# Improving TCP Throughput for Cloud Applications

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2014

Takamichi NISHIJIMA

# List of Publications

## Journal Papers

1. T. Nishijima, H. Ohsaki, and M. Imase, "Automatic parallelism tuning mechanism for heterogeneous IP-SAN protocols in long-fat networks," *Journal of Information Processing*, vol. 21, no. 3, pp. 423–432, July 2013.

2. T. Nishijima, N. Yokoi, Y. Nakamoto, and H. Ohsaki, "Estimation of Performance Improvement Derived from TCP/IP Offload Engine with Software Emulation," *International Journal of Computers & Technology*, vol. 12, no. 1, pp. 3117–3130, Dec. 2013.

## Refereed Conference Papers

1. T. Nishijima, F. Inoue, H. Ohsaki, Y. Nomoto, and M. Imase, "On maximizing IP-SAN throughput over TCP connections with automatic parallelism tuning for long-fat networks," in *Proceedings of the Third Workshop on Middleware Architecture in the Internet* (MidArc 2009), pp. 251–254, July 2009.

2. T. Nishijima, H. Ohsaki, Y. Nomoto, and M. Imase, "Performance evaluation of block device layer with automatic parallelism tuning using heterogeneous IP-SAN protocols," in *Proceedings of the First Workshop on High Speed Network and Computing Environments for Scientific Applications* (HSNCE 2010), pp. 343–346, July 2010.

## Non-Refereed Technical Papers

1. T. Nishijima, F. Inoue, H. Ohsaki, and M. Imase, "Automatic parallelism tuning mechanism for IP-SAN protocols in long-fat networks," in *Prodeedings of the IEICE General Conference 2009*, p. 243, Mar. 2009 *(in Japanese)*.

2. T. Nishijima, H. Ohsaki, and M. Imase, "Performance evaluation of IP-SAN throughput maximization mechanism with block device layer," in *Proceedings of the ITRC meet 28*, Nov. 2010 *(in Japanese)*.

3. Y. Morisita, T. Nishijima, H. Ohsaki, N. Yokoi, K. Nakagawa, N. Kohinata, and M. Imase, "Vose: Virtual offloading with software emulation for estimating performance improvement with TCP/IP protocol offloading," *Technical report of IEICE* (IN2011-79), pp. 1–6, Oct. 2011 *(in Japanese)*.

4. T. Nishijima, Y. Nakai, H. Ohsaki, N. Yokoi, K. Nakagawa, C. Lethanhman, Y. Takatani, and M. Imase, "On the impact of network environment on remote desktop protocols," *Technical report of IEICE* (CQ2012-21), pp. 23–28, July 2012 *(in Japanese)*.

5. Y. Urata, T. Nishijima, N. Yokoi, Y. Nakamoto, and H. Ohsaki, "On the effect of computer virtualization on the accuracy of network emulator," in *Proceedings of the Society Conference of IEICE 2012*, p. 283, Sept. 2012 *(in Japanese)*.

6. Y. Hayashi, T. Nishijima, and H. Ohsaki, "A method of smoothing burst traffic on OpenFlow by adding delay to packets," in *Prodeedings of the IEICE General Conference 2013*, Mar. 2013 *(in Japanese)*.

# Preface

With rapid advancements and developments of networking technologies, various cloud applications have emerged. In this thesis, we focus on cloud applications the performances of which TCP dominantly determines, and we improve performance of the applications by improving TCP performance. TCP performance issues are categorized by the number of TCP connections that derive issues; one or many connections. Because issues with one connection are always critical even if many connections are established, in this thesis, we tackle issues that are derived from performance degradation of one connection. Two methods for improving TCP performance are categorized depending on whether TCP is modified. Because of installability into current environments, we improve TCP performance with existing TCP, i.e., without modifying

Cloud applications are categorized depending on how devices such as servers and clients are located. One is a *DC-DC (Data Center-Data Center) cloud application* wherein servers are located on geographically distant data centers. DC-DC cloud applications mainly transfer large bulk data from a data center to another data center without making users intervene. The other is a *DC-CE (Data Center-Customer Equipments) cloud application* wherein a server is located a data center and a client, i.e., a customer equipment, is located at an edge of a wire area communication network. DC-CE cloud applications mainly transfer small data between a data center and customer equipments interactively. DC-DC and DC-CE applications have their own requirements to service quality: application-level performance and subjective performance. The reason of the difference is who uses an application. Since a user of a DC-DC application is a machine, only the applicaiton-level performance is focused on. Whereas, since a user of a DC-CE application is a human being, the subjective performance should be focused on as well as the applicaiton-level performance.

A key issue to provide good performance is how high TCP throughput is provided to the both applications because most cloud application use TCP for data transfer and because delays are not

controlled. However, requirements to providing high TCP throughput are not trivial and they depend on the both applications depending on their differences of required performance and network characteristics.

In the first place of this thesis, we tackle a couple of performance issues for DC-DC cloud applications. Because some DC-DC cloud applications are implemented as dedicated hardware devices for providing good performance, the DC-DC could applications still have a couple of open issues. One is a performance improvement with current hardware devices wherein changes of TCP protocols and devices are difficult, and the other is a design of hardware devices for providing good performance.

First, we focus on a performance improvement with existing dedicated hardware devices. IP-based Storage Area Network (IP-SAN), which connects storage devices located on distant data centers on an IP network, is a DC-DC cloud application wherein many dedicated hardware devices that perform protocol processing with hardware are deployed. Therefore, we propose *Block Device Layer with Automatic Parallelism Tuning (BDL-APT)*, a mechanism that maximizes the transfer speed of heterogeneous IP-SAN protocols in high delay networks. We propose block device layer, which is a new layer between IP-SAN protocols and applications, and parallelizing data transfer at a block device layer for improving performance of current DC-DC cloud applications. We evaluate the performance of BDL-APT through experiment using several IP-SAN protocols. Through our experiment, we show that BDL-APT realizes high performance of heterogeneous IP-SAN protocols in various network environments.

Second, we focus on a design of hardware devices for providing good performance. TCP/IP Offload Engine (TOE) has been studied by many researchers to solve the CPU bottleneck, but it is not clear which protocol processing should be performed with hardware for future DC-DC cloud applications. A method for easily estimating TCP/IP performance improvements derived from different type of TOE devices is required; i.e., an estimation method without implementing TOE devices really. Therefore, we propose a support method that is a technique for measuring TCP/IP performance improvements derived from different type of TOE devices without implementing TOE prototypes really. Our approach estimates performance improvements without requiring a hardware TOE device by virtually emulating TOE processing on both source and destination end hosts. We extensively examine the accuracy of virtual offloading, by comparing performance, i.e., end-to-end performance and CPU processing overhead, between our approach and a dedicated TOE device.

Moreover, we estimate performance improvements that are derived from several TOE devices of IPsec and combinations of those devices. Consequently, we show that performance improvements which are derived from TOE devices can be estimated correctly.

In the second place of this thesis, we tackle a performance issue for DC-CE cloud applications. Performances of DC-CE applications would be correlated with subjective performance. However, impacts of transport-level performance on subjective performance have not been clear sufficiently. In this thesis, because we focus on issues that are derived from TCP performance and a response time is determined by combinations of TCP throughput and delays, we mainly focus on the response time as a subjective performance of DC-CE cloud applications.

We focus on impact of transport-level performance on subjective performance of remote desktops in general wireless network environments. For elucidating essential relation, we categorize whole remote desktop protocols by their mechanisms for transferring updated screen information. One is an *image transfer mechanism*, which transfers the updated screen information as images, and the other is an *event transfer mechanism*, which transfers the information as update events, e.g., a window creation, setting of window positions and renderings of pictures or texts. We evaluate QoE of respective transfer mechanisms for elucidating a relation between the QoE and transport-level performance, i.e., TCP throughput and end-to-end delay. Moreover, we measure performance of the two transfer mechanisms under various network environments that assumed general wireless network environments for elucidating a practicability of the transfer mechanisms that are employed by current remote desktop protocols. Through subjective evaluation, we show that image transfer mechanisms are sensitive to TCP throughput and event transfer mechanisms are sensitive to propagation delay. Consequently, we conclude that image transfer mechanisms are more suitable for remote desktop services under general wireless network environments than event transfer mechanisms because controlling TCP throughput is easier than controlling propagation delay. Furthermore, through measurement, we conclude that image transfer mechanisms are feasible in general wireless environments.

This thesis contributes to realizing high quality DC-DC/DC-CE cloud applications the performances of which TCP dominantly determines, by focusing on TCP throughput improvements without modifying TCP. It takes different approaches to the both applications depending on their performance requirements, i.e., application-level performance and subjective performance, and network characteristics.

# Acknowledgments

First and foremost, I am sincerely grateful to Professor Toru Hasegawa of Graduate School of Information Science and Technology, Osaka University for his immeasurable support and guidance. He advised me to how to do quality research, reviewed my papers, and led me into growing up. Without his support, this thesis would not have been completed. What I have learned from him would become very valuable in my life.

I would like to express my heartfelt gratitude to the members of my thesis committee, Professor Masayuki Murata, Professor Takashi Watanabe, Professor Teruo Higashino, and Professor Morito Matsuoka of Graduate School of Information Science and Technology, Osaka University for their in-depth discussion, valuable advice, and reviewing my thesis. Especially, I would like to express my deepest appreciation to Professor Masayuki Murata. He provided valuable advice and useful suggestions to me in regular meeting related to optical networking.

I am deeply grateful to Dr. Makoto Imase, Vice President of National Institute of Information and Communications Technology, and Professor Hiroyuki Ohsaki of Department of Informatics, School of Science and Technology, Kwansei Gakuin University for their continuing support and guidance. They introduced me to the area of network technology. Their advices have been helpful till now and would be in the future.

I am cordially thankful to Assistant Professor Yuki Koizumi of Graduate School of Information Science and Technology, Osaka University, for practical advice, useful suggestions, and enormous help. When I was worried about something, he always gave me a helping hand and encouragement. I could not conduct my research without his support.

I would like to express my profound thank to Assistant Professor Yuichi Oshita of raduate School of Information Science and Technology, Osaka University, for his tremendous support related to optical networking.

I wish to express to thank to members of joint researches, Mr. Nobuhiro Yokoi, Mr. Yoichi Nakamoto, Mr. Kazushi Nakagawa, Mr. Cao Lethanhman, Mr. Nobuaki Kohinata, and Mr. Yukihiro Takatani of Hitachi, Ltd., and Mr. Yoshihiro Nomoto of NTT Corporation, and Mr. Yuhei Hayashi of Tokyo Institute of Technology Graduate School of Engineering for their constructive discussion.

I am thankful to Mr. Fumito Inoue for his kindhearted guidance during my bachelor's course. I am also thankful to members of my research group, Youta Morishita, Yuto Nakai, Hiroaki Iwami and Yuki Urata for their important contribution. Researches with them were very significant for this thesis.

Special thanks to all members of the Information Sharing Platform Laboratory. I am thankful to Ms. Namiko Okada, Ms. Hiroko Hatagami and Ms. Yoshimi Fujita for their kind help. I thank Keiichiro Tsukamoto, Kohei Watabe and all the other laboratory members for their constant encouragement. I also thank to Daichi, Kazuyuki, Yuya and other my friends for their warm encouragement.

Finally, I would like to express my deep gratitude to my family. Without their continuous and substantial support, I could not have done anything.

# Contents

**List of Publications** **i**

**Preface** **iii**

**Acknowledgments** **vii**

**1 Introduction** **1**

1.1 Background . . . . . 1

1.2 Approaches for DC-DC Cloud Applications . . . . . 4

1.2.1 Approaches for Using Existing Dedicated Hardware Devices . . . . . 5

1.2.2 Approaches for Designing TCP Software running on Dedicated Hardware Devices . . . . . 6

1.3 Approaches for DC-CE Cloud Applications . . . . . 7

1.4 Outline of Thesis . . . . . 9

**2 Improving throughput of heterogeneous IP-SAN protocols with existing hardware devices** **11**

2.1 Introduction . . . . . 11

2.2 Related Work . . . . . 13

2.3 IP-SAN Protocols . . . . . 13

2.4 Block Device Layer with Automatic Parallelism Tuning (BDL-APT) . . . . . 14

2.4.1 Overview . . . . . 14

2.4.2 Block device layer . . . . . 16

2.4.3 Data transfer parallelization . . . . . 16

2.4.4 Optimization of the number of parallel TCP connections . . . . . 17
2.4.5 Goodput monitoring . . . . . 19
2.5 Implementation . . . . . 21
2.6 Experiment . . . . . 25
2.6.1 Experiment design . . . . . 25
2.6.2 Evolution of IP-SAN goodput . . . . . 27
2.6.3 Effect of network bandwidth . . . . . 27
2.6.4 Effect of network delay . . . . . 29
2.6.5 CPU Processing Load . . . . . 30
2.7 Summary . . . . . 32

**3 Estimation of Performance Improvement Derived from TCP/IP Offload Engine with Software Emulation 41**

3.1 Introduction . . . . . 41
3.2 Related Work . . . . . 43
3.3 TOE (TCP/IP Offload Engine) . . . . . 45
3.4 Proposal of VOSE (Virtual Offloading with Software Emulation) for Estimating Performance Improvement with TCP/IP Protocol Offloading . . . . . 47
3.5 Evaluation of VOSE . . . . . 50
3.5.1 Experimental environments . . . . . 50
3.5.2 Virtual offloading of TCP checksum calculations . . . . . 51
3.5.3 Results . . . . . 52
3.5.4 Observation . . . . . 56
3.6 Application of VOSE to IPsec Protocol . . . . . 60
3.6.1 Virtual offloading of IPsec protocol . . . . . 60
3.6.2 Experiment . . . . . 61
3.7 Summary . . . . . 63

**4 Performance Comparison between Image and Event based Transfer Mechanisms for Remote Desktop Protocols 65**

4.1 Introduction . . . . . 65
4.2 Related Work . . . . . 68

4.3 Remote Desktop Protocols . . . 69
4.3.1 Image transfer mechanism . . . 69
4.3.2 Event transfer mechanism . . . 69
4.3.3 Examples of remote desktop protocols . . . 70
4.4 Experiment Design . . . 72
4.4.1 Experiment Environment . . . 72
4.4.2 Measurement Workload . . . 74
4.5 Subjectivity Evaluation . . . 75
4.5.1 Evaluation Method . . . 75
4.5.2 Simple Experiment . . . 76
4.5.3 Effect of TCP throughput . . . 78
4.5.4 Effect of propagation delay . . . 80
4.5.5 Observations . . . 81
4.6 Quantitative Evaluation . . . 81
4.6.1 Measurement Method . . . 82
4.6.2 Effect of network bandwidth . . . 83
4.6.3 Effect of network delay . . . 84
4.6.4 Effect of network loss rate . . . 85
4.6.5 Observations . . . 85
4.7 Summary . . . 85

**5 Conclusion** **91**

**Bibliography** **95**

# List of Figures

2.1 Overview of BDL-APT. BDL-APT realizes the data delivery over multiple TCP connections by using multiple IP-SAN sessions, monitors the incoming/outgoing goodput, and optimizes the number of active IP-SAN sessions automatically. . . . . 15

2.2 Example of BDL-APT operation when searching for a bracket . . . . . 20

2.3 Example of BDL-APT operation when searching for the optimal number of TCP connections . . . . . 21

2.4 Evolution of the total incoming/outgoing data size transferred through active IP-SAN sessions. . . . . 22

2.5 Structure of the RAID-0 module in the MD driver. The RAID-0 module (a) creates the RAID device, (b) manages striped storage devices, (c) splits block I/O requests (bio structure objects), and (d) assigns split block I/O requests to multiple storage devices. . . . . 23

2.6 Structure of the BDL-APT module, which is based on the RAID-0 module. The BDL-APT module has several added functions to the RAID-0 module. Namely, the BDL-APT module (d) dynamically assigns divided bio structure objects to a subset of storage devices, (e) optimizes the number of active IP-SAN sessions, and (f) continuously measures the goodput of block I/O requests. . . . . 24

2.7 Network configuration used in experiments. IP-SAN client and storage device are connected via the network emulator to simulate a long-fat network. . . . . 26

2.8 Evolution of NBD goodput. the number of active NBD sessions is optimized at approximately 1,200 [s], and the NBD goodput converges to 828 [Mbit/s]. . . . . 28

2.9 Evolution of the number of active NBD sessions. the number of active NBD sessions converges to 33 at approximately 1,200 [s]. . . . . 29
2.10 Bottleneck link bandwidth vs. IP-SAN goodput . . . . . 34
2.11 Bottleneck link delay vs. IP-SAN goodput . . . . . 35
2.12 The average CPU load of the IP-SAN client . . . . . 36
2.13 The average CPU load of the IP-SAN storage . . . . . 37
2.14 The number of CPU ticks consumed for a successful bit transfer on the IP-SAN client 38
2.15 The number of CPU ticks consumed for a successful bit transfer on the IP-SAN storage . . . . . 39

3.1 Changes in TCP/IP processing under TOE: (a) the flow when performing all TCP processing in software, and (b) the processing flow when offloading checksum calculations to TOE . . . . . 46
3.2 VOSE realizes virtual offloading of protocol processing by utilizing the symmetry of protocol processing between the sender and the receiver while preserving its integrity. . . . . 47
3.3 Changes in TCP/IP processing by VOSE: (a) the processing flow in the case of offloading checksum calculations to TOE, and (b) the processing flow in the case of virtually offloading checksum calculations. . . . . 48
3.4 Realization of the busy loop using the RDTSC command: A loop is carried out for the specified number of cycles using the IA32/IA64 processor RDTSC instruction, which waits for `$clocks_to_sleep` clocks. . . . . 51
3.5 End-to-end performance (effective throughput) when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A) . . . . . 52
3.6 End-to-end performance (effective throughput) when changing the sender and receiver CPU operating frequency over 10 Gigabit Ethernet (experimental environment B) . . . . . 53
3.7 CPU utilization of the sender when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A) . . . . . 54

3.8 CPU utilization of the sender when changing the sender and receiver CPU operating frequency over 10 Gigabit Ethernet (experimental environment B) . . . . . 55
3.9 CPU utilization of the receiver when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A) . . . . . 56
3.10 CPU utilization of the receiver when changing the sender and receiver CPU operating frequency in over 10 Gigabit Ethernet (experimental environment B) . . . . . 57
3.11 The end-to-end performance (effective throughput) when changing the time $T_S$ and $T_R$ of sleep processing . . . . . 59
3.12 The end-to-end performance (effective throughput) of five types of TOE devices when changing the time of sleep processing . . . . . 62

4.1 Experiment environment . . . . . 72
4.2 An example of the Web pages used for the experiments . . . . . 74
4.3 TCP throughput vs. QoE . . . . . 79
4.4 Propagation delay vs. QoE . . . . . 80
4.5 The flow from operation of a user to renewal of a screen . . . . . 82
4.6 Link bandwidth vs. response time, transfer data size, and data transfer rate. . . . . 87
4.7 Link delay vs. response time. . . . . 88
4.8 Link loss rate vs. response time. . . . . 89

# List of Tables

2.1 Default parameters used in experiments . . . . . 33

3.1 The profiling result of the sender when carrying out a hardware offloading with TOE in 3.2 GHz . . . . . 58
3.2 The profiling result of the sender when carrying out a virtual offloading with VOSE in 3.2 GHz . . . . . 58

4.1 5-grade impairment scale . . . . . 76
4.2 The QoE of four types . . . . . 77

# Chapter 1

# Introduction

## 1.1 Background

With rapid advancements and developments of networking technologies, various cloud applications have emerged [1]. Cloud users can access to resources, such as computers, storage, or application software, that are located on distant data centers as if those resources are located at hand [2]. Many cloud applications that have been conventionally used on a local computer located at hand have come to be used on distant computers [3–9].

In this thesis, we focus on cloud applications the performances of which TCP dominantly determines, and we improve performance of the applications by improving TCP performance. Except for cloud applications that transfer very small data, e.g., a web mail and a text chat, TCP is dominantly used in cloud applications that transfer data. Namely, performance of the cloud applications largely depends on TCP performance. Therefore, in this thesis, we improve performance of such cloud applications.

TCP performance issues are categorized by the number of TCP connections that derive issues; one or many connections. For instance, with one connection, TCP performance degradation, e.g., low TCP throughput and high end-to-end delay, deteriorates application-level performance such as transfer speed and response time. With many connection, access concentrations to one link and changes of network status such as background traffic cause congestion problem, i.e., one of flash crowd problems, and that deteriorates application-level performance. Because issues with one

connection are always critical even if many connections are established, in this thesis, we tackle issues that are derived from performance degradation of one connection.

Because of installability into current environments, we improve TCP performance with existing TCP, i.e., without modifying TCP. Two methods for improving TCP performance are categorized depending on whether TCP is modified. One is modification of TCP for improving its performance. An advantage of a method to modify TCP itself is to be able to solve all issues derived from TCP, e.g., TCP throughput degradation and occurrence of congestion, but its disadvantage is that it is difficult to install. The other is to improve performance without modifying TCP. An advantage of improving performance without modifying TCP is that it is easy to install into current environments, but its disadvantage is that only some issues are solved. Namely, trade-off between amount of TCP issues solved and installability exists.

Cloud applications are categorized depending on how devices such as servers and clients are located. One is a *DC-DC (Data Center-Data Center) cloud application* wherein servers are located on geographically distant data centers. DC-DC cloud applications mainly transfer large bulk data from a data center to another data center without making users intervene. A remote backup application and grid computing application, e.g., large-scale scientific calculation, are such examples wherein a client computer located on a data center transfers large data to a remote server with a storage device located on another data center [10–19]. Thus application-level performances, e.g., data transfer throughput, CPU load and reliability, is key metrics of DC-DC cloud applications. The other is a *DC-CE (Data Center-Customer Equipments) cloud application* wherein a server is located a data center and a client, i.e., a customer equipment, is located at an edge of a wire area communication network. DC-CE cloud applications mainly transfer small data between a data center and customer equipments interactively. A difference to a DC-DC application is that a user intervenes and that subjective performances, i.e., how users feel when using the applications are important. A remote desktop, a cloud gaming and a web mail are such examples wherein a client computer, i.e., customer equipment, transfers operation information, e.g., keyboard and mouse control information from a client, to a server computer located on a data center, and the server sends back response data, e.g., updated screen information, to the client [5, 9, 20–23]. Thus subjective performances are a key metric of DC-CE cloud applications.

In summary, DC-DC and DC-CE applications have their own requirements to service quality: application-level performance and subjective performance. The reason of the difference is who

uses an application. Since a user of a DC-DC application is a machine, only the applicaiton-level performance is focused on. Whereas, since a user of a DC-CE application is a human being, the subjective performance should be focused on as well as the applicaiton-level performance.

In order to provide the both applications with good application-level performance and subjective performance, respectively, we should be careful that network characteristics of the both applications differ. DC-DC cloud applications are used in high bandwidth and high delay networks such as dedicated lines, whereas DC-CE cloud applications such as remote desktops are used in low bandwidth and high delay networks such as the Internet. In general, DC-DC cloud providers establish a high bandwidth network such as a dedicated line between data centers because DC-DC cloud applications require high bandwidth for transferring large bulk data. Because each data center is located on geographically different location, the network delay between data centers is large. Therefore, mainly, a bandwidth of networks where DC-DC cloud applications are used is large, and its delay is large. On the other hand, DC-CE cloud users use the applications from various locations via various communication networks, e.g., users access application server from distant location by mobile terminal such as cellular phone via wireless networks. Therefore, mainly, a bandwidth of networks where DC-CE cloud applications are used is small, and its delay is large.

A key issue to provide good performance is how high TCP throughput is provided to the both applications because most cloud application use TCP for data transfer and because delays are not controlled. However, requirements to providing high TCP throughput are not trivial and they depend on the both applications depending on their differences of required performance and network characteristics. Because DC-DC applications that transfer large bulk data require high transfer speed, the requirement to TCP throughput of DC-DC application is obvious. The transfer speed is just TCP throughput. On the other hand, it depends on DC-CE cloud applications which subjective performance is important, and it depends on each subjective performance whether good TCP performance is necessary. A response time is an example of subjective performance that requires good throughput and end-to-end delay. Meanwhile, a clarity of an image and an audio noise are examples that do not require high TCP performance.

This thesis contributes to realizing high quality DC-DC/DC-CE cloud applications the performances of which TCP dominantly determines, by focusing on TCP throughput improvements without modifying TCP. It takes different approaches to the both applications depending on their

performance requirements, i.e., application-level performance and subjective performance, and network characteristics.

## 1.2 Approaches for DC-DC Cloud Applications

Performance degradation of the TCP protocol in a high bandwidth and high delay network is a well-known problem, and there have been a huge number of researches for improving the TCP performance. In particular, a couple of issues have been reported regarding TCP such as its inability to support the rapidly increasing speeds of recent networks. First, TCP throughput deteriorates in high delay networks because of its algorithm. As an example, the current TCP Reno (TCP version Reno) deteriorates its window size that determines transfer speed when packet loss occurs, and recovers the window size at every round-trip-time, i.e., double end-to-end delay. Therefore, the recovery speed of the transfer speed becomes slow when network delay is large. Network delay therefore should be concealed from upper layer software, i.e., applications. For concealments of network delay, a huge number of performance improvements such as changes of congestion control algorithm [24, 25], parameter tunings [26, 27] and parallel data transfers [28–31] have been proposed. Second, heavy protocol processing derive degradations of TCP throughput and end-to-end delays in high bandwidth environments because CPU becomes bottleneck, and therefore accelerations of heavy processing are required for providing good performance. For removing CPU bottleneck, TCP/IP Offload Engine (TOE), which accelerates protocol processing with hardware offloading, has been proposed and developed [32–41].

Because some DC-DC cloud applications are implemented as dedicated hardware devices for providing good performance, the DC-DC could applications still have a couple of open issues. One is a performance improvement with existing dedicated hardware devices wherein changes of TCP protocols and devices are difficult, and the other is a design of hardware devices for providing good performance.

Many current DC-DC cloud applications such as a remote backup, a data sharing and a large-scale scientific calculation transfer various large bulk data between data centers through an IP-based Storage Area Network (IP-SAN). An IP-SAN, which connects storage devices located on distant data centers on an IP network, is widely used for data transfers of DC-DC cloud applications because of low cost and high compatibility with existing network infrastructures [10–13]. IP-SAN

has an open issue that its performance degradation is derived from TCP performance degradation in high delay networks, and other DC-DC cloud applications also have the same issue.

First, we focus on a performance improvement with existing dedicated hardware devices. IP-SAN is utilized for a DC-DC cloud application wherein many dedicated hardware devices that perform protocol processing with hardware are deployed. Although IP-SAN still have open performance issues [42–45], the solutions that requires changes of TCP protocols or IP-SAN protocols can not be employed for current IP-SANs. Therefore, it is desirable to improve IP-SAN performance without need for device modifications. In addition, because many heterogeneous IP-SAN protocols and their devices have already been proposed and deployed in current IP-SANs, a solution for not individual IP-SAN protocol but whole IP-SAN protocols is required.

Second, we focus on a design of hardware devices for providing good performance. TOE has been studied by many researchers to solve the CPU bottleneck, but it is not clear which protocol processing should be performed with hardware for future DC-DC cloud applications. Moreover, offloading of multiple protocol processings simultaneously may be required for providing required performance in DC-DC cloud applications. In current TOE design, for designing TOE that provides required performance, it is necessary to develop many different types of TOE prototypes and to evaluate the prototypes actually. Therefore, a method for easily estimating TCP/IP performance improvements derived from different type of TOE devices is required; i.e., an estimation method without implementing TOE devices really.

### 1.2.1 Approaches for Using Existing Dedicated Hardware Devices

A number of solutions for preventing IP-SAN performance degradation in high bandwidth and high delay network have been proposed [28, 46–48]. Yang [47] improves iSCSI throughput by using multiple connections, each of which traverses a different path using multiple LAN ports and dedicated routers. However, LAN ports and dedicated routers are not always available, forcing significant restrictions on the network environment. Inoue *et al.* [46] improve iSCSI throughput by adjusting the number of parallel TCP connections using the iSCSI protocol's parallel data transfer feature. However, this solution cannot be used in IP-SAN protocols without that feature. Changes to the TCP congestion control algorithm for improving fairness and throughput of the iSCSI protocol have also been proposed [28]. Oguchi *et al.* analyze iSCSI with the proposed monitoring tool, and improve the throughput of iSCSI by adjustment of TCP socket buffer, or the correction inside a

Linux kernel [48]. However, solutions that replace or modify the transport protocol are unrealistic because they require changes in all storage devices and clients.

It is desirable to improve IP-SAN performance without need for device modifications and TCP modifications. In addition, because many heterogeneous IP-SAN protocols and their devices have already been proposed and deployed in current IP-SANs, a solution for not individual IP-SAN protocol but whole IP-SAN protocols is required.

Parallel data transfer, which uses multiple TCP connections for data delivery, is one of promising approaches for concealing network delay and achieving high TCP performance of IP-SANs. However, parallel data transfer still requires modifications of existing hardware devices, IP-SAN protocols or applications. To support many devices and applications and heterogeneous IP-SAN protocols used in current IP-SANs, means of applying parallel data transfer to IP-SANs regardless of existing hardware devices, IP-SAN protocols and applications is crucial.

To realize parallel data transfer regardless of existing hardware devices, IP-SAN protocols and applications, we propose a new layer called block device layer between IP-SAN protocols and applications. Block device layer receives read/write requests from an application or a file system and relays those requests to a storage device. Our key idea is parallelizing data transfer at a block device layer by dividing aggregated read/write requests into multiple chunks, then transferring a chunk of requests on every IP-SAN connection in parallel. We realize a performance improvement without modifications of existing hardware devices, TCP protocols and applications since our solution does not modify upper or lower layer of the block device layer.

### 1.2.2 Approaches for Designing TCP Software running on Dedicated Hardware Devices

It is not clear which protocol processing or which those combinations should be offloaded for providing TCP throughput that is enough to achieve required performance in DC-DC cloud applications.

To solve the CPU bottleneck, TOE has been studied by many researchers as a means of accelerating protocol processing [32–41], and many TOE devices have been developed and installed, particularly in high-end systems [49, 50]. For instance, the researches show the effectiveness of actual hardware devices for TCP checksum calculation [35], IPsec [32], ACK management [37] and all processing [40]. However, it has not been clarified which part of TCP/IP processing should

be performed with hardware and which with software, or how performance is affected by the introduction of a TOE device.

