High Speed Total Order for SAN infrastructure

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by
Gregory Greenman

Supervised by
Prof. Danny Dolev

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel.
May, 2005

Acknowledgments

I would like to thank all the people whose effort and collaboration made this work possible. I would like to thank my advisor, Prof. Danny Dolev for his most helpful guidance, support and belief in this work; I am especially thankful to Ilya Shnayderman for his bright ideas, insights and enthusiasm; I am also very thankful to Dr. Tal Anker for sharing his wealth of knowledge and experience in networking and distributed systems.

Abstract

The performance of many middleware distributed systems may be limited by the number of transactions they are able to support per unit of time. In order to achieve fault tolerance and to boost system performance, active state machine replication is frequently used. It employs total ordering service to keep the state of replicas synchronized. In this work we present an architecture that enables drastic increase in the number of ordered transactions in a cluster, using off-the-shelf network equipment. Performance supporting nearly one million ordered transactions per second was achieved, which substantiates our claim.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

The main focus of distributed computing has traditionally been the design that enables distributed software systems to achieve a common goal. Since most of participating components in a distributed system are software modules, it was naturally to assume that the number of "transactions" a distributed system could generate and handle was limited mainly by the CPU resources.

A recent technological trend is the introduction of hardware elements in distributed systems. Implementation of parts of a distributed system in hardware immediately imposes performance requirements on the software parts of the system. An example of a system that combines hardware and software elements is a high capacity Storage Area Network. Such a system combines a cluster of PC's, Disk Controllers and switches that inter-connect and can benefit from high-speed total order.

This work shows how message *ordering* can be guaranteed in a distributed setting, along with a ten-fold increase in the number of "transactions" produced and processed. The proposed architecture uses off-the-shelf technology with minor software adaptations.

Message ordering is a fundamental building block in distributed systems. "Total Order" is one of the basic message delivery order guarantees, allowing distributed applications to use the state-machine replication model to achieve fault tolerance and data replication. Extensive analysis of algorithms providing total ordering of messages can be found in [1]. One of the most popular approaches to achieve total order of messages is by using a sequencer that assigns order to all messages invoked. This scheme, however, is limited by the capability of the sequencer to order messages, e.g., CPU power. The goal of the methodology presented in this work is to achieve a hardware-based sequencer while using standard off-the-shelf network components. The specific architecture proposed uses two commodity Ethernet switches. The switches are edge devices that support legacy layer 2 features, 802.1q VLANs and inter VLAN routing, which are connected via a Gigabit Ethernet link and a cluster of dual homed PCs (two NICs per PC) that are connected to both switches. One of the

switches functions as the *virtual sequencer* for the cluster. Since the commodity switch supports wirespeed on its Gigabit link, we can achieve near wirespeed traffic of a totally ordered stream of messages.

In this work, we elaborate on the specific architecture, its assumptions, and the adjustments made to the software of the PCs. The performance results presented show near wirespeed traffic of totally ordered messages is now a reality. Part of our approach is a highly efficient optimistic delivery technique which can be utilized in various environments such as replicated databases, as shown in [2].

1.2 Overview of the SAN technology

Storage Area Network (*SAN*) is the major area where message ordering is a considerable bottleneck. We start the discussion by presenting a short overview of SAN and several popular SAN architectures.

A SAN is a storage solution, designed to provide enormous amounts of mass storage to an enterprise, along with high speed, reliability, scalability and additional advanced services. There exist various ways to access storage. The most simple one is the embedded (or directly attached) storage, when the storage devices are connected directly to a server. Another solution is network attached storage (NAS), when the storage devices are attached directly to a LAN. While these technologies can meet the requirements of a small environment, they have several drawbacks for larger systems. The disadvantage of the first one is the difficulty to achieve scalability and reliability. The disadvantage of NAS is that the storage related traffic can degrade the overall performance of a LAN. For a client to access data, it should first access a server, which will send a request to a storage device. The reply from the storage device (along with the data itself) will be sent back to the server, which in turn will reply to the client. In SAN, the storage is placed behind the servers using a separate network. Figure 1 shows the basic SAN topology, which can be built from the off-the-shelf components. The servers are connected to the storage devices using a combination of hubs and switches. There are multiple paths among servers and disks in order to avoid single point of failure. The limitations of distance and number of ports can be overcame by cascading the network devices (hubs and switches). The most common communication protocols used in SAN are Fibre Channel and Ethernet, although systems utilizing InfiniBand start to emerge. The standard data transfer protocols, such as SCSI can be extended for SAN, there are standards for SCSI over Fiber Channel (FCP) and over Ethernet (iSCSI).

In order to take the most of the potential suggested by the SAN approach, more complex systems can be built. An example of such system is Storage Tank [3], which is a SAN-based distributed file system and storage control system. It provides data sharing, centralization of the management functions (backup, restore, snapshot, allocation), reliability and I/O performance comparable to local file systems. In addition, it provides *virtualization* layer, i.e. logical storage devices, when one such device can span over several physical devices. The

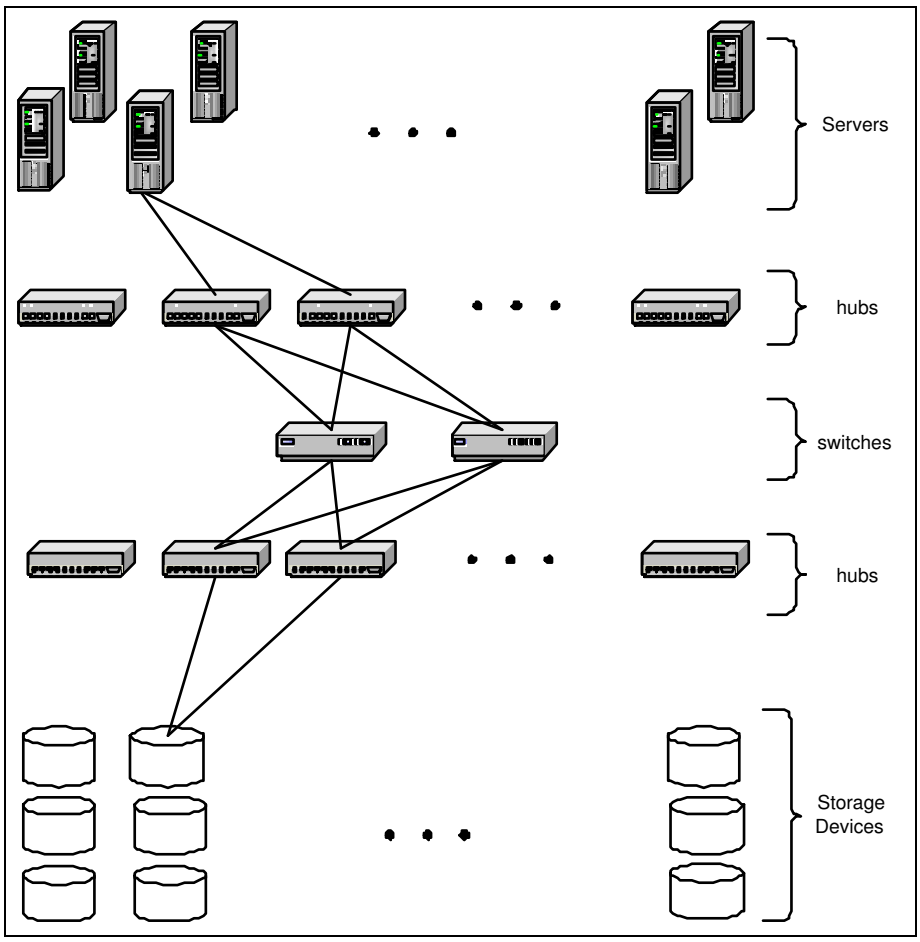

Figure 1: SAN General Architecture

architecture of Storage Tank is presented at Figure 2. The clients are the data consumers. The servers (on the right) are responsible for managing file system meta-data, such as file attributes, security information and data state locks. In addition, the servers provide load balancing and fail-over processing. There are two logical networks in the system. The first one is the control network, used for the communication between clients and the meta-data servers, it is a general purpose IP network. The second one is a high-speed SAN, used only for data transfer. One can note that in this system the data and meta-data are stored separately, the meta-data are never accessed directly by a client and should be accessible to all meta-servers.

