Optimizing Total Order Protocols for State Machine Replication

Thesis for the degree of

DOCTOR of PHILOSOPHY

by

Ilya Shnayderman

SUBMITTED TO THE SENATE OF

THE HEBREW UNIVERSITY OF JERUSALEM

December 2006

This work was carried out under the supervision of Prof. Danny Dolev

Acknowledgements

I would like to thank my family for their constant understanding and support, which was not always easy. My deep gratitude goes to my advisor professor Danny Dolev for his precise though unobtrusive guidance, to Tal Anker for his constant readiness to share knowledge and cooperate. And of course, I am grateful to all the members of our Transis/Danss lab, who managed to create just the right atmosphere for creative and efficient work. I would like to thank my Swiss colleagues Peter Urban and Andre Shipper for fruitful cooperation.

Contents

Abstract

Message delivery order is a fundamental service in distributed middleware tools. Total order (TO) is a basic message delivery order guarantee and has been acknowledged as a powerful building block for supporting consistency and fault-tolerance in distributed applications. TO is typically used in replicated objects and database applications. Although the applicability of TO appears to be wide enough, it is rarely used for other kinds of distributed applications. This is due to a number of reasons, the main reason being, as we believe, the perception that using TO seriously limits the performance of the system and seriously increases message delivery latency. The goal of our study was to prove that there are implementations of TO where those disadvantages are minimized without compromising the advantages of TO. We developed simple and efficient protocols providing TO, implemented them and evaluated their performance.

Total order in LAN. Many distributed algorithms rely on Failure Detectors (FD). FDs are used by a process in distributed systems to monitor another process(es), and are able to detect a process failure and notify other process(es) about it. Unfortunately, failure detectors sometimes yield incorrect information. In this thesis the impact of inaccuracy of FD on distributed middleware performance is studied.

In order to overcome negative impact of inaccuracy of FD, we propose a approach that allows ordering messages at high rates without rely on FD. Due to an optimization introduced into a known method for proving TO, we developed a novel architecture that is significantly more efficient at high load rates. The feasibility of the architecture has been proved by implementation that used only on-the-shelf hardware. The performance of the system was evaluated, and the experiments showed that this architecture enables to order more than million messages per second.

Towards total order in WAN. Distributed applications usually exploit UDP protocol. Among the advantages of this protocol, there is support of multicast message propagation and immediate message delivery. The major disadvantage of this protocol is that it lacks built-in congestion control (CC). CC in WAN is very important, as the network resources are shared by multiple users. In this work, an implementation of TCP-friendly CC for distributed applications is presented and its performance evaluated.

Link failures in WAN do not always separate the processes into two or more disjoint sets. In order to solve this problem, we propose a novel method that allows establishing connections between clusters of processes without a need to agree on the membership among all the processes. This method is based on the hierarchal approach. We proved this approach to be feasible, and performed an initial performance evaluation of our implementation.

In order to develop an efficient algorithm for TO in WAN, we studied the performance of two TO algorithms. The study showed that if no message is lost, the running times of the algorithms correspond to the theoretical running time. At the same time, it was observed that message losses significantly increased the latency of TO. We showed that a negative impact of message losses on the symmetrical algorithm was higher than on the centralized one.

Chapter 1

Introduction

In this thesis, some challenges facing modern distributed systems are analyzed and a few novel solutions aimed at overcoming those challenges are presented and evaluated. Nowadays, the amount of network services is constantly increasing. Those services have not only to support a large number of users but also to provide timely response to users worldwide. Since no single computer is able to provide such service, many network services rely on distributed systems that may be of different sizes, from a few computers connected to one Local Area Network to tens of thousands computers (e.g. Google) located in different countries. Creating distributed applications is highly complicated, as developers have to handle failure/disconnection of the computers without causing inconveniences to the user.

To solve this problem, distributed middleware is used, which simplifies the task, as the developer does not have to handle each possible failure scenario. Typically, a middleware tool allows to send and to receive messages over the network. One of the goals of the middleware is to deliver messages in a consistent way, despite the fact that the network may be not reliable. The middleware semantics usually guarantee that a message is delivered either to all the processes or to none of them. However, it is sometimes impossible to deliver a message due to a loss of connectivity. In this case, the middleware notifies the distributed system which responds appropriately.

Typically, a distributed system consists of processes that are running instances of distributed applications. Each instance can send and receive messages and to process them when received. The processing of a message can lead to sending other messages, as well as to changing the state of the application

instance. In order to provide meaningful service, the application keeps its instances synchronized. In order to keep the states of different instances consistent, the middleware tool is required to enforce an order on message delivery.

Message delivery order is a major service in distributed middleware tools. For more than two decades, extensive research has been conducted to define useful delivery guarantees and to develop efficient algorithms providing these guarantees. Total order (TO) is one of the basic message delivery order guarantees and has been acknowledged as a powerful building block for supporting consistency and fault-tolerance in distributed applications. By way of example, we may consider a server S that processes requests of the database. In order to make database service fault-tolerant, it is possible to replicate server S. In case S fails, one of the replicas (instances) picks up its functionality and starts responding to the database requests. Thus, TO allows to apply transactions to different instances in exactly the same order, which ensures that all the instances keep their state consistent.

A significant effort has been made to develop proficient algorithms supporting efficient semantics (guarantees) of total order. This effort is reflected in numerous publications. Extensive analysis and classification of the algorithms providing total order can be found in [40]. Although initially the above-described middleware was developed mostly by research bodies, the software industry quickly recognized the efficiency and practical value of total order. Total order is now used by leading software companies such as IBM, Sun and Microsoft [57, 71].

Total order is typically used in replicated objects and database applications. Although the potential applicability of total order appears to be wider [33, 87], it is rarely used for other kinds of distributed applications. This is due to a number of reasons, the main reason being, as we believe, the perception that using TO seriously limits the performance of the system and significantly increases message delivery latency. The goal of our study was to prove that there are implementations of TO where those disadvantages are minimized without compromising the advantages of TO. We developed simple and efficient protocols providing TO, implemented them and evaluated their performance.

1.1 Problem Description

1.1.1 Model

The goal of the present study is to optimize total ordering of messages in distributed systems. We assume the message-passing environment in distributed systems to be asynchronous, which means that processes communicate solely by exchanging messages and there is no bound on message delivery time. Processes may fail by crashing and later recover or not. Processes may any time voluntarily choose leaving or re-joining. Communication links may fail and recover.

1.1.2 Uniform Total Order - with Early Delivery

Total order is a semantics that insures that messages sent to a set of processes are delivered by all these processes in the same order, thus enabling fault-tolerance. Most algorithms attempt to guarantee the order required by a replicated database application, namely, *Uniform Total Order (UTO)* defined in [102] by the following primitives:

- **UTO1 - Uniform Agreement :** If a process (correct or not) has $U - delivered(m)$, then every correct process eventually $U - delivers(m)$.
- **UTO2 - Termination :** If a correct process $U - broadcast(m)$, then every correct process eventually $U - delivers(m)$.
- **UTO3 - Uniform Total Order :** Let m_1 and m_2 be two $U - broadcast$ messages. It is important to note that $m_1 < m_2$ if and only if a process (correct or not) $U - delivers$ m_1 before m_2. Total order ensures that the relation "$<$" is acyclic.
- **UTO4 - Integrity :** For any message m, every correct process $U - delivers(m)$ at most once, and only if m was previously $U - broadcasted$.

UTO enables processes to perform an easier recovery from faults, by ensuring that even faulty process does not $U - deliver$ a message out of order. It is important to note that UTO does not keep FIFO.

A new approach to usage of total order called "Optimistic" has been introduced recently. This approach, which in fact is UTO with Early Delivery, allows to reduce the latency of database transactions by guessing the final order of the transactions on the earlier stages of the protocol. After the order of a

transaction is guessed, the transaction is executed by the database. However, the transaction results are committed to the database only if the order of the transactions was guessed correctly. Otherwise, the transaction is aborted.

Our approach to optimization of UTO with early delivery stems from a number of acknowledged problems that are currently unsolved and which limit a wider implementation of UTO:

- TO protocols and semantics are highly complicated
- In Local Area Network (LAN), using TO limits significantly the performance of the system and increases message delivery latency.
- In Wide Area Network (WAN), implementation of TO faces problems of frequent message losses, congestion control and appropriate membership.

The aim of our study is to analyze the root causes of these problems and suggest ways to solving them.

1.2 Solution Highlights and Thesis Layout

The thesis is separated into two parts. The first part is dedicated to optimizing total order in LAN, while the second part describes essential building blocks needed in order to develop efficient total order algorithm in WAN and presents an evaluation of recently introduced total order algorithms for WAN.

Implementing TO often faces problems caused by *Failure Detectors* (FD), since distributed middleware usually relies on FD. FDs are used by a process in distributed systems to monitor another process(es), and are able to detect a process failure and notify other process(es) about it. Unfortunately, failure detectors sometimes provide incorrect information [31] e.g. when a correct process is erroneously suspected as a failed one. Another drawback is timing: if the monitored process fails, FD may not be able to provide the corresponding information instantly, due to the delay in message propagation over the network. Developers of distributed middleware are well aware of these drawbacks of FD, and the protocols currently used are able to cope with the problem. This, however, comes for a price, as the delay in discovering a failed process or an error in suspecting a correct process may slowdown the performance. In Chapter 2, the impact of *Failure Detectors* (FD) on distributed middleware performance is studied.

Most of the protocols for implementing total order incorporate failure detectors, thus making the algorithm performance dependent on the accuracy of FD. To reduce this dependence, it is possible to improve FD, as discussed in [11]. Another way is to use an algorithm for total order that does not include FD. Such an algorithm was developed by Pedone et al. [82]. This algorithm is based on random consensus proposed by Rabin in [84]. However, the idea suggested in [82] has a drawback, since at high load the performance drops drastically. From first sight, this algorithm is not better than that based on the FD, as the accuracy of FD usually also depends on the network utilization. In Chapter 3 we propose a novel solution that improves the idea proposed [82] by allowing to order messages at high rates. This solution based on on-the-shelf hardware has been implemented and its performance evaluated in the present work. The experiments showed that it is possible to order more than million messages per second.

In Chapters 2 and 3, the performance of total order protocols was studied in LAN. However, many of modern distributed systems are required to operate in WAN. WAN introduces new challenges to distributed systems. One of the challenges is frequent message losses. Many network applications use TCP protocol in order to overcome the message losses. However, TCP protocol supports only one message order, FIFO. Moreover, TCP can not be used for sending messages to multiple destinations (multicast). For this reason, many distributed applications prefer using a more flexible protocol, UDP. The drawback of this protocol is that it lacks built-in congestion control (CC). Congestion control in WAN is very important, as the network resources are shared by multiple users. As it was stated earlier, a great amount of network traffic is composed from TCP flows. This requires that distributed applications be able to compete with TCP communication on the network resources in a fair way. In Chapter 4, an implementation of TCP-friendly congestion control for distributed applications is presented and its performance evaluated.

When processes communicate over WAN, they can experience not only considerable amount of message losses, but also a temporal link failure. This link failure may disconnect the process from other processes. In traditional group communication systems, this situation is considered as a partition. Link failures in WAN do not always separate the processes into two or more disjoint sets[1]. In this case, it is difficult to agree on a stable membership. In order to solve this problem, we propose in Chapter 5 a novel method that allows to establish connections between clusters of processes, without a need to agree on the

[1] If at least one process belongs to different sets they should be the same.

membership among all the processes. This method is based on the hierarchal approach. In Chapter 5 we showed that this approach is feasible, and performed an initial performance evaluation of our implementation. The proposed solution can be extended to implement virtual synchrony [63]. Otherwise, it is difficult to agree on a stable membership. In order to solve this problem, we propose in Chapter 5 a novel method that allows to establish connections between clusters of processes, without a need to agree on the membership among all the processes. This method is based on the hierarchal approach. In Chapter 5 we showed that this approach is feasible, and performed an initial performance evaluation of our implementation. The proposed solution can be extended to implement virtual synchrony [63].

In Chapter 6 we studied the performance of two total order algorithms [102] and [92] which had been designed specially for WAN. The study showed that if no message was lost, the running times of the algorithms correspond to the theoretical running time. At the same time, it was observed that message losses significantly increased the latency of total order. In Chapter 6 we showed that the negative impact of the message losses on the symmetrical algorithm was higher than on the centralized one.

In this thesis, we studied issues that have impact on the performance of total order algorithms. The simulations were run both in LAN and in WAN. In Chapter 2, we showed that when run in LAN, inaccuracy of Failure Detectors may cause degradation of total order algorithm performance. We also showed that, when messages are sent at higher rates, the inaccuracy of Failure Detectors increases. To minimize this drawback, we proposed in Chapter 3 an approach which does not use Failure Detectors, and showed that its performance does not degrade significantly when network utilization increases. We presented in Chapter 4 an implementation and an evaluation of TCP-friendly congestion control. In Chapter 5 we propose a novel hierarchical approach to membership that takes into account specific Wide Area Network type of failures when processes A and B may be connected to process C, but are not able to exchange messages. In Chapter 6 we showed that message losses in WAN are a significant factor in total order algorithms for WAN.

1.3 Related Work

We relate our work to the following research areas:

1. Performance evaluation of total order algorithms

2. Congestion Control

3. Group Membership

1.3.1 Total Order

There are a number of works which deal 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 [40]. The authors cover tens of papers that describe different implementation of total order algorithms. Usually, when an algorithm is proposed, its performance is evaluated using simple metrics like time complexity (number of communication steps) and message complexity (number of messages). This, however, renders little information on the real performance of those algorithms. There are a few papers that provide a more detailed performance analysis of total order algorithms: [38] and [39] analyze four different algorithms using discrete event simulation; [95] uses a contention-aware metric to analytically compare the performance of four algorithms; [36, 35] analyze total order protocols for wireless networks, deriving assumption coverage and other performance-related metrics. However, all these papers analyze the algorithms only in failure-free runs, which gives just a partial understanding of their quantitative behavior.

Other papers analyze agreement protocols, taking into account various failure scenarios: [44] presents an approach for probabilistic verification of a synchronous round-based consensus protocol; [85] analyzes a Byzantine total order protocol; [68] evaluates the performability of a group-oriented multicast protocol;[91] compares the impact of different implementations of failure detectors on a consensus algorithm (simulation study); [37] analyzes the latency of the Chandra-Toueg consensus algorithm. Similar to the approach used as in Chapter 2, [37] models failure detectors using the quality of service (QoS) introduced by Chen et al. [31].

There are a number of works implementing total order relying on hardware. In [32], an 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 performed using Lamport timestamps [65]. The paper assumed a different network and presents only simulation results, which makes it hard to

perform any comparisons. An implementation of a total ordering algorithm in hardware was proposed in [61]. This work offloads the ordering mechanism into the NIC and uses CSMA/CD network as a virtual sequencer. The authors assume that a single collision domain connects all the participating processes. Using special software and hardware, the algorithm prevents processes that missed a message from broadcasting new messages, thus converting the network into a virtual sequencer. In our opinion, the use of a single collision domain is the main drawback of this approach, as it is known that collisions may significantly reduce the performance of system.

Another work that deals with total ordering and hardware is presented in [18]. In this work, a totally ordered multicast which preserves QoS guarantees is achieved. It is assumed that the network allows bandwidth reservation specified by average transmission rate and the maximum burst. The algorithm suggested in the paper preserves the latency and the bandwidth reserved for the application.

The new approach called "Optimistic Atomic Broadcast" was first presented in [81]. This approach allows to reduce database transactions latency by starting transactions earlier based on guessing the total order before it is finally established.

1.3.2 Congestion Control

Congestion control and flow control have attracted a lot of research in computer networking. There are numerous approaches to introducing congestion control mechanisms for multicast applications. A comprehensive survey on current unicast and multicast congestion control approaches can be found in [106].

In our study, we focus on a single-rate multicast congestion control in which the slowest receiver limits the transmission rate. The major alternative approach uses layered multicast scheme ([26, 70, 103]). Our choice of a single rate is accounted for by the fact that group communication framework requires that all receivers be synchronized. The proposed approach could be extended to multi-layer multicast by using the proposed technique on the rate of each layer.

Several papers propose congestion control schemes for systems with a large number of receivers. In such environment, it is impossible to handle feedback from each sender due to the ACK explosion problem. It is necessary to estimate the receiving capability of the slowest receiver. There are two main approaches used to avoid the ACK explosion problem, the first identifying the slowest receiver and

referring to it only, and the second enforcing a hierarchy on the receivers.

PGMCC and TFMCC use the first approach, whose main disadvantage comes from the fact that the "worst" receiver may change rapidly, while performing a switch-over among different slow receivers might be slow and extremely difficult ([41, 62]).

The second approach is more suitable for GCS, as group membership is maintained and available for group members. Xpand uses an external group membership service to receive group membership notification ([6]). MTCP ([86]), being a one-to-many scheme, creates a multi-level tree hierarchy on receivers. Obviously, a multi-level hierarchy is natural to one-to-many approach. Xpand, which is designed for many-to-many communication, uses a two-level hierarchy, which is a more natural approach for multi-cluster systems. Other protocols like RMTP ([79]) and TMTP ([108]) are one-to-many, taking advantage of the hierarchy, and are not specifically designed to be TCP-friendly. RMTP deploys end-to-end congestion control, whereas TMTP implements flow control only.

Multicast communication typically consumes more resources from the network than unicast, since multicast traffic usually traverses through more links, thus a protocol designed for bulk data transfer over multicast in WAN must exhibit TCP-friendliness. TEAR ([88]) and MTCP ([86]) use a window-based technique to provide a TCP-friendly congestion control mechanism. The authors report promising results, however, the model is not easily extensible to many-to-many multicast applications due to scalability problems.

An alternative approach to the window-based mechanism implemented in TEAR and in MTCP is the equation-based one. This approach uses a stochastic TCP model ([77]) which represents a throughput of a TCP sender as a function of packet loss rate and of round trip time. TFRC (RFC 3448 ([54]), see also the paper [47]) has been recently recognized and standardized by IETF as a sound approach to TCP-friendliness for unicast traffic. The congestion control mechanism deployed in Xpand was based on this approach.

Only few group communications systems have an end-to-end congestion control mechanism. As to flow control mechanisms for group communication, they have been widely employed (a comprehensive analysis of flow control mechanisms for LAN GCSs can be found in [72])

A system that is comparable to our system is Spread ([4]). Spread uses an overlay network, in which each overlay link behaves in a TCP-friendly manner. In addition, Spread implements an advanced end-to-end flow control mechanism based on a cost-benefit approach([3]).

1.3.3 Group Membership

The authors of Congress [6] and Maestro [24] were the first to observe that separating maintenance of membership from a group multicast will better enhance scalability of fault-tolerant distributed applications. This separation was later adopted by researchers who addressed the WAN environment ([63, 7, 93]). InterGroup [21] presents another WAN approach to address scalability.

Several research projects sought to relax the semantics of the middleware for distributed applications ([4, 56, 24, 93, 83, 42, 101, 48, 53]). Our work takes advantage of both approaches, in order to find a better balance between efficiency, scalability and guaranteed semantics.

Recently, new implementations [109, 89] of reliable multicast have appeared, whose protocols use peer-to-peer overlay systems. Those systems scale for large group, while members are not required to keep group membership information, which might be critical for some applications.

Part I

Optimizing Total Order for LAN

Chapter 2

Impact of Failure Detectors*

Middleware tools use a variety of building blocks like consensus, total order etc. Protocols that guarantee those building blocks have been extensively studied in various system models, and many protocols solving these problems have been published [19, 40], offering different levels of guarantees. However, these protocols have mostly been analyzed from the point of view of their safety and liveness properties, while little has been done to analyze their *performance*. In addition, most papers focus on analyzing failure-free runs, thus neglecting the performance aspects of failure handling. In our view, this limited understanding of performance aspects, both in failure-free scenarios and scenarios including failure handling, is an obstacle to adopting such protocols in practice.

Failure detectors impact The goal of this chapter is to study the impact of failure detectors, in particular their inaccuracy, on Uniform Total Order algorithms. Two algorithms providing Uniform Total Order were chosen. The first one is based directly on failure detectors, the other one on a *group membership service*. Both services provide processes with estimates concerning the set of crashed processes in the system.[1] The main difference between those two approaches is that failure detectors provide inconsistent information about failures, whereas group membership service provides consistent information. It is important to note that group membership service also depends on failure detectors.

*This chapter is based on a paper by P. Urban, I. Shnayderman and A. Schiper [98].

[1] Besides masking failures, a group membership service has other uses. This issue is discussed in Section 2.6.

The two algorithms. The algorithm using unreliable failure detectors (FD) is the Chandra-Toueg total order algorithm [28], which can tolerate $f < n/2$ crash failures and requires a failure detector $\Diamond\mathcal{S}$. As an algorithm using group membership, we chose one that implements total order with a mechanism close to the FD-based algorithm, i.e., a sequencer-based algorithm (which also tolerates $f < n/2$ crash failures). Both algorithms were optimized for (1) failure and suspicion-free runs (rather than runs with failures and suspicions), (2) minimizing latency under low load (rather than minimizing the number of messages), and (3) tolerating high load (rather than minimizing latency at moderate load).