TOE design must take into account improvements in end-to-end performance, e.g., throughput and latency, resulting from the introduction of TOE [51, 52]. However, measurement of the performance improvements generally requires actual experiments using TOE implementations. Namely, to measure performance improvements derived from introducing a TOE device, it is necessary to implement a TOE prototype really, and it means high cost and long development periods. Moreover, if TOE designers measure performance of the various patterns of offloading and clarify processing that should be performed with software and the processing that should be performed with hardware, the designers can develop optimal TOE.

Therefore, in this thesis, we propose a support method that is a technique for estimating TCP/IP performance improvements derived from different type of TOE devices with software emulation. By virtually emulating TOE processing on both source and destination end hosts, our approach enables measuring performance improvements without requiring a real hardware TOE device. Our approach realizes virtual protocol offloading, e.g., bypassing software protocol processing without violating protocol consistency, by utilizing symmetry of protocol processing between the sender and the receiver while preserving its integrity.

## 1.3 Approaches for DC-CE Cloud Applications

Because human being uses DC-CE cloud applications, not only tranfer speed but also a response time is an important metric. In this thesis, because a response time is determined by combinations of TCP throughput anddelays, i.e., TCP performance predominates over a response time, we mainly focus on the response time as a subjective performance of DC-CE cloud applications. There are a number of evaluation focusing on impacts of network characteristics on response times, however, impacts of transport-level performance on subjective performance have not been clear sufficiently. In order to provide good response time for DC-CE cloud applications, it is necessary to elucidate a relation between transport-level performance and subjective performance. In particular, we elucidate which transport-level performance deteriorates Quality of Experience (QoE) largely.

Many DC-CE cloud applications that TCP is dominant exist, in this thesis, we focus on a remote desktop application at first step for elucidating a relation between TCP performance and a response

time of subjective performance. A remote desktop is a cloud application that users can easily experience a response time because screen is updated once per one operation, i.e., mouse click or keyboard input. We consider that results of remote desktop are useful for similar applications that interactively send operation information and response data between a client and a server.

Several remote desktops such as an X window or thin clients have been evaluated [6,53–56], but impacts of transport-level performance on subjective performance are not clear. There are a number of evaluations focusing on impacts of network characteristics on application-level performance. Rhee *et al.* [53] show that the response time of Microsoft Terminal Service becomes long in a high delay network. Berryman *et al.* [54] show that transfer data size of remote desktop protocols that employ TCP becomes large in a high delay or high loss rate network and show transfer speed of remote desktop protocols that employ UDP becomes slow in a high delay or high loss rate network. Yang *et al.* show that a response time of thin clients that cannot cancel updated screen information becomes long in a low bandwidth network [6], and show that thin clients that employ high-level encoding can provide good response time even if network bandwidth is small [55]. On the other hand, there are a few evaluations focusing on impacts of network characteristics on subjective performance. Niraj *et al.* [56] evaluate subjective performance of one protocol by a simulation, and shows that the subjective performance is impacted by network delay. However, impacts of transport-level performance are not clear sufficiently, and tendencies of subjective performance at various transport-level performance is also not clear sufficiently.

Therefore, in this thesis, we elucidate the impacts through experiments. In this thesis, we focus on TCP throughput and delays as transport-level performance. We conjecture that difference of impacts is derived from transfer mechanism of DC-CE applications, and therefore we categorize the applications by their transfer mechanisms. We evaluate a couple of impacts of transport-level performance on subjective performance of respective transfer mechanisms through experiment; 1) an impact of TCP throughput on QoE and 2) an impact of end-to-end delay on QoE. Since many DC-CE cloud applications are used through wireless networks, we mainly evaluate the impacts in a network environment that assumed general wireless network environments, i.e., low bandwidth such as around 1–5 Mbit/s and high delay such as 10–100 ms.

## 1.4 Outline of Thesis

The structure of this thesis is as follows.

We focus on respective TCP performance issues in each chapter; In Chapter 2 and Chapter 3, we tackle two issues for DC-DC cloud applications to realize high quality cloud applications. Chapter 2 solves an issue for using existing dedicated hardware devices explained in Section 1.2.1, and Chapter 3 solves an issue for designing TCP software running on dedicated hardware devices explained in Section 1.2.2. In Chapter 4, we tackle an issue explained in Section 1.3 for clarifying a relation between TCP performance and subjective performance of remote desktops.

First, in Chapter 2, we propose *Block Device Layer with Automatic Parallelism Tuning (BDL-APT)*, a mechanism that maximizes the transfer speed of heterogeneous IP-SAN protocols in high delay networks. For improving IP-SAN performance, BDL-APT parallelizes data transfer using *multiple IP-SAN sessions* at *a block device layer* on an IP-SAN client and automatically optimizes the number of active IP-SAN sessions according to network status. We propose new layer called block device layer, which receives read/write requests from an application or a file system and relays those requests to a storage device. BDL-APT parallelizes data transfer by dividing aggregated read/write requests into multiple chunks, then transferring a chunk of requests on every IP-SAN session in parallel. BDL-APT automatically optimizes the number of active IP-SAN sessions based on the monitored network status using our parallelism tuning mechanism, because it is known that the number of the connections that is not optimal deteriorates TCP throughput [46,57]. We evaluate the performance of BDL-APT through experiment using several IP-SAN protocols. Through our experiment, we show that BDL-APT realizes high performance of heterogeneous IP-SAN protocols in various network environments.

Moreover, in Chapter 3, we propose *Virtual Offloading with Software Emulation (VOSE)*, which is a technique for measuring TCP/IP performance improvements derived from different type of TOE devices without implementing TOE prototypes really. VOSE enables *virtual offloading* without requiring a hardware TOE device by virtually emulating TOE processing on both source and destination end hosts. For demonstrating the effectiveness of VOSE, we apply VOSE to the TCP checksum and IPsec protocol. We extensively examine the accuracy of virtual offloading with VOSE, by comparing performance, i.e., end-to-end performance and CPU processing overhead, between VOSE and a dedicated TOE device. Moreover, we estimate performance improvements that are derived

from several TOE devices of IPsec and combinations of those devices, by applying VOSE to header authenticating and payload encryption in IPsec protocol. Consequently, we show that performance improvements which are derived from TOE devices can be estimated correctly.

Next, in Chapter 4, we focus on impact of TCP performance on subjective performance of remote desktops. For elucidating essential relation, we categorize whole remote desktop protocols by their mechanisms for transferring updated screen information. One is an *image transfer mechanism*, which transfers the updated screen information as images, and the other is an *event transfer mechanism*, which transfers the information as update events, e.g., a window creation, setting of window positions and renderings of pictures or texts.

We elucidate impacts of transport-level performance on performance of respective transfer mechanisms in a network environment that assumed general wireless network environments. We evaluate QoE of respective transfer mechanisms for elucidating a relation between the QoE and transport-level performance, i.e., TCP throughput and end-to-end delay. Moreover, we measure performance of the two transfer mechanisms under various wireless network environments for elucidating a practicability of the transfer mechanisms that are employed by current remote desktop protocols. Through subjective evaluation, we show that image transfer mechanisms are sensitive to TCP throughput and event transfer mechanisms are sensitive to propagation delay. Consequently, we conclude that image transfer mechanisms are more suitable for remote desktop services than event transfer mechanisms because controlling TCP throughput is easier than controlling propagation delay. Furthermore, through measurement, we conclude that image transfer mechanisms are feasible in general wireless environments.

Finally, Chapter 5 concludes this thesis and discusses future works.

# Chapter 2

# Improving throughput of heterogeneous IP-SAN protocols with existing hardware devices

## 2.1 Introduction

In recent years, IP-based Storage Area Networks (IP-SANs) have attracted attention for building SANs on IP networks, owing to the low cost and high compatibility of IP-SANs with existing network infrastructures [10–13].

Several IP-SAN protocols such as NBD (Network Block Device) [14], GNBD (Global Network Block Device) [15], iSCSI (Internet Small Computer System Interface) [16], FCIP (Fibre Channel over TCP/IP) [17], and iFCP (Internet Fibre Channel Protocol) [18] have been widely utilized and deployed for building SANs on IP networks. IP-SAN protocols allow interconnection between clients and remote storage via a TCP/IP network.

IP-SAN protocols realize connectivity to remote storage devices over conventional TCP/IP networks, but still have unresolved issues, performance in particular [12, 42, 44]. Several factors affect the performance of IP-SAN protocols in a long-fat network. One significant factor is TCP performance degradation in long-fat networks [28]; IP-SAN protocols generally utilize TCP for data delivery, which performs poorly in long-fat networks.

In this chapter, we propose *Block Device Layer with Automatic Parallelism Tuning (BDL-APT )*,

a mechanism that maximizes the goodput of heterogeneous IP-SAN protocols in long-fat networks and that requires no modification to IP-SAN storage devices. BDL-APT parallelizes data transfer using *multiple IP-SAN sessions* at *a block device layer* on an IP-SAN client, automatically optimizing the number of active IP-SAN sessions according to network status. A block device layer is a layer that receives read/write requests from an application or a file system, and relays those requests to a storage device. BDL-APT parallelizes data transfer by dividing aggregated read/write requests into multiple chunks, then transferring a chunk of requests on every IP-SAN session in parallel. BDL-APT automatically optimizes the number of active IP-SAN sessions based on the monitored network status by means of our APT mechanism [46, 57].

We evaluate the performance of BDL-APT with heterogeneous IP-SAN protocols, i.e., NBD, GNBD and iSCSI, in a long-fat network. We implement BDL-APT as a layer of the Multiple Device (MD) driver [58], one of the major software RAID implementations included in the Linux kernel. Through experiments, we demonstrate the effectiveness of BDL-APT with heterogeneous IP-SAN protocols in long-fat networks regardless of protocol specifics.

Our contributions are two-fold: 1) BDL-APT enable to parallelize data transfer and improve IP-SAN throughput without modifications of existing IP-SAN storage devices, TCP protocols, IP-SAN protocols and applications. 2) BDL-APT enables to improve throughput of heterogeneous IP-SAN protocols regardless of network bandwidth and delay. BDL-APT enables IP-SAN users to transfer large bulk data between a client and a geographically distant storage without performance degradation. Consequently, BDL-APT enables IP-SAN users to build an IP-SAN between more distant data centers.

This chapter is organized as follows. Section 2.2 summarizes related works. Section 2.3 introduces the IP-SAN protocols used in our BDL-APT experiments. Section 2.4 describes the overview and the main features of our BDL-APT. Section 2.5 explains our BDL-APT implementation. Section 2.6 gives a performance evaluation of our BDL-APT implementation in heterogeneous IP-SAN protocols. Finally, Section 2.7 summarizes this chapeter and discusses areas for future work.

## 2.2 Related Work

Several solutions for preventing IP-SAN performance degradation in long-fat networks have been proposed [28, 46–48]. For instance, solutions utilizing multiple links [47] or parallel TCP connections [46] have been proposed to prevent throughput degradation of the iSCSI protocol.

Yang [47] improves iSCSI throughput by using multiple connections, each of which traverses a different path using multiple LAN ports and dedicated routers. However, LAN ports and dedicated routers are not always available, forcing significant restrictions on the network environment. Inoue *et al.* [46] improve iSCSI throughput by adjusting the number of parallel TCP connections using the iSCSI protocol's parallel data transfer feature. However, this solution cannot be used in IP-SAN protocols without that feature. Changes to the TCP congestion control algorithm for improving fairness and throughput of the iSCSI protocol have also been proposed [28]. Oguchi *et al.* analyze iSCSI with the proposed monitoring tool, and improve the throughput of iSCSI by adjustment of TCP socket buffer, or the correction inside a Linux kernel [48]. However, solutions that replace or modify the transport protocol are unrealistic because they require changes in all IP-SAN targets, i.e., storage devices, and initiators, i.e., clients. In particular, because software of current many IP-SAN targets has been implemented on hardware devices, it is difficult to modify the transport protocol of IP-SAN targets.

While these and other solutions for specific IP-SAN protocols have been proposed, many heterogeneous IP-SAN devices have already been deployed, so it is desirable to improve IP-SAN performance independent of protocol and without need for device modifications.

In this chapter, we propose an initiator-side solution, which requires no modification to IP-SAN storage devices, to the performance degradation of heterogeneous IP-SAN protocols in long-fat networks.

## 2.3 IP-SAN Protocols

In this section, we briefly introduces three IP-SAN protocols used in our BDL-APT experiments.

- NBD (Network Block Device)

  The NBD protocol, initially developed by Pavel Machek in 1997 [14], is a lightweight IP-SAN protocol for accessing remote block devices over a TCP/IP network. The NBD protocol

allows transparent access of remote block devices via a TCP/IP network, and supports primitive block-level operations such as read/write/disconnect and several other I/O controls. All communication between NBD clients and servers occurs using the TCP protocol.

- GNBD (Global Network Block Device)

  The GNBD protocol is another lightweight IP-SAN protocol for accessing remote block devices over a TCP/IP network. The GNBD protocol was developed at the University of Minnesota as part of the GFS (Global File System) [15, 59]. As in the NBD protocol, the GNBD protocol allows transparent access to remote block devices via a TCP/IP network. All communication between a GNBD client and server are transferred via TCP. A notable difference between GNBD and NBD is that GNBD allows simultaneous connections between multiple clients and a single server, i.e., a block device.

- iSCSI (Internet Small Computer System Interface)

  The Internet Engineering Task Force standardized the iSCSI protocol in 2004 [16]. iSCSI protocol encapsulates a stream of SCSI command descriptor blocks (CDBs) in IP packets, allowing communication between a SCSI initiator, i.e., a client, and its target, i.e., a storage device, via a TCP/IP network. When a SCSI initiator receives read/write requests from an application or a file system, it generates SCSI CDBs and transfers those CDBs to the SCSI storage through a TCP connection. It is known that iSCSI performance is significantly degraded when the end-to-end delay, which is the delay between the iSCSI initiator and its target, is large [43–46].

## 2.4 Block Device Layer with Automatic Parallelism Tuning (BDL-APT)

### 2.4.1 Overview

We propose BDL-APT, a mechanism that maximizes the goodput of heterogeneous IP-SAN protocols in long-fat networks. BDL-APT realizes the data delivery over multiple TCP connections by using multiple IP-SAN sessions, and optimizes the number of parallel TCP connections automatically based on the IP-SAN goodput. BDL-APT is a mechanism that operates as a block device layer

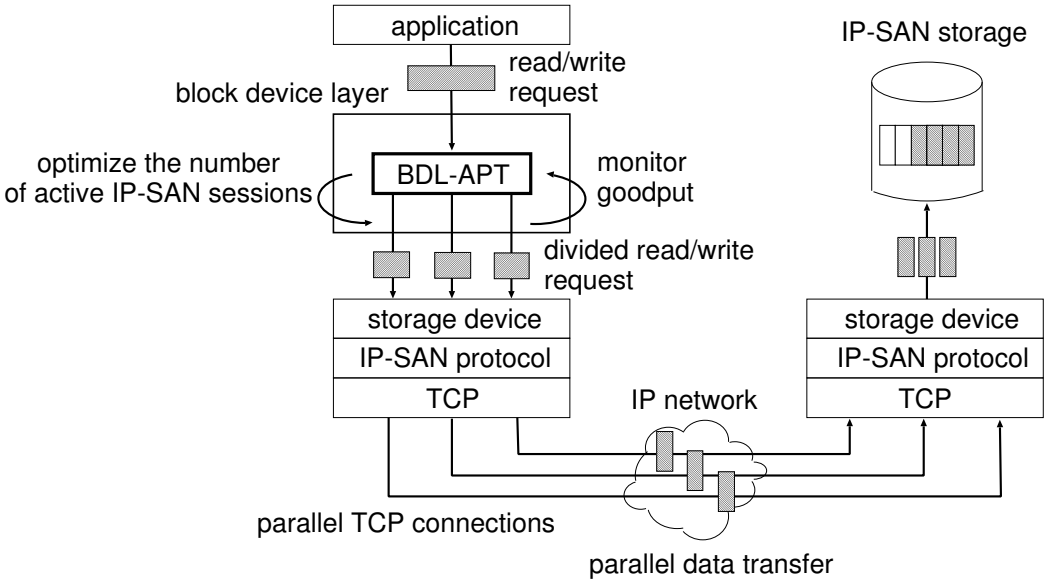



Figure 2.1: Overview of BDL-APT. BDL-APT realizes the data delivery over multiple TCP connections by using multiple IP-SAN sessions, monitors the incoming/outgoing goodput, and optimizes the number of active IP-SAN sessions automatically.

in an IP-SAN initiator, i.e., the client, (see Figure 2.1). BDL-APT parallelizes data transfer by dividing aggregated read/write requests into multiple chunks, then transferring a chunk of requests on every IP-SAN session in parallel. BDL-APT automatically optimizes the number of active IP-SAN sessions based on the monitored network status using our parallelism tuning mechanism APT [46], which is based on a numerical computation algorithm called the Golden Section Search method.

The main advantage of BDL-APT is its independence from the underlying IP-SAN protocol. Namely, BDL-APT can operate with any IP-SAN protocol since it works as a block device layer without reliance on features specific to the underlying block device or protocol. BDL-APT realizes both parallel data transfer and network status monitoring independently from the underlying block device or protocol.

Another advantage of BDL-APT is its initiator-side implementation. Namely, BDL-APT works within an IP-SAN initiator, so modification to IP-SAN targets are unnecessary. Thus, BDL-APT can be easily deployed in various IP-SAN environments.

BDL-APT is primarily designed for bulk data transfer applications such as a remote backup, a data sharing and a large-scale scientific calculation. It is because the problem of throughput degradation in long-fat networks is problematic when a large amount of data is transfered continuously.

Because BDL-APT runs at a block device layer, using techniques of other layers is possible without their modifications. Therefore, to improve further performance, BDL-APT is possible to use along with other techniques such as storage techniques and backup techniques.

In what follows, we explain the design and the implementation of our BDL-APT.

### 2.4.2 Block device layer

A block device layer is a layer that receives read/write requests from an application or a file system, and relays those requests to a storage device. In IP-SAN, a storage device driver handles data delivery to and from an IP-SAN storage device using an IP-SAN protocol (see Figure 2.1).

Block device layers are not new; they are adopted, for instance, in the MD driver, a software RAID implementation in Linux, and Violin [60], a framework for extensible block-level storage. The block device layer can perform various types of processing, such as mirroring, striping, and encryption. Such block device layers can be stacked to build new types of block device, for example to improve reliability, access speed, and security.

The following describes the main features of BDL-APT, *data transfer parallelization*, *optimization of the number of parallel TCP connections*, and *goodput monitoring*.

### 2.4.3 Data transfer parallelization

BDL-APT realizes parallel data transfer by establishing multiple IP-SAN *sessions* to a single storage device. Note that BDL-APT intentionally establishes multiple IP-SAN sessions, instead of multiple *connections* within a single IP-SAN session. Generally, one or more TCP connections carry a single IP-SAN session, meaning that using multiple IP-SAN sessions is equivalent to using multiple TCP connections. BDL-APT splits read/write requests, then transfers split requests in parallel over multiple IP-SAN sessions, making it possible to perform parallel data transfer with any IP-SAN protocol, which does not support parallel data transfer.

BDL-APT maintains multiple IP-SAN sessions to a single IP-SAN storage device. When BDL-APT receives read/write requests (hereafter called *block I/O requests*) from an application or a file system, BDL-APT splits those block I/O requests into multiple chunks and generates multiple block I/O requests for each chunk. BDL-APT parallelizes data transfer by assigning those generated block I/O requests to multiple IP-SAN sessions.

Note that I/O requests are generated by applications on an IP-SAN client regardless of read or write requests, and an IP-SAN storage never generates I/O requests. Therefore, dividing/aggregating a request on an only IP-SAN client can realize paralle data transfer without modifications of IP-SAN storages.

BDL-APT enables to generate multiple independent block I/O requests by dividing an original request into multiple requests and modifying addresses of the requests. Therefore, an original request is realized by independently performing each divided request on IP-SAN storage. The following describes two cases of read and write processing with our BDL-APT, respectively. First, when an application on a client reads data, BDL-APT divides an original block I/O request to multiple requests and assigns those requests to multiple IP-SAN sessions. The each request is performed on IP-SAN storage, and the storage sends back a block I/O response, i.e., a read data, per the requests to the client. BDL-APT aggregates those response into a data, i.e., a data that is read by an original request, and sends the data to the application. Second, when an application on a client writes data, BDL-APT divides an original block I/O request and assigns those requests to multiple IP-SAN sessions. The each data sending by the each request is sent to on IP-SAN storage, and the storage writes those data to physical address independently. Therefore, if all divided request data are written by the storage, a data that is written by an original request is written on the storage. Consistency of each divided I/O request is assured by existing underlying storage device driver. Therefore, BDL-APT leads to consistency of original I/O request.

Consequently, BDL-APT enables to parallelize data transfer without modifications of IP-SAN storages regardless of read/write data transfer, by dividing/aggregating a request on an IP-SAN client.

### 2.4.4 Optimization of the number of parallel TCP connections

BDL-APT optimizes the number of parallel TCP connections by optimizing the number of active IP-SAN sessions. An IP-SAN protocol utilizing TCP for data delivery establishes at least one TCP connection per IP-SAN session.

BDL-APT maintains multiple IP-SAN sessions. BDL-APT determines the required number of parallel TCP connections, and dynamically changes the number of *active* IP-SAN sessions. By assigning generated block I/O requests to a subset of established IP-SAN sessions, BDL-APT optimizes the number of active IP-SAN sessions used for parallel data transfer. BDL-APT determines

the required number of IP-SAN sessions using our Automatic Parallelism Tuning mechanism, APT.

We explain the overview of APT mechanism. Refer to [46] for the details of APT mechanism. The basic idea of APT is that a client splits data to transfer into blocks called *chunk*, and adjusts the number of parallel TCP connections at the end of every chunk transfer.

In what follows, $N$ is the number of parallel TCP connections used for a chunk transfer, and $G(N)$ the IP-SAN goodput measured at the chunk transfer.

- *Searching the range of the number of parallel TCP connections covering the optimal value that maximizes the IP-SAN goodput*

  First, BDL-APT searches the bracket. BDL-APT starts from a small number of parallel TCP connections, and multiplicatively increases the number of parallel TCP connections at every chunk transfer until IP-SAN goodput decreases. BDL-APT determines the bracket — the range of the number of parallel TCP connections covering the optimal value that maximizes the IP-SAN goodput.

  We illustrate an example operation of BDL-APT. Figure 2.2 shows an example operation of BDL-APT when searching for a bracket. BDL-APT searches for a bracket. The number $k$ shown in a circle indicates the $k$-th chunk transfer. First, BDL-APT initializes the number $N$ of parallel TCP connections to $N_0$ (= 1). BDL-APT multiplicatively increases the number of parallel TCP connections as $1 \rightarrow 2 \rightarrow 4 \rightarrow 8$ at every chunk transfer until the IP-SAN goodput starts to decrease. Since the IP-SAN goodput decreases when the number $N$ of parallel TCP connections changes as $4 \rightarrow 8$, the bracket is determined as (2, 4, 8).

- *Using the GSS algorithm for maximizing the IP-SAN goodput within the bracket*

  Next, using the GSS algorithm which is one of numerical computation algorithm for a maximization problem, BDL-APT searches the number of parallel TCP connections that maximizes the IP-SAN goodput within the bracket $(l, m, r)$ during succeeding chunk transfers.

  BDL-APT searches the optimal number $N$ of parallel TCP connections as follows.

  1. Update the number $N$ of parallel TCP connections:

$$N \leftarrow \begin{cases} \text{int}(l + (m-l)\nu) & \text{if } m-l > r-m \\ \text{int}(m + (r-m)\nu) & \text{otherwise} \end{cases} \tag{2.1}$$

where $\nu$ is the golden ratio ($= (3-\sqrt{5})/2$) and int($x$) is the nearest integer of $x$.

2. Transfer a chunk while measuring the IP-SAN goodput $G(N)$.
3. If the following inequality is satisfied, proceed to the step 4.

$$G(N) > G(m) \tag{2.2}$$

If the above inequality is not satisfied, change the bracket as follows and return to the step 1.

$$(l,m,r) \leftarrow \begin{cases} (l,m,N) & \text{if } m < N \\ (N,m,r) & \text{otherwise} \end{cases} \tag{2.3}$$

4. Change the bracket as follows, and return to the step 1.

$$(l,m,r) \leftarrow \begin{cases} (m,N,r) & \text{if } m < N \\ (l,N,m) & \text{otherwise} \end{cases} \tag{2.4}$$

Figure 2.3 shows an example operation of BDL-APT when searching for the optimal number of parallel TCP connections. Since the bracket is (2, 4, 8), the number of parallel TCP connections at the 5-th chunk transfer $N$ is determined as $N = 6$ from Eq. (2.1). The IP-SAN goodput in the 5-th chunk transfer is $G(6)$, and since $G(4) < G(6)$ is satisfied, the bracket is updated as (4, 6, 8) from Eq. (4). Hereafter, in a similar way, BDL-APT changes the number of parallel TCP connections $N$ as $6 \rightarrow 7 \rightarrow 5$, and updates the bracket as (4, 6, 8 ) $\rightarrow$ (4, 6, 7) $\rightarrow$ (5, 6, 7). Finally, when the bracket is (5, 6, 7), the number of parallel TCP connections is fixed at $N = 6$, which maximizes the IP-SAN goodput.

### 2.4.5 Goodput monitoring

During parallel data transfer, BDL-APT monitors the goodput at every goodput measurement interval $\Delta$ in the block device layer. $\Delta$ is one of APT parameters [46].

We measure the incoming/outgoing goodput as follows, respectively. BDL-APT calculates goodput by dividing total of the incoming/outgoing data size through all active IP-SAN sessions

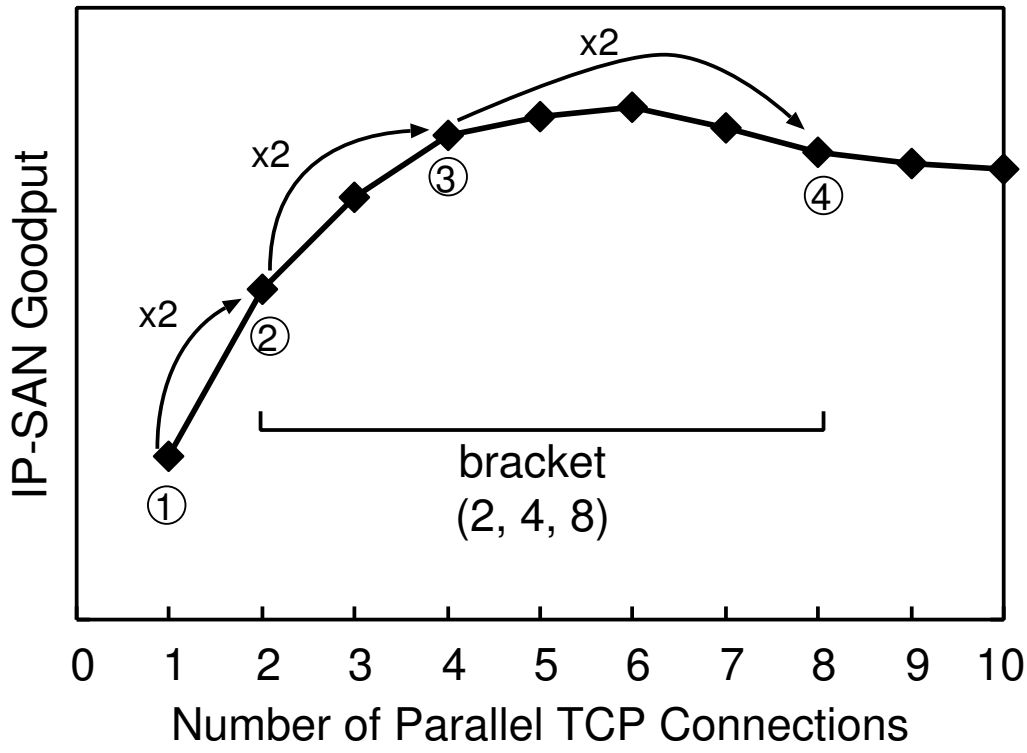



Figure 2.2: Example of BDL-APT operation when searching for a bracket

among $\Delta$ by $\Delta$ that is a measurement interval. Data are incoming from a storage device or outgoing to a storage device through multiple IP-SAN sessions during parallel data transfer. BDL-APT gathers those incoming/outgoing data and calculates goodput for them as a "chunk" of APT.

Figure 2.4 shows evolution of total incoming/outgoing data size transferred through active IP-SAN sessions. We calculate the goodput G between $T_{r1}$ and $T_{r2}$. $T_{r2}$ has passed $\Delta$ times since $T_{r1}$. $X_{r1}$ and $X_{r2}$ are the total data size transferred at $T_{r1}$ and $T_{r2}$, respectively. $X$ is the total data size transferred between $T_{r1}$ and $T_{r2}$, and is the difference of $X_{r1}$ and $X_{r2}$. Goodput G is calculated by $G = \frac{X}{\Delta}$.

When BDL-APT assigns generated block I/O requests to multiple IP-SAN sessions, BDL-APT records the size of each data transfer request. Then, BDL-APT calculates the total data size transferred.

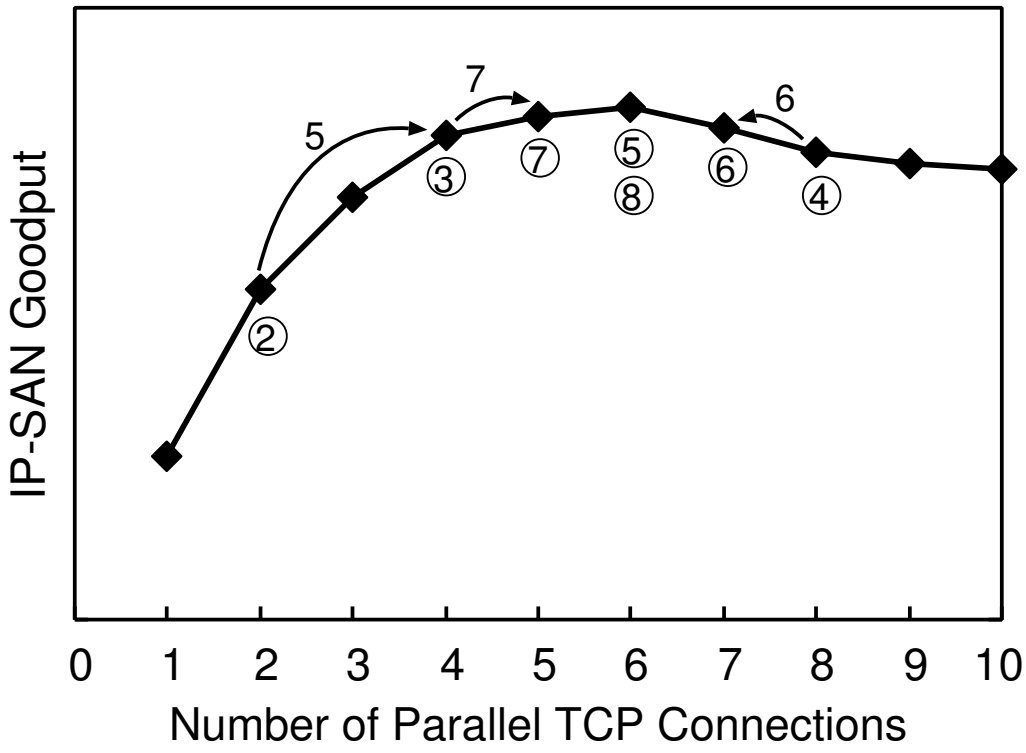



Figure 2.3: Example of BDL-APT operation when searching for the optimal number of TCP connections

## 2.5 Implementation

We implemented BDL-APT as a block device layer in the MD driver, a popular software RAID implementation included in the Linux kernel. The MD driver enables creation of a virtual block device composed of one or more underlying block devices.