There are a lot of challenges when designing such system, a few of them are listed next:

- An effective and scalable way to manage data locks should be found. The situation of failure or isolation of a client holding a lock should be considered;
- Efficient caching scheme for such distributed environment should be implemented. In addition to caching the data itself, the meta-data (such as locks) could be cached as

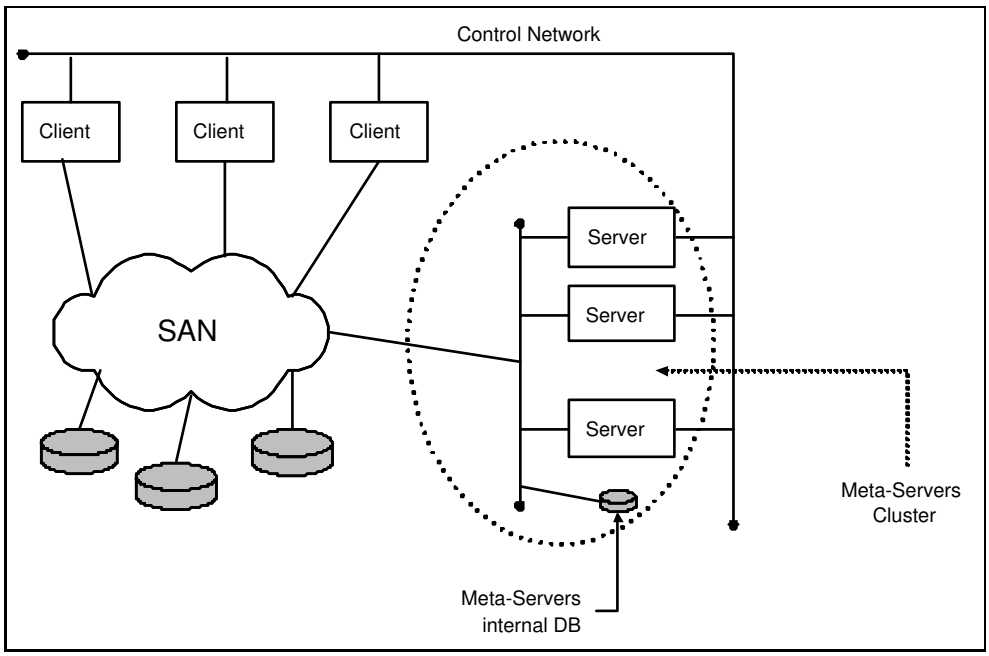

Figure 2: Storage Tank Architecture

well;

- An efficient way to authenticate clients should be suggested. This problem is particularly difficult in the environment described above, since clients can access storage devices directly and the computation power of the storage devices is relatively low;
- Another question is how to implement the system administration tasks (including backup and snapshot) without degrading the performance;

The architectures shown above are not the only way to implement SAN. In the following section yet another solution will be presented and an application that emerged as a result of a real problem encountered during development of a SAN device will be discussed.

1.3 Special SAN Implementation Review

One of the popular SAN architectures, shown at Figure 3, consists of powerful clients, such as mainframes, connected to storage devices (*disks*) via the network. Special switches implement the connection between clients and disks. Those switches, called SAN fabric, implement standard protocols for communication with storage devices. The protocols allow simultaneous disc access to the same block via different paths. One of the purposes for such redundant connectivity is to provide fault-tolerance and achieve better performance. The standard installation uses more than one switch in parallel to avoid a single point of failure.

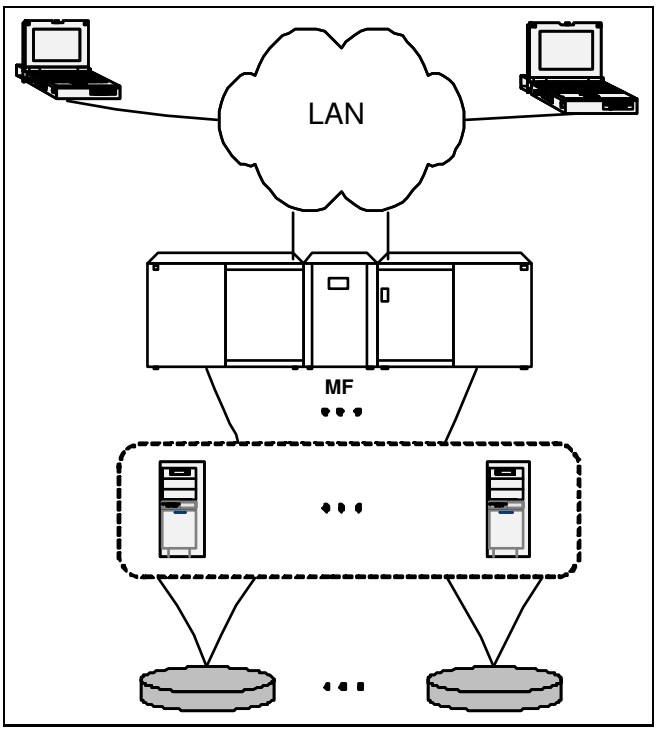

Figure 3: SAN with mainframe

There is a need for a middleware that allows, on the one hand, to enhance the SAN with advanced services, and, on the other hand, to support the old legacy protocols and to be transparent for both the mainframes and the disks. An example of such advanced service is a snapshot[1]. A common practice is to replace the SAN fabric with a cluster of PCs, thus implementing SAN services in software. Each PC in the cluster is equipped with a number of Host Based Adapters (*HBA*).

In order to avoid a single point of failure, each storage device (e.g., disk, RAID, JBOD) is connected to at least two PCs in the cluster. To implement the snapshot service, the PCs use a replicated state. For each block on the disks, the state can be *copied*, *uncopied* or *locked*. When a snapshot is started, all the blocks are marked as *uncopied*, and a room for a copy of each block is allocated on the disks. When a write request arrives at a PC, it checks the state of the block, and if it is *uncopied* the PC issues a "copy-on-write" command to the disk controller. So that two PCs will not send "copy-on-write" for the same block, the block state should be synchronized. An effective way of synchronization is to enforce an order on the requests that will guarantee that no two "copy-on-write" commands for the same block are executed simultaneously. Each PC sends a lock request when it is required to write on *uncopied* block. When a node receives lock request to a *uncopied* block b, the node changes the state of b to *locked*. The sender of the lock request also performs "copy-on-

[1]A snapshot is an instantaneous global picture of a system.

write" command. When the execution of the command is completed, the node sends unlock request. Each PC keeps the state of each block locally, and the state is updated only when the requests are delivered in the final order. The state changes according to the following state machine:

	lock		**unlock**		**snapshot**	
uncopied	$\rightarrow$	**locked**	$\rightarrow$	**copied**	$\rightarrow$	**uncopied**

However, total order by itself does not provide a solution, since a PC could crash during sending the "copy-on-write" command, making it impossible to distinguish between the cases when the block was/was not copied and overwritten. In order to overcome this difficulty, journal file system implemented by the disks can be used. When a node fails, it is possible to use the journal on the disk to know the last operation.

