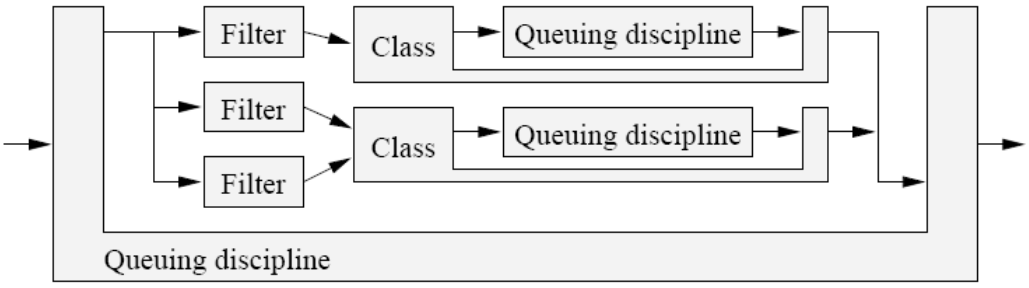


Figure 3.2: A Queuing Discipline with Multiple Classes. [5]

A *queuing discipline* can be considered as just a queue, where packets come and go. It is the heart of the traffic control system, as it implements how packets enter the queue and in which order the packets leave. It can facilitate packet buffering, reordering, throttling, dropping and classifying, plus all of the above to each individual class of the classified network traffic. Queuing disciplines provide great controllability as well as immense flexibility. Without any queuing discipline, the NIC is just like a space-limited FIFO for packets - packets come and leave in the same order, if not dropped due to the overflow of the FIFO buffer.

Not all queuing disciplines support traffic classification. The ones that do allow classes are called *classful* queuing disciplines. Figure 3.2 illustrates a classful queuing discipline with three *filters* and two classes. The major role of these filters is to assign incoming packets to one of the classes or attached *policing actions* (not shown), such as dropping a packet. An interesting and useful fact about classful queuing disciplines is that more queuing disciplines can be further attached to the classes inside, one new queuing discipline for one class, and the new inner queuing disciplines do not even have to be the same as the outer one. Moreover, a new queuing discipline may contain more classes, which means "newer" disciplines can be even further concatenated; therefore, a highly sophisticated traffic control scheme can be implemented through a traffic control chain. [5] [8]

HTB, or *Hierarchical Token Bucket*, is an example of a classful queuing discipline. In Section 3.2, we will explain the concepts of queuing discipline and class within the context of HTB. As we are only concerned about the traffic regulation part of HTB, filters and policing are out of the scope of this thesis. Please refer to Brown et al. [8] for more details.

3.2 Hierarchical Token Bucket

Hierarchical Token Bucket, or HTB, is one of the classful queuing disciplines within the Linux traffic control subsystem. As the name implies, HTB is based on the token bucket theory; it builds a tree structure of token buckets to exert sophisticated control on outbound traffic flows. Each of these token buckets is considered as a class in traffic control terms.

Figure 3.3 illustrates the structure of a simple HTB policy. Each circle in the graph represents one class, which can contain either multiple child classes or a single child queuing discipline. The

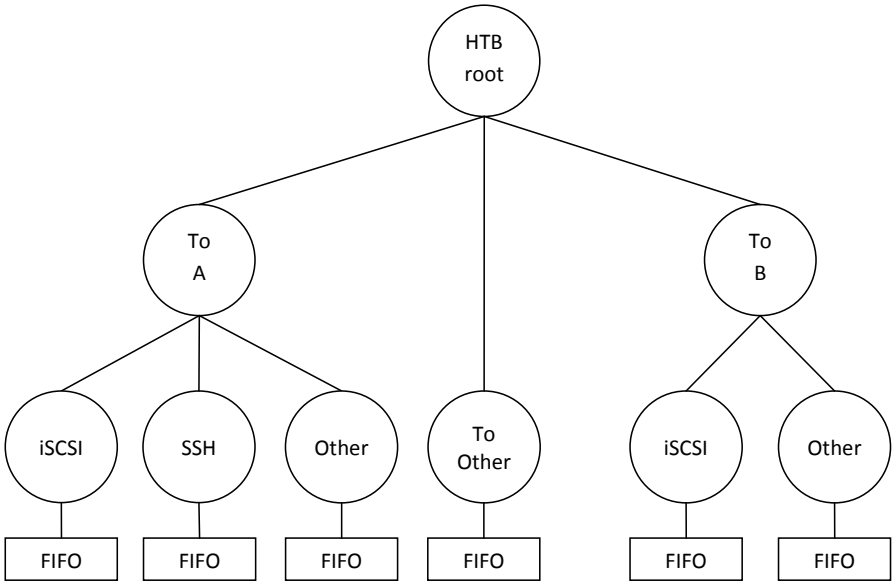


Figure 3.3: An Example of a HTB Queuing Discipline Structure.

classes on the second level (counting from the HTB root and thus including the "To Other" class on the leaf level) break down the network traffic to three categories based on its destination; the classes on the leaf level associate the traffic with the same destination into different categories according to the nature of the traffic (recognized by their destination TCP port). For leaf classes, a queuing discipline must be attached. If not, `tc` will have a default queuing discipline attached. All the packets entering HTB are assigned to one of the leaf classes and eventually dequeued from the attached queuing discipline.

This hierarchy of classes allows HTB to perform control over network traffic on various levels, which presents immense flexibility for the traffic control in complicated scenarios.

3.2.1 The Class in HTB and Bandwidth Regulation

As mentioned before, HTB is a hierarchical structure of classes. Each class represents one certain type of network traffic. We can regulate the speed[1] of a traffic class by using the set of parameters HTB has to offer.

The two most important parameters are `rate` and `ceil` (standing for ceiling). The `rate` parameter indicates the assigned traffic speed of the class, i.e., the class is definitely allowed to send packets at that speed. `rate`, however, does *not* serve as the upper limit of the traffic speed. If a class has more than `rate` to send, it can borrow "`rate`" from its parent, provided that the parent class has some `rate` available to lend. On the other hand, `ceil` specifies the maximum traffic speed, and it is a hard limit that cannot be exceeded, which means that when a class reaches the `ceil` limit, it

[1]Here we choose "traffic speed" over "traffic rate" because "rate" may cause confusion, as it is also the name of a parameter within HTB. When we refer to this parameter, we use a monospace font like this: `rate`.

cannot borrow `rate` from its parent even if the parent has `rate` available for lending. Thus, `ceil` should be at least as high as the `rate` value of the class and at least as high as the highest `ceil` value of all the child classes.

`rate` and `ceil` provide explicit indications for bandwidth regulation. To enforce these indications, the classic *token bucket theory* is utilized to actually limit the traffic speed. In this theory, the bucket is an imaginary container holding tokens, each of which represents a certain amount of data that can be transmitted. The bucket also has a depth, which indicates how many tokens, if not used immediately after generated, can be buffered in the bucket.

According to Kuznetsov et al. [16], if we denote the assigned traffic rate and the bucket depth by R (in bits/second) and D (in bits) respectively, for any time interval between t_i, ..., t_k $(k > i)$ (in seconds), the amount of transmitted bits cannot exceed

$$D + R \times (t_k - t_i).$$

Considering that the transmitted bits are actually sent out in a series of packets, if we denote the sizes of these packets by S_m, ..., S_n $(n > m)$ (in bits), we get a formalized version of the token bucket theory, with the last item denoting the number of tokens generated (also the amount of data allowed to be sent in bits) [16]:

$$S_m + S_{m+1} + \cdots + S_{n-1} + S_n \leq D + R \times (t_k - t_i) \tag{3.1}$$

According to Eq. 3.1, if we can generate tokens at the rate R, when a packet arrives for transmitting, either we deduct the corresponding amount of tokens from the bucket if there are enough tokens, or we put the packet on hold until enough tokens are generated. In practice, however, this algorithm is implemented slightly differently [16]: The comparison between the packet size and the amount of available tokens is converted to the comparison between two times. For the time related to tokens, we denote $\frac{D}{R}$ by $N(t_i)$, and $N(t)$ grows linearly over time before the bucket is completely filled, that is:

$$N(t + \Delta t) = min\left\{\frac{D}{R}, N(t_i) + \Delta t\right\}. \tag{3.2}$$

$N(t)$ here gives the time allowed by available tokens for transmitting at full rate R. When a packet of size S arrives for transmission, it can be emitted to the network only at the time t_e when all the previous arrived packets are sent and

$$\frac{S}{R} \leq N(t_e). \tag{3.3}$$

Eq. 3.3 basically means that the amount of time to transmit the incoming packet should not exceed the available amount time for transmitting. If it is not satisfied, the packet has to wait until $N(t_e)$ grows large enough. Eq. 3.3 describes the core algorithm of the implementation of the token bucket theory. After a packet is emitted to the network, $N(t_e)$ jumps [16][2] and $N(t_e + 0)$ is used for

[2]Note that this $N(t)$ jump happens too when the previous arrived packets are emitted to the network.

subsequent packets:

$$N(t_e + 0) = N(t_e - 0) - \frac{S}{R}. \tag{3.4}$$

When a class reaches its `rate` limit but still has more to send, it may borrow tokens from its parent, provided the class has not exceeded its `ceil` limit and the parent class has tokens available. One may wonder what will happen if more than one child class attempts to borrow from its parent class. The solution provided by HTB is fairly elegant: The available tokens from the parent are distributed between the child classes according to the ratio of their `rate` values but no more than limited by their `ceil` values. This borrowing and ceiling mechanism allows the classes in HTB to make a friendly share and a full utilization of all the available bandwidth [10].

3.3 The Implementation of HTB

To explain the implementation details of HTB, we add the relevant concepts about the HTB policy (Figure 3.3) we talked about before, as shown in Figure 3.4. In this section, we gradually cover all the newly added elements.

3.3.1 Essential Concepts in HTB Algorithm - Level, Priority, and Mode

This HTB policy shown in Figure 3.4 has classes on three *level*s, as delimited by the dashed lines. The classes on the leaf level have the level number (`struct htb_class::level`[3]) 0, and they each contain a single queuing discipline as their children, from which packets are ultimately dequeued. Then from bottom to top, the level numbers of the inner classes increase by one as their level rises, with an exception that the top level, i.e. the HTB root class, has as its level number the maximum number of levels HTB can hold (`TC_HTB_MAXDEPTH`, which is 8 in HTB 3.0) minus one.

HTB allows classes to have different priorities (`struct htb_class::prio`). That is, classes with a higher priority (lower `prio` value) are dequeued before the ones with a lower priority (higher `prio` value).

As a class is dequeued, the *mode* of the class may change over time. The mode (`struct htb_class::cmode`) of a class is represented by three colours in the implementation: red, yellow and green. Before introducing how this colour system works, we need to recall two variables aforementioned to help understanding: `rate` (`struct htb_class::rate`) and `ceil` (`struct htb_class::ceil`) (Please refer to Section 3.2.1 for more details). They are used to indicate the assigned and the absolute rate limit of a class respectively. If we consider the actual traffic rate of class c, denoted by $R(c)$ of a class as a real continuum, then `rate` and `ceil` divide the continuum into three parts, and thus the three colours:

[3] Implementation specific function and variable names are put in between parenthesis to relate the concepts to the source code. This idea is originally from [9].

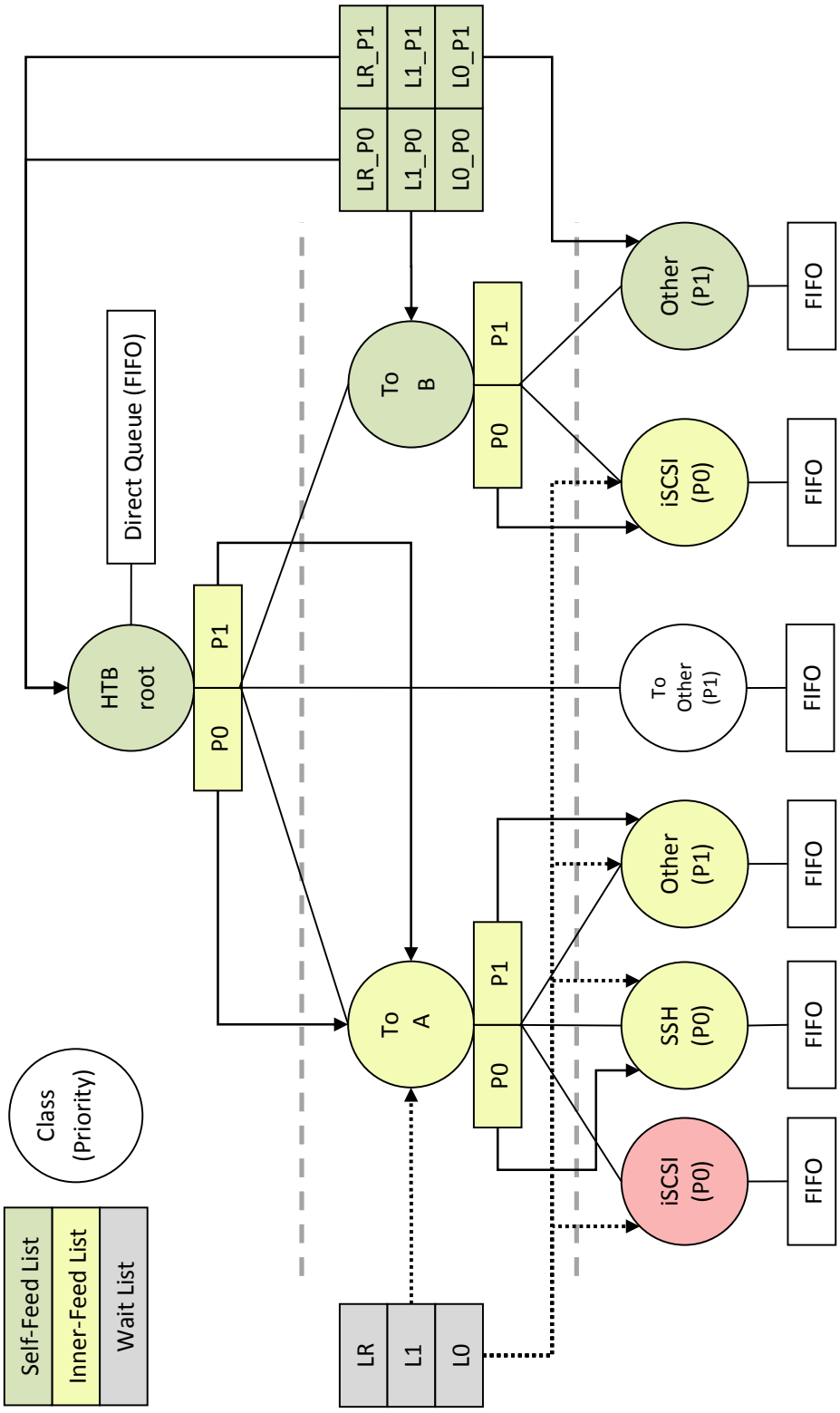


Figure 3.4: HTB Queuing Discipline Structure with More Details.

$$mode(c) = \begin{cases} Red & \text{if } R(c) > \texttt{ceil} \\ Yellow & \text{if } \texttt{rate} < R(c) \leq \texttt{ceil} \\ Green & \text{otherwise} \end{cases} \tag{3.5}$$

3.3.2 Self-feed List, Inner-feed List, Wait List and Direct Queue

As stated in Section 3.2.1, all the packets are ultimately dequeued from the queue disciplines attached to the leaf classes; when a leaf class reaches its `rate` limit, it is allowed to borrow "`rate`" from its parent class. That is, when borrowing occurs, the packet is dequeued using the tokens belonging to its parent; therefore, in this case, it is equivalent to think that the packets are being dequeued from the leaf class and all the parent classes lending the tokens. Consequently, packets can be "conceptually" dequeued from any level despite being held on the leaf level.

To facilitate the dequeue and borrow process, HTB maintains three data structures: *self-feed list* (`struct htb_sched::row`), *inner-feed list* (`struct htb_class::feed`) and *wait list* (`struct htb_sched::wait_pq`). They respectively store the *heads* of three types of class lists. Self-feed list is a global (as in one HTB policy) two dimensional array across all the levels and priorities. It stores the heads of the lists that hold *active* classes belonging to a particular priority on a particular level. Self-feed list is so called because all the classes linked to this list are dequeuing packets without borrowing tokens from their parents, i.e. these classes are self-sufficient in terms of token supply. When a class has to borrow from its parent, it is removed from the self-feed list and inserted to a class queue linked to the inner-feed list of that parent class; therefore, inner-feed list stores the heads of the list of classes that are borrowing tokens. Since each class queue contains borrowing child classes with a particular priority, the inner-feed list organizes these list heads in a one dimensional array across all the possible priorities supported by the HTB policy. The classes in the inner-feed list queue are also in a queue linked to the wait list on its own level. This wait list is also a global data structure within one HTB policy. It is a one dimensional array across all the levels within the HTB policy.

Figure 3.4 illustrates the concepts of self-feed list and inner-feed list. The use of these three data structures in the HTB implementation will be covered in more detail in subsequent sections.

In addition to these three lists, HTB maintains two other data structures to store the current class to dequeue in each individual class queue. We call these two data structures *current-class list*s. One of them is for the self-feed list (`struct htb_sched::ptr`) and the other one is for inner-feed list (`struct htb_class::un.inner.ptr`).

Although self-feed list and inner-feed lists give us a hold on all the classes within one HTB queuing discipline, they do not cover all the possible places an incoming packet can go: There is a queue within HTB but not related to any of the classes - the *direct queue*. The direct queue is actually just an ordinary FIFO queue directly attached to the queuing discipline, which keeps the packets that are either fast matched to this direct queue (this will be covered in more details in Section 3.3.3.) or

not able to fit in any existing class (this should not happen in a well designed HTB policy).

3.3.3 The Enqueue Process of HTB

The challenge of the enqueue process lies in finding the correct leaf class for the incoming packet, which is handled by `htb_classify()`. `htb_classify()` first does a fast match for the incoming packet - by trying to match the `skb->priority` to an existing (leaf) class id or the HTB queuing discipline handle. On success, the matched class or the direct queue of the HTB queuing discipline will be chosen accordingly. If the fast match does not yield any valid result, `htb_classify()` will try to apply the `tc` filters to find the right spot for the packet. If a leaf class is found, it is chosen. Otherwise, `htb_classify()` will try the *default class* of the HTB queuing discipline, which should be a leaf class specified when the HTB policy is created. If the default class is not properly set up, `htb_classify()` will use the direct queue as a last resort.

Please note that so far we have only chosen a class or the direct queue for the incoming packet, i.e. the packet has not been put in there yet. To proceed, if the direct queue is selected, `htb_enqueue()` will check if there is still space in the queue for the incoming packet. If so, the packet will be enqueued there; otherwise, the packet will be dropped. If a leaf class is selected instead of the direct queue, the corresponding enqueue function of the queuing discipline will be called to try to include the packet in that queuing discipline. On success, `htb_enqueue()` will activate (`htb_activate()`) this leaf class, by adding it to the corresponding active class queue, if it is not in there yet; otherwise, the packet will be dropped.

The enqueue process ends here.

3.3.4 The Dequeue Process of HTB

To achieve a fair dequeue process, HTB implements a Weighted Round -Robin (WRR) algorithm [24] for each priority on each level - the weight of each class is proportional to its `rate` value; Classes with a lower priority are only dequeued after the ones with higher priorities are served.

The implementation of the dequeue process of HTB is actually more complicated than the enqueue process because it contains the algorithm to do WRR within each priority and implements the lending-borrowing mechanism. Pseudo-code for the HTB dequeue implementation is given in Algorithm 3.1. Please note that this pseudo code does not cover the error handling part in the HTB dequeue source code.

When `htb_dequeue()` is called, it tries to dequeue the direct queue first. If no packets are pending there, it will try to loop through the self-feed list (Section 3.3.2) starting from the highest priority on the leaf level to find a class to dequeue from. As explained in Section 3.3.2, each element in the self-feed list is the head of a queue of classes with a certain priority on a certain level and the corresponding element in the current-class list is keeping which class in this queue to dequeue next. These two data structures provide sufficient information for the class look-up process.

When a class is chosen to dequeue from, HTB will next look for a leaf class associated with the chosen class. If the chosen class is already a leaf class, the associated leaf class will of course be itself. Otherwise, HTB will consult the inner-feed class hierarchy to look for a leaf class that is borrowing from this chosen class. If no valid leaf class can be traced down through the class queue with the current priority, the loop will move on to the next iteration (lower priority). The same rule applies to the outer loop through levels: the loop will move up a level if no valid leaf class can be found on the current level.

After a valid leaf class is found, a packet will be dequeued from that leaf class. Each class can dequeue up to quantum bytes before HTB moves to the next class in the same queue, provided the class does have one or more packets to send. The size of the quantum value of each class is by default one tenth of the amount the bytes the class is allowed to send as indicated by the `rate` parameter. It can be assigned independently (and manually) when creating the HTB queuing discipline.