The BDL-APT module in the MD driver is implemented based on the RAID-0 module with several added functions required for BDL-APT: data transfer parallelization, optimization of the number of parallel TCP connections, and goodput monitoring.

The structure of the RAID-0 module is shown in Figure 2.5. The RAID-0 module (a) creates the RAID device, (b) manages striped storage devices, (c) splits block I/O requests, and (d) assigns split block I/O requests to multiple storage devices. The block device layer in the Linux kernel handles block I/O requests from/to an application or a file system as *bio structure objects*. The RAID-0 module splits bio structure objects into multiple smaller bio structure objects. It then assigns

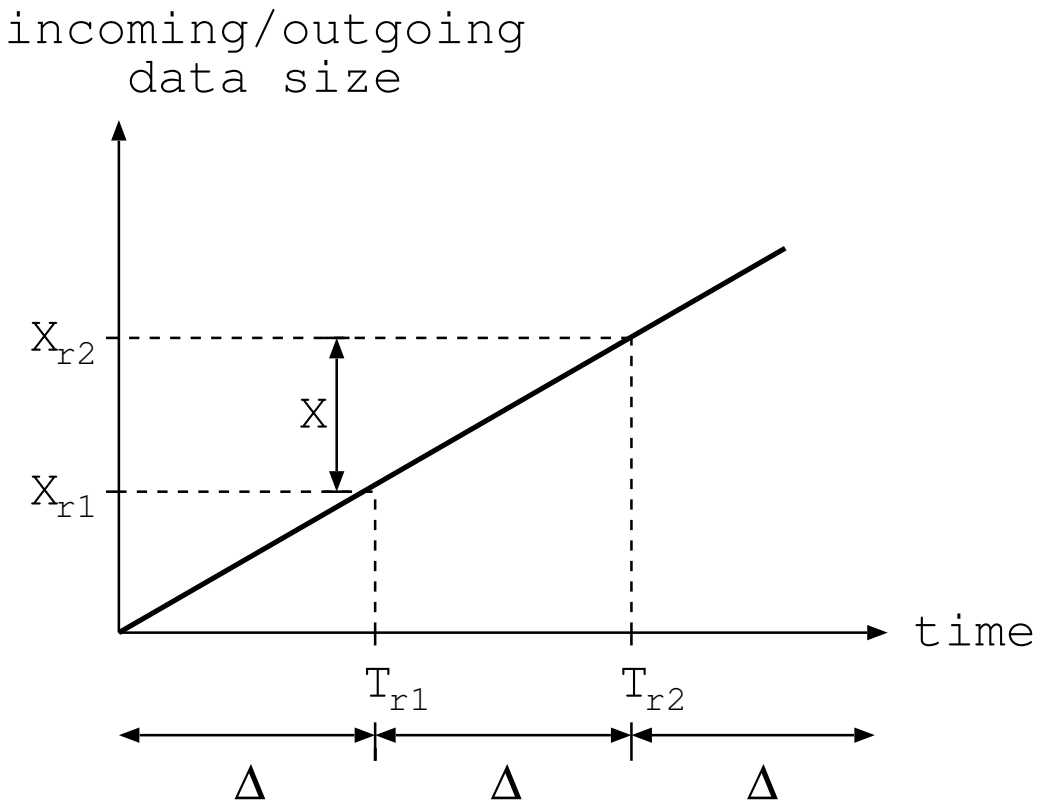



Figure 2.4: Evolution of the total incoming/outgoing data size transferred through active IP-SAN sessions.

multiple bio structure objects to the striped storage devices by changing the `bio_bdev` field in each bio structure object.

The BDL-APT module utilizes RAID-0 module features mostly as-is (see Figure 2.6). Data transfer parallelization is naturally realized by the RAID-0 module, which provides striping over multiple disks. The optimization of the number of parallel TCP connections is realized by dynamically changing the number of active IP-SAN sessions module. Goodput monitoring is realized by recording the total data size of all read/write block I/O requests and calculating the goodput at regular intervals.

In what follows, we describe how three functions required for BDL-APT, data transfer parallelization, optimization of the number of parallel TCP connections, and goodput monitoring, are realized in the BDL-APT module. Refer to, for example, [58, 60] for details of the MD driver

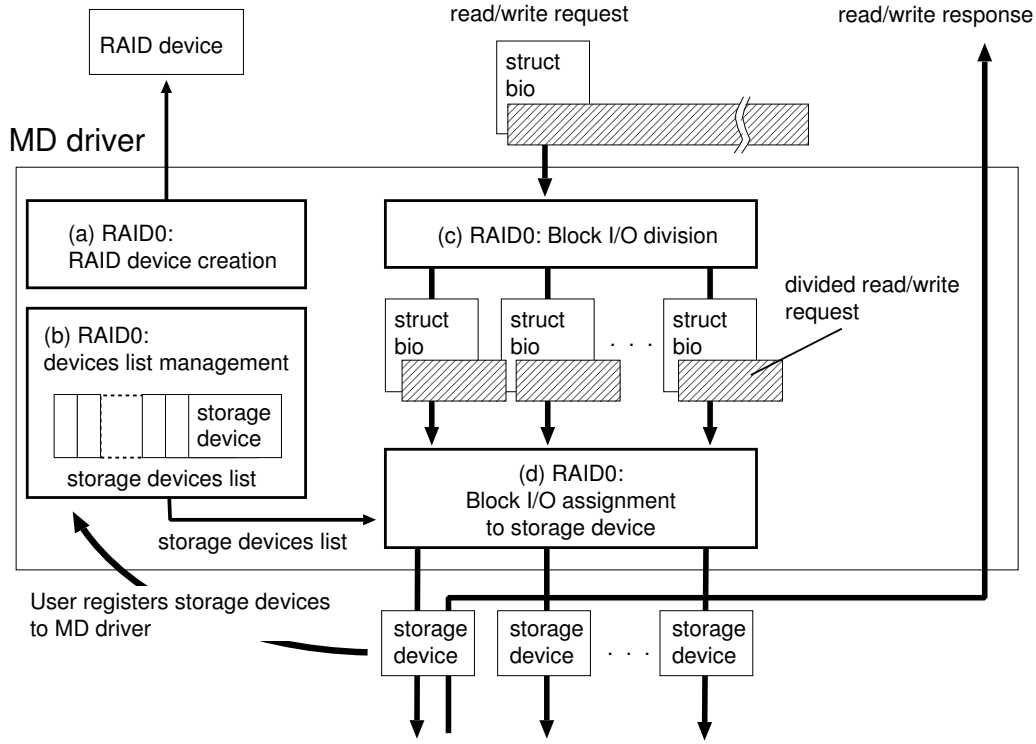



Figure 2.5: Structure of the RAID-0 module in the MD driver. The RAID-0 module (a) creates the RAID device, (b) manages striped storage devices, (c) splits block I/O requests (bio structure objects), and (d) assigns split block I/O requests to multiple storage devices.

internal and the Linux block device layer.

- Data transfer parallelization

  Data transfer is parallelized by splitting bio structure objects passed from the application or the file system and assigning split bio structure objects to the storage devices corresponding to the multiple IP-SAN sessions (see Figure 2.6).

  At the IP-SAN client, the BDL-APT module maintains one storage device per IP-SAN session. Data transfer over multiple IP-SAN sessions is thus realized because the BDL-APT module assigns bio structure objects to each storage device. Note that BDL-APT does not utilize all established IP-SAN sessions, but rather parallelizes data transfer for only a number of sessions determined by the network status. Thus, the BDL-APT module is modified from the RAID-0 module so that it dynamically assigns split bio structure objects to a subset of

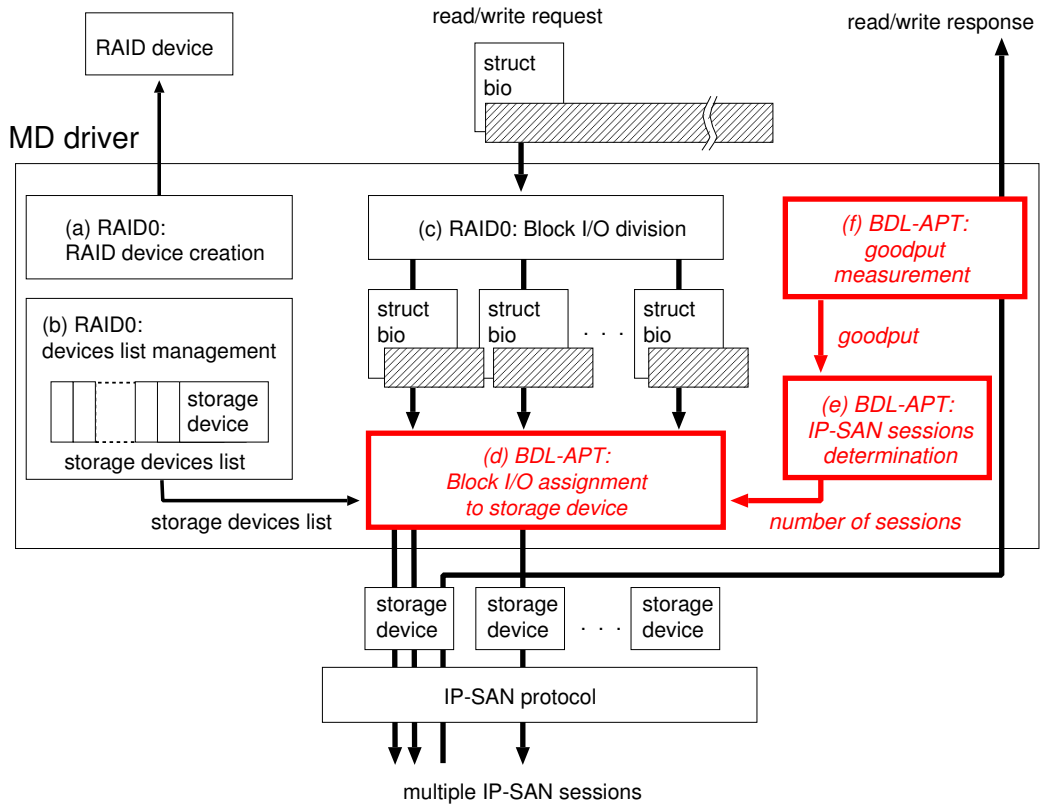



Figure 2.6: Structure of the BDL-APT module, which is based on the RAID-0 module. The BDL-APT module has several added functions to the RAID-0 module. Namely, the BDL-APT module (d) dynamically assigns divided bio structure objects to a subset of storage devices, (e) optimizes the number of active IP-SAN sessions, and (f) continuously measures the goodput of block I/O requests.

storage devices (see Figure 2.6).

- Optimization of the number of parallel TCP connections

  The APT algorithm [46, 57] is implemented in the BDL-APT module. The APT algorithm automatically determines the optimal number of parallel TCP connections according to the network status. More specifically, the APT algorithm periodically determines the optimal number of parallel TCP connections based on the goodput measurement. The BDL-APT module then changes the number of storage devices used for data transfer parallelization, thereby adjusting the degree of multiplexing. Refer to [46, 57] for the details of the APT algorithm.

- Goodput monitoring

  For monitoring the goodput, the BDL-APT module records the total block I/O response size using a callback function of the block device layer in the Linux kernel. In the Linux kernel, a callback function `endio()` can be used to invoke any function immediately after the block I/O request is completed. For every bio structure object, the pointer to a callback function can be specified in the filed `bio_end_io` of the bio structure object.

  structure objects so that the total block I/O response size can be The BDL-APT module overrides the filed `bio_end_io` of all bio calculated. The BDL-APT module calculates the goodput of data transfer at a given point in time by dividing the total block I/O response size by elapsed time. Then, the BDL-APT module resets the recorded total block I/O response size.

Note that proposed BDL-APT does not need an additional data copy. However, the block device layer needs an additional data copy between buffers. Moreover, BDL-APT does not ensure atomicity and consistency. Instead, users need to ensure atomicity and consistency, they should utilize a block device layer or a file system that ensure atomicity and consistency such as Linux journaling file system.

## 2.6 Experiment

### 2.6.1 Experiment design

We evaluated the performance of BDL-APT with several heterogeneous IP-SAN protocols, i.e., NBD, GNBD and iSCSI, in a long-fat network. To show the effectiveness of BDL-APT in a realistic environment, we conducted experiments with a network emulator while varying its bandwidth and delay settings.

The network environment comprised an IP-SAN client and storage device, and a network emulator (see Figure 2.7). We continuously transferred data from the IP-SAN storage device to the IP-SAN client. We measured the goodput for continuous read from a storage to an application. The application repeatedly requests data of 10 [Mbyte] to the storage. We conducted ten experiments and measured the average and 95% confidence interval of measurements.

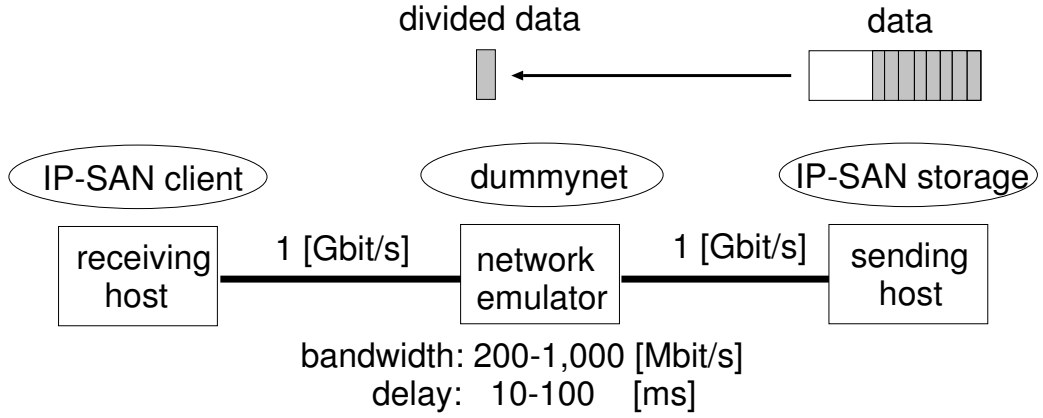



Figure 2.7: Network configuration used in experiments. IP-SAN client and storage device are connected via the network emulator to simulate a long-fat network.

We used computers with Intel Xeon 3.06 [GHz] processors (SL72G) based on NetBurst microarchitecture, 2 [Gbyte] ECC PC266 DDR SDRAM (266MHz) memory and ServerWorks GC-LE chipsets. The IP-SAN client and storage device run on Debian GNU/Linux 5.0.2 (lenny) with Linux kernel 2.6.26. The network emulator [61] runs on FreeBSD 6.4.

We used several open-source IP-SAN implementations: NBD version 2.9.11 [14], GNBD version 2.03.09 [15], Open iSCSI version 2.6-869 [62], and iSCSI enterprise target version 1.4.20 [63]. The maximum number of IP-SAN sessions in NBD, GNBD, and iSCSI are increased to 128 from their default values.

To avoid the disk drives on IP-SAN client and storage to become the bottleneck, we implemented a virtual storage device in the Linux kernel. When the network bandwidth is high enough, the access speed of the disk drives on either IP-SAN client or storage might become the performance bottleneck. In our experiments, we used our implementation of a virtual storage device, which is a virtual disk drive of an arbitrary size. The virtual storage device does not perform any physical disk drive access. Namely, reading from the virtual storage device simply returns a dummy data, and writing to the virtual storage device always succeeds but all data are simply discarded. In all experiments, we created 128 virtual storage devices with 500 [Gbyte] size. We should note that the goodput of the virtual storage device was approximately 3.2 [Gbit/s], which is sufficiently faster than the network bandwidth in our experiments, i.e., 1 [Gbit/s] at maximum.

We note that the parameter, read-ahead size of the MD driver (`/sys/block/md0/bdi/-read_ahead_kb`), in the block device must be configured appropriately in long-fat networks. The default value of read-ahead size of the MD driver in Linux 2.6.26 is the value that multiplied the number of striped storage devices by 128 [KByte], which is not large enough in long-fat networks.

Table 2.1 shows the parameter configuration used in the experiments (see [46] for the meaning of BDL-APT parameters).

### 2.6.2 Evolution of IP-SAN goodput

First, we investigate the optimization of the number of parallel TCP connections in realistic network configurations with NBD protocol. Figure 2.8 shows the aggregated NBD goodput, which means the total goodput of all active NBD sessions, when the bandwidth of the network emulator was fixed at 900 [Mbit/s] and the delay of the network emulator was fixed at 40 [ms]. For comparison purposes, NBD goodput in steady state when fixing the number of active NBD sessions at 16, 32, 64 and 128 are also plotted in the figure. One can find that the optimal number of active NBD sessions seems to exist between 16–64 from the NBD goodput with the fixed number of active NBD sessions. Moreover, this figure shows that the number of active NBD sessions is optimized at approximately 1,200 [s], and the NBD goodput converges to 828 [Mbit/s].

The evolution of the number of active NBD sessions in this scenario is shown in Figure 2.9. This figure shows that the number of active NBD sessions converges to 33 in 12 steps, i.e., approximately 1,200 [s] with $\Delta = 100$ [s]. This agrees with the result in Figure 2.8 where the optimal number of active NBD sessions exists between 16–64. From these observations, we find that BDL-APT optimizes the number of active NBD sessions at approximately 1,200 [s], and utilizes the network resource quite effectively.

In our experiments, the number of active IP-SAN sessions converged in 6–17 steps, i.e., 615–1,740 [s] with $\Delta = 100$ [s], and the average steps to converge was 12.1, i.e., 1,240 [s] with $\Delta = 100$ [s].

### 2.6.3 Effect of network bandwidth

First, the goodput of the IP-SAN protocols with and without BDL-APT in steady state was measured by changing the bottleneck link bandwidth, i.e., the bandwidth throttling at the network emulator. Figure 2.10 shows the aggregated IP-SAN goodput, which means the total goodput of all active

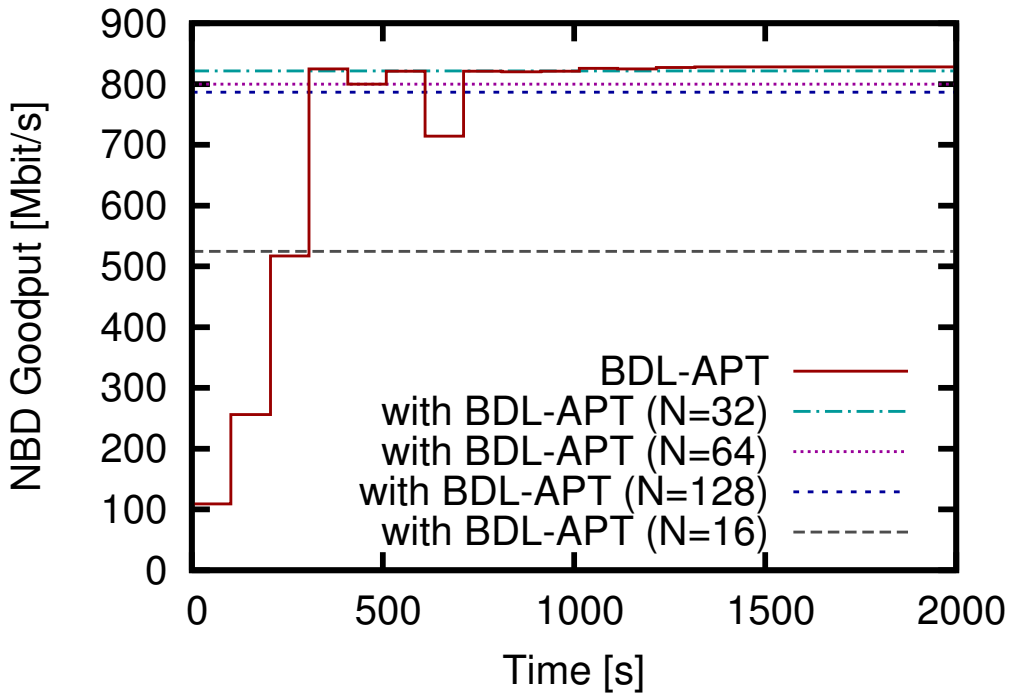



Figure 2.8: Evolution of NBD goodput. the number of active NBD sessions is optimized at approximately 1,200 [s], and the NBD goodput converges to 828 [Mbit/s].

IP-SAN sessions after the number of active IP-SAN sessions is optimized, when the bandwidth of the network emulator was varied between 200–1,000 [Mbit/s] while the delay of the network emulator was fixed at 10 [ms]. For comparison, the steady-state IP-SAN goodput with 2, 8, 32, and 128 fixed IP-SAN sessions are also plotted. In order to fix the number of active IP-SAN sessions at 2 or more, we used IP-SAN protocols with our data transfer parallelization in the BDL (see Section 2.4.3). Figure 2.10(a) shows the aggregated NBD goodput when the bottleneck link bandwidth is changed. Similarly, Figure 2.10(b) shows the aggregated GNBD goodput, and Figure 2.10(c) shows the aggregated iSCSI goodput.

Figure 2.10 shows that the IP-SAN protocols without BDL-APT cannot fully utilize the network bandwidth when the number of active IP-SAN sessions is fixed at a small value or BDL-APT is not utilized. In particular, when BDL-APT is not used, the IP-SAN protocol cannot fully utilize the network bandwidth regardless of IP-SAN protocol. This is because the bandwidth delay product

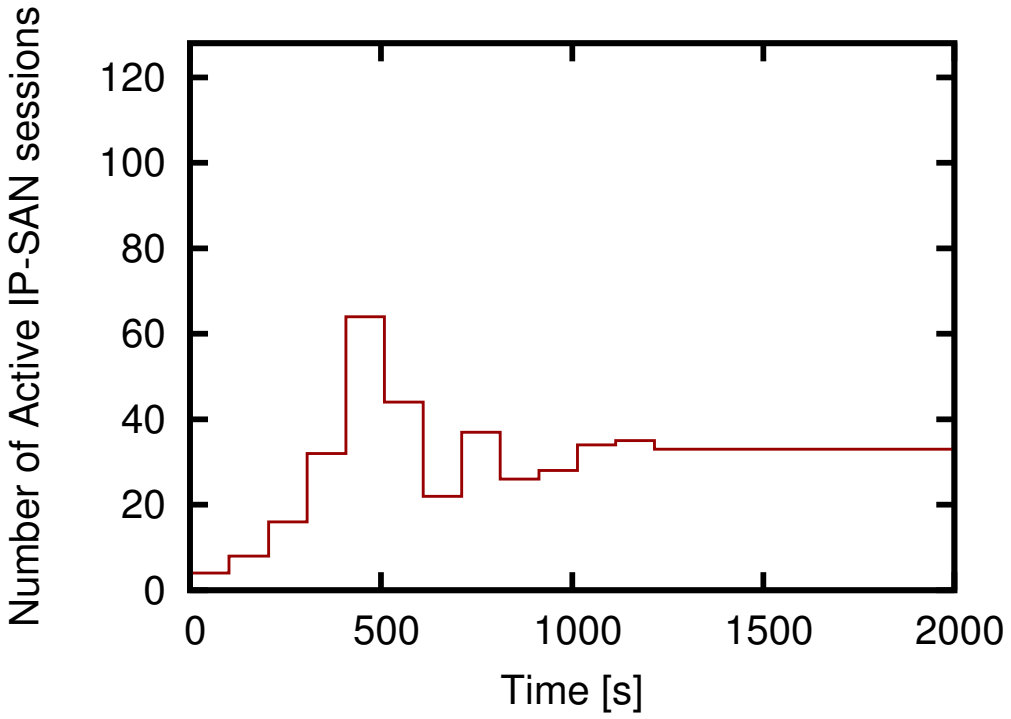



Figure 2.9: Evolution of the number of active NBD sessions. the number of active NBD sessions converges to 33 at approximately 1,200 [s].

increases as the network bandwidth becomes large. Therefore, the number of parallel TCP connections required for fully utilizing the network resources increases. Figure 2.10 shows that BDL-APT fully utilizes the bottleneck link bandwidth, regardless of IP-SAN protocols.

### 2.6.4 Effect of network delay

We next measured the goodput of the IP-SAN protocols with and without BDL-APT in steady state by changing the network delay (the delay at the network emulator). Figure 2.11 shows the aggregated IP-SAN goodput when the network emulator delay was varied between 20–100 [ms] while the bandwidth of the network emulator was fixed at 900 [Mbit/s]. For comparison, the steady-state IP-SAN goodput with 32, 64, and 128 fixed IP-SAN sessions are also plotted. Figure 2.11(a) shows the aggregated NBD goodput when the bottleneck link delay varies. Similarly, Figure 2.11(b) shows the aggregated GNBD goodput, and Figure 2.11(c) shows the aggregated iSCSI goodput.

Figure 2.11 shows that the IP-SAN goodput drops rapidly as the delay increases when the

number of active IP-SAN sessions is not fixed at an optimal value. This result shows that NBD goodput is the highest when the number of NBD sessions is fixed at 32 with 20–40 [ms] delay, at 64 with 60–80 [ms] delay, and at 128 with 100 [ms] delay. This result also shows that GNBD goodput and iSCSI goodput are the highest when the numbers of GNBD sessions and iSCSI sessions are fixed at 64 with 20–40 [ms] delay and at 128 with large delay. In particular, when BDL-APT is not used, the aggregated IP-SAN goodput was less than 20 [Mbit/s] regardless of IP-SAN protocol with 100 [ms] delay, despite the 900 [Mbit/s] network bandwidth.

Figure 2.11 shows that BDL-APT improves goodput regardless of bottleneck-link delay and IP-SAN protocol. We found that BDL-APT resolves TCP performance degradation which causes IP-SAN performance degradation and improves goodput in a long-fat network. In particular, when the NBD and GNBD protocols are used, the network bandwidth can be mostly used up regardless of bottleneck link delay. Conversely, goodput degrades as the bottleneck link delay increases when the iSCSI protocol is used. This is because in long-fat networks there are other factors that degrade the performance of the iSCSI protocol besides the performance degradation of TCP. When the bottleneck link delay is large and the iSCSI protocol is used, goodput also degrades even more than with fixed values. When parallel data transfer was performed using the iSCSI protocol, goodput was unstable in some cases. This may have caused failed adjustments, and requires further investigation. However, goodput significantly increased even when BDL-APT was used with iSCSI, and we think that practical applications will perform satisfactorily. To realize further goodput gains, custom iSCSI protocol tuning is required.

### 2.6.5 CPU Processing Load

As we have explained in Section 2.4, BDL-APT parallelizes data transfer at IP-SAN session level, rather than at TCP connection level. Establishing multiple and sometimes many IP-SAN sessions may cause a significant amount of CPU processing overhead. In this section, we therefore evaluate the amour of CPU processing overhead caused by the introduction of BDL-APT compared with vanilla NBD, GNBD, and iSCSI. The average CPU processing load during data transfer was measured by monitoring `/proc/stat` in the Linux kernel. Numbers that can be obtained through `/proc/stat` are *cumulative number of ticks*, each of which corresponds to the amount of time, measured in units of `USER_HZ`, that the system spent in idle, running, I/O-request, or interrupt states of the Linux kernel [64].

Figures 2.12 and 2.13 show the average CPU load of the IP-SAN client, i.e., receiver, and the IP-SAN storage, i.e., sender, respectively. In both figures, the average CPU loads with and without BDL-APT are plotted for NBD, GNBD, and iSCSI protocols. Similar to Section 2.6.4, the bandwidth of the network emulator was set to 900 [Mbit/s] and the bottleneck ilk delay was varied from 20 to 100 [ms]. For comparison, the average CPU loads with 32, 64, and 128 fixed IP-SAN sessions are also plotted. Not surprisingly, these figures show that the average CPU loads of both the IP-SAN client and the IP-SAN storage with BDL-APT are significantly higher than that without BDL-APT and those with BDL-APT with the fixed number of active IP-SAN sessions. This is simply because the goodput with BDL-APT is much higher than that without BDL-APT or those with BDL-APT with the fixed number of active IP-SAN sessions (see Figure 2.11). As can be seen from Figure 2.11, the goodputs with and without BDL-APT are comparable when the bottleneck link delay is very small. For instance, when the bottleneck link delay is 20 [ms] and NBD protocol is used, the goodputs with BDL-APT and with BDL-APT ($N = 32$) are approximately 800 [Mbit/s] (see Figure 2.11). In this case, the average CPU load with BDL-APT and with BDL-APT ($N = 32$) are almost identical, which indicates that multiple IP-SAN sessions causes non-negligible CPU processing overhead but the effect of APT (Automatic Parallelism Tuning) algorithm in BDL-APT on the CPU processing load is negligible.

It is not surprising that aggregating multiple IP-SAN sessions results in higher goodput than a single IP-SAN session at the cost of non-negligible CPU processing overhead. But it is still unclear how efficient the aggregation of multiple IP-SAN sessions is compared with a single IP-SAN session in terms of *the amount of CPU processings per a successful bit transfer*.

We therefore calculated the number of CPU ticks consumed for a successful bit transfer (see Figures 2.14 and 2.15). These figures show the number of CPU ticks consumed for a successful bit transfer, which is defined as the total number of CPU ticks consumed during a file transfer divided by the size of the transferred file, with NBD, GNBD, and iSCSI, respectively.

These figures show introduction of multiple IP-SAN sessions increases the amount of CPU processing per a successful bit transfer. Note that different protocols, i.e., NBD, GNBD, and iSCSI, show different tendencies. Namely, the overhead of multiple IP-SAN sessions in NBD is minimal, i.e., approximately 20–30% in both the IP-SAN client and the IP-SAN storage. On the contrary, the overhead of multiple IP-SAN sessions in iSCSI reaches approximately 100% in both the IP-SAN client and the IP-SAN storage. Such a difference should be caused by the difference in IP-SAN

protocol implementations — in particular, multiple IP-SAN sessions management. This implies that current IP-SAN protocol implementations could be improved to make it more scalable to the number of active IP-SAN sessions.