Chapter 2

Functional Specification

2.1 Model and Environment

The distributed setting is composed of a set of computing elements (PCs, CPU based controllers, etc.) residing on a LAN connected by switches. The computing elements, referred to as nodes, can either be transaction initiators (senders), or receivers, or both.

The nodes are connected via full-duplex links through commodity switches. We assume that the switches support IGMP snooping [4]. Support of traffic shaping is not mandatory, but is highly recommended. In addition, the switches can optionally support jumbo frames, IP-multicast routing and VLANs.

The communication links are reliable, with a minimum chance of packet loss. The main source of packet loss is buffer overflow rather than a link error. In Section 3.4.2, we discuss the fault tolerance issues. We assume that the participating group of nodes is already known. Dynamic group technology can be used to deal with changes in group membership, although this case is not considered in this work.

2.2 Problem Definition

The main goal of our study is to provide an efficient mechanism for total ordering of messages. The target is to drastically increase the number of messages that can be invoked and handled concurrently.

When comparing ordering algorithms, it is important to consider the guarantees provided by each algorithm. Most algorithms attempt to guarantee the order required by a replicated database application, namely, *Uniform Total Order (UTO)*. We present here the formal definition which appears in [5].

UTO is defined by the following primitives :

- **UTO1 - Uniform Agreement :** If a process (correct or not) has UTO-delivered(m), then every correct process eventually $UTO-delivers(m)$.

- **UTO2 - Termination :** If a correct process sends m, then every correct process eventually delivers m according to UTO.
- **UTO3 - Uniform Total Order :** Let m_1 and m_2 be two sent messages. It is important to note that $m_1 < m_2$ if and only if a node (correct or not) delivers m_1 before m_2. Total order ensures that the relation "$<$" is acyclic.
- **UTO4 - Integrity :** For any message m, every correct process delivers m at most once, and only if m was previously broadcasted.

In addition to the above definition, our system guarantees FIFO for each process.

- **FIFO Order :** If m_1 was sent before m_2 by the same process, then each process delivers m_1 before m_2.

Before a UTO is agreed on, a Preliminary Order (PO) is "proposed" by each of the processes. If the PO is identical for all correct (non-faulty) processes, it is called Total Order (TO). PO and TO should either be confirmed or changed by the UTO later.

Chapter 3

Implementation

As noted above, our implementation of Total Ordering follows the philosophy behind a sequencer-based ordering. However, we implement this sequencer using off-the-shelf hardware which is comprised of two Ethernet switches and two Network Interface Cards (NICs) per node. For simplicity of presentation, we assume that all the nodes are directly connected to the two switches. However, our algorithm can work in an arbitrary network topology as long as the topology maintains a simple constraint: all the paths between the set of NICs for TX and the set of NICs for RX share (intersect in) at least one link (see Section 4.7 for scalability discussion).

We assume that all the network components preserve FIFO order of messages. This implies that, once a packet gets queued in some device, it will be transmitted according to its FIFO order in the queue. It is noteworthy that if QoS is not enabled on a switch, the switch technology ensures that all frames, received on a network interface of the switch and egressing via the same arbitrary outgoing link, are transmitted in the order they had arrived; i.e., they preserve the FIFO property. We verified this assumption and found that most switches indeed comply with it, the reason being that the performance of TCP depends on it. Similarly to TCP, our algorithm makes use of this feature for performance optimization but does not require it for the algorithm correctness.

In our implementation, multicast is used in order to efficiently send messages to the nodes' group. Our goal is to cause all these messages to be received in the same order by the set of nodes that desire to get them (the receivers group). To achieve this, we dedicated a single link between the two switches on which the multicast traffic flows. Figure 4 shows the general network configuration of both the network (the switches) and the attached nodes. The methodology of the network is such that all the nodes transmit frames via a single NIC (TX NIC connected to the "left" switch in the figure) and receive multicast traffic only via the other NIC (RX NIC connected to the "right" switch in the figure). This ensures that received multicast traffic traverses the link between the switches. Since **all** multicast traffic traverses a single link, this ensures that all traffic is transmitted to the nodes in the same order via the second switch. As the switches and the links preserve the FIFO order, this in

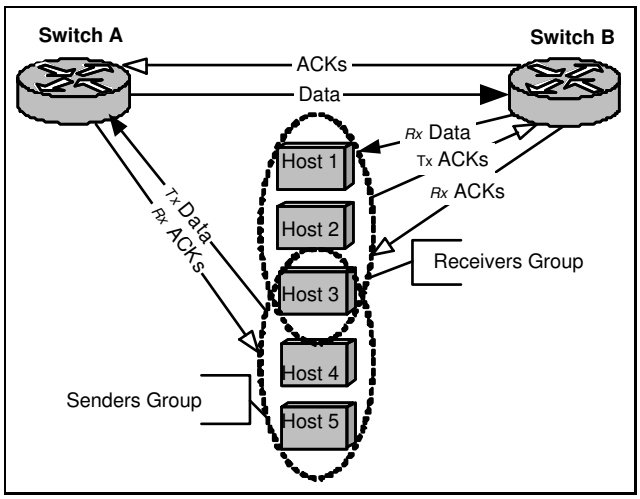

Figure 4: Architecture

turn implies that all the messages are received in the same order by all the nodes.

In a general network setting, there is a chance, albeit a small one, that a message omission may occur due to an error on the link or a buffer overflow (e.g. in the NIC, OS or in the switch). In a collision-free environment (like full-duplex switched environment), a link error is very rare. In addition, buffer overflow can be controlled using a flow control mechanism. Thus, the hardware mechanism enhanced with the proposed flow control (described in the next section), ensures, with high probability, the same order for all received messages. Ways to handle message omission when faults occur are discussed in Section 3.4.2.

3.1 Providing UTO

The preliminary ordering of the hardware configuration is not enough to ensure UTO because messages may get lost or nodes may fail. To address this issue, our protocol uses a simple positive acknowledgment (ACK) scheme for ensuring that the PO is identical at all the receivers. Each receiver node UTO-delivers a message to the application only after it has collected ACKs from each receiver node in the system. In order to reduce the number of circulating auxiliary control messages in the system, the ACKs are aggregated according to a configurable threshold parameter. If the system settings are such that each sender node is also a receiver, the ACK messages can be piggybacked on regular data messages.

For the sake of reliability, the sender node needs to hold messages for some period of time. This implies that sender nodes need to collect ACK messages, even though they do not deliver messages to the application. The ACK messages are used by a flow control mechanism (termed as *local* flow control in [6]) in order to maintain the transmission window. Each sender node is allowed to send the next data message only if the number of messages which were originated *locally* and are still unacknowledged by all the receiver nodes is less than

a defined threshold value (the transmission window size). Since the ACKs are aggregated, the number of messages that could be sent each time may vary. In order to increase the performance for small messages, a variation of a Nagle algorithm [7] is used as described in Section 3.2.1. Since the main source of message losses is buffer overflow, careful tuning of the flow control mechanism combined with ACKs aggregation can reduce the risk of losing messages. For our particular configuration, we identified the right combination of window size and number of aggregated ACKs to achieve maximum throughput. The specific implementation of the flow control mechanism presented in this work allows overall performance to converge to the receiving limit of the PCI bus.