Algorithm 3.1 HTB Dequeue Process - Conceptual Illustration.

if *NotEmpty(direct_queue)* **then**
  return $DequeueOnePacket(direct\_queue)$
end if

$ProcessTheWaitList$ {This will be covered in more detail later.}

$Packet \leftarrow NULL$
$Level \leftarrow 0$ {Dequeue from the leaf level.}
repeat
  $Priority \leftarrow 0$ {Dequeue from the highest priority.}

  repeat
    $CurrentDequeueClass \leftarrow current\_class\_list[Level][Priority]$
    $AssociatedLeafClass \leftarrow FindLeaf(CurrentDequeueClass)$
    $OldAssociatedLeafClass \leftarrow AssociatedLeafClass$

    repeat
      if *IsValid(AssociatedLeafClass)* **then**
        $Packet \leftarrow DequeueOnePacket(AssociatedLeafClass)$
        goto $Done$
      else
        $current\_class\_list[Level][Priority] \leftarrow \ldots$
        $\ldots NextClass(self\_feed\_list[Level][Priority], current\_class\_list[Level][Priority])$

        $CurrentDequeueClass \leftarrow current\_class\_list[Level][Priority]$
        $AssociatedLeafClass \leftarrow FindLeaf(CurrentDequeueClass)$
      end if
    until *AssociatedLeafClass == OldAssociatedLeafClass*

    $Priority \leftarrow Priority + 1$
  until *Priority* $\geq$ *HTBMAXPRIO or a packet to dequeue is found*

  $Level \leftarrow Level + 1$
until *Level* $\geq$ *HTBMAXDEPTH or a packet to dequeue is found*

$Done$:
if *Packet* $\neq$ *NULL* **then**

  {Weighed round-robin}
  $AssociatedLeafClass.Deficit[Level] \leftarrow AssociatedLeafClass.Deficit[Level] - sizeof(Packet)$

  if *AssociatedLeafClass.Deficit[Level]* $<$ *0* **then**
    $AssociatedLeafClass.Deficit[Level] \leftarrow AssociatedLeafClass.Deficit[Level] + AssociatedLeafClass.Quantum$
    $current\_class\_list[Level][Priority] \leftarrow \ldots$
    $\ldots NextClass(self\_feed\_list[Level][Priority], current\_class\_list[Level][Priority])$
  end if

  $UpdateClassStats(AssociatedLeafClass, Level, Packet)$
  $ChangeClassModeIfNecessary(AssociatedLeafClass, Level, Packet)$
end if

return $Packet$

3.4 The Results of iSCSI Traffic Regulation with HTB

3.4.1 The Test Scheme

We explored the option of using tcng [6] as a `tc` script generator in this set of experiments, as tcng provides a more human-friendly syntax. A tcng script to configure the HTB policy shown in Figure 3.5 is provided in Listing 3.1. For all the test results, an average value of 5 runs is used unless otherwise notified.

```
#include "fields.tc"
#include "ports.tc"

#define ISCSI_IF eth0

dev ISCSI_IF {
    egress {
           class (<$iscsi_1>) if tcp_sport == 53291;
           class (<$iscsi_2>) if tcp_sport == 53292;
           class (<$iscsi_3>) if tcp_sport == 53293;

      htb() {
          class (rate 10 Mbps, ceil 10 Mbps) {
             $iscsi_1 = class (rate 1 Mbps, ceil 10 Mbps) {};
             $iscsi_2 = class (rate 3 Mbps, ceil 10 Mbps) {};
             $iscsi_3 = class (rate 6 Mbps, ceil 10 Mbps) {};
       }
      }
    }
}
```

Listing 3.1: A Simple tcng Script Example.

3.4.2 The Effectiveness of HTB for iSCSI Traffic

Figure 3.6 illustrates the effectiveness of HTB regulating a single iSCSI flow. The bar on the left indicates the iSCSI throughput when no HTB policy is applied. The bar on the right represents the iSCSI throughput when the traffic speed is limited to 40 Mbps. As shown by the data label, the mean error of the actual throughput is less than 2 percent.

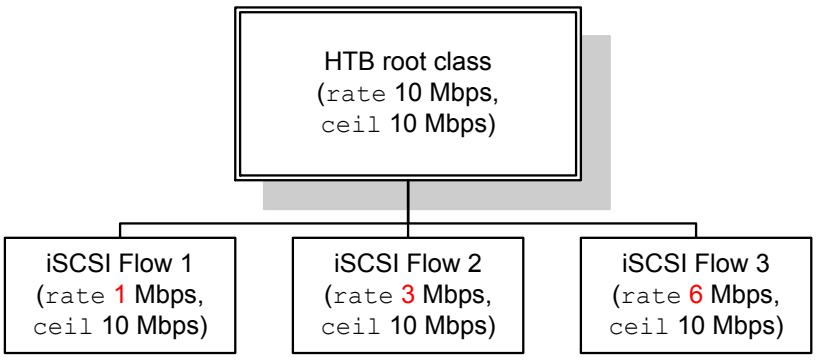


Figure 3.5: The HTB Queuing Discipline Structure Example for the tcng Script.

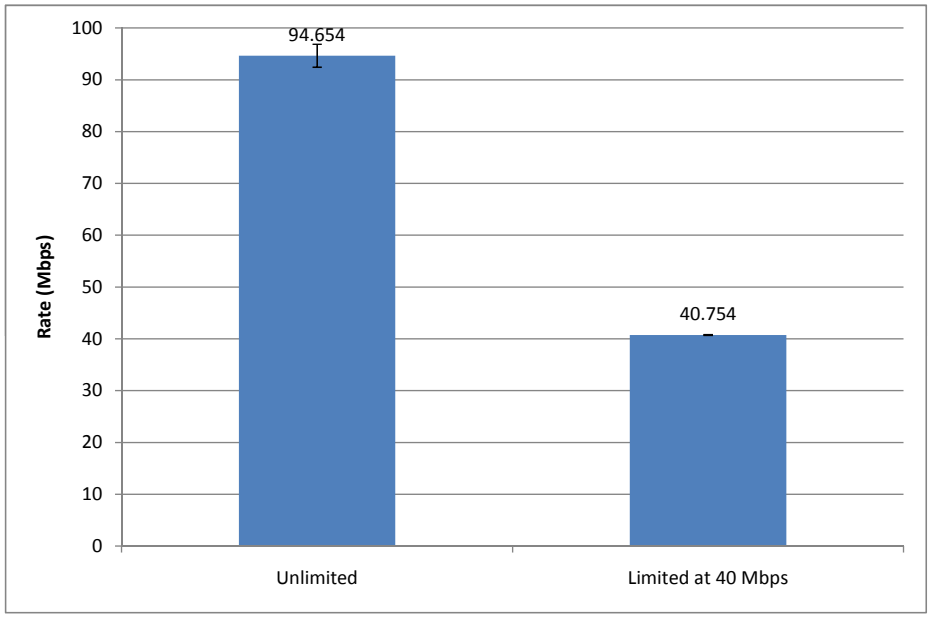


Figure 3.6: The Regulation Effectiveness of HTB on a Single iSCSI Traffic Flow.

Figure 3.7 and Figure 3.8 are the I/O graphs of 3 traffic flows sharing the same physical link. (Please note that the results in these two figures were collected from one single run.) The `rate` and `ceil` values of the root HTB class are set to 10 Mbps, as shown by the global throughput (the blue curve). The `rate` parameters of Flow 1, 2, and 3 are set to 1 Mbps, 3 Mbps, and 6 Mbps respectively. Their `ceil` values are all set to 10 Mbps, which is equal to the `rate` and `ceil` values of the root class.

As shown in Figure 3.7, every flow gets its fair share when they are all saturated with traffic. HTB's `rate` borrowing mechanism starts to kick in when one or more of the traffic flows cannot fully utilize the assigned `rate`, as illustrated in Figure 3.8. The three flows are paused and resumed in turn manually. The unused bandwidth is redistributed instantly according to the ratio of the `rate` values of the flows that require more bandwidth than assigned. For example, at 60 s, Flow 3 is paused Flow 2 is resumed, and Flow 1 stays at the same transmit speed. The inactivity of Flow 3 spares 6 Mbps bandwidth for Flow 1 and Flow 2, which share the 6 Mbps by their `rate` ratio 1:3; therefore, the throughput of Flow 1 should increase to 2.5 Mbps and that of Flow 2 should increase to 7.5 Mbps, which conform to the curves between 60 s and 80 s. At and after 80 s, Flow 3 is resumed and all the 3 flows start to fairly share the bandwidth again.

This bandwidth allocation and redistribution feature of HTB can be utilized to prioritize different flows as well as to provide differentiated services, without sacrificing any available bandwidth resources.

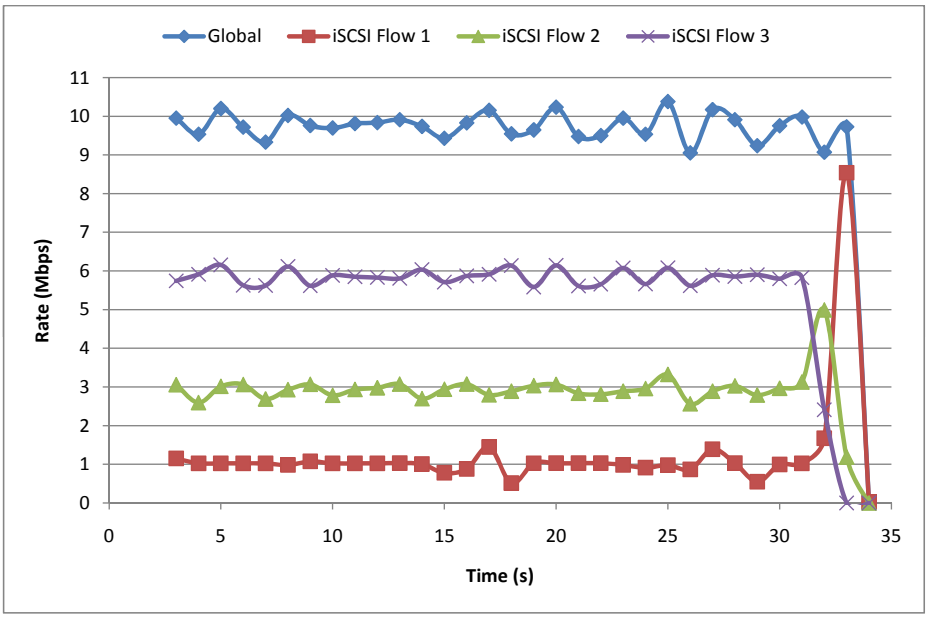


Figure 3.7: The Effectiveness of HTB on Bandwidth Sharing of 3 iSCSI Traffic Flows.

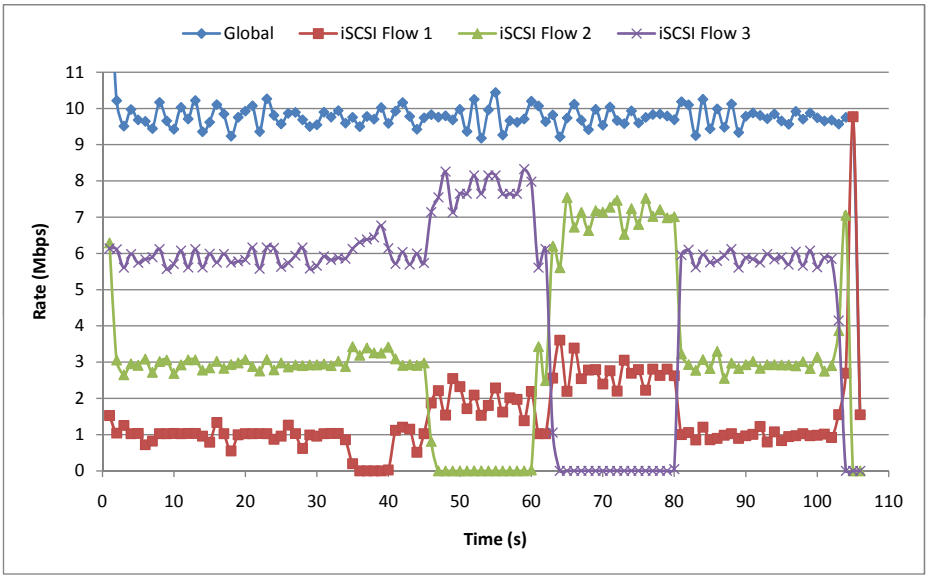


Figure 3.8: The Effectiveness of HTB on Bandwidth Sharing of 3 iSCSI Traffic Flows - with Borrowing.

3.4.3 The Dynamic Features of HTB

Section 3.4.2 has demonstrated the effectiveness of a static HTB policy. In this section we examine the performance of HTB when the policy has to change over time.

We first apply a HTB policy at 20 s to limit the traffic to 20 Mbps, then change the policy to 40 Mbps at 40 s, and eventually remove the HTB policy at 60 s. As shown in Figure 3.9, the traffic speed dips briefly at 20 s. (Please note that the results in this figure were collected from one single run.) Otherwise no visible throughput drop can be observed at the instants when the HTB policy is changed. Between the policy change points, the throughput is as stable as it is in the single flow scenario.

If we zoom in by a factor of 10 on Figure 3.9 and look at a time granularity of 0.1 s, transient traffic pauses do appear on the I/O graph, as shown by the example in Figure 3.10. (Please note that the results in this figure were collected from one single run.) The example illustrates an example at 40 s, when the HTB policy is changed from 20 Mbps to 40 Mbps. Please note that the unit of the vertical axis is Mbits/0.1s instead of Mbps. Because the pause is sufficiently short, it does not have noticeable negative effect on the link throughput, as long as the frequency of the policy change is not excessively high.

This dynamic feature of HTB allows it to fit in the situations where the traffic control policy needs to be altered frequently.

3.4.4 The Impact of Uncontrolled Traffic

Typically, each HTB policy has one root class, which represents the physical link. Although this mapping between the root class of HTB and the physical link is not mandatory, not following this rule, that is, allowing network traffic to go around the control of HTB, may result in unexpected bandwidth sharing: The uncontrolled traffic "bullies" the controlled traffic, as discussed below.

We examined the behavior of two traffic flows when there is another flow that is not regulated by HTB. The test was done on a simulated link of 10 Mbps bandwidth. In the test, the two controlled flows, Flow 1 and Flow 2, were limited to 1 Mbps and 2 Mbps by HTB respectively. The uncontrolled flow, however, was not controlled by HTB and could consume up to all 10 Mbps. As shown in Figure 3.11, the uncontrolled flow steals bandwidth from Flow 1 and Flow 2; neither Flow 1 nor Flow 2 reach their allocated bandwidth. This is an adverse effect caused by the HTB direct queue (Section 3.3.2).

This test demonstrates that HTB is not capable of guaranteeing the bandwidth allocated to the traffic flows it is regulating when other flows from outside HTB attempt to aggressively seize bandwidth.

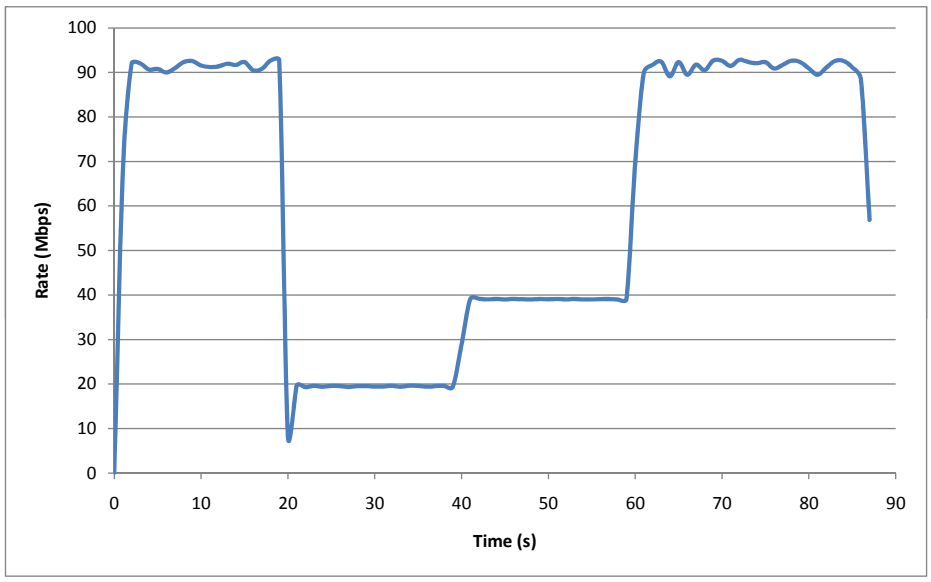


Figure 3.9: The Performance of HTB - Dynamic.

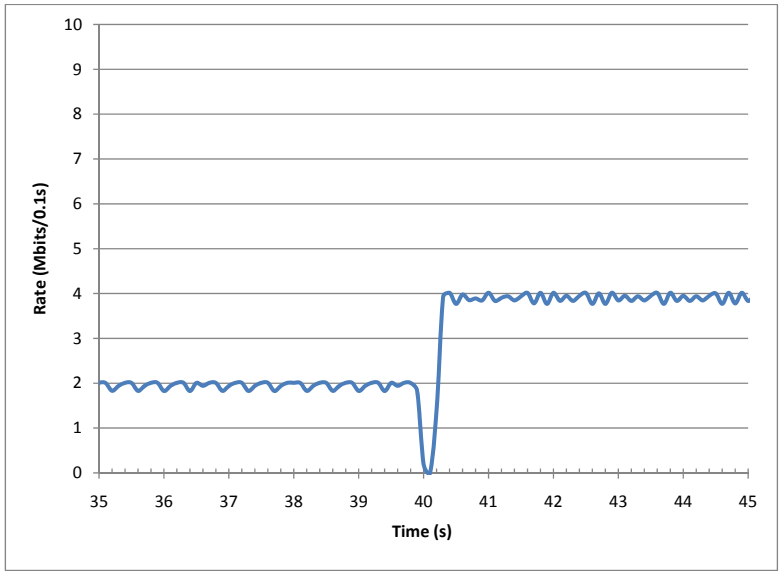


Figure 3.10: The Performance of HTB - Dynamic with the Time Granularity of 0.1 s.

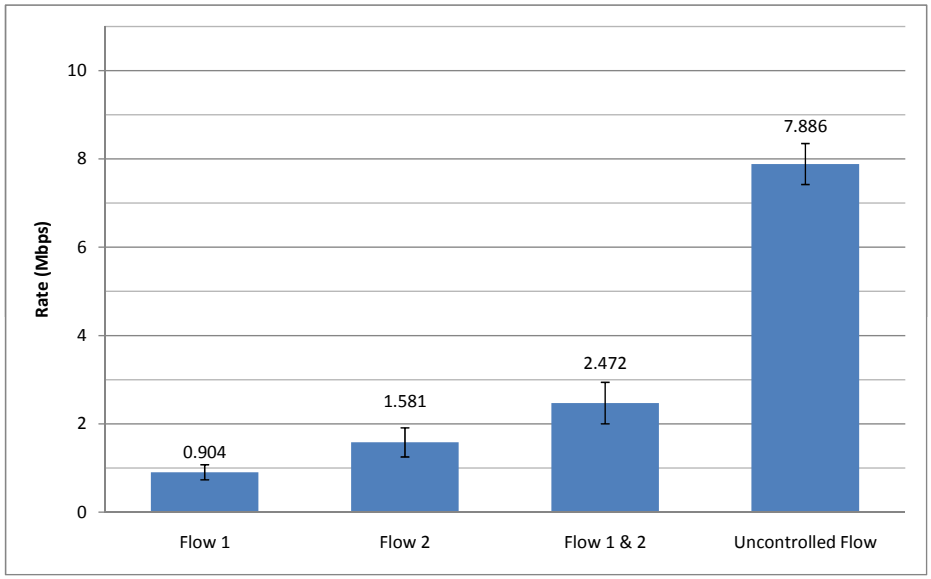


Figure 3.11: The Bully Impact of Uncontrolled Traffic.

Chapter 4

Open-iSCSI Performance Enhancements

The motivation of this chapter is the drastic performance degradation observed in a throughput measurement of Open-iSCSI over a LFN, as shown in Figure 4.1. The blue curve represents the throughput of one iperf session; it indicates the throughput capacity of a general TCP session. Throughput stays at a stable level around 90 Mbps regardless of the link RTT. On the contrary, the achieved bandwidth of Open-iSCSI, represented by the red curve, starts to drop dramatically right after the RTT grows to 12 ms and eventually dives to only around 14 Mbps when the RTT reaches 100 ms. Obviously, the iSCSI initiator traffic suffers severely from long RTTs and it is unable to fully utilize the TCP bandwidth when there is a noticeable delay on the link, which means that, in most WAN settings, users of the Open-iSCSI initiator will suffer from the iSCSI throughput cut-off.

We also ran a similar group of tests of Open-iSCSI over lossy network links. The results are shown in Figure 4.2. As the overlap of the two curves demonstrate, iSCSI does not suffer any more than a general TCP session does over a lossy link. Therefore, we focus our study on the iSCSI performance degradation over long RTT links.

We investigated closely this iSCSI initiator performance degradation issue and present in this thesis a combination of tuning methods, which result in a throughput gain of 70 Mbps on a 100 Mbps link with an RTT of 100 ms.

4.1 Experimental Setup

Experimental environment:

Operating System (iSCSI initiator side): Ubuntu 9.10 Desktop (32 bit) with Linux kernel 2.6.32-22-generic-pae (used out-of-box unless otherwise indicated).

Operating System (iSCSI target side): Ubuntu 9.04 Desktop (32 bit) with Linux kernel 2.6.28-18-generic (TCP receive buffer size tuned for over 100 ms RTT unless otherwise indicated).

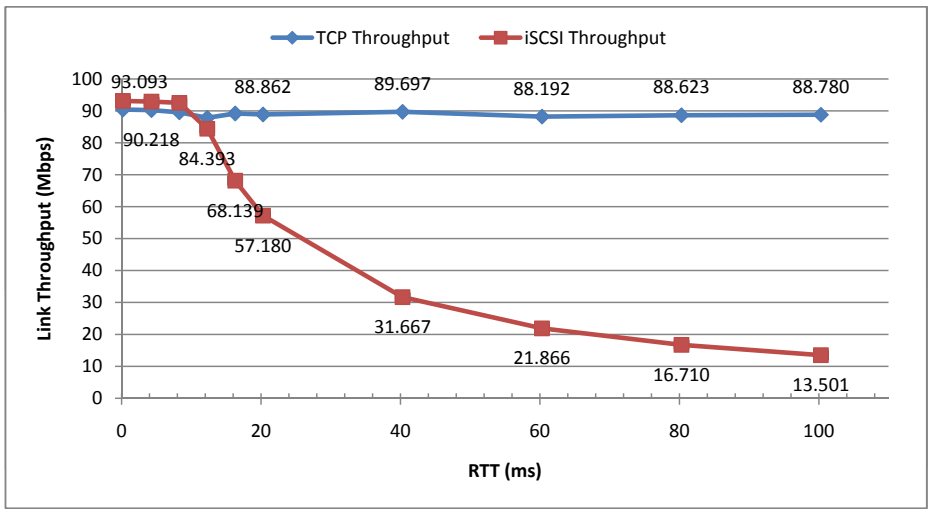


Figure 4.1: Open-iSCSI Performance Degradation over Long RTT Links.

