University of Florida
Computer and Information Sciences

Two Approaches for High Concurrency in Multicast-Based Object Replication

by
Theodore Johnson and Lionnel Maugis
Dept. of CIS, University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu

Department of Computer and Information Sciences
Computer Science Engineering Building
University of Florida
Gainesville, Florida 32611

Two Approaches for High Concurrency in Multicast-Based Object Replication

Theodore Johnson and Lionnel Maugis
Dept. of CIS, University of Florida
Gainesville, Fl 32611-2024
ted@cis.ufl.edu

November 28, 1994

Abstract

This report presents a replica control protocol for atomic objects. The protocol is derived from an atomic broadcast primitive, and places constraints on the delivery of messages to provide a consistent message order among sites. Several heuristic techniques are proposed to reduce the latency of message delivery, for two types of orders. Messages are delivered either in the same order for all sites, or in an order semantically equivalent to this unique ordering. The equivalence relation is based on the commutativity property of operations on objects, i.e: two deposit operations commute.

The protocol uses a reliable causal multicast primitive, and is fully distributed. The first set of heuristics is based on a voting scheme, and delivers messages in a unique order. Totally ordered atomic multicast can be built on top of a reliable causal multicast by waiting until each processor in the group has multicast a message, inserting them in a causal graph, and then delivering the roots of this graph. This latency can be reduced with a voting scheme that allows messages to be inserted in a total order without waiting for a message from all group members. Dolev, Kramer and Malki proposed such a protocol, but it is more restrictive than it needs to be, and can be extended for a wider range of operating conditions. In particular, we show how to deliver a message even if it is acknowledged by less than half of the processors.

If there remains some undelivered messages (not yet placed in the total order), a second set of heuristics are available to a processor. If messages correspond to operations on objects, it can use local and type specific information to minimize delivery constraints. More specifically, if operations commute, it is possible to initiate some of them early, provided that the execution is equivalent to a linear sequence of operation invocation, the same for all sites. The commutativity of operations depends on the state of the data, thus allowing more concurrency than otherwise possible. We present a general algorithm for ordering operations provided that messages are causally ordered, and give two heuristics to make it practical.

Finally, we present a performance evaluation of the protocols based on discrete-event simulation. Our multicast primitives perform well, and adapt better than previous work to various network configurations. It is also more scalable.

Keywords: atomic broadcast, delivery constraints, commutativity based concurrency control, abstract data type

Contents

1 Introduction **5**

1.1 High Available Replicated Atomic Data 5

1.2 Contributions 7

1.3 Organization 8

2 Related Work **8**

2.1 Replica Control Protocols 8

2.2 Process Group Communication 9

2.3 Replicated Databases And Atomic Data Types 11

3 Group Communication **12**

3.1 System Model 13

3.2 Reliable Communication 14

3.3 Causality Graph 15

3.4 Group Membership 16

3.5 Agreed Communication 18

4 Atomic Group Multicast **18**

4.1 Sketch Of The Total Ordering Protocols 18

4.2 Early Delivery 19

4.3 A Refinement Of The ToTo Delivery Rule 21

4.4 Generalized Total Ordering Protocol 23

4.5 Hierarchical Delivery Rule 24

4.6 Early Delivery In A Wave 26

4.7 Example 27

5 Semantic Based Ordering Multicast **29**

5.1 Commutativity Based Consistency 29

5.2 Semantic Ordering Protocol 31

5.3 Practical Heuristics 33

5.4 Revised Semantic Ordering Protocol 34

5.5 Example 36

6 Performance **36**

6.1 Performance Goals . . . 37

6.2 System Model . . . 37

6.3 Network topology . . . 37

6.4 Workload . . . 38

6.5 Communication delays . . . 38

6.6 Service time . . . 39

6.7 Metrics . . . 39

6.8 Simulation . . . 39

6.9 Results . . . 40

7 Approval Voting **46**

7.1 Topics In Voting Theory . . . 47

7.2 Plurality voting and its critiques . . . 47

7.3 Normative properties of voting systems . . . 49

7.4 Properties Of Approval Voting . . . 51

7.5 Discussion . . . 51

8 Conclusion **52**

8.1 Future Areas Of Research . . . 53

Bibliography **54**

1 Introduction

This work investigates high performance replication techniques for atomic objects in asynchronous distributed systems.

The algorithms provide a consistent order for operation processing on replicas and are geared towards meeting both availability and response time requirements. The system model takes advantage of the multicast capability of modern networks, and of the commutativity property of operations applicable to objects. The protocol is derived from an efficient family of atomic multicast primitive that supports open-group communication, fail-stop failures, and membership changes.

This service can be used effectively to implement atomicity in a fault-tolerant distributed system. It propagates updates to remote copies, schedule operations and guarantees the serializability of transactions. We demonstrate the adequacy of such order in the design of a replicated main-memory database server.

1.1 High Available Replicated Atomic Data

When a typed object is stored redundantly at multiple locations, it becomes more available. In the face of failures or communication delays, copies can still be accessible. Furthermore, if a local copy exists, it can be accessed more quickly. Unfortunately, this increase in availability and fault-tolerance has a cost. Due to the complexity and expense of maintaining replicated objects, most distributed database systems limit the amount of replication to a few copies. More than the necessary additional storage space required, maintaining mutually consistent copies is a complex task. When an object changes, updates[1] have to be propagated in a consistent way to all copies.

This cost manifests itself in the time taken to perform some action. Sites have to cooperate and some amount of synchronization is required to maintain correctness. For example, one has either to make sure that at least one copy is up-to-date or that all replica follow the same sequence of updates. The cost also includes increase in message traffic both in the number of messages and the size of each message.

More formally, a system should be one-copy equivalent to be consistent; it should behave as if each object has only one copy that the user can tell. A replica control protocol maintains the consistency of distributed data by ensuring that the database is one-copy equivalent. If objects are accessed through groups of operations or transactions[2] the system should also behave as if these were executed in some serial order. This is the role of concurrency control protocols, such as two phase locking or timestamp ordering. An atomic commitment protocol is then usually invoked so that all sites agree on the outcome of the transaction (abort, commit).

The development of group communication protocols make available an alternative paradigm to building reliable distributed applications. Indeed, communication plays a central role in the management of replicated data and early replica control protocols underestimated total communication costs [4]. Process group primitives provide several notions of order for the occurrence of particular events: message delivery, failure notification, process joining, leaving, as well as the reliable sending and receiving of messages to a group. These orders are more and more being used to understand, model and design distributed software. Ordinary messages have no delivery constraints, causal messages are delivered when all their predecessors have been delivered, and totally ordered messages are delivered in the same order at all sites. At the application level, processes agree on the construction of a total order to master the uncertainties due to the unreliability and asynchronism of the underlying communication channels. They view the evolution of the system in a consistent way.

[1] An update message describes the changes that have to be applied to bring the database up-to-date. It may contain new values for particular data items, or just a description of some operations to be performed.

[2] A transaction is a set of operations representing one A.C.I.D unit of work (atomic, consistent, isolated and durable).

Virtual synchrony and distributed state machines are two well known approaches that use this total order to build fault-tolerant services.

Introduced by Birman [16], *virtual synchrony* is an execution model with a collection of properties that allow processes to cooperate and appear *logically* synchronized. Each process of a group has an associated *group view* of the working processors. Multicast communication to the group is reliable, atomic (either all or no non-failed members of a group deliver a message), and totally ordered[3] (processes deliver the same sequence of messages). Membership changes are coordinated with communication: group views changes are reported in the same order to all members, and failures are reported according to the fail-stop model. If a failure is reported to any processes, all processes see the same event. Between two configuration changes, all working processes receive the same set of messages. Also, if the network partitions, only one partition is allowed to make progress.

The *state machine* approach [83, 73] is a general method for implementing a fault-tolerant service by replicating servers and coordinating clients interactions with server replica. A *state machine* is a program that cyclically reads input from its sensors, performs a deterministic computation based on this data, and writes output to an external activator. The key for implementing fault-tolerant state machines is to ensure replica coordination. All replica receive and process the same sequence of requests. More specifically, every non faulty state machine receives every request (Agreement), and every non faulty state machine replica processes these requests in the same relative order (Order). Deterministic state machines can also implement replicated transaction processing systems by executing a transaction when it is totally ordered with respect to other transactions. The communication substrate ensures that all transactions will eventually reach all sites and that all sites will execute the transactions in the same order.

In this report, we explore the design and implementation of a family of atomic multicast primitives which minimize the constraints placed on message delivery. We are primarily concerned with two types of message order for operations on objects: a unique order, which is the same for all sites, and a semantic order, which uses type specific information to increase concurrency. Such semantic order can be used to build replicated transactional servers based on the state machine approach. Several protocols are used for this facility, and have the property of being *all* fully distributed.

1. Reliable causal multicast. Networks are quite reliable, but messages can get lost or discarded *(Omission failure)*. We use the reliable multicast engine of Trans [61]/Transis [6] to overcome this problem. It incorporates available hardware multicast primitive as part of the system model. In a failure-free environment, a broadcast requires only one single message. It uses unreliable datagrams and performs recovery by piggybacking positive and negative acknowledgments on messages. The acknowledgments also directly provide a causal order.

2. Virtually synchronous process groups *(Crash failure)*. The failure of a process, the joining of a node has to be carefully monitored to ensure that configuration changes are *logically* synchronized with all the regular messages. We use the group membership protocol of Amir et al. [5] for its adequacy to the Transis protocol. It operates on top of the causal order and does not block the regular flow of messages while membership changes are handled.

3. Atomic multicast and total ordering. The atomic multicast primitive delivers messages in a linear order. We designed a set of rules to deliver concurrent messages early as soon as their position in the total order is decided. Another solution would be to deliver message using logical clocks and timestamps [56], but this approach makes the system run at the speed of the slowest processor.

4. Semantic based ordering for replicated abstract data types. The total ordering can be relaxed by using the commutativity property of operations. We used Weihl's notion of forward commutativity [88] to better exploit concurrency. It is a local dynamic property that ensures the serializability of transactions.

[3] Some families of protocols supports weaker delivery semantics. FIFO order: messages are delivered in the order in which they were sent; causal order: related messages are delivered in the same order; semantic order where the order depends on application dependent information. We consider here a total order which extends the causal ordering of messages.

This particular notion of commutativity is interesting in the sense that conflicts depend on the state of objects and also has subtle interactions with recovery algorithms.

5. Main-memory databases recovery strategy. We designed a replicated server for which the database would reside in memory and also have intention lists for recovery (no undo/redo). To provide better performance, we preferred a deferred update strategy for concurrency control to the more traditional update-in-place and undo logging.

1.2 Contributions

This work improves upon previous work in several significant ways. First, we refined an atomic broadcast protocol (ToTo) to reduce its latency of message delivery. This algorithm orders linearly causal messages and gives a priority to the first messages acknowledged by a majority of processors. To construct a total order on messages, we designed a decision procedure that generalizes this priority scheme. With this extension, our algorithm provides better concurrency, and is able to cope with a wider range of communication patterns.

Second, we proposed a framework which allows operations to be ordered in such a way that the consistency of replicated objects is preserved. Separate sites may schedule operations in a different relative order as long as this ordering is equivalent to some linear order, the same for all sites. The notion of equivalence is based on the commutativity property of operations and depends on the state of the data. This ordering offers a level of concurrency unattainable with a total ordering protocol.

Finally, we evaluated the performance of our family of protocols by discrete event simulation. The algorithms are genuinely distributed and perform well under most operating conditions. The proposed improvements are also straightforward to implement and do not bear a significant processing cost.