We chose these algorithms because they are well-known and easily comparable, offering the same guarantees within the same model. Moreover, they behave similarly in cases when neither failures nor failure suspicions occur (in fact, they generate the same exchange of messages given the same arrival pattern). This allows us to focus the analysis on the differences and similarities concerning handling failures and suspicions.

Methodology for performance studies. The two algorithms were evaluated using simulation. We modeled message exchange by taking into account contention on the Local Area Network and the hosts [95]. We modeled failure detectors (including the ones underlying group membership) in an abstract way, using the quality of service (QoS) metrics proposed by Chen et al. [31]. Our performance metric for uniform total order is called *latency*, defined as the time that elapses between sending a message m and its earliest delivery of m. We studied the uniform total order algorithms in several benchmark scenarios, including scenarios with failures and suspicions, by evaluating the steady state latency in (1) runs with neither crashes nor suspicions, (2) runs with crashes and (3) runs with no crashes in which correct processes are wrongly suspected to have crashed, as well as (4) the transient latency after a crash.

We believe that our methodology can be generalized to analyze other fault-tolerant algorithms. This makes the results of the comparison a basis for the proposed methodology.

The results. The analysis shows that the two algorithms have the same performance in runs with neither crashes nor suspicions. At the same time, the group membership-based algorithm has an advantage in terms of performance and resiliency that remain in force for a long time after crashes occur. In other scenarios involving wrong suspicions of correct processes and the transient behavior after crashes, the failure- detector-based algorithm offers better performance. The results clearly indicate that the im-

pact of inaccuracy of failure detectors on both algorithms is high. The results we obtained have several implications concerning the design of fault-tolerant distributed systems.

2.1 System model

In this study, we used asynchronous message passing system model, similar to the one described in 1.1.1. In addition, we assumed that channels are reliable, which is easily achieved in practice by retransmitting lost messages. We considered only those processes that failed by crashing and, therefore, do not send any further messages. Process crashes are rare, processes fail independently from one another, and process recovery is slow, since both the time between crashes and the time to repair are much greater than the latency of total order.

Similar to almost all the fault-tolerant algorithms described in the literature, the total order algorithms considered in this chapter use some form of crash detection. We call the parts of the algorithms that implement crash detection*failure detectors*. A failure detector maintains a list of processes it suspects to have crashed. Mistakes are possible, namely, it might suspect correct processes to fail and it might not suspect crashed processes immediately. To make sure that the total order algorithms terminate, we need some assumptions regarding the behavior of the failure detectors [27]. Both algorithms require $\Diamond\mathcal{S}$failure detectors. In other words, this requires strong completeness (eventually every process that crashes is permanently suspected by every correct process) and weak accuracy (some correct processes are never suspected). These requirements are rather weak and can usually be fulfilled in real systems by tuning implementation parameters of the failure detectors [96].

It should be noted that whereas we assume that process crashes are rare, (wrong) failure suspicions may occur frequently, depending on the tuning of the failure detectors.

2.2 Algorithms

This section introduces the two total order algorithms and the group membership algorithm (a more detailed description can be found in [99]). The analysis of the expected performance of the two total order algorithms follows.

2.2.1 Chandra-Toueg uniform total order algorithm

The Chandra-Toueg uniform total order algorithm uses failure detectors directly [28]. We refer to it as the FD total order algorithm, or simply as the *FD algorithm*. A process executes U-broadcast by sending a message to all the processes. [2] When a process receives such a message, it buffers it until the delivery order is decided. In order to agree on delivery order, consensus is used. The delivery order is decided by a sequence of consensus numbered 1, 2, ...

The initial value and the decision of each consensus is a *set of message identifiers*. Suppose $msg(k)$ be the set of message IDs decided by consensus $\#k$. The messages denoted by $msg(k)$ are U-delivered before the messages denoted by $msg(k+1)$, and the messages denoted by $msg(k)$ are U-delivered according to a deterministic function, e.g., according to the order of their IDs.

Chandra-Toueg $\Diamond\mathcal{S}$ consensus algorithm. To solve the consensus, we use the Chandra-Toueg $\Diamond\mathcal{S}$ algorithm [28]. The algorithm can tolerate $f < n/2$ crash failures. It is based on the rotating coordinator paradigm: each process executes a sequence of asynchronous rounds (i.e., not all processes necessarily execute the same round at a given time t), and in each round a process takes the role of *coordinator* (p_i is coordinator for rounds $kn + i$). The role of the coordinator is to impose a decision value on all the processes. If it succeeds, the consensus algorithm terminates. It may fail if some processes *suspect* the coordinator to have crashed, regardless whether the coordinator really crashed or not. In this case, a new round is started. We skip the details of the execution which are not relevant for our analysis

Example run of the FD algorithm. Figure 2.1 illustrates execution of the FD total order algorithm in which one single message m is U-broadcast and neither crashes nor is suspected to do so At first, m is sent to all the processes. Upon receipt, the consensus algorithm starts. The coordinator sends its proposal to all other processes. Each process acknowledges this message. Upon receiving ACKs from the majority of processes (including itself), the coordinator makes up its own proposal and sends the decision (using reliable broadcast) to all other processes. The other processes decide upon receiving the decision message.

[2] This message is sent using reliable broadcast. We use an efficient algorithm inspired by [50] that uses only one broadcast message in most cases; see [99] for more details.

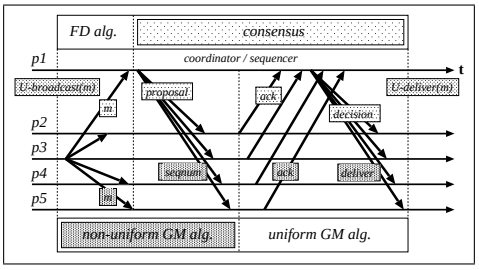


Figure 2.1: Example run of the total order algorithm. Labels on the top/bottom refer to the FD/GM algorithm, respectively.

2.2.2 Fixed sequencer uniform total order algorithm

The second uniform total order algorithm is based on a fixed sequencer [23]. It uses a group membership service for reconfiguration in case of a crash. We shall refer to it as GM total order algorithm, or simply as the *GM algorithm*. We describe here the *uniform* version of the algorithm.

In GM algorithm, one of the processes takes the role of a *sequencer*. When a process U-broadcasts a message m, it first broadcasts it to the sequencer. Upon reception, the sequencer (1) assigns a sequence number to m, and (2) broadcasts the sequence number to all processes. When the non-sequencer processes have received m and its sequence number, they send an acknowledgment to the sequencer.[3] The sequencer waits for ACKs from the majority of processes, then delivers m and sends a message indicating that m can be U-delivered. The other processes U-deliver m when they receive this message. The execution is shown in Fig. 2.1. It should be noted that the messages denoted *seqnum*, *ack* and *deliver* can carry several sequence numbers, which is essential for achieving good performance under high load. It is important that the FD algorithm has a similar "aggregation" mechanism, i.e. one execution of the consensus algorithm can decide on the delivery order of several messages.

When the sequencer crashes, processes need to agree on the new sequencer. This is why we need a group membership service which provides a consistent *view* of the group to all its members, i.e., a list of the processes which have not crashed (informally speaking). The sequencer is the first process in the current view. The group membership algorithm described below can tolerate $f < n/2$ crash failures (more in some runs) and requires a failure detector $\Diamond\mathcal{S}$.

[3] Figure 2.1 shows that the acknowledgments and subsequent messages are not needed in the non-uniform version of the algorithm. This issue is considered later in the chapter.

2.2.3 Group membership algorithm

A group membership service [34] maintains the *view* of a group, i.e. the list of correct processes of the group. The current view[4] might change because processes in the group might crash or exclude themselves, while processes outside the group might join. The group membership service guarantees that processes see the same sequence of views (except for processes which are excluded from the group that miss all views after their exclusion until they join again). In addition to maintaining the view, our group membership service ensures *View Synchrony* and *Same View Delivery*: correct and not suspected processes deliver the same set of messages in each view, and all deliveries of a message m take place in the same view.

Our group membership algorithm [69] uses failure detectors to start view changes, and relies on consensus to agree on the next view. This is done as follows. A process that suspects another process starts a view change by sending a "view change" message to all the members of the current view. As soon as a process learns about a view change, it sends its unstable messages[5] to all the processes. When a process has received the unstable messages from all the processes it does not suspect, say P, it computes the union U of the unstable messages received, and starts consensus with the pair (P, U) as its initial value. Let (P', U') be the decision of the consensus. Once a process decides, it delivers all messages from U' not yet delivered, and installs P' as the next view. The protocol for joins and explicit leaves is practically the same.

State transfer. When a process joins a group, its state needs to be synchronized with the other members of the group. The exact meaning of "state" and "synchronizing" is application- dependent. We only need to define these terms in a limited context. In our study, the only processes that ever join are correct processes which have been wrongly excluded from the group. Consequently, the state of such a process p is mostly up-to-date. For this reason, it is feasible to update the state of p in the following way: when p rejoins, it asks some process for the messages it has missed since it was excluded. Process p delivers these messages, and then starts to participate in the view it has joined. It should be noted that the procedure works only because our total order algorithm is uniform. In case of a non-uniform total order, the excluded process might have delivered messages never seen by the others, thus having

[4] There is only one current view, since we consider in this chapter a *non-partitionable* or *primary partition* group membership service.

[5] Message m is *stable* for process p when p knows that m has been received by all other processes in the current view.

an inconsistent state, which means that state transfer would be more complicated.

2.2.4 Expected performance

We now discuss, from a qualitative point of view, the expected relative performance of the two total order algorithms (FD algorithm and GM algorithm).

Figure 2.1 shows executions with neither crashes nor suspicions. In terms of the pattern of message exchanges, the two algorithms are identical, only the content of messages differ. Therefore, we expect the same performance from the two algorithms in failure-free and suspicion-free runs.

We investigated how the algorithms slow down when a process crashes. There are two major differences. The first difference is that GM algorithm reacts to the crash of *every* process, while FD algorithm reacts only to the crash of p_1, which is the first coordinator. The second difference is that GM algorithm takes longer time to restart delivering total order messages after a crash. This is true even if we compare GM algorithm to the case which is the worst one for FD algorithm, i.e., when the first coordinator p_1 fails. FD algorithm needs to execute Round 2 of the consensus algorithm. This additional cost is comparable to the cost of an execution with no crashes (3 communication steps, 1 multicast and about $2n$ unicast messages). On the other hand, GM algorithm initiates an expensive view change (5 communication steps, about n multicast and n unicast messages). Hence, we expect that if the failure detectors in both algorithms detect the crash at the same time , FD algorithm performs better. However, when GM algorithm establishes the new membership and reaches steady-state, its latency is a bit lower than that of FD algorithm, as it can deliver messages after collecting lower number of acknowledgments.

Next we consider the case when a correct process is wrongly suspected. The algorithms react to a wrong suspicion in the same way as they react to a real crash. Therefore, we expect that if the failure detectors generate wrong suspicions at the same rate, FD algorithm will suffer less performance penalty.

2.3 Context of our performance study

2.3.1 Performance measures

Our main performance measure is the *latency* of total order. Latency L is defined for a single total order message m as follows. Suppose *U-broadcast(m)* occurs at time t_0, and *U-deliver(m)* on p_i at time t_i,

for each $i = 1, \ldots, n$. Then latency is defined as the time elapsed until the first U-delivery of m, i.e., $L \stackrel{\text{def}}{=} (\min_{i=1,\ldots,n} t_i) - t_0$. In our study, we compute the mean for L over multiple messages and several executions.

This performance metric is efficient in practice. By way of example, we can consider a service replicated for fault tolerance using active replication [90]. Clients of this service send their requests to the server replicas using total order. Once a request is delivered, the server's replica processes the client request, and sends back a reply. The client waits for the first reply, and discards the other ones identical to the first one. If we assume that the time to service a request is the same on all replicas, and the time to send the response from a server to the client is the same for all servers, then the first response received by the client is the response sent by the server to which the request was delivered first. Thus, there is a direct link between the response time of the replicated server and the latency L.

Latency is always measured under a certain workload. We chose simple workloads: (1) all destination processes send total order messages at the same constant rate and (2) the U-broadcast events come from a Poisson stochastic process. We call the overall rate of total order messages *throughput*, denoted by T. In general, we determine how the latency L depends on the throughput T.

2.3.2 Scenarios

We evaluate the latency of the total order algorithms in various scenarios. Below, the scenarios are described in detail, mentioning the parameters that influence latency in the scenario. The parameters that influence latency in all the scenarios are the algorithm (A), the number of processes (n) and the throughput (T).

Steady state of the system. We measure latency after it stabilizes (a sufficiently long time after the start of the system or after a crash). We distinguish three scenarios, based on whether crashes and wrong suspicions (failure detectors suspecting correct processes) occur:

- **normal-steady:** Neither crashes nor wrong suspicions in the experiment.
- **crash-steady:** One or several crashes occur before the experiment. Besides A, T and n, an additional parameter is the set of crashed processes. As we assume that the crashes happened a long time ago, all failure detectors in the system permanently suspect all crashed processes at this

point. No wrong suspicions occur.

- **suspicion-steady:** No crashes, but failure detectors generate wrong suspicions, which causes the algorithms to take extra steps and thus increases the latency. Besides A, T and n, additional parameters include the frequency at which wrong suspicions occur and how long they last. These parameters are discussed in detail in Section 2.4.2.

It would be meaningful to combine the crash-steady and suspicion-steady scenarios in order to have both crashes and wrong suspicions. However, this case is beyond the scope of the chapter, , since we wanted to observe the effects of crashes and wrong suspicions independently.

Transient state after a crash. In this scenario, we force a crash after the system reached a steady state. After the crash, we can expect a halt or a significant slowdown of the system for a short period. Here, we define latency so that it reflects the latency of executions that are affected by the crash and thus happen around the moment of the crash. Also, we must take into account that not all crashes affect the system in the same way. Below, we consider the worst case , when the crash that slows down the system most. Our definition is the following:

- **crash-transient:** Process p crashes at time t (neither crashes nor wrong suspicions occur, except for this crash). We have process q ($p \neq q$) execute *U-broadcast*(m) at t. Let $L(p, q)$ be the mean latency of m, averaged over a lot of executions. Then $L_{crash} \stackrel{def}{=} \max_{p,q \in P} L(p, q)$, i.e., we consider the crash that affects the latency most. In this scenario, we have one additional parameter, describing how fast the failure detectors detect the crash (discussed in Section 2.4.2).

We could combine the crash-transient scenario with the crash-steady and suspicion-steady scenarios, to include other crashes and/or wrong suspicions. These cases were not considered, , since we wanted to independently observe the effects of (i) the recent crash, (ii) old crashes and (iii) wrong suspicions . Another reason is that we expect the effect of wrong suspicions on latency to be secondary in comparison to the effect of the recent crash, as wrong suspicions usually happen on a larger timescale.

2.4 Simulation models

Our approach to performance evaluation is simulation, which allowed for more general results in comparison to those feasible to obtain with measurements in a real system , since we can use a parameter in our network model to simulate a variety of different environments. We used the Neko prototyping and simulation framework [97] to conduct our experiments.

2.4.1 Modeling the execution environment

The transmission of messages was modeled in the following way. We use the model of [95], inspired by simple models of Ethernet networks [94]. The key point in the model is that it accounts for *resource contention*. This point is important, as resource contention is often a limiting factor for the performance of distributed algorithms. Both a host and the network itself can be a bottleneck. These two kinds of resources appear in the model (see Fig. 2.2): the network resource (shared among all processes) represents the transmission medium, and the CPU resources (one per process) represent the processing performed by the network controllers and the layers of the networking stack, during the emission and the reception of a message (the cost of running the algorithm is negligible). A message m transmitted for process p_i to process p_j uses the resources (i) CPU_i, (ii) network, and (iii) CPU_j, in this order. Message m is put in a waiting queue before each stage if the corresponding resource is busy. The time spent on the network resource is our time unit. The time spent on each CPU resource is λ time units; the underlying assumption is that sending and receiving a message has a roughly equal cost.

The λ parameter ($0 \leq \lambda$) shows the relative speed of processing a message on a host compared to transmitting it over the network. Different values model different networking environments. We conducted experiments with a variety of settings for λ.

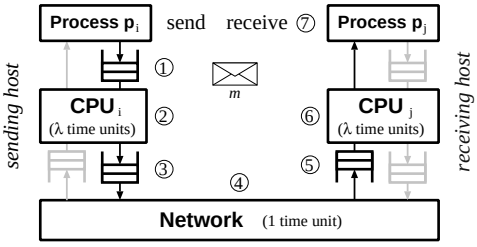


Figure 2.2: Transmission of a message in our network model.

Crashes are modeled as follows. If a process p_i crashes at time t, no messages can pass between p_i and CPU_i after t. However, the messages on CPU_i and the attached queues are still sent, even after time t. In real systems, this corresponds to a (software) crash of the application process (operating system process), rather than to a (hardware) crash of the host or a kernel panic. We chose to model software crashes because they are more frequent in most systems [51].

2.4.2 Modeling failure detectors

One approach to modeling a failure detector is to use a specific failure detection algorithm and model all its messages. However, this approach would restrict the generality of our study, as another choice for the algorithm would likely give different results. Also, it is not justified to model the failure detector in so much detail, while other components of the system, e.g. the execution environment, are modeled in much less detail. We built a more abstract model instead, using the notion of quality of service (QoS) of failure detectors introduced in [31]. The authors consider the failure detector at a process q that monitors another process p, and identify the following three primary QoS metrics (see Fig. 2.3):

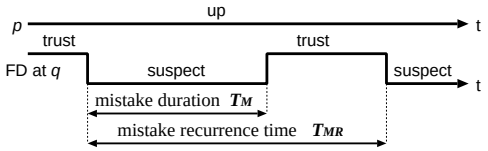


Figure 2.3: Quality of service metrics for failure detectors. Process q monitors process p.

Detection time T_D: The time that elapses from p's crash to the time when q starts suspecting p permanently.

Mistake recurrence time T_{MR}: The time between two consecutive mistakes (q wrongly suspecting p), given that p did not crash.

Mistake duration T_M: The time it takes a failure detector component to correct a mistake, i.e., to trust p again (given that p did not crash).

Not all of these metrics are equally important in each of our scenarios (see Section 2.3.2). In Scenario *normal-steady*, the metrics are not relevant. The same holds for Scenario *crash-steady*, because we observe the system for a sufficiently long time after all crashes, long enough to have all failure detectors to suspect the crashed processes permanently.

In Scenario *suspicion-steady* no crash occurs, hence the latency of total order only depends on T_{MR} and T_M. In Scenario *crash-transient* no wrong suspicions occur, hence T_D is the relevant metric.

In [31], the QoS metrics are random variables, defined on a pair of processes. In our system, where n processes monitor each other, we thus have $n(n-1)$ failure detectors in the sense of [31], each characterized with three random variables. In order to have an executable model for the failure detectors, we have to define (1) how these random variables depend on each other and (2) how the distribution of each random variable can be characterized. To keep our model simple, we assume that all failure detector modules are independent and the tuples of their random variables are identically distributed. Moreover, we do not need to model how T_{MR} and T_M depend on T_D, as the two former parameters. are only relevant in Scenario *suspicion-steady*, whereas T_D is only relevant in Scenario *crash-transient*. In our experiments, we considered various settings for T_D, as well as various settings for combinations of T_{MR} and T_M. As for the distributions of the metrics, we took the simplest possible choices: T_D is a constant, and both T_{MR} and T_M are exponentially distributed with (different) constant parameters.

It should be stressed that these modeling choices do not completely reflect the complex reality, since suspicions from different failure detectors are probably correlated. However, our work presents a new methodology in studying impact of failure detectors on distributed algorithms and can serve a starting point for the future research. We are not aware of any previous work we could build on (apart from [31] that makes similar assumptions.

2.5 Results

We now present the results for all the four scenarios. Due to lack of space, we only present the results obtained with $\lambda = 1$ in this chapter. In current LANs, the time spent on the CPU is higher than the time spent on the wire, and thus $\lambda > 1$. Results for such values of λ are presented in [99].

Most graphs show latency vs. throughput. For easier understanding, we set the time unit of the network simulation model to 1 ms. The 95% confidence interval is shown for each point of the graph. The two algorithms were executed with 3 and 7 processes, to tolerate 1 and 3 crashes, respectively.

Normal-steady scenario (Fig. 2.4). In this scenario, the two algorithms have the same performance. Each curve thus shows the latency of *both* algorithms (as we remember, the represented throughput shows

the number of generated U-broadcasts per second).

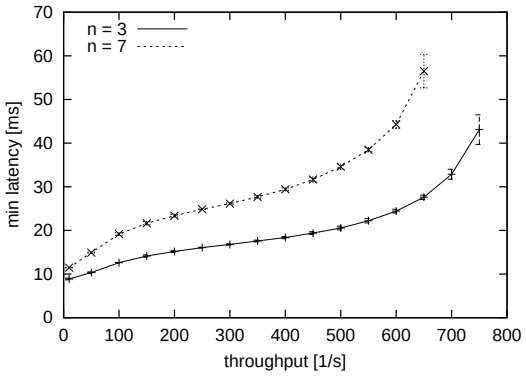


Figure 2.4: Latency vs. throughput in the normal-steady scenario.