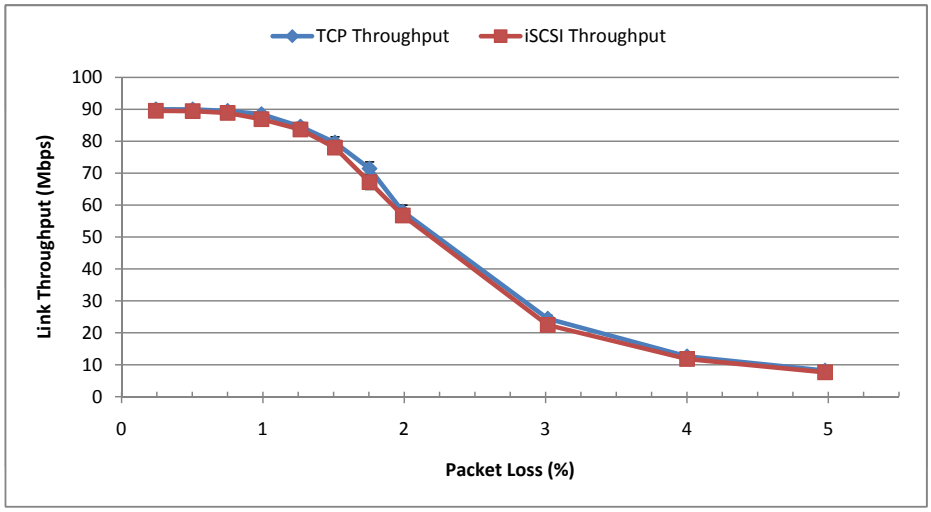


Figure 4.2: Open-iSCSI Performance over Lossy Links.

Network: A direct connection between two PCs using a straight-through cable. This connection works on 100 Mbps with the average RTT of 0.280 ms and the average packet loss of 0%.

iSCSI Implementation: Open-iSCSI Initiator 2.0.871 [4] and iSCSI Enterprise Target 0.4.16 [2].

Traffic Generator: iperf for general TCP traffic and dd for iSCSI traffic

Measurement of Achieved Bandwidth on the Link Layer: Wireshark 1.2.5

We need various link RTTs to investigate the iSCSI throughput over long delay links. On our test network, however, the original link RTT is negligible (0.280 ms) and non-adjustable (the link is a straight-through cable); therefore, we adopted the traffic control command line utility `tc` to emulate delays on the link. `tc` manipulates the output queues of the network interfaces. It attaches a queue discipline (qdisc) to the output interface in order to reschedule, delay, duplicate and/or drop the

Table 4.1: The Accuracy of `netem` in RTT Emulation.

Emulated RTT (ms)	0	4	8	12	16	20	40	60	80	100
Actual RTT (ms)	0.192	4.290	8.293	12.289	16.295	20.292	40.293	60.296	80.293	100.292
Precision	N/A	92.8%	96.3%	97.6%	98.2%	98.5%	99.3%	99.5%	99.6%	99.7%

Table 4.2: The Accuracy of `netem` in Packet Loss Emulation.

Emulated Loss (%)	0.25	0.5	0.75	1	1.25	1.5	1.75	2	3	4	5
Actual Loss (%)	0.243	0.503	0.751	0.990	1.266	1.510	1.754	1.990	3.015	4.001	4.977
Precision	97.0%	99.4%	99.9%	99.0%	98.7%	99.4%	99.8%	99.5%	99.5%	100.0%	99.5%

qualified packets. `tc` belongs to the `iproute2` package, which comes with the Linux kernel 2.4 or later. [13] [8]. The qdisc we used to emulate the link RTT is `netem`. It is capable of emulating Wide Area Network (WAN) properties, such as RTT and packet loss, with a sufficiently high accuracy, as demonstrated by the results of `ping` shown in Table 4.1 and Table 4.2. `netem` is widely used for protocol testing. An example of using `netem` and `tc` to emulate a link with a 4 ms RTT is given in Listing 4.1.

```
1 $ tc qdisc add dev eth0 root netem delay 4ms loss 0%
```

Listing 4.1: Using `tc` and `netem` to Emulate a Link with 4 ms RTT.

4.2 TCP Flow Control and iSCSI Performance

TCP uses a sliding window protocol to control the rates of individual flows, so that the receiver buffer does not get overrun. The size of the sliding window indicates the amount of outstanding data allowed, which is to be sent in one batch unacknowledged. After sending this amount of data, TCP stops and waits for the ACKs of the sent segments.

The minimum amount of time for the ACKs to get back to the transmitter is the link RTT. If the RTT is sufficiently short and the link bandwidth is sufficiently low, the resulting link BDP can be smaller than the sliding window size. In this case, TCP has enough outstanding data to "fill" the network link. As a result, the transmitter keeps sending new segments while waiting for the ACKs of the previously sent segments. Since new data is constantly put on the link, the link is fully utilized during this period of time.

If, however, the link RTT is large and the bandwidth is high, the resulting link BDP can exceed the sliding window size. In this case, after TCP sends out a batch of segments of the sliding window size, the ACK for the first segment still has not reached the transmitter yet, if indeed it has already left the receiver at all. As a result, TCP has to idle to wait for the ACKs of the sent segment before sending any more segments. This idling is caused by TCP not having enough outstanding data to

"fill" the link. During the idling period, the link is not used because no new data is sent to the link; therefore, the link is not fully utilized in general.

Let t_{send} and t_{idle} denote the time TCP spends in sending and idling respectively. Let $B_{available}$ denote the capacity of the link. The actually achieved bandwidth is

$$B_{achieved} = \frac{t_{send}}{t_{send} + t_{idle}} \times B_{available}. \tag{4.1}$$

As the sending time and the idling time depend on the BDP of the link and the send window size, i.e. the sliding window size, (denoted as S_{send_window}), Eq. 4.1 can be rewritten as

$$\begin{aligned} B_{achieved} &= \frac{S_{send\_window}}{BDP} \times B_{available} \\ &= \frac{S_{send\_window}}{B_{available} * RTT} \times B_{available} \\ &= \frac{S_{send\_window}}{RTT}. \end{aligned} \tag{4.2}$$

Eq. 4.2 shows clearly how the send window size and RTT affect the actually achieved bandwidth. For example, a link between Beijing in China and Edmonton in Canada can have an RTT of 200 ms [27]. Suppose the bandwidth is 50 Mbps. Hence, the link BDP is 1280 KB. If the send window size is 640 KB, it takes TCP only 100 ms to put the data within the window on the wire. The ACKs of this batch of data, however, do not come back until after another 100 ms, as the link RTT is 200 ms. During the second half of the RTT, TCP can do nothing but idle. Therefore, the 50 Mbps bandwidth is wasted during this 100 ms period. The overall achieved bandwidth is 640 KB $\div$ 200 ms, which is 25 Mbps. This number is only half of the link capacity but it is not a surprise. After all, we are only sending data for half of the time. Also, not surprisingly, the bandwidth waste is expected to be even worse if the link capacity is larger - the faster the link is, the more bandwidth is wasted.

Such a bandwidth waste is unacceptable, especially for today's long-distance / high-bandwidth links. Recent Layer 1 technology has boosted the link capacity on a large scale. Gigabit Ethernet NICs are standard configuration on modern PCs and 2.488 Gbps OC48 fiber optic cables are typical on Internet backbones. Verizon, one of the major network providers in the U.S, just deployed a commercial ultra-long-haul optical system that which enables 100 Gbps link capacity[1]. On such a link, idling of 1 ms means the loss of more than 100 MB throughput.

Therefore, keeping TCP running without idling is crucial for achieving the highest possible throughput. In order to prevent TCP from idling, the send window size needs to be properly adjusted.

4.2.1 TCP Send Buffer Size and Open-iSCSI Performance

In Section 4.2, we discussed the effect of the TCP send window size on the TCP throughput. In implementation, the TCP send window size is realized as the send buffer size of the TCP socket. We

[1] `http://www.dailytech.com/Verizon+Deploys+First+100+Gbps+Backbone+in+Europe/article17125.htm`

investigated the effect of varying send buffer sizes have over Open-iSCSI throughput on links with RTTs of 4 ms, 8 ms, 12 ms, 16 ms, 20 ms, 40 ms, 60 ms, 80 ms, and 100 ms. The initial 4 ms step between 4 ms and 20 ms was chosen because of the kernel timer resolution, which is configured to be 250 HZ at compile time. As shown by the results of `ping` in Table 4.1, the average accuracy of the actual RTTs is ~98%, which is sufficiently accurate for our experiments.

There are generally two ways of adjusting the TCP send buffer size: statically or dynamically. In the static method, the buffer size is set by calling the function `setsockopt()` (in C on Linux). Once the buffer size is set, the socket works with this size unless explicitly told otherwise (usually through another call to `setsockopt()`. The results of iSCSI throughput with statically set buffer size are shown and discussed in Section 4.2.1.1. On the contrary, in the dynamic method, one does not set the send buffer size of the TCP socket. Instead, the send buffer size is determined by TCP autotuning and it changes in accordance with the change of network link parameters. The results of iSCSI throughput with dynamically set buffer size are shown and discussed in Section 4.2.1.2. The details of how to set the send buffer size in both ways are discussed in Section 4.2.2.

4.2.1.1 Statically Set TCP Send Buffer Size and iSCSI Performance

The first batch of iSCSI throughput results gathered by setting the TCP send buffer size statically are plotted in Figure 4.3, Figure 4.4, and Figure 4.5. Please note that the vertical axis in Figure 4.5 starts from 50 Mbps instead of 0 Mbps. In each of these figures, the iSCSI throughput is compared to the throughput of a baseline TCP application, namely iperf, whose send buffer size is autotuned by the kernel. An obvious trend in all of the figures is that the iSCSI throughput is significantly improved as the send buffer size increases from 128 KB to 1280 KB[2].

The throughput values, however, are still lower than expected. For example, according to Eq. 4.2, at 100 ms RTT, to achieve the maximum TCP throughput, we only need the send buffer size to be 1280 KB, which means that in Figure 4.5d the iSCSI throughput should have at least matched the TCP baseline application output. On the contrary, we observe a disparity between the TCP baseline throughput and iSCSI throughput. To investigate if this disparity is caused by insufficient send buffer size, we further increased the send buffer size up to 2560 KB and tested the iSCSI performance. The results show that no significant throughput benefit is achieved with further increases in the send buffer size above 1280 KB, as shown in Figure 4.6.

To have a closer look of the iSCSI throughput change versus various send buffer sizes, we plotted the throughput against the send buffer sizes at 40 ms and 100ms, as shown in Figure 4.7 and Figure 4.8 respectively. The curves in both figures show that the iSCSI throughput continues to benefit from the increase of the send buffer size until the send buffer size hits a certain value. The points where iSCSI last benefits from an increase in send buffer size are 512 KB at 40 ms RTT and

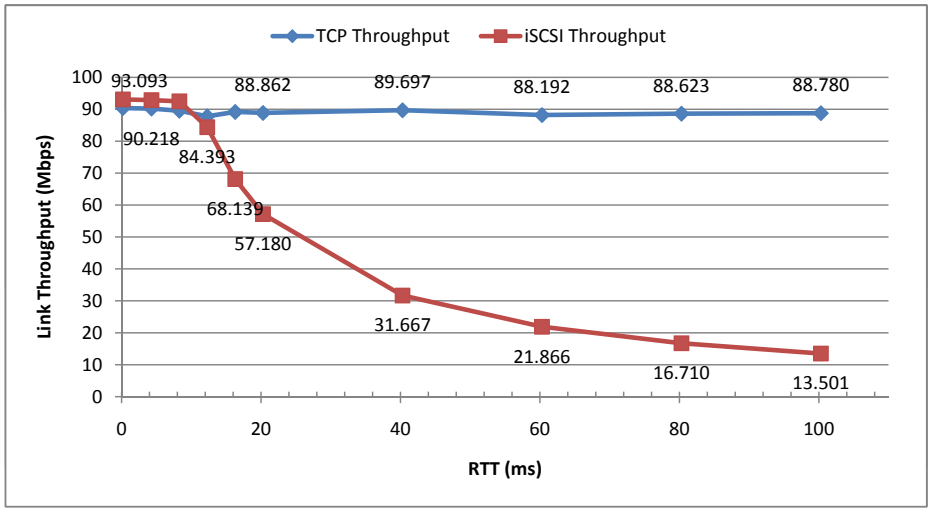
[2]From send buffer size of 1152 KB to 1280 KB, the iSCSI throughput at 100 ms seemingly decreased. Considering the standard deviation of the results (as shown by the error bar in Figure 4.5d), however, this value can be considered as without significant change.

1152 KB at 100 ms RTT. 512 KB is the theoretical send buffer size to achieve 100 Mbps bandwidth at 40 ms RTT and 1152 KB is close to the theoretical send buffer size to achieve 100 Mbps at 100 ms RTT. Similar results are also observed at RTTs of 20 ms, 60 ms and 80 ms: The iSCSI throughput continually benefits significantly from the increase of the send buffer size up to the point where the send buffer size is the theoretical value to achieve the maximum bandwidth. We call this buffer size *iSCSI Maximum Benefit Buffer Size*, which is denoted as $S_{max\_benefit}$.

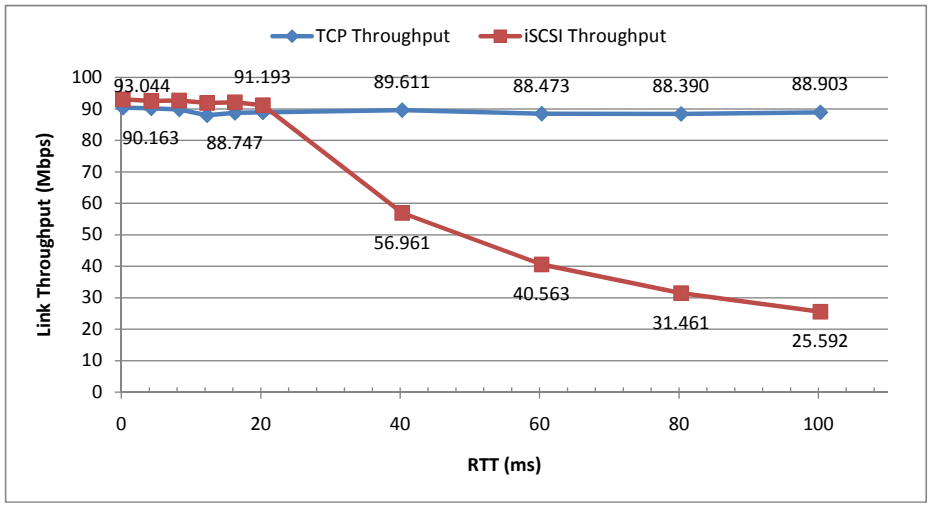
As soon as the send buffer size exceeds $S_{max\_benefit}$, the iSCSI throughput is limited by the capacity of the network link and Network Interface Controller (NIC) as well as other implementation factors instead of the TCP send buffer size.

Moreover, exceedingly large send buffer size can have adverse effects on the iSCSI throughput, as demonstrated by the gap between 90 Mbps and the iSCSI throughput starting from send buffer size 1024 KB in Figure 4.7. Excessively large send buffer size has additional devastating effects on lossy links as well, but the discussion of this matter is out of the scope of this thesis.

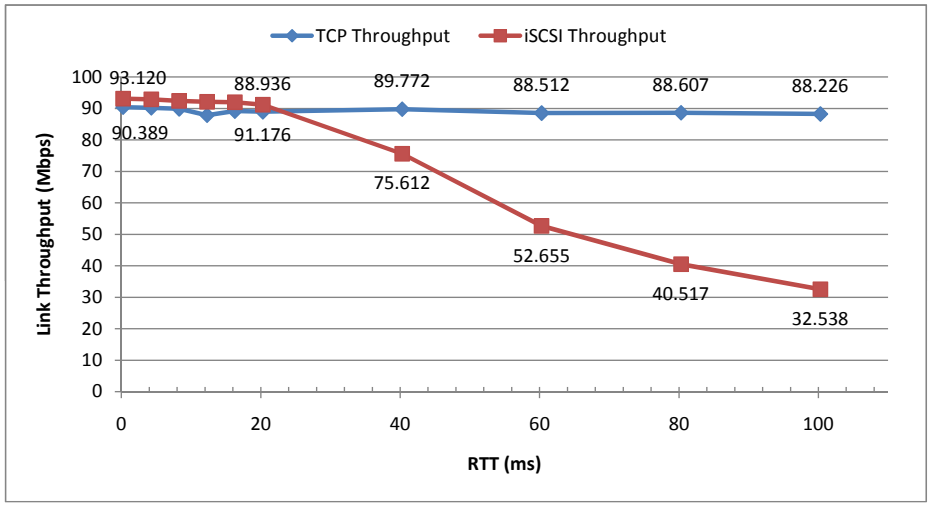
Therefore, with the socket buffer size statically set to a proper value, Open-iSCSI can achieve the desired performance. Section 4.2.2.1 explains how to statically set the TCP send buffer size.



(a) Default Send Buffer Size (128 KB)

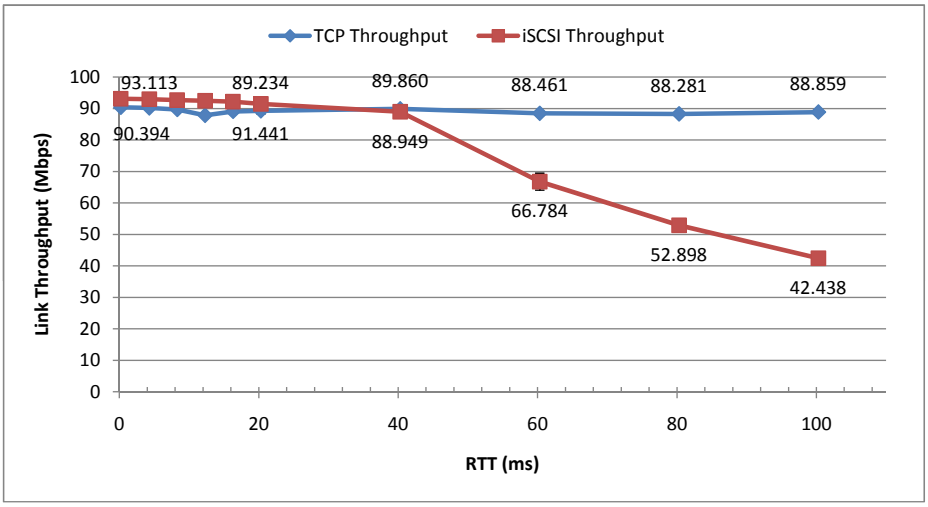


(b) Send Buffer Size 256 KB

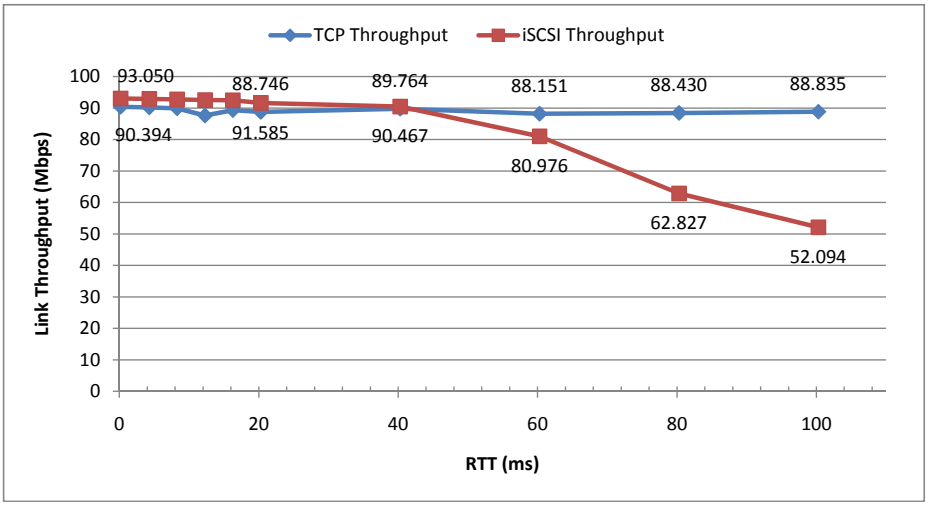


(c) Send Buffer Size 384 KB

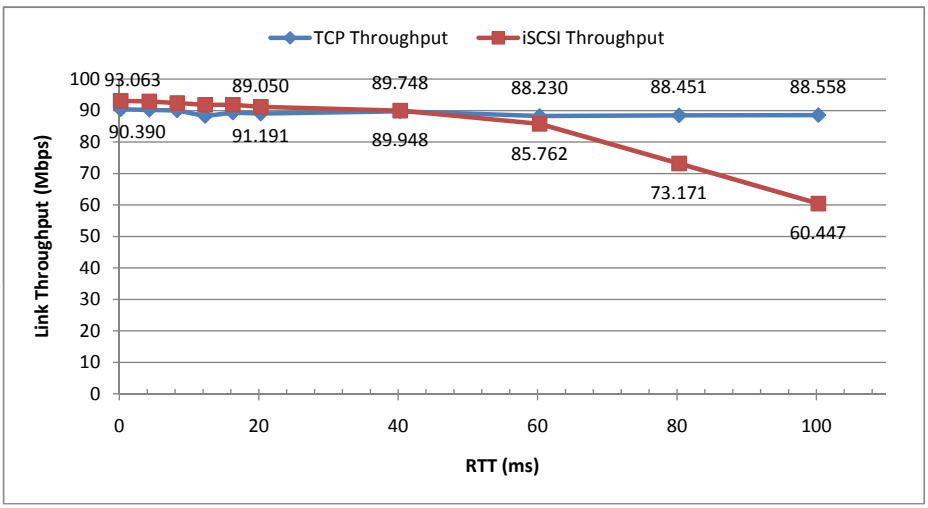
Figure 4.3: Open-iSCSI Performance with Various Send Buffer Sizes and RTTs - 1.