## 2.7 Summary

In this chapter, we proposed *BDL-APT*, a mechanism that maximizes the goodput of heterogeneous IP-SAN protocols in long-fat networks and that requires no modification to IP-SAN storage devices. We implemented BDL-APT as a layer of the MD driver, one of the major software RAID implementations included in the Linux kernel. We evaluated the performance of BDL-APT with heterogeneous IP-SAN protocols (NBD, GNBD and iSCSI) in a long-fat network.

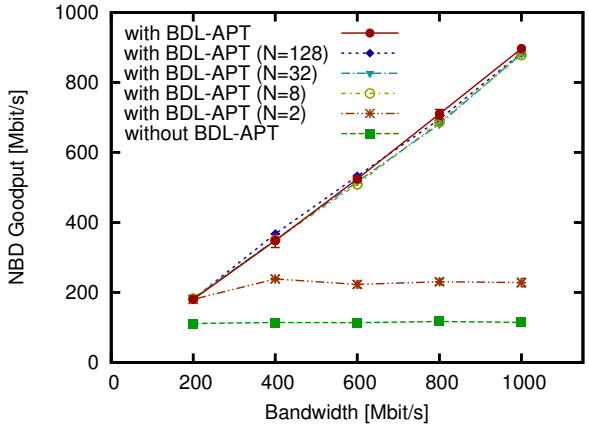
We found that BDL-APT improved IP-SAN goodput regardless of the IP-SAN protocol used. We showed that network bandwidth could be mostly used up in long-fat networks when the NBD and GNBD protocols were used. In long-fat networks, we found that BDL-APT resolves TCP performance degradation which causes IP-SAN performance degradation but does not maximize IP-SAN goodput under the iSCSI protocol. This is because in long-fat networks there are other factors that degrade the performance of the iSCSI protocol besides the performance degradation of TCP.

In the network environment with little traffic changes, such as the leased line, our BDL-APT accelerates a data transfer sufficiently. Since storage networks are mainly used on leased lines now, we believe that increase in the speed of storage networks using the leased line is sufficient. However, we expect that the future storage networks might be used on the networks with intense traffic changes, such as the Internet. High speed data transfer for such a network is also required.

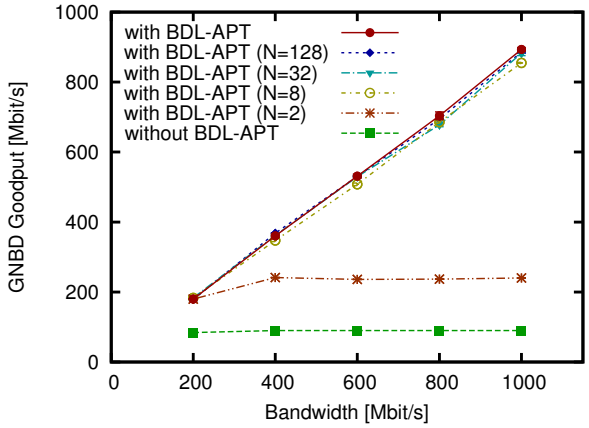
Therefore, investigation of the high speed data transfer technology for networks where traffic changes dynamically is one of our future research directions. Specifically, we show clearly robustness of BDL-APT to traffic changes. Moreover, to achieve high goodput constantly against changes in traffic, we will develop the fast adjustment of the number of sessions.

Table 2.1: Default parameters used in experiments

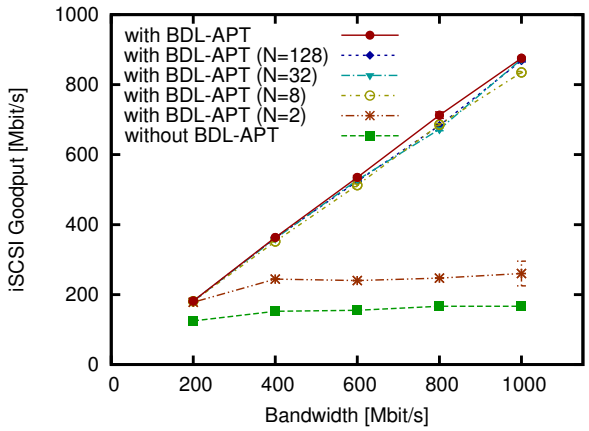
| | | |
|---|---|---|
| BDL-APT parameters | | |
| chunk size | 128 | [Kbyte] |
| initial number of IP-SAN sessions $N_0$ | 4 | |
| maximum number of IP-SAN sessions | 128 | |
| multiplicative increase factor $\alpha$ | 2 | |
| target value of chunk transfer time $\Delta$ | 100 | [s] |
| Block device parameter | | |
| read-ahead size of the MD driver | 128 | [Mbyte] |
| NBD parameter | | |
| block size | 1,024 | [Kbyte] |
| GNBD parameters | | |
| No parameter | | |
| iSCSI parameters | | |
| MaxBurstLength | 16 | [Mbyte] |
| ImmediateData | no | |
| InitialR2T | yes | |
| TCP parameter | | |
| TCP socket buffer size | 512 | [Kbyte] |
| TCP version | NewReno | |
| TCP SACK | enable | |
| NIC parameter | | |
| MTU | 1500 | [Byte] |
| Network emulator | | |
| bandwidth | 900 | [Mbit/s] |
| delay | 10 | [ms] |
| packet loss rate | 0 | |
| buffer size | 5,000 | [packet] |




(a) NBD

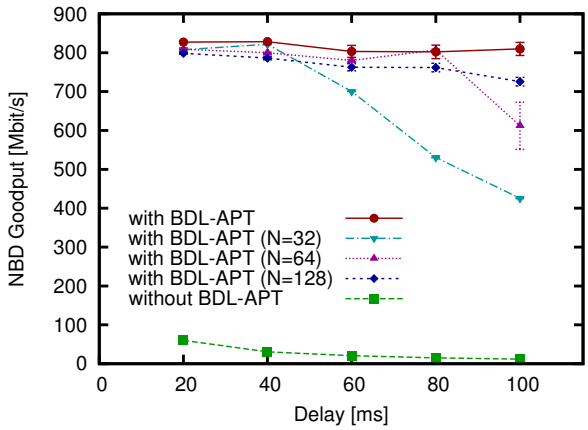



(b) GNBD

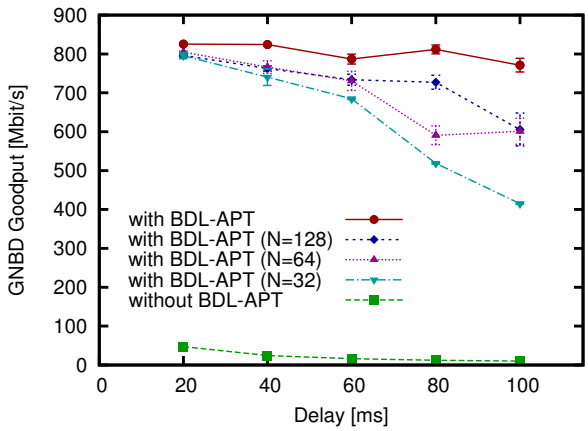



(c) iSCSI

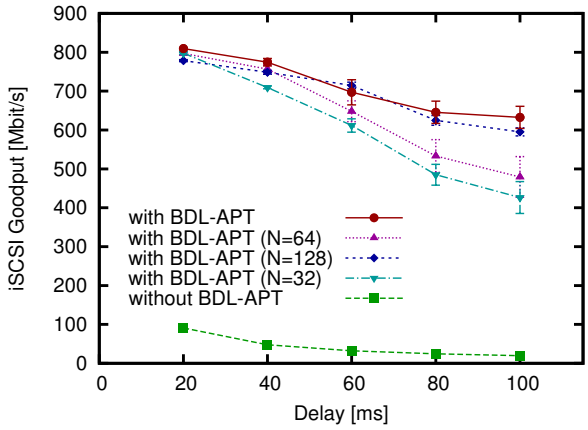
Figure 2.10: Bottleneck link bandwidth vs. IP-SAN goodput




(a) NBD

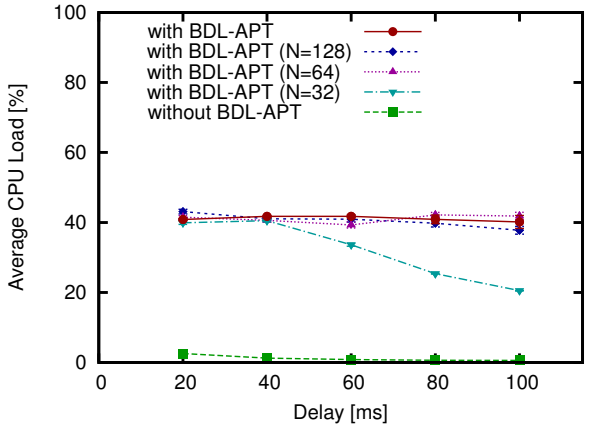



(b) GNBD

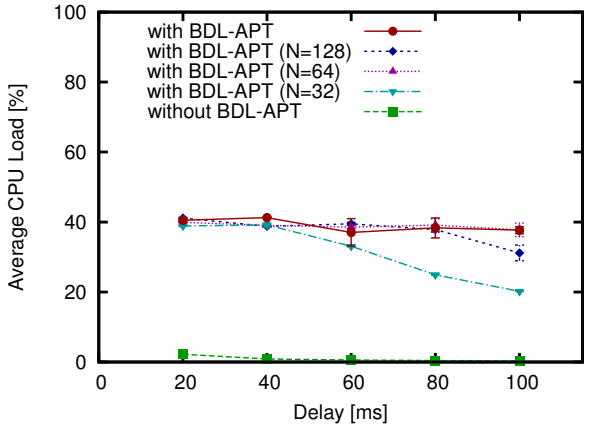



(c) iSCSI

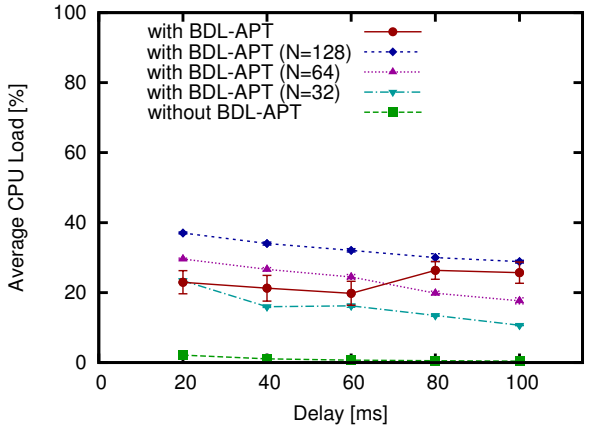
Figure 2.11: Bottleneck link delay vs. IP-SAN goodput




(a) NBD

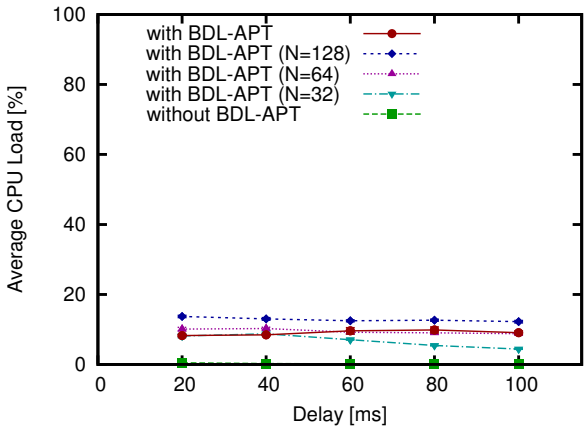



(b) GNBD

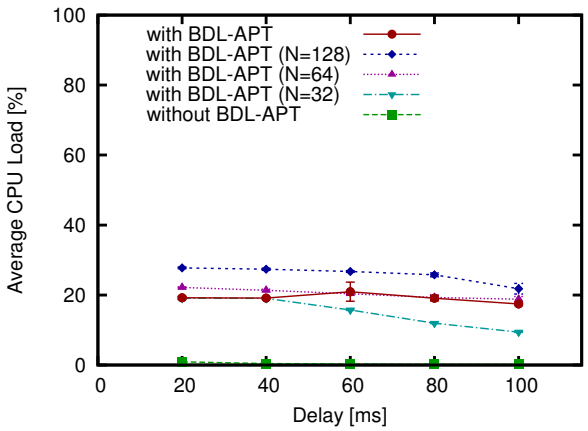



(c) iSCSI

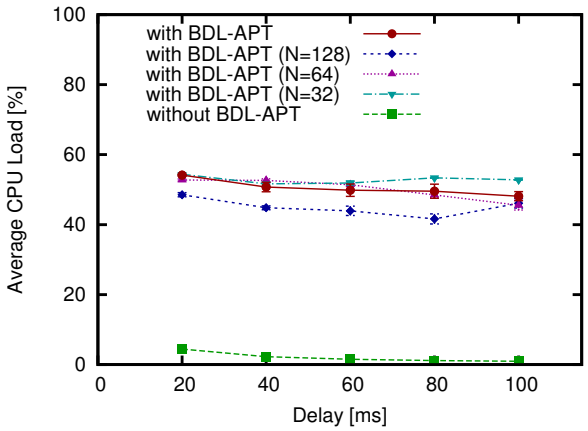
Figure 2.12: The average CPU load of the IP-SAN client




(a) NBD

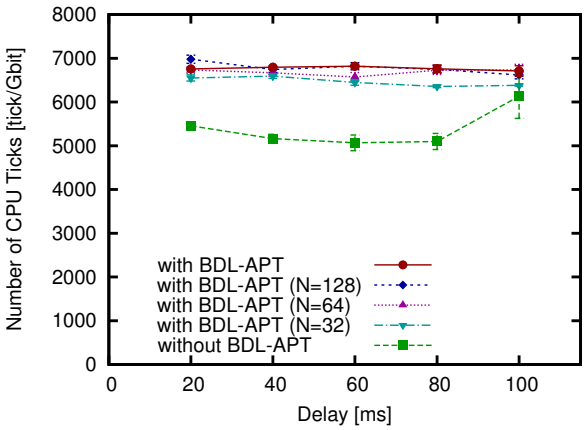



(b) GNBD

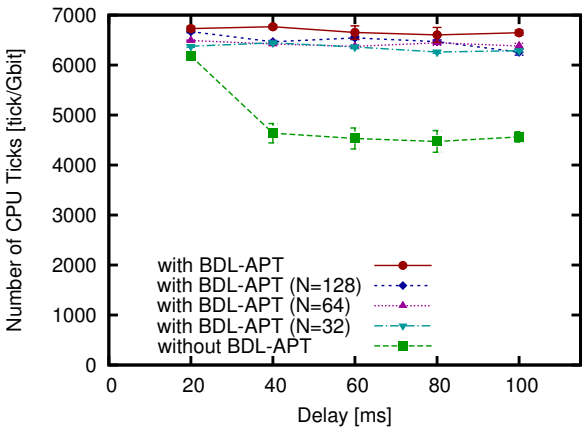



(c) iSCSI

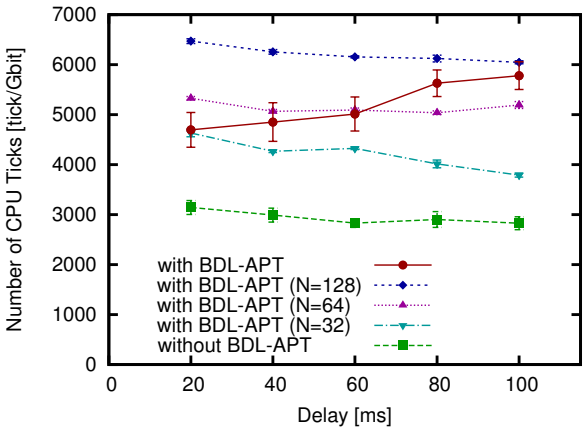
Figure 2.13: The average CPU load of the IP-SAN storage




(a) NBD

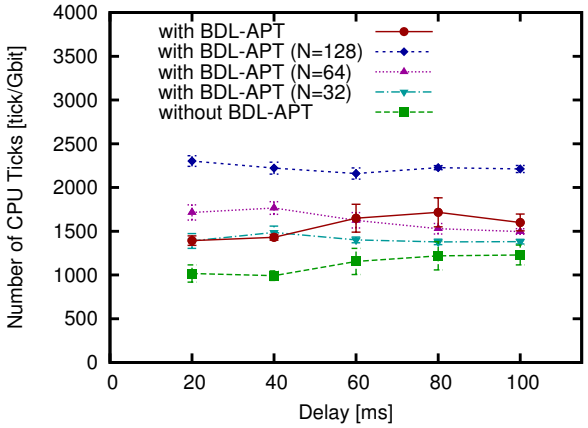



(b) GNBD

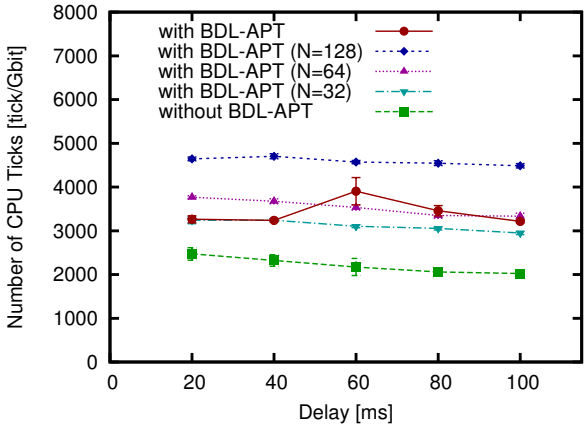



(c) iSCSI

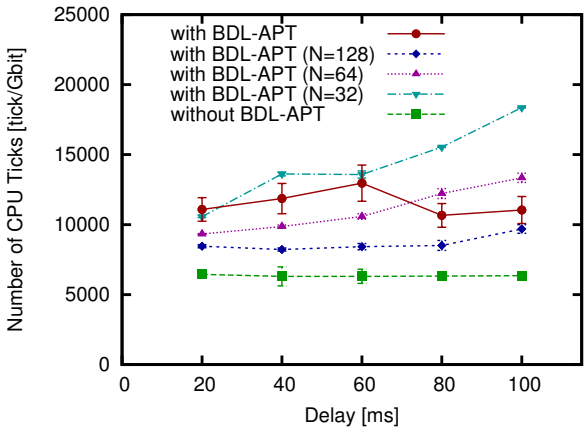
Figure 2.14: The number of CPU ticks consumed for a successful bit transfer on the IP-SAN client




(a) NBD




(b) GNBD




(c) iSCSI

Figure 2.15: The number of CPU ticks consumed for a successful bit transfer on the IP-SAN storage

# Chapter 3

# Estimation of Performance Improvement Derived from TCP/IP Offload Engine with Software Emulation

## 3.1 Introduction

End system protocol processing is becoming a bottleneck in end-to-end communication because of rapid increases in the speed and bandwidth of networks such as 10 Gigabit Ethernet [65]. Since the performance of end-to-end communication is limited by the processing speed of the bottleneck, such bottlenecks must be remedied to increase the speed of communication networks. Most internet traffic is transferred by TCP (Transmission Control Protocol) [66] and its derivatives [67, 68]. Hence, improved TCP protocol processing would result in a significant network performance gain [69, 70]. According to past experimental verifications, with the exception of very high-speed networks, 1 bit/s protocol processing generally requires about a 1 Hz processor [71]. This generalization implies that sufficient protocol processing to fully utilize the capacity of 10 Gigabit Ethernet with a processor that operates at several gigahertz would be challenging.

TOE (TCP/IP Offload Engine) has been studied by many researchers as a means of accelerating protocol processing. TOE is a technology that uses a hardware accelerator to perform heavy TCP/IP processing in end systems. There have been many performance studies of TOE devices [32–41], and many TOE devices have been developed and installed, particularly in high-end systems [49,50].

There remain, however, open issues related to TOE devices. For example, it has not been clarified which part of TCP/IP processing should be performed with hardware and which with software, or how end-to-end TCP/IP performance is affected by the introduction of a TOE device.

In TOE design, the amount of CPU processing load reduction alone is not a good measure. TOE design must also take into account improvements in end-to-end network performance, e.g., throughput and latency, resulting from the introduction of TOE [51, 52]. For instance, it is known that memory copies between the user and kernel spaces and packet checksum calculations are bottlenecks in TCP processing [69, 72]. CPU processing load reduction resulting from the introduction of a TOE device can be easily measured using modern profilers. However, end-to-end network performance improvements when TOE is introduced into a machine can not be easily measured. The improvements depend on all software processing that runs on the machine and also depend on the machine performance such as CPU processing speed, memory access speed and I/O bus speed. It is difficult to calculate the improvements by analysis and simulation. Therefore, in current TOE design, measurement of end-to-end network performance improvements generally requires actual experiments with a TOE implementation. Namely, in TOE design, to measure performance improvements derived from introducing a TOE device, it is necessary to implement a TOE prototype really. A method for measuring the effectiveness of protocol offloading with a specific TOE device that does not require installation of the device itself would significantly reduce the cost of TOE design and development.

In this chapter, we therefore propose VOSE (Virtual Offloading with Software Emulation), which is a technique for measuring TCP/IP performance improvements derived from different type of TOE devices without implementing TOE devices really. VOSE enables offloading without hardware by virtually emulating TOE processing, e.g., bypassing software protocol processing without violating protocol consistency, on both the source and destination end hosts. VOSE realizes protocol processing offloading by utilizing symmetry of protocol processing between the sender and the receiver while preserving its integrity.

In addition, for demonstrating the effectiveness of VOSE, we apply VOSE to the TCP checksum and IPsec protocol. First, we apply VOSE to the TCP checksum in a Linux kernel. We extensively examine the accuracy of virtual offloading with VOSE by comparing performance, i.e., end-to-end performance and CPU processing overhead, between virtual offloading with VOSE and hardware offloading with a dedicated TOE device. Second, we apply VOSE to header authenticating and

payload encryption in IPsec protocol. We estimate the performance improvement that is derived from several TOE devices of IPsec and combinations of the TOE devices because TOE devices of IPsec have not been evaluated sufficiently yet.

One of the important contributions of this work is as follows; VOSE enables to measure performance improvements derived from introducing a TOE device, without implementing TOE prototypes really. More specifically, unlike simulators, the improvements in real environments, i.e., operating environments, can be measured by using VOSE because VOSE enables measurements using real computers, networks, applications and protocols. This leads to 1) minimizing costs of design and development, 2) minimizing development periods and 3) optimizing TOE performance. 1) VOSE enables that TOE designers elucidate bottlenecks of various TCP/IP processings by trial and error and implement a TOE device that offloads the bottlenecks only. Therefore, VOSE realizes the maximum effect at the minimum cost of design and development. 2) VOSE eliminates the need for implementing many TOE prototypes to satisfy performance requirements and enables testing various patterns of TCP/IP offloads in a short period. 3) VOSE enables optimizing the balance between software and hardware processings because TOE designers measure performance of the various patterns of offloadings and clarify processings that should be performed with software and the processings that should be performed with hardware.

The construction of this chapter is as follows: Section 3.2 summarizes previous work related to TCP/IP processing loads and TOE. Section 3.3 gives an overview of TOE, and Section 3.4 describes VOSE. Section 3.5 describes the TCP checksum calculations experiments: end-to-end performance measurements of virtual offloading with VOSE versus hardware offloading with TOE, and system loads. Section 3.6 describes the performance improvement caused by introduction of several types of TOE devices for header authenticating and payload encryption in IPsec protocol. Finally, Section 3.7 summarizes this chapter and discusses directions for future study.

## 3.2 Related Work

Many studies have measured TCP processing loads, and in particular there are several studies that measure processing loads in relatively high-speed networks exceeding 1 Gbit/s [70, 71, 73].

Chase *et al.* surveyed high-speed methods related to the TCP/IP stack and experimentally evaluated their effectiveness [73]. They targeted overhead mitigation at the packet level (using large

frames, mitigating the number of device interruptions) and the byte level (integration of copy avoidance, checksum offloading, copy processing, and checksum calculations). Chase *et al.* show that integrating copy processing and checksum calculations was ineffective, and that copy avoidance was effective (approximately a 50–100% improvement in throughput). They also found that performing checksum calculations with hardware was effective, giving an approximately 30% improvement in throughput.

Foong *et al.* measured the processing load of the TCP/IP stack under Linux kernel 2.4 [71]. In particular, they investigated the scalability of the TCP/IP stack with CPU clock speed, and found that the convention that 1 bit/s protocol processing requires 1 Hz of processing power fails when the CPU clock exceeds 1 GHz. Foong *et al.* showed that the balance of sender and receiver loads changes with data size, and that checksum offloading was effective only for large data sizes, thus making increasing TCP/IP stack speeds via this method a complex task.

Although many other experimental evaluations of the effectiveness of TOE exist [32–40], most measure performance under a specific environment. For instance, Fukuju and Ishihara used a Toshiba TOE and measured the throughput when offloading IPsec processing to the TOE [32]. Ghadia evaluated the throughput and delay when processing all aspects of TCP/IP by hardware, and showed that when doing so throughput more than doubles, while delay is reduced approximately 20% [40]. Jang *et al.* evaluated performance when establishing connections and controlling flow by software, and processing packet buffers management, TCP headers, and ACK packets by hardware, and found that doing so reduces CPU load to $\frac{1}{5}$ or less [37]. Benjamin and Patrick compared throughput when performing TCP checksum calculations on a TOE versus processing solely by software, and found that the TOE approximately doubled throughput [35].

Thus, many studies have demonstrated the effectiveness of TOE. Each of these studies, however, measures performance when offloading TCP/IP processing under specific experimental environments. Moreover, there has been insufficient examination of how TCP/IP protocol processing should be divided between hardware and software to maximize performance gains.

Westrelin *et al.* measured the TOE offloading effect using software emulation under symmetrical type multiprocessing, a computer design by which multiple processors can share and manage physical memory [41]. Doing so, they realized processing equivalent to hardware by performing part of TCP/IP handling on processors other than the main processors. However, because this method uses code particular to Solaris to perform processing with the regular core, this method

cannot be used with other operating systems. Moreover, because this method connects processor boards over the PCI bus, there is overhead associated with data transport between memories. Because there are limits to the number of cores and amount of physical memory that symmetrical type multiprocessing can use, TOE performance can be mimicked only to a certain degree. Therefore, there remains a need for methods of offloading effect measuring performance improvement with the introduction of a TOE device, regardless of the type of computer used or experimental environment.

## 3.3 TOE (TCP/IP Offload Engine)

TOE is a technology that uses a hardware accelerator to perform heavy TCP/IP processing in end systems [36–40]. In conventional systems, TCP/IP protocol processing is performed by the host computer's CPU. Under TOE, however, a part of TCP/IP processing is directly performed by hardware without CPU intervention.

Below, as an example, we explain TOE operation when offloading TCP checksum calculations. Figure 3.1 shows how introducing TOE changes the TCP/IP processing flow. Figure 3.1 (a) is the processing flow when performing all TCP/IP processing by software. In the transport and network layers, several TCP/IP activities such as flow control and checksum addition are performed on data received from the application layer. Generally, all of these processes are performed by the CPU. As the network transmission speed increases, the number of CPU memory accesses will also increase, in turn increasing CPU load.

In contrast, Figure 3.1 (b) shows the processing flow when offloading checksum calculations to TOE. In this case, dedicated hardware performs checksum calculations in place of the CPU, alleviating the need for memory access and mitigating increased CPU load.

TCP/IP processing tasks that are generally suited to hardware offloading have the following characteristics:

**(A)** Processing suited to hardware

Some data processing is better performed by dedicated hardware than by a general purpose CPU. CPUs perform many operations in succession at high speeds. Dedicated hardware, on the other hand, can be advantageous when processing can be performed in parallel. Examples of such processing includes checksum calculations and handling of fixed length data, such as encryption and decryption. In the case of TCP/IP processing, therefore, tasks such as TCP

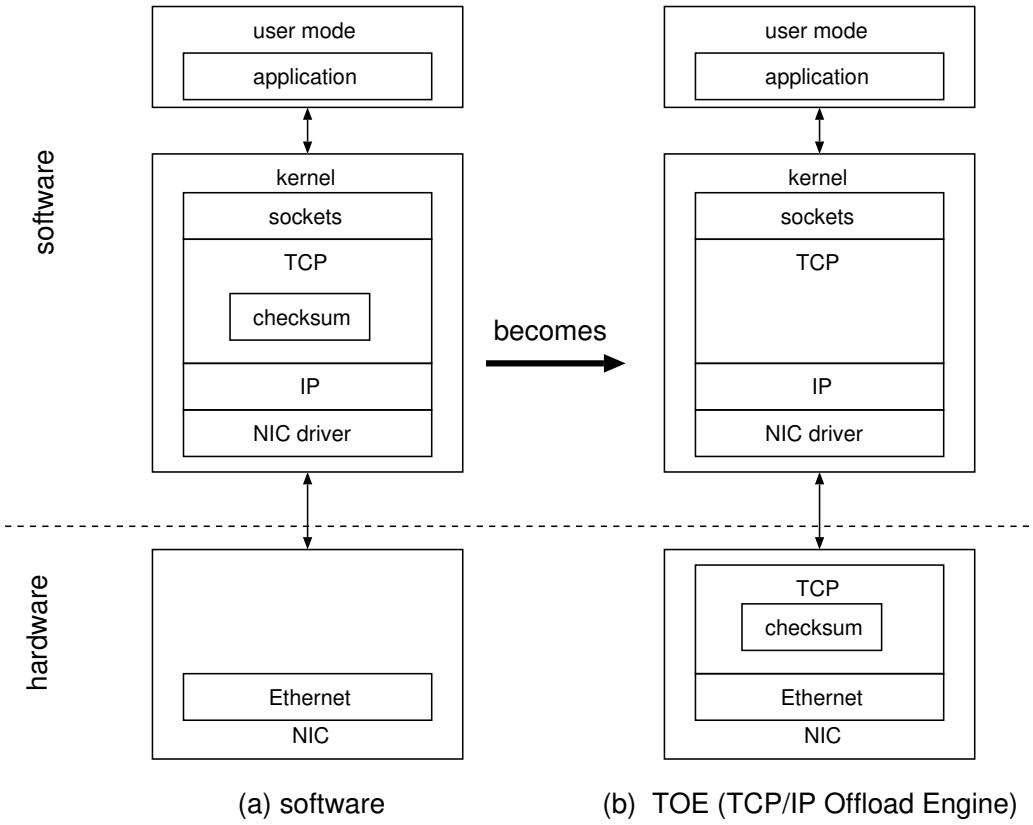



Figure 3.1: Changes in TCP/IP processing under TOE: (a) the flow when performing all TCP processing in software, and (b) the processing flow when offloading checksum calculations to TOE

checksum calculations, IPsec encryption and decryption, and memory copies are considered to be suited to hardware processing.