Crash-steady scenario (Fig. 2.5). For both algorithms, the latency decreases as more processes crash. This is due to the fact that the crashed processes do not load the network with messages. GM algorithm has an additional feature that improves performance: the sequencer waits for fewer acknowledgements as the group size decreases with the crashes. By comparison, the coordinator in the FD algorithm always waits for the same number of acknowledgments. This explains why GM algorithm shows slightly better performance at the same number of crashes.

For GM algorithm, it does not matter which process(es) crash. For FD algorithm, the crash of the coordinator of Round 1 gives a worse performance than the crash of another process. However, the performance penalty when the coordinator crashes is easily avoided: (1) each process tags its consensus proposal with its own identifier and (2) upon decision, each process re-numbers all processes so that the process with the identifier in the decision becomes the coordinator of Round 1 in subsequent consensus executions. In this way, crashed processes will eventually stop being coordinators, hence the steady-state latency is the same, no matter which process(es) we forced to crash. Moreover, the optimization incurs at no cost. Hence, Fig. 2.5 shows the latency in runs in which non-coordinator processes crash.

It should be also emphasized that GM algorithm has higher resiliency in the long term if crashes occur, as the group size decreases with the crashes. E.g., with $n = 7$ and 3 crashes, GM algorithm can still tolerate one crash after excluding the crashed processes, whereas FD algorithm can tolerate none.

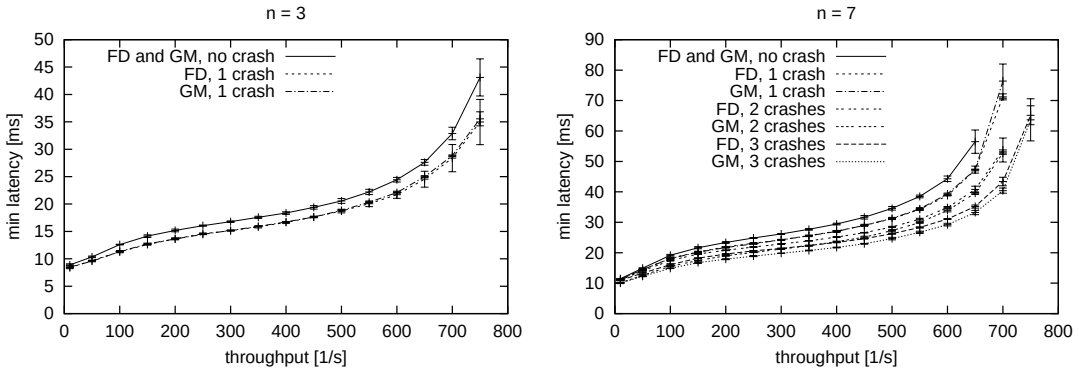


Figure 2.5: Latency vs. throughput in the crash-steady scenario. The legend lists the curves from the top to the bottom.

Suspicion-steady scenario (Fig. 2.6, 2.7). The occurrences of wrong suspicions are quantified with the T_{MR} and T_M QoS metrics of the failure detectors. As this scenario involves crashes, we expect the mistake duration T_M to be short. In our first set of results (Fig. 2.6) we hence set T_M to 0, and the latency is shown as a function of T_{MR}. We have four graphs: the left column shows results with 3 processes, the right column those with 7; the top row shows results at a low load (10 s^{-1}) and the bottom row at a moderate load (300 s^{-1}); as we saw earlier in Fig. 2.4, the algorithms can take a throughput of about 700 s^{-1} in the absence of suspicions.

The results show that both algorithms are sensitive to wrong suspicions. It is also evident that GM algorithm is much more sensitive to suspicions: even at $n = 3$ and $T = 10$ s^{-1}, it only works if $T_{MR} \geq 50$ ms, whereas t FD algorithm still works at $T_{MR} = 10$ ms; the latency of the two algorithms is only equal at $T_{MR} \geq 5000$ ms.

In the second set of results (Fig. 2.7) T_{MR} is fixed and T_M is on the x axis. We chose T_{MR} such that the latency of the two algorithms is close but not equal at $T_M = 0$: (i) $T_{MR} = 1000$ ms for $n = 3$ and $T = 10$ s^{-1}; (ii) $T_{MR} = 10000$ ms for $n = 7$ and $T = 10$ s^{-1} and for $n = 3$ and $T = 300$ s^{-1}; and (iii) $T_{MR} = 100000$ ms for $n = 7$ and $T = 300$ s^{-1}.

The results show that GM and FD algorithms are sensitive to the mistake duration T_M as well, and not just to the mistake recurrence time T_{MR}, while again GM algorithm is much more sensitive.

Crash-transient scenario (Fig. 2.8). In this scenario, we only present the latency after the crash of the coordinator and the sequencer, respectively, as this is the case resulting in the highest transient latency

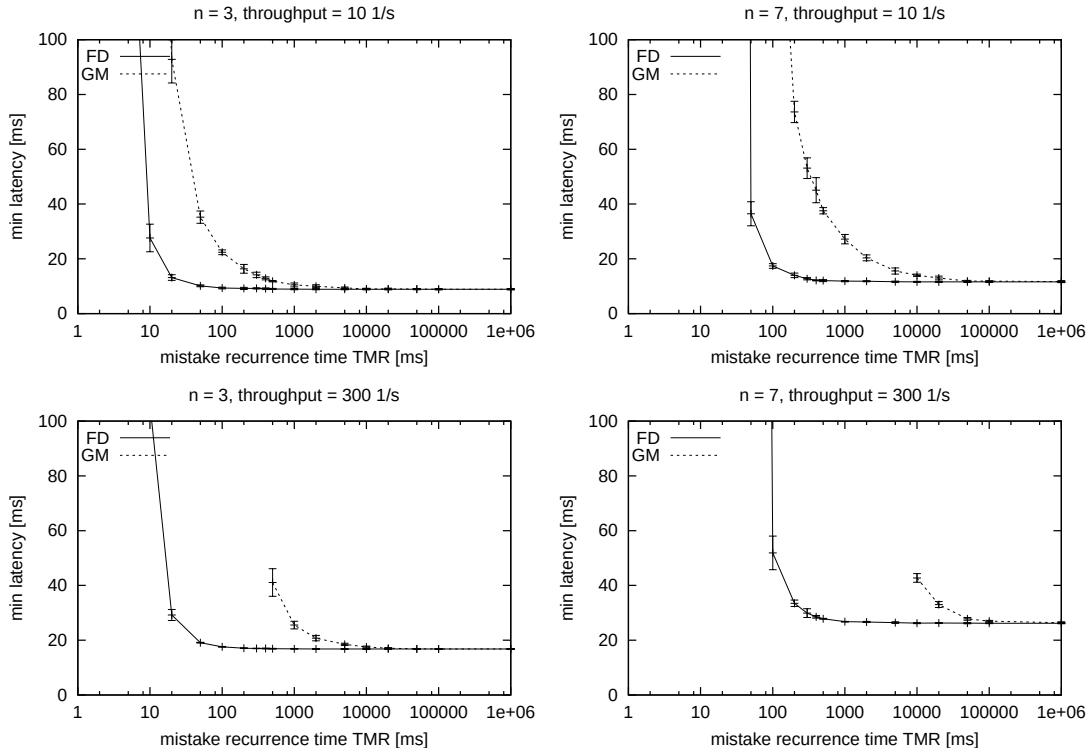


Figure 2.6: Latency vs. T_{MR} in the suspicion-steady scenario, with $T_M = 0$.

and is, in fact, and the most interesting comparison If another process is crashed, the GM algorithm performs roughly the same as when a view change occurs. In contrast, FD algorithm outperforms GM algorithm, as it performs slightly better than in the normal-steady scenario (Fig. 2.4), as fewer messages are generated, just like in the crash-steady scenario (Fig. 2.5).

Figure 2.8 shows the *latency overhead*, i.e., the latency minus the detection time T_D, rather than the latency. Graphs showing the latency overhead are more illustrative; as can be seen, the latency is always greater than the detection time T_D in this scenario, as no total order can terminate until the crash of the coordinator/sequencer is detected. The latency overhead of both algorithms is shown for $n = 3$ (left) and $n = 7$ (right) and a variety of values for T_D.

The results show that (1) both algorithms perform rather well, i.e. the latency overhead of both algorithms is only a few times higher than the latency in the normal-steady scenario; see Fig. 2.4) and that (2) FD algorithm outperforms GM algorithm in this scenario.

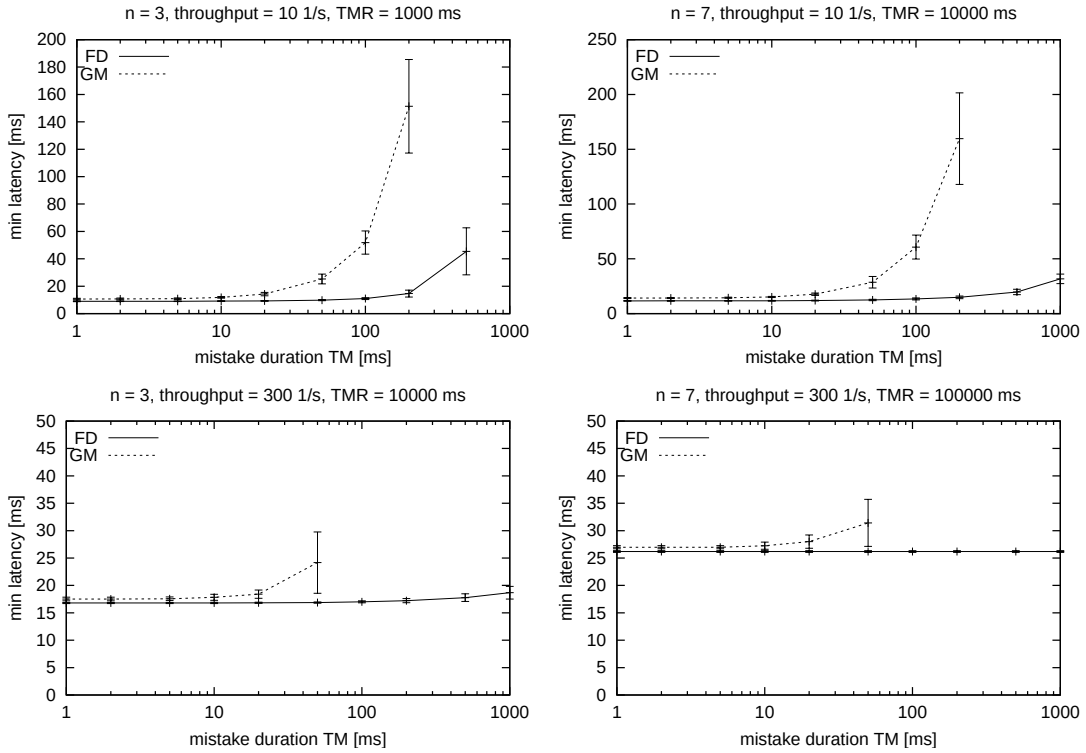


Figure 2.7: Latency vs. T_M in the suspicion-steady scenario, with T_{MR} fixed.

2.6 Discussion

We investigated two uniform total order algorithms designed for the same system model: an asynchronous system (with a minimal extension to allow us to have live solutions to the total order problem) and $f < n/2$ process crashes (the highest f that our system model allows). We found that in the absence of crashes and suspicions, the two algorithms have the same performance. However, a long time after any crashes, the group membership-based algorithm (GM) performs slightly better and has better resilience. In the scenario involving wrong suspicions of correct processes and the one describing the transient behavior after crashes, the failure-detector-based algorithm(FD) outperformed the GM-based algorithm. The difference in performance is much greater when correct processes are wrongly suspected.

Combined use of failure detectors and group membership. Based on our results, we advocate a combined use of the two approaches [29]. Failure detectors should be used to make failure handling more responsive (in case of a crash) and more robust (tolerating wrong suspicions). A different failure detector,

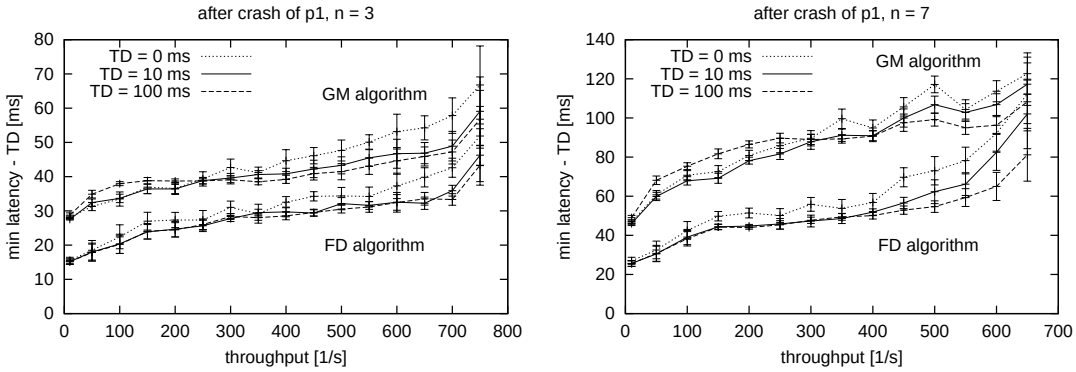


Figure 2.8: Latency overhead vs. throughput in the crash-transient scenario.

making fewer mistakes at the expense of slower crash detection should be used in group membership service, to provide the long-term performance and resiliency benefits after a crash. A combined use is also desirable because the failure detector approach is only concerned with failure handling, whereas a group membership service has a lot of essential features besides failure handling: processes can be taken offline gracefully, new processes can join the group, and crashed processes can recover and join the group. Also, group membership can be used to garbage-collect messages in buffers when a crash occurs [29].

Generality of our results. We chose total order algorithms with a centralized communication scheme where one process coordinates the others. The algorithms are practical, since in the absence of crashes and suspicions, they are optimized both to have small latency under low load and to work under high load as well , because messages needed to establish delivery order are aggregated.

Methodology for performance studies. In this chapter, we proposed a methodology for performance studies of fault-tolerant distributed algorithms. Its main characteristics are the following.

1. We define:
   - repeatable benchmarks, i.e., scenarios specifying the workload
   - the occurrence of crashes and suspicions
   - and the performance measures of interest;
2. We include various scenarios with crashes and suspicions into the benchmarks

3. We describe failure detectors using quality of service (QoS) metrics

The methodology allowed us to compare the two algorithms easily, as only a small number of parameters are involved. Currently, the methodology is defined only for total order algorithms, but it is our intention to extend it to analyze other fault tolerant algorithms.

Chapter 3

Wire-Speed Total Order*

In Chapter 2, we have shown that inaccuracy of failure detectors have a significant impact on Total Order (TO) algorithms. However, it is possible to provide TO without relying on failure detectors as it was demonstrated in [82]. The results presented by the authors show that at a low throughput the latency of the algorithm is low, although the algorithm latency grows significantly when the throughput is increased. In this chapter, we propose a novel mechanism that allows to implement Wire-Speed TO while keeping the latency low.

In order to achieve those results, we chose a recent technological trend that implies introducing hardware elements into distributed systems. Implementing parts of a distributed system in hardware immediately imposes performance requirements on its software parts. An example of a system that combines hardware and software elements is a high-capacity Storage Area Network, combining a cluster of PC's, Disk Controllers and interconnected switches that can benefit from high-speed TO. The rationale for using TO in this combined system is elaborated on in greater detail in [11, 52].

This chapter shows how message *ordering* can be guaranteed in a distributed setting, along with a significant increase in the number of "transactions" produced and processed. The proposed architecture uses off-the-shelf technology with minor software adaptations. One of the most popular approaches to achieve TO implies 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., by CPU power.

The goal of the methodology presented in this chapter is to achieve a hardware-based sequencer while

*This chapter is based on a paper by T. Anker, D.Dolev, G. Greenman and I. Shnayderman [9].

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 a near wirespeed traffic of a totally ordered stream of messages.

Below, the architecture, its assumptions and the adjustments made to the PC are described. The performance results proved efficient, as a near wire-speed traffic of totally ordered messages has been achieved. The proposed architecture can be adjusted to various high-speed networks, among them InfiniBand [17] and Fiber-Channel [16], which currently do not support multicast with ordering guarantees. In addition, our approach includes a highly efficient optimistic delivery technique which can be utilized in various environments, e.g. replicated databases, as shown in [64].

3.1 Contribution

The following contributions have been made as a result of the study:

- A new cost-effective approach that uses only off-the-shelf hardware products is proposed . The approach is not limited to CSMA/CD networks and can be applied to other networks as well.
- The approach has been implemented and evaluated within a real network.
- We managed to remove 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 [25]. We reduced the message latency and increased the throughput of the system that can now perform ordering of more than a million messages per second.

3.2 Model and Environment

The model used in the study is the same asynchronous message-passing model that was presented in 1.1.1. In addition, we assume that the distributed setting is composed of a set of computing elements processes

are (PCs, CPU based controllers, etc.) residing on a LAN connected by switches. The computing elements, referred to as processes, can be either transaction initiators (senders), or receivers, or both.

The processes are connected via full-duplex links through commodity switches. We assume that the switches support IGMP snooping [67]. 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 minimal chance of packet loss. The main source of packet loss is a buffer overflow rather than a link error. In section 3.5, the fault tolerance issues are discussed . Our assumption is that the participating group of processes is already known. Dynamic group technology can be used to deal with changes in group membership, although this case is not considered in this chapter.

3.3 Problem Definition

The main goal of the study is to provide efficient mechanism for total ordering of messages. Most algorithms attempt to guarantee the order required by a replicated database application, namely, *Uniform Total Order (UTO)* defined in 1.1.2. Our system guarantees not only UTO, but also 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 TO. PO and TO should be later either confirmed or changed by the UTO

3.4 Implementation

As noted above, our implementation of Total Ordering follows the methodology based on 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 the simplicity of

presentation, we assume that all the processes 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, namely, that all the paths between the set of NICs for transmission (TX) and the set of NICs for reception (RX) share (intersect in) at least one link (see Section 3.7 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 the frames are received on a network interface of the switch and egress 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 in order to improve performance, even though this feature is not required for the algorithm's correctness.

In our implementation, multicast is used in order to efficiently send messages to the processes group. Our goal is to have all these messages received in the same order by the set of processes that desire to get them (the receivers group). To achieve this, we dedicate a single link between the two switches for the multicast traffic flows. Figure 3.1 shows the general network configuration of both the network (the switches) and the attached nodes. The methodology of the network ensures that all the processes 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, all the traffic is transmitted to the processes in the same order via the second switch. As the switches and the links preserve the FIFO order, this in turn implies that all the messages are received in the same order by all the processes.

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 3.5.

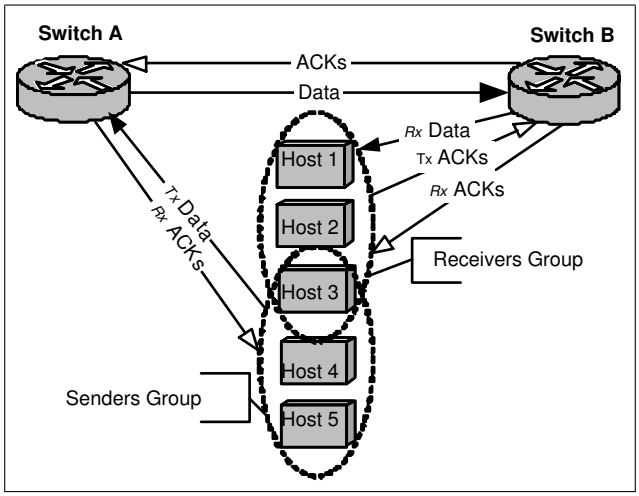


Figure 3.1: Architecture

3.4.1 Providing UTO

The preliminary ordering of the hardware configuration is not enough to ensure UTO because messages may get lost or processes may fail. To address this issue, our protocol uses a *simple* positive acknowledgment (ACK) scheme (similar to the TCP/IP protocol) to ensure that the PO is identical at all the receivers. Each receiver process U-delivers (see Section 3.3) a message to the application only after it has collected ACKs from each receiver process in the system. In order to reduce the number of circulating auxiliary control messages within 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 process needs to hold messages for some period of time. This implies that sender processes 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 [72]) in order to maintain the transmission window. Each sender process 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 processes 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 when most of the application messages are significantly smaller

In [11] more efficient schemes to collect ACKs are considered

than network Maximum Transmission Unit (*MTU*), a variation of a Nagle algorithm [75] was used as described in section 3.4.2.1. In order to increase the performance for small messages, a variation of a Nagle algorithm [75] is used as described in Section 3.4.2.1. Since the main source of message losses is the buffer overflow, careful tuning of the flow control mechanism combined with ACKs aggregation can reduce the risk of losing messages. For our particular configurations, we identified the appropriate combination of the window size and the number of aggregated ACKs to achieve maximum throughput. The specific implementation of the flow control mechanism presented in this chapter allows overall performance to converge with the receiving limit of the PCI bus.

3.4.2 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 number of small messages (where the messages 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.

3.4.2.1 Packet Aggregation Algorithm

It was stated by [49] that at high loads, message packing is the most influential factor for total ordering protocols. We use an approach similar to that in the Nagle algorithm [75], in order to cope with a large number 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.