(a) Send Buffer Size 512 KB

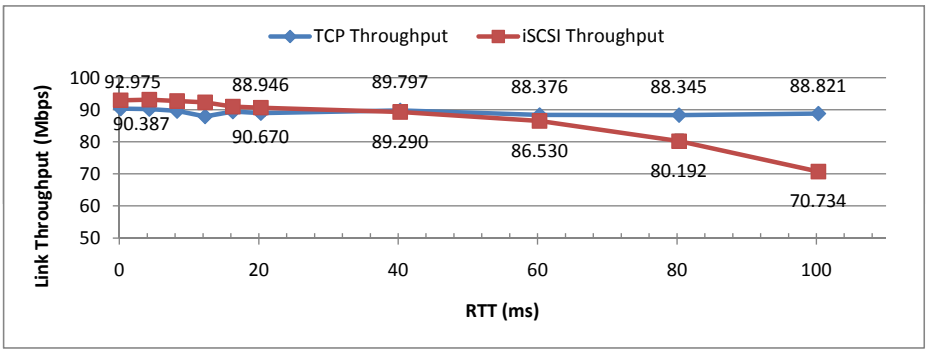


(b) Send Buffer Size 640 KB

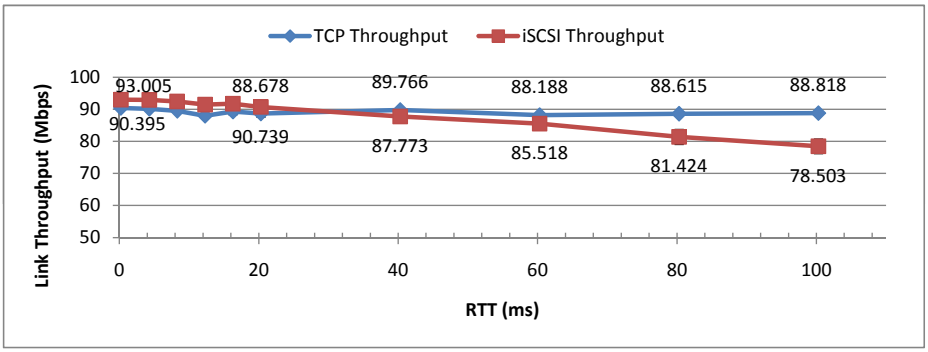


(c) Send Buffer Size 768 KB

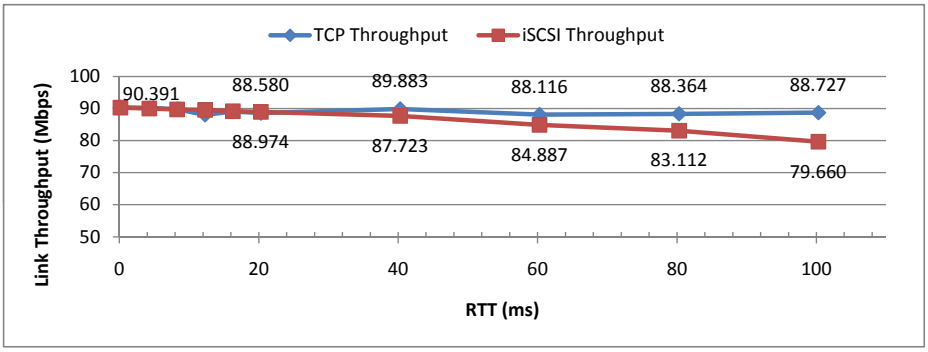
Figure 4.4: Open-iSCSI Performance with Various Send Buffer Sizes and RTTs - 2.



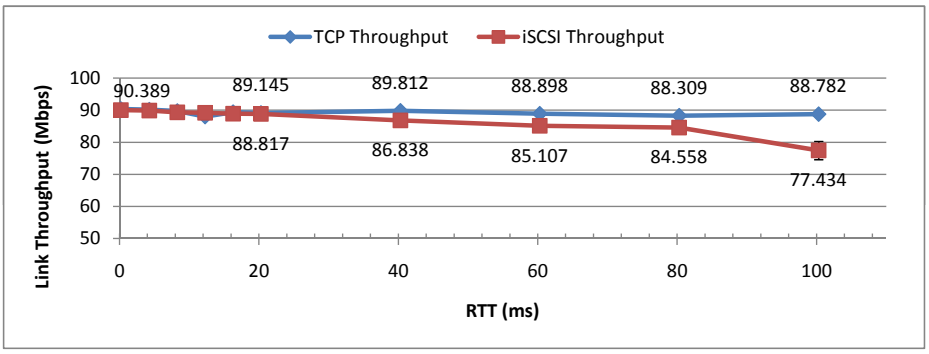
(a) Send Buffer Size 896 KB



(b) Send Buffer Size 1024 KB

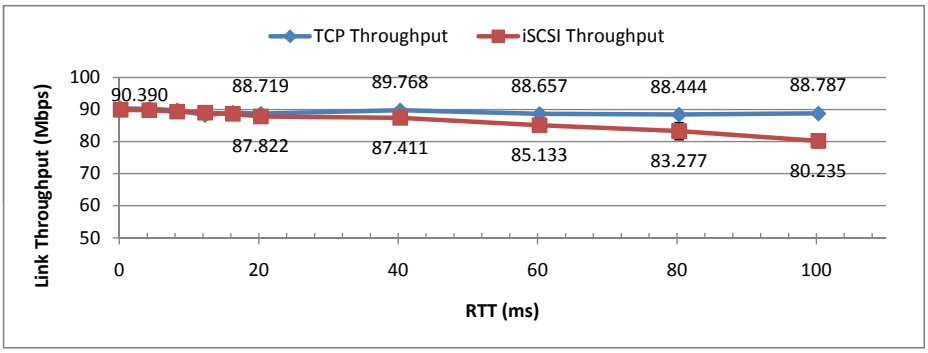


(c) Send Buffer Size 1152 KB

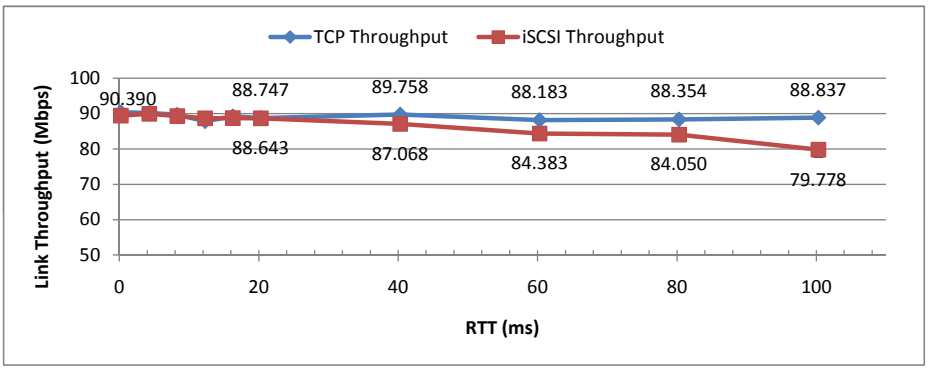


(d) Send Buffer Size 1280 KB

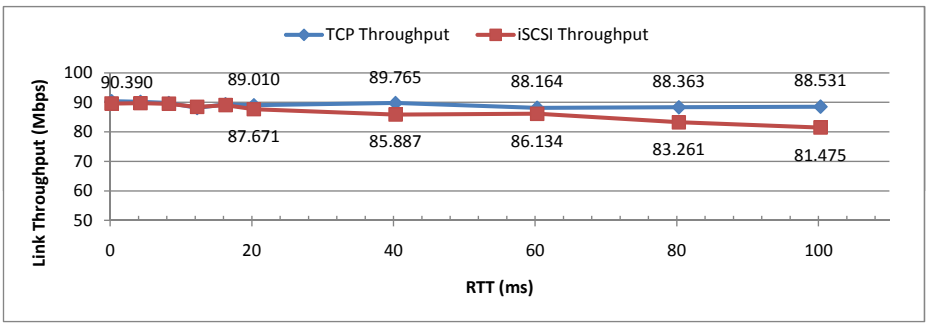
Figure 4.5: Open-iSCSI Performance with Various Send Buffer Sizes and RTTs - 3.



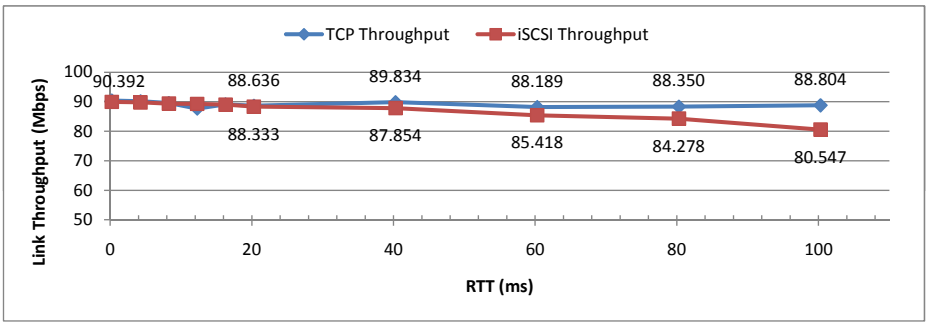
(a) Send Buffer Size 1408 KB



(b) Send Buffer Size 1536 KB



(c) Send Buffer Size 2048 KB



(d) Send Buffer Size 2560 KB

Figure 4.6: Open-iSCSI Performance with Various Send Buffer Sizes and RTTs - 4.

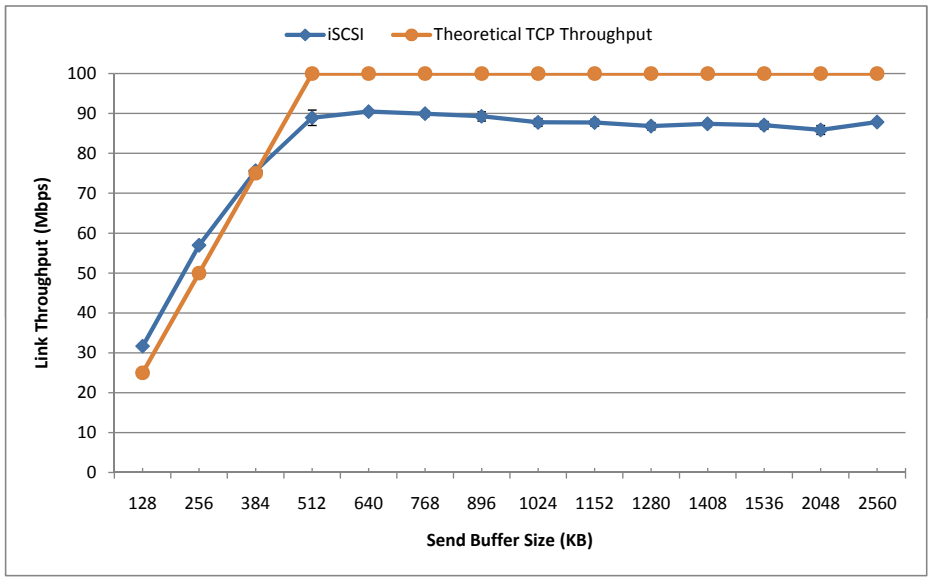


Figure 4.7: Open-iSCSI Performance with Various Send Buffer Sizes @ 40 ms RTT

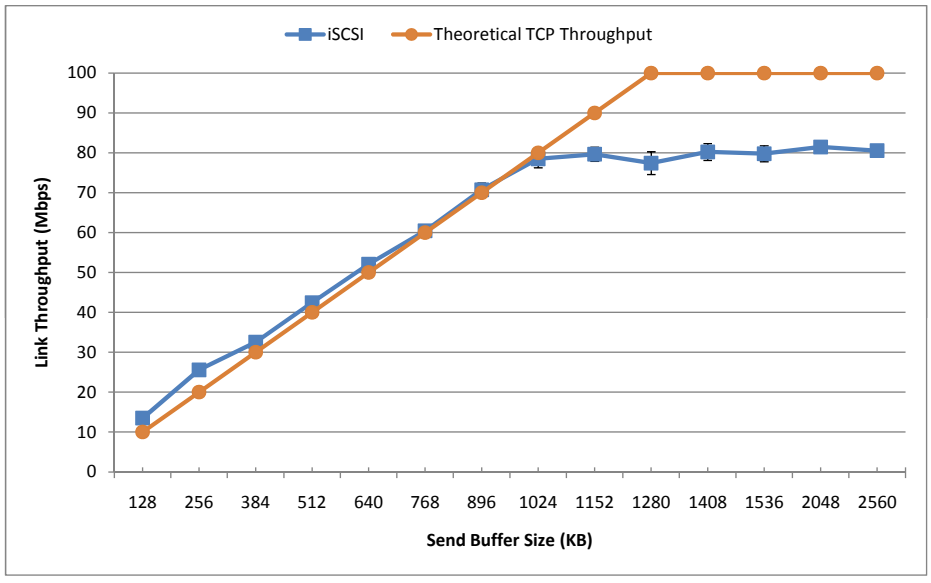


Figure 4.8: Open-iSCSI Performance with Various Send Buffer Sizes @ 100 ms RTT

4.2.1.2 Dynamically Set TCP Send Buffer Size and iSCSI Performance

The TCP send buffer size can be set dynamically through TCP autotuning. The iSCSI throughput results with dynamically set TCP send buffer size were collected using the default TCP autotuning settings, which are explained in detail in Section 4.2.2.2. Figure 4.9 illustrates the results, from which we can see that TCP autotuning does a generally good job in buffer size adjustment to adapt to different network link RTTs.

At this point, one may be wondering whether to use static or dynamic tuning of TCP send buffer size. Figure 4.10 can provide some insight on this matter. The blue curve represents the iSCSI throughput when the TCP send buffer size is set to $S_{max\_benefit}$ (iSCSI Maximum Benefit Buffer Size, please refer to Section 4.2.1.1.). If the $S_{max\_benefit}$ for a certain RTT is not included in our experimental buffer sizes, we use the least possible buffer size that is larger than the $S_{max\_benefit}$ instead. For example, the $S_{max\_benefit}$ is 153.6 KB for 12 ms RTT on a 100 Mbps link, but 153.6 KB is not included in the buffer size list; therefore, it will be replaced by 256 KB, as 256 KB is the least possible buffer size in the list that is larger than 153.6 KB.

As shown in Figure 4.10, when the RTT is less than or equal to 40 ms, the iSCSI session with the buffer size set statically outperforms the iSCSI session with TCP autotuning. As the RTT increases above 40 ms, however, TCP autotuning starts to show its advantage.

4.2.2 The TCP Send Buffer Size in Implementation

In implementation, the TCP send window size is determined by three factors: the send buffer size of the transmitter, the receive buffer size of the receiver, and the transmitter's congestion window size. In this thesis, however, we work with an error-free link; thus, no packet loss would occur and the congestion window size does not constrain the TCP send window size. Also, the receive end is always properly tuned in our experiments for RTTs of ~100 ms; hence, the only factor determining the TCP send window size is the send buffer size of the socket. We consider the send buffer size and the send window size interchangeable in this section.

The send buffer size can be set in two ways - statically and dynamically, which are addressed in

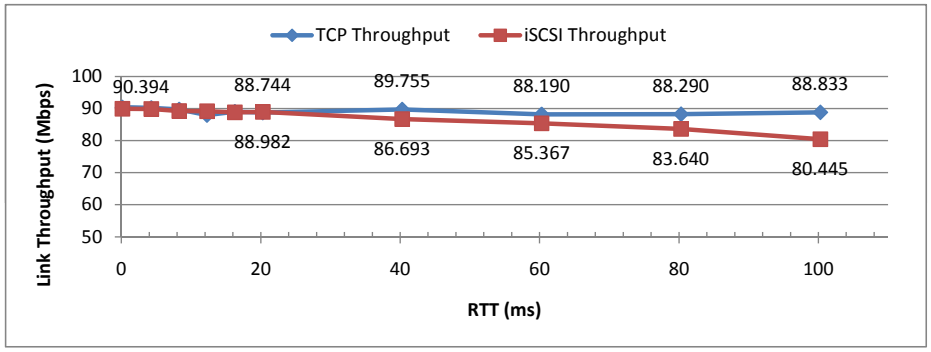


Figure 4.9: Open-iSCSI Performance with Various RTTs with Dynamically Set Send Buffer Size.

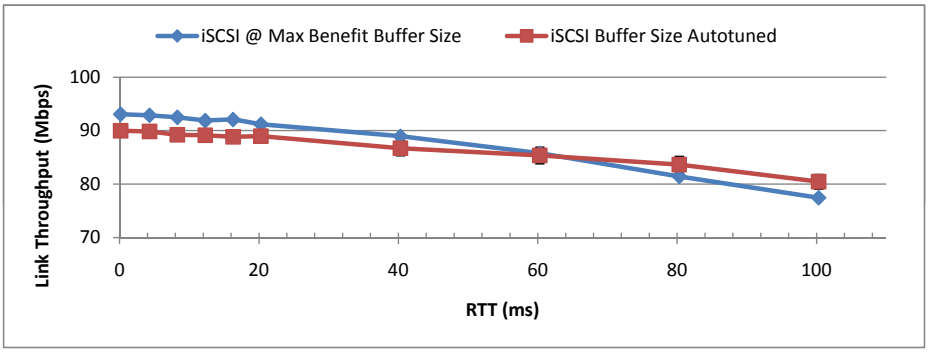


Figure 4.10: Open-iSCSI Performance - Send Buffer Size Adjustment Dynamic vs. Static.

the following two sections.

4.2.2.1 `setsockopt()` - A Static Approach

To set the TCP send buffer statically, we call function `setsockopt()`. Listing 4.2 shows how to statically set the send buffer size to 1 MB.

```
1 #include <sys/socket.h>
2
3 // suppose we have a socket with the file descriptor soc_fd
4 // set the send buffer size to 1 MB (1048576 B)
5 // no error checking is included
6
7 setsockopt(soc_fd, SOL_SOCKET, SO_SNDBUF, &1048576, sizeof(1048576))
```

Listing 4.2: Set the Send Buffer Size to 1 MB.

According to Dunigan [11], Linux 2.4 kernel *doubles* the requested buffer size in `setsockopt()` and so do we observe with Linux 2.6 kernel. `setsockopt()`, however, does not always get what is required: The actual buffer size returned by the OS is subject to the maximum value specified by the system environment variable `net.core.wmem_max`. Interestingly, the kernel also doubles this maximum value as the upper limit of send buffer size [20]. For example, if `net.core.wmem_max` is set to 131071 (128 K - 1) B and we use the code in Listing 4.2 to set the send buffer size, the kernel will try to acquire a buffer size of 2097152 (1048576×2) B, but the actual buffer size will be only 262142 (256 K, 131071×2) B, due to the limit of the doubled `net.core.wmem_max`. Another interesting fact about the buffer size setting is that the returned buffer size (the 262142 in our example) is not completely available for the network application: The kernel reserves a certain portion[3] of it for metadata storage, i.e. the housekeeping data for the actual payloads.

We examined the Open-iSCSI initiator source code and noticed that Open-iSCSI sets the buffer size to a fixed value of 512 KB by default. Supposedly, this default setting should give us a 1 MB send buffer. The actual buffer size that iSCSI gets by default, however, as revealed by the Open-iSCSI log file, is merely 256 KB. The cause of this “shrunken” buffer size is the small out-of-box

[3] According to the man page of socket(7), the Linux kernel reserves 50 percent of the returned buffer size. Other sources on the Internet, however, claim that it is not 50 percent but 25 percent.

value of `net.core.wmem_max`, which is only 131071 on our test system and for most Linux distributions as well. According to Eq. 4.2, the theoretical maximum throughput achieved with this buffer size over a link with 100 ms RTT is 40 Mbps[4], which is less than half of the link bandwidth 100 Mbps.

To change this fixed buffer size, one may be tempted to change the 512 KB constant in the Open-iSCSI code, as shown in the following code:

```
#define TCP_WINDOW_SIZE 512 * 1024
```

Unfortunately, however, this method does *not* work. Open-iSCSI maintains a database for each of the targets it has already discovered. The records of those targets are created the first time the target discovery occurs[5] and these records stay in the database unless deleted explicitly (recompiling the Open-iSCSI source code does not delete previously saved records). The value "512 * 1024" in the code above is used as the *default* value for the send buffer size in the record *creation*; therefore, even if the Open-iSCSI source code is recompiled with an updated value for the macro `TCP_WINDOW_SIZE`, the change to this value is only going to affect the connections to the targets discovered after this change; the records of previously discovered targets do not get rebuilt when they are re-connected, so the change in this macro `TCP_WINDOW_SIZE` does not update the send buffer size of the sockets related to these targets. To actually change the buffer size, we need to change the `tcp_window_size` (not the macro in the source code) value saved in the target record entry using the Open-iSCSI `iscsiadm` command line utility as shown in Listing 4.3.

```
1 $ iscsiadm -m node -T <iSCSI target name> -p <iSCSI target portal> --op -n node.
    conn[<connection number>].tcp_window_size -v <the new send buffer size in
    bytes>
```

Listing 4.3: Set the Send Buffer Size in Open-iSCSI.

Theoretically, the most desirable actual send buffer size is the BDP of the link.

4.2.2.2 TCP Autotuning - A Dynamic Approach

Section 4.2.2.1 explained how we can manually adjust the send buffer size to achieve the highest possible throughput. The problem with this static approach is, however, that if either or both of the link bandwidth or the RTT changes dynamically, the old BDP may no longer be the optimal buffer size after each change. Thus, we have to constantly monitor the link parameters to keep up with the change, otherwise we may only achieve a suboptimal throughput.

Fortunately, however, Linux has officially introduced a TCP autotuning mechanism since kernel 2.6.7 (and back-ported to 2.4.27) [17]. With this autotuning mechanism, on the transmitter side,

[4]This value is larger than the actually achieved bandwidth as shown in Figure 4.1. We believe this gap is caused by certain implementation issue of Open-iSCSI.

[5]Please refer to `http://www.open-iscsi.org/docs/README` for more details about the iSCSI target discovery process.

TCP automatically adjusts the send buffer size to maximize throughput. The send buffer size can be up to the limit set by the system variable `net.ipv4.tcp_wmem = "min default max"`. The meanings of the values are explained below [1]:

`min`: The minimum amount of memory every TCP socket allowed to use for the send buffer. The default value of `min` is 4 KB.

`default`: The initial size of the send buffer used by a TCP socket. The default value of `default` is 16 KB.

`max`: The maximum amount of memory allowed for automatically tuned TCP send buffers. The default value of `max` is between 64 KB and 4 MB, depending on the system RAM size.

On our test system, this variable is set to `net.ipv4.tcp_wmem = "4096 16384 2674688"` out-of-box.

Please note that the Linux kernel stops autotuning the TCP send buffer size if the application explicitly sets the socket buffer size using `SO_SNDBUF` through `setsockopt()`. To use TCP autotuning in Open-iSCSI initiator, the send buffer size should be set to zero using the command shown in Listing 4.3.

4.3 Nagle's Algorithm and iSCSI Performance

To attempt to further increase the iSCSI throughput, we reviewed the Open-iSCSI implementation. In the source code, we noticed that the socket option `TCP_NODELAY` is set, as shown in Listing 4.4. This `TCP_NODELAY` option is the switch for a TCP transmission policy: Nagle's algorithm. When `TCP_NODELAY` is set, Nagle's algorithm is *disabled*.

```
1 int rc, onearg;
2 ...
3 onearg = 1;
4 rc = setsockopt(conn->socket_fd, IPPROTO_TCP, TCP_NODELAY, &onearg,
5     sizeof (onearg));
```

Listing 4.4: Set the Socket Option `TCP_NODELAY`.