**(B)** Processing that increases with increased transmission speed

Some processing that is not problematic in a low-speed network can become an issue as network speed increases. For example, the burden of payload processing increases with longer packet length, faster transmission speed, and larger frame size. Specific examples from TCP/IP processing include TCP payload checksum calculations and IPsec encryption and decryption.

**(C)** Processing that requires real-time execution

This characteristic refers to processing that must complete within a guaranteed time. When

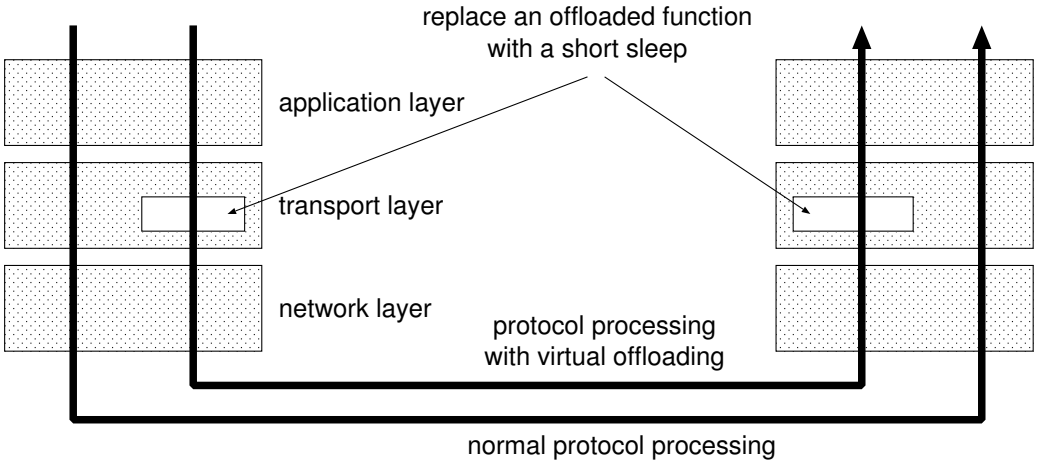



Figure 3.2: VOSE realizes virtual offloading of protocol processing by utilizing the symmetry of protocol processing between the sender and the receiver while preserving its integrity.

dedicated hardware is used, hardware processing is not affected by other processing, allowing completion within a constant time. In contrast, because processing time can vary with CPU status or memory conditions, software processing cannot guarantee completion times. Video and audio streaming protocols are examples of processing that requires real-time execution. However, TCP/IP networks are best-effort networks where packet delivery is not guaranteed, and end-to-end delays may occur. Hence, processing that requires real-time execution under TCP/IP are limited to, for example, retransmission timers.

Processing with the above characteristics is the main target for TOE offloading. In the following section, we propose and evaluate a method of measuring end-to-end performance gains when such processes are performed on hardware.

## 3.4 Proposal of VOSE (Virtual Offloading with Software Emulation) for Estimating Performance Improvement with TCP/IP Protocol Offloading

The fundamental idea of VOSE is partial TCP/IP processing of sender or receiver data by virtualized hardware under software emulation. Since TCP/IP is an end-to-end communication protocol, the bulk of TCP/IP processing is symmetrical on both sender and receiver sides.

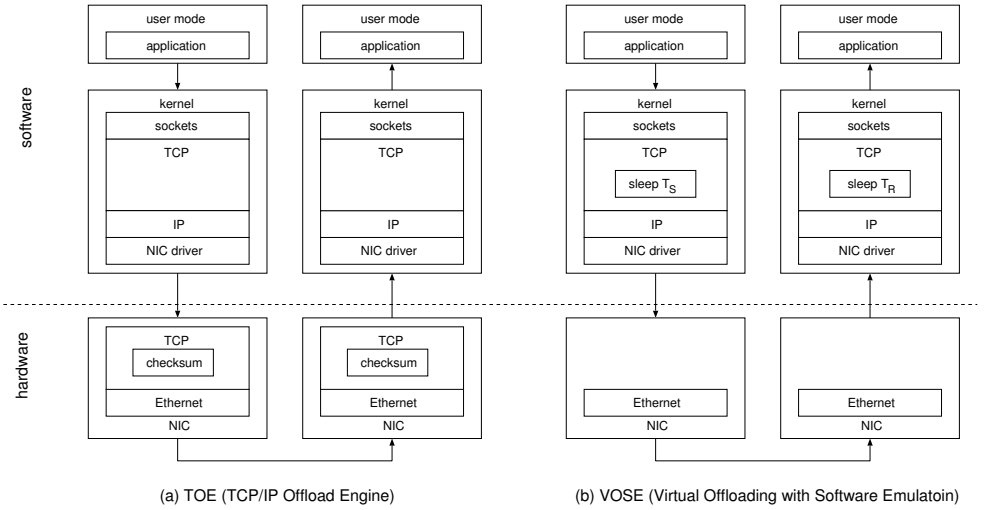



Figure 3.3: Changes in TCP/IP processing by VOSE: (a) the processing flow in the case of offloading checksum calculations to TOE, and (b) the processing flow in the case of virtually offloading checksum calculations.

VOSE emulates offloading of protocol processing by utilizing this symmetry. For instance, offloading is replaced by short-time sleep in the transport layer, in a form where the integrity of the processing is maintained between sender and receiver (see Figure 3.2).

VOSE can emulate arbitrary TOE devices with different processing speeds by changing the duration of a short sleep. Thus, with VOSE, a TOE designer can measure the overall performance with different types of TOE devices without developing and implementing the TOE hardware.

However, because TCP/IP requires processing of multiple transport factors, it is necessary to consider which processes should be offloaded. We classify protocol processing according to the following three types:

1. Symmetrical

   Symmetrical processing involves processing between sender and receiver. Examples include TCP checksum addition and verification by the sender and the receiver, packet order control, and management of multiple connections.

2. Independent

   Independent processing is performed by either sender or receiver. An example is memory

data copies from user space to the kernel space by the sender, or data copies from the kernel space to the user space by the receiver.

3. Cooperative

   Cooperative processing is performed by sender and receiver together. Examples include congestion control such as window control based on ACK packets from the receiver, and flow control to the network based on the advertising window from the receiver.

Of these, virtual offloading is possible for symmetrical protocol processing by bypassing symmetrical sender and receiver protocol processing simultaneously.

Independent protocol processing is completed by sender or receiver, so virtual offloading is possible if integrity in protocol processing are maintained.

Cooperative protocol processing, however, may affect other processing, so the potential for virtual offloading and how it should be performed depends on the particular processing performed.

It should be noted that most *heavy* protocol processing are classified as either symmetrical or independent, which can be virtually offloaded with VOSE. Cooperative protocol processing needs *cooperation* between the sender and the receiver, which results in slow processing. So, TOE devices for cooperative processing are rarely required.

In what follows, as an example of virtual offloading, we use TCP checksum calculations that is symmetrical processing. In the transport layer, TCP checksums are used to detect bit errors in the TCP packet headers and payloads. When sender sends a packet, checksums are calculated for the pseudo TCP header, the TCP header, and the TCP payload, and the value is stored in the TCP header [66]. When receiver receives the packet, the checksums are calculated again and compared with the value in the header to ensure that the packet is not corrupt.

TCP checksum calculations are performed strictly in the transport layer. Hence, if, as shown in Figure 3.3 (b), sender and receiver checksum calculations are replaced by sleep operations of $T_{\mathrm{S}}$ [s] and $T_{\mathrm{R}}$ [s], respectively, those calculations can be considered to have been virtually offloaded (see Figure 3.3). How $T_{\mathrm{S}}$ and $T_{\mathrm{R}}$ should be configured depends on the TOE architecture, the access speed of memory and the bus, and interrupt processing of the operating system. Nonetheless, the theoretical maximum performance of TOE can be investigated by setting $T_{\mathrm{S}}$ and $T_{\mathrm{R}}$ to 0.

Recall that VOSE is not specific to virtual offloading of TCP checksum. Instead, as we have explained above, it can be used different types of protocol processing. For instance, application of

VOSE to other protocol processing than TCP checksum will be discussed in Section 6.

Thereby, we can perform communication normally between hosts in which virtual offloading is carried out by VOSE, allowing us to then measure the effects of TCP/IP offloading.

## 3.5 Evaluation of VOSE

In this section, we evaluate the accuracy of virtual offloading with VOSE. We apply VOSE to the TCP checksum calculations in a Linux kernel. We compare end-to-end performance and system loads between virtual and hardware offloading.

### 3.5.1 Experimental environments

To evaluate the accuracy of virtual offloading with VOSE, we perform experiments by carrying out virtual offloading of TCP checksum calculations. Hardware offloading of checksum calculations has already been carried out by NIC (Network Interface Card). We can therefore evaluate the accuracy of virtual offloading with VOSE by comparing experimental results with NIC hardware offloading.

In the experiments, we use two experimental environments as follows.

- Experimental environment A

  Two computers are connected by Gigabit Ethernet. The computers with a Intel Pentium 4 3.03 GHz CPU and 2 Gbyte RAM are used. A NIC installed to the computers is an Intel PRO/1000 Network Driver.

- Experimental environment B

  Two computers are connected by 10 Gigabit Ethernet. The computers with an Intel Core i7 3.2 GHz CPU and 3 Gbyte RAM are used. The processor is operated with only a single core active. A NIC installed to the computers is an Intel PRO/10G Network Driver.

The operating system in both environments is Debian GNU/Linux 5.0.2 (Linux kernel 2.6.30). Each experiment is conducted with hardware offloading of TCP checksum calculations was carried out by TOE on the NIC, and with virtual offloading carried out by VOSE. To eliminate the

```
      rdtsc
      addl $clocks_to_sleep, eax
      adcl $0, edx
      movl eax, ecx
      movl edx, ebx
loop:
      subl ecx, eax
      sbbl ebx, edx
      js   loop
```

Figure 3.4: Realization of the busy loop using the RDTSC command: A loop is carried out for the specified number of cycles using the IA32/IA64 processor RDTSC instruction, which waits for `$clocks_to_sleep` clocks.

effects of offloading processing other than TCP checksum calculations, we turn off with TSO (TCP Segmentation Offload), GSO (Generic Segmentation Offload), and LRO (Large Receive Offload).

In the experiments, we use iperf as benchmarking software for continuously transferring data for 60 [s]. We measure CPU utilization for 60 [s] using the information in `/proc/stat`. We repeat the experiments 10 times, and then measure average values of throughput and CPU utilization and calculate their 95% confidence intervals.

### 3.5.2 Virtual offloading of TCP checksum calculations

We explain how TCP checksum calculations in the Linux kernel version 2.6.30 can be virtually offloaded with VOSE.

Virtual offloading of TCP checksum calculations can be realized by replacing `tcp_v4_send_check()` of the sender and `tcp_v4_checksum_init()` of the receiver functions with a short sleep of $T_S$ and $T_R$ in both the sender and the receiver, respectively. To bypass TCP checksum calculations in `csum_and_copy_from_user()` of the sender, we replace that function call with function call of `copy_from_user()` in the sender. To bypass TCP checksum calculations in `tcp_v4_checksum_init()` of the receiver, `skb->ip_summed` is set to `CHECKSUM_UNNECESSARY`, and `__tcp_checksum_complete()` is not performed.

We realize a sleep processing using a busy loop of CPU. To realize the busy loop, the loop is carried out for the specified number of cycles using the IA32 and IA64 Intel processor RDTSC instruction (see Figure 3.4).

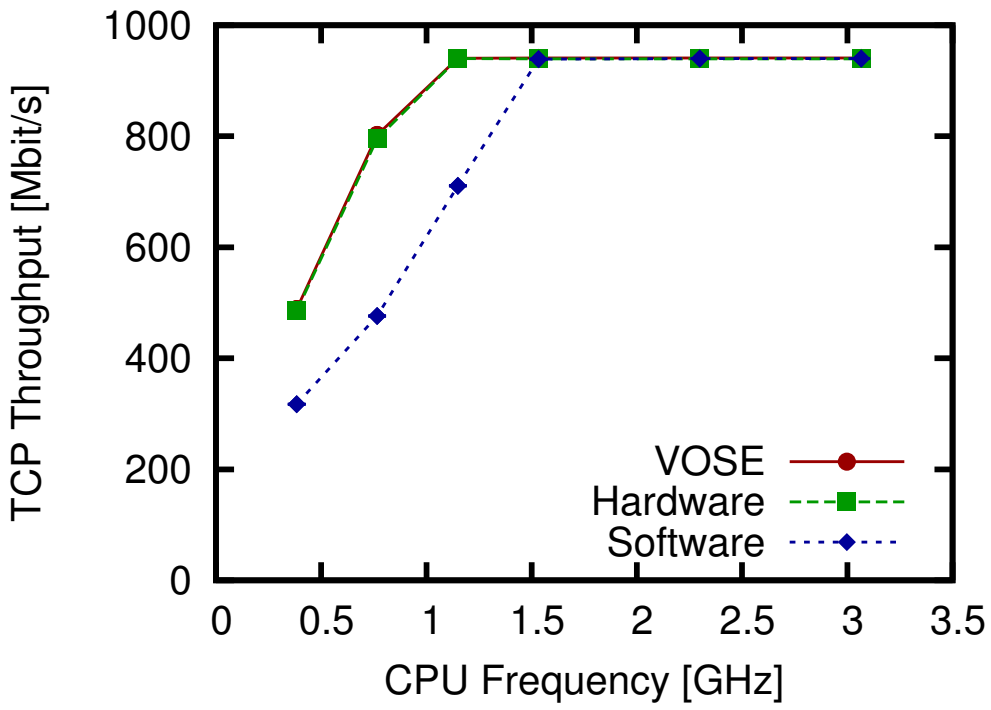



Figure 3.5: End-to-end performance (effective throughput) when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A)

We program a busy loop by using an inline assembler to realize the sleep processing of $T_S$ and $T_R$. Note that each loop requires approximately 24 clock cycles to process and that the granularity of sleep times is specified during the busy loop.

### 3.5.3 Results

First, Figures 3.5 and 3.6 show the end-to-end performance, i.e., effective throughput, with virtual offloading by VOSE (labeled as 'VOSE') in the Gigabit Ethernet (experimental environment A) and the 10 Gigabit Ethernet (experimental environment B) when the CPU operating frequency of the sender and receiver is varied. For comparison purposes, the end-to-end performance with TOE (labeled as 'Hardware') without TOE (labeled as 'Software') are also shown in the figure. In these experiments, we set $T_S$ and $T_R$ to 0.

Figures 3.5 and 3.6 show that throughput mostly coincides in both the case where checksum

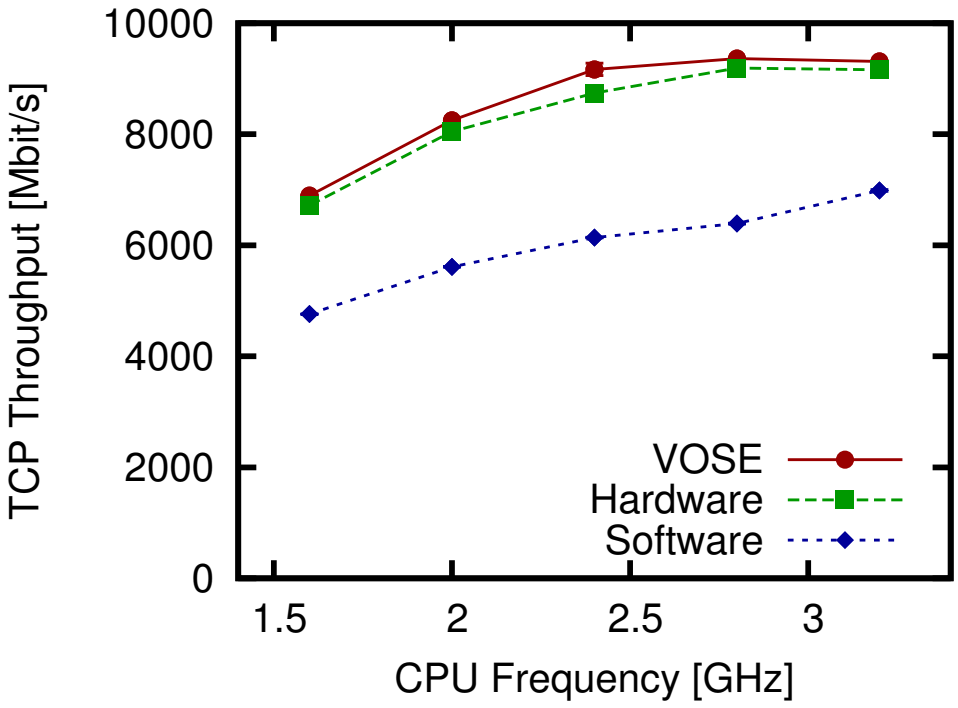



Figure 3.6: End-to-end performance (effective throughput) when changing the sender and receiver CPU operating frequency over 10 Gigabit Ethernet (experimental environment B)

calculations are offloaded by VOSE and the case where calculations are offloaded by hardware. The average error was below 1% over Gigabit Ethernet, and below 3% over 10 Gigabit Ethernet. These observations indicate that VOSE correctly emulates the offloading effect of TCP checksum calculations.

Next, we focus on CPU utilization of the sender (see Figures 3.7 and 3.8). Figure 3.7 shows that CPU utilization of the sender over Gigabit Ethernet mostly coincides both when checksum calculations are offloaded by VOSE and when that are performed by hardware. In both cases the average error is 3% or less. Thus we find that emulation of the offloading effect is correctly carried out by VOSE. Over 10 Gigabit Ethernet (see Figure 3.8), when virtually offloading with VOSE, the sender CPU utilization is 19 % less on average than the hardware offloading.

Figures 3.9 and 3.10 show that CPU utilization of the receiver mostly coincides, both when checksum calculations are offloaded by VOSE and when that are performed by hardware. When operating over Gigabit Ethernet, both error rates averaged below 1%. Over 10 Gigabit Ethernet,

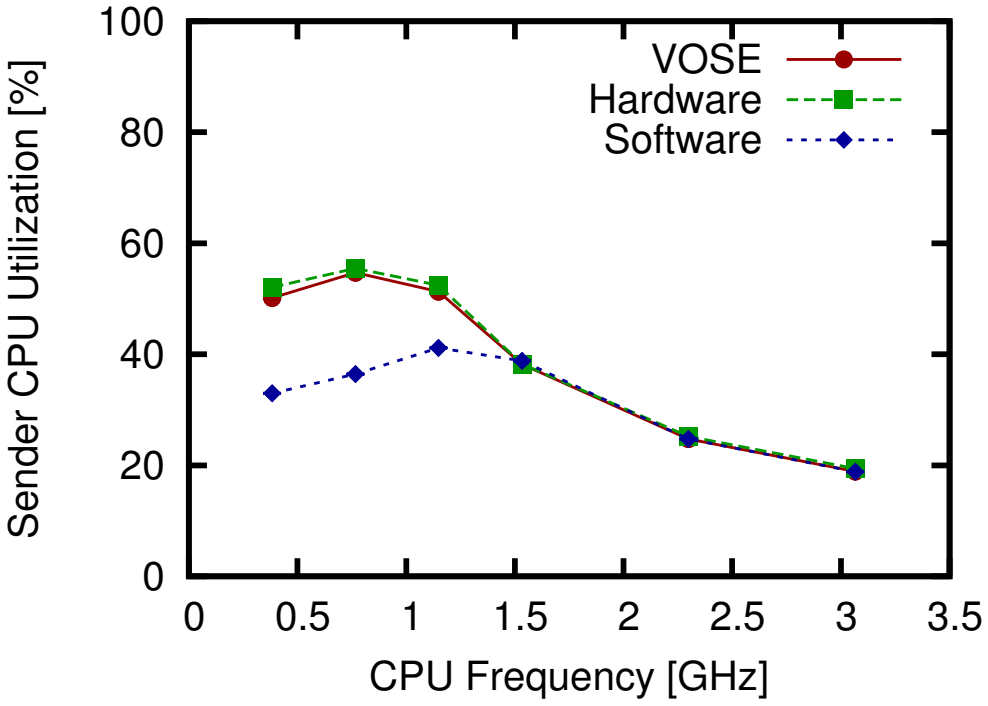



Figure 3.7: CPU utilization of the sender when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A)

both error rates averaged below 5%. These observations indicate that VOSE correctly emulates the offloading effect.

Over 10 Gigabit Ethernet, the sender CPU utilization when virtually offloading the TCP checksum calculations by VOSE is less than the case where hardware offloading is performed. In these experiments, TCP checksum calculations in the sender and receiver is simply bypassed in the virtual offloading in VOSE ($T_S = T_R = 0$). Hence, in the virtual offloading by VOSE, the call overhead of TOE produced in a hardware offloading is not contained. Hence, the sender CPU utilization in the case of the virtual offloading by VOSE serves as a small value from the value in the case of a hardware offloading.

To confirm the call overhead of TOE of the sender, we compared the profiling result of the sender when carrying out a hardware offloading with TOE in 3.2 GHz (see Table 3.1) with the result when carrying out a virtual offloading (see Table 3.2) using the profiler [74]. We describe only the processings whose rate in the whole exceeds 1 % among results. Results show that the

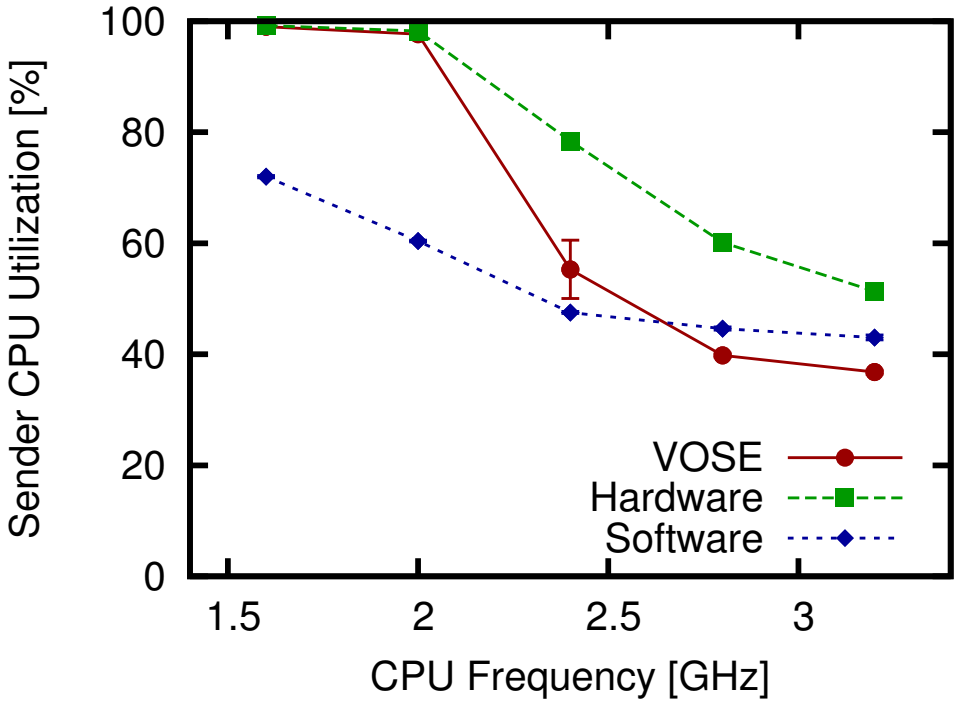



Figure 3.8: CPU utilization of the sender when changing the sender and receiver CPU operating frequency over 10 Gigabit Ethernet (experimental environment B)

total amount of processings, i.e., samples in Tables 3.1 and 3.2, of `ixgbe_clean_tx_irq()` and `ixgbe_xmit_frame()` decreases when a virtual offloading is carried out using VOSE. These functions are contained in an Intel 10GbE driver and are the transportation processing to hardware. From these observations, when carrying out a virtual offloading, we find that the load of these transportation processing to hardware is not contained in CPU utilization of the sender (see VOSE in Figure 3.8).

If required, we guess that a more exact emulation becomes possible by configuring the value of $T_S$ and $T_R$ appropriately according to the call overhead of TOE produced in a hardware offloading. In other words, we guess that a more exact emulation becomes possible by configuring $T_S$ and $T_R$ so that the amount of processings between VOSE and hardware may coincide.

On the contrary, if the processing time of TOE hardware is converged to 0, we expect that the performance with a TOE hardware offloading approaches the performance with a virtual offloading by VOSE. In other words, supposing a TOE designer creates TOE hardware with the processing

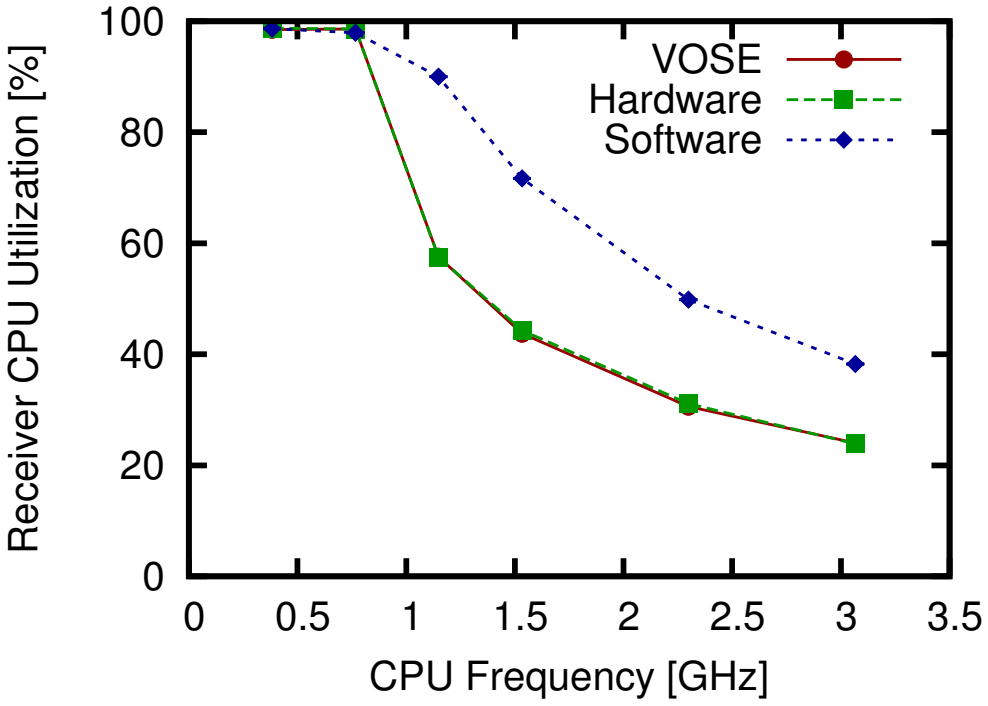



Figure 3.9: CPU utilization of the receiver when changing the sender and receiver CPU operating frequency over Gigabit Ethernet (experimental environment A)

speed near 0, we expect that throughput improves to the value of the throughput of VOSE in Figure 3.6. Similarly, we expect that CPU utilizations decrease to the value of the CPU utilizations of VOSE in Figures 3.8 and 3.10.

### 3.5.4 Observation

We next describe the offloading effect of performing TCP checksum calculations over Gigabit Ethernet (experimental environment A). First, we explain the offloading effects of TOE from the results of Section 5.3. Next, we apply VOSE to TCP checksum calculations, and estimate the performance improvement caused by introduction of TOE device.

First, from the results of Section 5.3, we compare the case where offloading TCP checksum calculations is carried out by hardware, the case where simulated offloading is carried out by VOSE, or the case where the processing is performed entirely by software.

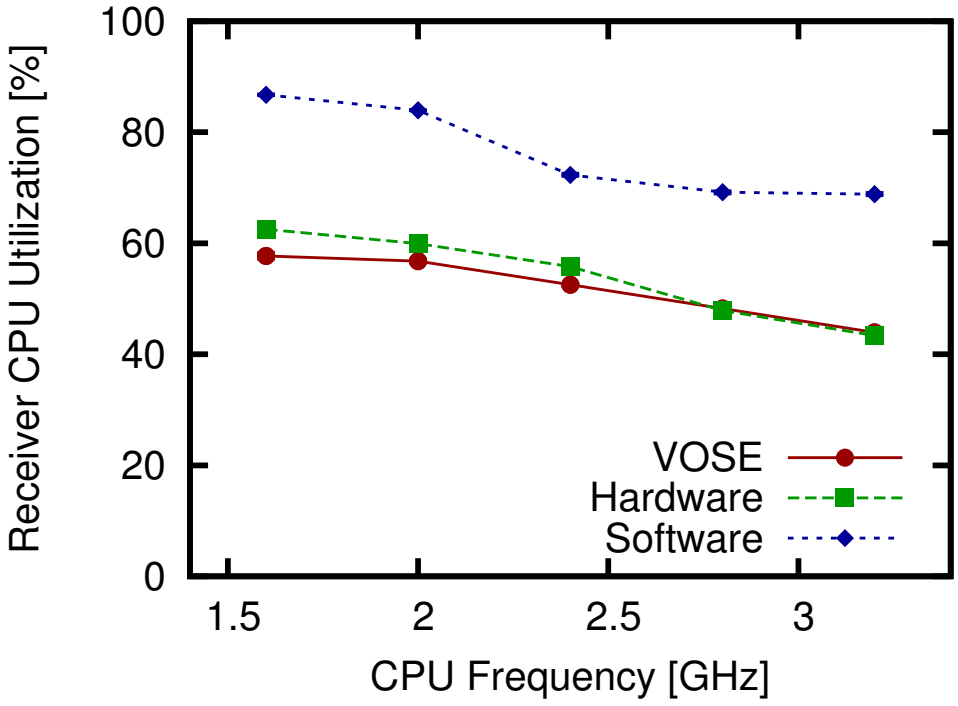



Figure 3.10: CPU utilization of the receiver when changing the sender and receiver CPU operating frequency in over 10 Gigabit Ethernet (experimental environment B)

Figure 3.5 shows that offloading the TCP checksum calculations is effective when the CPU frequency is low. Specifically, when the CPU frequency is under 1.5 GHz, throughput improves significantly (approximately 1.3 to 1.7 times) in Gigabit Ethernet by offloading the checksum calculations. When CPU frequencies exceed 1.5 GHz, however, throughput does not improve. These results indicate that offloading TCP checksum calculations is effective in Gigabit Ethernet when CPU frequency is less than 1.5 GHz.