3.2 Optimizations for Achieving High Performance

Various applications may be characterized by different message sizes and packet generation rates. For example, one application may be in a SAN environment in which it is reasonable to assume that the traffic can be characterized by a very large amount of small messages (where the messages will carry meta-data, i.e. a lock request). Another application can be a "Computer Supported Cooperative Work" (CSCW) CAD/CAM in which data messages may be large. In view of these modern applications, the need to achieve high performance is obvious. Below a description is presented of the mechanisms and techniques we have implemented and measured in order to reach that goal.

3.2.1 Packet Aggregation Algorithm

It was stated by [8] that at high loads, the message packing is the most influential factor for total ordering protocols. We use an approach similar to that in the Nagle algorithm [7], in order to cope with a large amount of small packets. Only the messages whose transmission is deferred by flow control are aggregated in buffers. The most reasonable size of each buffer is the size of an MTU. When the flow control mechanism shifts the sliding window by n messages, up to n "large" messages will be sent.

3.2.2 Jumbo frames

The standard frame size in Gigabit Ethernet is ~1512 bytes. The size of the jumbo frame is ~9000 bytes. Numerous studies show there is influence of the MTU size on the overall performance, such as [9], which reports increased performance for jumbo frames. The main reasons for the performance improvement include:

- lower number of interrupts (when moving the same amount of data) and
- less meta-data overhead (headers).

In order to fully benefit from the use of jumbo frames, all components of the system should be configured to support it; otherwise, fragmentation will occur. Since we control all the

components in the proposed system, we do not face this problem. Performance results prove that jumbo frames allow to obtain better throughput. For example, in the configuration of two senders and three receivers we achieve a maximum throughput of 722Mb/s.

3.3 Multicast Implementation Issues

As mentioned above, every node is dual-homed, i.e. is connected to the network with two NICs. In the IP multicast architecture, a packet accepted on some interface must be received on the same interface from which the node sends unicast traffic towards the source of the multicast packet. This condition is called the Reverse-Path-Forwarding (RPF) test, which is performed in order to detect and overcome transient multicast routing loops in the Internet. However, this poses a problem for our network settings, since we intend to receive the multicast traffic from the RX NIC while we are transmitting it from the TX NIC. There are several options for overcoming this difficulty, including:

- disabling the RPF test on the particular node;
- ensuring that the source address of the multicast packets has the same subnet portion as the NIC on which it is received (i.e., the RX NIC in our case).

We used the second approach and modified the RX flow in the NIC driver, so that it spoofs the source IP address of the packet. Another issue related to the usage of IP multicast in our settings is that self-delivery of multicast packet is usually done via *internal loopback*. Packets that are sent by the local host and are supposed to be received by it are usually delivered immediately by the operating system. We disabled this feature, so that ALL delivered packets are received via the RX NIC and thus all the packets pass through the same delivery process (so that total order is maintained).

3.4 Fault Tolerance and Failure Detection

Faults may occur at various levels of packet handling. Over the years, a variety of techniques have been proposed for building fault-tolerant systems. The techniques used in our implementation can currently handle some types of faults. Below, we discuss fault-tolerance techniques that are applicable to our system.

3.4.1 Failure Detectors

Failures can be caused by different sources: a switch failure, a physical link disconnection, a failure of a process and a crash of a node running the process. All these failures can be identified by failure detectors. The easiest event to reveal is a failure of a physical link or of a switch, which can be detected by the hardware. Network equipment sends a SNMP trap notifying about the failure and generated by software that operates network components.

For example, when a switch discovers that a link to its peer is down, it usually sends a SNMP message to a node. Upon receiving such a message, the node informs all the nodes about the configuration change.

A process crash failure is detected by the node's operating system. We propose to use TCP/IP connections to propagate this information to other nodes, using Congress [10] implementation. When a process fails, the operating system closes the process's connections on its node. The peers of its TCP/IP connections recognize this as an indication of the process crash and notify other nodes. To enhance the reliability of this mechanism, it is possible either to reduce the TCP KEEPALIVE timer or to issue heartbeat messages above the TCP connections, in order to facilitate a faster TCP failure detection. It is important to note that Congress maintains a tree of TCP/IP connections, but not a full mesh among the groups of nodes mentioned.

Those failure detector mechanisms, however, are still not robust enough: for instance, a SNMP trap message can be lost. A more reliable mechanism is an application-level "heartbeat" which usually works in connectionless mode and monitors the "liveness" of a set of peers. If the mechanism suspects that a monitored process has failed, e.g., when it does not send a heartbeat message for a long time, the node's failure is declared and the other nodes are notified.

3.4.2 Fault tolerance

Typically, leader-based message ordering systems like Isis [11] suggest how to handle faults. Our approach is compatible with these systems and can be used as the main module in their ordering protocols. When a failure is detected, the proposed system returns to a known software-based ordering scheme that is slower than our protocol. When the system is stabilized, our ordering scheme can be resumed. Below, we outline the main aspects of this transition.

For example, Isis implements *virtual synchrony* [12] approach that informally guarantees that processes moving from view v to a new view v' deliver the same set of messages in v. A recent work [13] proposes a way to implement virtual synchrony most of the time within one round. It relies on an existing membership service which delivers two kinds of messages, i.e. *start_membership_change* and *view*. A client-server architecture is suggested where the membership service is the server and the participating nodes are the clients. The work also suggests how to merge message dissemination service with the membership service in order to achieve virtual synchrony. In brief, when the membership service suspects a process, it sends *start_membership_change* notifications to all the processes, and they then reliably exchange information about their state. When the membership service converges to an agreed membership view, it sends the new view v' to the processes. The group members use this view v' and the state data received from other processes listed in v' in order to decide which set of messages should be delivered in the previous view v.

An alternative approach is Paxos [14]. Our virtual sequencer may serve as the leader

in Paxos. When a process receives message m from the virtual sequencer, it sends the *announce* message to all the processes. The *announce* message contains m's id and the corresponding PO number. When a process receives equal *announce* messages from the majority of processes, it sends *precommit* message. When the majority of *precommit* messages are collected and all the preceding messages are delivered, the process is able to deliver message m and send decision message to all processes, which resembles Paxos algorithm [14][1].

A similar approach to the above mentioned protocol was presented by Pedone et al. [15]. The authors define a weak ordering oracle as an oracle that orders messages that are broadcast, but is allowed to make mistakes (i.e., the broadcast messages might be delivered out of order). The paper shows that total-order broadcast can be achieved using a weak ordering oracle. The approach is based on the algorithm proposed in [16]. In [15] another algorithm is also proposed that solves total order broadcast in two communication steps, assuming $f < \frac{n}{3}$. This algorithm is based on the randomized consensus algorithm proposed in [17]. It should be noted that this solution requires collecting ACKs only from $n - f$ processes. Our virtual sequencer may serve as the weak ordering oracle for the algorithm proposed by Pedone et al. [15].

In our study, we implemented a loss-of-packet failure handling. The proposed algorithm contains a built-in method for identifying and retransmitting a missing packet by introducing a leader node whose order takes over when a conflict occurs. It is noteworthy that nodes in our system do not wait for the leader's ordering in failure-free scenarios. The leader's order is used only when a conflict occurs, thus our implementation follows the approach proposed in [18].