Proposed in RFC 896 [21], Nagle's algorithm aims to reduce the TCP/IP header overhead when the data comes to TCP in small chunks. When there is unacknowledged data on the link, Nagle's algorithm puts data that arrives in small chunks on hold and does not transmit the data until all the previous sent data is acknowledged or the accumulated data exceeds the maximum segment size[6] of TCP.

In our tests, iSCSI is sending big chunks of data; therefore, we do not expect significant change in the iSCSI throughput if we enable Nagle's algorithm by *disabling* `TCP_NODELAY`.

[6] Please refer to `http://en.wikipedia.org/wiki/Maximum_segment_size` for more details on TCP maximum segment size.

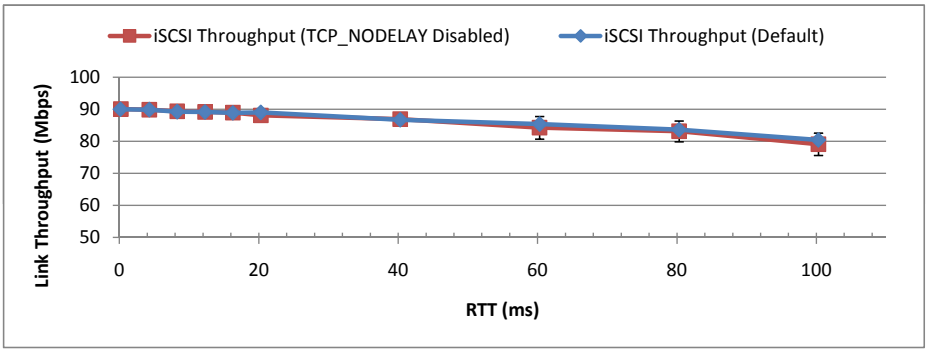
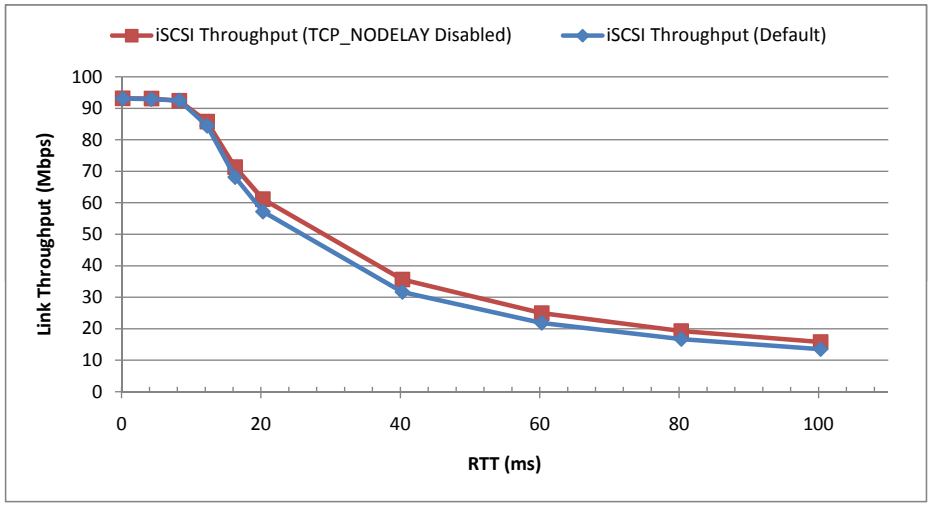


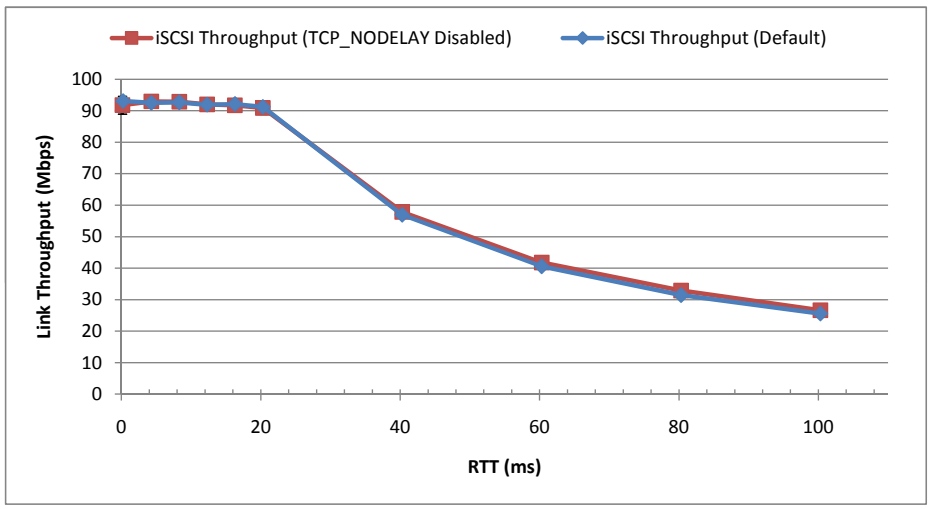
Figure 4.11: TCP_NODELAY and Open-iSCSI Performance (Dynamically Set Send Buffer Size).

Figure 4.11 shows the iSCSI throughput results, with and without TCP_NODELAY enabled, with dynamically adjusted send buffer size. No significant difference is observed between the two curves. Figure 4.12 to Figure 4.15 present the throughput result with statically set send buffer sizes, with and without TCP_NODELAY enabled. Again, no significant difference can be observed between the two implementations, except for the small performance gain through disabling TCP_NODELAY at 384 KB send buffer size and the slight performance degradation due to disabling TCP_NODELAY at 100 ms RTT with 1536 KB send buffer size.

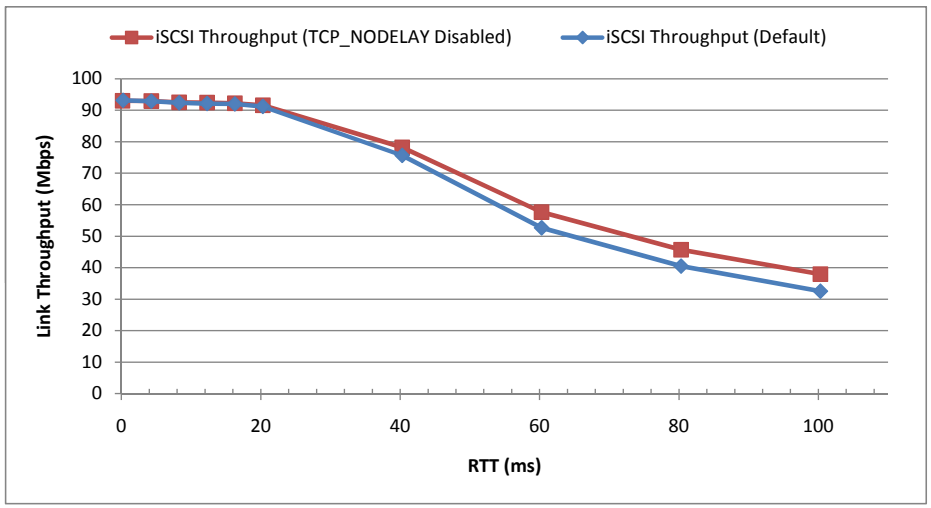
Therefore, Nagle's algorithm, or the TCP_NODELAY option, alone does not have any noticeable effect on the iSCSI performance. It is therefore unclear why the default implementation touches this option at all from the perspective of the iSCSI throughput.



(a) Default Send Buffer Size (128 KB)

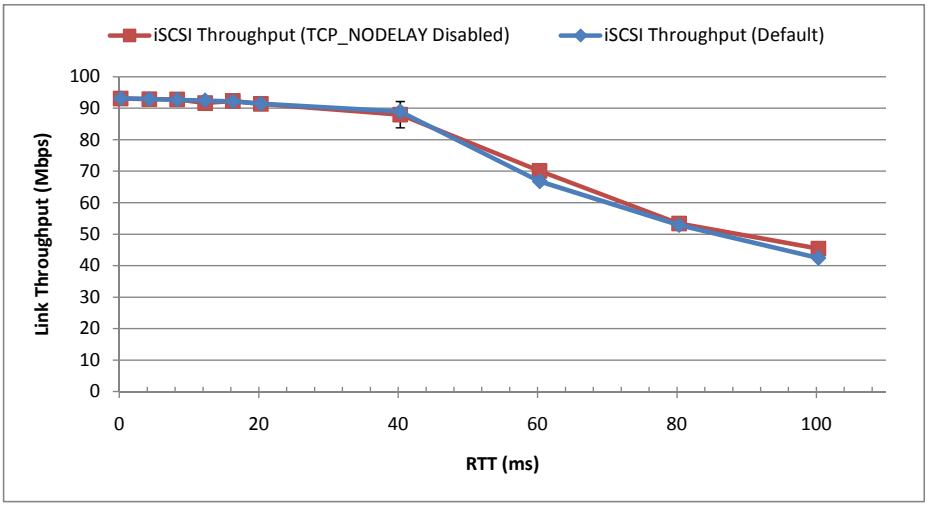


(b) Send Buffer Size 256 KB

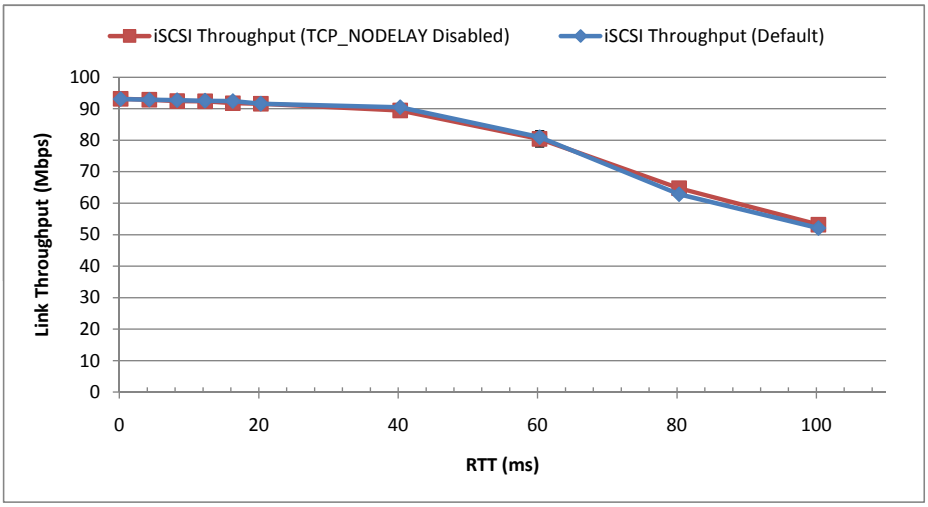


(c) Send Buffer Size 384 KB

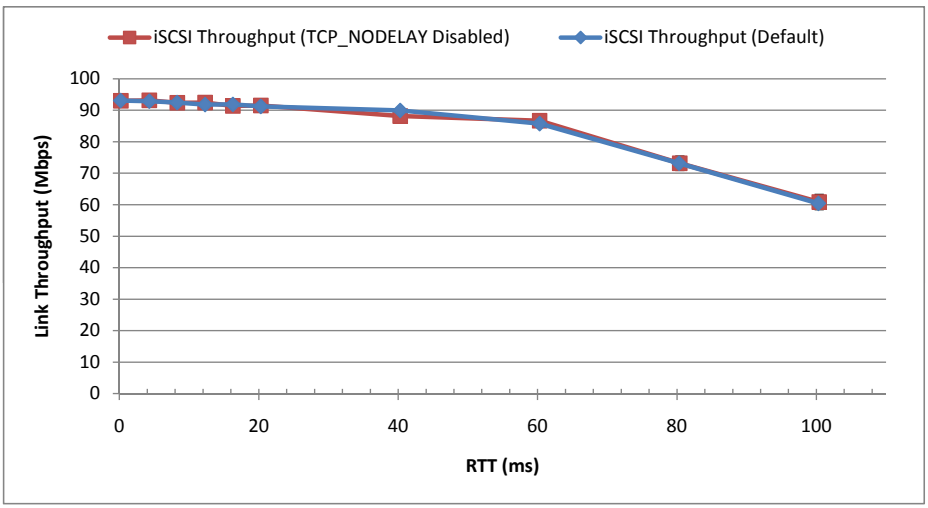
Figure 4.12: TCP_NODELAY and Open-iSCSI Performance (Statically Set Send Buffer Size) - 1.



(a) Send Buffer Size 512 KB

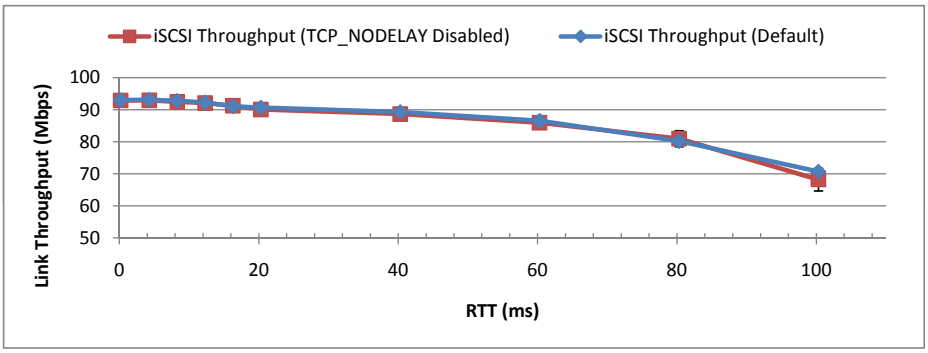


(b) Send Buffer Size 640 KB

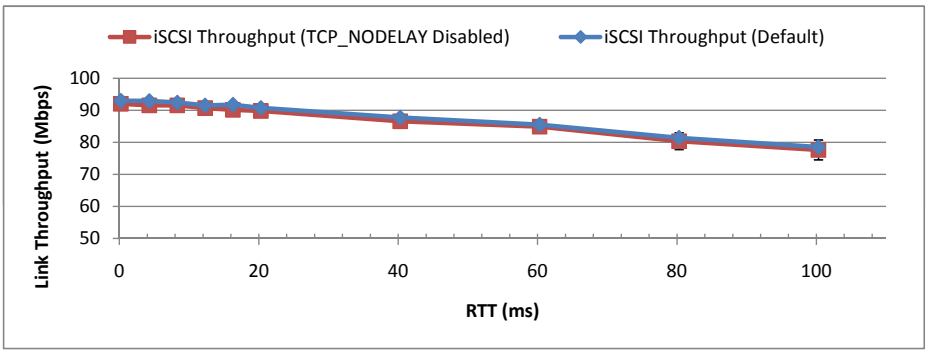


(c) Send Buffer Size 768 KB

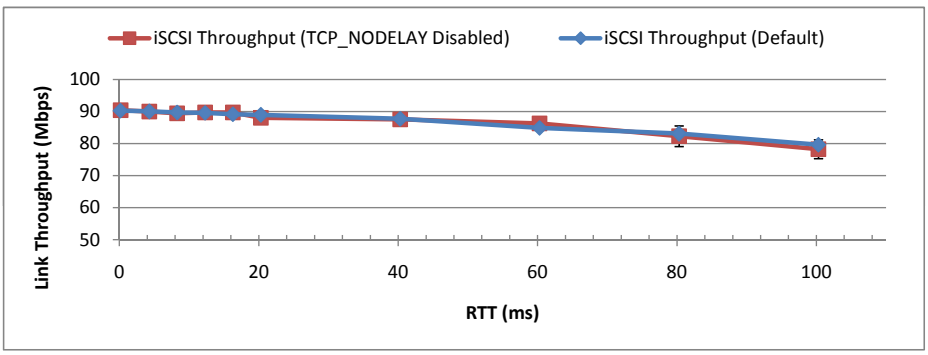
Figure 4.13: TCP_NODELAY and Open-iSCSI Performance (Statically Set Send Buffer Size) - 2.



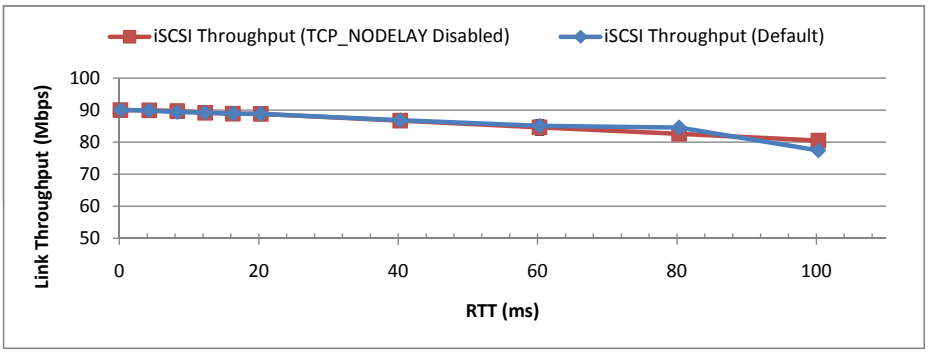
(a) Send Buffer Size 896 KB



(b) Send Buffer Size 1024 KB

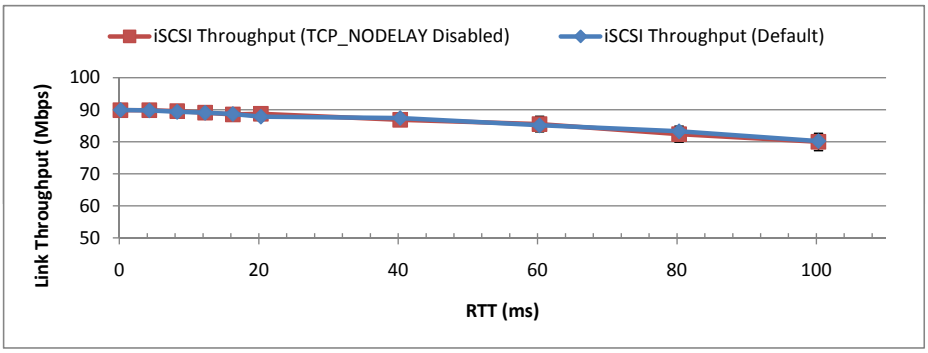


(c) Send Buffer Size 1152 KB

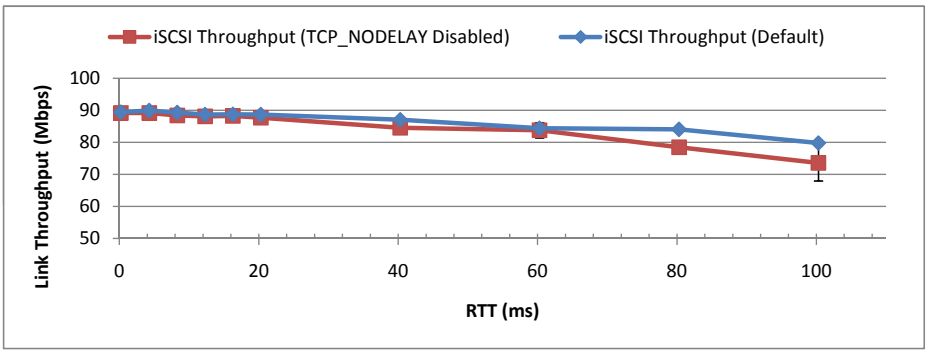


(d) Send Buffer Size 1280 KB

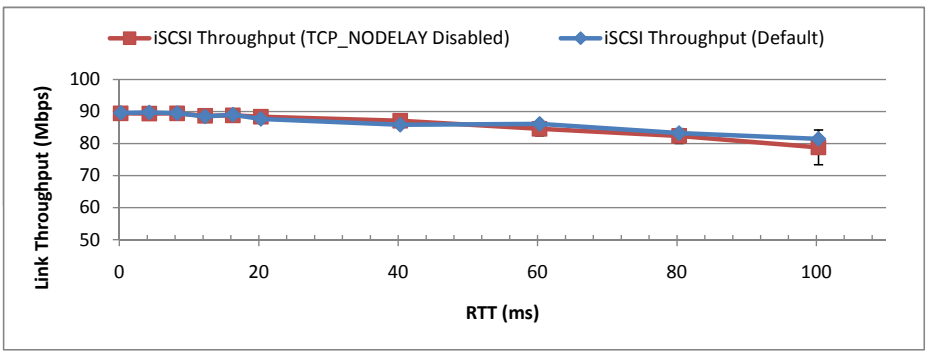
Figure 4.14: TCP_NODELAY and Open-iSCSI Performance (Statically Set Send Buffer Size) - 3.



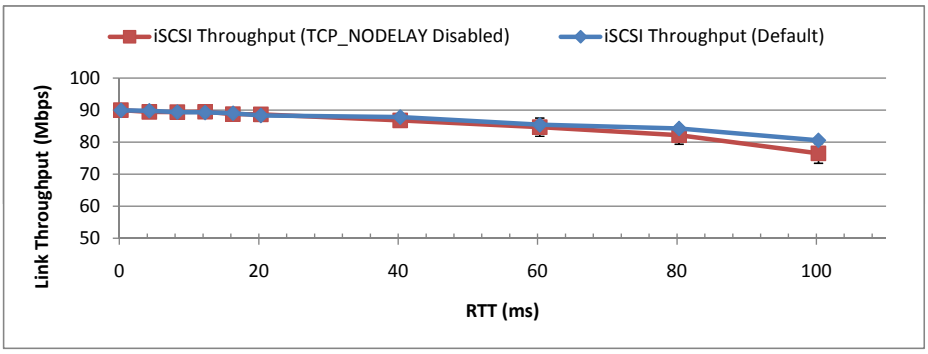
(a) Send Buffer Size 1408 KB



(b) Send Buffer Size 1536 KB



(c) Send Buffer Size 2048 KB



(d) Send Buffer Size 2560 KB

Figure 4.15: TCP_NODELAY and Open-iSCSI Performance (Statically Set Send Buffer Size) - 4.

4.4 `use_clustering` and iSCSI Performance

Further examination of the Open-iSCSI initiator source code reveals that Open-iSCSI disables the SCSI device option `use_clustering` by default. `use_clustering` is the option to control if more than one (memory) page is allowed in one scatter-gather list entry, which is used for DMA in iSCSI and SCSI. With this option disabled, each scatter-gather list entry can only hold one page[7].