Figure 3.9 shows that offloading TCP checksum calculations significantly reduces CPU loads at the receiver. For instance, when compared to the case where the CPU frequency is above 1 GHz and offloading is not performed, hardware offloading or virtual offloading reduces CPU utilization by approximately 35 to 45%. When the CPU frequency is 1.5 GHz or less, however, hardware offloading or virtual offloading increases CPU loads at the sender shown in Figure 3.7. The increase in such CPU loads originates in throughput having improved (see Figure 3.5). These results show that the receiver of the offloading of TCP checksum calculations is more effective than the sender.

Table 3.1: The profiling result of the sender when carrying out a hardware offloading with TOE in 3.2 GHz

| samples | % | symbol name |
|---|---|---|
| 171700 | 26.3 | `copy_from_user()` |
| 90286 | 13.8 | `ixgbe_clean_tx_irq()` |
| 70741 | 10.9 | `ixgbe_xmit_frame()` |
| 32196 | 4.9 | `kmem_cache_alloc()` |
| 31117 | 4.8 | `__kmalloc()` |
| 30769 | 4.7 | `kfree()` |
| 21116 | 3.2 | `kmem_cache_free()` |
| 19136 | 2.9 | `ixgbe_intr()` |
| 14830 | 2.3 | `cache_alloc_refill()` |

Table 3.2: The profiling result of the sender when carrying out a virtual offloading with VOSE in 3.2 GHz

| samples | % | symbol name |
|---|---|---|
| 169239 | 27.9 | `copy_from_user()` |
| 57922 | 9.5 | `ixgbe_clean_tx_irq()` |
| 56400 | 9.3 | `ixgbe_xmit_frame()` |
| 32694 | 5.4 | `__kmalloc()` |
| 31417 | 5.2 | `kmem_cache_alloc()` |
| 30547 | 5.0 | `kfree()` |
| 21537 | 3.5 | `kmem_cache_free()` |
| 18950 | 3.1 | `ixgbe_intr()` |
| 18055 | 3.0 | `cache_alloc_refill()` |

Moreover, as mentioned above, if CPU frequencies exceed 1.5 GHz, it is possible to use up the bandwidth even if offloading the TCP checksum calculations is not performed. However, the point of introducing TOE is not only increasing throughput, but also decreasing CPU load. Hence, even if CPU frequency is 1.5 GHz or more, hardware offloading of TCP checksum calculations is effective.

Next, with VOSE-enabled Linux kernel, we experimentally estimate the performance improvement caused by introduction of TOE device for TCP checksum calculations. Figure 3.11 shows the end-to-end performance, i.e., effective throughput, with TOE device in the experimental environment B, i.e., 10 Gigabit Ethernet, Core i7 3.2 [GHz] CPU, when the speedup factor of TOE devices is varied. The speedup factor is the ratio of the TOE processing speed to that of the software processing speed.

We calclates $T_{\mathrm{S}}$ and $T_{\mathrm{R}}$ as follows. In what follows, $T_{\mathrm{S}_{\mathrm{software}}}$ is the processing time of software in the sender, and $T_{\mathrm{R}_{\mathrm{software}}}$ is the processing time of software in the receiver.

$$T_{\mathrm{S}} = \frac{T_{\mathrm{S}_{\mathrm{software}}}}{\text{speedup factor}} \tag{3.1}$$

$$T_{\mathrm{R}} = \frac{T_{\mathrm{R}_{\mathrm{software}}}}{\text{speedup factor}} \tag{3.2}$$

For instance, when a speedup factor was 2, we set $T_{\mathrm{S}}$ and $T_{\mathrm{R}}$ to the half of the processing time of the software in the sender and receiver, respectively. The processing time of the software was measured by counting the average number of CPU clocks spent in `tcp_v4_send_check()` and

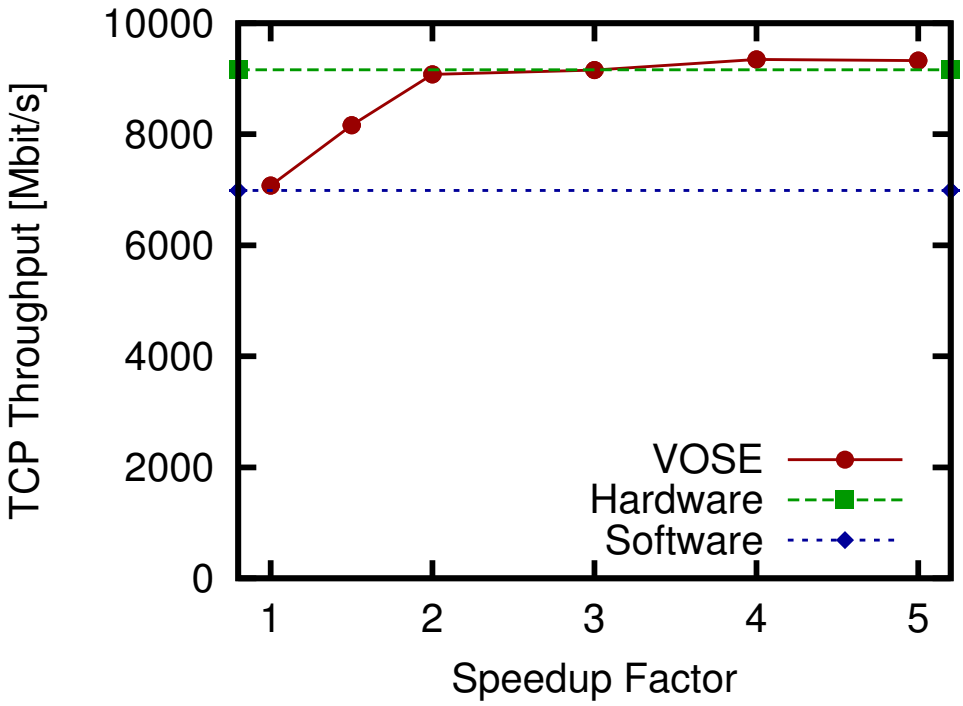



Figure 3.11: The end-to-end performance (effective throughput) when changing the time $T_S$ and $T_R$ of sleep processing

`tcp_v4_checksum_init()`. Specifically, RDTSC operations were embedded at the start and the end of `tcp_v4_send_check()` and `tcp_v4_checksum_init()`, and the difference in the number of CPU clocks were measured. For comparison purposes, the end-to-end performance with TOE (labeled as 'Hardware') and without TOE (labeled as 'Software') are also shown in the figure.

Figure 3.11 shows that throughput increases large when a speedup factor is set to 1.5–2. However, throughput hardly increases when the speedup factor is 3 or more. From these observations, we find that we should use TOE that finishes processing of TCP checksum calculations within the half of the processing time of the software to increase throughput sufficiently.

Consequently, VOSE enables TOE designers to know the overall performance improvements which are derived from introducing a TOE device in advance. Moreover, VOSE enables TOE designers to know in advance how much processing speed of TOE hardware is required for obtaining overall required performance. This leads to minimizing costs of design and minimizing development periods. For instance, TOE designers can elucidate bottlenecks of various TCP/IP processings

by trial and error and implement a TOE device that offloads the bottlenecks only. Moreover, TOE designers can test various patterns of TCP/IP offloads in a short period. Because the need for really implementing many TOE prototypes to satisfy performance requirements is eliminated, VOSE reduces the cost and periods in designing TOE devices.

## 3.6 Application of VOSE to IPsec Protocol

In this section, to demonstrate the effectiveness of VOSE, we apply VOSE to header authenticating and payload encryption in IPsec protocol, and estimate the performance improvement caused by introduction of several types of TOE devices.

### 3.6.1 Virtual offloading of IPsec protocol

IPsec is a protocol suite for realizing secure communication by authenticating and encrypting every IP packet [75–77]. IPsec is basically composed of three elements: the header authentication with AH (Authentication Header), the payload encryption with ESP (Encapsulating Security Payload), and the key exchange with IKE (Internet Key Exchange protocol).

IPsec supports variety of hashing and encryption algorithms for header authentication and payload encryption, respectively. It has been known that hashing and encryption algorithms are computing intensive. Hence, packet processing in the IPsec protocol easily becomes the performance bottleneck between end systems.

In what follows, we explain how header authentication and payload encryption of the IPsec protocol in the Linux kernel version 2.6.30 can be virtually offloaded with VOSE. Among several hashing and encryption algorithms implemented in the Linux kernel, we use rather modern algorithms: SHA-1 (Secure Hash Algorithm 1) [78] for header authentication in AH and payload encryption in ESP, and AES (Advanced Encryption Standard) with CBC (Ciphertext Block Chaining) [79] for payload encryption in ESP.

- SHA-1 virtual offloading

  Virtual offloading of header authentication and payload encryption with SHA-1 can be realized by replacing `sha1_update()` function with a short sleep of $T_{\mathrm{S_{SHA\text{-}1}}}$ and $T_{\mathrm{R_{SHA\text{-}1}}}$ in both the sender and the receiver, respectively.

- AES virtual offloading

  Virtual offloading of payload encryption with AES can be realized by replacing `aes_encrypt()` and `aes_decrypt()` functions with a short sleep of $T_{S_{AES}}$ and $T_{R_{AES}}$ in both the sender and the receiver, respectively.

- CBC virtual offloading

  Virtual offloading of payload encryption with CBC can be realized by replacing `crypto_cbc_encrypt()` and `crypto_cbc_decrypt()` functions with a short sleep of $T_{S_{CBC}}$ and $T_{R_{CBC}}$ in both the sender and the receiver, respectively.

### 3.6.2 Experiment

With VOSE-enabled Linux kernel, we experimentally estimate the performance improvement caused by introduction of several types of TOE devices for the IPsec protocol.

We compare performances of five types of TOE devices: AES-only, AES-CBC, SHA1-only, SHA1-AES, and SHA1-AES-CBC, each which corresponds to the case of AES-only offloading, AES and CBC offloading, SHA-1-only offloading, SHA-1 and AES offloading, and full offloading. Note that selection of protocol processings to be offloaded is a design choice of the TOE device. In general, offloading more protocol processings should result in better performance, but as well result in a more expensive TOE device. The crucial problem is to choose the best combination of protocol processings to be offloaded, under a given cost and hardware constraint and performance requirements. As we will see in the following experiments, VOSE is quite effective for estimating performance improvements caused by different types of TOE devices *without* implementing multiple types of TOE devices.

Figure 3.12 shows the end-to-end performance with different types of TOE devices in the experimental environment B, i.e., 10 Gigabit Ethernet, Core i7 3.2 [GHz] CPU, when the speedup factor of TOE devices is varied. Recall that the speedup factor is the ratio of the TOE processing speed to that of the software processing speed. Note that "inf" in the x-axis denotes the infinite speedup factor, i.e., $T_S = T_R = 0$. For comparison purposes, the end-to-end performance without TOE (labeled as 'Software') is also shown in the figure.

Figure 3.12 quantitatively show the effectiveness of TOE devices for the header authentication and the payload encryption of the IPsec protocol. This figure clearly indicates that SHA-1 is the

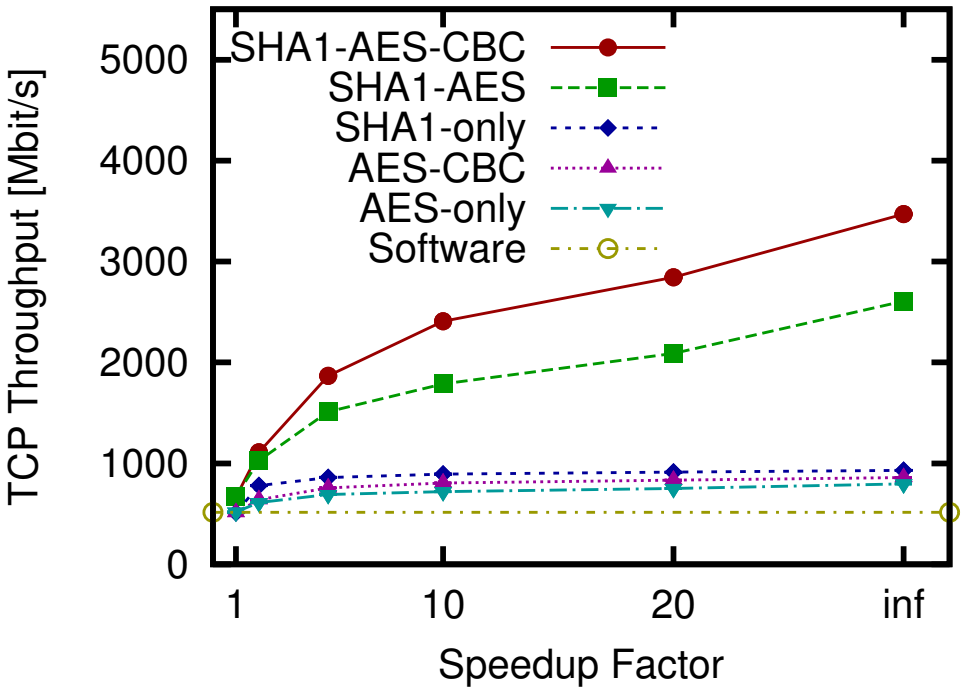



Figure 3.12: The end-to-end performance (effective throughput) of five types of TOE devices when changing the time of sleep processing

heaviest protocol processing among SHA-1, AES, and CBC, and that SHA-1 offloading is absolutely necessary to achieve more than 1 [Gbit/s] throughput. Interestingly, this figure also indicates that SHA-1 alone is not the performance bottleneck of end-to-end communication. For instance, the throughputs with SHA1-AES-CBC and SHA1-AES are high, but throughputs with other TOE devices, including SHA-1-only TOE device, are all less than 1 [Gbit/s]. This means that offloading the heaviest protocol processing alone is not sufficient for achieving high performance. Instead, we need to choose an appropriate combination of protocol processings to be offloaded to the TOE device. This demonstrates that VOSE is quite effective for quantitatively comparing the performances of different types of TOE devices *without* implementing real TOE devices.

Moreover, VOSE tells how fast the TOE device for every protocol processing should be. In Figure 3.12, the speedup factor is varied from 1 to infinity. This figure shows that increasing the processing speed of the hardwared does not always contribute to improve the end-to-end performance. For instance, the curve labeled 'SHA1-AES-CBC' always increases as the speedup factor

increases, which means that increasing the hardware processing speed contributes to the performance improvement when all of SHA-1, AES, and CBC are offloaded onto the TOE device. However, somewhat surprisingly, the curve labeled 'SHA-1-only' saturates when the speedup factor is around 10, and further increasing the speedup factor does not improve the end-to-end performance at all. In other words, providing very fast hardwared, i.e., more than ten times faster than the software processing, on the TOE device is simply worthless.

As we have observed, TOE design under several constraint and performance requirements are quite complex and difficult task. Such complexity and difficulty are resulted from non-linearity of the protocol offloading. Namely, the performance improvement caused by multiple-protocol offloading is hard to predict from performance improvements caused by individual-protocol offloadings. For instance, the performance improvement caused by SHA-1 and AES offloading (SHA1-AES in Figure 3.12) is hard to predict from performance improvements caused by individual SHA-1 and AES offloadings (SHA1-only and AES-only in Figure 3.12). Also, the performance improvement caused by increasing the speedup factor, i.e., providing a faster hardware, changes non-linearly and it is heavily dependent on the protocol processing to be offloaded. As we have demonstrated, VOSE dramatically reduces the burden of TOE designers. With VOSE, TOE designers can easily and flexibly examine the effectiveness of several combinations of TOE designs with experiments.

Consequently, VOSE enables TOE designers to measure performance of the various patterns of offloadings and clarify processings that should be performed with software and the processings that should be performed with hardware, without experiments with a real TOE implementation. Therefore, VOSE optimizes the TOE performance because TOE designers can design the TOE hardware that considered the appropriate balance between the scale of TOE hardware and processing speed.

## 3.7 Summary

In this chapter, we have proposed VOSE, which is a technique for measuring TCP/IP performance improvements derived from different type of TOE devices without implementing TOE devices really. VOSE enables virtual offloading without requiring a hardware TOE device by virtually emulating TOE processing, e.g., bypassing software protocol processing without violating protocol consistency, at both the source and destination end hosts.

The accuracy of virtual offloading with VOSE was thoroughly examined in terms of end-to-end

performance and CPU processing overhead. Specifically, we applied VOSE to the TCP checksum calculations in a Linux kernel, and have compared the performance and processing overhead between hardware offloading with a dedicated TOE and virtual offloading with VOSE. Consequently, we have shown that performance improvements which are derived from TOE devices can be estimated correctly.

Moreover, we have applied VOSE to header authenticating and payload encryption in IPsec protocol. We have estimated the performance improvement which are derived from several types of TOE devices of IPsec.

Areas for future study include the virtual offloading of protocol processing other than symmetrical processings and the effectiveness verification of VOSE using a multicore computer.

# Chapter 4

# Performance Comparison between Image and Event based Transfer Mechanisms for Remote Desktop Protocols

## 4.1 Introduction

Remote desktop services, which are one of cloud services, enable users to use a virtual machine that is running in a geographically distant data center as it would be located at hand. The services are becoming popular as both applications and data saved in data centers increase security that is one of the most stringent requirements to services in business sectors. Thus a number of remote desktop protocols are designed for remote desktop services and most of them are implemented as open source code or commercial one [5, 7, 20–23].

Remote desktop protocols are designed focusing on either how prompt screen updates should be or how prompt events such as key strokes should be. We call the two designed mechanisms *image transfer mechanism* and *event transfer mechanism*, respectively. In remote desktop protocols that employ image transfer mechanisms, when a server receives operation information, e.g., keyboard and mouse control information, from a client, the server updates its screen and sends back the updated screen information as images. At the client side, it displays received images on its screen.

On the contrary, in remote desktop protocols that employ event transfer mechanisms, when a server receives operation information from a client, the server sends back update events, e.g., window creations, settings of window positions and renderings of pictures or texts, to the client. At the client, it interprets received events and updates its screen accordingly. To simplify the notation, we simply refer to remote desktop protocols that employ image and event transfer mechanisms as *image transfer mechanisms* and *event transfer mechanisms*, respectively. As image transfer mechanisms, VNC (Virtual Network Computing) [5] and SPICE (Simple Protocol for Independent Computing Environments) [21] are implemented. As event transfer mechanisms, X Window System and its extensions, such as NX [22] and X2Go [23], are implemented.

The mechanisms designed based on different principles have their own complementary pros and cons. Since image transfer mechanisms run on a layer of frame buffers, remote desktop protocols that employ the image transfer mechanism runs independently of the window systems or applications running at server and client computers. However, image transfer mechanisms require high bandwidth for transferring screen image data. In contrast, event transfer mechanisms do not require high bandwidth since the size of update event is smaller than that of screen images. However, since event transfer mechanisms require both clients and servers to deal with update events, environments where remote desktop services with event transfer mechanisms can run are limited.

Currently, since users use remote desktop services in various environments, many remote desktop protocols employ image transfer mechanisms.

However, the reasons why they choose image transfer mechanisms are not justified in a quantitative way. At least, there are few studies quantitatively addressing this issue. In other words, quantitative evaluations of performance with respective transfer mechanisms under various network environments are not tackled sufficiently. For using the remote desktop services as cloud services, knowing their performance under a variety of network environments, i.e., various bandwidths, delays, and packet loss rates, is crucial issue. As the performance of remote desktop services, remote desktop users might be interested in the response time and clarity of screen. In the same way, service providers might be interested in transferred data size, required bandwidth and server loads. In particular, when users use remote desktop that employs each transfer mechanism under various network environments, it is important how they feel. Meanwhile, it is naturally expected that network characteristics are largely different by access means, e.g., accessing remote desktop services through the Internet or through intranets. Therefore, we elucidate that impacts of network

characteristics on Quality of Experience (QoE), which is how users feel, with respective transfer mechanisms.

We conjecture that QoE largely depends on transport-level performance since actual data transfer of remote desktop is realized by a transport layer protocol. In general, many remote desktop protocols employ Transmission Control Protocol (TCP) as a transport layer protocol for transferring operation information and updated screen information. An application-level performance of a remote desktop protocol depends on a transport-level performance. For instance, a time of transferring updated screen information may be large if a transfer speed is small, and arrival time of an operation information may be large if end-to-end delay is large. Therefore, we conjecture that these increases of time influence QoE. We refer to the former and latter transport-level performance as *TCP throughput* and *propagation delay*.

First, in this chapter, we focus on impacts of transport-level performance, i.e., TCP throughput and propagation delay, on QoE of respective transfer mechanisms. In our experiment, we use three major remote desktop protocols, i.e., VNC, SPICE and X2Go. We evaluate QoE of those protocols that employ each mechanism under various network environments and evaluate QoE of respective transfer mechanisms under various TCP throughputs and propagation delays. Second, we focus on a region that is competent to use respective transfer mechanisms under various network environments. We measure performance, i.e., a response time, transfer data size and data transfer rate, of the two transfer mechanisms under various network environments.

Our contributions are three-fold: 1) We elucidate a relation between QoE of those transfer mechanisms and transport-level performance under general wireless network environments. We show that image transfer mechanisms are sensitive to TCP throughput and event transfer mechanisms are sensitive to propagation delay. 2) We conclude that image transfer mechanisms are more suitable for remote desktop services under general wireless network environments than event transfer mechanisms because controlling TCP throughput is easier than controlling propagation delay. 3) We elucidate practicability of two transfer mechanisms under various network environments and conclude that image transfer mechanisms are feasible in general wireless environments such as Long Term Evolution (LTE).

This chapter is organized as follows. Section 4.2 summarizes related works. Section 4.3 introduces the respective transfer mechanism and three major remote desktop protocols used in our experiments. Section 4.4 describes the experiment design. Section 4.5 gives QoE evaluations of

each transfer mechanism under various TCP throughputs and propagation delays. Section 4.6 gives a performance evaluation of the remote desktop protocols under various network environments. Finally, Section 4.7 summarizes this chapter and discusses areas for future work.

## 4.2 Related Work

Several remote desktops such as an X window or thin clients have been evaluated [6, 53–56].

A number of evaluations have focused on protocols that employ an image transfer mechanism. Rhee *et al.* [53] shows an impact of network delay on a response time of a remote desktop protocol [20], which is used in Microsoft Terminal Service. The result shows that the response time is long when the network delay is high. Berryman *et al.* [54] evaluate an impact of a network delay and a loss rate on a transfer data size and a transfer speed. The result shows that a protocol employing TCP increases the transfer data size when delay or loss increases, and the result shows that a protocol employing UDP reduces the transfer speed when delay or loss increases.

Several protocols that employ an event transfer mechanism have been evaluated together with protocols that employ an image transfer mechanism. Yang *et al.* [6] evaluate an impact of a bandwidth on performance of six thin clients that employ one of respective transfer mechanisms. The result shows that X Window System and Sun Ray, which employ an event transfer mechanism, increases a response time at low bandwidth since the both protocols do not cancel updated screen information. Yang *et al.* [55] compare response time of four thin client environments, i.e., VNC, Citrix, Microsoft Terminal Service and Sun Ray. Note that VNC, Citrix and Microsoft Terminal Service employ an image transfer mechanism and Sun Ray employs an event transfer mechanism. Citrix and Microsoft Terminal Service provide good response time at low bandwidth since these protocols employ high-level encoding. However, these evaluations focus on application-level performance and do not focus on subjective performance.

Niraj *et al.* [56] evaluates subjective performance of VNC by a simulation. The result shows that subjective performance is impacted by network delay. This evaluation focuses on impact of network performance on protocols that employ an image transfer mechanism, but we conjecture that transport-level performance such as TCP throughput also influences subjective performance.

Little is known about impacts of differences between transfer mechanisms on performance. In particular, an important one is not application-level performance but subjective performance.

Moreover, a relation of transport-level performance and subjective performance is not evaluated sufficiently. Therefore, we elucidate that an impact of transport-level characteristics on QoE with respective transfer mechanisms.

## 4.3 Remote Desktop Protocols

Two types of remote desktop protocols categorized by their mechanisms for transferring updated screen information are proposed.

### 4.3.1 Image transfer mechanism

In *image transfer mechanisms*, updated screen information is transferred as screen images to a client when a screen is updated at a server.

In general, image transfer mechanisms are designed as independently of window systems or applications running at server and client computers. Image transfer mechanisms are implemented on a layer of a frame buffer since screen information is stored as images in the frame buffer at a server. In image transfer mechanisms, when the server receives operation information from a client, the server updates a screen and sends back updated screen information as images. At the client side, it displays received images on its screen.

An advantage of image transfer mechanisms is that we can use any window systems and any applications running on server and client computers. Currently, many remote desktop protocols employ image transfer mechanisms so that users can use these remote desktops in various environments. In contrast, a disadvantage of image transfer mechanisms is to require high bandwidth for transferring screen image data since those mechanisms transfer large screen image to a client from a server periodically. Developers devise to reduce amount of transferred data size. For instance, they compress screen images and transfer only an updated part instead of the whole screen.

### 4.3.2 Event transfer mechanism

In *event transfer mechanisms*, updated screen information is transferred as update events to a client when a screen is updated at a server. The update events are transferred to the client directly when the update events occur at the server. Update events mean various information such as window

creations, settings of window positions and renderings of pictures or texts. Note that the server transfers multiple events for updating the screen one time.

In general, event transfer mechanisms are designed so that a screen at a client reflects all changes of a server. Event transfer mechanisms are implemented in window systems. In event transfer mechanisms, when a server receives operation information from a client, update events occur in a window system at a server. Then, the server sends back update events to the client directly. At the client, its window system interprets received events, and updates its screen.

An advantage of event transfer mechanisms is that they do not require high bandwidth because they calculate the next screen image locally based on the received update events. In those mechanisms, update events are transferred instead of screen images such as pixel data and vector data. Since the size of update events is smaller than that of screen images, those mechanisms do not require high bandwidth. In contrast, disadvantages of event transfer mechanisms are that we can use those mechanisms only on limited window systems and that an increase of delay largely impacts performance. Event transfer mechanisms require both a server and a client to deal with update events. Therefore, environments in which remote desktop services employing event transfer mechanisms can run are limited. In addition, to update a screen one time, a server that employs an image transfer mechanism just transfers one image, but a server that employs an event transfer mechanism needs to transfer multiple events. Therefore, an increase of delay more largely impacts performance in event transfer mechanisms than image transfer mechanisms.

### 4.3.3 Examples of remote desktop protocols

In this section, we briefly explain three remote desktop protocols that are used in our experiment.

#### VNC (Virtual Network Computing)

VNC is one of basic remote desktop protocols that employ an image transfer mechanism. VNC is a simple protocol for remote controlling a machine via a network and was developed by AT&T Laboratories Cambridge [5].

Various versions of VNC have been proposed and developed. In general, a VNC client runs as an application on a machine connected to a screen. A VNC server runs as an application or as a part of a hypervisor, which is a software for controlling a virtual machine. A VNC server transfers

updated screen information as pixel data to a VNC client, and then the VNC client displays received pixel data on the screen.

In order to reduce a response time under low bandwidth and high delay, developers reduce transfer data size by compressing of screen images and/or transferring an updated part instead of whole screen [54, 80–83].

**SPICE (Simple Protocol for Independent Computing Environments)**

SPICE is one of new remote desktop protocols that employ an image transfer mechanism. SPICE is a high level protocol to control a virtual machine via a network and was developed by Qumranet [21].

A SPICE client runs as an application on a user's machine, and a SPICE server runs as a part of QEMU, which is one of hypervisors. A SPICE server transfers updated screen information as SPICE commands, e.g., pixel data, vector data, and character data, to a SPICE client. Then, the SPICE client displays those SPICE commands on the screen.

A SPICE client and a server transfer a certain amount of control/screen information via each independent TCP connections [21]. Moreover, SPICE measures a network speed, and then adjusts an update frequency and transfer data size according to the network speed.

**X2Go**

X2Go is one of new remote desktop protocols that employ an event transfer mechanism. X2Go is a simple protocol based on X Window System and was developed by X2Go project [23]. X Window System is originally a protocol of network penetration. We can display a screen of X applications that run on a server to an X display of a client, i.e., a screen, by using X Window System. X2Go is a protocol of X Window System fundamentally, and compresses X events for running via low bandwidth.

An X2Go server runs on an operating system of a virtual machine if user wants to operate desktops of a virtual machine. An X2Go server transfers update events of X Window System to an X2Go client. Then, the X2Go client that runs as X server interprets those events and updates the screen.

For transferring data through global network, X2Go compresses data of X events and encrypts the data. Therefore, X2Go safely run under low bandwidth. Moreover, in general, X2Go can

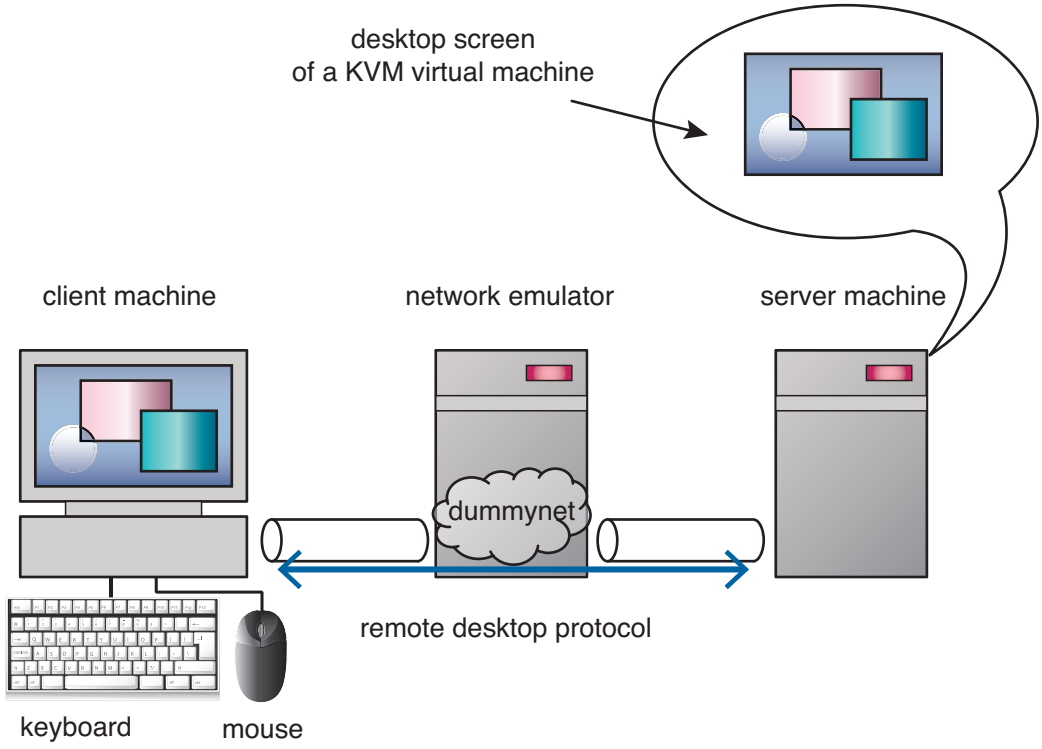



Figure 4.1: Experiment environment

restrain required bandwidth than VNC and SPICE since size of X events is smaller than that of pixel/vector data.