3.4.2.2 Jumbo frames

The standard frame size in Gigabit Ethernet is ~1512 bytes. The size of the jumbo frame is ~9000 bytes. Numerous studies show MTU size has an impact on the overall performance, such as [30], which reports increased performance when jumbo frames are exploited. 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 occurs. Since we control all the components in the proposed system, we configured all the network components to prevent fragmentation of the messages and thus avoided the problem . Performance results prove that jumbo frames allow to obtain better throughput. For example, in the configuration of two senders and three receivers, we achieved a maximum throughput of 722Mb/s.

3.4.3 Multicast Implementation Issues

As mentioned above, every process 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 process 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 process;
- 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 which ensures that TO is maintained.

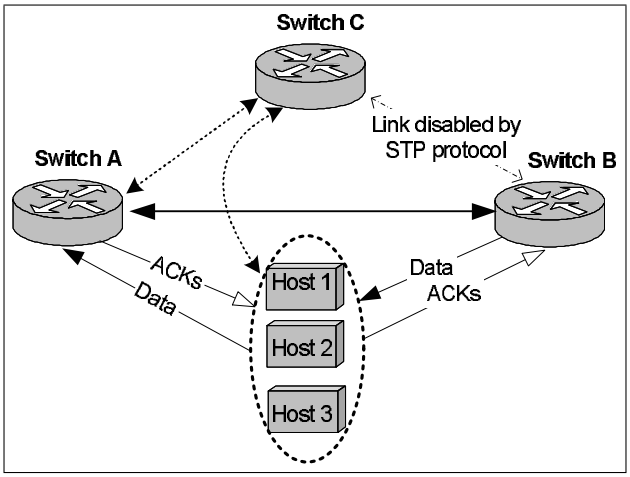


Figure 3.2: Network with 3 switches

3.5 Fault-Tolerance

Faults may occur at various levels of packet handling and can be caused by different events: a switch failure, a physical link disconnection, a failure of a process and a crash of a process running the process. All these failures can be identified by failure detectors (FD). In Chapter 2, it was shown that inaccuracy of FD can lead to significant drop in the performance.

An interesting approach that does not rely on FD was presented by Pedone et al. [82]. 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 lost or delivered out of order). The paper shows that TO broadcast can be achieved using a weak ordering oracle. The approach is based on the algorithm proposed in [20]. In [82], another algorithm is also proposed that solves TO broadcast in two communication steps, assuming $f < \frac{n}{3}$. This algorithm is based on the randomized consensus algorithm proposed in [84]. 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. [82]. This approach allows our architecture to tolerate losses. It is also noteworthy that in our implementation, losses that have occurred before the message reached switch B (recall 3.1), can be easily overcome by retransmission of the message by its originator. The impact of such loss on the system performance of such loss is not significant, as is showed at [52].

In our architecture one can easily recognize a single point of failure: if one of the switches or the

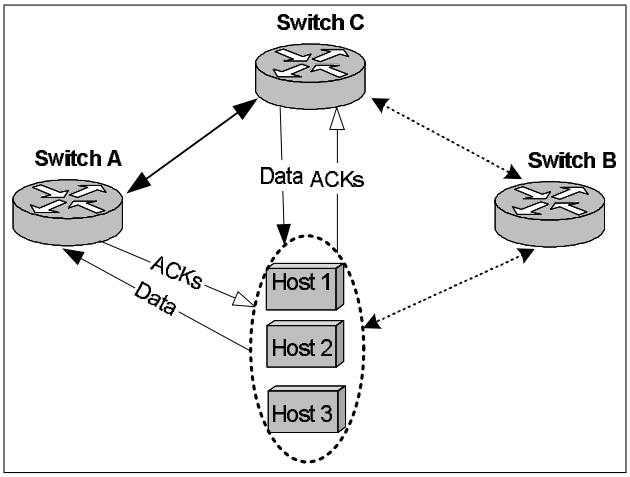


Figure 3.3: State after failure of link between switch A and switch B

link between them fails, the system stops. 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 the virtual sequencer. The spanning-tree protocol (*STP*) [58] is used to prevent loops. A dedicated IP multicast group is associated with each virtual sequencer. This solution allows building a system with f+2 switches, where f is the maximum number of tolerated switch/link failures. Figure 3.2 demonstrates a network that is able to tolerate failure of a switch or of a link between switches. Figure 3.3 shows the state of the network, after a failure of the link between switches A and B.

3.6 Performance

This section presents the results of the experiments performed to evaluate the above-described 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 was 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 at every configurable time interval. 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.

3.6.1 Theoretical bounds

It is important to observe that, regardless of the algorithm used to achieve the TO of messages, there are other system factors that limit the overall ordering performance. One of the bottlenecks that we encountered resulted from the PCI bus performance. In [104] 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 Gb/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 process in our experiments. There are various studies on PCI bus performance, e.g. [73], which suggest several benchmarks and techniques for tuning. It will be shown later in the chapter that our solution approximates the theoretical and experimental upper bounds of PCI bus. In future work, we plan to evaluate our architecture over PCI Express whose throughput is higher and thus is to yield significantly better performance.

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 processes were both senders and receivers (all-to-all configurations), and those in which the sets of senders and receivers

processes 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 3.1: Throughput and Latency for all-to-all configuration

were disjoint. It is important to note that for some configurations, such as the all-to-all configuration and those with 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 caused by the PCI bus limitations occurred on a node. The value serving to limit the traffic was selected by measuring the maximum achievable throughput for each setting. The main benefit of using traffic shaping is the limit it imposes on traffic bursts that were found to be the major cause of packet drops in our experiments.

3.6.1.1 All-to-all Configurations

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

Table 3.1 presents throughput and latency measurements for all-to-all configurations, along with the corresponding confidence intervals (shown in parentheses). The processes 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 between transmission of message by a sender and its delivery by the network back to the sender. UTO Latency is defined as the time that 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 processes that participated in this experiment increases from 3 to 5. As presented in Table 3.1, the achieved throughput increases with the number of participating processes. This is accounted for by the PCI bus behavior (See Section 3.6.1). Since each node both sends and receives data, the load on the PCI is high, and the limitation is the boundary on the total throughput that can go through the PCI bus. As the number of processes grows, the amount of data each individual process

	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 3.2: 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 3.3: UTO Latency (ms) for different configurations

can send decreases. When a process sends less data, the PCI bus enables it to receive more data. The nonlinearity 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.

3.6.1.2 Disjoint Groups of Senders and Receivers

Table 3.2 presents the performance results of throughput measurements for disjoint sets of processes. We used from two to five processes 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 processes. Possible solutions to this problem are discussed in Section 3.7.

Table 3.3 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 latency. The explanation is similar to that for the throughput measurement experiments, and it is 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 process either sends or receives packets, the utilization of the PCI bus and other system components is different from the case when a process acts as both a sender and a receiver. For this reason, the results presented in this section cannot be compared with those described above.

3.6.2 Tradeoffs of Latency vs. Throughput

In this section, we discuss the impact 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. In contrast with the previous scheme when the application sent as much as the flow control allowed, this scheme was implemented by a benchmark application that generated a configurable amount of data.

3.6.2.1 All-to-all Configuration

In this section, all-to-all configuration is considered. Figure 3.4 shows the latencies for the 5-process 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, 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,

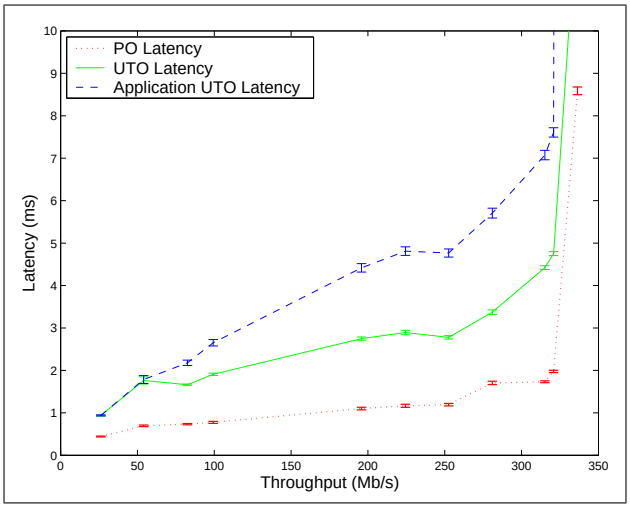


Figure 3.4: Latency vs. Throughput (all-to-all configuration)

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 by 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.

3.6.2.2 Large Packet Sizes

Figure 3.5 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, in case of jumbo frames, a 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 the system is relatively free, the high transmission latency of jumbo frames dominates; in other words, the time for putting a jumbo frame

The latest versions of Linux support $1ms$ granularity

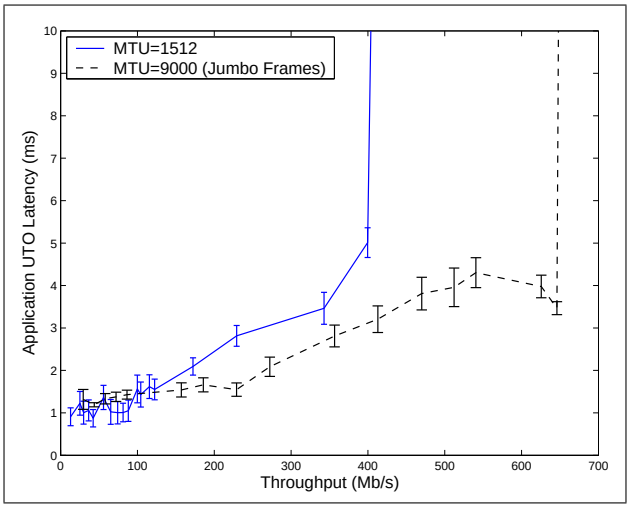


Figure 3.5: Latency vs. Throughput for different MTU sizes

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.

3.6.2.3 Packet aggregation

The experiment evaluated the effect of using the packet aggregation algorithm described in 3.4.2.1. Figure 3.6 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 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.

Comparing corresponding curves in Figures 3.5 and 3.6, one can see that packet aggregating causes a higher latency and a lower maximum achievable throughput. It could be explained by the amount of CPU resources spent on aggregating the messages.

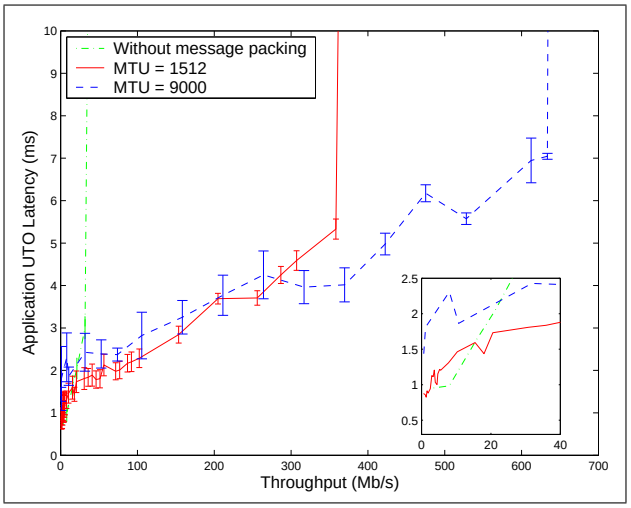


Figure 3.6: Packet aggregation (the low throughput area is extended)

3.6.3 Comparisons with previous works

There are only few papers that evaluate performance of TO algorithms over real networks. The rapid advancement of networking technology in recent years often makes the comparison irrelevant. For example, [49] presented performance evaluations of several TO algorithms. However, the measurements were carried out on a shared 10 Mb/s Ethernet network, which is 100 times slower than Gigabit Ethernet which is widely deployed today.

In the experiment described below, we compared the performance of our system with results of an algorithm based on weak ordering oracles ([82], described in Section 3.5), and of an algorithm based on failure detectors [27]. 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 to 3. The experiments in [82] were performed at 4 processes for weak ordering oracles and at 3 processes for the algorithm based on failure detectors. In our experiments, we used 4 processes. 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 [82] 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

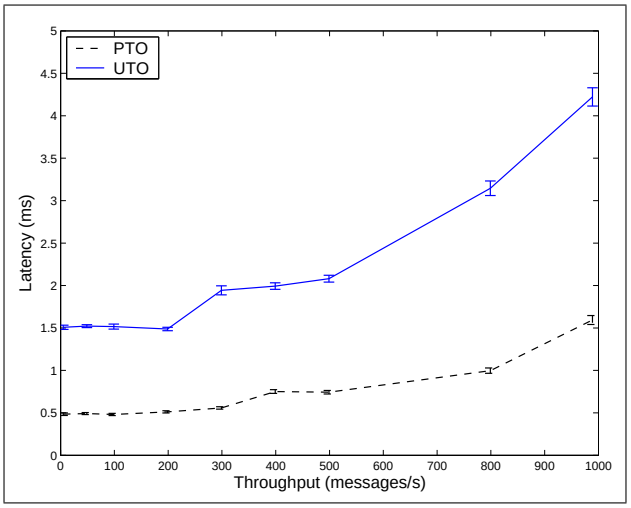


Figure 3.7: Number of ordered messages/s in 100Mb/s network.

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 within the range of 1.5-2ms, and for small timeout ($2ms$) the latency was within the range of 8-10ms.

Figure 3.7 presents the results of our experiments in 100Mb/s network and shows that the throughput of about 1000 messages/sec was achieved. The throughput of 300 messages/sec induces the PO latency of about $0.7ms$, and the UTO latency was within the range of 1.7-2.2ms. 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. [82], for a high throughput we reach lower latency. The reason is that in our system, the order is not disrupted even if a message m is lost, as losses happen mostly in switch A (see Figure 3.1). 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 that in a one-switch topology. Theoretically, the propagation time in a Gigabit network over a single link is $\frac{1500*8}{10^9}$=0.012ms, 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 0.021ms. We performed two experimental measurements of propagation time. In the first experiment, the ping utility was used to

measure the latency of 1500-size packet, and $0.05ms$ propagation time was obtained in both topologies. In the second experiment, we 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 the two-switch topology, without significantly increasing the latency, allows to predict message order with much higher probability!

3.7 Scalability

The performance evaluation presented above was carried out only for up to five processes. This evaluation proves that the architecture can be useful in small storage systems. The scalability issues addressed in this section show that the architecture is also applicable for systems consisting of dozens of processes.

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 [82], an algorithm was proposed that reduces the number of ACKs required to deliver a message. This approach can be further improved by using recently introduced NICs [43] 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 an efficient flow control for big number of senders is a more serious challenge. In future work, we are going to explore hardware flow control [60]. The main idea is to slow down switch B (see Figure 3.1) when the number of free buffers in a receiver (R) is below a threshold. As a result, switch B starts accumulating messages. In order to prevent switch B from dropping message, R also requires that the senders slowdown as well.

The number of ports in the switches is an important parameter that may limit the system's scalability. The simplest way to expand the two-switch network is to use trees of switches. Figure 3.8 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 process belongs to both groups, i.e. the senders and the receivers, it is connected to the two intermediate

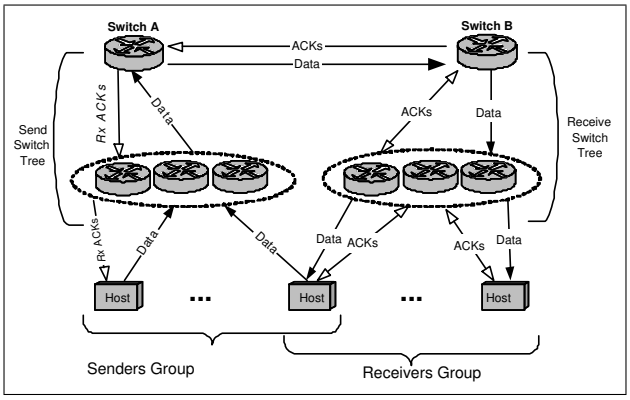


Figure 3.8: Expanded topology

switches which are connected to switches A and B, respectively. In this topology, the link between switches A and B continues to serve as the virtual sequencer. The path traversed by each message is longer, but, as shown above, the propagation time is very short.

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 the optimum one, as we force each process 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. Since 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 the host may stay away from participating in all the groups.

If the link between the switches is near saturation, one can use the new 10 Gb/s standard which is supposed to be soon available, for uplinks connecting the two switches. Another option that has already been implemented to increase throughput between two network components is trunk [59]. 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 in the way described in Section 3.5. It is noteworthy that all total ordering protocols are not able to order messages at a rate that is close to the link bandwidth. Thus, if the link between the switches is used only for ordering, it will not be saturated, and will not turn into a bottleneck but rather will remain an efficient ordering tool.

Part II

Toward Efficient Total Order in WAN

Chapter 4

TCP-Friendly Many-to-Many End-to-End Congestion Control*

Congestion control is a fundamental building block that enables WAN-deployable applications to function in the Internet environment. Many applications of this kind involve a group of processes that are to collaborate by communicating via multicast. There are several middleware implementations that offer group communication services with a variety of different semantics ([4, 5, 34, 55, 74, 100]). In the current chapter, we address the issue of TCP-friendly congestion control mechanism for group communication middleware. The importance of having a TCP-friendly mechanism stems from the necessity for an application to co-exist in the Internet with a variety of concurrent TCP sessions generated by other applications.

The specific group communication middleware that we focus on is *Xpand* see 4.1.1 and [8] Wide Area Network (WAN) Group Communication System (GCS). However, our results are applicable, with some adaptation, to other middleware systems as well. The main objective of the Xpand design is to address the needs of a wide spectrum of collaborative WAN applications without compromising the semantics of traditional GCSs, namely, group abstraction, group membership monitoring and reliable multicast. The services we exploited in order to add congestion control to Xpand were group membership and the ability to detect message losses which are inherent to any reliable multicast scheme.

*This chapter is based on a paper by T. Anker, D.Dolev, I. Shnayderman and I. Sukhov [15].

Xpand and similar group communication systems typically use UDP for all their communication needs, which allows them to benefit from the native IP multicast infrastructure for message delivery. For this reason, Xpand lacks the TCP built-in congestion and flow control mechanisms, which may lead to unfair bandwidth share, unstable transmission rate or even severe network congestion in WAN environment. Moreover, using TCP congestion control mechanism over UDP would neither scale nor provide the optimal results in a GCS.

In order to overcome these drawbacks in our study, a congestion control mechanism was added to Xpand, based on the "TCP-friendly rate control protocol (TFRC)" [47], with modifications that extend it to a many-to-many framework and to general message traffic, without focusing on multi-media applications only. TFRC is an equation-based unicast congestion control mechanism in which the equation derived from a model of TCP long-term throughput ([78]) is used to limit the sender's transmission rate. We have extended the TFRC equation-based approach from unicast to many-to-many message exchange, where every member can be either a sender, or a receiver, or both.

A recent paper ([107]) that addresses a similar problem focuses on directly expanding TFRC to deal with multicast. The main difference between the two mechanisms is in the method used to determine the slowest receiver in a scalable manner. TFMCC uses the original randomized feedback cancelation scheme, while in Xpand we take the advantage of the built-in hierarchy, thus using a more natural approach to achieve scalability in multi-cluster based system. This approach enables building a simpler feedback mechanism without feedback separation, as well as a simpler round-trip time estimation.

In Xpand, the feedback is aggregated based on the built-in hierarchical structure, and a sender is explicitly provided with an acceptable transmission rate from each remote receiving LAN.

Xpand was implemented and tested over the WAN environment described in [105]. Xpand not only utilizes a congestion control mechanism, but also contains a built-in flow control mechanism that follows the standard techniques described in the literature ([2, 45, 46]). The relationships between Xpand's components and its environment are shown in Fig. 4.1. The congestion control mechanism is integrated within Xpand's core module.

The measurements performed in this study validated the implemented congestion control mechanism. The results showed that our extended TFRC model is applicable to WAN, as well as to many-to-many communication patterns. The resulting behavior is TCP-friendly, and fairness is achieved among multiple senders sharing resources.

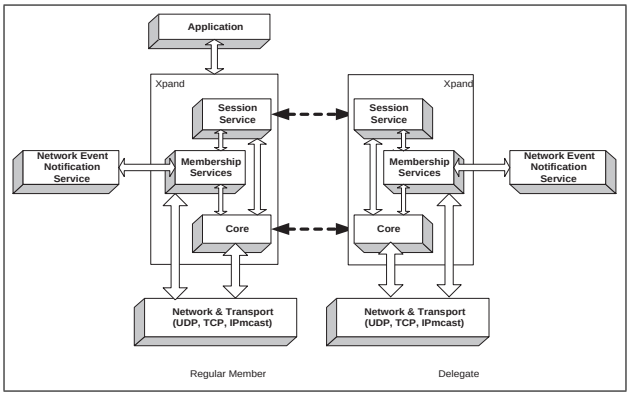


Figure 4.1: Xpand's layer structure

4.1 Environment

4.1.1 Xpand

The Xpand group communication system is a middleware for distributed many-to-many applications and is fully described in [8]. Here we present only the essence of the system and the relevant assumptions. Processes participating in Xpand are grouped into clusters, so that all members of each cluster belong to the same LAN. The clusters are spread over WAN. We distinguish between two types of processes: *regular* and *delegate*. Regular processes run the user's application. In each LAN, the delegate is a designated daemon that serves as a representative of its LAN to all the other Xpand clusters. Although a LAN delegate is replicated for fault-tolerance purposes, only a single delegate is active at each particular moment. This delegate is called an *active delegate*. All the other LAN delegates, called *backup delegates*, serve as warm backups of the active delegate. In the context of this thesis, we refer only to a single delegate per LAN and ignore the issue of replacing a failed delegate. The relationship between Xpand's components and its environment is shown in Fig. 4.1.