We modified the Open-iSCSI source code to enable this `use_clustering` option and support the transmission of scatter-gather list entries containing more than one page. Listing 4.5 and Listing 4.6 show the source code before and after the modification.

```
1  static int iscsi_sw_tcp_xmit_segment(struct iscsi_tcp_conn *tcp_conn,
2                  struct iscsi_segment *segment)
3  {
4     struct iscsi_sw_tcp_conn *tcp_sw_conn = tcp_conn->dd_data;
5     struct socket *sk = tcp_sw_conn->sock;
6     unsigned int copied = 0;
7     int r = 0;
8
9     while (!iscsi_tcp_segment_done(tcp_conn, segment, 0, r)) {
10       struct scatterlist *sg;
11       unsigned int offset, copy;
12       int flags = 0;
13
14       r = 0;
15       offset = segment->copied;
16       copy = segment->size - offset;
17
18       if (segment->total_copied + segment->size < segment->total_size)
19          flags |= MSG_MORE;
20
21       if (!segment->data) {
22          sg = segment->sg;
23          offset += segment->sg_offset + sg->offset;
24             /*
25              * Please note here, the default implementation assumes that
26              * there is only one page in the sg (scatter-gatter) list entry.
27              *
28              */
29          r = tcp_sw_conn->sendpage(sk, sg_page(sg), offset,
30                   copy, flags);
31       } else {
32             ...
33       }
34
35       if (r < 0) {
36          iscsi_tcp_segment_unmap(segment);
37          return r;
38       }
39       copied += r;
40    }
41    return copied;
42 }
```

Listing 4.5: Default Open-iSCSI I/O Segmentation Transmission with `use_clustering` Disabled.

```
1  static int iscsi_sw_tcp_xmit_segment(struct iscsi_tcp_conn *tcp_conn,
2                  struct iscsi_segment *segment)
3  {
```

[7] For more implementation details, please refer to the function `blk_rq_map_sg()` in the Linux kernel source code, the 2.6.34 version of which can be found at `http://lxr.linux.no/linux+v2.6.34/block/blk-merge.c`

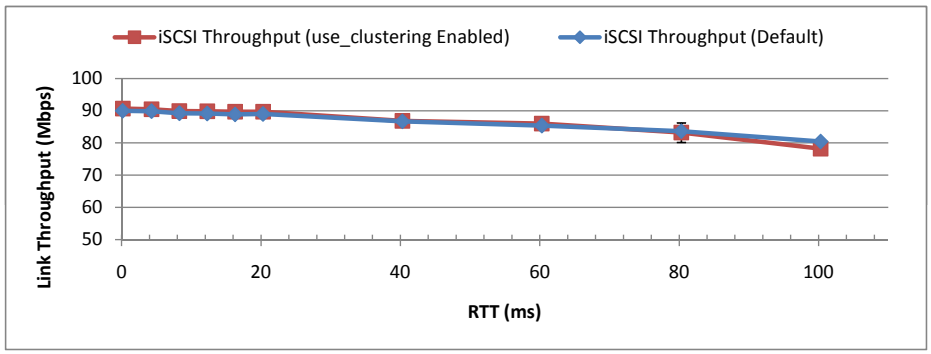


Figure 4.16: `use_clustering` and Open-iSCSI Performance (Dynamically Set Send Buffer Size).

```
4     struct iscsi_sw_tcp_conn *tcp_sw_conn = tcp_conn->dd_data;
5     struct socket *sk = tcp_sw_conn->sock;
6     unsigned int copied = 0;
7     int r = 0;
8
9     while (!iscsi_tcp_segment_done(tcp_conn, segment, 0, r)) {
10       struct scatterlist *sg;
11       unsigned int offset, copy;
12       int flags = 0;
13
14       r = 0;
15       offset = segment->copied;
16       copy = segment->size - offset;
17
18       if (segment->total_copied + segment->size < segment->total_size)
19          flags |= MSG_MORE;
20
21         if (!segment->data) {
22             unsigned int curr_sg_copied = 0;
23             unsigned int page_offset_in_sg_entry = 0;
24             unsigned int num_page_in_sg_entry = 0;
25             int once = 0;
26
27          sg = segment->sg;
28          offset += segment->sg_offset + sg->offset;
29
30
31             if (unlikely(NULL == sg_page(sg))) {
32                 printk(KERN_ERR "sg_page(sg) returns NULL.\n");
33                 break;
34             }
35
36             curr_sg_copied = 0;
37             page_offset_in_sg_entry = offset / PAGE_SIZE;
38
39             /*
40              * Please note that page_count(sg_page(sg)) did not give us
41              * the correct page number in one sg entry in our tests.
42              */
43
44             num_page_in_sg_entry = (sg->offset + sg->length + PAGE_SIZE - 1) >>
                   PAGE_SHIFT;
45
46
47             while ((curr_sg_copied < copy) && (page_offset_in_sg_entry <
                   num_page_in_sg_entry)) {
48                 struct page *page_to_be = NULL;
49                 size_t offset_to_be = 0;
50                 size_t copy_to_be = 0;
```

```
51
52                  once++;
53
54                  // figure out the send params
55                  // which page to send
56
57                  page_to_be = sg_page(sg) + page_offset_in_sg_entry;
58                  offset_to_be = offset % PAGE_SIZE ;
59                  copy_to_be = min_t(size_t, copy - curr_sg_copied, PAGE_SIZE -
                        offset_to_be);
60
61                  r = tcp_sw_conn->sendpage(sk, page_to_be, offset_to_be,
62                      copy_to_be, flags);
63                  if (r < 0) {
64                      iscsi_tcp_segment_unmap(segment);
65                      return r;
66                  }
67
68                  curr_sg_copied += r;
69                  offset += r;
70                  page_offset_in_sg_entry = offset >> PAGE_SHIFT;
71
72              }
73
74              // not necessary, just to be compatible with the default iscsi
                    implementation
75              r = curr_sg_copied;
76
77          } else {
78              ...
79          }
80
81          if (r < 0) {
82              iscsi_tcp_segment_unmap(segment);
83              return r;
84
85          }
86
87          copied += r;
88      }
89
90      return copied;
91  }
```

Listing 4.6: Modified Open-iSCSI I/O Segmentation Transmission with `use_clustering` Enabled.

We measured the iSCSI throughput with and without this modification. Figure 4.16 shows the results with dynamically set send buffer size. The overlap of the two curves demonstrates that with TCP autotuning the difference `use_clustering` makes in iSCSI throughput is insignificant.

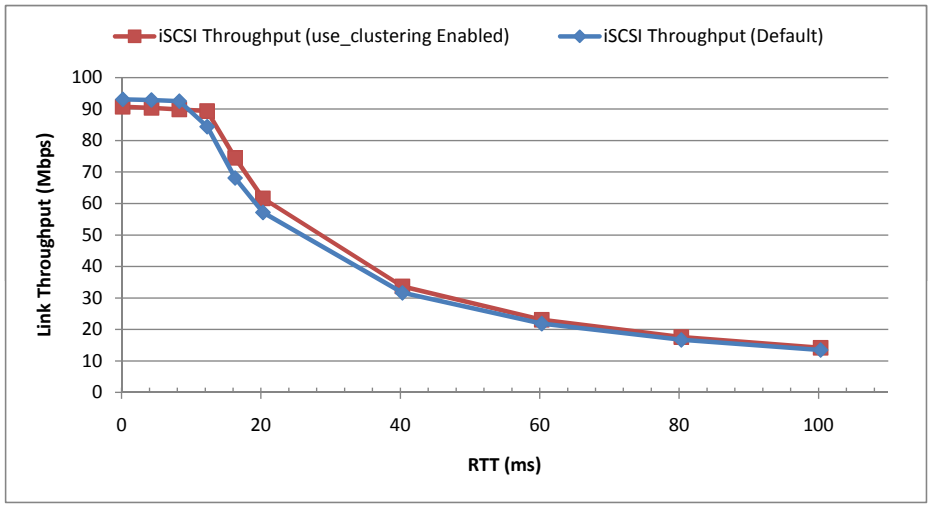
On the other hand, in Figure 4.17 and Figure 4.18 we notice that the default Open-iSCSI implementation (with `use_clustering` disabled) outperforms the modified version (with `use_clustering` enabled) before the send buffer size is increased to the corresponding $S_{max\_benefit}$ (Please refer to Section 4.2.1.1 for more details) at RTTs shorter than or equal to 40 ms. Other than that, even if the RTT is shorter than or equal to 40 ms, as long as the send buffer size is less than or equal to the corresponding $S_{max\_benefit}$, the modified version with `use_clustering` enabled slightly outperforms the default version. This performance gain also extends to the situations with the RTT longer than 40 ms. Moreover, with the RTTs over 60 ms, the throughput of the modified implementation of

iSCSI is at least as high as the throughput of the default implementation.

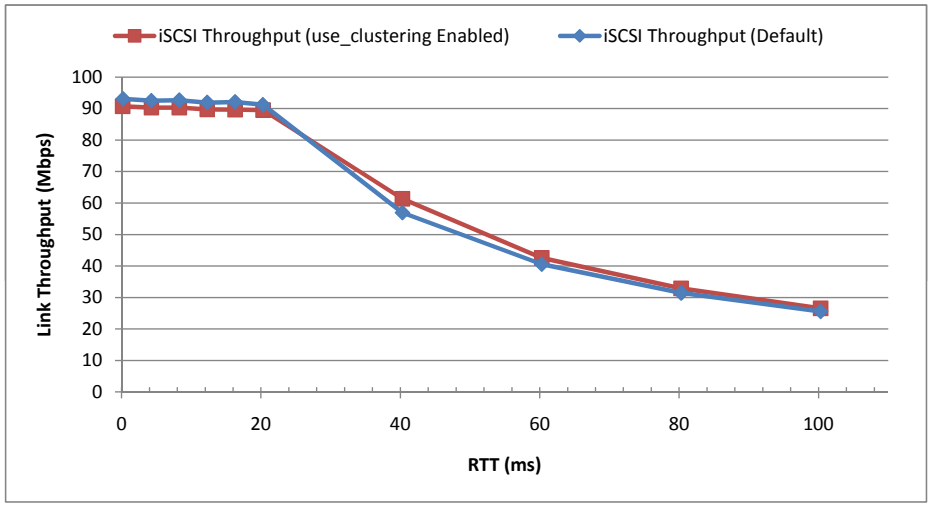
The same pattern continues all the way through all the buffer sizes we used in the test, as shown in Figure 4.19 and Figure 4.20,with an exception of 1152 KB send buffer size, which exposes a case without any significant difference as observed in Figure 4.16. Figure 4.20 also shows that `use_clustering` does not make a significant difference in iSCSI throughput with the size of the send buffer greater than the $S_{max\_benefit}$.

We summarize these results in the following formula:

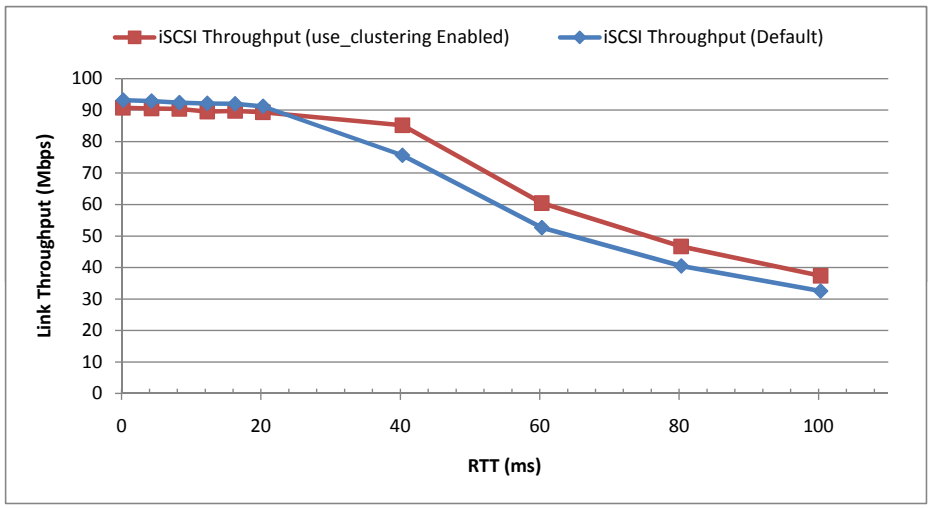
$$\begin{aligned} Throughput_{use\_clustering} \quad &\geq \quad Throughput_{default} \\ \text{is} \quad &\begin{cases} true & \text{if } RTT > 40\text{ms} \\ true & \text{if } RTT \leq 40\text{ms and } S_{send\_buffer} < S_{maxbenefit} \\ false & \text{if } RTT \leq 40\text{ms and } S_{send\_buffer} \geq S_{maxbenefit} \end{cases} \end{aligned} \tag{4.3}$$



(a) Default Send Buffer Size (128 KB)

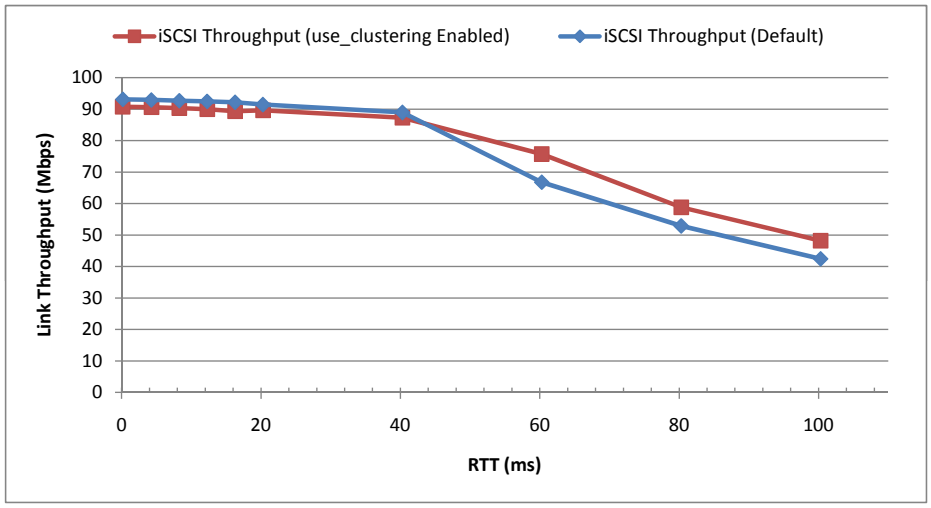


(b) Send Buffer Size 256 KB

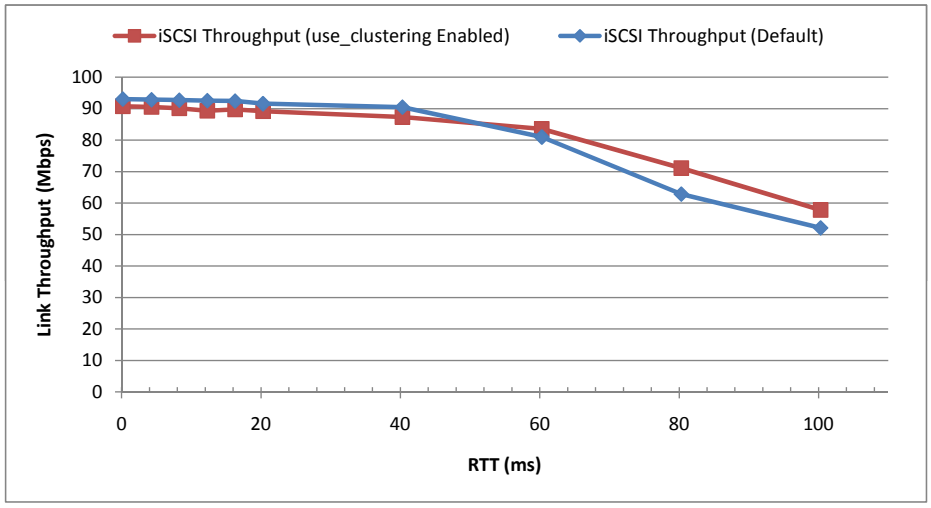


(c) Send Buffer Size 384 KB

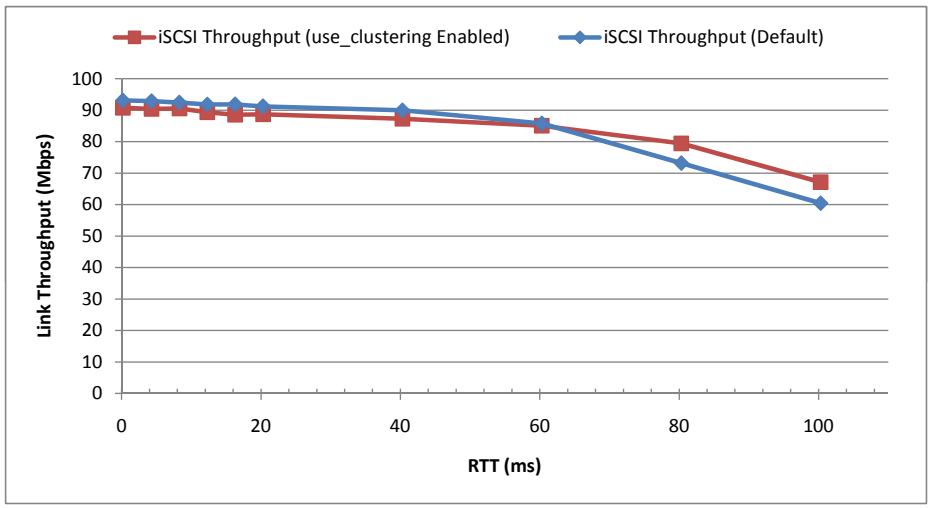
Figure 4.17: `use_clustering` and Open-iSCSI Performance (Statically Set Send Buffer Size) - 1.



(a) Send Buffer Size 512 KB

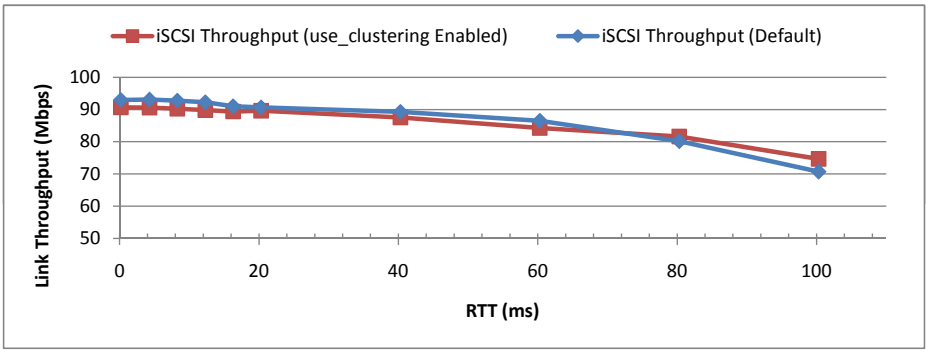


(b) Send Buffer Size 640 KB

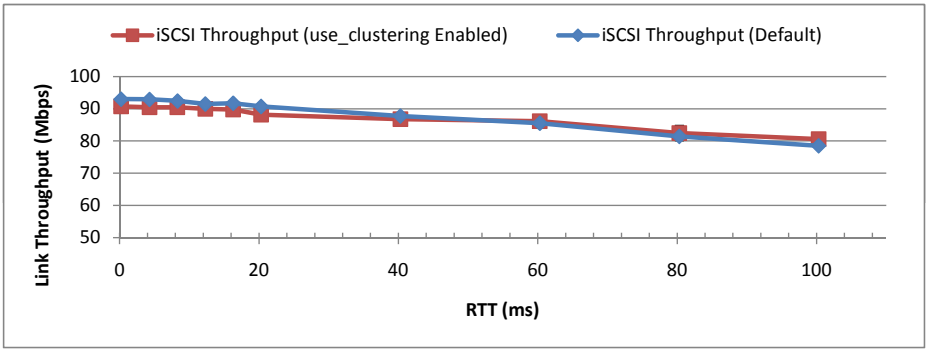


(c) Send Buffer Size 768 KB

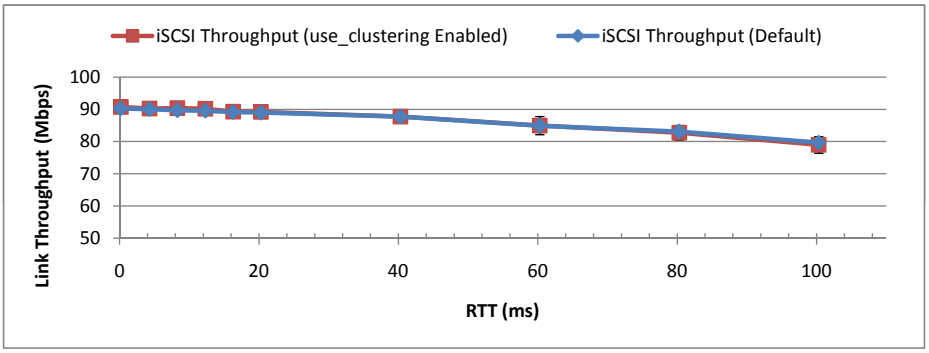
Figure 4.18: `use_clustering` and Open-iSCSI Performance (Statically Set Send Buffer Size) - 2.