- Total Ordering Protocol
  1. We use a variation of "approval" voting to give a priority to some messages. With this particular voting scheme, sites arrive quickly to a consensus for the ordering of messages.
  2. Our parameterized delivery predicate $R(\Phi)$ allows messages to be totally ordered even if they are acknowledged by a minority of sites. This improvement has a noticeable impact on performance.
  3. We designed a hierarchical delivery rule to relax the notion of majority in the voting procedure. If a consensus cannot be achieved with a majority of Φ, we try with another Φ'.
  4. The voting procedure elects a set of messages, and messages within this set are then delivered in a deterministic order. We show how to use this order to deliver messages even if the outcome of the election is not decided (yet).
- Semantic Ordering Protocol
  1. We formally defined the correctness of our protocol based Weihl's notion of commutativity.
  2. We designed a general algorithm to order operations on objects and presented two heuristics that makes it very practical.
- Performance evaluation. We compared the latency and synchronization constraints of our family of protocols for different types of networks and various numbers of processors.

From a more general perspective, this work applies advances of voting theory to distributed computing, to fasten the construction of a consensus for the ordering of messages. As agreement is a fundamental problem in distributed systems, the set of arcane tricks presented in the following chapters could prove to be useful for other problems.

1.3 Organization

Following this introduction, the report is organized as follows: chapter 2 discusses the related work, chapter 3 the system model and group communication services. Chapter 4 presents our atomic multicast protocols. The semantic ordering protocol is defined in chaper 5. Chapter 6 evaluates the performance of our family of multicast protocols. Chapter 7 presents the properties of our voting scheme and provide a summary of results in the theory of voting.

2 Related Work

An important body of research has been devoted to the management of replicated data. Some form of correctness needs to be maintained, and information about updates has to be propagated in the network. We review below some replica control protocols and discuss the benefits of using advances in communication protocols to provide high availability and low response time.

2.1 Replica Control Protocols

Numerous algorithms for the management of replicated data can be found in the research literature. Popular techniques for providing high availability are based on voting or primary copy [33]. Thomas proposed *majority voting* [87], extended by Gifford for *quorum consensus* protocols [46]. Each site is assigned a vote and, in order to perform a read (write) operation, a number of votes, or quorum, need to be acquired. Herlihy defined quorums for abstract data types [48], in which each operation of an object is assigned a set of quorums or set of sites whose cooperation is needed to execute that operation. As the votes assignment is fixed a priori, these schemes are classified as static voting.

Dynamic protocols adjust as sites fail and recover. Voting protocols adjust votes assignment and quorum sizes, while other algorithms use network information to reconfigure and do additional work. In the *missing writes* method of Eager and Sevcivk [41], a read-one, write-all (rowa) strategy is used when all sites are up; however if a sites goes down, the size of the read quorum is increased and that of the write quorum is decreased. In the *virtual partition* method of El Abbadi, Skeen and Cristian [1], each site maintains a view of all operating sites it can communicate with and applies a rowa strategy within this view.

Other dynamic quorum based methods of this type are the *vote reassignment* of Garcia-Molina, Barbara and Spauster [11], the *quorum adjustment* of Herlihy [47], and the *dynamic voting* of Jajodia and Mutchler [51]. In the vote reassignment and the quorum adjustment methods, sites that are up can change their votes on detecting failures of other sites by following a particular protocol. The dynamic voting method stores additional information regarding the number of sites at which the most recent update is performed. Another dynamic algorithm is the *available copies* of Bernstein and Goodman [13]. In this method, query transactions can read any single site, while update transactions must write at all sites that are up. To guarantee correctness, each transaction must perform two additional steps called *missing writes* and *access validation*.

Voting with witnesses, proposed by Paris [75, 76], is an interesting idea that can be applied to all of the vote reassignment algorithms. Certain sites are witnesses; they have full voting rights but do not maintain a physical copy of the replica, only its last version number. Such inexpensive entities can enhance the performance of the protocol as they attest to the state of the replicated data object. Van Renesse and Tanenbaum presented a similar scheme, *voting with ghosts* [80], in which some sites do not even keep the state of the replica. These ghosts are used to to detect that a copy is unavailable due to a node crash, not due to a network partition.

Kumar [53], and Agrawal and Abadi [2] introduced *hierarchical quorums* to limit the size of the quorums

as the number of processors increases. In [45] the notion of coteries[4] is proposed as an alternative to quorums. A *coterie* is a collections of intersecting minimal subsets of sites. Interestingly, if the system has six or more nodes, there exist several coteries that do not have equivalent quorum assignments. A logical structure based on *grids* is also presented in Cheung, Ammar and Ahamad [26]. Ahamad and Ammar [4, 25] evaluated the performance of quorum based protocols when data becomes unavailable. Barbara and Garcia-Molina [9, 10] studied the vulnerability of voting assignment in the presence of failures. Kumar and Segev [54] proposed heuristics to minimize communication costs under availability constraints. Spasojevic and Berman [84] designed an algorithm for computing optimal static pessimistic vote assignments when relative frequencies of read and write operations are known.

The idea of regenerating replica has also been employed by Pu, Noe and Proudfoot [78] to restore the level of replication when the system recovers, provided that at least one copy has survived.

In the *primary copy* method [74, 73], one replica is designated as master and the others as backups or slaves. The master sites orders the sequence of operations and propagates these updates to all sites. As all updates have to go through the master, the primary site may exhibit performance bottlenecks. It may also become unavailable if a failure occurs. At that time, an *election* algorithm is initiated to decide on a new leading copy [44]. A similar type of centralized scheme is based on the passing of a token. The holder of the token is given the privilege of updating the replica and propagating updates. When the token is lost, the system has to reconfigure to regenerate it. The primary copy method has been used in high available file systems [58] or main-memory databases [82].

Another tactic to enhance availability is to use system semantics, or application dependent information. Probabilistic algorithms and lazy-replication are two examples of this approach. Probabilistic algorithms were proposed for applications with weaker consistency requirements. If copies are allowed to diverge temporarily, diffusion techniques become applicable. The protocols ensure that a replica will eventually be up-to-date with probability one as time passes. Efficient diffusion of update messages involve the construction of a minimum spanning tree over the recipient sites. This construction is trivial with a global view of the network, but becomes very delicate without this global vision since failures can occur during the construction. This is the case in a distributed asynchronous system as a site does not know the state (working or failed) of the other sites. Flooding is an another classic diffusion technique: when a site receives an update, it propagates this update to its neighbors and so on. More recently, *epidemiologic* algorithms based on rumors or viruses were also proposed for the management of replicated data [35, 40].

Lazy replication, proposed by Landin and al. [55] for the Gossip system, enhances response time and availability, provided that the user has specified the potential causal dependencies among requests. Unrelated requests do not incur any latency due to communication delays. It relies on gossip, or lazy propagation of updates. This approach can be very beneficial for particular classes of application, like a mail service.

2.2 Process Group Communication

A number of group communication[5] algorithms have been proposed to guarantee reliable delivery and some defined order of messages for all recipients. Most of them are based on variations of two- or three-phase commit, central protocol, or token ring. In addition, a membership protocol guarantees that processes view configuration changes in a unique order, receiving the same messages between two configuration changes. With such services, processes appear to be virtually synchronous.

Distributed agreement, say on a particular order or group membership, has been addressed within the Byzantine Generals framework [38]. Typical models within this framework have assumed synchronized

[4] Webster's definition: "a coterie is a close circle of friends who share a common interest."

[5] Group communication refers to the abstraction of a group of processes cooperating by exchanging messages. Multicast and broadcast are two hardware mechanisms that can be used to implement the abstraction of group communication. Broadcast messages are received by all processors in the network, while multicast messages explicitly select the set of destinations. In this paper, we often interchange the use those terms.

processors and systems in which malicious processors try to prevent the protocol from reaching consensus [32]. We present below protocols for more realistic systems models.

In this work, we do not consider synchronous systems in which upper bounds are placed on communication and processing delays. They are designed with strong assumptions to cope with the worst possible situation, but may not take advantage of the average case.[6] The Advanced Automation System [29] of the Federal Aviation Administration is an example of these systems. Fault-tolerance is achieved through server group replication on independent redundant hardware. Server replication is supported by underlying synchronous services like group communication [32, 28], membership changes [30] and clock synchronization [31].

The V system [24] first introduced the notion of group communication, but without guaranteeing the delivery or order of messages. Current extensions of the IP layer provide similar services [34]. As in most early solutions, the communication topology is assumed to be fully connected and point to point. The channels are also reliable (eventual, exactly once delivery of uncorrupted messages) and first-in first-out.

Chang and al. [23] proposed a family of atomic broadcast protocols using a token revolving in a *token list*. One site, the token holder, defines a unique order for all messages. When a message is sent, the token site sends an acknowledgment containing a timestamp. The protocols tolerate L failures as follows: to deliver a message, the sequencing token has to be transferred L times after the message transmission, and L token holders need to acknowledge the message. This scheme introduces a very long delay before a message can be delivered. It requires also between two and three broadcast messages per atomic broadcast.

Kaashoek and al. [52] proposed another type of *sequencer* based protocol for the Amoeba operating system. A unique broadcast server, the *sequencer*, imposes a total ordering on all updates originating from all broadcast servers. Their protocol differs from Chang in that the sequencer is fixed. If the sequencer goes down, the invitation algorithm of Garcia-Molina [44] is invoked to elect a new sequencer. Positive and negative acknowledgments are used to ensure that broadcast updates are received by all group members.

In Isis, Birman and al. [15] developed a family of multicast primitives with different order semantics. The CBCAST primitive uses vector timestamps to order messages according to their causal dependencies. This primitive is used by the ABCAST protocol to extend the causal order and deliver messages in the same order at all sites. Messages to be totally ordered are queued and marked as "undeliverable." A special receiver, the *token-holder*, multicasts a *set-order* message which contains information about the order of messages received. This set-order message causes all other receivers to sort their local buffers and deliver the messages from the top of the queue. A reimplementation of Isis, called Horus [79], achieves very high performance by using IP-multicast and packing multiple messages in a single network packet.

Psync [77] is a multicast facility based on causal messages. The primitive is implemented using a single message, but group members wait before delivering it. At most, one message from each member must be received before the total order can be established. It supports also a semantic ordering, delivering messages early by giving a priority to read operations. Failure notifications are causally ordered, but the membership protocol does not guarantee that all group members view the same sequence of configuration changes (virtual synchrony).

Luan and Gligor [59] derived an atomic broadcast from the three phase commit protocol. In the first phase, a process initiates a voting round. When enough acknowledgments have been collected, the initiator enters a notification phase to specify which messages need to be ordered. If enough acknowledgments are also received, a "commit" message causes all receivers to extend their global order by the messages identified in the notification message. With n sites, the protocol requires $4n$ messages under normal operation.

More recent protocols take advantage of the communication medium. Indeed, simulating reliable point to point links over a broadcast network severely limits performance. A simple implementation of a two-phase commit for n processes using tcp/ip generates as much as $10 * n$ packets instead of one, provided that the

[6] Arjomandi, Fischer and Lynch discuss [8] the relative performance of synchronous and asynchronous systems. Although synchronous systems are usually slower, it is possible to find problems for which an asynchronous system is provably slower than a synchronous one.

network supports broadcast, and messages are in some linear order [17].

Amir and al. [7] proposed in Totem a logical token passing scheme for broadcast channels. It achieves good performance as the token coordinates the transmission and flow control of broadcast messages. However, as with all token schemes, the algorithm is not fully distributed and the system has to reconfigure when the token is lost.

From a theoretical standpoint, the requirements for distributed agreement in a broadcast domain are given by Moser, Melliar-Smith and Agrawala [69]. They proved that a necessary and sufficient condition for the existence of a deterministic consensus protocol is delivery of each broadcast message to at least $\lceil (n+k+1)/2 \rceil$ process in an n-process system subject to k crash failures. In their broadcast model, if a message is delivered, it is delivered immediately and in order but not necessarily to all processes.