4.1.1.1 Application Layer Multicast

Xpand's reliable multicast service relies on the availability of a many-to-many communication substrate. The network service that supports many-to-many communication in IP networks is IP multicast. There is a limitation on using IP multicast, since it is only partially deployed in various networks, which creates isolated regions supporting IP multicast.

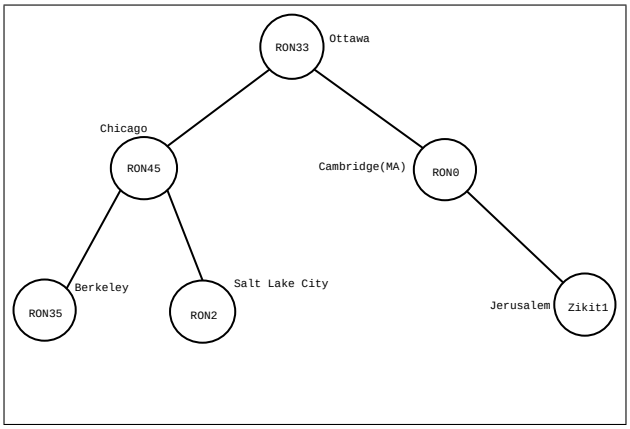


Figure 4.2: ALM Layout

In order to overcome this difficulty and to "bridge" regions supporting IP multicast, Xpand introduces a layer called *Application Layer Multicast* (ALM), which is used only as an interim solution. This layer constructs a message distribution tree incorporating delegate nodes, each node being chosen from a region that supports a native multicast. The tree construction is performed automatically and is beyond the scope of this study (see Chapter 5 and [8]).

4.1.2 RON Testbed Environment

In our WAN experiments, we used the Netbed's RON ([105]) wide-area nodes. The tests involved 6 sites, 5 located in North America and one in Jerusalem, Israel. The nodes are Celeron/733 or PentiumIII/1.1G machines running FreeBSD 4.7 / Linux 2.4 operating systems, connected via commodity Internet and Internet2. The links have diverse bandwidths, delays and packet loss rates (see Section 4.3.4). Since there is no IP multicast among RON machines, Xpand builds an ALM network over the nodes. The ALM tree created for our specific set of RON nodes is described in Fig. 4.2.

4.2 Xpand Flow/Congestion Control Mechanism Design

The design of Xpand congestion/flow control mechanisms takes advantage of Xpand's hierarchical structure. In Xpand, both senders and receivers from each LAN are represented by the LAN's delegate. We designed many-to-many single-rate mechanism that is TCP-friendly, in which the delegates of the senders cooperate with the delegates of the receivers in order to adjust the message flow to be TCP-friendly.

To introduce the congestion control mechanism into Xpand, some parameters characterizing the network are measured by the delegates and used by the senders in order to adjust their transmission rate in accordance with the network conditions. In [54], a message flow is defined to be TCP-friendly if under the same conditions its sending rate is within a factor of two of the sending rate of TCP flow. To achieve that, Xpand must be able to estimate the appropriate TCP rate. To reach that goal, we use Eq. 4.1 from [78] characterizing TCP steady state throughput as a function of packet loss rate, estimated round trip time and TCP retransmission timeout:

$$T_{TCP} \approx \frac{s}{RTT\sqrt{\frac{2bp}{3}} + RTO \min\left(1, \sqrt{\frac{27bp}{8}}\right) p\left(1 + 32p^2\right)} \ , \tag{4.1}$$

where T is the desired transmission rate in bits/sec; s is the average packet size in bits; RTT is the estimated round trip time in seconds; $p \in [0, 1]$ is the loss event rate; $RTO = 4RTT$ is the TCP retransmission timeout value in seconds recommended in [54], and $b = 2$ is the number of packets acknowledged by a single TCP acknowledgement.

In order to use this equation, we designed mechanisms for RTT estimation and for loss rate measurement. These parameters are to be continuously calculated and translated into an acceptable rate for each receiving delegate.

While TCP is unicast, our model calls for many sending processes and several sending delegates. The congestion control mechanism is used to allocate an aggregate transmission rate for each LAN and, within it, to estimate each local sender application transmission rate. Afterwards, each sending delegate is to fairly distribute the aggregate rate among its local senders.

4.2.1 Design Decisions

Xpand design distributes the load of calculating the congestion parameters between senders' delegate and receivers' delegates. Each calculation is performed at the optimum location, as detailed below.

RTT Estimation: performed by the sender's delegate for each <remote delegate, group> pair. The RTT estimation does not require synchronized clocks.

Loss Rate Estimation: Since in most cases there are many more receivers than senders, it is logical to distribute the loss rate calculations among the receivers. The hierarchical structure requires that the calculation be done at the delegates. It appears reasonable to calculate the loss rate at the receivers'

delegates which have more accurate information on message losses. The loss rate is estimated for a <receiver delegate, sender delegate, group> tuple.

Loss rate calculations require RTT estimation, which is performed by senders' delegate and passed to receivers' delegates. As a result, a receiver's delegate gets RTT calculated half RTT earlier, which may slow down the responsiveness of the scheme. This inaccuracy is smoothed by using the weighed RTT average.

We assume that RTT and the calculated loss rate are uniform for all receivers belonging to the same group within the same LAN. This is a safe assumption since message loss and RTT in LAN are negligible, compared to those in WAN.

Local sender's detection. The senders' delegate needs to identify potential senders in its LAN, as well as their sending rate demand.

Local sender's rate allocation. The senders' delegate their needs to distribute the aggregate transmission rate among its local senders. We use the "Max-Min fairness" ([22]) criteria to allocate the rates.

Local sender's traffic shaping. We use a token-bucket-like mechanism in order to enable the sender to conform to the assigned rate.

"Slow start" mechanism. We incorporate a slow start mechanism into the local sender's rate limit mechanism, so that it would be TCP-friendly even prior to a loss event.

Single LAN group. The congestion control mechanism is able to deal with a special case when all group members reside on the same LAN. Due to the lack of space, we do not present this aspect in detail.

Issues with Multicast Delivery Tree: There are several ways to build a multicast distribution tree in the Internet environment, using different multicast routing protocols. As we assume that multicast trees might be different for different groups, we measure RTT and loss rate for each group.

4.3 The Congestion Control Mechanism Description

In this section, the general algorithm flow is presented, followed by a detailed description of each step. While the mechanism is designed for many-to-many framework, the algorithm is presented from a single source LAN perspective and for a single group (G). The source LAN delegate (SD) communicates with the delegates (RD) of LANs that contain receivers for the given group. Every delegate implements the

Congestion control among multiple groups is achieved implicitly.

same mechanism and thus imposes congestion control over all the senders and the receivers. At the same time, Xpand message flows are friendly to TCP flows that use the same links.

The basic steps of the algorithm are as follows:

1. *RTT*s are computed at the senders' delegate per receiver's delegate and per multicast group and afterwards passed to the receiver's delegate;

2. Loss rate is measured at every receiver's delegate per sending delegate and group <SD, RD, G>. In these measurements, the receiver's delegate uses the *RTT* estimation obtained from SD;

3. The acceptable receiving rate is calculated by RDs using the measured loss rate and the *RTT*. RD sends its acceptable rate to the corresponding SD;

4. SD obtains the acceptable (restricting) rates reported by all remote delegates and chooses the minimal value to be the aggregate transmission rate for the senders belonging to the group within its LAN;

5. SD adjusts the local senders' transmission rate according to the aggregate transmission rate.

4.3.1 RTT Estimation by Senders' Delegate

For a given group G, *RTT* is measured by SD for every LAN that contains members of G. Intra-LAN delays are negligible in WAN environment. We assume that all the messages flowing from the sending LAN to each receiving LAN (targeted for the same group) traverse the same route. We assume that the control traffic from RD to SD also takes a steady route, which may be different from SD to RD route. As a consequence, all G senders within SD LAN have actually the same *RTT* to RD. Thus, for each group we actually measure *RTT*s from SD to RD. The algorithm is straightforward, as can be seen in Fig. 4.3.

Note: *RTT* is ***not*** updated using ***retransmitted*** messages, as the original and the retransmitted messages have the same sequence number. This scheme allows *RTT* variations to change the transmission rate within approximately one round trip time, which is important for the scheme responsiveness.

The RTT Calculation:

A sender multicasts packet P to the entire group G;

Upon receiving P, SD records P's arrival time T_{out};
/* T_{out} is an approximation of the sending time */

Upon receiving P, RD records P's arrival time T_{arr};

RD sends ACK for P to SD at time T_{ack} containing $\Delta = T_{ack} - T_{arr}$;

Upon receiving the ACK for P at time T_{now}, SD calculates $RTT = (T_{now} - T_{out}) - \Delta$;

Figure 4.3: RTT estimation at the sender's delegate

4.3.2 Receiver Loss Rate Estimation

Loss rate is measured by a receiver's delegate (RD). Regular receivers do not participate in loss measurements. RD may receive multiple flows from the same sending LAN for a group. All such flows are aggregated and considered to be a single meta-flow. This is done to improve the statistics of message loss rate and to address scalability issues.

A packet loss in our implementation is detected when the sequence number of a received message within a specific flow is out of the order. Since we assume packets to traverse the same multicast (unicast) delivery path, packet reordering is infrequent and has no significant influence on the overall picture.

The model in [78] requires that two loss events be separated by at least RTT seconds to be statistically independent. A loss event starts on a packet loss and continues RTT seconds after it. To determine whether a lost packet should start a new loss event or be considered as a part of a current loss event, we compare the approximated time stamp of a lost packet with the time associated to the beginning of the last lost event. To approximate the time stamp of a lost packet, we use the suggested interpolation to infer its nominal "arrival time" (see [47, 54] for details). In the calculation that follows, we ignore packets that arrive or are lost within RTT seconds of the time associated to the lost packet that started the loss event.

In order to compute the loss rate, we need to count the number of packets contiguously received between loss events. If a loss event is determined to have started at time T_1 and the next loss event started at time T_2, the number of packets between the loss events is the total number of packets in all n

In TFRC ([47, 54]) the term "inter-loss interval" is used, defined as the interval from the beginning of a loss event until the beginning of the next loss event.

flows that have arrived within period $[T_1 + RTT,\ T_2)$.

The Xpand approach matches the model presented in [78], whereas the original specification ([47, 54]) also counts packets that arrive within the first RTT seconds of a loss event, thus underestimating the actual loss rate.

To calculate the loss rate p, we first calculate I_{mean}, i.e. the average number of packets received between loss events. This is done by calculating the moving average over the past average I_{past} and the most recent estimation (with a quotient 0.5).

$$I_{\text{mean}} = \frac{1}{2} I_{past} + \frac{1}{2} I_{\text{sample}} \ , \tag{4.2}$$

where I_{sample} is the number of packets that have arrived since the end of the last loss event. At the beginning of a new loss event I_{mean} becomes I_{past}. Equation 4.2 presents the average number of packets once a new event takes place. Immediately after a loss event, the value of I_{sample} is smaller than I_{past}, which results in an inaccurate value of I_{mean}. To deal with this undesirable effect, we do not update I_{mean} until either a new loss event arrives, or $I_{\text{sample}} \geq I_{past}$.

The quotient is chosen to provide responsiveness, on the one hand, and to filter out samples that are too far from the average value due to inaccuracy of measurements, on the other hand. One of the original goals of [47, 54] was to achieve congestion control that is TCP-friendly, while allowing traffic rate that is smoother than that of TCP. This property is important for various applications, e.g. multimedia. Since for GCS the smoothness is not an important property, we used a simpler equation (Eq. 4.2) that produces a TCP-friendly rate change response, though not that smooth.

RD computes the loss rate ($p = 1/I_{mean}$). Using this value and the RTT, RD calculates the acceptable receiving rate (Eq. 4.1) and sends it to SD.

SD computes the updated aggregate rate by taking the minimum over all the acceptable receiving rates obtained from the relevant RDs. It then divides this rate among its local senders and notifies them about their new sending rate (See Section 4.3.3). Each local sender uses a token-bucket-like mechanism in order to shape its traffic to comply with its sending rate.

The two-way communication between SD and its local sender regarding sending rate allocation is conducted via a reliable channel and is beyond the scope of this thesis.

4.3.3 Multiple Senders in LAN

As was noted above, we determine the transmission rate per group, as we assume the possibility of different multicast distribution trees for different groups. Given an aggregate transmission rate for a particular group, there is a problem of dividing the aggregate group transmission rate among multiple senders residing on the same LAN. This is the task of the senders' delegate to distribute the transmission rate among local regular senders. The delegate uses a "Max-Min fairness" ([22]) criteria to ensure a fair rate allocation.

4.3.3.1 Determining local senders' rate demands

We describe here a scheme that enables SD to learn its local senders' demands for transmission rates to be used as an input to the fair rate allocation scheme.

The demand evaluation scheme is to be responsive enough to avoid under-utilization of the resource (transmission rate), e.g. in case when one sender receives a rate higher than it actually needs, whereas the other one receives a rate lower than it needs.

Application transmission rate may change more frequently than the aggregate rate, since the latter is computed infrequently and is smoothed, as opposed to the bursty traffic pattern of an individual sending application.

Thus, SD must sample its local senders' demands much more frequently than it receives aggregate rate updates. Each time a new vector of senders' demands is collected, the delegate recomputes the sending rate allocation.

Each a local sender determines its application demand by measuring the rate at which Xpand's sending window fills up and gets drained. To avoid unnecessary fluctuations when evaluating a transmission demand, the sender uses weighed "history" of its window's occupancy. To provide fast responsiveness, the weight of the "history" must be lower than that of more recent values.

When the window is full, the local sender cannot measure its application's transmission demand, as the application is blocked. In this case, a special approach is taken to estimate transmission rate demand. There is no point requesting a larger rate from SD if the last requested rate has not been complied with. Since SD uses the Max-Min fairness criteria, increasing the requested rate will not increase the allocated

The sender can explicitly ask its delegate to send an update if its local application transmission rate demands drastic changes

rate. If the delegate complies with the last rate request and the window is still full, the sender doubles its requested rate demand. This process continues until the window opens up.

When a new sender starts transmitting, it is always permitted to start sending at some predefined initial rate. The delegate recognizes the presence of the new sender and takes it into account by recalculating the sending rate allocation.

4.3.4 Implementation Details

Xpand has a complementary built-in flow control mechanism, which is used as long as no congestion is detected over the WAN. Once congestion is detected, the above described TCP-friendly congestion control mechanism takes over. If no remote receiver exists, i.e. all the members reside within a single LAN, the TCP-like window congestion avoidance mechanism is used.

We also use a slow-start mechanism for some specific cases, e.g. if no remote members exist and a new sender starts its transmission. Another case is when no remote receiver has detected any loss event, thus the acceptable receiving rate that is sent to the SD is infinite. Obviously, when a loss is detected, the sending rate is immediately updated.

4.4 Performance Results

The congestion control was tested in multi-continent (WAN) setting of RON testbed Network. The tests included six sites: Zikit1 (Jerusalem), RON0 (Boston), RON2 (Salt Lake City), RON33 (Ottawa), RON35 (Berkeley), RON45 (Chicago). All measurements were carried out within the same time-window every day. The time-window was chosen to be early morning time in the North America, which is mid-day in Israel. As there is no IP multicast connectivity among these sites, a propriety Application Layer Multicast mechanism was used ([8]). The obtained message distribution tree is presented in Fig. 4.2. RON33 was selected to be the tree root by the algorithm used in the dynamic ALM tree construction.

In tables 4.1, 4.2 and 4.3, the columns represent senders and the rows represent receivers. These numbers were collected during the first test (see below).

Table 4.1 presents the median round trip time (RTT) among the sites as it was measured by the senders. The messages from the sender to the receiver were sent along the ALM tree, and the ACKs were sent back to the sender via unicast.

Table 4.1: Round trip time (RTT) (ms)

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1		170.92	282.15	205.25	277.99	213.99
RON0	170.62		120.18	51.67	122.20	62.82
RON2	260.60	91.95		57.65	77.15	57.87
RON33	236.30	50.00	85.70		85.59	26.68
RON35	299.13	121.32	96.58	76.23		59.80
RON45	242.47	62.29	58.75	25.27	59.64	

Table 4.2: Loss rate (%)

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1	0.00	0.00	2.90	0.00	7.63	2.33
RON0	0.33	0.00	3.42	0.00	7.40	2.47
RON2	4.20	2.87	0.00	3.39	6.35	1.96
RON33	0.33	0.00	3.31	0.00	7.84	2.67
RON35	4.80	3.26	3.34	4.36	0.00	1.50
RON45	1.15	0.82	0.71	0.63	4.85	0.00

Table 4.2 presents the median loss ratio as calculated by the receivers. One can notice that loss ratio increases along the ALM paths, as the number of ALM overlay links increases.

Table 4.3 presents the theoretical rate calculated by Eq. 4.1. There are some blank cells in the table, since no message was lost on the corresponding links due to the fact that the rate was limited by another receiver.

In the following sections, we present performance measurements of the congestion control mechanism within Xpand system. The measurements validate the congestion control mechanism, its TCP-friendliness and its applicability to multicast in WAN.

4.4.1 Rate Restriction

In the first test, six members join the same group G. Each member in turn sends messages to G for 60 seconds, at the rate allowed by the congestion control mechanism. Zikit1 measures data arrival rate for each sender (including itself). Figure 4.4 presents the received rate measured by Zikit1 for each sender.

Table 4.3: TCP-friendly rate (Mb/s)

Site	Zikit1	RON0	RON2	RON33	RON35	RON45
Zikit1			0.16		0.07	0.24
RON0	0.96		0.33		0.16	0.80
RON2	0.13	0.49		0.69	0.30	1.02
RON33	0.69		0.48		0.22	1.79
RON35	0.10	0.34	0.42	0.43		1.17
RON45	0.34	1.60	1.84	4.58	0.50	

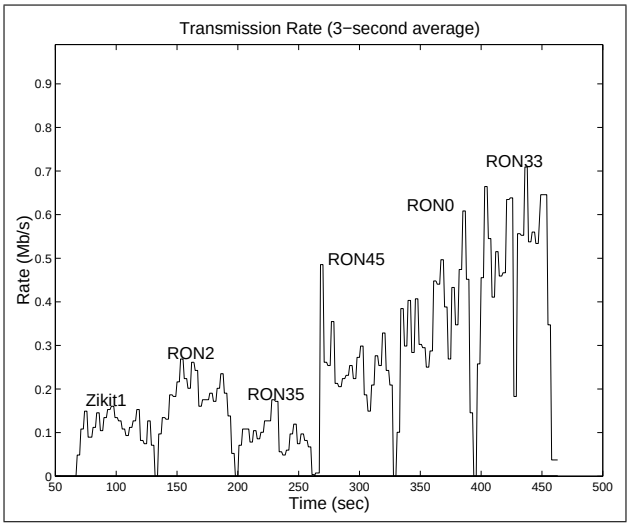
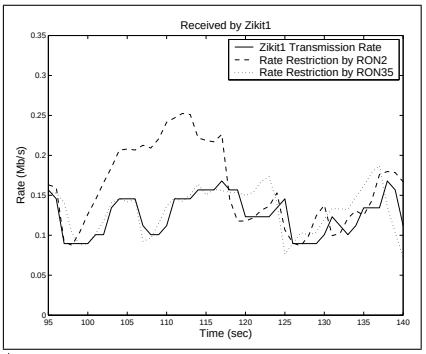
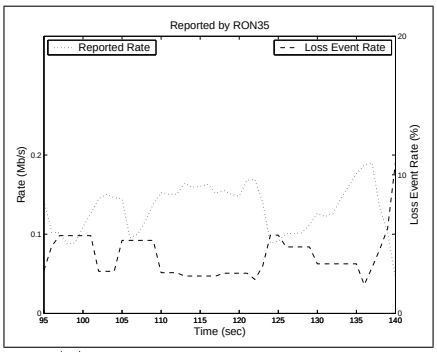


Figure 4.4: Sequential sending



(a) Zikit1 is limited by RON2 and RON35



(b) Reported rate vs. loss rate

Figure 4.5: Rate limitation factors

In order to evaluate the congestion control mechanism, we examined the performance of Zikit1. The variation in its rate is caused by the changes in rate forced by the slowest receiver.

Figure 4.5(a) shows the slowest receiver limiting Zikit1 sending rate. It is clear that the sending rate is limited by the rate reported by the slowest receiver. We present only the two slowest receivers that affect the sending rate, since other receivers reported a much higher receiving rate.

Figure 4.5(b) presents the loss rate and the reported rate calculated by RON35 for Zikit1 messages over the same time period. One can see that the dominating factor in the rate reported by the receiver is the loss rate, since the RTT remains relatively stable during this period.