Another type of failure is a crash of a switch or disconnection of a link between switches. In order to solve the problem, we propose to increase the number of switches, to connect them in a mesh network and to enable each pair of switches to serve as virtual sequencer. The spanning-tree algorithm [19] is used to prevent loops. A dedicated IP multicast group is associated with each virtual sequencer. This solution allows to build a system with f+2 switches, where f is the maximum number of tolerated switch/link failures.

[1] While in Paxos there is a stage at which the leader collects the ACK messages from the majority of the processes, in our system it is enough to collect the majority of *precommit* messages only, since all the processes send the *precommit* messages in multicast to all group members.

Chapter 4

Performance Analysis

4.1 Overview

This section presents the results of the experiments performed to evaluate the architecture. The following configuration was used:

1. **Five end hosts**: Pentium-III/550MHz, with 256 Mb of RAM and 32 bit 33 MHz PCI bus. Each machine was equipped also with two Intel®Pro/1000MT Gigabit Desktop Network Adapters. The machines ran Debian GNU/Linux 2.4.25.

2. **Switches**: Two Dell PowerConnect 6024 switches, populated with Gigabit Ethernet interfaces. These switches are "store and forward" switches (i.e., a packet is transmitted on an egress port only after it is fully received).

The experiments were run on an isolated cluster of machines. For each sample point on the graphs below and for each value presented in the tables, the corresponding experiment was repeated over 40 times with about 1 million messages at each repetition. We present the average values with confidence intervals of 95%. Unless otherwise specified, the packet size in the experiments was about 1500 bytes (we also experimented with small packets and with jumbo frames). The throughput was computed at the receiver side as $\frac{packet\ size \times average\ number\ of\ delivered\ packets}{test\ time}$. In order to simulate an application, we generated a number of messages every configurable timeout. However, in most Operating Systems, and in particular in Linux 2.4, the accuracy of the timing system calls is not sufficient to induce the maximal load on the system. We therefore implemented a traffic generation scheme that sends as many messages as possible after each received ACK. Since the ACKs were aggregated, the size of the opened flow control window varied each time.

4.2 Theoretical bounds

It is important to observe that, regardless of the algorithm used to achieve the Total Order of messages, there are other system factors that limit the overall ordering performance. One of the bottlenecks that we encountered results from the PCI bus performance. In [20] it is shown that the throughput achieved by PCI bus in the direction from the memory to the NIC is about 892Mb/s for packets of 1512 bytes size and about 1 Gbit/s for jumbo frames. However, a serious downfall in the PCI bus performance was detected in the opposite direction, when transferring the data from the NIC to the memory. The throughput of 665Mb/s only for packets of 1512 bytes size and 923Mb/s for jumbo frames was achieved. Thus, the throughput allowed by PCI bus imposed an upper bound on the performance of a receiver node in our experiments. There are various studies on PCI bus performance, e.g. [21], which suggest several benchmarks and techniques for tuning. We chose not to focus on this issue, since the latest PCI bus technology (e.g., PCI-X) achieves much larger throughput. As will be shown later, we have nearly reached the theoretical and experimental upper bounds of the PCI bus.

4.3 Overview of Measurements

We first discuss the best throughput results obtained for each configuration. The latency obtained per result is presented as well. Two types of configurations were used: those where all the nodes were both senders and receivers (all-to-all configurations), and those in which the sets of senders and receivers were disjoint. It is important to note that for some configurations, such as the all-to-all configuration and the experiments with the jumbo-frames, we utilized the traffic shaping feature of the switching device, namely the one that is connected to the TX NICs. This ensured that no loss occurred on a node due to the PCI bus limitations. The main benefit of using traffic shaping is the limit it imposes on traffic bursts that were the major cause for packet drops in our experiments. Since the purpose of this series of the experiments was to obtain the best performance achievable, we tuned the system parameters (i.e., traffic shaping, flow control window size, etc.) for each configuration on an individual basis.

4.3.1 All-to-all Configurations

Results for all-to-all configurations and configurations with dedicated senders are discussed separately, since when a node serves as both a sender and a receiver, the CPU and PCI bus utilization patterns differ and the node is overloaded.

Table 1 presents throughput and latency measurements for all-to-all configurations along with the corresponding confidence intervals, shown in parentheses. The nodes generate traffic at the maximum rate bound by the flow control mechanism. Two different latency values are presented: PO Latency and UTO Latency. PO Latency is defined as the time that elapses

Nodes Number	Throughput *Mb/s*	PO Latency *ms*	UTO Latency *ms*
3	310.5 (0.08)	4.2 (0.03)	6.5 (0.03)
4	344.4 (0.04)	4.4 (0.02)	6.8 (0.02)
5	362.5 (0.09)	4.1 (0.02)	6.7 (0.02)

Table 1: Throughput and Latency for all-to-all configuration

between transmission a of message by a sender and its delivery by the network back to the sender. UTO Latency is defined as the time elapsed between a message transmission by a sender and the time the sender receives ACKs for this message from every receiver.

The number of the nodes that participated in this experiment increases from 3 to 5. As presented in Table 1, the achieved throughput increases with the number of participating nodes. This is accounted for the PCI bus behavior (See Section 4.2). Since each node both sends and receives data, the load on the PCI is high, and the limitation is the boundary of the total throughput that can go through the PCI bus. As the number of nodes grows, the amount of data each individual node can send decreases. When a node sends less data, the PCI bus enables it to receive more data. The nonlineariity of the increase in throughput in this experiment can be attributed to the above mentioned property of the PCI bus, where the throughput of transferring data from memory to NIC is higher than in the opposite direction.

4.3.2 Disjoint Groups of Senders and Receivers

Table 2 presents the performance results of throughput measurements for disjoint sets of nodes. We used 2-5 nodes for various combinations of groups of senders and receivers. The maximum throughput of ~512.7Mb/s was achieved. In the trivial configuration of a single sender and a single receiver, the result is close to the rate achieved by TCP and UDP benchmarks in a point-to-point configuration, where the throughput reaches 475Mb/s and 505Mb/s, respectively. The lowest result was registered for a single sender and four receivers, the achieved throughput of 467Mb/s not falling far from the best throughput.

For a fixed number of receivers, varying the number of senders yields nearly the same throughput results. For a fixed number of senders, increasing the number of receivers decreases the throughput. The reason is that a sender has to collect a larger number of ACKs generated by a larger number of receivers. It is noteworthy that the flow control mechanism opens the transmission window only after a locally originated message is acknowledged by all the receiver nodes. Possible solutions to this problem are discussed in Section 4.7.

Table 3 presents the results of UTO latency measurements at the receiver's side. As can be seen, in case of a fixed number of senders, increasing the number of receivers increases the

	Receivers			
Senders	1	2	3	4
1	512.7 (0.47)	493.0 (0.17)	477.0 (0.34)	467.1 (0.40)
2	512.5 (0.27)	491.7 (0.67)	475.7 (0.33)	
3	510.0 (0.55)	489.6 (0.41)		
4	509.2 (0.30)			

Table 2: Throughput (Mb/s) for different configurations

	Receivers			
Senders	1	2	3	4
1	2.3 (0.003)	3.1 (0.035)	3.2 (0.045)	3.1 (0.012)
2	2.5 (0.002)	3.1 (0.025)	3.4 (0.040)	
3	3.2 (0.004)	3.6 (0.041)		
4	4.9 (0.003)			

Table 3: UTO Latency (ms) for different configurations

latency. The explanation is similar to that for the throughput measurement experiments: the need to collect ACKs from all the receivers. Increasing the number of senders while the number of receivers is fixed causes an increase in the UTO Latency. Our hypothesis is that this happens due to an increase in the queues both at the switches and at the hosts.