In the Trans system [62, 61], they designed a reliable broadcast primitive for unreliable datagrams. It avoids the several rounds of message of most consensus agreement by placing a total order on messages *only* with a high probability. Messages contain piggyback information to perform recovery and decide which process has received a particular message. A message m_p sent by p contains two lists of message identifiers, $ack(m_p)$ and $nack(m_p)$. All i in $ack(m_p)$ have been received by p (positive acknowledgments), and all j in $nack(m_p)$ have been lost by p and need to be retransmitted (negative acknowledgments). Using the transitive closure of these acknowledgments, they derived a partial order which can in turn be converted into a total order [68]. For a group of 10 processors, a message can be delivered on average after receiving 7.5 messages.

They also designed a membership algorithm [67] that operates on top of the total ordering, but suffers from its inherent problems. A majority of the processors need to be operational and there is a small chance that messages cannot be totally ordered if failures occur. Furthermore, even if reliable sends and receives are achieved efficiently, the processing cost of the total ordering protocol is high: the delivery predicate is not built incrementally, and the ordering has to take into account the recovery of lost messages.

Dolev and al. refined this approach in the Transis system [6]. If i is in $ack(m_p)$ then p has received i, *all* j in $ack(i)$, *and so on*. In other words, if i is in $ack(m_p)$ then p has received i, *and all* j in $ack^*(i)$, where ack^* is the transitive closure of ack.

Therefore, acknowledgments directly create a causal order, simplifying the total ordering (ToTo) and group membership protocol. Both protocols use this partial order and have a reasonable processing cost. The ordering protocol tries to deliver messages early, before a message is received from each processor, by maintaining a graph in which messages are arranged by the causal relation. Progress is guaranteed as it operates on top of the membership protocol. The latter has also the unusual capability of supporting more than one network partition [39].

As we studied the ToTo protocol, we found that the delivery rules are somewhat redundant and we took on the quest of making the delivery even more aggressive.

2.3 Replicated Databases And Atomic Data Types

Distributed and replicated databases have long been used in applications where the manipulation and preservation of on-line data is of primary importance. In the presence of concurrency or hardware failures, the database maintains the integrity and consistency of the data. The data items are individually read and written by the operations of programs called *transactions*. Consistency is supported by making the transactions *atomic*. Atomic transaction are characterized informally by two properties: serializability and recoverability [85]. *Serializability* means that the concurrent executions of a group of transactions is equivalent to some serial execution of the same transactions. *Recoverability* means that each transaction appears all-or-nothing. Either it executes successfully to completion (in which case we say that it *commits*), or it has no effect on data shared with other transactions (in which case we say that it *aborts*).

Since the replication of data should be transparent to the user, the interleaved execution of his transactions on a replicated database should be equivalent to a serial execution of those transactions on a one-copy database. Such executions are called one-copy serializable. Many concurrency control protocols have been proposed to guarantee one-copy serializability. Most of them fall into three categories: locking, timestamp, and optimistic (certification) algorithms. We refer the reader to the texts by Bernstein, Hadzilacos, and Goodman [14, 12] for a good survey of the algorithms, and Lynch, Merrit, Weihl and Fekete [60] for a general theory of atomic transactions.

Initial work on concurrency control left the data uninterpreted, or viewed operations as simple reads and writes. Recently, several researchers have considered placing more structure on the data to increase concurrency [50, 49, 3]. Such techniques can be used in databases to deal with "hot spots," or frequently accessed data. One approach is to use the specifications of abstract data types to design type-specific concurrency control algorithms. In this case, atomicity is achieved through the shared data, which must be implemented in such a way that the transactions using them appear to be atomic. Objects that provide appropriate synchronization and recovery are called *atomic objects*, and are accessible only through a set of primitive operations.

Weihl presented such concurrency control algorithms for abstract data types [88, 89, 91]. The algorithms ensure the serializability of transactions by using conflict relations based on the commutativity of operations (if certain operations of a type commute, less synchronization is required to maintain correctness). The algorithms are based on locking: to execute an operation on an object, a transaction must acquire a lock on the object in a mode appropriate for the operation. The algorithms ensure a local atomicity property called *dynamic atomicity*[7], which means that if every object in a system is dynamic atomic, transactions will be atomic [90].

The algorithms are based on a simple intuition: two operations conflict if they do not commute. For reads and writes, two operations conflict if one of them is a write. For abstract data types, the commutativity definitions are somewhat more subtle. Weihl defined a notion of commutativity dependent of the object's state, thus permitting more concurrency than otherwise possible. The state is captured by considering an "operation" as being both the invocation *and* the result of that invocation. For example, a successful withdraw operation on a bank account X is noted:

X:[withdraw(amount), ok]

This framework is quite general, permitting operations to be both partial (if they fail) and non-deterministic (if an operation can have more than one result, as for dequeuing any item from a bag). Weihl's two notions of commutativity, forward and backward, differs by the constraints they place on recovery. To guarantee dynamic atomicity (serializability *and* recoverability), forward (resp. backward) commutativity has to be used in conjunction with a deferred update (update in place) recovery strategy. A formal definition of these two notions and their impact on recovery is beyond the scope of this paper. The curious reader may consult [60] for further details.

This idea of commutativity can be applied in general distributed systems to increase concurrency. We will use it to allow different sites to invoke operations on objects in different orders. If consistency is maintained, less synchronization will offer better performance.

3 Group Communication

As processes replicate to provide fault-tolerance and availability, they need to cooperate when executing a global event – updating a replica for example. Even though groups of processes arise naturally in distributed

[7] Dynamic atomicity is in fact optimal: no strictly weaker local constraint on objects suffices to ensure global atomicity of transactions.

systems, little support is generally available to handle group communication communication patterns and programming. These issues are left to the developer who is unable to cope with uncertainties due to random communication delays and component failures. These problems prevent processes from having the knowledge of global system states that shared storage (like shared variable) provides in a centralized system [28].

Process group communication is an attractive abstraction to structure fault-tolerant distributed applications: it supports primitives to send and receive messages from 1 to n destinations. The semantics of these multicast primitives are simple, powerful, and easy to understand. The programmer views the system as if it was synchronous: message delivery and configuration changes are reported in the same order at all sites. This execution model, "virtual synchrony", is based on several properties. Even though variations among systems exist, we can identify core principles [15, 6, 32, 7].

1. Reliable atomic ordered delivery. A message is eventually received by all or no working process. Messages are efficiently disseminated in the network, recovered if they get lost, and delivered in the same order at all sites.
2. Group membership. Processes have a local view of the operational processes and go through the same sequence of configuration changes. Failure are reported according to the fail-stop model. When a process fails, this event is detected by all working processors, and communication from it to other processes is stopped.
3. Coordination of membership service and delivery of regular messages. Between two configuration changes, every pair of working processes receive the same set of messages.

In this report, we assume that we are building an atomic multicast primitive on top of a reliable causal broadcast and group membership service. These two services are presented in the following sections. The reliable broadcast protocol provides us a causality graph in which messages are arranged by a "happened before" relation, and the membership service guarantees the liveliness of the total ordering procedure. Either we receive a message for every processor in the group view within a finite amount of time, or it is removed from the view.

3.1 System Model

As the notions of order are central to our discussion, we defined them as follows:

Definition 1 *A partial order on the set A is a relation $\rho \subseteq A \times A$ that is reflexive ($\forall x \in A,\ (x,x) \in \rho$), anti-symetric ($\forall (x,y) \in A^2,\ [(x,y) \in \rho \wedge (y,x) \in \rho] \Rightarrow x = y$) and transitive ($\forall (x,y,z) \in A^3,\ [(x,y) \in \rho \wedge (y,z) \in \rho] \Rightarrow (x,z) \in \rho$).*

Definition 2 *A strict partial order on the set A is a relation $\rho \subseteq A \times A$ that is irreflexive ($\forall x \in A,\ (x,x) \notin \rho$) and transitive.*

Definition 3 *A partial order ρ on the set A is total or linear if $\rho \cup \rho^{-1} = A \times A$. In this case (A,ρ) is called a chain. In other words, ρ is a total order if $\forall (x,y) \in A^2$ we have either $(x,y) \in \rho$ or $(y,x) \in \rho$.*

Definition 4 *The minimal set according to the a partial order ρ on a finite set A is defined $Low(A,\rho) = \{a | a \in A \wedge \forall b \neq a \in A,\ (b,a) \notin \rho\}$*

Sites. We consider a set S of n sites geographically distributed. Each site has a unique identifier $S_i,\ 1 \leq i \leq n$. Sites identifiers are order by the relation $<$ and $(S,<)$ is a chain. Each site has a local monotically increasing clock, or counter.

Topology. Sites are interconnected by a communication network. The topology of the network is modeled as a graph T in which the vertices correspond to the sites and the edges represent the communications links.

Asynchronous system. No assumption is made on the existence of a global clock or synchronization mechanism. Sites communicate only by the exchange of messages. Communication and processing delays are finite but unpredictable. Each message has a non-zero probability of arriving intact at its destination. We also use the term process as an alternative to site, as no common memory or direct synchronization between processes is allowed in this model.

Communication. The network supports some form of hardware multicast. A site can send a message to a group of sites. The network is unreliable, messages can get lost. If the message arrives but corrupted, or if the local reception buffer is full, the message is discarded. Messages can be received in different orders at different sites, and in a different order than the order in which they were sent.

Failures The system supports omission, crash, and partitions failures but not Byzantine failures. Faulty processes do not behave maliciously and do not collude with other faulty processes.

Objects and histories. Objects encapsulate data. Each object has a type which defines a set of possible states and a set of primitive operations that provide the only means to create and manipulate objects of that type. In the absence of failures or concurrency, a computation is modeled as a history, which is a sequence of object-operations pairs. An operation is written as $op(args*), term(res*)$, where op is an operation name, $args*$ a sequence of arguments values, $term$ a termination condition, and $res*$ a sequence of result values. For example, the following is a history of a queue object q:

q:[Enq(x),Ok()]
q:[Enq(y),Ok()]
q:[Deq(),Ok(x)]

An object sub-history $h|x$ (h at x), is the subsequence of object-operation pairs whose object names are x. Each object has a serial specification, which defines a set of legal histories for that object. A history h is legal if each object sub-history $h|x$ lies within the serial specification for x.

3.2 Reliable Communication

At the transport level, we used the reliable causal multicast service of Trans [61] / Transis [6]. It ensures that

1. every message broadcast or received by a working processor is received by any working processors
2. message delivery preserve causality. Intuitively, a response is never delivered before the corresponding request. This ordering is much like FIFO guarantees for point to point communications.

The notion of causality was formalized by Lamport in his seminal paper [56]. The set of events in a distributed system is structured by "happened before" relation, which is also termed causality. This relation, " $\rightarrow$," is the smallest relation satisfying the three following properties:

1. If a and b are events in the same processor, and a comes before b then $a \rightarrow b$.
2. If a is the sending of a message by one process and b is the receipt of the same message by another process then $a \rightarrow b$.
3. If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two distinct events a and b are said to be *concurrent* if $a \not\rightarrow b$ and $b \not\rightarrow a$. We assume that $a \not\rightarrow a$.