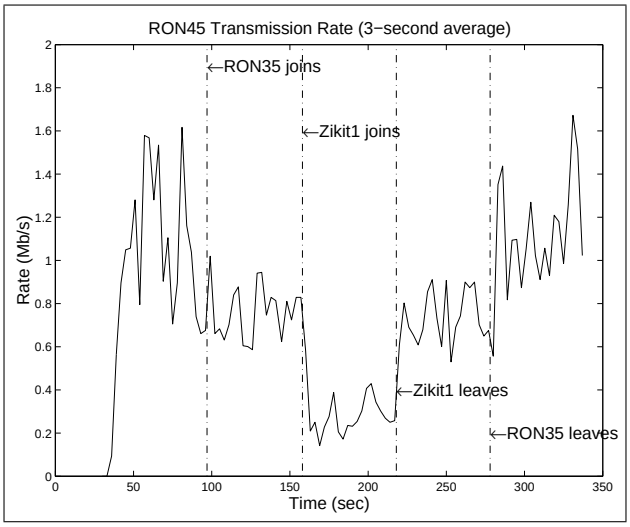


Figure 4.6: Responsiveness to membership changes

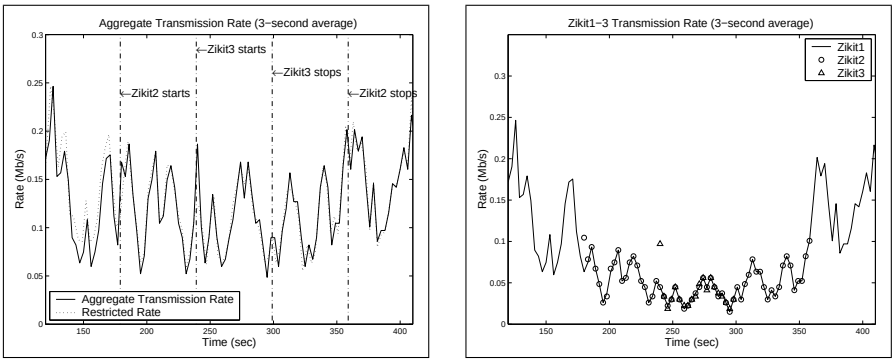


Figure 4.7: Aggregation of multiple senders and fairness

4.4.2 Rate Adaptation to Membership Changes

In this test, all RON machines except RON35 and Zikit1 were members of group G. The application running at RON45 sent as much traffic as it was allowed by other delegates. In Fig. 4.6 we see that when RON35 joins G, the sending rate of RON45 drops. Afterwards, when Zikit1 joins G, the rate again drops significantly. 60 seconds later, when Zikit1 leaves G, the rate increases and then increases again when RON35 leaves G.

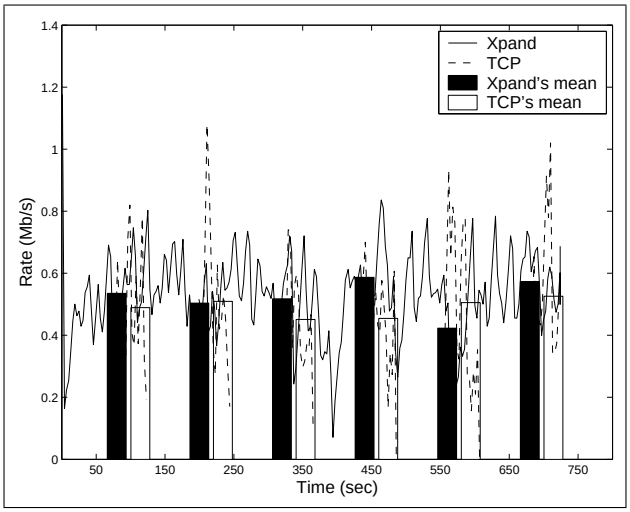


Figure 4.8: TCP-friendliness

4.4.3 Fairness Among Senders in a LAN

In Fig. 4.7(a) we see aggregate rates generated by Zikit1, Zikit2 and Zikit3 while all RON computers receive the messages (Zikit[1-3] all residing in the same LAN at the Hebrew University of Jerusalem). Zikit1 sends messages through the entire duration of the test, while Zikit2 and Zikit3 send messages for a shorter period (180 seconds and 60 seconds respectively). The graph also presents the rate by which Zikit's LAN was limited by the other delegates. One can see that the influence of the reported rate, which in turn depends on the loss rate, is much higher than the impact of the number of senders.

In Fig. 4.7(b), we present the achieved sending rate of three senders sharing the same aggregate transmission rate during the test. The results clearly show that Max-Min fairness is achieved. The measured results were averaged over 3-second intervals. For clarity, we chose a relatively large average interval to be shown in Fig. 4.7(b), while the results were similar on smaller scales.

4.4.4 TCP-friendliness

Figure 4.8 presents the TCP-friendliness test results. In this test, we ran six cycles by the same scenario: every cycle started with a 60-second Xpand session followed by a 60-second TCP and Xpand joint session.

In this test, only Zikit1 and RON35 were the group members. Both Xpand and TCP messages were sent by Zikit1. The achieved rate was averaged over a 3-second interval. The average rate over the periods when both flows shared the network resources, are also presented in the figure.

In all the cycles, the performance of Zikit1 showed that the bandwidth limited by our congestion control mechanism is comparable to that of TCP and, therefore, the message flow is TCP-friendly.

4.5 Conclusions and Future Work

Our study has proved that it is possible to create a many-to-many congestion control mechanism that is TCP-friendly and provides a built-in fairness among senders that share resources.

Future research needs to focus on improving the results, in particular, the responsiveness of the mechanism to message losses and group membership changes. In our current implementation, retransmissions were not considered to be a part of the sending rate, though a conservative approach was taken. We intend to find an appropriate way to incorporate retransmissions in the congestion control mechanism while allowing for heterogeneous (unicast and multicast) retransmission schemes.

Chapter 5

Ad Hoc Membership for Scalable Applications*

There are two major widely used approaches toward building distributed data-based applications and replicating objects. The first approach, known as Group Communication Systems (GCSs) [1], presents powerful building blocks for supporting consistency and fault-tolerance in distributed applications, whereas the Paxos [66] approach focuses on ordering actions among a group of servers. Implementing either of the approaches causes the system's performance to degrade significantly as group size and message transmission volume increase. These performance problems become more severe in wide-area network environments where message latency is often high and unpredictable. Only few implementations specifically address WAN environment, e.g., Spread [4], InterGroup [21] and *Xpand* [8]).

In this chapter, we present an ad hoc membership algorithm that is implemented in *Xpand* [8]. The membership services offered by Xpand are designed to be flexible. On the one hand, they can be used for maintaining a group of participants and potential participants of a group, by means of applications that do not need strong semantics.. On the other hand, while focusing on efficiency, these services allow to consistently provide stronger semantics for applications that require it.

The efficiency of the ad hoc membership is due to the separation it implies between message flow,

*This chapter is based on a paper by T. Anker, D.Dolev and I. Shnayderman [13].

For instance, these applications may not want to block or perform roll back actions per change in the system as GCS or Paxos require.

membership algorithms, and failure detection (Network Event Notification Service). Moreover, two separate reliable message dissemination services are used, one for control messages and the other for application messages.. This enables to remove reliability maintenance overhead from the critical path of delay-sensitive applications.

Within the ad hoc approach, the membership notifications serve only as *approximations* of the current group membership, without being synchronized with message stream. Continuous message reliability is guaranteed only among those group members that remain permanently alive and interconnected. This approach allows handling any number of simultaneous join/leave events concurrently, without waiting for the network or the group of members to stabilize. The traditional membership approaches require a certain time period of stability in order to consent on a new membership. As a result, they tend to limit the adoption of joining processes during periods of instability. The ad hoc approach provides stability of message flow against on-going membership changes, which is more suitable for large groups and for wide area networks where the probability of instability periods increases. The scalability of Xpand is accounted for by both its hierarchical architecture and the ad hoc membership approach.

The development of the ad hoc membership approach faces three challenges, interrelated to one another. (1) The architectural issue: selecting a channel preferable for a certain message- the more reliable "sequential" channel (a slow one), or the less reliable concurrent channel where a loss of some messages does not slow down the delivery of other messages. (2) The algorithmic issue: the dual channel architecture increases the asynchrony among various blocks that provide membership service and, as a result, produce conflicting race conditions that need to be controlled. (3) The design issue: designing a system that takes advantage of the ad-hoc membership, while still maintaining the ability to provide a stronger semantics for applications that require it.

The ad hoc approach best suits the requirements of collaborative networking applications, fault-tolerant applications that require weaker synchronization among a set of servers, one- to-many push applications (e.g., stock market monitoring) and the like. In Section 5.1.1, we present several examples of such applications.

It is possible to obtain the properties of an ad hoc membership using other WAN toolkits, like combining Spread [4] and PBcast [56], though the challenge remains of providing the properties we look for in an efficient and robust way. Moreover, Spread and similar toolkits assume a predefined global set of servers, whereas Xpand [8] does not require any pervious knowledge of the potential set of group

members.

The ad hoc membership is fully implemented within the Xpand middleware. Section 5.4 presents performance results of the implementation showing that membership changes have a negligible effect on the existing message flows.

5.1 Xpand Membership Service Model

Xpand membership algorithm is optimized to ensure the following properties:

Smooth-Join: A new member joins the group without affecting the current message flow in the group;

Fast-Join: Once a joiner is registered in its LAN, it can start emitting messages to the group;

Dynamic Membership: Allows processes to dynamically join and leave the group in a WAN setting.

Xpand membership algorithm achieves these three properties without compromising the following basic message delivery services of Xpand:

FIFO order: Any receiver that starts receiving messages from a given source will get all emitted messages in FIFO order from the moment it joins the message flow;

Gap-Free: Two processes that remain connected during consecutive membership changes continue to receive each other's messages without any gap in the message stream.

The above-listed membership services combined with Xpand's reliable multicast service can be used as a dynamic and reliable point-to-multipoint service layer, on top of which stronger semantic services can be built. For instance, a virtual synchrony layer can be implemented as an extension of the ad hoc membership service Moshe [63]. Another example is Paxos, which can be implemented on top of our ad hoc service by having the leader communicating to existing majorities via the ad hoc services to carry out the three phases of Paxos [66].

5.1.1 Implementation Issues

In order to enable maintaining of large groups of clients, Xpand is implemented using a two-level hierarchical architecture. Each LAN is represented by a delegate that coordinates the protocol activities of this LAN within the WAN group.

In the current implementation, we assume that the number of LANs with processes that emit messages to the group is of the order of several dozens. The number of processes per LAN is assumed to be of the order of a couple of hundreds. These limitations are imposed due to the need for maintaining state information per sender and per receiver. We can further improve the implementation by enabling a client to join either as a receiver-only, a sender-only, or as a full member (i.e. both as a sender and a receiver). Such an approach enables to increase the number of processes, while keeping a reasonable amount of state information. To increase scalability, the architecture is used for aggregation. A delegate can represent a collection of senders in its LAN, and the senders should not necessarily be explicit members of the WAN group. They will forward their messages to the delegate which will emit them to the WAN group.

We expect the above improvements to enable the ad hoc system to handle tens of thousands of clients.

The protocol is presented in this chapter as a collection of state machines within various processes. The actual implementation of the protocol closely followed the state machines. This method enabled us to easily detect protocol and implementation bugs that emerge in unforeseen transitions. We found out that by adding history (log) within every state machine we were able to drastically shorten the debugging time.

The ad hoc approach best suits the requirements of collaborative networking applications where the importance of service timeliness prevails over message reliability, and the consistency requirements are weaker than those of replicated database. For example, multimedia conferencing and distance learning applications benefit from the ad hoc GCS services in workspace sharing and parties coordination [33, 87].

Another example is a loosely coupled set of servers. For example, in the fault-tolerant video-on-demand service by Anker et al. [12], a GCS based on the ad hoc membership service can be used for video server fault-tolerance and QoS negotiation. The efficiency of handling membership changes by means of our protocol allows for smooth client hand-offs with smaller video-frame buffers at the client.

Yet another application is a case when multiple servers emit information to a loosely coupled set of clients. Each client wishes to receive the stream as soon as possible. For example, in stock market monitoring, application messages are generated at several centers located all over the world. The access to the information is critical, but there is no justification for blocking the stream of messages during network changes. In an on-going audio conference, a newcomer can benefit from Xpand in the sense

The effect of not using a virtual synchrony in this VoD implementation is that the client's machine may receive, in transient situations, duplications of video frames.

that Xpand will deliver a reliable multicast service from every session participant to the newcomer (and vice versa) right after the establishment of the corresponding new connections, without affecting the previously established connections.

5.2 The Environment Model

We assume that the message-passing environment is asynchronous, the processes communicate solely by exchanging messages, and there is no bound on message delivery time. Processes fail by crashing and may later recover, or may voluntarily choose joining or leaving. Communication links may fail and recover.

Xpand exploits the following underlying services:

- A network event notification service (Section 5.2.1), through which Xpand learns about the status of processes and links;
- An integral reliable FIFO communication layer within the network event notification service. It is assumed that notification events and messages delivered via this service are causally consistent;
- A reliable point-to-point service for control messages of the protocol;
- A simple reliable point-to-multipoint service *in a LAN* for control messages of the protocol.

The reliability of the above transport services implies that messages sent via them are eventually received, or the recipient will eventually be suspected by the network notification service. For efficiency reasons, all the reliable transport services are used only for control messages (i.e. for protocol messages, but not for user application messages).

5.2.1 Network Event Notification Service (NS)

Clients use the notification service to request joining or leaving the groups, or to get updated information regarding the group. The notification service accumulates and disseminates failure detection information along with information about these requests. The service is provided to clients by an interface that consists of the following basic functions:

This mechanism is not a substitute for the general services of Xpand, but rather a simple mechanism focused on guaranteeing reliability over a small number of messages exchanged among protocol participants in a single LAN.

NS_join(G) is a request by a client to make it a member of group G;

NS_leave(G) is a request to be removed from the membership of G;

NS_resolve(G) is a request to receive the current membership list (in a Resolve_Reply message/event) as it is reflected by the notification service;

NS_sendmsg(data, destination) is a request to reliably send a message (data) to a set of receivers (destination) via the notification service.

Clients of the notification service receive notifications via process events that indicate the type of the event and the data associated with it. The event associated with the reception of a message via NS (a message that was sent using the *NS_sendmsg* function) is called *NS Recvmsg*.

The notification service contains a failure detector module. Since we assume an asynchronous environment, the notification service is bound to be unreliable in some runs [28], which means it may wrongly suspect correct processes. Since we deal with a service that can be implemented in an asynchronous environment, we do not require the notification service be accurate. However, we assume that the notification service is always complete [28]. The overall liveness of Xpand does not depend on the notification service providing consistent sets, since it communicates with any connected set of clients. Congress [6] fulfills the requirements of the notification service. Other group communication membership services (cf. Spread [4]) can also provide similar notification service.

For the sake of simplicity, the membership algorithm is described in this chapter in the context of a single invocation of a process, which means that it may join and leave only once. In practice, an instance identifier per process distinguishes among different incarnations of the same process. The relationships between Xpand's components and its environment are shown in Figure 4.1.

5.3 Xpand's Ad Hoc Membership Algorithm

Xpand's membership algorithm deals with "views" of a group G. The view at p (meaning the view currently available to a specific process p's) is the current list of connected and alive members of G, as perceived by p. The maintenance of the view of G within p is based on an initial Resolve_Reply event received by p from NS. p updates its view by applying the information received in NS notifications, such as processes' joining or leaving.

Before describing the membership algorithm, we focus on the goals it aims to achieve. The algorithm

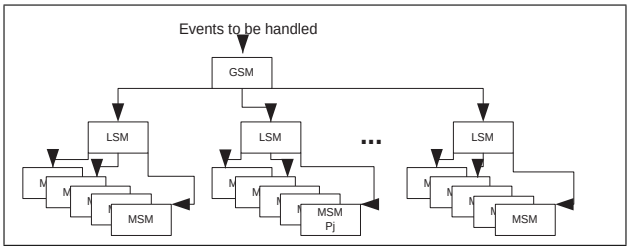


Figure 5.1: State Machine Flow

should allow a new member p_i to join as fast as possible, while maintaining the following properties:

- p_i will start receiving messages from any active member (sender) p_j as soon as possible (given that the sender is alive and transmitting messages).
- For each sender p_j, a new member p_i will receive messages sent by p_j in the same order they were sent by p_i (i.e., FIFO order will be maintained).
- Once p_i receives a message from a specific member, which remains alive, and will receive .within the same network partition as p_i, p_i, all the messages sent by that member from this message on without any gaps. The only situation in which gaps are allowed in the received message stream is when the source is declared disconnected by NS.

The first property is obtained if a new joining member is notified of the sequence number of the following message from any sending member without any unjustified delay. In order to guarantee the other two properties, there is no need for a global synchronization on sequence numbers that are distributed to any new member with respect to a specific member p_j. This means, for example, that if any two new members p_1 and p_2 are joining simultaneously, a sending member p_j can announce a sequence number $SeqN$ to p_1 and, after a short time interval, announce $SeqN+k$ to p_2, in case it had managed to transmit k messages during this interval of time.

The asynchrony makes it impossible to describe the protocol in the traditional manner. The protocol can be viewed as an embedded set of state machines executed at the regular member and at the delegate. The top state machine is the *group state machine (GSM)*. That state machine invokes multiple *LAN state machine (LSM)*, one per LAN that contain members in the current view. The LAN state machine invokes *member state machines (MSM)*, one per each member that is listed in the view and is residing within that specific LAN. Figure 5.1 presents the general flow of control among the different state machines.

Message Type	From	To	Message role
Force_Join	Regular member	Delegate	Causes Delegate to join a group
Start_Sync	Regular member	Delegate	Causes Delegate to reply with information about group members
Sync_Reply	Delegate	Regular member	Delegate's reply to the Sync_Reply (contains a list of members message sequence numbers as currently known to Delegate)
View_Id	Delegate	Delegate	Causes Receiver to reply with information about its **local** members
Cut_Request	Regular member	Delegate	Causes Delegate to reply with information about specific group members
Cut_Reply	Delegate	Delegate / Regular member	Delegate's reply to the View_Id / Cut_Request messages (contains a list of the requested members message sequence numbers as currently known to Delegate)

Figure 5.2: Protocol messages

The GSM receives events and forwards them to the appropriate LAN state machines, from which the events are applied to the corresponding MSMs. The events are handled concurrently by the relevant state machines.

For brevity, we included the details of only some of the state machines. We have chosen to present the group state machine of a regular member and the LAN state machine of a delegate. For the sake of simplicity, we removed the issue of handling of various timeouts from our presentation of the state machines.

The membership algorithm uses the set of messages described in Figure 5.2. All the messages in the figure are messages sent by either delegates or regular members. These messages are sent via the transport services (Section 5.2).

The ad hoc membership algorithm handles the joining/leaving event, as well as network partitions and network merges. The state machines respond to external events received from NS, as well as to control messages sent by other members of the group (via the corresponding state machines).

The simplified pseudo code of a regular member joining is shown in Figure 5.3. The code covers the state machine shown in Figure 5.5 and portions of other two state machines related to this specific case (see GSM-delegate and MSM-delegate in [14]). A regular joining member p_1 is notified about a new member p_2 in the group by receiving either a NS Join message or a Sync_Reply/Cut_Reply message (sent by its own delegate). In either case, p_1 initiates a new member state machine for p_2. In the former case, the MSM goes directly to "active" state, in which messages sent by p_2 will be processed.

The related states and state transitions in Figure 5.5 are indicated.
E.g., the NS_Join in the code is actually part of the GSM of the delegate.

```
            Invoke NS_join(G) (piggyback next data message SeqN);
            label wait for RR:
(1)         wait for NS Resolve_Reply msg (RR);
(1->2)      if RR includes local delegate {
(1->2)            NS_sendmsg(Start_Sync of G, local delegate);
(2)               wait for a Sync_Reply message;
                  allow the user application to send data messages
to G;
(2->3)            for each LAN listed in the Sync_Reply message
                        invoke the corresponding LSM;
(2->3)            for each new LSM
                        invoke the corresponding MSM;
(2->3)            for each new MSM {
                        extract sender message SeqN;
                        init sender data structure
                  }
            }
            otherwise {
(1->5)            build Force_Join message m for G;
(1->5)            NS_sendmsg(m, local delegate);
(5)               wait for local delegate to join G;
(5->1)            issue a NS_resolve(G);
(1)               goto wait for RR
            }
```

Figure 5.3: Highlights of Join Operation at Regular Member.

In the latter case, the NS notification regarding the joining of p_2 has not arrived yet. Thus, the MSM goes into "semi-active". In this state, p_1 waits for the proper NS Join message concerning p_2. When the NS Join message arrives, the MSM shifts into "active" state. This complication is caused by the asynchrony resulting from the separation of the membership algorithm within Xpand and the external NS mechanism.