As was mentioned above, in case a node either sends or receives packets, the utilization of the PCI bus and other system components is different from the case when a node acts as both a sender and a receiver. For this reason, the results presented in this section cannot be compared with those described above.

4.4 Tradeoffs of Latency vs. Throughput

In this section, we discuss the effect of an increased load on latency. In order to study the tradeoff of Latency vs. Throughput, a traffic generation scheme different from that in the previous experiments was used. The scheme was implemented by a benchmark application that generated a configurable amount of data.

4.4.1 All-to-all Configuration

In this section, all-to-all configuration is considered. Figure 5 shows the latencies for the 5-node configuration. Obviously, the UTO latency is always larger than the PO latency. One can see an increase in the latencies when the throughput achieves the 50Mb/s value,

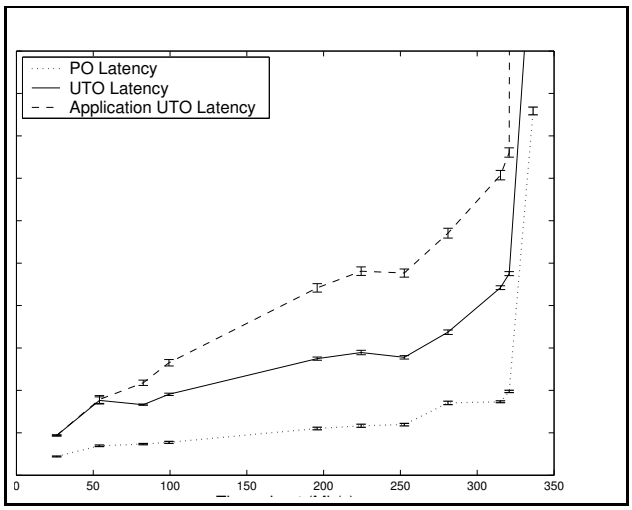

Figure 5: Latency vs. Throughput (all-to-all configuration)

i.e. a point from which small transient packet backlogs were created, and then a slight increase until the throughput approaches about 250Mb/s. After this point, the latencies start increasing. The PO latency reaches the value of about $1ms$ and UTO of about $3ms$ for throughput of about 330Mb/s.

We also measured the Application UTO Latency, which is the time interval from the point when the application sent a message until it can be "UTO delivered". One can see that when throughput increases, the Application UTO Latency increases too. This happens because the Linux 2.4 kernel allows events to be scheduled with a minimal granularity of $10ms$. Thus, in order to generate a considerable load, the benchmark application has to generate an excess number of packets every $10ms$. Packets that are not allowed to be sent due to the flow control mechanism are stored in a local buffer data structure. When ACKs arrive, the flow control mechanism enables sending some more packets previously stored for transmission. Packets that cannot be immediately sent increase the Application UTO Latency.

4.4.2 Large Packet Sizes

Figure 6 shows how increasing the application packet size, along with increasing the MTU size,affects the Application UTO Latency. In this experiment, we used disjoint groups of two senders and three receivers. We compared results achieved for jumbo frames with those obtained for regular Ethernet frames of MTU size. As expected, with jumbo frames larger throughput can be achieved, mainly due to the significantly reduced amount of PCI transactions.

When throughput increases, the Application UTO Latency increases, too, the reasons being the same as for the "all-to-all configuration". One can see that at lower throughput values, the jumbo frames show higher latency. This can be attributed to the fact that when

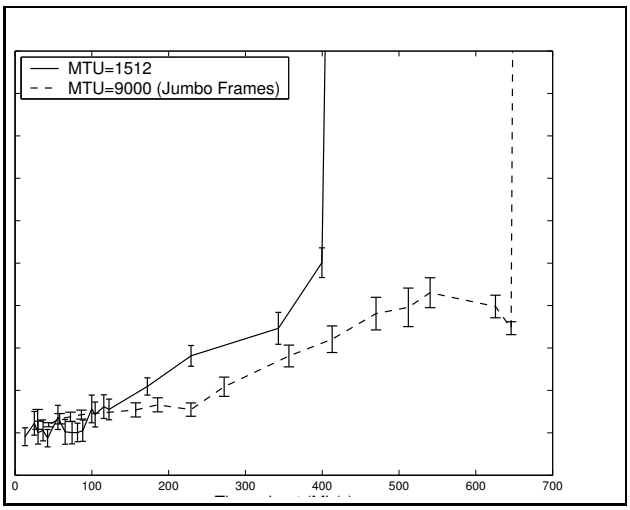

Figure 6: Latency vs. Throughput for different MTU sizes

the system is relatively free, the high transmission latency of jumbo frames dominates; in other words, the time for putting a jumbo frame on the wire is larger. As the load on the system increases, the overhead of the PCI bus and packet processing becomes the dominating factor, and using jumbo frames helps to reduce this overhead and thus to achieve the UTO faster.

4.4.3 Packet aggregation

The experiment evaluated, the effect of using the packet aggregation algorithm described in 3.2.1. Figure 7 shows the performance of the system with small packets, the payload size being about 64 bytes. Two accumulating packet sizes were used, Ethernet MTU of 1500B and jumbo frame size of 9000B. In addition, the same tests were conducted also without packet aggregation. Since the throughput without packet aggregation is considerably smaller, in the same figure the area corresponding to the throughput values between 0 and 40Mb/s is shown. One can see that the maximum throughput without packet aggregation is about 50Mb/s. On the other hand, using an accumulating size of 1500B increased the maximum throughput up to 400Mb/s. With accumulating size of jumbo frames, the throughput climbed as high as 630Mb/s, which is about one million small packets per second.

4.5 Losses

Table 4 presents the results of the system performance measured in the presence of message losses. The losses in this experiment were generated artificially, according to the specified loss probability parameter. One can see that the effect of a message loss is less severe when it occures on the sender side. The reason is that in this case the message order (PO) is

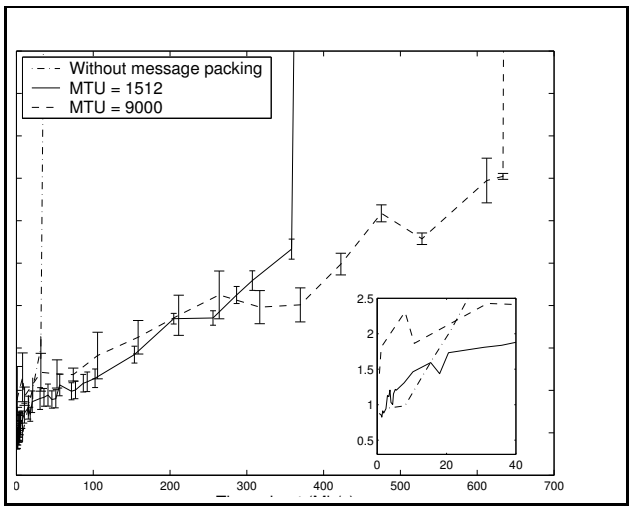

Figure 7: Packet aggregation (the low throughput area is extended)

the same at the receivers' side and the receivers need not to rearrange the already received messages.