In Transis, each newly emitted message carries a list of positive acknowledgments that preserve causality:

Lemma 1 *Let $m_{i,j}$ be the jth message sent by process i, and $ack(m_{i,j})$ the list of message identifiers carried by $m_{i,j}$. We have $\forall m \in ack(m_{i,j}) \Leftrightarrow m \rightarrow m_{i,j}$*

This property is fundamental as we can make the observation that if m in in $ack(m_{i,j})$, then m happened before $m_{i,j}$. In particular, it was already received and processed by site i before i sent $m_{i,j}$. Similarly at site k, $m_{i,j}$ will be delivered by Transis after all messages in $ack(m_{i,j})$ have been delivered (including m). Transitively, all messages $n \in ack^*(m_{i,j})$ were already present at site k before k delivered $m_{i,j}$.

Positive and negative acknowledgments are used to propagate broadcast messages efficiently and reliably. We refer the reader to [61, 6] for details on how message recovery and delivery is done. The main advantage of this protocol is that in a failure-free environment, one single message needs to be broadcast per atomic broadcast.

3.3 Causality Graph

The Transis causal multicast service is used by our protocol to decide who has received a particular message and in what order. Each processor p maintains a local data structure, G_p, of the messages delivered by Transis. These messages are causally arranged in the graph G_p, defining an ordering *context* for each message. By definition of the causal multicast service, message m can enter the graph only after all messages acknowledged by m (all i in $ack(m)$) have been already delivered, and are already present in G_p.

Definition 5 *A causality graph $G = (V, E)$ is a graph defined for a set P of processes, such that:*

1. *V is a set of messages. $m_{i,j} \in V$ denotes the jth message sent by processor $i \in P$.*
2. *E is a set of edges. $(a, b) \in E \Leftrightarrow a \rightarrow b \Leftrightarrow a \in ack(b)$*

This graph has several properties.

Lemma 2 *G is acyclic.*

Proof: "$\rightarrow$ " is a strict partial order relation. If G contains a cycle, and as "$\rightarrow$" is transitive, $\exists (x, y) \in A^2$, $x \rightarrow y \wedge y \rightarrow x \Rightarrow x \rightarrow x$ which is a contradiction as "$\rightarrow$" is irreflexive. □

Lemma 3 *If a group of processes receive the same set of messages, G is the same for all of them.*

Proof: As causality is defined by the acknowledgments present in each message, if any two processes receive intact the same messages, they construct the same graph. □

Note that any reliable causal multicast primitive can used to construct the causality DAG. We just used Transis for its efficiency. Before presenting our atomic broadcast primitive and total ordering algorithm, we still need another protocol to cope with dynamic groups of processes, namely a membership service.

Figure 1 depicts an example of causality graph for four processes. Messages A1 and B1 are concurrent; Message B2 acknowledges message A1, therefore A1 happened before B2.

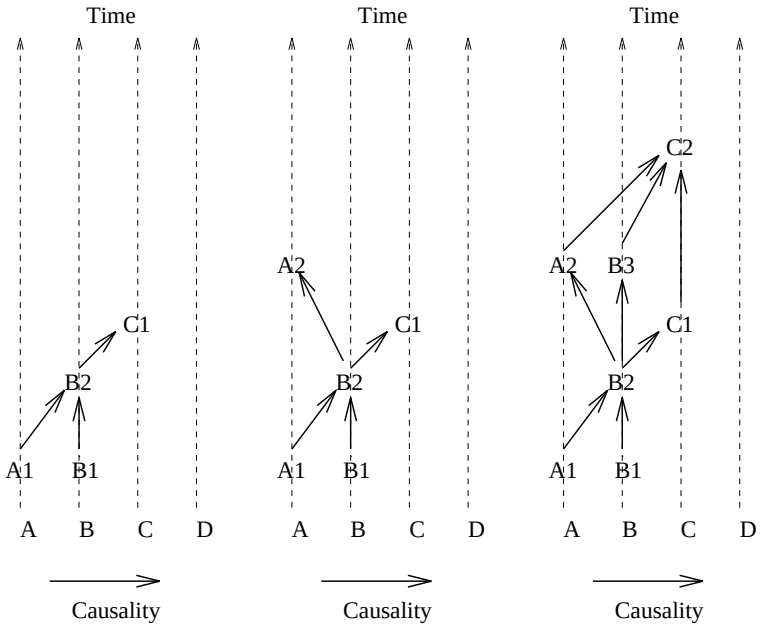


Figure 1: Evolution of a causality graph

3.4 Group Membership

In a dynamically evolving distributed system where processes can fail to communicate, crash and recover, maintaining agreement among group members on group membership is a fundamental problem. Even if it is impossible to solve the consensus problem in a true asynchronous system [42], due to the inability to distinguish between communication delays and crash failures, we can use unreliable failure detectors (like timeouts) to emit suspicions of processor failures [20].

If a processor is suspected to have failed, it is removed from the group. The notion of failure in this model is weak in the sense that a process has failed if it is not a member of the group. The role of the membership service is to provide the illusion of a fail-stop environment via membership changes given by a single, non-faulty process [81]. If the network partitions, it needs to track a primary partition which is the only partition where progress is allowed. Members of the primary partitions take actions on behalf of the system as a whole. If a removed process has not crashed, but was only temporarily unreachable, subsequent communication from it to the rest of the system is inhibited. The process will eventually realize that it is no longer part of the group and can start to re-enter the group.

Even though the problem seems straightforward[8], a formal definition of the membership problem and its interaction with the flow of regular messages is worth presenting since subtle issues can arise:

1. Without a global clock and in an environment prone to failures or partitions, the notion of group view and membership agreement can only be defined in relation to consistent snapshots of the system, for which a global state can be constructed.

2. The membership protocol may or may not accommodate cascaded joining and/or failures during its execution.

[8] Membership consensus is closely related to the election problem. It is a similar agreement problem that appears so innocent that David Bell and Jane Crimson wrote in *Distributed Database Systems* that "The election protocol is straightforward. All sites involved in the election (i.e. all those which are operational) agree on a linear ordering of sites and the first site in this ordering is elected as the new coordinator." The vast research literature on the subject seems to suggest otherwise.

3. The protocol might block the flow of regular messages.
4. It might also be asymetric, with an accepting side and a joining side, or support joining of multiple processes at a time.
5. It might provide eventual agreement but not virtual synchrony if processes can go through different configuration changes.
6. If more than one primary partition are allowed to make progress, the virtual synchrony rule need to be extended: any two processes of the *same* group view should receive the same set of messages between two configuration changes.

We give below a summary of definitions to solve the membership problem. A formal and extended version of the problem statement appears in [81].

Definition 6 *Given a consistent cut c of the system, as defined by Chandy and Lamport in [21], a set S of processes are in membership consent on c if all working processes of S have identical local views of the group.*

Definition 7 *A membership protocol should have the following properties:*

1. *Triviality. A group view eventually exist.*
2. *Validity. A process is declared faulty if it is not in the local view.*
3. *Uniqueness. Only a primary partition is permitted. Non-null group views are unique along all consistent cuts.*
4. *Sequence. Members of the primary partition see the same sequence of configuration change.*
5. *Liveness. If a process is suspected of failure (respectively of joining), it is removed from (added to) the view within a finite amount of time.*

In addition to the membership consensus problem, the group communication abstraction and its virtually synchronous execution model also requires that regular communication be coordinated with configuration changes.

1. Regular messages from processors not in the current group view are discarded and ignored.
2. Between two configuration changes, each working member of the primary partition delivers the same set of messages.

To provide the above services, we used the group membership service of Amir, Dolev, Kramer, and Malki [5] for its adequacy to the Transis environment. It operates on top of the causal ordering, and is genuinely distributed. This protocol:

1. maintains the current configuration in consensus among connected and active processes (Definition 7);
2. handles partitions and merges of multiple processors (cascaded joins and failures);
3. handles merges in a symmetrical way;
4. does not block the regular flow of messages while configuration changes take place;
5. provides "virtual synchrony": it guarantees that members of the same configuration receive the same set of messages between every pair of configuration changes.

6. Although it allows multiple partitions to make progress, we did not use this facility. For a replicated main memory database, we believe that a primary partition scheme is more appropriate. If two partitions merge, it is not clear to define which one has valid data. The primary partition was defined as the partition, if it exists, containing more than half of the sites.

3.5 Agreed Communication

The agreed multicast primitive completes the set of services needed to provide a virtually synchronous environment to the programmer. The implementation of this service is the focus of the following chapter. This causal, reliable, atomic group multicast primitive delivers messages in the same order at all sites.

Definition 8 *The Agreed multicast service is defined with the following two axioms.*

Correctness. *Let m, m' be two messages.*

- *If m happens before m' ($m \rightarrow m'$), then all operational machines deliver m before m'.*
- *If m and m' are concurrent, all operational machines deliver either m before m', or m' before m.*

Liveliness. *For every pair of machine (p, q) in the current group view P, there exist a time interval Δ, such that for all T, between T and $T + \Delta$, either p delivers a message from q, or q is declared faulty and removed from the group.*

4 Atomic Group Multicast

To implement the *agreed* multicast service, we designed several total ordering protocols (G-Top, LG-Top and H-Top). All sites have to use the same protocol, so that they can deliver messages in the same unique order. Messages enter a protocol in causal order, are possibly delayed but are delivered however within a finite amount of time.

Let $\mathcal{C}$-$deliver_p(m_{i,j})$ and $\mathcal{A}$-$deliver_p(m_{i,j})$ be the events at processor p corresponding to the delivery by Transis and Top, of the jth message sent by processor i. The main objective of our protocol is to reduce the latency or time interval between those two events.

$$\mathcal{C}\text{-deliver}(m_{i,j}) \rightarrow \boxed{\text{Total Ordering Protocol}} \rightarrow \mathcal{A}\text{-deliver}(m_{i,j})$$

As the protocol operates on top of the membership service, it needs to accommodate two other events corresponding to the situation where a process, say q, is included ($join_p(q)$) or removed ($leave_p(q)$) from the group view.

4.1 Sketch Of The Total Ordering Protocols

The flow of the protocol can be described as follows: it loops indefinitely, delivering one set of messages at a time. All sites deliver the same group of messages S_1, then group S_2, and so on. The delivery of each set S_i corresponds to an activation (or wave) of the protocol. As the sequence $S_0, S_1, S_2, \ldots$ is unique, each wave i clearly defines a set of messages S_i, referred to as *source* messages. Messages in S_i are then delivered deterministically.

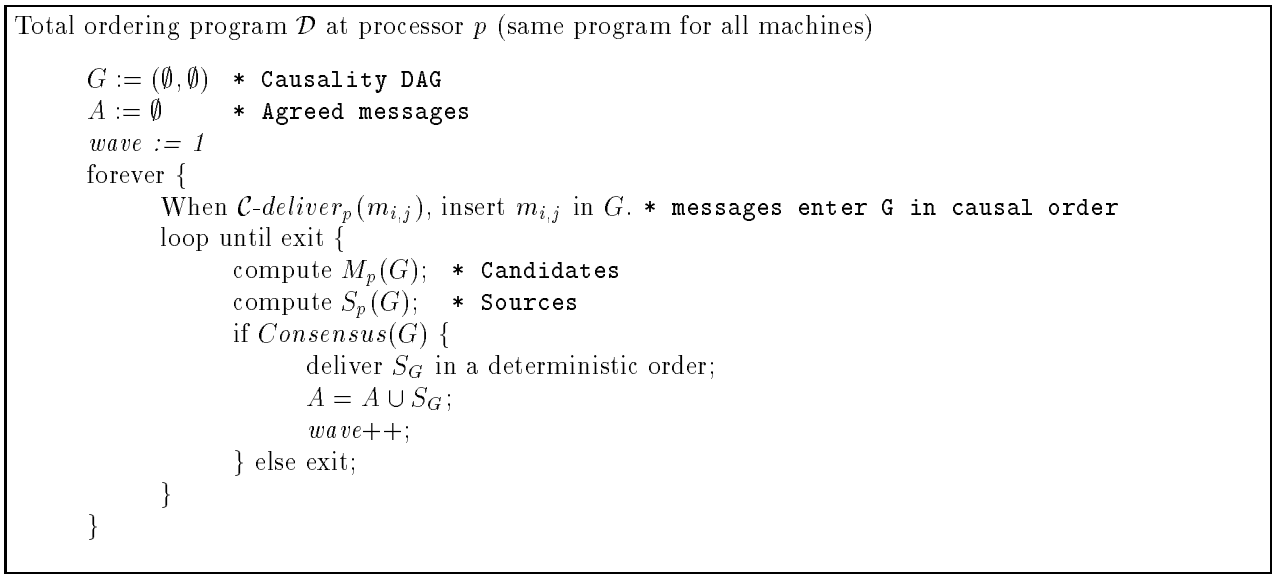


Figure 2: Protocol sketch

A pending message is defined as a message submitted to the protocol that has not yet been delivered. A pending message considered to be inserted in the set of sources is called a *candidate* message. A wave is complete when the set of sources S cannot change, even if more messages are received. As messages are eventually received by all sites, they all agree on the composition of S. In other words, when the set of sources S is *stable*, a *consensus* is reached, and S can be safely delivered (Figure 2).