To limit some of the potential race conditions, the Start_Sync message is sent via NS. This ensures that by the time a delegate receives this message via NS, it has already received and processed all the previous NS messages, especially those received by the regular member at the moment of sending this request. This doesn't cover the multi-join case that is created when several members join at once, some of them not yet having delegates in the group. In a more complicated case, we may face a situation when members are in the group, but their remote delegates are not, or a situation when the Sync_Reply doesn't include all the necessary information about them. The regular LSM (in [14]) copes with those situations. LSM essentially needs to identify the case and to wait for its resolution, while allowing the sender to begin sending its information. While waiting for the delegate of a LAN to join the group,

By the causality assumption of the reliable FIFO comm. layer within the NS.

```
      label init-group:
            A Force_Join for group G is received from a local member
            NS_join(G);
            wait for NS Resolve_Reply msg (RR);

            split RR into separate LAN lists;
            for each LAN list in RR {
                  invoke LAN state machine (Figure 5.6);
                  if remote delegate is NOT listed in the RR
                        wait for remote delegate to join G;

      label peer sync:
(1->2)            NS_sendmsg(View_Id msg, remote delegate);
(2)               wait for Cut_Reply msg corresponding to View_Id msg;
(2->3)            for each member listed in the Cut_Reply msg {
                        invoke the corresponding MSM and within it:
                                    extract member message SeqN;
                                    init member data structure;
                  };
(3)               put LSM into "active" state
            } /* for each LAN in the group... */
```

Figure 5.4: Highlights of First Join Operation at Delegate.

the member can process further NS messages for LANs on which it has the full information. Other NS messages need to be buffered and processed after the delegate joins and integrates within the group.

As regards the delegate part, a delegate joins a new group when its first local member joins it and "forces" the delegate to join also, by sending a Force_Join message. Upon receiving such a message, the delegate invokes a GSM for the relevant group. Upon receiving the resolve reply, the GSM invokes a set of LSMs which, in turn, invokes a set of MSMs per LSM. The delegate GSM and MSM appear in [14]. Below we discuss the delegate LSM (LAN State Machine).

Figure 5.4 shows the simplified pseudo-code of the delegate, executed upon receiving a Force_Join. The code covers the state machine in Figure 5.6 and *portions* of the other two state machines related to this specific case (see MSM and GSM of delegate in [14]). The Force_Join message causes the delegate to join a specific group. When the delegate joins the group and receives the resolve reply through NS, it needs to spawn LAN state machines per LAN which is represented in that resolve reply message. While the delegate is waiting for the resolve reply message, it may receive View_Id messages or Start_Sync

Full information here means acquiring message sequence numbers for each known member in the remote LAN.
The related states and state transitions in Figure 5.6 are indicated.
E.g., the NS_Join in the code is actually a part of the GSM of the delegate.

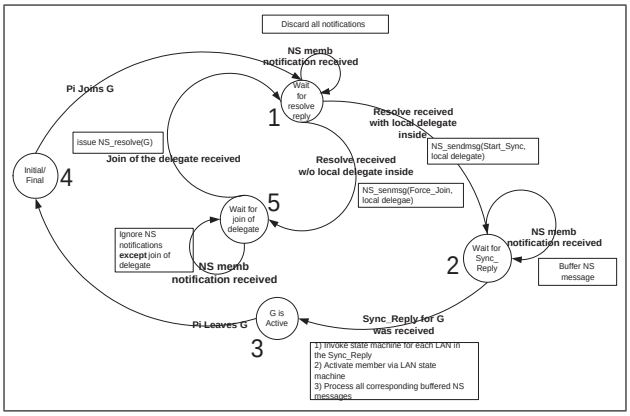


Figure 5.5: Group State Machine at Regular Member

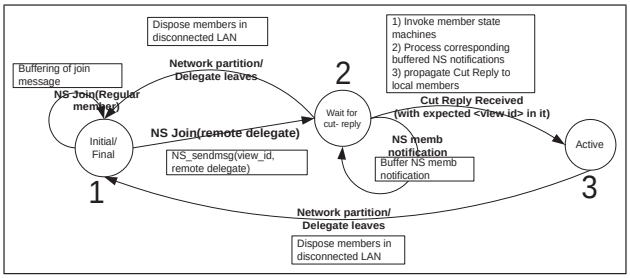


Figure 5.6: LAN State Machine at Delegate

messages via NS. Those messages will be buffered and processed later when the resolve reply is received.

When the delegate is already a member of a group and is notified about a new member of its own LAN (via an NS Join notification), it takes the new member message SeqN that is listed in NS Join notification and initializes the new member data structure.

A different case occurs when a remote member joins the group while its own delegate is not a member yet. In this case, the local delegate waits for the remote delegate to join via the NS. Once the remote delegate has joined, the local delegate performs the protocol in Figure 5.4 starting from peer sync label.

In the above description, we have dealt only with a common case from the overall protocol. A careful use of the NS channel enables us to cope with some of consistency problems. The full protocol is a combination of all the state machines being executed concurrently while responding to the arriving events. Due to the lack of space, we are not considering all the particular issues. The description of the state machines here and in [14] enables to identify some of those issues.

5.4 Implementation and Performance Results

To test the efficiency of the ad hoc membership algorithm and its implementation, we conducted several tests to measure the effect of membership changes on ongoing message flow. Specifically, we investigated its fast_join and smooth_join (Section 5.1) properties.

The ad hoc algorithm was tested in multi-continent (WAN) setting. The tests included three sites: The Hebrew University of Jerusalem in Israel (HUJI), the Massachusetts Institute of Technology (MIT) and the National Taiwan University (NTU). We had estimated the round trip times and the loss percentage between these sites, in order to have the characteristics of the network interconnecting them. The round trip times measured between MIT and HUJI and between MIT and NTU were both about 230ms. The round trip time between HUJI and NTU was about 390ms. The loss percentage was not as persistent as that of the round trip times, varying from 1 percent to 20 percent.

In all the three sites, the machines used were PC computers running the Linux operating system. As there is no IP multicast connectivity among these sites, a propriety Application Layer Multicast (ALM) mechanism was used [8]. The obtained message distribution tree is presented in Figure 5.7(a). HUJI was selected to be the tree root by the algorithm used in the dynamic ALM tree construction. In all the tests, the senders emitted a message every 100ms.

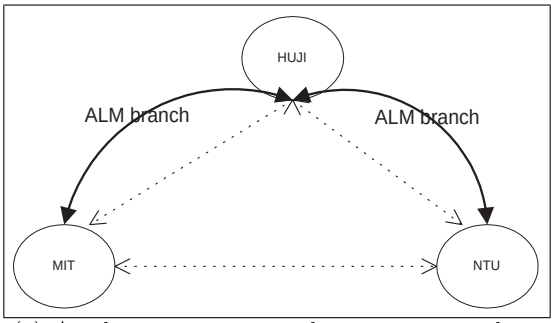
Fast_Join

In the first test, we measured the time it takes a new joiner to start receiving messages from active sources, the participants being 4 senders at HUJI (S1-S4) and one sender at NTU (S5). There were two joiners, one at MIT and one at HUJI (S6). Both joined the group during the test and started sending messages right after joining the group.

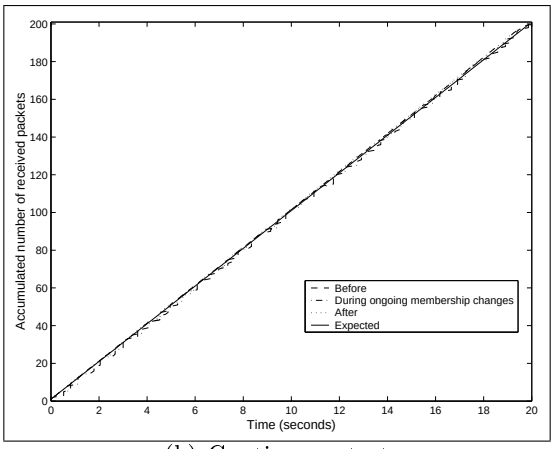
The graph in Figure 5.8(a) shows that once the joiner at MIT (which recorded the events) receives the proper event notification (Sync_Reply message), it begins receiving messages from all current senders within a short time (23ms). It takes the joiner longer time to begin receiving from the other joiner. This extra time is a result of the difference between the arrival of the Resolve_Reply notifications between MIT and HUJI.

To evaluate the impact of our results, it should be mentioned that in a similar scenario Moshe [63], the minimal time should be at least one and a half round trip, which amounts to at least 600ms.

In some cases, a higher loss percentage was observed when the interval between the ping (ICMP) messages was less then 100ms.



(a) Application Layer Multicast tree topology



(b) Continuous test

Figure 5.7: The Basic Layout

Smooth_Join

In this test, we measured the impact that a set of joiners might have on the ongoing message flow. The sender was at HUJI, the receiver at MIT, and during the test, 4 processes joined at HUJI and one at NTU. Later on, all the joiners left . The tests show that the impact on the message flow was negligible 5.8(b). Messages were still received every 100ms.

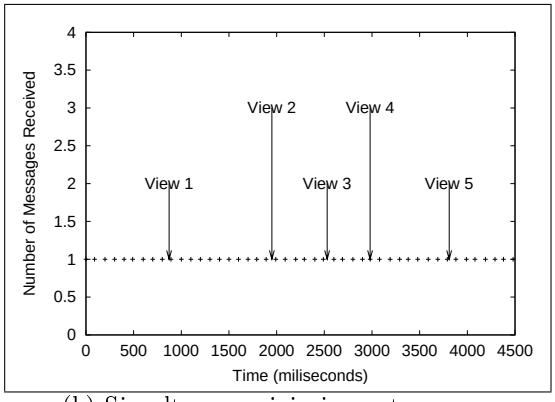
In the virtual synchrony implemented by Keidar et. al [63], the message flow needs to be stopped for at least one round-trip time (in the best possible scenario). In our specific setting, it should have taken at least 390ms.

Continuous behavior

The third test measured the behavior of the ad hoc membership implementation over a long period of time, during which the loss rate was rather high. In this test, the sender was located at HUJI, the receiver at MIT, while the process at NTU and the processes at HUJI joined and left repeatedly. The

Event	Time (milliseconds)
NS Resolve	T=0
Sync Reply	T+50
Msg from S1 (HUJI)	T+60
Msg from S2 (HUJI)	T+60
Msg from S3 (HUJI)	T+68
Msg from S4 (HUJI)	T+68
Msg from S5 (NTU)	T+73
Msg from S6 (HUJI)	T+253

(a) First message receiving time



(b) Simultaneous join impact

Figure 5.8: Impact of membership changes.

tests show that the behavior of the system, both during the periods with no membership changes and the periods with such changes, is almost identical 5.7(b), which demonstrates the efficiency of the algorithm.

Discussion

All the tests described above proved that the ad hoc approach enables processes to join the group promptly, with minimal impact on the ongoing message flow. We observed that applications that do not need strong characteristics will face the minimal impact. Applications that require stronger semantics will still need to wait for full synchronization, as found by Keidar et. al [63].

5.5 Conclusions

The focus of this chapter is to address the needs of a class of distributed applications that require high bandwidth, reliability and scalability, while not requiring the strong semantics of current distributed middleware solutions. Since current middleware cannot scale well when it is required to guarantee the

strong semantics, there is a need to identify a better tradeoff between the semantics and the efficiency.

The ad hoc membership algorithm that we have developed and implemented presents such a tradeoff. The performance results prove that our approach is feasible and can scale well.

The implementation shows that it is possible to integrate an external membership service with a hierarchical system for message distribution. We believe that other systems with hierarchical architecture and/or external membership service can may apply similar techniques to their algorithms. A reliable multicast based on forward error correction is a simple one-to-many application. However, the rest of the applications listed in Section 5.1.1 need better reliability and coordination than our approach offers. Future research is intended to provide better characterization of these classes.

Chapter 6

Evaluating Total Order Algorithms in WAN*

Agreed message delivery is an important service in distributed systems, especially when dealing with fault-tolerant applications. This chapter focuses on factors influencing protocols that provide global order in Wide Area Networks. Performance evaluation in real network conditions was conducted in order to compare two recently published algorithms. We studied the algorithms' latency in a steady state case in a real network and discovered that two factors, namely, a loss rate and variations in message propagation time, have a significant impact on the algorithms' performance.

6.1 Algorithms under Comparison

When comparing ordering algorithms, it is important to consider the guarantees that each algorithm provides. We start by presenting the order required by a replicated database application, namely, *Uniform Total Order (UTO)*. A formal definition of a UTO broadcast guarantee can be found in 1.1.2. In essence, it provides the same message delivery order for all the processes, including the faulty ones.

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

*This chapter is based on a paper by T. Anker, D.Dolev, G. Greenman and I.Shnayderman [10]. We do not consider Byzantine Failures in this work.

be either confirmed or changed by the UTO at a later stage. The algorithm presented in [102] (*Hybrid Algorithm*) provides both TO and UTO, while the algorithm presented in [92] (*Optimistic Algorithm*) provides PO (which the authors call Optimistic Order) and TO. Although no protocol is proposed for UTO, it could easily be built atop TO by collecting acknowledgments from the majority of the processes that have delivered messages in TO. The most important difference between Optimistic Order and TO is that in the latter the order may be changed *iff* a process is suspected to have failed, while Optimistic Order may be changed even if no process has been suspected. Below we discuss the protocols used by these algorithms in order to achieve the above ordering services.

In Optimistic Algorithm, a process is selected to serve as the Sequencer. For every message it receives, the Sequencer gives a TO number and notifies all the processes about the message order. The Optimistic Order is achieved by predicting (with a high probability) the time when the message is received by the Sequencer.

In Hybrid Algorithm, each process is marked as either passive or active, according to the location of the process in the network and its message sending rate. Active processes act as sequencers for passive ones. Lamport symmetric protocol [65] is used to achieve TO in the set of active processes. In order to compare a sequencer-based approach with a symmetric approach, we chose the topology and process sending rates so that all the participating processes were marked as active ones. The latency of Lamport symmetric protocol [65] can be improved by using local clocks, as was shown in [102].

6.2 Methodology

This section describes the methodology used for the evaluation of the ordering algorithms.

It is important to note that throughout the experiments we did not consider processes suspicions/failures. Although these are important factors, they should be studied in a separate work, since the current study focuses on the steady state case. However, message losses do occur in the steady state and may have significant impact on the algorithms' performance [76]. In order to achieve TO/UTO guarantee, lost messages must be re-acquired. Xpand [8] guarantees reliability by retransmitting lost messages, thus providing an appropriate framework for this study.

In the experiments, we used nodes from the RON project ([105]). The tests involved 6 sites, 5 residing in North America and one in Jerusalem, Israel. While the number of sites is relatively small, Xpand's

hierarchical structure and its support for multiple groups allow to extend the algorithms to large-scale systems. The nodes themselves were Celeron/733 or PentiumIII/1.1G machines running FreeBSD/Linux operating system, connected via commodity Internet and Internet2. The links had diverse delays and various packet loss rates.

We used Xpand to collect information about the network characteristics. Each host generated a new data packet every 10ms. Although data packets were sent by UDP protocol, the rate was limited by Xpand's TCP-friendly congestion control mechanism (see Chapter 4). Each node listed the time when a packet was received from the network. The ordering algorithms were evaluated by emulating their protocols behavior on the message logs.

6.2.1 Clock Skews

In order to estimate the ordering algorithm latency, we needed a good evaluation of the clock differences among all the participating processes. This was achieved using the mechanism described in [76]. The assumption made in calculating the clock differences was that message Round Trip Time (RTT) in a WAN was stable and rarely changed. The RTT was also assumed to be symmetric. Although these assumptions may not hold for a general case, they were applicable in our experiments. Time drift may be another concern when evaluating time differences. In our experiments, the logs were collected over a relatively short interval (several minutes' span), thus making time drift between the clocks negligible.

6.2.2 Implementation and Optimization of the Algorithms

This section describes essential implementation and optimization details. In order to emulate the ordering protocols, we assumed that control information can be piggybacked on data messages. This technique did not introduce any additional delay into the ordering protocols latency, as data messages were generated at a high rate.

Optimistic Algorithm relies on delay calculations. In order to emulate this algorithm, we used the logs to estimate the delays, and then emulated the protocol running on the logs using the estimated delay. High message sending rate and small variation in message propagation time may cause Optimistic Order to wrongly predict TO (on messages received in the same time window). To improve the TO prediction, Optimistic Algorithm batches messages [92]. For the same purpose, we used message-sending

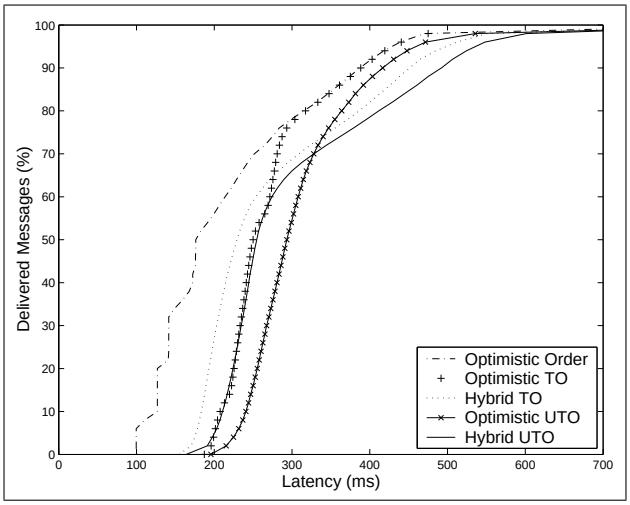


Figure 6.1: Comparing the Algorithms

time.

Some applications may require messages to be delivered in the same order they have been sent (*FIFO*). Since this is not part of the UTO definition, our implementation of Optimistic Algorithm does not provide FIFO, in order to achieve improved latency.

Since we achieved clock synchronization, in Hybrid Algorithm (See 6.2.1) we could put a timestamp on messages and use it to achieve TO.

In order to reduce the impact of message losses, we duplicated previous acknowledgments on each data message. This is feasible since (1) the size of an acknowledgment is very small and (2) an acknowledgment on message m also acknowledges messages preceding m.

6.3 Performance Results

The logs were collected in multi-continent (WAN) setting of RON Testbed Network. The experiment included six sites: Zikit1 (Jerusalem), RON0 (Boston), RON2 (Salt Lake City), RON33 (Ottawa), RON35 (Berkeley), RON45 (Chicago).

As there is no IP multicast connectivity among these sites, a proprietary application layer multicast (ALM) mechanism was used (see Section 4.1.1.1 and [8]). The obtained message distribution tree is the same as in Chapter 4.

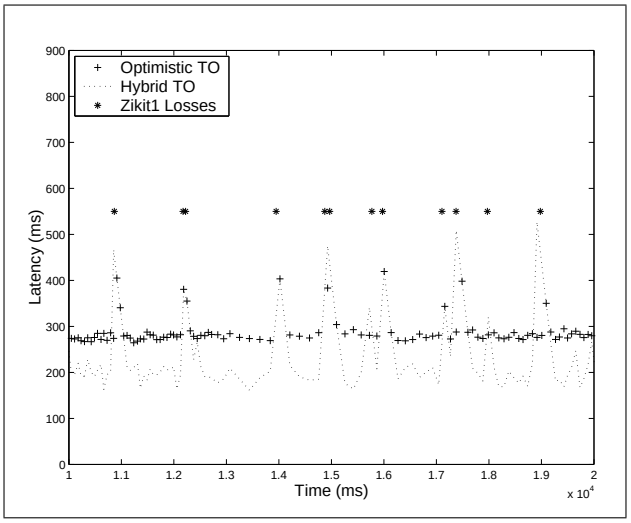


Figure 6.2: Impact of Losses

The maximum One-way Trip Time (OTT) registered between RON2 and Zikit1 was 330ms. It is worth noting that in some links, a significant percentage of messages experienced substantial delays. Also, we noticed that loss rate varied significantly over different links (the worst link experienced the average of 3.6% loss rate).

In order to compare the algorithms, we studied the latencies. For Optimistic Algorithm, we measured the time between message sending and its delivery by all the processes (*message latency*) in Optimistic Order, TO and UTO. For Hybrid Algorithm, we measured the latencies of TO and UTO. For some messages, very high latencies were observed due to loss bursts. Therefore, the average values are not representative, and we chose to use Cumulative Distribution Function (CDF) (Figure 6.1). Here, the X axis presents latencies, while the Y axis shows the percentage of messages delivered by specific time.

As can be seen from the data represented in the graph, the behavior of the curves changes approximately at the 60% level. Below this point, the latencies are similar to those expected. A message m is deliverable in Optimistic TO only after m's TO number has been received from the Sequencer. Hybrid TO delivers m when a subsequent message from each process is received. Therefore, Hybrid TO imposed a shorter delay on m, as the sending rate was high during the experiment. Optimistic Order latency is better than that of Hybrid TO, since it waits only for a precomputed delay to deliver m and does not require reception of any additional message.

In the upper part of the graph, the pattern of the curves changes drastically. Optimistic Order curve converges with that of Optimistic TO. It means that Optimistic Order was predicted wrongly, hence, m's Optimistic Order delivery time was substituted by Optimistic TO delivery time. The decrease in Optimistic TO slope indicates an increase in the number of messages delivered with a high latency. This increase is even more significant in Hybrid TO.

As we found out, this slowdown of TO algorithms was caused by message losses. The graph in Figure 6.2 shows the latencies measured by Zikit1 for its own messages. Only a short time interval is presented (time is measured from the start of the experiment) in order to make the graph more comprehensible. It is evident that message losses have significant influence on the latency. It is worth noting that the impact on Hybrid TO is higher than on Optimistic TO. In order to deliver message m, the Lamport Algorithm (used in Hybrid TO) requires that a message with a timestamp higher than m be received from every process, while keeping the FIFO order. To understand why losses increase the Optimistic TO latency, we need to consider message completion mechanism in Xpand [8]. This mechanism causes a lost message m to be retransmitted point-to-point, and consequently, the Sequencer receives m much quicker than Zikit1 and then sends its TO number to all the nodes. This causes Zikit1 to postpone the delivery of the messages following m, until m is recovered.