4.6 Comparisons with previous works

In the experiment described below, we compared the performance of our system and the results presented in [15]where the performance of an algorithm based on weak ordering oracles (described in Section 3.4.2) and of the algorithm based on failure detectors [22] was analyzed. When carrying out the measurements for the comparative experiment, we tried to provide similar settings. All links were configured to 100Mb/s rate, the message size was 100 bytes, no message packing was used and the aggregation of ACKs limit was set up to 3. The experiments in [15] were performed at 4 nodes for weak ordering oracles and at 3 nodes for the algorithm based on failure detectors. In our experiments we used 4 nodes. Since the main parameters of the experiments under comparison coincide, while there might be differences in equipment and implementation environments, it is likely that the approximation is sufficient.

As the comparison presented in [15] shows, the maximum throughput for both algorithms was 250 messages per second. The latency of the weak ordering oracle algorithm increased from about 2.5s for the throughput of 50 messages/sec up to about $10ms$ for the throughput of 250 messages/sec. The performance of the algorithm based on failure detectors depends largely upon the timeout set for heartbeat messages. For large timeout of about $100ms$, the latency was in the range of 1.5-$2ms$, and for small timeout ($2ms$) the latency was in the range of 8-$10ms$.

Figure 8 presents the results of our experiments and shows that the throughput of about

Configuration	Loss Probability	Throughput *Mb/s*	PO Latency *ms*	UTO Latency *ms*
All-2-All	10	54	1.3	13.3
	5	118	4	15.1
	2	207	3	8.4
	1	251	2.4	9.2
	0.1	327	3.4	7.1
	0	348	2.7	6.4
2 Senders, 3 Receivers	5	173	1.0	30
	2	283	0.6	11
	1	346	0.9	6.7
	0.1	445	1.0	3.2
	0	475	1.8	3.4
2 Senders, 3 Receivers Only senders lose	10	252	0.6	1.7
	5	319	1.3	2.7
	2	397	1.5	3.0
	1	432	1.1	2.9
	0.1	468	1.4	3.0
	0	475	1.8	3.4

Table 4: Influence of message losses on Throughput and Latency

1000 messages/sec was achieved. The throughput of 300 messages/sec induces the PO latency of about 0.7*ms*, and the UTO latency was in the range of 1.7-2.2*ms*. The 95%-confidence interval was also computed and found practically negligible, as one can see in the graphs. It is important to note that while for low throughput our results do not differ significantly from those achieved by Pedone et al. [15], for high throughput they are much higher. The reason is that in our system, order does not break even if a message m is lost, as losses happen mostly in switch A (see Figure 4). So, if m is missed by a process, there is a high probability that m is lost by all the processes and PO order remains the same among all the processes. When m's sender discovers that m is lost, it retransmits m promptly.

Another question is whether the propagation time of a message in our two-switch topology is much higher than in a one-switch topology. Theoretically, the propagation time in a gigabit network over a single link is $\frac{1500*8}{10^9}$=0.012*ms*, the speed of signal transmission over the cable is negligible and the maximum processing time in the switch that we used is not more than 0.021*ms*. We performed two experimental measurements of propagation time. In the first experiment, ping utility was used to measure the latency of 1500-size packet, and 0.05*ms* propagation time was obtained in both topologies. In the second experiment, we

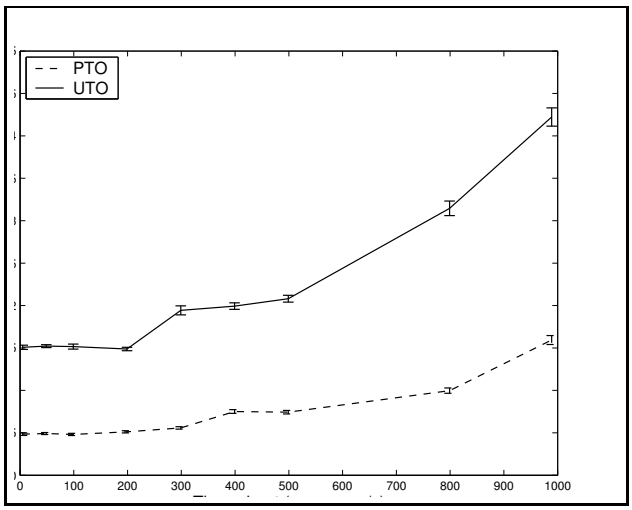

Figure 8: Comparisons with previous work.

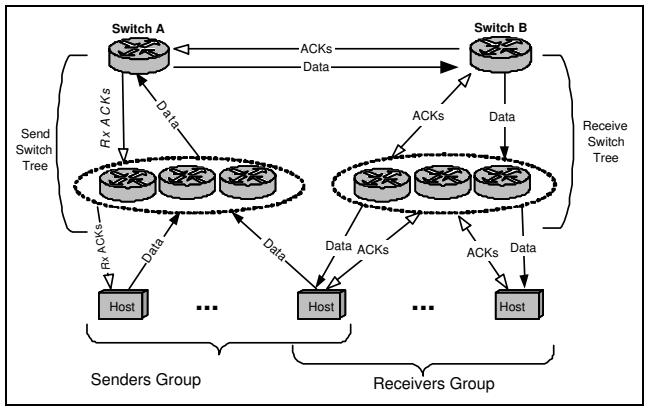

Figure 9: Expanded topology

used application level ping based on UDP protocol, as opposed to the original ping utility which works on kernel level. In the application level ping, we registered $0.12ms$ latency in both topologies. The results show that packet processing time ($\sim 0.1ms$) is much higher than message propagation time ($\sim 0.012ms$). We can conclude, therefore, that two-switch topology, without significantly increasing the latency, allows to predict message order with much higher probability!

4.7 Scalability Discussion

The number of ports in the switches is an important parameter setting the size of the system. The simplest way to expand the two-switch network is to use trees of switches. Figure 9 shows an example of such expanded topology. Each sender is connected to an intermediate switch

which is in turn connected to switch A. Also, each receiver is connected to switch B via an intermediate switch. If a node belongs to both groups, senders and receivers, it is connected to two intermediate switches which are connected to switches A and B, respectively. In this topology, the link between switches A and B continues to serve as virtual sequencer. The path traversed by each message is longer, but, as shown above, the propagation time is very small.

4.7.1 Ack aggregation

The measurements showed that increasing the number of receivers decreases the throughput. The reason is that a sender has to collect a larger number of ACKs generated by a larger number of receivers. There are a few ways to make a system scalable in number of receivers. In [15], an algorithm was proposed that reduces number of ACKs required to deliver a message. This approach can be further improved by using recently introduced NICs [23] which have an embedded CPU that enables to offload some of the tasks currently running on the host CPU. Our system can offload to those NICs the tasks of sending, collecting and bookkeeping ACK messages.

Our measurements showed only a small degradation of throughput (about 0.5% per sender) when the number of senders increases. Implementing efficient flow control for big number of senders is a more serious challenge. In future work we are going to explore hardware flow control [24] over each link. The main idea is to slowdown switch B (see Figure 4), when the number of free buffers in a receiver is below a threshold. As a result, switch B starts accumulating messages, and when its number of free buffers falls significantly, it causing switch A to slow down. Switch A may now either drop messages or slowdown the senders. Our current implementation already knows how to deal with message drops. If a message was missed by all the receivers, the impact on the throughput is insignificant.