In the following sections, we define more formally the candidates, the sources, the consensus decision, and also the deterministic order used in each wave to deliver sources messages to the application.

The candidates are defined as follows:

Definition 9 *Candidate message.*

$$m \in M_p \Leftrightarrow m \text{ follows in } G \text{ only delivered messages } \Leftrightarrow m \text{ is a root of } G$$

A simple algorithm for totally ordered delivery is to wait until there is a message from each machine in the causality graph (*Consensus*), and delivering the roots (*Sources*) of the dag in a deterministic order. This approach is very similar to putting a timestamp on messages and delivering messages in timestamp order [56]. While this algorithm provides total ordering at no additional cost in messages or acknowledgments, it delays delivery until every processor has sent a message. The latency can be large, and it forces the system to operate at the speed of the slowest processor. Using the causality information already carried by the messages, we can give priority to the first messages acknowledged by a *majority* of processors.

4.2 Early Delivery

We characterize below sufficient conditions for any protocol $\mathcal{D}$ to deliver messages in a unique order.

Delivery is triggered when an *consensus* predicate is satisfied. At this point some number of candidate messages, the *sources*, can be delivered if the protocol is guaranteed that every other processor will deliver the same set of messages. Thus the set of source messages must be stable, namely remain the same in every possible extension G' of the current DAG G. Stability is a form of safety condition: nothing bad will ever happen if the sources are delivered as soon as they are stable.

We note that G' extends G if and only if $G \subseteq G'$. We define

$$Future(G) = \{G' | G' \text{ extends } G\}$$

In addition, we define the set of processors that have a message in G as:

$$tail(G) = \{p_i | \exists m_{i,j} \; st. \; m_{i,j} \in G\}$$

The set of sources is stable if for all possible extensions G' of G (when more messages are received) a) every non source candidate message in G cannot become a source in G'; b) every source message in G is also a source in G'; c) unseen messages cannot become sources in G'.

Definition 10 *Stability conditions (Safety)*

1. **Candidate stability:** *If $m \in M_G \setminus S_G$, then $m \in M_{G'} \setminus S_{G'}$ for all $G' \in Future(G)$.*
2. **Source stability:** *If $m \in S_G$, then $m \in S_{G'}$ for all $G' \in Future(G)$.*
3. **External stability:** *If $i \notin tail(G)$, then for all j, $m_{i,j} \notin S_{G'}$ for all $G' \in Future(G)$.*

The idea of *early delivery* is to devise a stability rule that works without requiring that a message be delivered from every processor. If G satisfies candidate, source, and external stability then G is said to be *stable*. While the stability conditions will guarantee that every processor will deliver the same set of messages, we still need to guarantee that eventually, some messages are delivered (something good will eventually happen). Thus, we need the following progress conditions:

Definition 11 *Liveliness condition*

1. *If $|tail(G)| = n$, then $S_G \neq \emptyset$.*
2. *If $|tail(G)| = n$, then G is stable.*

Lemma 4 *Suppose that messages are delivered at processor p only when the delivery protocol $\mathcal{D}$ applied to G_p satisfies the three stability conditions. Let S_p^1 be the first set of sources delivered by processor p. Then $S_p^1 = S_q^1$ for every $p, q \in P$.*

Proof: Suppose that there exists $p, q \in P$ such that $S_p^1 \neq S_q^1$. Then there is a message m such that $m \in S_p^1$ and $m \notin S_q^1$. Let G_p (respectively G_q) be the causality DAG that is first stable at p (q). Lemma 3 and the virtual synchrony property of the membership service guarantee that there exists a common extension of G_p and G_q, which include $\mathcal{G} = G_p \cup G_q$. Suppose that m is a candidate message in G_q. By candidate stability, m is not a source message in every extension of G_q, including $\mathcal{G}$, so m is not a source message in $\mathcal{G}$. By source stability, m is a source message in every extension of G_p, including $\mathcal{G}$. Therefore, m both is and isn't a source message in $\mathcal{G}$, a contradiction. Therefore, m cannot be a candidate message in G_q, and thus has not yet been delivered to q (m is a root of G_p). But then, by external stability, m is not a source message in every extension of G_q, including $\mathcal{G}$, leading to another contradiction. Therefore $S_p^1 = S_q^1$. □

Each wave of this protocol considers only the messages in G that have not been totally ordered (yet). In fact, $\mathcal{D}$ maintains only the most recent portion G^* of the DAG G.

Definition 12 *$G^* = (E^*, V^*)$ is the most recent portion of G if given A, the set of messages delivered in all previous waves, it satisfies:*

$$G^* \subseteq G \wedge E^* = E \setminus A \wedge \forall (a,b) \in E^*, \; (a,b) \in V \Rightarrow (a,b) \in V^*$$

As each wave is complete only when a consensus decision is reached on the messages included in the set of sources, A is incremented by steps, at the end of each wave. To support multiple waves, the consensus decision does not consider messages already agreed (present in A). It is therefore defined on G^*. In the following sections, we assume without loss of generality that all messages from previous waves have been removed from $G (G = G^*)$. Note that if $ntail(G) = n$ $(G \neq \emptyset)$, then the set of candidates is non-empty as the "$\rightarrow$" strict partial order relation defines necessarily a non-empty minimal set in G (see Def 4).

Theorem 1 *Suppose that the early delivery protocol $\mathcal{D}$ satisfies the three stability conditions and the two progress conditions. Then $\mathcal{D}$ delivers all messages in the same order at all working processors.*

Proof: The liveliness of the agreed multicast service (Def 8) ensures that every G_p contains within a finite amount of time a message from each processor. Condition (Def 11) then guarantee that every G_p is eventually stable, and with a non-empty set of source messages. Therefore, every working processor eventually delivers the same set $S^1 \neq \emptyset$. Then, apply Lemma 4 inductively to G^*. □

4.3 A Refinement Of The ToTo Delivery Rule

In this section, we present a simplification and improvement of the ToTo early delivery algorithm of Dolev, Kramer, and Malki [37]. We generally follow their terminology, but modify it where convenient to simplify the presentation.

In the following discussion, we define some functions that the early delivery protocol uses. The functions are evaluated on the current causality DAG G at processor p. If there can be confusion about which causality DAG on which processor the functions are evaluated, we annotate the functions. Otherwise, we suppress the extra notation for simplicity.

Definition 13 *Processor vote*

> *The first message $m_{p,i}$ from processor p in G votes for the candidate messages of the dag it causally follows. In addition, a candidate message votes for itself.*

Message $m_{p,i}$ gives one vote to each candidate message it causally follows (the ones he approves of), or one to itself if $m_{p,i}$ is a candidate message.

Each candidate message, m, tracks the processors that voted for it in its *vote vector*, $VV(m)$. The i^{th} component of the vote vector, $VV(m)[i]$ is vote that processor i casts for m, and is defined by:

$$VV(m)[i] = \begin{array}{ll} 1 & \text{If processor } i \text{ votes for } m \\ 0 & \text{If processor } i \text{ votes, but not for } m \\ * & \text{if processor } i \text{ has not cast a vote.} \end{array}$$

We define the *tail* of a candidate message, m, to be the set of processors that sent a message that causally follow m. Then, $ntail(m) = |tail(m)|$. The tail of the causality DAG G is the set of all processors that have a message in G, and $ntail(G) = |tail(G)|$. Let u be the number of processors that have not voted yet (unseen votes):

$$u = |P \ - ntail(G)| = |\{j \ : \ VV(m1)[j] = *\}| = n - ntail(G)$$

We denote the number of votes that a candidate message m receives to be

$$nvt(m) = |\{i \ : \ VV(m)[i] = 1\}|$$

Candidate messages are compared on the basis of the votes they receive. Let $\Phi \geq n/2$ be a threshold parameter. Let $m1$ and $m2$ be two candidate messages. We define a function $votes$ by

$$votes(m1, m2) = \quad |\{j \ : \ VV(m1)[j] = 1 \wedge VV(m2)[j] = 0\}|$$

Candidate message $m1$ wins over candidate message $m2$ if it beats $m2$ in more than Φ votes. We define a function, Win as

$$Win(m1, m2) = \begin{array}{ll} 1 & \text{If } votes(m1, m2) > \Phi \\ 0 & \text{If } votes(m2, m1) > \Phi \\ X & \text{Otherwise} \end{array}$$

The value of Win is defined on the current causality DAG, G. We need a function that tells us about all possible future extensions of G, G'. Thus, we use $Future$, which is all possible future values of Win:

$$Future_G(m1, m2) = \{Win_{G'}(m1, m2) \text{ in } G' \text{ extending } G\}$$

The only interesting ways that G' can extend G is by specifying the votes of the processors that have no message in G. Thus, $1 \in Future(m1, m2)$ if and only if the number of processors that vote for $m1$ but not $m2$, plus the number of unseen votes ("*" votes) is greater than Φ. That is,

$$\begin{array}{rll} 1 \in Future(m1, m2) & \text{If} & votes(m1, m2) + u > \Phi \\ X \in Future(m1, m2) & \text{If} & X \in Win(m1, m2) \\ 0 \in Future(m1, m2) & \text{If} & 1 \in Future(m2, m1) \end{array}$$

Note that

$$\begin{array}{rcl} 1 \notin Future(m1, m2) & \Leftrightarrow & votes(m1, m2) + u \leq \Phi \Leftrightarrow votes(m1, m2) \leq \Phi + ntail(G) - n \\ Future(m1, m2) = \{1\} & \Leftrightarrow & Win(m1, m2) = 1 \end{array}$$

Let M_G be the set of candidate messages in the causality DAG G, The set of *source* messages S_G is defined as

$$i \in S_G \Leftrightarrow i \in M_G \wedge \forall j \in M_G, \ 1 \notin Future(j, i)$$

A message i is a source if it will never lose against any other candidate message j already in G (j cannot *beat* i in more than Φ votes in any possible extension of G).

As configuration changes are reported in a consistent order by the group membership service, the ToTo protocol accommodates failures as follows: A failed process votes, but for no particular candidate message ($VV(m)[i] = 0$ if i has failed). The ToTo protocol is then specified by its delivery rules:

ToTo Delivery Rules

1) (early delivery rule)
   if a) (internal stability)
   $\forall m \in M_G \backslash S_G, \ \exists m' \in S_G, \ Future(m', m) = \{1\},$
   and b) (external stability)
   $\exists s \in S_G, \ nvt(s) > \Phi \wedge \forall s' \in S_G, \ ntail(s') \geq n - \Phi.$
   Then deliver S_G.
2) (default delivery rule)
   Else, if $ntail(G) = n$,
   Then deliver M_G.