In Figure 6.3 the reasons for Optimistic Order delivery slowdown are shown. A replicated database may only benefit from correctly "guessed" Optimistic Order. In case of a wrong guess, hereafter referred to as *miss*, Optimistic TO delivery time is to be used. If TO is not equal to Optimistic Order, the replicated database has to perform a rollback. We assumed that this rollback may be performed immediately after TO is received. Optimistic Order latency measured by Zikit1 for its own messages is shown in Figure 6.3.

The percentage of misses varied from 11% measured by RON0 to 43% measured by RON35 for Optimistic Order. It is noteworthy that once an application learns that Optimistic Order is incorrect, it has to perform the rollback for all the messages received starting from the missing message. The peaks in the graph correspond to the first miss. Afterwards, Optimistic Order latency decreases for each following message, due to the reduction of the time interval from the moment when a message was sent till it is TO-delivered.

Although the numbers of misses might seem very high, it is not of great significant, as they largely

The misses were not registered by Hybrid Algorithm as no failures happened.
Our model considers all messages sent to a single group as potentially conflicting transactions [64].

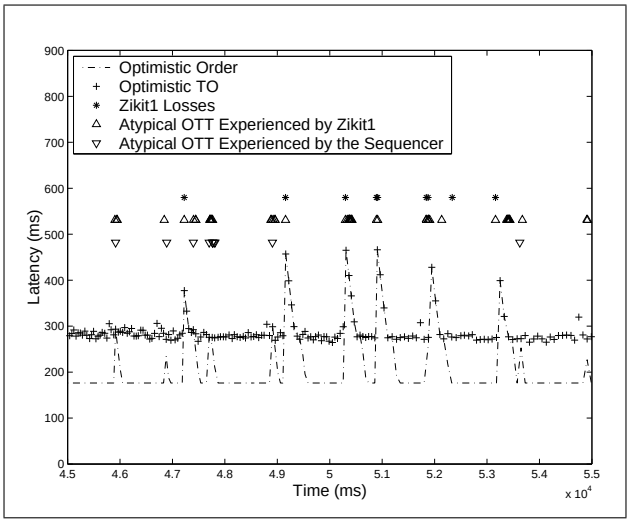


Figure 6.3: Impact of OTT Variations

depend on the rollback model and message sending rate. However, it is important to understand the reasons that caused those misses. As can be seen in Figure 6.3, those misses are not evenly distributed over time, but occur in bursts. Those bursts are instigated by either message losses or high (*atypical*) OTT's experienced by Zikit1 as well as the Sequencer. Such OTT's, in turn, may be caused both by the message completion protocol and by the network. To study the network links' characteristics, we used *ping* utility and found out that RTTs of some messages were significantly higher than the average RTT on the same link. On some links, the number of atypical RTT's was as high as 2%.

6.4 Conclusions and Future Work

The study focused on factors that have considerable impact on the Total Order algorithms' latency in a WAN. The algorithms under study were found to be vulnerable to message losses and to variations in message propagation time which are inherent in a WAN environment. Hence, while developing a distributed algorithm for a WAN, it is essential to consider the impact of those factors on the performance. We also found that the message completion mechanism in Xpand instigates misses in Optimistic Order and should be modified.

Another important observation is that the loss rate of a link does not vary considerably. This fact was also mentioned in [80]. We believe that an efficient protocol in WAN is to combine the usage of

unreliable channels built over lossy links with reliable channels over lossless links.

In order to further improve the performance of the studied algorithms, some optimizations are to be carried out, that require from a process to measure loss rate on its links. In case when a receiver finds out that the loss rate of incoming messages is too high, it is to stop delivering messages in Optimistic Order. In the other case, when a sender finds out that its messages are frequently lost by other processes Hybrid Algorithm is to mark the sender as passive. (See section 6.1.) Optimistic Algorithm is to use a similar approach. As an alternative Forward Error Correction code can be used to minimize the impact of message losses.

The study strategy is comprehensive enough to be extended to cover various network topologies, sending rates and other total order algorithms.

The set of collected logs reflecting the real network conditions, as well as the simulator designed for this study, allow an algorithm developer to evaluate the performance. The logs and the simulation code are located at www.cs.huji.ac.il/labs/danss/TO_logs.html.

Bibliography

[1] ACM. *Communications of the ACM 39(4), special issue on Group Communications Systems*, April 1996.

[2] M. Allman, V. Paxson, W. Stevens. TCP Congestion Control. RFC 2581, April 1999. Internet Engineering Task Force, Network Working Group.

[3] Y. Amir, B. Awerbuch, C. Danilov, J. Stanton. Global Flow Control for Wide Area Overlay Networks: A Cost-Benefit Approach. *OPENARCH-2002*, strony 155–166, Czerw. 2002.

[4] Y. Amir, C. Danilov, J. Stanton. A Low Latency, Loss Tolerant Architecture and Protocol for Wide Area Group Communication. *Proceedings of ICDSN'2000*, 2000.

[5] Y. Amir, D. Dolev, S. Kramer, D. Malki. Transis: A Communication Sub-System for High Availability. *22nd Annual International Symposium on Fault-Tolerant Computing*, strony 76–84, July 1992.

[6] T. Anker, D. Breitgand, D. Dolev, Z. Levy. Congress: Connection-oriented group-address resolution service. *SPIE*, 1997.

[7] T. Anker, G. Chockler, D. Dolev, I. Keidar. Scalable group membership services for novel applications. M. Mavronicolas, M. Merritt, N. Shavit, redaktorzy, *Networks in Distributed Computing (DIMACS workshop)*, wolumen 45 serii *DIMACS*, strony 23–42. American Mathematical Society, 1998.

[8] T. Anker, G. Chockler, I. Shnayderman, D. Dolev. The Design and Performance of Xpand: A Group Communication System for Wide Area Networks. Raport instytutowy 2001-56, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, August 2001. URL: http://leibniz.cs.huji.ac.il/research/, See also the previous version TR2000-31.

[9] T. Anker, D. Dolev, G. Greenman, I. Shnayderman. Wire-speed total order. *IPDPS'06.*

[10] T. Anker, D. Dolev, G. Greenman, I. Shnayderman. Evaluating total order algorithms in wan. 2003.

[11] T. Anker, D. Dolev, G. Greenman, I. Shnayderman. Wire-speed total order. Technical report, January 2006.

[12] T. Anker, D. Dolev, I. Keidar. Fault Tolerant Video-On-Demand Services. *Proceedings of the 19th International Conference on Distributed Computing Systems, (ICDCS'99)*, June 1999.

[13] T. Anker, D. Dolev, I. Shnayderman. Ad Hoc Membership for Scalable Applications. *16th Intl. Conference on Distributed Computing Systems*, Oct. 2002.

[14] T. Anker, D. Dolev, I. Shnayderman. Ad Hoc Membership for Scalable Applications. Raport instytutowy 2002-21, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, April 2002.

[15] T. Anker, D. Dolev, I. Shnayderman, I. Sukhov. TCP-Friendly Many-to-Many End-to-End Congestion Control. *Proc. 22st IEEE Symposium on Reliable Distributed Systems*. IEEE CS, October 2003.

[16] ANSI. Fibre Channel Physical and Signaling Interface (FC-PH). X3.230-1994.

[17] I. T. Association. InfiniBand Architecture Specification. Release 1.2.

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

[19] M. Barborak, M. Malek, A. Dahbura. The consensus problem in distributed computing. *ACM Comput. Surv.*, 25(2):171–220, Czerw. 1993.

[20] M. Ben-Or. Another advantage of free choice:completely asynchronous agreement protocols. *Annual ACM Symposium on Principles of Distributed Computing*, strony 27–30, 1983.

[21] K. Berket, D. A. Agarwal, P. M. Melliar-Smith, L. E. Moser. Overview of the intergroup protocols. *International Conference on Computational Science (1)*, strony 316–325, 2001.

[22] D. P. Bertsekas, R. G. Gallager. *Data Networks*. Prentice-Hall, 1992.

[23] K. Birman, A. Schiper, P. Stephenson. Lightweight causal and atomic group multicast. *ACM Trans. Comput. Syst.*, 9(3):272–314, 1991.

[24] K. P. Birman, R. Friedman, M. Hayden, I. Rhee. Middleware support for distributed multimedia and collaborative computing. *Proceedings of the Multimedia Computing and Networking (MMCN'98)*, 1998.

[25] R. Burns. *Data Management in a Distributed File System for Storage Area Networks*. Praca doktorska, March 2000. University of California, Santa Cruz.

[26] J. Byers, M. Frumin, G. Horn, M. Luby, M. Mitzenmacher, A. Roetter, W. Shaver. FLID-DL: Congestion Control for Layered Multicast. *Second Int'l Workshop on Networked Group Communication (NGC 2000)*, November 2000.

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

[28] T. D. Chandra, S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[29] B. Charron-Bost, X. Defago, A. Schiper. Broadcasting messages in fault-tolerant distributed systems: The benefit of handling input-triggered and output-triggered suspicions differently. *SRDS '02: Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, strona 244, Washington, DC, USA, 2002. IEEE Computer Society.

[30] J. Chase, A. Gallatin, K. Yocum. End system optimizations for high-speed TCP. *IEEE Communications Magazine*, 39(4):68–74, 2001.

[31] W. Chen, S. Toueg, M. K. Aguilera. On the quality of service of failure detectors. *Proceedings of the International Conference on Dependable Systems and Networks (DSN 2000)*, New York, 2000. IEEE Computer Society Press.

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

[33] G. Chockler, N. Huleihel, I. Keidar, D. Dolev. Multimedia Multicast Transport Service for Groupware. *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, September 1996. Full version available as Technical Report CS96-3, The Hebrew University, Jerusalem, Israel.

[34] G. V. Chockler, I. Keidar, R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Comput. Surv.*, 33(4):427–469, 2001.

[35] A. Coccoli, A. Bondavalli, F. Giandomenico. Analysis and estimation of the quality of service of group communication protocols, 2001.

[36] A. Coccoli, S. Schemmer, F. Di Giandomenico, M. Mock, A. Bondavalli. Analysis of group communication protocols to assess quality of service properties. *HASE00 - 5th IEEE High Assurance System Engineering Symposium*, strony 247–256, Albuquerque, NM, USA, 2000.

[37] A. Coccoli, P. Urban, A. Bondavalli, A. Schiper. Performance analysis of a consensus algorithm combining stochastic activity networks and measurements, 2002.

[38] F. Cristian, R. de Beijer, S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1(4):177–201, Czerw. 1994.

[39] F. Cristian, S. Mishra, G. Alvarez. High-performance asynchronous atomic broadcast, 1997.

[40] X. Defago, A. Schiper, P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, December 2004.

[41] D. DeLucia, K. Obraczka. Multicast feedback suppression using representatives. *INFOCOM (2)*, strony 463–470, 1997.

[42] D. Dolev, D. Malki. The Transis Approach to High Availability Cluster Communication. *Communications of the ACM*, 39(4), April 1996.

[43] S. Donnelly. *High Precision Timing in Passive Measurements of Data Networks.* Praca doktorska, June 2002. University of Waikato, New Zeland.

[44] H. S. Duggal, M. Cukier, W. H. Sanders. Probabilistic verification of a synchronous round-based consensus protocol. *Symposium on Reliable Distributed Systems*, strony 165–174, 1997.

[45] S. Floyd. Congestion Control Principles. RFC 2914, September 2000. Internet Engineering Task Force, Network Working Group.

[46] S. Floyd, K. Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking*, 7(4):458–472, 1999.

[47] S. Floyd, M. Handley, J. Padhye, J. Widmer. Equation-based congestion control for unicast applications. *SIGCOMM 2000*, strony 43–56, Stockholm, Sweden, August 2000.

[48] R. Friedman, R. van Renesse. Strong and weak virtual synchrony in Horus. *Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, (SRDS'96)*, October 1996.

[49] R. Friedman, R. van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. *HPDC*, strony 233–242, 1997.

[50] S. Frolund, F. Pedone. Revisiting reliable broadcast. Raport instytutowy HPL-2001-192, 2001.

[51] J. Gray. Why do computers stop and what can be done about it. *Proceedings of the 5th Symposium on Reliablity in Distributed Software and Database systems*, Sty. 1986.

[52] G. Greenman. Msc. thesis: High speed total order for san infrastructure, June 2005. The Hebrew University of Jerusalem, http://www.cs.huji.ac.il/labs/danss/.

[53] K. Guo, L. Rodrigues. Dynamic Light-Weight Groups. *Proceedings of the 17th International Conference on Distributed Computing Systems, (ICDCS'97)*, May 1997.

[54] M. Handley. TCP Friendly Rate Control (TFRC): Protocol Specification. RFC 3448, January 2003. Internet Engineering Task Force, Network Working Group.

[55] M. Hayden. *The Ensemble System.* Phd thesis, Cornell University, Computer Science, 1998.

[56] M. Hayden, K. Birman. Probabilistic Broadcast. TR 96-1606, dept. of Computer Science, Cornell University, Jan 1996.

[57] IBM. *RS/6000 SP High Availability Infrastructure.* SG24-4838, available online at: http://www.redbooks.ibm.com/abstracts/sg244838.html.

[58] IEEE. 802.1s Multiple Spanning Tree Standard. IEEE standard.

[59] IEEE. 802.3ad Link Aggregation Standard. IEEE standard.

[60] IEEE. 802.3x Flow Control Standard. IEEE standard.

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

[62] S. K. Kasera, G. Hjálmtýsson, D. F. Towsley, J. F. Kurose. Scalable reliable multicast using multiple multicast channels. *IEEE/ACM Transactions on Networking*, 8(3):294–310, 2000.

[63] I. Keidar, J. Sussman, K. Marzullo, D. Dolev. Moshe: A group membership service for wans. *ACM Trans. Comput. Syst.*, 20(3):191–238, 2002.

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

[65] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[66] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[67] M. Christensen, K. Kimball, F. Solensky. Considerations for IGMP and MLD Snooping Switches. IETF draft, February 2005.

[68] L. M. Malhis, W. H. Sanders, R. D. Schlichting. Numerical evaluation of a group-oriented multicast protocol using stochastic activity networks. *PNPM '95: Proceedings of the Sixth International Workshop on Petri Nets and Performance Models*, strona 63, Washington, DC, USA, 1995. IEEE Computer Society.

[69] C. P. Malloth, A. Schiper. View synchronous communication in large scale distributed systems. *Proceedings of the 2nd Open Workshop of the ESPRIT project BROADCAST*, Grenoble, France, Lip. 1995.

[70] S. McCanne, V. Jacobson, M. Vetterli. Receiver-driven layered multicast. *ACM SIGCOMM*, wolumen 26,4, strony 117–130, New York, Sier. 1996. ACM Press.

[71] Mircosof. *Microsoft Wolfpack*.

[72] S. Mishra, L. Wu. An evaluation of flow control in group communication. *IEEE/ACM Transactions on Networking (TON)*, 6(5):571–587, 1998.

[73] L. Moll, M. Shand. Systems performance measurement on PCI pamette. K. L. Pocek, J. Arnold, redaktorzy, *IEEE Symposium on FPGAs for Custom Computing Machines*, strony 125–133, Los Alamitos, CA, 1997. IEEE Computer Society Press.

[74] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, C. A. Lingley-Papadopoulos. Totem: A Fault-Tolerant Multicast Group Communication System. *Communications of the ACM*, 39(4), April 1996.

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

[76] O.Bakr, I. Keidar. Evaluating the running time of a communication round over the internet. *21th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, strony 243–252, 2002.

[77] J. Padhye, V. Firoiu, D. Towsley, J. Krusoe. Modeling TCP throughput: A simple model and its empirical validation. *ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, strony 303–314, Vancouver, CA, 1998.

[78] J. Padhye, V. Firoiu, D. F. Towsley, J. F. Kurose. Modeling TCP Reno performance: a simple model and its empirical validation. *IEEE/ACM Transactions on Networking*, 8(2):133–145, April 2000.

[79] S. Paul, K. Sabnani, J. Lin, S. Bhattacharyya. Reliable multicast transport protocol (RMTP). *IEEE Journal on Selected Areas in Communications*, April 1997.

[80] V. E. Paxson. *Measurements and Analysis of End-to-End Internet Dynamics*. Praca doktorska, University of California, Lawrence Berkeley National Laboratory, April 1997.

[81] F. Pedone, A. Schiper. Optimistic atomic broadcast. *Proceedings of the 12th International Symposium on Distributed Computing*, strony 318–332. Springer-Verlag, 1998.

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

[83] D. Powell. *Delta-4: A Generic Architecture for Dependable Distributed Computing*. Springer-Verlag, 1991.

[84] M. Rabin. Randomized byzantine generals. *24th Annual ACM Symposium on Foundations of Computer Science*, strony 403–409, 1983.

[85] H. V. Ramasamy, P. Pandey, J. Lyons, M. Cukier, W. H. Sanders. Quantifying the cost of providing intrusion tolerance in group communication systems. *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, strony 229–238, Washington, DC, USA, 2002. IEEE Computer Society.

[86] I. Rhee, N. Ballaguru, G. N. Rouskas. MTCP: Scalable TCP-like congestion control for reliable multicast. *INFOCOM*. IEEE, Mar. 1999.

[87] I. Rhee, S. Cheung, P. Hutto, V. Sunderam. Group Communication Support for Distributed Multimedia and CSCW Systems. *17th Intl. Conference on Distributed Computing Systems*, May 1997. Also available as technical report of Dept. of Mathematics Computer Science, Emory University, Atlanta, GA 30322.

[88] I. Rhee, V. Ozdemir, Y. Yi. Tear: Tcp emulation at receivers – flow control for multimedia streaming, 2000.

[89] A. I. T. Rowstron, A.-M. Kermarrec, M. Castro, P. Druschel. SCRIBE: The design of a large-scale event notification infrastructure. *Networked Group Communication*, strony 30–43, 2001.

[90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, Gru. 1990.

[91] N. Sergent, X. Défago, A. Schiper. Impact of a failure detection mechanism on the performance of consensus. *PRDC '01: Proceedings of the 2001 Pacific Rim International Symposium on Dependable Computing*, strona 137, Washington, DC, USA, 2001. IEEE Computer Society.

[92] A. Sousa, J. Pereira, F. Moura, R. Oliveira. Optimistic total order in wide area networks. *Proc. 21st IEEE Symposium on Reliable Distributed Systems*, strony 190–199. IEEE CS, October 2002.

[93] J. B. Sussman, I. Keidar, K. Marzullo. Optimistic virtual synchrony. *Symposium on Reliability in Distributed Software*, strony 42–51, 2000.

[94] K. Tindell, A. Burns, A. J. Wellings. Analysis of hard real-time communications. 9(2):147–171, Wrze. 1995.

[95] P. Urbán, X. Défago, A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. *Proc. 9th IEEE Int'l Conf. on Computer Communications and Networks (IC3N 2000)*, Paz. 2000.

[96] P. Urbán, X. Défago, A. Schiper. Chasing the FLP impossibility result in a LAN or how robust can a fault tolerant server be? *Proc. 20th IEEE Symp. on Reliable Distributed Systems (SRDS)*, strony 190–193, New Orleans, LA, USA, October 2001.

[97] P. Urbán, X. Défago, A. Schiper. Neko: A single environment to simulate and prototype distributed algorithms. *Journal of Information Science and Engineering*, 18(6):981–997, November 2002.

[98] P. Urban, I. Shnayderman, A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms. 2003.

[99] P. Urbán, I. Shnayderman, A. Schiper. Comparison of failure detectors and group membership: Performance study of two atomic broadcast algorithms (extended version). Raport instytutowy IC/2003/15, École Polytechnique Fédérale de Lausanne, Switzerland, April 2003.

[100] R. van Renesse, K. P. Birman, S. Maffeis. Horus: A Flexible Group Communication System. *Communications of the ACM*, 39(4), April 1996.

[101] R. van Renesse, T. M. Hickey, K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. TR 94-1442, dept. of Computer Science, Cornell University, August 1994.

[102] P. Vicente, L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. *Proc. 21st IEEE Symposium on Reliable Distributed Systems*. IEEE CS, October 2002.

[103] L. Vicisano, L. Rizzo, J. Crowcroft. TCP-like congestion control for layered multicast data transfer. *INFOCOM (3)*, strony 996–1003, 1998.

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

[105] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, A. Joglekar. An integrated experimental environment for distributed systems and networks. *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, strony 255–270, Boston, MA, Gru. 2002. USENIX Association. url: www.emulab.net.

[106] J. Widmer, R. Denda, M. Mauve. A survey on TCP-friendly congestion control. *IEEE Network*, 15(3):28–37, 2001.

[107] J. Widmer, M. Handley. Extending Equation-Based congestion control to multicast applications. strony 275–286.

[108] R. Yavatkar, J. Griffoen, M. Sudan. A reliable dissemination protocol for interactive collaborative applications. *ACM Multimedia*, strony 333–344, 1995.

[109] S. Zhuang, B. Zhao, A. Joseph, R. Katz, J. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant widearea data dissemination, 2001.