Another important issue related to the scalability problem is the ability to support multiple groups. The most naive solution is to use only one group and to implement multiple group support on the application level. However, this solution is not always optimum, as we force each node to receive all the traffic. In future work, we intend to investigate another approach in which an IP Multicast address is associated with each group. As modern switches support IGMP, a message will be delivered only to hosts that are members of this group. Considering possible bottlenecks in this solution, we see that the link from switch B to a host is not a bottleneck, as host may stay away from participating in all the groups. However, the link between the switches may introduce a new bottleneck to the overall system. There are several solutions to this problem. One is to use the new 10 Gb/s standard which soon will be available for uplinks connecting two switches. Another solution that has already been implemented to increase throughput between two network components is trunk [25]. We assume that switches can be configured to associate a link in trunk with a destination IP address. In addition, it is possible to support more groups by using more switches connected as it was described in Section 3.4.2.

Chapter 5

Summary

5.1 Related Work

There is a multitude of work which deals with the problem of Total Ordering of messages in a distributed system. A comprehensive survey of this field, covering various approaches and models can be found in [1]. The authors distinguish ordering strategies based on where the ordering occurs, i.e. senders, receivers or sequencers. We use the network as a virtual sequencer, so our algorithm falls into the last category.

As was noted in Section 1.1, the approach called "Optimistic Atomic Broadcast" was first presented in [26]. In [27] interesting approach to achieving total order with no message loss was presented. The authors introduced buffer reservation at intermediate network bridges and hosts. The networking equipment connecting the senders and receivers was arranged in a spanning tree. The reservation was made on the paths in the spanning tree so that no message loss could occur. The ordering itself was done using Lamport timestamps [28]. The paper assumed a different network and presents only simulation results, which makes it hard to perform comparisons.

Another implementation of a Total Ordering algorithm in hardware was proposed in [29]. This work offloads the ordering mechanism into the NIC and uses CSMA/CD network as a virtual sequencer. The authors assume that single collision domain connects all the participating nodes. Using a special software and hardware, the algorithm prevents nodes that missed a message from broadcasting new messages, thus converting the network to a virtual sequencer. In our opinion, use of single collision domain is the main drawback of this approach. It is known that collisions may reduce the performance of system significantly.

Another work that deals with Total Ordering and hardware is presented in [30]. In this work, a totally ordered multicast which preserves QoS guarantees is achieved. It is assumed that the network allows bandwidth reservation which is specified by average transmission rate and the maximum burst. The algorithm suggested in the paper preserves that latency and the bandwidth reserved for the application.

5.2 Contribution

In this work we made the following contributions:

- We proposed a new cost-effective approach that uses only off-the-shelf hardware products. The approach is not limited to CSMA/CD networks and can be applied to other networks.
- The approach was implemented and evaluated in a real network.
- We removed significant overhead from middleware that implements active state machine replication. It is known that replication usually provides good performance for read requests, but incurs a significant overhead on write requests [3]. We reduced the latency and increased the throughput of the middleware, providing total order.

Bibliography

[1] Defago, X., Schiper, A., Urban, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Computing Surveys **36** (2004) 372–421

[2] Kemme, B., Pedone, F., Alonso, G., Schiper, A.: Processing transactions over optimistic atomic broadcast protocols. In: Proceedings of 19th International Conference on Distributed Computing Systems (ICDCS'99). (1999)

[3] Burns, R.: Data Management in a Distributed File System for Storage Area Networks. PhD thesis (2000) University of California, Santa Cruz.

[4] M. Christensen, K. Kimball, F. Solensky: Considerations for IGMP and MLD Snooping Switches. (IETF draft)

[5] Vicente, P., Rodrigues, L.: An indulgent uniform total order algorithm with optimistic delivery. In: Proc. 21st IEEE Symposium on Reliable Distributed Systems, IEEE CS (2002)

[6] Mishra, S., Wu, L.: An evaluation of flow control in group communication. IEEE/ACM Trans. Netw. **6** (1998) 571–587

[7] Nagle, J.: Congestion Control in TCP/IP Internetworks. RFC 896 (1984)

[8] Friedman, R., van Renesse, R.: Packing messages as a tool for boosting the performance of total ordering protocols. In: HPDC. (1997) 233–242

[9] Chase, J., Gallatin, A., Yocum, K.: End system optimizations for high-speed TCP. IEEE Communications Magazine **39** (2001) 68–74

[10] Anker, T., Breitgand, D., Dolev, D., Levy, Z.: Congress: Connection-oriented group-address resolution service. In: SPIE. (1997)

[11] Schiper, A., Birman, K., Stephenson, P.: Lightweight causal and atomic group multicast. ACM Trans. Comput. Syst. **9** (1991) 272–314

[12] Chockler, G.V., Keidar, I., Vitenberg, R.: Group communication specifications: a comprehensive study. ACM Comput. Surv. **33** (2001) 427–469

[13] Keidar, I., Khazan, R.: A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In: ICDCS '00: Proceedings of the The 20th International Conference on Distributed Computing Systems (ICDCS 2000), IEEE Computer Society (2000) 344

[14] Lamport, L.: The part-time parliament. ACM Transactions on Computer Systems **16** (1998) 133–169

[15] Pedone, F., Schiper, A., Urban, P., Cavin, D.: Solving agreement problems with weak ordering oracles. In: EDCC-4: Proceedings of the 4th European Dependable Computing Conference on Dependable Computing, Springer-Verlag (2002) 44–61

[16] M.Ben-Or: Another advantage of free choice:completely asynchronous agreement protocols. In: Annual ACM Symposium on Principles of Distributed Computing. (1983) 27–30

[17] M.Rabin: Randomized byzantine generals. In: 24th Annual ACM Symposium on Foundations of Computer Science. (1983) 403–409

[18] F.Pedone, A.Schiper: Optimistic atomic broadcast: a pragmatic viewpoint. Theor. Comput. Sci. **291** (2003) 79–101

[19] IEEE: 802.1s Multiple Spanning Tree Standard. (IEEE standard)

[20] Wadge, W.: Achieving gigabit performance on programmable ethernet network interface cards. (2001) http://www.cs.um.edu.mt/ ssrg/wthesis.pdf.

[21] Moll, L., Shand, M.: Systems performance measurement on PCI pamette. In Pocek, K.L., Arnold, J., eds.: IEEE Symposium on FPGAs for Custom Computing Machines, Los Alamitos, CA, IEEE Computer Society Press (1997) 125–133

[22] Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. In Herlihy, M., ed.: Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing (PODC'92), Vancouver, BC, Canada, ACM Press (1992) 147–158

[23] Donnelly, S.: High Precision Timing in Passive Measurements of Data Networks. PhD thesis (2002) University of Waikato, New Zeland.

[24] IEEE: 802.3x Flow Control Standard. (IEEE standard)

[25] IEEE: 802.3ad Link Aggregation Standard. (IEEE standard)

[26] Pedone, F., Schiper, A.: Optimistic atomic broadcast. In: Proceedings of the 12th International Symposium on Distributed Computing, Springer-Verlag (1998) 318–332

[27] Chen, X., Moser, L.E., Melliar-Smith, P.M.: Reservation-based totally ordered multicasting. In: Proc. 16st Intl. Conf. on Distributed Computing Systems (ICDCS-16), IEEE CS (1996)

[28] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21** (1978) 558–565

[29] Jalote, P.: Efficient ordered broadcasting in reliable csma/cd networks. In: Proc. 18st Intl. Conf. on Distributed Computing Systems (ICDCS-18), IEEE CS (1998)

[30] Bar-Joseph, Z., Keidar, I., Anker, T., Lynch, N.: Qos preserving totally ordered multicast. In: Proc. 5th International Conference On Principles Of Distributed Systems (OPODIS). (2000) 143–162