The ToTo protocol can be improved and simplified by observing that part of rule 1b) is redundant. This observation leads to the Simplified Total Ordering Protocol (S-Top) protocol:

Simplified Total Ordering Protocol: delivery rules

1) (early delivery rule)
   if a) (internal stability)
   $\forall m \in M_G \backslash S_G,\ \exists m' \in S_G,\ Future(m', m) = \{1\}$.
   and b) (external stability)
   $\exists s \in S_G,\ nvt(s) > \Phi$,
   Then deliver S_G.
2) (default delivery rule)
   Else, if $ntail(G) = n$,
   Then deliver M_G.

S_G is stable when a) all non-source candidate messages will not become sources in the current activation (if they lose once) and b) all unseen messages will not become sources (if a source message gets more than Φ votes, they will lose against it). The observation is that the first part of rule 1b) already guarantees external stability.

Theorem 2 *The S-Top protocol is correct.*

Proof: As messages are inserted in a causal order, a processor p that already has a message in the graph will not vote for an unseen message. The first message from p, m_p, votes for the candidate messages it causally follows which are already in G (or itself if it is a candidate message).

Suppose that the early delivery rule is satisfied. Then there exists a source s with at least $\Phi + 1$ votes. s can share these votes with other candidate messages in G, but all $\Phi + 1$ processors that have voted for s will not vote for an unseen message s'. When G is extended by inserting s', if s' is a candidate message, s' will not become a source message. Indeed, s' will lose against s, as a majority of processors will still vote for s and not for s': $votes(s, s') > \Phi$. If s' gather all unseen votes, $nvt(s) + nvt(s') = n \ \Rightarrow\ nvt(s') < n - \Phi$. As $\Phi \geq n/2$, $nvt(s') < \Phi$. Therefore, $\forall i \in S_G, votes(s', i) < \Phi$ (s' will not take any source message i out of S_G). Therefore, the S-Top protocol satisfies external stability and source stability.

The internal stability rule then guarantees that every process delivers the same set of messages on every activation. A non-source candidate message i will not become a source message if i loses against one candidate message j (if $win(j, i) = 1$ in G, $win(j, i) = 1$ in any extension of G). Thus S-Top satisfies candidate stability.

Finally, the S-Top protocol satisfies the progress conditions since if $ntail(G) = n$, either the early deliver rule is satisfied or all candidate messages are delivered. □

4.4 Generalized Total Ordering Protocol

The S-Top protocol permits the delivery of a message after messages have been received from as few as $n/2 + 1$ processors. Thus, the S-Top protocol will work well if all processors are connected by a bus. If the network consists of several LANs connected by gateways, it might become unlikely that any message will receive half of the votes. Thus, it would be useful to trigger early delivery by messages that receive only a large minority of the votes, or less half of the votes but still enough to agree on a set of sources. The cost to the protocol is a requirement that more total votes be gathered, to ensure external stability.

Again, we define our threshold to be Φ, and we restrict $1 < \Phi < n$. We need a definition of a source message that is simultaneously weak enough to allow the delivery of many source messages, but strong

enough to quickly determine which candidates are non-source messages. We retain the meanings of Win and $Future$ from the ToTo protocol, but we generalize the meaning of a source message as follows:

Definition 14 *Source message.*

$$\begin{aligned} i \in S_G \Leftrightarrow & \quad i \in M_G \wedge [nvt(i) > \Phi \vee \forall j \in M_G, 1 \notin Future(j,i)] \\ \Leftrightarrow & \quad i \in M_G(\Phi) \wedge [nvt(i) > \Phi \vee \forall j \neq i \in M_G(\Phi), votes(j,i) + u \leq \Phi] \end{aligned}$$

The early delivery predicate is now specified as follows on G:

Definition 15 *Early delivery predicate*

$$\begin{aligned} R_G(\Phi) \quad \Leftrightarrow \; & \textit{int. stab.)} \quad \forall i \in M_G \backslash S_G(\Phi),\ nvt(i) + u \leq \Phi \wedge \exists j \neq i \in S_G(\Phi),\ votes(j,i) > \Phi \\ & \wedge \textit{ ext. stab.)} \quad u \leq \Phi \wedge \exists i \in S_G(\Phi),\ nvt(i) > \Phi \end{aligned}$$

It is necessary to allow a message with at least $\Phi + 1$ votes to become source message, because otherwise two messages with non-overlapping sets of $\Phi + 1$ votes may block each other from becoming sources. Note that the generalized definition of a source is the same as that for the ToTo protocol when $\Phi \geq n/2$.

With the generalized definition of a source the Generalized Total Ordering Protocol (G-Top) is:

Generalized Total Ordering Protocol: delivery rules

1) (early delivery rule)
   if $R_G(\Phi)$ Then deliver S_G
2) (default delivery rule)
   Else, if $ntail(G) = n$,
   Then deliver M_G.

Proof: Note that when $\Phi \geq n/2$, the G-ToTo protocol reduces to the S-Top protocol. The correctness of the G-Top protocol is a matter of showing that delivery occurs only when source, candidate, and external stability are achieved. Candidate stability is explicitly tested. External stability is ensured because any possible source s must be in the causality DAG G when the delivery occurs. Suppose s is a source not in G when delivery occurs. But, s cannot receive more than Φ votes, because $ntail(G) \geq n - \Phi$. Since the early delivery rule was triggered, there is a source s' with at least $\Phi + 1$ votes, so $1 \in Future(s', s)$. Therefore s cannot be a source. Similarly, s cannot gather enough votes to revoke the source status of a message.

4.5 Hierarchical Delivery Rule

Let $\Phi_1 > \Phi_2 > \cdots > \Phi_k > 0$ be a sequence of delivery thresholds. Let $R_G(\Phi_i)$ be the G-ToTo early delivery rule that corresponds to threshold Φ_i and evaluated on G, and let $S_G(\Phi_i)$ be the corresponding set of sources. let $G'_{u=0} \supset G$ be any possible extension of G such that there remain no unseen votes ($u = 0$).

Suppose that the current causality dag G satisfies $R_G(\Phi_i)$. Then, we can we can deliver $S_G(\Phi_i)$ if and only if we are guaranteed that no future extension G' of G will satisfy $R_{G'}(\Phi_j)$ for $1 \leq j < i$.

If G' might be able to satisfy $R_{G'}(\Phi_j)$ for some $j < i$, then there can be a confusion about which rule should be applied at different processors. We can replace the early delivery rule above by the following:

Definition 16 *Hierarchical delivery predicate*

$$H_G(\Phi_i) \quad \Leftrightarrow \quad R_G(\Phi_i) \wedge \forall j\ 1 \leq j < i,\ \forall m \in M_G,\ nvt(m) + u \leq \Phi_j \tag{1}$$

Hierarchical Total Ordering Protocol: delivery rule (H-Top)

1) (early delivery rule)
   if $\exists i \in [1,k],\ H_G(\Phi_i)$ Then deliver $S_G(\Phi_i)$
2) (default delivery rule)
   Else, if $ntail(G) = n$,
   Then deliver M_G.

Theorem 3 *the H-Top protocol is correct.*

Proof: The proof first explores the necessary and sufficient conditions for $R_G(\Phi)$ to be never satisfied.

By setting $u = 0$, the source set and early delivery rule are simplified as follows on G':

$$i \in S_{G'}(\Phi) \Leftrightarrow i \in M_{G'}(\Phi) \wedge [nvt(i) > \Phi \vee \forall j \neq i \in M_{G'}(\Phi), votes(j,i) \leq \Phi] \tag{2}$$

$$\begin{aligned} R_{G'}(\Phi) \quad &\Leftrightarrow \text{is}) \quad \forall i \in M_{G'} \backslash S_{G'}(\Phi),\ nvt(i) \leq (\Phi) \wedge \exists j \neq i \in S_{G'}(\Phi),\ votes(j,i) > \Phi \quad (3) \\ &\wedge \text{ es}) \quad \exists i \in S_{G'}(\Phi),\ nvt(i) > \Phi \quad (4) \end{aligned}$$

Note that $S_{G'}$ cannot be empty. Suppose $S_{G'}(\Phi) = \emptyset$, therefore $\forall i \in M_{G'},\ nvt(i) \leq \Phi) \wedge \exists j \neq i \in M_{G'},\ votes(j,i) > \Phi$. But if $\exists j \neq i \in M_{G'},\ votes(j,i) > \Phi \Rightarrow nvt(j) > \Phi \Rightarrow j \in S_{G'}(\Phi)$: contradiction. $\Rightarrow S_{G'}(\Phi) \neq \emptyset$.

As $|S_{G'}| > 0$, $\exists i \in S_{G'}(\Phi)$. By definition of $S_{G'}$ we have: $nvt(i) > \Phi \vee \forall j \neq i \in M_{G'},\ votes(j,i) \leq \Phi$

We show that $R_{G'}(\Phi)$ is satisfied if and only if the external condition is satisfied. By definition of $S_{G'}$: $i \notin S_{G'} \Rightarrow nvt(i) \leq \Phi \wedge \exists j \neq i \in S_{G'},\ votes(j,i) > \Phi$ which corresponds to the internal stability of $R_{G'}(\Phi)$. When no more message are expected, a non source candidate message cannot become a source message (in the current wave). $R_{G'}(\Phi)$ can be simplified as:

$$R_{G'}(\Phi) \Leftrightarrow \exists i \in S_{G'}(\Phi),\ nvt(i) > \Phi \tag{5}$$

Given a dag G where $\neg R_G(\Phi)$, we can be sure that $\neg R_{G'}(\Phi)$ if $\forall i \in S_{G'}(\Phi),\ nvt(i) \leq \Phi$. Even if $S_{G'}$ is unknown, as G' is any possible extension of G, we can define similar stability arguments for $\neg R_{G'}(\Phi)$.

1. Internal: no candidate message $i \in M_G$ can ever satisfy $nvt(i) > \Phi$
2. External: no unseen message i message can satisfy $nvt(i) > \Phi$

More formally, we have

$$\begin{aligned} \neg R_{G'}(\Phi) \quad &\Leftrightarrow \text{ is}) \quad \forall i \in M_G,\ nvt(i) + u \leq \Phi \\ &\wedge \text{ es}) \quad u \leq \Phi \end{aligned}$$

which can be simplified as

$$\forall G'_{u=0} \supset G,\ \neg R_{G'}(\Phi) \quad \Leftrightarrow \quad \forall i \in M_G,\ nvt(i) + u \leq \Phi$$

To complete the proof, we can remark that if no extension G' of G can satisfy the delivery rule $R_{G'}(\Phi)$, we can try with $\Phi' < \Phi$. This scheme can be applied recursively, for a finite set of Φ.

$H_G(i)$ is exactly $R_G(\Phi_i)$, provided that $\forall j < i$, no extension G' of the current dag can satisfy $R_G(\Phi_j)$. (If $R(\Phi_1)$ will never be satisfied, we can try $R(\Phi_2)$, and so on).

$$H_G(\Phi_i) \quad \Leftrightarrow \quad R_G(\Phi_i) \wedge \forall j\ 1 \leq i < k,\ \forall G'_{u=0} \supset G,\ \neg R_{G'}(\Phi_j) \tag{6}$$

$$\Leftrightarrow \quad R_G(\Phi_i) \wedge \forall j\ 1 \leq j < i,\ \forall m \in M_G,\ nvt(m) + u \leq \Phi_j \tag{7}$$

It is important for the protocol that all sites apply the *same* sequence of delivery thresholds.