## 4.4 Experiment Design

In this experiment, we elucidate impacts of transport and network level performance on performance and QoE of image and event transfer mechanisms. Through experiments, we also discuss the practicability of remote desktop protocols with each transfer mechanisms.

### 4.4.1 Experiment Environment

Because we conjecture that the performance of image transfer mechanisms and event transfer mechanisms shows different tendency, we evaluate three major remote desktop protocols that employ

each transfer mechanisms. More specifically, we use VNC and SPICE as image transfer mechanisms, and use X2Go as an event transfer mechanism.

Our experimental network consists of a remote desktop client, server, and a network with a network emulator that acts as a router, as shown in Figure 4.1. For VNC and SPICE, we use Kernel-based Virtual Machine (KVM) at the server side. VNC and SPICE servers run as a part of the KVM hypervisor. An X2Go server runs as a software on a virtual machine that runs on the KVM hypervisor since there is no implementations for X2Go running on a part of a KVM hypervisor. Clients of those three remote desktop protocols run as a software on the client machine. To emulate DC-CE networks, we use a network emulator, dummynet, and vary bandwidth, delay and loss rate between the remote desktop servers and clients. Since many DC-CE cloud applications are used through wireless networks, we evaluate performance in a network environment that assumed general wireless network environments.

To see impacts of transport-level performance, our evaluations use TCP cubic, which is widely used in current networks. Moreover, TCP cubic is known as one of high performance implementation of a transport layer protocol.

For server and client computers, we use computers with Intel Core i7 3.20 GHz processors, 12 Gbyte memory and a Realtek RTL8168c/8111c Gigabit Ethernet interface card. The client and server run on Fedora 15 with Linux kernel 2.6.41, and the virtual machine runs on Debian 6.0 with Linux kernel 2.6.32. The network emulator is dummynet [61] and it runs on FreeBSD 9.0-RELEASE. The KVM hypervisor is qemu-kvm 0.14.0. We used several remote desktop implementations: VNC server contained in the qemu-kvm 0.14.0, TightVNC client version 4.1.3 [83], SPICE version 0.10.1 [21] contained in the Fedora 15, X2Go version 3.0.99.8 [23].

To elucidate fundamental impacts of transfer mechanisms, we use the default settings of each remote desktop protocol. More specifically, for TightVNC, we set a compression level to 9 and a JPEG quality to 5. For SPICE, we set a compression method to auto_glz. For X2Go, we set a network quality to ADSL, a compression method to 16m-jpeg and a picture quality to 9. Namely, the tuning for each protocol is omitted. Note that results of our experiment are not a performance comparison of current remote desktop implementations but a performance comparison of image and event transfer mechanisms. We set a resolution of a screen to 800 $\times$ 600 pixels that is applicable maximum resolution among three remote desktop implementations.

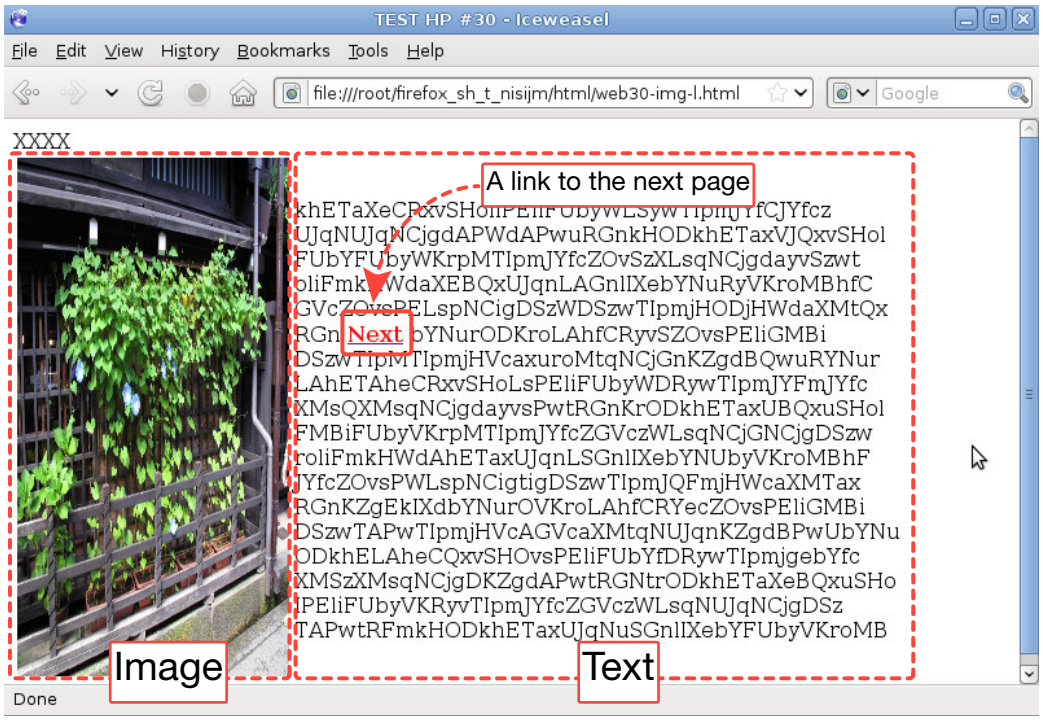



Figure 4.2: An example of the Web pages used for the experiments

### 4.4.2 Measurement Workload

To elucidate impacts of transport and network level performance on performance and QoE, we measure performance when a screen changes largely. If only a small part of screen changes during experiments, the amount of transferred data might be underestimated compared to actual remote desktop usage. This results in wrong observations for impacts of network environments on performance of remote desktop protocols. Therefore, impacts of transport-level performance becomes unclear. In addition, in remote desktop services, users use various applications that cause large screen changes, e.g., a presentation software, Web browser, and media player. Therefore, we measure performance when a screen is largely changed.

For changing a screen variously, we create a screen where images and texts coexist by several ratios and display those screens one by one. Specifically, we create several Web pages where images and texts coexist by several ratios (see Figure 4.2). Those Web pages include a hyperlink to a next Web page in a random position. By following that hyperlink, we display those Web pages on a web browser. We place those pages on the server machine directly to neglect an access time to a Web

server. Note that our experiment results do not mean only a result of a Web browsing but mean a result of general applications that change a screen largely.

## 4.5 Subjectivity Evaluation

In this section, we present Quality of Experience (QoE) that is felt by users when they use remote desktop protocols that employ respective transfer mechanisms under various network environments.

By a subjective evaluation that evaluators experience a remote desktop under various network environments really, we directly measure QoE with a questionnaire. Two methods to evaluate QoE have been proposed; an evaluation that evaluators really experience, and an evaluation using models of relations between application performance and subjective performance. However, relations between application performance and subjective performance of a remote desktop are not clear sufficiently.

### 4.5.1 Evaluation Method

Several evaluation methods for subjective evaluations have been proposed in ITU-T P.910 [84]. For instance, in Absolute Category Rating (ACR) method, which is basic evaluation methods to measure QoE, evaluators experience an evaluation object and then they assess that absolute QoE, i.e., Mean Opinion Score (MOS), on a five-grade scale. In Degradation Category Rating (DCR) method, evaluators experience original object at first, next they experience object that deteriorates, and they compare those objects. Then, they assess a QoE degradation, i.e., Degradation Mean Opinion Score (DMOS), on a five-grade scale. The DCR method is used to measure QoE that is sensitive to deterioration. In Comparison Category Rating (CCR) method, evaluators experience two evaluation objects. Evaluators experience original object and object to evaluate at random turn. Then, they assess a relative QoE, i.e., Comparison Mean Opinion Score (CMOS), on a seven-grade scale. The CCR method is used to measure QoE improvements. In Pair Comparison (PC) method, evaluators experience all pairs of evaluation objects, and they compare those objects. Then, they assess a relative QoE on a seven-grade scale. Since relative relation between the all targets is evaluated, the PC method enables accurately evaluating QoE but evaluation time becomes large.

In this thesis, we evaluate QoE on a five-grade impairment scale as with DCR method for elucidating impacts of using a remote desktop, i.e., transfering data through a network because DCR

Table 4.1: 5-grade impairment scale

| Score | Assessment |
|---|---|
| 5 | Imperceptible |
| 4 | Perceptible, but not annoying |
| 3 | Slightly annoying |
| 2 | Annoying |
| 1 | Very annoying |

method is sensitive to degradation. Moreover, impacts of using a remote desktop on subjective performance depend on various factors such as TCP performance, software processing of remote desktop, virtual machine processing and mental condition of evaluators. DCR method enables measuring impacts of TCP performance on subjective performance more correctly than other methods because it can neglect several factors such as virtual machine processing and mental condition of evaluators. Evaluators compare QoE of using a remote desktop and QoE of not using a remote desktop. They assess score of QoE degradation, i.e., DMOS, on a five-grade impairment scale as shown in Table 4.1.

In our experiments, evaluators evaluate QoE when 5 screens are displayed under each network environment. Namely, each evaluator waits until feeling that a screen is completely updated after he clicks a hyperlink included in a Web page. An evaluator repeats this process 5 times and assesses a score of each remote desktop protocol in each network environment. Evaluators are five students studying information engineering in our laboratory.

### 4.5.2 Simple Experiment

First, we assess four types of QoE, i.e., response time of a screen, response time of a mouse, response time of a keyboard, and clarity of an image, with respective remote desktop protocols. The respective QoE means an impairment that users feel about a waiting time for screen update, about a gap of a mouse moving, a gap of using a keyboard, and a clarity of a picture, respectively. We conjecture that these four qualities largely affect an experience quality. Namely, remote desktop user will strongly feel stress when renewal of a screen takes large time, when a mouse cursor does not follow, or when a keyboard input is not reflected expeditiously. Moreover, user may feel a

Table 4.2: The QoE of four types

| Type | Delay [ms] | Score | | |
|---|---|---|---|---|
| | | VNC | SPICE | X2Go |
| response time of a screen | 10 | 2.2 | 3 | 4 |
| | 100 | 2.4 | 3 | 2.4 |
| | 200 | 2.2 | 2.6 | 1.8 |
| respnse time of a mouse | 10 | 3.6 | 2 | 5 |
| | 100 | 1.8 | 1.2 | 4.6 |
| | 200 | 2.2 | 1.2 | 4.8 |
| response time of a keyboad | 10 | 4.6 | 4.2 | 5 |
| | 100 | 2.8 | 3.2 | 2.6 |
| | 200 | 2.6 | 2.6 | 2.6 |
| clarity of an image | 10 | 5 | 5 | 5 |
| | 100 | 4.8 | 4.8 | 4.4 |
| | 200 | 5 | 4.6 | 4.6 |

discomfort for a screen since remote desktop protocol compresses screen information.

We assess the score of remote desktop protocols by changing the bottleneck link delay, i.e., the delay at the network emulator, since we conjecture that a delay between a client and a server especially affects an experience quality. Table 4.2 shows scores when the delay of the network emulator was varied between 10, 100, 200 ms while the bandwidth of the network emulator was fixed at 1 Mbit/s.

As a whole tendency, Table 4.2 shows that three scores of response time decreases regardless of remote desktop protocols when a network delay increases.

The results mean that a response time of a screen of an event transfer mechanism is more easily influenced by a network delay than that of an image transfer mechanism. With VNC and SPICE, a response time of a screen decreases slightly when a network delay increases. Conversely, with X2Go, a response time of a screen decreases rapidly when a network delay increases. Note that both of VNC and SPICE employ an image transfer mechanism and X2Go employs an event transfer mechanism.

An increase of a network delay deteriorates an end-to-end performance and we conjecture that

the performance deterioration causes the QoE deterioration. An increase of network delay decreases a transfer speed of TCP sending data in a network really, and that increases an arrival interval of update information. We refer to the former and latter performance as *TCP throughput* and *propagation delay*. We conjecture that a response time of a screen is increased by TCP throughput degradation and propagation delay increase. We therefore confirm an impact of a TCP throughput degradation and a propagation delay increase on the QoE in following sections.

A response time degradation of a mouse is not caused by a transfer mechanism, but caused by a drawing mechanism of a mouse cursor. With VNC and SPICE, a mouse cursor is displayed at a server and server sends an image of a cursor to a client. With X2Go, a mouse cursor is displayed at a client without transferring data to a server. Therefore, with X2Go, a response time of a mouse is not influenced by a network delay increase. Therefore, for enabling user to comfortably utilize remote desktop services, we advocate that remote desktop implementation should display a mouse cursor at a client independently without transferring data to a server.

A response time of a keyboard decreases regardless of remote desktop protocols when a network delay increases. With three remote desktop protocols, key inputs are sent to a server and the server sends back a response to a client. Therefore, a response time of a keyboard is influenced rapidly by a network delay increase. Regardless of remote desktop protocols, we suppose that it is difficult to reduce a response time of a keyboard since we cannot control a network delay.

In our experiments, clarity of an image hardly decreases even if a network delay increases. Namely, we cannot feel an impairment about a picture. Therefore, we suppose that an impairment about a picture does not become a problem for remote desktop services.

### 4.5.3 Effect of TCP throughput

In undermentioned sections, we assess QoE about a response time of a screen when they use SPICE and X2Go. VNC and SPICE tend to show the similar result since both VNC and SPICE employ an image transfer mechanism. We therefore compare SPICE and X2Go.

We assess the score by changing TCP throughput, which depends on a network bandwidth and delay. Namely, we change a network bandwidth and delay simultaneously for changing TCP throughput and for not changing propagation delay. We confirm TCP throughput in each environment by using iperf [85]. Figure 4.3 shows the score when TCP throughput was varied between 1–5 Mbit/s while a one-way propagation delay was fixed at 10, 50 and 100 ms.

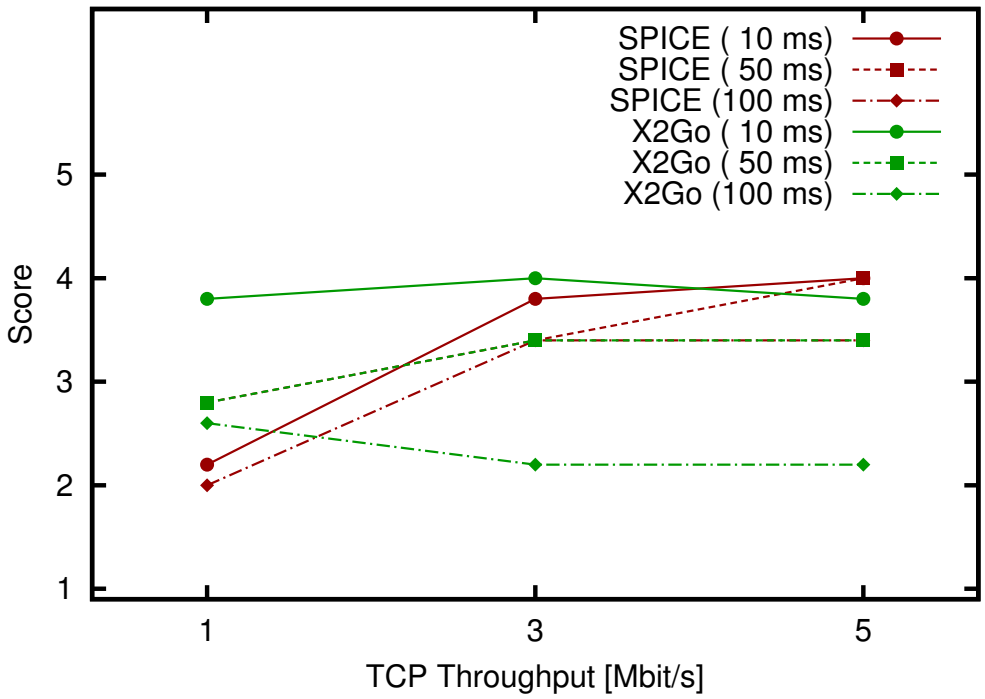



Figure 4.3: TCP throughput vs. QoE

Note that an event transfer mechanism cannot utilize TCP throughput fully. An event transfer mechanism transfers multiple small data as events to client and does not transfer data between each event. Therefore, with an event transfer mechanism, average transfer speed of application level is smaller than TCP throughput.

Figure 4.3 shows that the QoE of an image transfer mechanism is sensitive to TCP throughput. This QoE decreases regardless of propagation delay when TCP throughput decreases. This QoE is large regardless of propagation delay in particular when TCP throughput is more than 3 Mbit/s. Conversely, the QoE of an event transfer mechanism is almost constant regardless of TCP throughput.

We can interpret this result as a difference of the amount of transferred data. In general, a remote desktop server that employs an image transfer mechanism continuously transfers large data to a client since that transfers update information as one large image. Therefore, the transfer time is influenced by TCP throughput decrease, and this result causes the QoE deterioration. Conversely, a remote desktop server that employs an event transfer mechanism transfers multiple small data to a

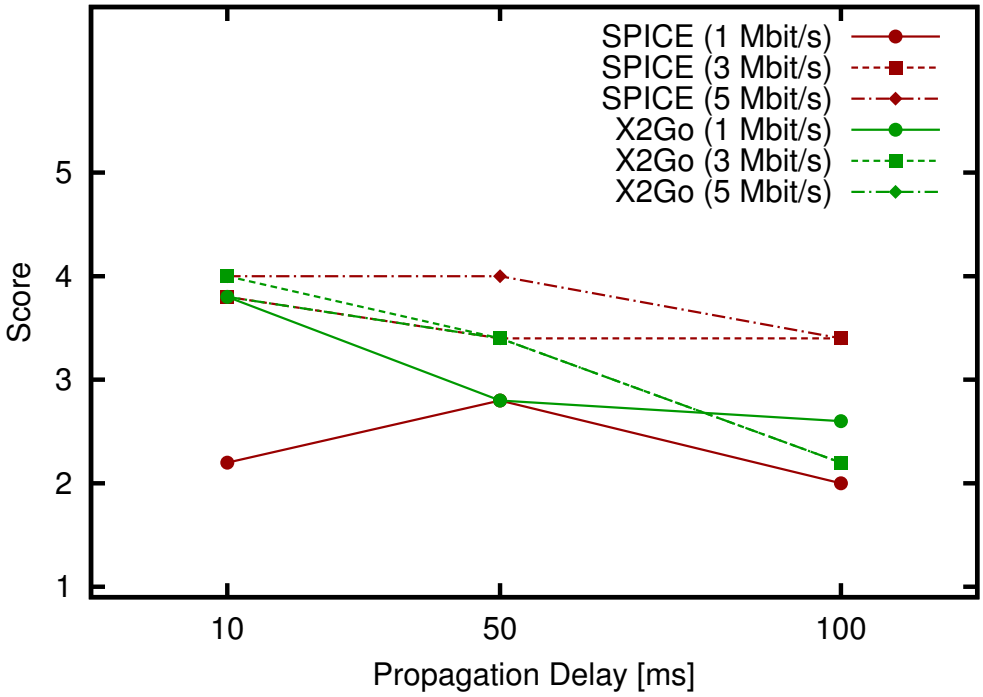



Figure 4.4: Propagation delay vs. QoE

client since that transfers update information as multiple events. Therefore, the transfer time is not influenced by TCP throughput decrease, and the QoE does not deteriorate.

### 4.5.4 Effect of propagation delay

We assess score of remote desktop protocols by changing propagation delay. Namely, we change a network bandwidth and delay simultaneously for changing propagation delay and for not changing TCP throughput. Figure 4.4 shows score when a one-way propagation delay was varied between 10–100 ms while a TCP throughput was fixed at 1, 3 and 5 Mbit/s.

Figure 4.4 shows that the QoE of an event transfer mechanism is sensitive to propagation delay. The QoE of an event transfer mechanism decreases rapidly when a propagation delay increases.

We can interpret this result as a relationship between a response time and a synchronicity of a transfer. We conjecture that a response time influences the QoE and that a synchronicity of a transfer also influences a response time. With an image transfer mechanism, a server can asynchronously transfer images to a client for updating a screen without a response of the client. Namely, this server

performs a bulk transfer. Therefore, the response time of an image transfer mechanism is hardly influenced by a propagation delay if TCP throughput is constant. Conversely, with an event transfer mechanism, a server transfers events to a client while synchronizing with the client. In general, for updating a screen one time, a server sends multiple update events to a client and the client sends back a response to the server per each event. The server cannot send next update event to the client until the server receives a response of the client. Therefore, the response time of an event transfer mechanism is largely influenced by a propagation delay increase regardless of TCP throughput.

### 4.5.5 Observations

We present that it is necessary to control a TCP throughput for comfortably using remote desktop protocols that employ an image transfer mechanism under general wireless network environments. Fortunately, we can easily control a TCP throughput since several improvements of TCP throughput have been proposed. Therefore, we suggest that an image transfer mechanism is appropriate for remote desktop service under wireless network environments.

On the contrary, we present that it is necessary to control a propagation delay for comfortably using remote desktop protocols that employ an event transfer mechanism under general wireless network environments. However, we can hardly decrease a propagation delay unlike a TCP throughput. Therefore, we suggest that an event transfer mechanism is not appropriate for remote desktop service under wireless network environments.

## 4.6 Quantitative Evaluation

In this section, we elucidate practicability of two types of transfer mechanisms that are employed by current remote desktop protocols under various network environments. With three major remote desktop protocols that employ respective transfer mechanisms, we focus on a screen update since the screen update is largely influenced by a difference of these mechanisms. Under various network environments, i.e., a bandwidth, delay, and loss rate, we quantitatively evaluate performance, i.e., a response time, transfer data size, and data transfer rate, of the screen update.

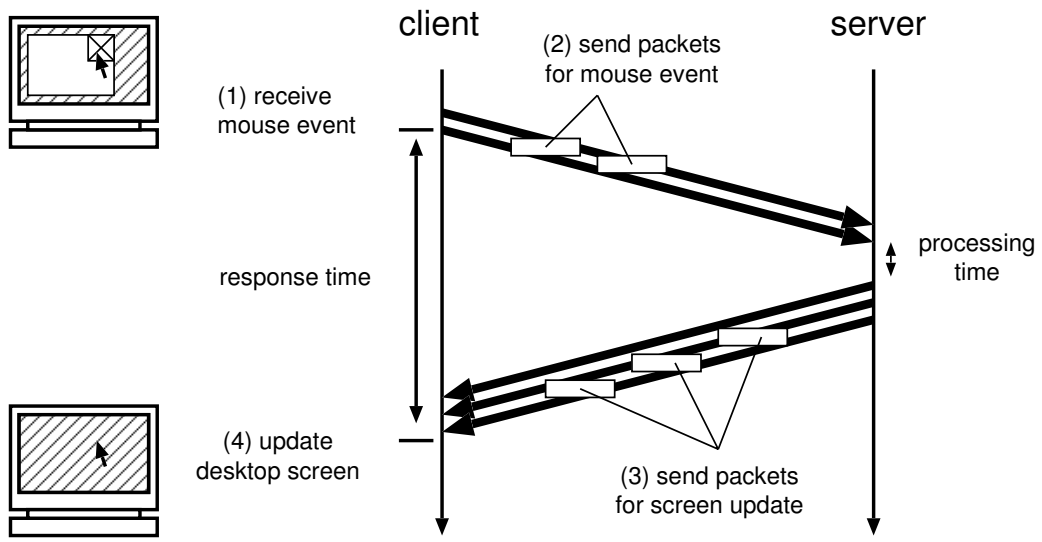



Figure 4.5: The flow from operation of a user to renewal of a screen

### 4.6.1 Measurement Method

As performance of remote desktop protocols, we evaluate a response time, transfer data size, and data transfer rate. We conduct ten experiments per each parameter and measure an average and 95% confidence interval of measurements.

To emulate a motion of a mouse in our subjectivity evaluation, we create a program that automatically operates a mouse on a client. The program generates X events of a mouse by using `XSendEvent()` function of X Window System. For measuring the performance per one update, we use fundamental idea of the slow-motion benchmarking [55]. Namely, we delay an interval from an operation for a renewal of a screen to a next operation, i.e., a generation of an X event that means a mouse click of a hyperlink.

Since VNC, SPICE and X2Go are different protocols, we measure the performance of those protocols by observing a transfer between a client and a server. Figure 4.5 shows a flow from operation of a user to renewal of a screen. When the user operates a keyboard or mouse, the client transfer operation information to the server. The server receives operation information from the client and sends back updated screen information to the client. Therefore, by observing the transfer between the client and the server, we can measure a data size and a time that were required for an update.

We measure the response time, transfer data size, and data transfer rate as follows. As the response time $T$, we subtract a start time of transferring operation information from an end time of transferring update information. As the start time, we record a time when the client sends out a final bit of operation information to the network. As the end time, we record a time when the server receives a final bit of update information from the network. As the transfer data size $X$, we measure the amount of accumulation data from the start time to the end time. As the data transfer rate ($= X/T$), we calculate an average transfer rate from the start time to the end time by dividing the transfer data size $X$ by the response time $T$.

### 4.6.2 Effect of network bandwidth

The performance of remote desktop protocols was measured by changing the bottleneck link bandwidth, i.e., the bandwidth throttling at the network emulator. Figure 4.6 shows the response time, transfer data size, and data transfer rate when the bandwidth of the network emulator was varied between 0.5–1,000 Mbit/s while the delay of the network emulator was fixed at 10 ms. Figure 4.6 shows that X2Go provides better response time than others when the bandwidth is some Mbit/s or less, and SPICE provides better response time than others when the bandwidth is more than 10 Mbit/s. The transfer data size and required data transfer rate of X2Go is always small than that of others. Moreover, we find that a ratio of the response time and the transfer data size is almost equal when the bandwidth is less than some Mbit/s by comparing the response time (see Figure 4.6(a)) with the transfer data size (see Figure 4.6(b)).

We can interpret this response time in the following way: The response time depends on TCP throughput and transfer data size when the delay is low. When the bandwidth is less than some Mbit/s, TCP throughput is limited less than some Mbit/s regardless of transfer mechanisms. Therefore, the response time is determined by the transfer data size in low bandwidth, and a mechanism not requiring large data transfer is appropriate in low bandwidth. Consequently, an event transfer mechanism is appropriate in low bandwidth such as some Mbit/s since small data is transferred in the mechanism.

Note that a peculiarity of both the transfer data size and data transfer rate changes largely with SPICE when the bandwidth is around 10 Mbit/s. SPICE adjusts an amount of screen information and an updating frequency of a screen according to a bandwidth by measuring an available bandwidth between a server and a client. A threshold of SPICE version 0.10.1 is fixed to 10 Mbit/s in

that source code.

### 4.6.3 Effect of network delay

In the following, we explain only the response time since impacts of network environments on the response time are larger than other measurement indexes. Note that a data transfer rate can be inferred from a response time in following experiments since a transfer data size is almost constant regardless of a delay and loss rate in our experiments.

The response time of remote desktop protocols was measured by changing the bottleneck link delay, i.e., the delay at the network emulator. Figure 4.7 shows the response time when the delay of the network emulator was varied between 10–200 ms while the bandwidth of the network emulator was fixed at 1 Mbit/s or 10 Mbit/s. Figure 4.6 shows that X2Go provides better response time than others when the bandwidth is 1 Mbit/s and the delay is 50 ms or less. Namely, that case is that both bandwidth and delay are small. In other cases, SPICE provides better response time than others. The response time increases according to an increase of the network delay regardless of protocols since TCP throughput decreases when the delay increases. However, we find that the response time of X2Go increases more largely than that of VNC and SPICE when a network delay increases. Namely, an event transfer mechanism is more sensitive to the delay than an image transfer mechanism similarly to our subjectivity evaluation.

We can interpret this response time in the following way: When the bandwidth or delay is large, the response time depends on the network delay, i.e., propagation delay, and the number of transfers that is for updating a screen one time. In image transfer mechanisms, a server can asynchronously transfer images to a client for a screen update. Namely, in these protocols, the server sends one bulk data to the client for updating a screen one time. Contrarily, in event transfer mechanisms, a server synchronizes with a client while transferring update events between the client and server. The server sends multiple update events to the client for updating a screen one time, and the client sends back a response to the server per each event. Since a waiting time for the response largely increases when the propagation delay is large, and the response time of an event transfer mechanism also increases largely. Consequently, an image transfer mechanism is appropriate in high delay such as 50 ms since the number of transfers is few in the mechanism.

### 4.6.4 Effect of network loss rate

The response time of remote desktop protocols was measured by changing the bottleneck link loss rate, i.e., the packet loss rate at the network emulator. Figure 4.8 shows the response time when the loss rate of the network emulator was varied between 0.01–10 % while the bandwidth and delay of the network emulator were fixed at 1 Mbit/s or 10 Mbit/s and 100 ms, respectively. Figure 4.8 shows that SPICE provides better response time than others when the loss rate is less then 3 % and X2Go provides better response time than others when the loss rate is more than 5 %.

We can interpret this response time in the following way: Since TCP throughput decreases according to an increase of the loss rate, the response time eventually depends on TCP throughput. Consequently, an image transfer mechanism is appropriate in low loss rate and an event transfer mechanism is appropriate in high loss rate.

### 4.6.5 Observations

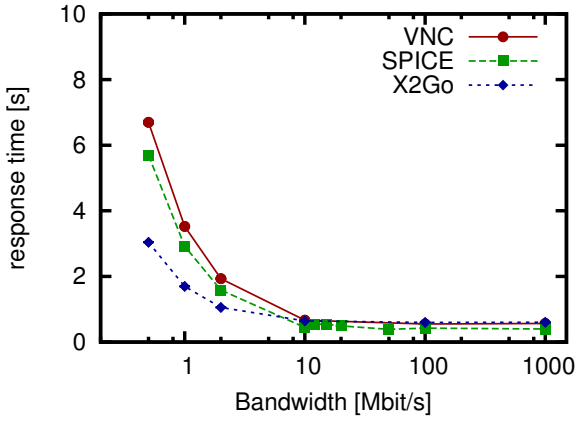
As remote desktop services, we quantitatively conclude that image transfer mechanisms are better than event transfer mechanisms in general wireless environments. Currently, many remote desktop services are utilized via wireless environments, e.g., LTE. In LTE, network characteristics are as follows: the bandwidth is around 10 Mbit/s, the delay is around 50 ms, and the loss rate is less than 1%. For image transfer mechanisms, these characteristics are enough to transfer data. Consequently, we conclude that image transfer mechanisms are better in general wireless networks since the mechanisms are not sensitive to the network delay.