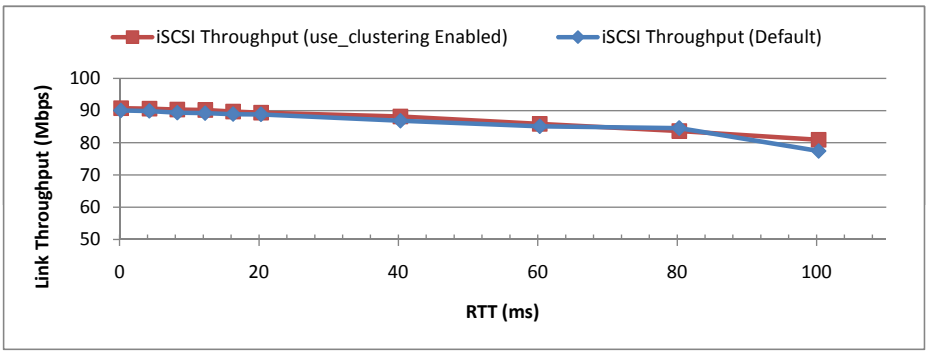
(a) Send Buffer Size 896 KB



(b) Send Buffer Size 1024 KB

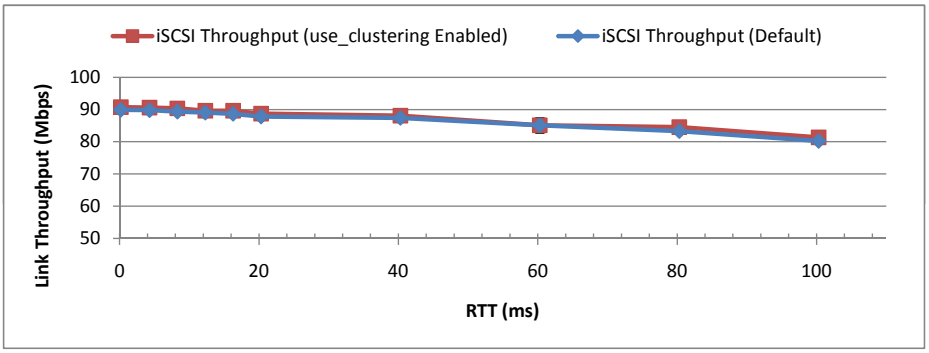


(c) Send Buffer Size 1152 KB

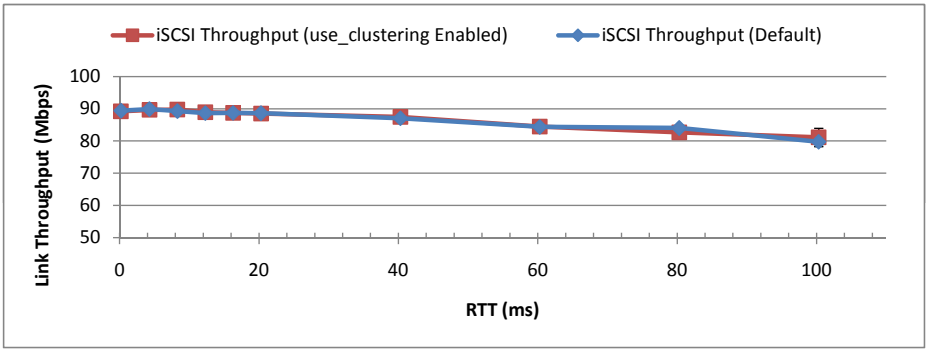


(d) Send Buffer Size 1280 KB

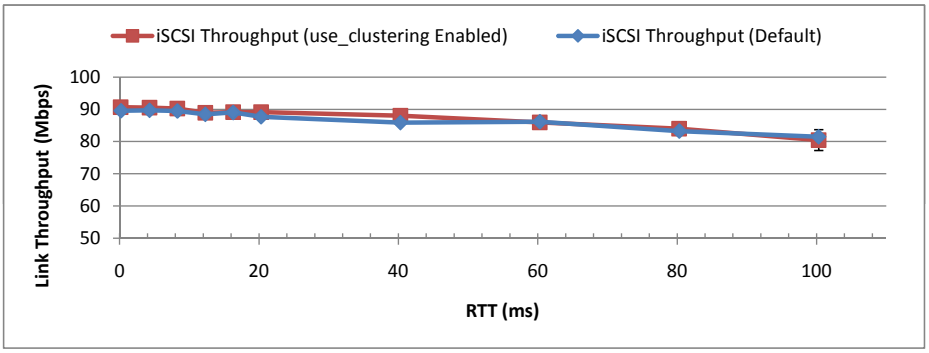
Figure 4.19: `use_clustering` and Open-iSCSI Performance (Statically Set Send Buffer Size) - 3.



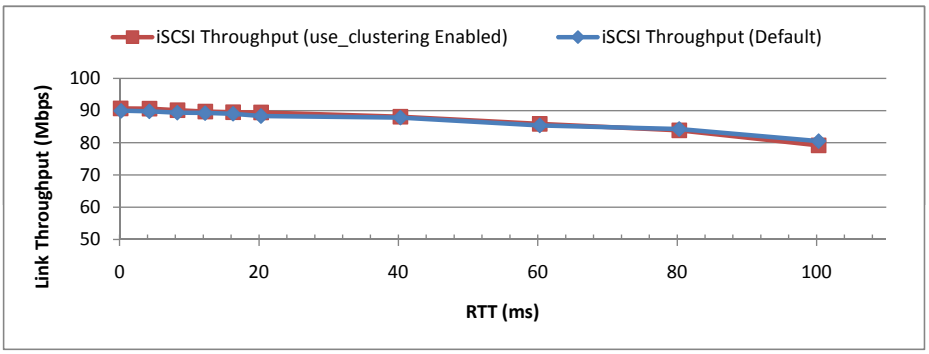
(a) Send Buffer Size 1408 KB



(b) Send Buffer Size 1536 KB



(c) Send Buffer Size 2048 KB



(d) Send Buffer Size 2560 KB

Figure 4.20: `use_clustering` and Open-iSCSI Performance (Statically Set Send Buffer Size) - 4.

4.5 iSCSI Performance Boost Observed with Nagle's Algorithm and `use_clustering`

After seeing the slight difference the `use_clustering` option introduces to the iSCSI performance, we were curious about using Nagle's algorithm (`TCP_NODELAY` disabled) and the `use_clustering` option together.

First, we tested the iSCSI throughput with the modifications under autotuned send buffer size. It turns out that the these two options again do not make a noticeable difference in this case, as shown in Figure 4.21.

Under statically set send buffer size, however, Nagle's algorithm (`TCP_NODELAY` disabled) and `use_clustering` combined together becomes a throughput booster for iSCSI connections over long RTTs and/or with insufficient send buffer size, as shown in Figure 4.22 and Figure 4.23. Figure 4.24 and Figure 4.25 show that the throughput of iSCSI with Nagle's algorithm (`TCP_NODELAY` disabled) and `use_clustering` falls back to the throughput of the default implementation as the send buffer size grows close to $S_{max\_benefit}$ (Section 4.2.1.1).

One disadvantage we also noticed about the modification with `TCP_NODELAY` disabled and `use_clustering` enabled is that with short RTTs, typically shorter than or equal to 40 ms, this modification brings the performance down by about 2 Mbps when the send buffer size is greater than or equal to the corresponding $S_{max\_benefit}$.

The observations found with this modification follow Eq. 4.3, but with the performance gain very much larger.

4.6 Open-iSCSI Initiator Performance Tuning Suggestions

In the previous sections, we discussed the performance of the Open-iSCSI initiator and its three modified versions - with Nagle's algorithm (`TCP_NODELAY` disabled), with the SCSI `use_clustering` option enabled, and with both of the previous two.

The results with statically set send buffer size show that Nagle's algorithm alone does not make a significant difference in the iSCSI throughput.

The `use_clustering` modification brings mild changes to the iSCSI throughput: it achieves slight performance gain if the send buffer size is smaller than the $S_{max\_benefit}$ (Section 4.2.1.1) corresponding to the RTT; Also, it causes slight performance degradation if the RTT is less than or equal to 40 ms and the send buffer size is larger than the corresponding $S_{max\_benefit}$. Eq. 4.3 summarizes this pattern of performance change.

When Nagle's algorithm (`TCP_NODELAY` disabled) and the `use_clustering` option are both applied, we see a similar performance change pattern but with the performance gain very much larger.

To have a closer look at how these four Open-iSCSI implementations (default, with Nagle's

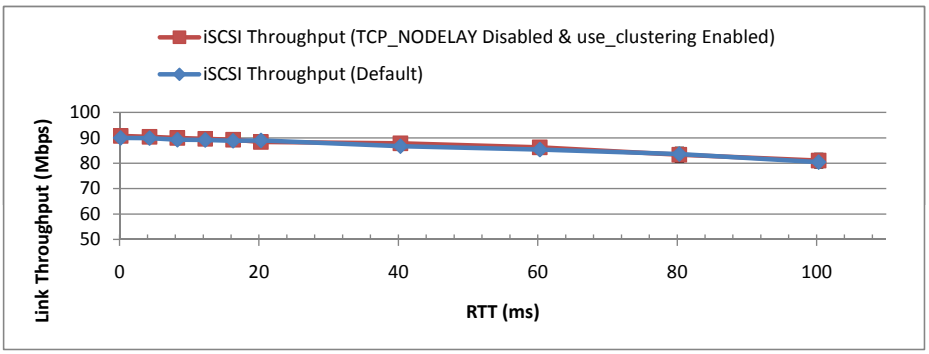


Figure 4.21: Open-iSCSI Performance with `use_clustering` Enabled and `TCP_NODELAY` Disabled (Dynamically Set Send Buffer Size).

algorithm (`TCP_NODELAY` disabled), with the SCSI `use_clustering` option enabled, and with both of the previous two) perform, we plotted the throughput of the four implementations against the send buffer sizes at 40 ms and 100ms, as shown in Figure 4.26 and Figure 4.27.

As for the choice between dynamically set send buffer size, i.e. with TCP autotuning, and statically set send buffer size, the iSCSI session with the buffer size statically set outperforms the one with TCP autotuning when the RTT is less than or equal to 40 ms. As the RTT increases above 40 ms, however, TCP autotuning starts to show its advantage. For more details, please refer to Section 4.2.1.2.

All the results so far have concerned RTTs of 100 ms or less. Cross-continental Internet links, however, can have an average RTT as long as 200 ms[27]; therefore, we also tested the iSCSI throughput over a link with an emulated RTT of 200 ms. The results are shown in Figure 4.28. TCP autotuning did not get an equally high throughput as statically set buffer size can provide. This is because TCP autotuning needs extra tuning to work properly on very long delay links. That is the reason we do not recommend TCP autotuning as the answer to all situations.

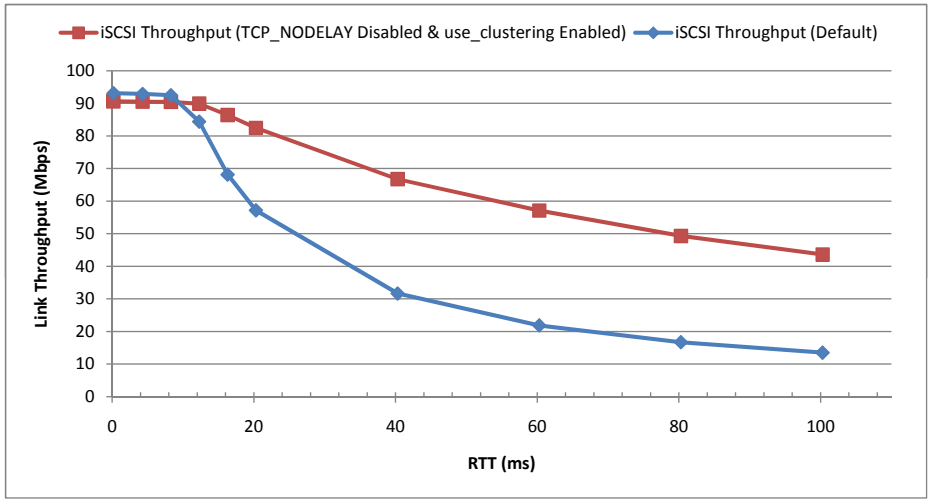
Based on the discussion above, we recommend the an Open-iSCSI initiator tuning scheme to achieve the best performance, as demonstrated in Table 4.3. The “Free Memory Usage” in the table means that the socket send buffer size can be set an arbitrarily large value and the “Restricted Memory Usage” means that there is a socket (send) buffer size cap and the maximum allowed value is not smaller than $S_{max\_benefit}$. The latter case can represent the situations of servers running as virtual machines: if a large number of virtual machines are running on a physical host, the memory share of each individual virtual machine is relatively small; therefore, the virtual server is running under memory size restrictions and thus cannot afford all the open sockets with large buffer sizes.

Please note that one can also choose TCP autotuning with the default Open-iSCSI implementation as an alternative, which provides the convenience of spending the least effort on configuration at the expense of a certain amount of performance sacrifice.

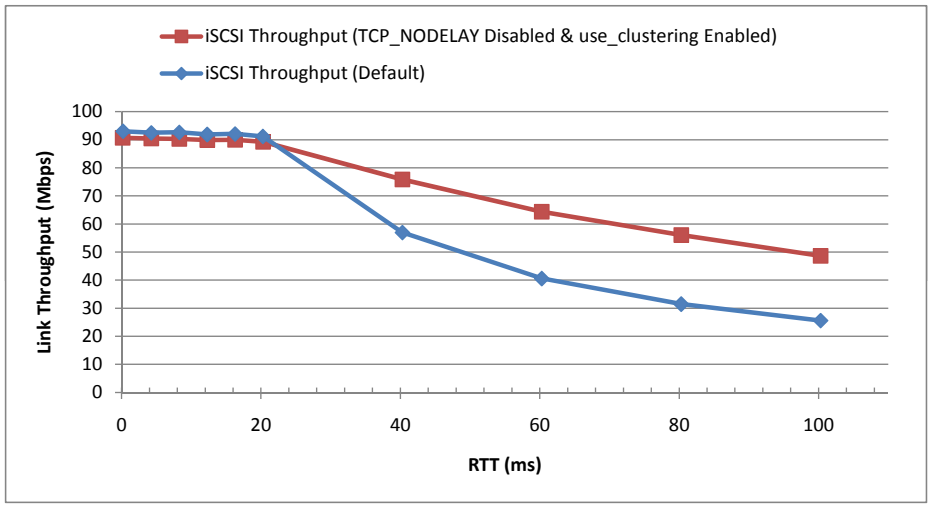
Table 4.3: Recommended Tuning Scheme of Open-iSCSI

<table>
<tr><th>RTT (ms)</th><th>Free Memory Usage</th><th>Restricted Memory Usage</th></tr>
<tr><td rowspan="2">$\leqslant 40$, stable</td><td>Statically set the send buffer size to $S_{max\_benefit}$.</td><td rowspan="6">Statically set the send buffer size to the maximum value allowed.

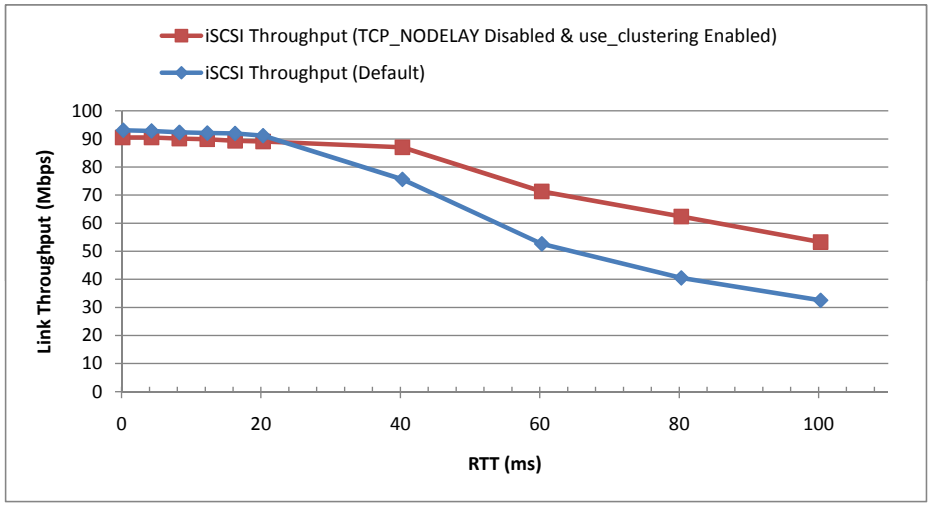
Choose the Open-iSCSI implementation with Nagle's algorithm (<code>TCP_NODELAY</code> disabled) and the SCSI <code>use_clustering</code> option enabled.</td></tr>
<tr><td>Choose the default Open-iSCSI implementation.</td></tr>
<tr><td rowspan="2">> 40, stable</td><td>Statically set the send buffer size to $S_{max\_benefit}$.</td></tr>
<tr><td>Choose the Open-iSCSI implementation with Nagle's algorithm (<code>TCP_NODELAY</code> disabled) and the SCSI <code>use_clustering</code> option enabled.</td></tr>
<tr><td rowspan="2">$\leqslant 40$, variable</td><td>Statically set the send buffer size to $S_{max\_benefit}$ according to the estimated mean RTT.</td></tr>
<tr><td>Choose the Open-iSCSI implementation with Nagle's algorithm (<code>TCP_NODELAY</code> disabled) and the SCSI <code>use_clustering</code> option enabled.</td></tr>
<tr><td rowspan="2">> 40, variable</td><td>Use TCP autotuning with caution.</td><td rowspan="2">Choose the Open-iSCSI implementation with Nagle's algorithm (<code>TCP_NODELAY</code> disabled) and the SCSI <code>use_clustering</code> option enabled.</td></tr>
<tr><td>Choose the Open-iSCSI implementation with Nagle's algorithm (<code>TCP_NODELAY</code> disabled) and the SCSI <code>use_clustering</code> option enabled.</td></tr>
</table>



(a) Default Send Buffer Size (128 KB)

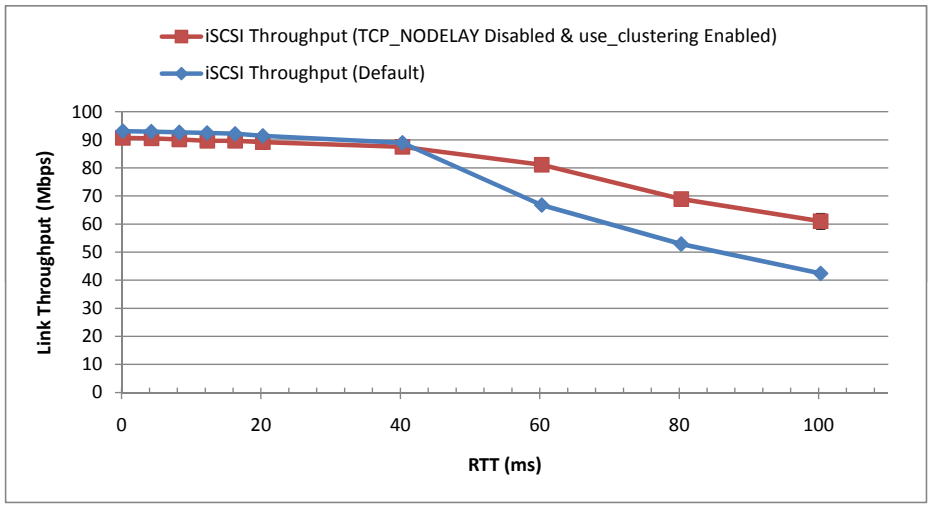


(b) Send Buffer Size 256 KB

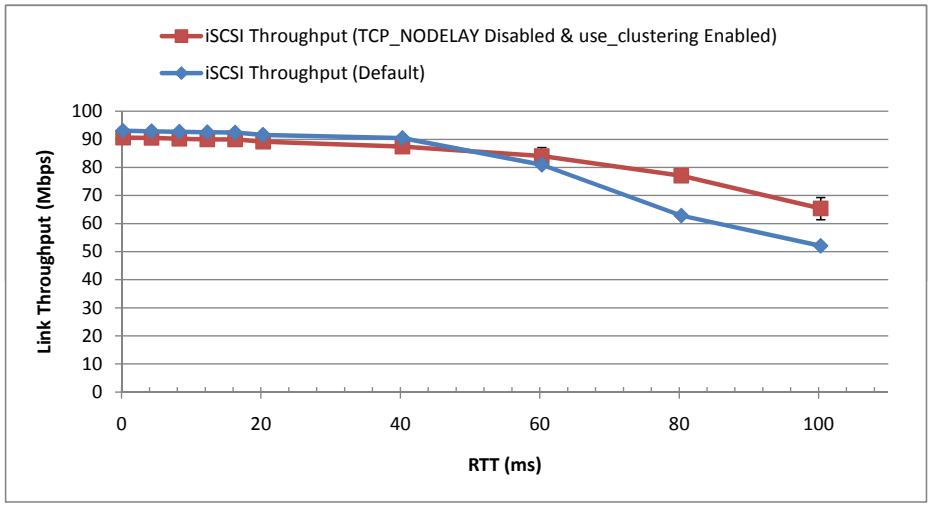


(c) Send Buffer Size 384 KB

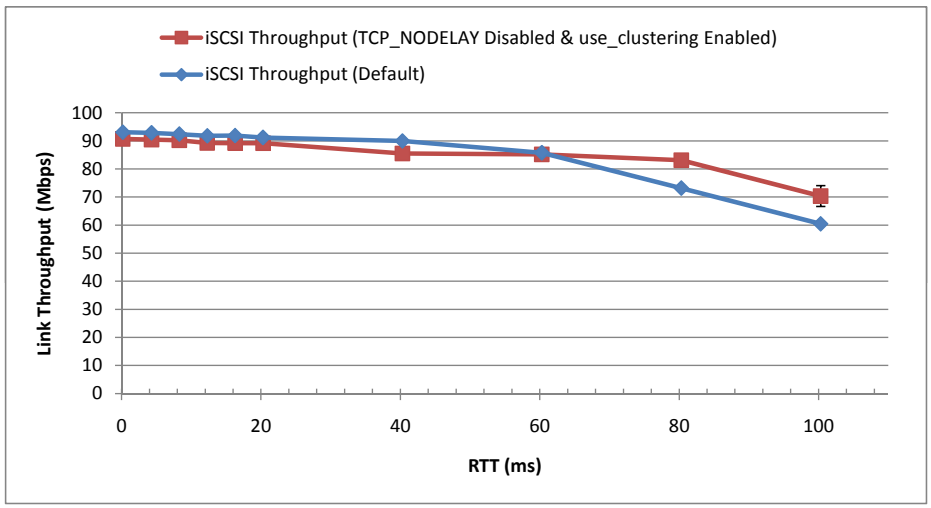
Figure 4.22: Open-iSCSI Performance with `use_clustering` Enabled and `TCP_NODELAY` Disabled (Statically Set Send Buffer Size) - 1.