4.6 Early Delivery In A Wave

We make one final observation to permit the delivery of some messages while the activation is being calculated (LG-Top). Each waves defines a set of sources that need to be delivered in some deterministic order. Suppose that $S = \{m_A, m_C, m_D\}$, we can deliver messages in S by process id (lexical order), m_A first, m_C second and so on. However, any other topological sort is also possible, as long as all sites use the same one[9]. Indeed, as source messages are concurrent, any type of sort will extend the causal order to a total order.

Messages can be delivered in a wave by taking advantage of the type of sort applied on S. Suppose sources are delivered using a lexical order. If a message m_A from processor A becomes a source and will not be taken out of the sources when further votes are gathered, then m_A can be delivered even if the protocol has not reached a stable state.

Let $p_1, p_2, \ldots, p_n$ be the n processors in the group. If there exists l such that all messages (seen or unseen yet) from $p_1, \ldots, p_l$ are stable, then we can deliver all messages in the sources up to processor l. A message is stable if it is a) an unseen message that will not be included in the sources, b) a pending message that causally follow other messages, c) a candidate messages that cannot become a source message, or d) a source message that will not be taken out of the source set. The predicates for the four cases are:

- a) stable unseen message $u \leq \Phi \wedge \exists j \in S_G, nvt(j) > \Phi$
- b) $m_i \in G$ is a stable pending non-candidate if $m_i \notin M_G$
- c) $m_i \in G$ is a stable candidate if $m_i \in M_G \backslash S_G, nvt(m_i) + u \leq \Phi \wedge \exists m_j \in M_G,\ Future(m_j, m_i) = \{1\}$.
- d) $m_i \in G$ is a stable source if $m_i \in S_G, nvt(m_i) > \Phi \vee ntail(G) \geq n - \Phi$,

The protocol proceeds as follows: Starting from $i = 1$ to n, if p_i hasn't voted yet (no message $m_{p_i,r}$ from p_i is in G yet), it makes sure that $m_{p_i,r}$ cannot become a source message, and stops otherwise. If p_i has a message $m_{p_i,r}$ in G, it checks whether $m_{p_i,r}$ will not change from candidate to source, and vice-versa (otherwise stops). Of course, if the first message from p_i is a non-candidate pending message, and causally follows other messages, it is stable as it can not become a source (even a candidate) in the current wave. Finally, if $m_{p_i,r}$ is a stable source it delivers it. The pseudo code of the protocol is shown below, and is to be tested in the current activation, every time a new message is inserted in G.

Lexical G-Top program at processor p

```
LG-Top()
{
  G := (∅, ∅)
  wave := 1
  forever {
    When C-deliver(m_{i,j}), insert m_{i,j} in G.
    loop until exit {
      if R_G(Φ) {deliver S_G(Φ); wave++;}
      elseif ntail = n {deliver M_G; wave++;}
      else {L-delivery(); exit;}
    }
  }
}
```

[9] If messages correspond to read and write operations, we can partition the sources into reads and writes, and use in each partition a lexicographic order.

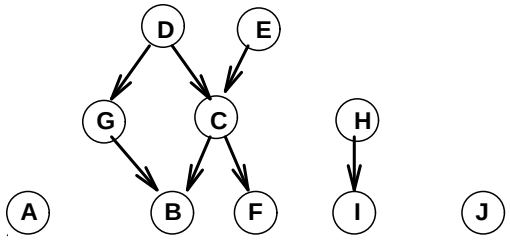


Figure 3: The sources $\{B, F\}$ are stable and can be delivered

```
L-Delivery()
{
  for i = 1 to n
      if p_i has a message in G
      {
          let m_i be the first message from p_i
          if m_i ∈ S_G
          {
              if not (nvt(m_i) > Φ ∨ ntail(G) ≥ n − Φ) return;
              /* m_i is a stable source */
              if not marked m_i {mark m_i ; deliver m_i ;}
          } else if (m_i ∈ M_G)
          {
                  if not (nvt(m_i) + u ≤ Φ ∧ ∃j ∈ M_G Future(j, i) = {1}) return;
                  /* m_i is a stable candidate , continue */
          } else /*m_i ∉ M_G, therefore stable, continue */;
      } else {
          if not (u ≤ Φ ∧ ∃j ∈ S_G, nvt(j) > Φ) return;
          /* the unseen message from i cannot become a source, continue */
      }
}
```

This optimization is straightforward for the G-Top protocol. It is not so practical for the H-Top protocol, as it does not use a predefined Φ. With H-Top, it is cumbersome (though possible) to define a prefix stability predicate for the candidates, as multiple Φ are defined.

4.7 Example

The following example[10] (Figure 3) presents an activation of the protocol. Only the first message from a particular processor is shown. The causality DAG is depicted after the insertion of message H.

$n = 12$ `processors`, $\Phi = 4$ `majority threshold`
$ntail(G) = 10$ `number of processors that have voted`
$u = n - ntail(G) = 2$ `unseen votes`, $M_G = \{A, B, F, I, J\}$ `roots of the dag`

m_i	A	B	F	I	J
$nvt(m_i)$	$\lvert\{A\}\rvert = 1$	$\lvert\{B, C, D, E, G\}\rvert = 5$	$\lvert\{C, D, E, F\}\rvert = 4$	$\lvert\{H, I\}\rvert = 1$	$\lvert\{J\}\rvert = 1$

[10] Neither Total [61] nor ToTo [36] are able to elect messages with such DAG, as they require a candidate set to receive more than $n/2$ votes to be elected.

$votes(row, col)$	A	B	F	I	J
A	$-$	$\lvert\{A\}\rvert = 1$	1	1	1
B	$\lvert\{B, C, D, E, G\}\rvert = 5$	$-$	$\lvert\{B, G\}\rvert = 2$	5	5
F	$\lvert\{C, D, E, F\}\rvert = 4$	$\lvert\{F\}\rvert = 1$	$-$	4	4
I	$\lvert\{H, I\}\rvert = 2$	2	2	$-$	2
J	$\lvert\{J\}\rvert = 1$	1	1	1	$-$

$S_G = \{B, F\}$ is computed as follows:

- $A \notin S_G$ as $nvt(A) = 1 \leq \Phi \wedge \exists B \in M_G, votes(B, A) + u = 5 + 2 > \Phi$
- $B \in S_G$ as $nvt(B) > \Phi$
- $F \in S_G$ as $votes(A, F) + u = 1 + 2 \leq \Phi \wedge votes(B, F) + u = 4 \leq \Phi \wedge$ $votes(I, F) + u = 4 \leq \Phi \wedge votes(J, F) + u = 3 \leq \Phi$
- $I \notin S_G$ as $nvt(I) = 2 \leq \Phi \wedge \exists B \in M_G, votes(B, I) + u = 5 + 2 > \Phi$
- $J \notin S_G$ as $nvt(J) = 1 \leq \Phi \wedge \exists B \in M_G, votes(B, J) + u = 5 + 2 > \Phi$

Sp is stable and can be delivered:

1. $M_G \backslash S_G = \{A, I, J\}$
   $nvt(A) + u = 3 \leq \Phi \wedge \exists B \in S_G,\ votes(B, A) = 5 > \Phi \wedge$
   $nvt(I) + u = 4 \leq \Phi \wedge \exists B \in S_G,\ votes(B, I) = 5 \wedge$
   $nvt(J) + u = 3 \leq \Phi \wedge \exists B \in S_G,\ votes(B, J) = 5$
2. $ntail(G) = 10 \geq n - \Phi \wedge \exists B \in S_G,\ nvt(B) > \Phi$

Now suppose that H has not been inserted yet. The functions are modified as follows:

$ntail(G) = 9,\ u = 3,\ nvt(I) = 1,\ votes(I, A) = votes(I, B) = votes(I, F) = votes(I, J) = 1$
$S_G = \{B\}$ as

- $\forall i \in \{A, I, J\},\ i \notin S_G$ as $nvt(i) \leq \Phi \wedge \exists B \in M_G, votes(B, i) + u = 5 + 3 > \Phi$
- $B \in S_G$ as $nvt(B) > \Phi$
- $F \notin S_G$ as $nvt(F) \leq \Phi \wedge \exists B \in M_G, votes(B, F) + u = 2 + 3 > \Phi$

Furthermore, S_G is not internally stable, thus can cannot be delivered:

$$\exists F \in M_G \backslash S_G = \{A, F, I, J\},\ nvt(F) + u = 4 + 3 > \Phi$$

Even if the protocol has not reached a stable state, we can safely deliver B. The messages from sites $\{A, B\}$ are stable.

1. $A \in M_G \backslash S_G \wedge nvt(A) + u = 4 \leq \Phi \wedge \exists B \in M_G, votes(B, A) = 5 > \Phi$: A will not become a source message.
2. $B \in S_G \wedge nvt(B) > \Phi$: B will stay a source message.

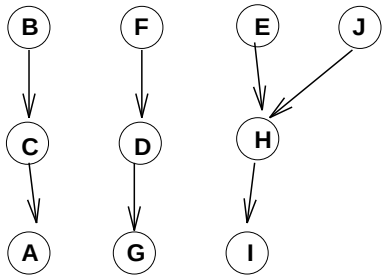


Figure 4: Hierarchical rule

Figure 4 illustrates the use of the H-rule. Suppose $n = 12, ntail(G) = 10, u = n - ntail(G) = 2, M_G = \{A, G, I\}$.

With a majority threshold of 6 ($\Phi = 6$), S_G is not stable: no message (seen or unseen) can get strictly more than 6 votes. Therefore, we need to wait for the two remaining messages to decide if they are part of the sources or not.

However with $\Phi = 2$, we have $S_G = M_G$, and S_G is stable ($R_G(\Phi)$ holds). The last two votes cannot change the outcome of the current election: to win, one needs 3 or more votes. Clearly, $\{A, G, I\}$ have won, and as there remains only two unseen messages, they will not be able to defeat any message, or be elected. With such a dag, if the H-rule is used with $\Phi_1 = 6, \Phi_2 = 2$, it is possible to deliver messages all messages in S_G.

5 Semantic Based Ordering Multicast

Virtual synchrony and the state-machine approach are two very practical tools to develop fault-tolerant applications. With a totally ordered multicast primitive, geographically distributed processes can execute a unique sequence of actions. This primitive, however, places strong constraints on the delivery of messages, and more concurrency is possible with weaker delivery orderings. These include reception order (no constraints), first-in first-out (messages are delivered in the order in which they were sent), and causal order.

Even though it is argued that these orderings are adequate for particular programs, their applications are rather limited. Indeed, the orderings arise from the communication pattern without considering the state of the data. However for most replicated data structures, correctness is maintained only if the schedule of operations applied to each copy is equivalent to one that would result in a system without concurrency. The equivalence relation is based on the commutativity of operations, usually for reads and writes [14].

To implement a replicated object, we propose a multicast primitive with delivery semantics based on the commutativity of operations, the object's state and the communication pattern. All sites agree on the construction of a linear order for operation processing, but relax this order to enhance response time. When Top is unable to deliver messages, they use additional heuristics, informations about the causality graph and the operations carried by the messages, to improve concurrency.

5.1 Commutativity Based Consistency

Mishra, Peterson, and Schlichting have proposed a semantic based ordering multicast primitive for Psync [64, 65, 66]. It optimizes delivery by using the commutativity property of read operations.

Their algorithm works as follows. Operations are sent to all sites using a reliable causal multicast

primitive. Each site arranges the operations it receives in a *causality* graph and executes them in waves. If a write operation is causally delivered, it is delayed. If a read operation is delivered and does not causally follow a delayed write, it is executed immediately. As soon as a write is fully acknowledged by all processors, the wave is complete and all operations that happened before can be executed.