## 4.7 Summary

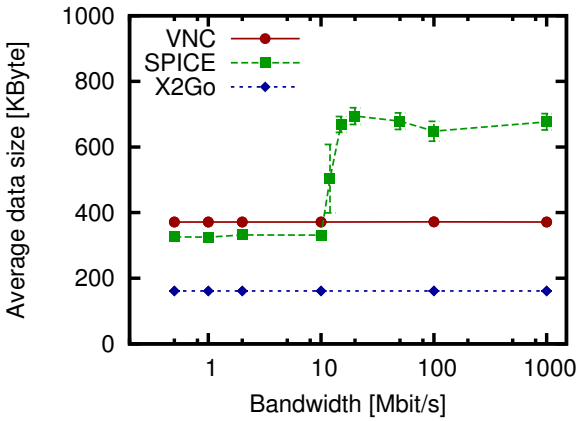
In this chapter, we focused on impacts of transport-level performance, i.e., TCP throughput and propagation delay, on QoE of two types of transfer mechanisms. We evaluated QoE of respective transfer mechanisms under various TCP throughputs and propagation delays that assumed general wireless network environments. Through subjective evaluation, we showed that image transfer mechanisms are sensitive to TCP throughput and event transfer mechanisms are sensitive to propagation delay. Consequently, we concluded that image transfer mechanisms are more suitable for remote desktop services under general wireless network environments than event transfer mechanisms because controlling TCP throughput is easier than controlling propagation delay.

Moreover, we measured performance of the two transfer mechanisms under various network environments. We concluded that image transfer mechanisms are feasible in general wireless environments such as LTE.

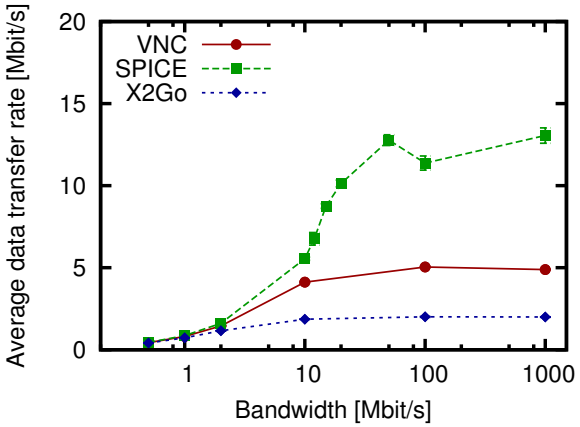
In the future work, we plan to propose a method for tuning parameters according to a network status, and we plan to propose a hybrid system that employs both image transfer mechanisms and event transfer mechanisms.




(a) response time

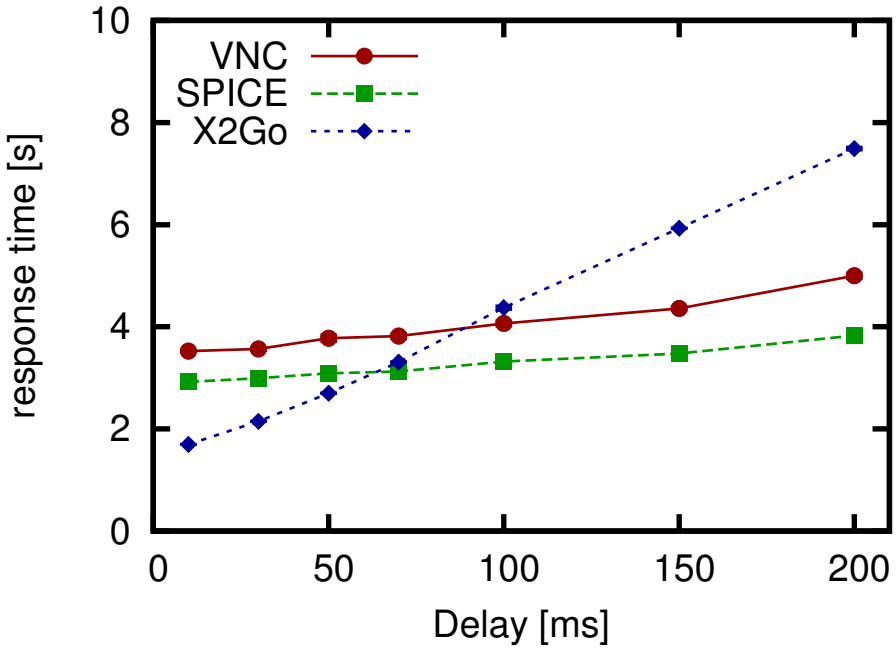



(b) transfered data size

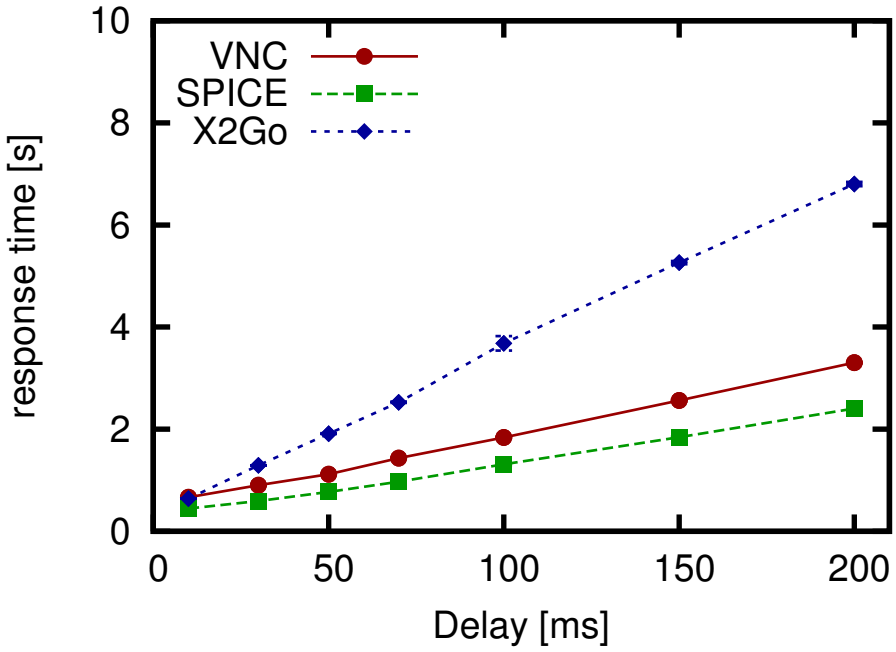



(c) data transfer rate

Figure 4.6: Link bandwidth vs. response time, transfer data size, and data transfer rate.

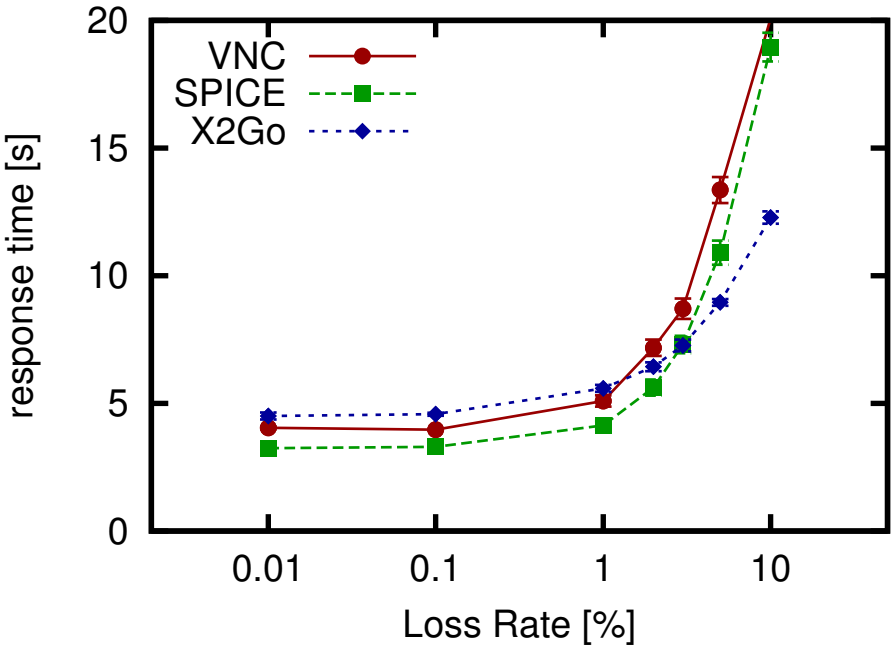



(a) 1 Mbit/s

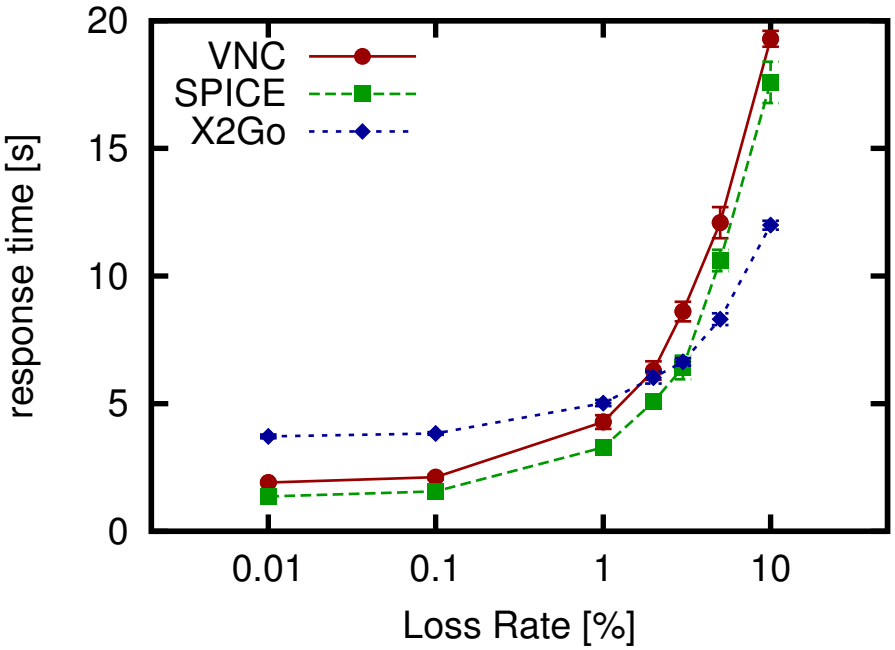



(b) 10 Mbit/s

Figure 4.7: Link delay vs. response time.




(a) 1 Mbit/s, 100 ms




(b) 10 Mbit/s, 100 ms

Figure 4.8: Link loss rate vs. response time.

# Chapter 5

# Conclusion

In this thesis, we have focusd on cloud applications the performances of which TCP dominantly determines, and we have improved performance of the applications by improving TCP performance. TCP performance issues are categorized by the number of TCP connections that derive issues; one or many connections. Because issues with one connection are always critical even if many connections are established, in this thesis, we have tackled issues that are derived from performance degradation of one connection. Two methods for improving TCP performance are categorized depending on whether TCP is modified. Because of installability into current environments, we have improved TCP performance with existing TCP, i.e., without modifying TCP.

This thesis contributes to realizing high quality DC-DC/DC-CE cloud applications that TCP is dominant, by focusing on TCP throughput improvements. We have categorized cloud applications into DC-DC cloud applications and DC-CE cloud applications, and have tackled three issues derived from TCP performance on respective applications. It takes different approaches to the both applications depending on their performance requirements, i.e., application-level performance and subjective performance, and network characteristics.

First, in Chapter 2, we have focused on a performance improvement with existing dedicated hardware devices. We have proposed *Block Device Layer with Automatic Parallelism Tuning (BDL-APT)*, a mechanism that maximizes the transfer speed of heterogeneous IP-SAN protocols in high delay networks. For improving IP-SAN performance, BDL-APT parallelizes data transfer using *multiple IP-SAN sessions* at *a block device layer* on an IP-SAN client and automatically optimizes the number of active IP-SAN sessions according to network status. We have proposed new layer

called block device layer, which receives read/write requests from an application or a file system and relays those requests to a storage device. BDL-APT parallelizes data transfer by dividing aggregated read/write requests into multiple chunks, then transferring a chunk of requests on every IP-SAN session in parallel. BDL-APT automatically optimizes the number of active IP-SAN sessions based on the monitored network status using our parallelism tuning mechanism, because it is known that the number of the connections that is not optimal deteriorates TCP throughput [46, 57]. We have evaluated the performance of BDL-APT through experiment using several IP-SAN protocols. Through our experiment, we show that BDL-APT realize high performance of heterogeneous IP-SAN protocols in various network environments.

Next, in Chapter 3, we have focused on a design of hardware devices for providing good performance. We have proposed *Virtual Offloading with Software Emulation (VOSE)*, which is a technique for measuring TCP/IP performance improvements derived from different type of TOE devices without implementing TOE prototypes really. VOSE enables *virtual offloading* without requiring a hardware TOE device by virtually emulating TOE processing on both source and destination end hosts. For demonstrating the effectiveness of VOSE, we have applied VOSE to the TCP checksum and IPsec protocol. We have extensively examined the accuracy of virtual offloading with VOSE, by comparing performance, i.e., end-to-end performance and CPU processing overhead, between VOSE and a dedicated TOE device. Moreover, we have estimated performance improvements that are derived from several TOE devices of IPsec and combinations of those devices, by applying VOSE to header authenticating and payload encryption in IPsec protocol. Consequently, we show that performance improvements which are derived from TOE devices can be estimated correctly.

Moreover, for providing good performance of DC-CE cloud applications that transfer data interactively in wireless network environments, we have focused on impacts of transport-level performance on subjective performance of remote desktops. For elucidating essential relation, we have categorized whole remote desktop protocols by their mechanisms for transferring updated screen information.

We have elucidated impacts of transport and network level performance on performance of respective transfer mechanisms. We have evaluated QoE of respective transfer mechanisms under various TCP throughputs and propagation delays that assumed general wireless network environments. Moreover, we have measured performance of the two transfer mechanisms under various

network environments. Through subjective evaluation, we have shown that image transfer mechanisms are sensitive to TCP throughput and event transfer mechanisms are sensitive to propagation delay. Consequently, we concluded that image transfer mechanisms are more suitable for remote desktop services under general wireless network environments than event transfer mechanisms because controlling TCP throughput is easier than controlling propagation delay. Furthermore, through measurement, we have concluded that image transfer mechanisms are feasible in general wireless environments.

The studies in this thesis mostly focused on static network environments such as an environment. Realizing adaptability to changes in DC-DC/DC-CE network environments is our future research direction. In the future, more and more applications would be aggregated to a data center. Since the many applications transfer various data independently, network status, e.g., the amount of background traffic, would change rapidly and largely compared with current DC-DC/DC-CE networks. For instance, our mechanism proposed in this thesis is designed for static network environments, therefore, we are planning to adapt our mecanisim to change in the amount of background traffic. Moreover, since this thesis shows that subjective performance is sensitive to throughput or delay, we conjecture that performance would be also correlated with jitter of throughput and delays. We are therefore planning to elucidate impacts of changes of transport-level performance on subjective performance.

# Bibliography


[1] Microsoft Corporation, "Windows Azure." http://www.windowsazure.com/.

[2] P. Mell and T. Grance, "The NIST definition of cloud computing." National Institute of Standards and Technology, Information Technology Laboratory, Version 15, Oct. 2009.

[3] R. P. King, N. Halim, H. G. Molina, and C. A. Polyzois, "Management of a remote backup copy for disaster recovery," *ACM Transactions on Database Systems (TODS)*, vol. 16, pp. 338–368, June 1991.

[4] W. Zheng, F. Wang, and Y. Y. Zhang, "A new backup model based on SAN system," in *Proceedings of The 8th International Conference on Computer Supported Cooperative Work in Design (CSCWD 2004)*, pp. 702–707, July 2003.

[5] T. Richardson, Q. S. Fraser, K. R. Wood, and A. Hopper, "Virtual network computing," *IEEE Internet Computing*, vol. 2, pp. 33–38, Jan. 1998.

[6] S. J. Yang, J. Nieh, M. Selsky, and N. Tiwari, "The performance of remote display mechanisms for thin-client computing," in *Proceedings of USENIX Annual Technical Conference*, pp. 131–146, June 2002.

[7] M. Abaza and D. Allenby, "The effect of machine virtualization on the environmental impact of desktop environments," *The Online Journal on Electronics and Electrical Engineering*, vol. 1, pp. 49–51, July 2009.

[8] T. Ercan, "Effective use of cloud computing in educational institutions," *Procedia Social and Behavioral Sciences*, vol. 2, pp. 938–942, Mar. 2010.

[9] Y. C. Chang *et al.*, "Understanding the performance of thin-client gaming," in *Proceedings of IEEE International Workshop Technical Committee on Communications Quality and Reliability (CQR 2011)*, pp. 1–6, May 2011.

[10] P. Sarkar and K. Voruganti, "IP storage: The challenge ahead," in *Proceedings of the 19th IEEE Symposium on Mass Storage Systems*, pp. 35–42, Apr. 2002.

[11] P. Wang, R. E. Gilligan, H. Green, and J. Raubitschek, "IP SAN – from iSCSI to IP-addressable ethernet disks," in *Proceedings of 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, pp. 189–193, Apr. 2003.

[12] H. Yang, "Fibre channel and IP SAN integration," in *Proceedings of the 12th NASA Goddard/21st IEEE Conference on Mass Storage Systems and Technologies (MSST 2004)*, pp. 101–115, Apr. 2004.

[13] S. Aiken, D. Grunwald, A. R. Pleszkun, and J. Willeke, "A performance analysis of the iSCSI protocol," in *Proceedings of the 20th IEEE/11th NASA Goddard Conference on Mass Storage Systems and Technologies (MSST 2003)*, pp. 123–134, Apr. 2003.

[14] P. T. Breuer, A. M. Lopez, and A. G. Ares, "The network block device," *Linux Journal*, vol. 73, May 2000.

[15] "GNBD project page." http://sourceware.org/cluster/gnbd/.

[16] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner, "Internet small computer systems interface (iSCSI)," *Request for Comments (RFC) 3720*, 2004.

[17] M. Rajagopal, E. Rodriguez, and R. Weber, "Fibre channel over TCP/IP (FCIP)," *Request for Comments (RFC) 3821*, July 2004.

[18] C. Monia *et al.*, "iFCP - a protocol for internet fibre channel storage networking," *Request for Comments (RFC) 4172*, Sept. 2005.

[19] I. Foster and C. Kesselman, *The Grid: Blueprint for a New Computing Infrastructure*. San Francisco: Morgan Kauffman, 1999.

[20] Microsoft Corporation, "White paper: RDP features and performance," June 2000. http://www.microsoft.com/.

[21] Red Hat, Inc., "SPICE," 2009. http://spice-space.org/home.html.

[22] F. Project, "FreeNX - free software (gpl) implementation of the NX server." http://freenx.berlios.de/.

[23] M. Gabriel *et al.*, "X2Go." http://www.x2go.org/doku.php/start.

[24] S. Floyd, "Highspeed TCP for large congestion windows," *Internet Draft* `draft-ietf-tsvwg-highspeed-01.txt`, Aug. 2003.

[25] C. Jin *et al.*, "FAST TCP: From theory to experiments," *IEEE Network*, vol. 19, pp. 4–11, Jan. 2005.

[26] J. Semke, J. Mahdavi, and M. Mathis, "Automatic TCP buffer tuning," in *Proceedings of ACM SIGCOMM '98*, vol. 28, Oct. 1998.

[27] T. Dunigan, M. Mathis, and B. Tierney, "A TCP tuning daemon," in *Proceedings of Super-Computing: High-Performance Networking and Computing*, Nov. 2002.

[28] B. K. Kancherla, G. M. Narayan, and K. Gopinath, "Performance evaluation of multiple TCP connections in iSCSI," in *Proceedings of the 24th IEEE Conference on Mass Storage Systems and Technologies*, pp. 239–244, Sept. 2007.

[29] L. Qiu, Y. Zhang, and S. Keshav, "On individual and aggregate TCP performance," in *Proceedings of International Conference on Network Protocols*, pp. 203–212, Oct. 1999.

[30] H. Sivakumar, S. Bailey, and R. L. Grossman, "PSockets: The case for application-level network striping for data intensive applications using high speed wide area networks," in *Proceedings of ACM/IEEE Conference on Supercomputing*, pp. 4–10, Nov. 2000.

[31] T. J. Hacker, B. D. Athey, and B. Noble, "The end-to-end performance effects of parallel TCP sockets on a lossy wide-area network," in *Proceedings of the 16th IEEE-CS/ACM International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 434–443, Apr. 2002.

[32] Y. Fukuju, K. Mera, and T. Ishihara, "Evalution of hardware-based IPsec processing method for embedded devices," *IEICE Technical Report*, vol. 107, pp. 79–84, Dec. 2007.

[33] W. Feng, P. Balaji, C. Baron, L. N. Bhuyan, and D. K. Panda, "Performance characterization of a 10-gigabit ethernet TOE," in *Proceedings of the 13th Symposium on High Performance Interconnects*, pp. 58–63, Aug. 2005.

[34] C. S. Ha, J. H. Lee, D. S. Leem, M. S. Park, and B. Y. Choi, "ASIC design of IPsec hardware accelerator for network security," in *Proceedings of IEEE Asia-Pacific Conference on Advanced System Integrated Circuits (AP-ASIC 2004)*, pp. 168–171, Aug. 2004.

[35] W. Benjamin and C. Patrick, "Network I/O acceleration in heterogeneous multicore processors," in *Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, pp. 9–14, Aug. 2006.

[36] D. W. Kim, W. O. Kwon, K. Park, and S. W. Kim, "Internet protocol engine in TCP/IP offloading engine," in *Proceedings of the 10th International Conference on Advanced Communication Technology(ICACT 2008)*, pp. 270–275, Feb. 2008.

[37] H. Jang, S. H. Chung, and S. C. Oh, "Implementation of a hybrid TCP/IP offload engine prototype," in *Proceedings of the 10th Asia-Pacific Computer Systems Architecture Conference(ACSAC 2005)*, pp. 464–477, Oct. 2005.

[38] H. Jang, S. H. Chung, and D. H. Yoo, "Design and implementation of a protocol offload engine for TCP/IP and remote direct memory access based on hardware/software coprocessing," *Microprocessors and Microsystems archive*, vol. 33, pp. 333–342, Aug. 2009.

[39] H. Jang, S. H. Chung, D. K. Kim, and Y. S. Lee, "An efficient architecture for a TCP offload engine based on hardware/software co-design," *Journal of Information Science and Engineering*, vol. 27, pp. 493–509, Mar. 2011.

[40] H. Ghadia, "Benefits of full TCP/IP offload (TOE) for NFS services," in *Proceedings of NFS Industry Conference*, Sept. 2003.

[41] R. Westrelin, N. Fugier, E. Nordmark, K. Kunze, and E. Lemoine, "Studying network protocol offload with emulation: approach and preliminary results," in *Proceedings of the 12th IEEE International Symposium on High Performance Interconnects*, pp. 84–90, Aug. 2004.

[42] W. Ng *et al.*, "Obtaining high performance for storage outsourcing," in *Proceedings of the First USENIX Conference on File and Storage Technologies*, pp. 145–158, Jan. 2002.

[43] Y. Lu and D. H. C. Du, "Performance study of iSCSI-based storage subsystems," *IEEE Communications Magazine*, vol. 41, pp. 76–82, Aug. 2003.

[44] Y. Lu, N. Farrukh, and D. H. C. Du, "Simulation study of iSCSI-based storage system," in *Proceedings of 12th NASA Goddard & 21st IEEE Conference of Mass Storage Systems and Technologies (MSST 2004)*, pp. 101–110, Apr. 2004.

[45] C. M. Gauger, M. Kohn, S. Gunreben, D. Sass, and S. G. Perez, "Modeling and performance evaluation of iSCSI storage area networks over TCP/IP-based MAN and WAN networks," in *Proceeding of the Second International conference on Broadband Networks (BROADNETS 2005)*, pp. 915–923, Oct. 2005.

[46] F. Inoue, H. Ohsaki, Y. Nomoto, and M. Imase, "On maximizing iSCSI throughput using multiple connections with automatic parallelism tuning," in *Proceedings of the 5th IEEE International Workshop on Storage Network Architecture and Parallel I/Os (SNAPI 2008)*, pp. 11–16, Sept. 2008.

[47] Q. K. Yang, "On performance of parallel iSCSI protocol for networked storage systems," in *Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA 2006)*, vol. 1, pp. 629–636, Apr. 2006.

[48] M. Oguchi, R. Higa, K. Matsubara, T. Okamawari, and S. Yamaguchi, "Performance improvement of iSCSI remote storage access," in *Proceedings of the Fourth International Conference on Ubiquitous Information Management and Communication (ICUIMC2010)*, pp. 232–338, Jan. 2010.

[49] "SafeXcel." http://safenet-inc.com/products/chips/index.asp.

[50] "IXP4XX." http://www.intel.com/design/network/products/npfamily/ixp4xx.htm.

[51] J. C. Mogul, "TCP offload is a dumb idea whose time has come," in *Proceedings of the Ninth Workshop on Hot Topics in Operating Systems*, May 2003.

[52] D. X. Wei, P. Cao, and S. H. Low, "Time for a TCP benchmark suite?," 2005.

[53] J. Rhee, A. Kochut, and K. Beaty, "DeskBench: Flexible virtual desktop benchmarking toolkit," in *Proceedings of IFIP/IEEE International Symposium on Integrated Network Management (IM 2009)*, pp. 622–629, June 2009.

[54] A. Berryman, P. Calyam, M. Honigford, and A. M. Lai, "VDBench: A benchmarking toolkit for thin-client based virtual desktop environments," in *Proceedings of IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom 2010)*, pp. 480–487, Dec. 2010.

[55] S. J. Yang, J. Nieh, and N. Novik, "Measuring thin-client performance using slow-motion benchmarking," *ACM Transactions on Computer Systems*, vol. 21, pp. 87–115, Feb. 2003.

[56] N. Tolia, D. G. Andersen, and M. Satyanarayanan, "Quantifying interactive user experience on thin clients," *IEEE Computer Society*, vol. 39, pp. 46–52, Mar. 2006.

[57] T. Ito, H. Ohsaki, and M. Imase, "On parameter tuning of data transfer protocol GridFTP for wide-area networks," *International Journal of Computational Science and Engineering (IJCSE)*, vol. 2, pp. 177–183, Oct. 2008.

[58] M. de Icaza, I. Molnar, and G. Oxman, "The linux RAID-1, 4, 5 code," *Linux Expo*, Apr. 1997.

[59] S. R. Soltis, T. M. Ruwart, and M. T. O'Keefe, "The global file system," in *Proceedings of the 5th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 319–342, Sept. 1996.

[60] M. D. Flouris and A. Bilas, "Violin: A framework for extensible block-level storage," in *Proceedings of the 22nd IEEE/13th NASA Goddard Conference on Mass Storage Systems and Technologies*, pp. 128–142, Apr. 2005.

[61] L. Rizzo, "Dummynet: a simple approach to the evaluation of network protocols," *ACM Computer Communication Review*, vol. 27, pp. 31–41, Jan. 1997.

[62] "Open iSCSI project." http://www.open-iscsi.org/.

[63] "The iSCSI enterprise target project." http://iscsitarget.sourceforge.net/.

[64] "proc(5) - linux manual page." http://www.kernel.org/doc/man-pages/online/pages/man5/proc.5.html.

[65] Z.-Z. Wu and H.-C. Chen, "Design and implementation of TCP/IP offload engine system over gigabit ethernet," in *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN 2006).*, pp. 245–250, Oct. 2006.

[66] J. Postel, "Transmission Control Protocol," *Request for Comments (RFC) 793*, Sept. 1981.

[67] S. Floyd, "A report on some recent developments in TCP congestion control," *IEEE Communications Magazine*, vol. 39, pp. 84–90, June 2001.

[68] A. Mark and F. Aaron, "On the effective evaluation of TCP," *ACM SIGCOMM Computer Communication Review*, vol. 29, pp. 59–70, Oct. 1999.

[69] D. D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Communications Magazine*, vol. 27, pp. 23–29, June 1989.

[70] N. Bierbaum, "MPI and embedded TCP/IP gigabit ethernet cluster computing," in *Proceedings of the 27th Annual IEEE Conference on Local Computer Networks*, pp. 733–734, Nov. 2002.

[71] A. P. Foong, T. R. Huff, H. H. Hum, J. P. Patwardhan, and G. J. Regnier, "TCP performance re-visited," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2003)*, pp. 70–79, Mar. 2003.

[72] R. Braden, D. Borman, and C. Partridge, "Computing the internet checksum," *ACM SIGCOMM Computer Communication Review*, vol. 19, pp. 86–94, Apr. 1989.

[73] J. Chase, A. Gallatin, and K. Yocum, "End-system optimisation for high-speed TCP," *IEEE Communications*, vol. 39, pp. 68–74, Apr. 2001.

[74] J. Levon, "Oprofile - a system profiler for linux." http://oprofile.sourceforge.net/.

[75] S. Kent and K. Seo, "Security architecture for the internet protocol," *Request for Comments (RFC) 4301*, Dec. 2005.

[76] S. Kent, "IP authentication header," *Request for Comments (RFC) 4302*, Dec. 2005.

[77] S. Kent, "IP encapsulating security payload (ESP)," *Request for Comments (RFC) 4303*, Dec. 2005.

[78] C. Madson and R. Glenn, "The use of HMAC-SHA-1-96 within ESP and AH," *Request for Comments (RFC) 2404*, Nov. 1998.

[79] S. Frankel, R. Glenn, and S. Kelly, "The AES-CBC cipher algorithm and its use with IPsec," *Request for Comments (RFC) 3602*, Sept. 2003.

[80] R. Ltd., "RealVNC." http://www.realvnc.com/.

[81] K. V. Kaplinsky, "VNC tight encoder-data compression for VNC," in *Proceedings of the 7th International Scientific and Practical Conference of Students, Post-graduates and Young Scientists on Modern Techniques and Technology 2001 (MTT 2001)*, pp. 155–157, Feb. 2001.

[82] U. Team, "UltraVNC." http://www.uvnc.com/index.php.

[83] K. V. Kaplinsky, "TightVNC: VNC-compatible free remote control / remote desktop software:," 2001. http://www.tightvnc.com/.

[84] ITU, "Subjective video quality assessment methods for multimedia applications." Recommendation P.910, Apr. 2008.

[85] J. Dugan *et al.*, "Iperf." http://sourceforge.net/projects/iperf/.