(a) Send Buffer Size 512 KB

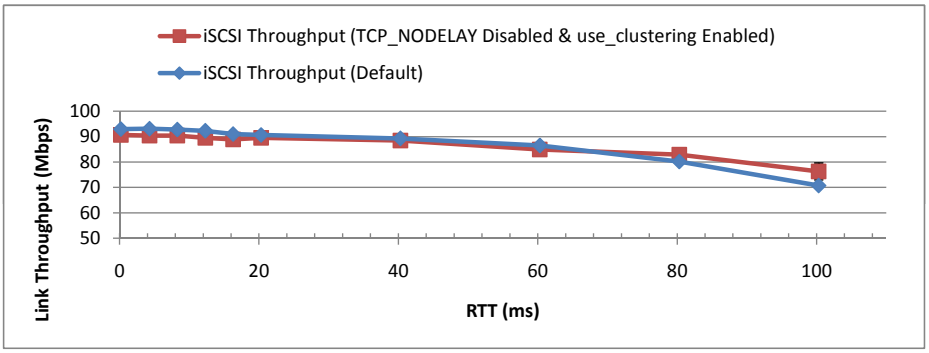


(b) Send Buffer Size 640 KB

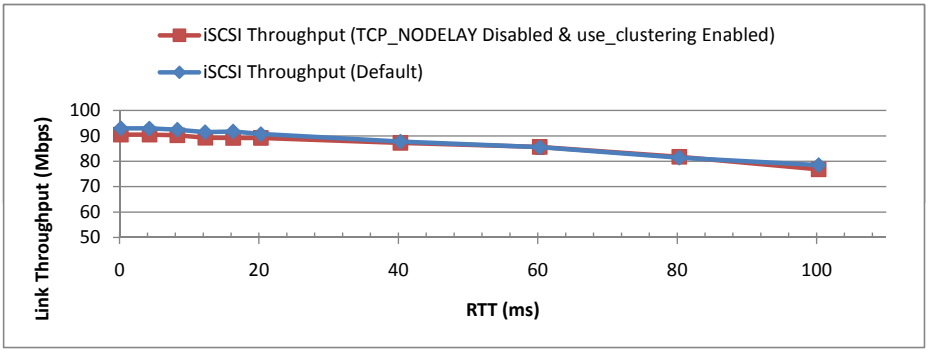


(c) Send Buffer Size 768 KB

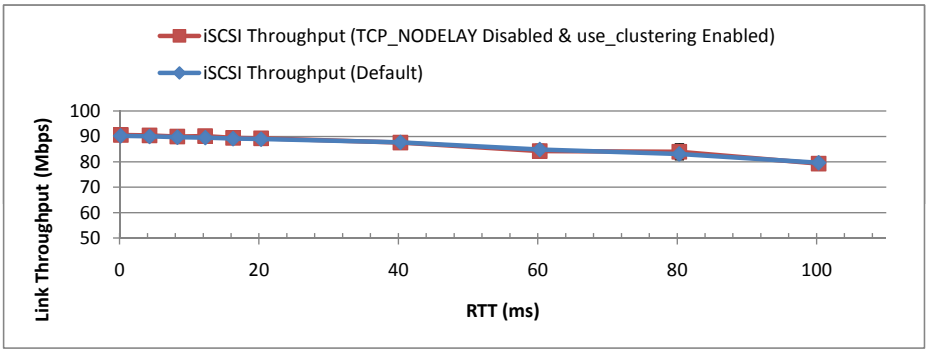
Figure 4.23: Open-iSCSI Performance with `use_clustering` Enabled and `TCP_NODELAY` Disabled (Statically Set Send Buffer Size) - 2.



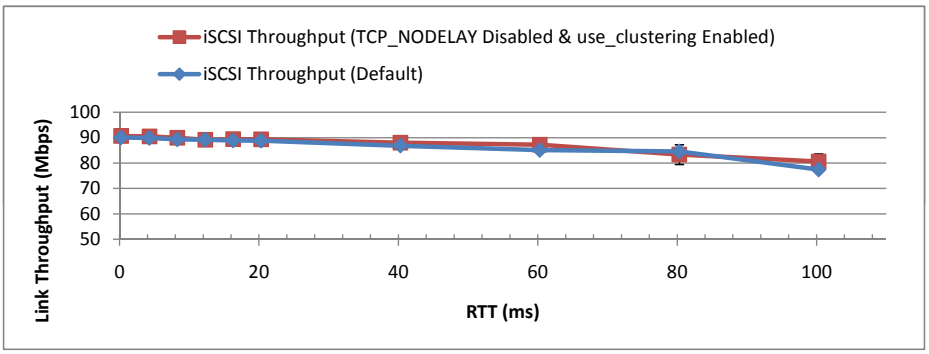
(a) Send Buffer Size 896 KB



(b) Send Buffer Size 1024 KB

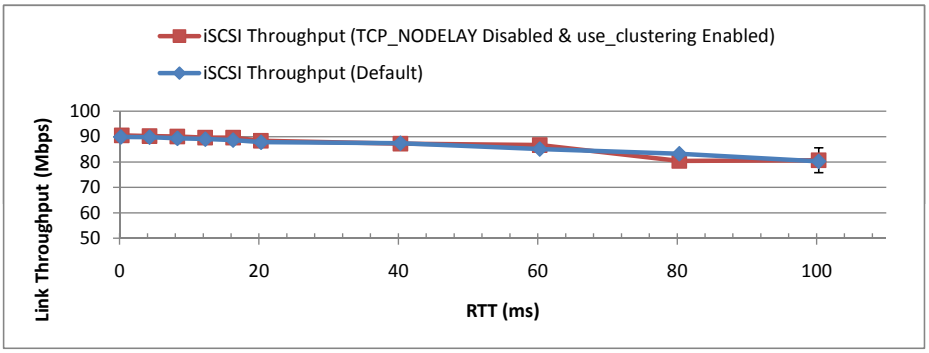


(c) Send Buffer Size 1152 KB

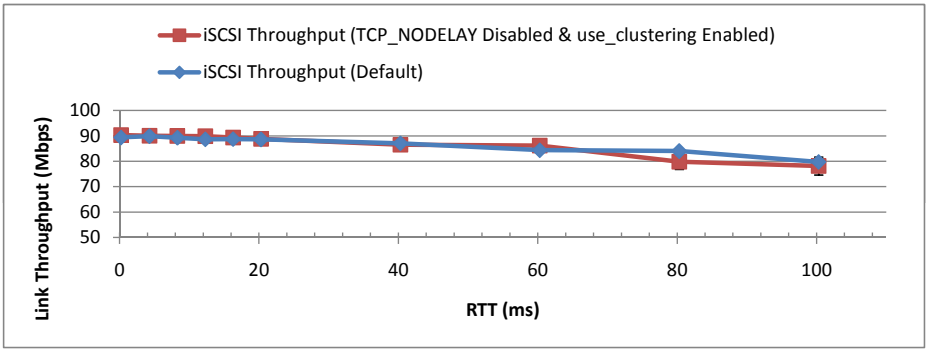


(d) Send Buffer Size 1280 KB

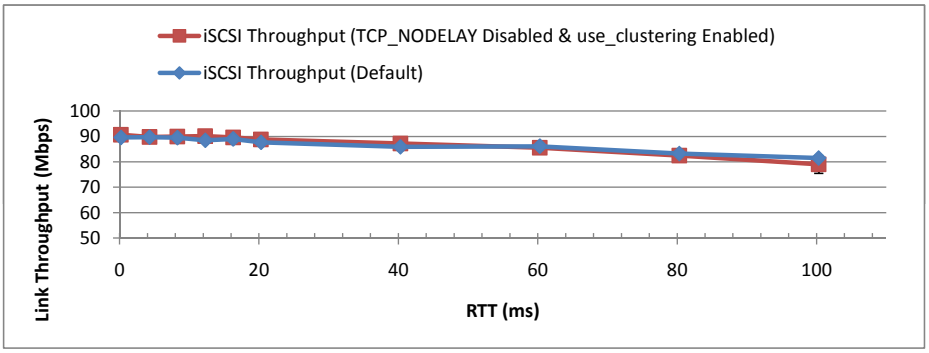
Figure 4.24: Open-iSCSI Performance with `use_clustering` Enabled and `TCP_NODELAY` Disabled (Statically Set Send Buffer Size) - 3.



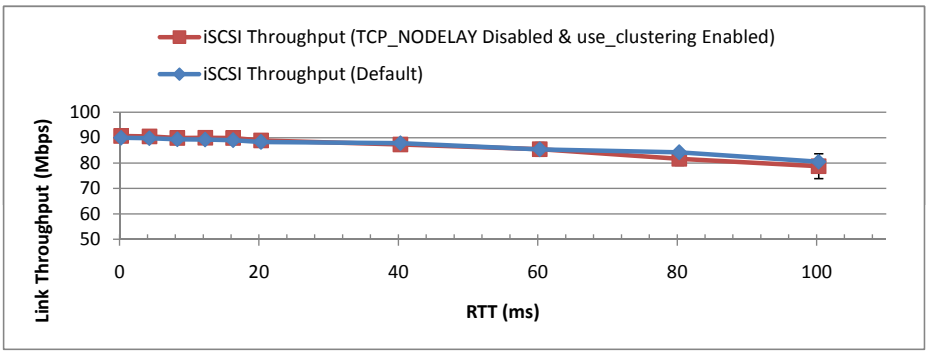
(a) Send Buffer Size 1408 KB



(b) Send Buffer Size 1536 KB



(c) Send Buffer Size 2048 KB



(d) Send Buffer Size 2560 KB

Figure 4.25: Open-iSCSI Performance with `use_clustering` Enabled and `TCP_NODELAY` Disabled (Statically Set Send Buffer Size) - 4.

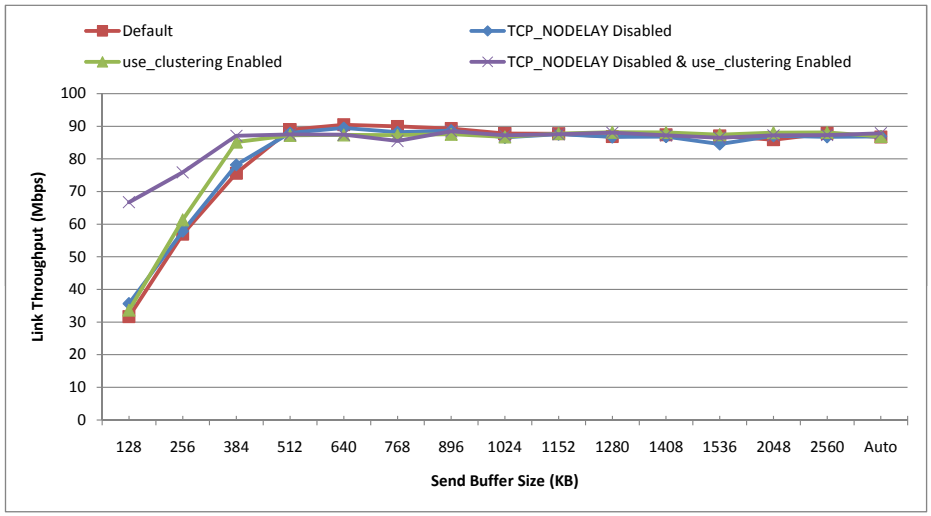


Figure 4.26: Open-iSCSI Performance Comparison @ 40 ms RTT

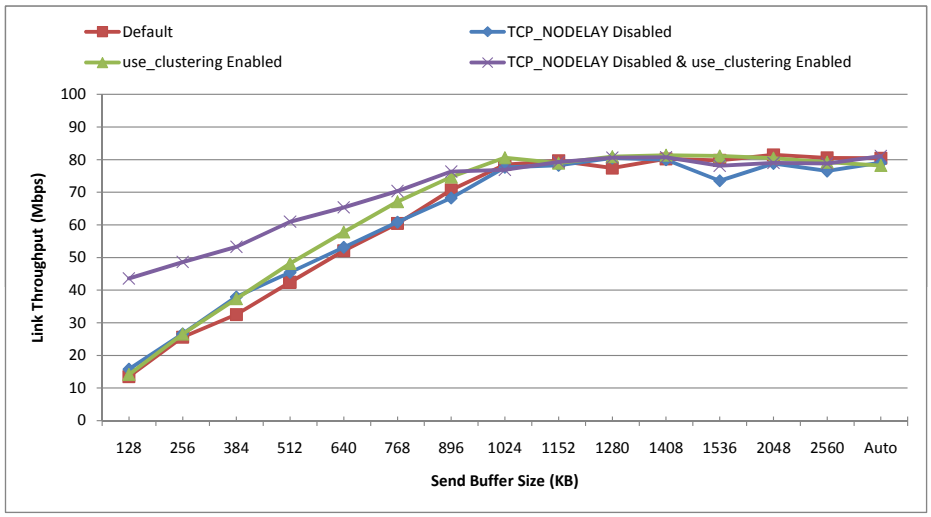


Figure 4.27: Open-iSCSI Performance Comparison @ 100 ms RTT

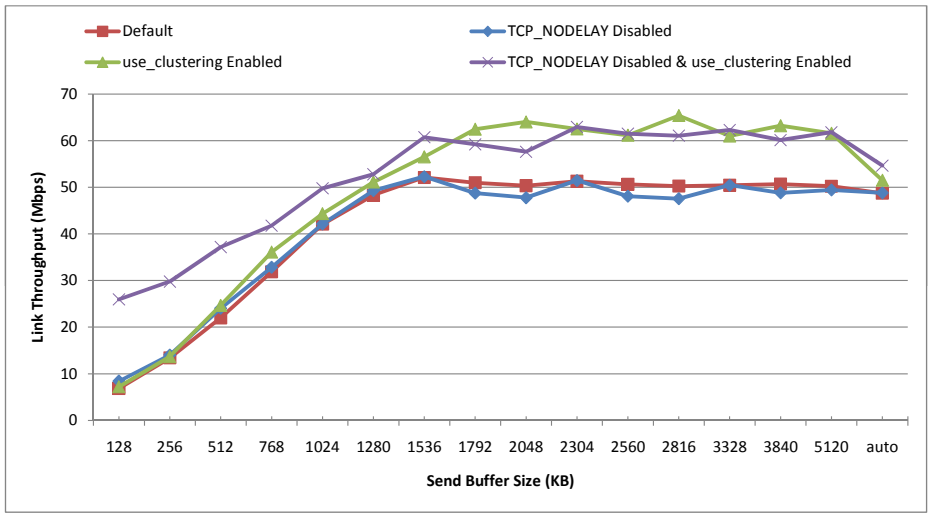


Figure 4.28: Open-iSCSI Performance Comparison @ 200 ms RTT

Chapter 5

Conclusions and Future Work

In this thesis, we first discussed and examined the feasibility and effectiveness of using the native Linux traffic control components, such as `tc` and HTB (Hierarchical Token Bucket) to regulate the network traffic rate in order to provide differentiated quality of service for the disk traffic of virtual machines and throttle the traffic rate of the physical host as well. Our results in Chapter 3 demonstrate that HTB is capable of achieving the network traffic throttling goal with high accuracy. In addition, HTB can redistribute the available bandwidth dynamically between concurrent traffic flows when one or more flows are not fully utilizing the assigned bandwidth; therefore, HTB not only accurately maintains the global bandwidth regulation goal, but also makes the most efficient use of the available bandwidth. Moreover, when the traffic control policy is required to change dynamically, the switch between different HTB policies leaves minimal footprint in the traffic profile. Therefore, HTB can be used as an effective bandwidth regulation tool to meet the demand in providing differentiated quality of service.

In the course of this investigation, however, we noticed that the HTB implementation is not very well documented. Thus, also in Chapter 3, we provided details about what we consider the most critical implementation details of HTB. We believe this information can provide more solid reference for others working in similar areas.

In addition, we examined a drastic performance degradation of the Open-iSCSI initiator. We thoroughly tested the performance of the Open-iSCSI initiator and its three modified versions - with Nagle's algorithm (`TCP_NODELAY` disabled), with the SCSI `use_clustering` option enabled, and with both of the previous two - under two methods of setting the TCP send buffer size - statically and dynamically. The results with statically set send buffer size show that Nagle's algorithm alone does not make a significant difference in the iSCSI throughput. The `use_clustering` modification brings mild changes to the iSCSI throughput: It achieves slight performance gain but causes minor performance degradation under certain circumstances as well. When Nagle's algorithm (`TCP_NODELAY` disabled) and the `use_clustering` option are both applied, we see a performance change pattern similar to the pattern the `use_clustering` modification exhibits, but with the performance gain very much larger.

As for the choice between dynamically set send buffer size, i.e. with TCP autotuning, and statically set send buffer size, the iSCSI session with the buffer size statically set outperforms the one with TCP autotuning when the RTT is less than or equal to 40 ms. As the RTT further increases, however, TCP autotuning starts to show its advantage.

Based on these results, we proposed a performance tuning scheme for the Open-iSCSI initiator, which covers network links with varying combinations of RTTs and link bandwidth. We believe that by following this tuning scheme, users of Open-iSCSI, especially those using Open-iSCSI over an LFN, can gain significant throughput benefits.

5.1 Future Work

The core theory behind the implementation of HTB is DRR, which stands for deficit round-robin. Without the priority parameter, HTB, as mentioned in Section 2.4, does not give sufficient consideration to latency-critical flows; therefore, HTB, without differentiating the priority of classes, may not be able to schedule latency-critical flows effectively when they are mixed with best-effort flows. Even with the priority parameter, HTB may not be able to achieve the desired results either because it handles this parameter in a relatively naive manner: Classes with a higher priority are always served first, which can result in starving the classes with lower priorities. Thus, mapping latency-critical and best-effort flows to different priorities may not always give desirable scheduling results. Further investigation into this matter is of importance as it will provide valuable insight into the scheduling effectiveness of HTB. If the outcome of this investigation suggests non-optimal scheduling of HTB, it would be of interest to explore the option of re-implementing HTB using DRR++.

As for the performance of the Open-iSCSI initiator, future work should explore the possibility of implementing an autotuning component for the Open-iSCSI initiator so that end users, who may not possess the required privileges / knowledge / skills for performance tuning, would no longer have to be concerned with the potential performance loss. Also, there is still a 10 percent throughput gap between the highest iSCSI throughput we have achieved so far and the throughput of a general TCP session at the RTT of 100 ms. The cause of this throughput gap is certainly worthwhile investigating, which may enlighten us on the possibility of endowing Open-iSCSI with this last bit of performance gain.

Bibliography

[1] /proc/sys/net/ipv4/* variables. http://lxr.linux.no/linux+v2.6.34/Documentation/networking/ip-sysctl.txt, May 2010.

[2] The iSCSI Enterprise Target project. http://iscsitarget.sourceforge.net, July 2010.

[3] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke. A performance analysis of the iSCSI protocol. In *MSS '03: Proceedings of the 20 th IEEE/11 th NASA Goddard Conference on Mass Storage Systems and Technologies (MSS'03)*, page 123, Washington, DC, USA, 2003. IEEE Computer Society.

[4] A. Aizman and D. Yusupov. Open-iSCSI project: Open-iSCSI - RFC3720 architecture and implementation. http://www.open-iscsi.org/, February 2005.

[5] W. Almesberger. Linux network traffic control - implementation overview. In *Proceedings of 5th Annual Linux Expo*, pages 153–164, May 1999.

[6] W. Almesberger. Traffic control - next generation. http://tcng.sourceforge.net/, October 2004.

[7] A. Bianco, J. Finochietto, M. Modesti, and F. Neri. Distributed storage on networks of Linux PCs using the iSCSI protocol. In *High Performance Switching and Routing, 2008. HSPR 2008. International Conference on*, pages 252 –256, 2008.

[8] M. A. Brown. Traffic control howto. http://tldp.org/HOWTO/Traffic-Control-HOWTO/index.html, October 2006.

[9] M. Devera. Hierarchical token bucket theory. http://luxik.cdi.cz/~devik/qos/htb/manual/theory.htm, May 2002.

[10] M. Devera and D. Cohen. HTB Linux queuing discipline manual - user guide. http://luxik.cdi.cz/~devik/qos/htb/manual/userg.htm, May 2002.

[11] T. Dunigan. TCP auto-tuning zoo. http://www.csm.ornl.gov/~dunigan/netperf/auto.html, February 2006.

[12] C. Gauger, M. Kohn, S. Gunreben, D. Sass, and S. Perez. Modeling and performance evaluation of iSCSI storage area networks over TCP/IP-based MAN and WAN networks. In *Broadband Networks, 2005. BroadNets 2005. 2nd International Conference on*, pages 850 –858 Vol. 2, 7-7 2005.

[13] B. Hubert. Linux advanced routing and traffic control howto. http://lartc.org/howto/, March 2004.

[14] J. L. Hufferd. *iSCSI: The Universal Storage Connection*, volume 1. Addison-Wesley Professional, November 2002.

[15] V. Jacobson, R. Braden, and D. Borman. RFC1323 - TCP extensions for high performance. http://www.ietf.org/rfc/rfc1323.txt, 1992.

[16] A. Kuznetsov and D. Torokhov. Token bucket filter queue. http://lxr.linux.no/linux+v2.6.33/net/sched/sch_tbf.c#L1, April 2010.

[17] Lawrence Berkeley National Laboratory. TCP tuning guide. http://fasterdata.es.net/TCP-tuning/linux.html, July 2010.

[18] Y. Lu, F. Noman, and D. H. C. Du. Simulation study of iSCSI-based storage system. In *12th NASA Goddard and 21st IEEE Conference on Mass Storage Systems and Technologies (MSST2004*, pages 399–408, 2004.

[19] M. H. MacGregor and W. Shi. Deficits for Bursty Latency-Critical Flows: DRR++. In *ICON '00: Proceedings of the 8th IEEE International Conference on Networks*, page 287, Washington, DC, USA, 2000. IEEE Computer Society.

[20] M. Mathis and R. Reddy. Enabling high performance data transfers. `http://www.psc.edu/networking/projects/tcptune/`, February 2008.

[21] J. Nagle. RFC896 - congestion control in IP/TCP Internetworks. `http://www.faqs.org/rfcs/rfc896.html`, January 1984.

[22] Z. Pang. High Performance Live Migration over Low-Bandwidth, High-Delay Network with Loss Prevention. Master's thesis, University of Alberta, Edmonton, AB, Canada, 2010.

[23] Y. Sheng. Dynamic Network Resource Allocation. Master's thesis, University of Alberta, Edmonton, AB, Canada, 2010.

[24] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. *SIGCOMM Comput.Commun.Rev.*, 25(4):231–242, 1995.

[25] B. Tierney. TCP tuning techniques for high-speed wide-area networks. `http://fasterdata.es.net/TCP-tuning/TCP-Tuning-Tutorial.pdf`, June 2005.

[26] U. Troppens, R. Erkens, W. Mueller-Friedt, R. Wolafka, and N. Haustein. *Storage Networks Explained: Basics and Application of Fibre Channel SAN, NAS, iSCSI,InfiniBand and FCoE*. Wiley Publishing, 2009.

[27] Q. Ye, D. Cui, Z. Wang, L. Wang, and M. H. MacGregor. Long-haul transmission performance in the internet. In *CNSR '10: Proceedings of the 2010 8th Annual Communication Networks and Services Research Conference*, pages 387–395, Washington, DC, USA, 2010. IEEE Computer Society.

[28] C. Zhang and M. H. MacGregor. Scheduling latency-critical traffic: a measurement study of DRR+ and DRR++. In *High Performance Switching and Routing, 2002. Merging Optical and IP Technologies. Workshop on*, pages 262–267, 2002.