As write operations delay reads and are executed *only when* they are fully acknowledged, this scheme does not much improve performance. To make the semantic ordering valuable, it should first be able to deliver operations faster than with a totally ordered multicast primitive. In other words, the level of synchronization should be lower than what is required with an early delivery scheme like the one presented in previous sections. This is the goal of our semantic based multicast primitive.

Weihl showed [90] also that concurrency can be increased for abstract data types when the notion of commutativity depends on the state of the data. Informally, two operations commute if they can be done in either order and produce the same result. We give below a more formal definition of our notion commutativity.

We model an object as an input-output automata, an automata that accepts only certain sequences of operations. An I/O automata is a four tuple $\mathcal{M} =< STATE, s_0, OP, \delta >$, where

- $STATE$ is the object's set of states (possibly infinite);
- $s_o \in STATE$ is its initial state;
- OP is a set of operations (the automaton's input alphabet);
- $\delta : STATE \times OP \rightarrow STATE$ is a partial transition function. If δ is defined for the pair (s, π), then we say that operation π is *legal* in state s.
- $\delta^* : STATE \times 2^{OP} \rightarrow STATE$ extends δ for finite operations sequences $\alpha = \pi_1 \pi_2 \ldots \pi_n$. $\delta^*(s, \alpha)$ is defined recursively as $\delta^*(s, \pi) = \delta(s, \pi)$ and $\delta^*(s, \pi\alpha) = \delta^*(\delta(s, \pi), \alpha)$.
- A finite sequence α of operations is legal or *accepted* by $\mathcal{M}$ if δ^* is defined for the pair (s_0, α).

We define the *language* of a machine $\mathcal{M}$, denoted $\mathcal{L}(\mathcal{M})$, as the set of finite sequences accepted by $\mathcal{M}$. $\mathcal{L}(\mathcal{M})$ describes a prefix-closed set of operations sequences: if $\alpha \in \mathcal{L}(\mathcal{M})$ then any prefix β of α is also in $\mathcal{L}(\mathcal{M})$.

Consider Weihl's example of a bank account object BA, with operations[11] to deposit and withdraw money and to retrieve the current balance. Assume that a withdraw has two possible results OK and NO. A state s of automata $\mathcal{M}$(BA) is a positive integer; the initial state is 0. The transition function of $\mathcal{M}$(BA) defines the behavior of the automata and can be described by specifying the triples (s, π, s') such that $\delta(s, \pi) = s'$, with precondition and effects for each operations π_i. If the preconditions of π_i are satisfied in state s, then π_i is legal in s. We follow the convention that an omitted precondition is true and an omitted effects indicates that $s' = s$.

π_1 = BA:[deposit(i), ok], $i > 0$
  Effects:
    $s' \leftarrow s + i$
π_2 = BA:[withdraw(i), ok], $i > 0$
  Precondition:
    $s \geq i$
  Effects:
    $s' \leftarrow s - i$
π_3 = BA:[withdraw(i), no], $i > 0$
  Precondition:

[11] An operation π on object X is both the invocation of an action and the result of that action. It is noted: "π = X:[operation(parameters),result]."

	BA:[depo.(j),ok]	BA:[with.(j),ok]	BA:[with.(j),no]	BA:[balance,j]
BA:[deposit(j),ok]			x	x
BA:[withdraw(j),ok]		x		x
BA:[withdraw(j),no]	x			
BA:[balance,j]	x	x		

x indicates that the operations for the given row and column do **not** commute forward.

Figure 5: Conflict relation for the bank account BA

$$\begin{array}{c} s < i \\ \pi_4 = \text{BA:[balance, i]} \\ \text{Precondition:} \\ s = i \end{array}$$

Let $\mathcal{M} =< STATE, s_0, OP, \delta >$. Two operation sequences α_1 and α_2 are *equivalent*, noted $\alpha_1 \equiv \alpha_2$, if and only if $\delta^*(s_0, \alpha_1) = \delta^*(s_0, \alpha_2)$. Recall that an operation sequence α is *legal* if δ^* is defined for the pair (s_0, δ).

Definition 17 *Forward commutativity. Two operations π_1 and π_2 commute forward if and only if for every sequence of operations α in which $\alpha\pi_1$ and $\alpha\pi_2$ are both legal, $\alpha\pi_1\pi_2$ and $\alpha\pi_2\pi_1$ are both legal and equivalent.*

The forward commutativity relation on operations of BA is given in figure 5. Deposits and successful withdrawals do not commute with balance operations, since the former change the state. The following lemma characterizes the set of schedules allowed when operations (in the formal sense) forward commute.

Lemma 5 *Let π_i, $1 \leq i \leq n$ be n operations, and α any sequence of operations for which $\alpha\pi_i$ is legal for all i. Let $i_1, i_2, \ldots, i_n$ be a permutation of $1, 2, \ldots, n$. If π_i commutes forward with π_j for all i and j, $1 \leq i < j \leq n$, then $\alpha\pi_1\pi_2 \ldots \pi_n$ is legal and $\alpha\pi_{i_1}\pi_{i_2} \ldots \pi_{i_n} \equiv \alpha\pi_1\pi_2 \ldots \pi_n$.*

Proof: in [90]. □

The idea with our semantic ordering protocol is to allow different schedules of operations as long as they are equivalent to an agreed linear sequence of operations. Equivalence is specified with respect to the forward commutativity relation.

Definition 18 *Let $\mathcal{M}_i$, $1 \leq i \leq m$ be the same automata $\mathcal{M}$ replicated at m sites. Let π_i, $1 \leq i \leq n$ be n operations and let $j_1, j_2, \ldots, j_n$ be a permutation of $1, 2, \ldots, n$ such that $\alpha_j = \pi_{j_1}\pi_{j_2} \ldots \pi_{j_n}$ denotes the ordering of operations π_i at site j. We say that the system is* correct *if and only if there exist some permutation k_i of $1, 2 \ldots n$ such that $\alpha = \pi_{k_1}\pi_{k_2} \ldots \pi_{k_n} \in \mathcal{L}(\mathcal{M})$ and $\alpha \equiv \alpha_j$ for all j.*

5.2 Semantic Ordering Protocol

A totally ordered atomic broadcast can be used to implement a replicated object that is consistent with definition 18. With the early delivery protocol presented previously, each site receives messages in a causal order, sorts concurrent messages by giving a priority to the first messages acknowledged by a majority of processors, and then executes the corresponding operations in a unique order. Let $ord(m)$ be the ordinal position of message m in this total order.

Even if a message is not yet totally ordered ($ord(m)$ might be unknown if not enough messages have been received yet), we might be able to deliver it immediately if the operation $\pi(m)$ corresponding to message m commutes forward with all other unordered operations that correspond to undelivered messages concurrent to m, or that happened before m. Intuitively, this ordering is equivalent, with respect to definition 18, to the total order.

We present below a general protocol which deliver messages early by taking advantage of the forward commutativity of operations. We also give several heuristics to make it practical. Before exhibiting the protocol, we still need to clarify further the differences between messages and operations. A message contains an invocation of an operation on a particular object. An operation is an invocation *and* matching response. For each message, there might be multiple corresponding operations, depending on the order the message in the total order. We show below that given a finite set of messages, there is a finite set of operation sequences possible corresponding to these messages.

let $A = \{m_i =< inv_i, X_i >,\ 1 \leq i \leq n\}$ be any finite set of messages that correspond to the invocation of operations inv_i on objects X_i. Let P be the set of all permutations of $1, 2, \ldots, n$. For each $u \in P$, we might be able to define an operation sequence $\alpha_u(A) = \pi_1\pi_2 \ldots \pi_n$, where $\pi_i = X_{u(i)} : [inv_{u(i)}, res_i]$. $\alpha_u(A)$ is the operation sequence, if it exists, corresponding to the invocation of operations in the order u. Let $L(A) = \{\alpha_u(A),\ u \in P\}$. $L(A)$ is set of all operation sequences corresponding to the messages in A. We will abuse the notation to denote that an operation is in one the sequences of the language: $\pi \in L(A) \Leftrightarrow \exists \alpha \in L(A), \pi \in \alpha$.

Lemma 6 *Let A be a finite set of operation invocations. The language $L(A) \subset \mathcal{L}(\mathcal{M})$ corresponding to all possible orders of operation invocation is finite.*

Proof: For each $u \in P$, there exists at most one operation sequence α_u. Therefore $|L(A)| < |P|$. As $|A| = n$ is finite, $|P|$ is also finite. □

As the semantic ordering protocol tries to deliver messages only when the total ordering heuristics fail, we consider only the messages still present in the current wave; that is, we assume in the following paragraphs that G does *not* contain messages delivered in previous waves of the total ordering protocol.

Let $Concurrent(m) = \{n | n \in G \wedge n \not\rightarrow m \wedge m \not\rightarrow n\}$ be the set of *all* messages that are concurrent to m, and that have not been totally ordered yet by Top. $Concurrent(m)$ is defined on G and therefore includes not only the messages present at processor p, but also all the messages concurrent to m not yet received at p.

Let $Pred(m) = \{n | n \in G \wedge n \rightarrow m\}$ be the set of messages that causally precede m in G, but that have not been delivered in previous waves. Note that if m is in G_p, then $Pred(m)$ is finite as $\forall i \in Pred(m),\ i \in G_p$.

The semantic ordering protocol (Sop) works in coordination with the total ordering protocol (Top). Both of them construct a linear ordering for the operations carried by messages.

Semantic ordering protocol $\mathcal{GS}$
Let α be the sequence of operations totally ordered by Top.
let β be the sequence of operations ordered by Sop.
Notation: $\pi(m)$ is the operation corresponding to message m.
$\alpha = \emptyset$;
$\beta = \emptyset$;
forever {
    When $\mathcal{C}$-$deliver_p(m_{i,j})$, insert $m_{i,j}$ in G.
    while there exists $m \in G_p$ can be delivered by Top
    {
        $\alpha \leftarrow \alpha\pi(m)$

```
            if π(m) ∉ β, β ← βπ(m)
        }
        If no m ∈ G_p can be delivered by Top
        {
            for all m ∈ G_p
            {
                SemanticDelivery(m)
            }
        }
}

SemanticDelivery(m)
{
        Let A_m = Concurrent(m) ∪ Pred(m).
        If    1) A_m is finite,
        and   2) there exists only one π(m) in L(A_m) corresponding to m,
        and   3) all π_i in L(A_m) are legal in the current state,
        and   4) all π_i in L(A_m) commute forward with π(m),
        Then m can be delivered by Sop: β ← βπ(m).
}
```

Theorem 4 *The protocol $\mathcal{GS}$ is correct according to definition 18.*

Proof: The proof is essentially a consequence of Lemma 5. The four conditions above guarantee that if a message is delivered early, the ordering of the corresponding operations is equivalent to the ordering of the operations using the total order (Def 18).

By definition of A_m, m will be totally ordered with respect to the messages in A_m (any message that happen after m will be delivered by Top after m is delivered). If A_m is finite, the ordinal position of m is bounded by the set of already delivered messages and the size of A_m: if $|A_m| = n \Rightarrow ord(m) \leq |\alpha| + |A_m|$. Also as A_m is finite, $L(A_m)$ can be defined (Lemma 6). As all messages in A_m will eventually be ordered by Top, there exists an operation sequence $\eta = \alpha\pi_1\pi_2 \ldots \pi_n \in \mathcal{L}(\mathcal{M})$, where π_i is an operation corresponding to some message in A ($L(A_m) \neq \emptyset$). Condition 2 and 3 are just the conditions so that lemna 5 is applicable. Even if we do not know $\eta = \alpha\pi_1 \ldots \pi(m) \ldots \pi_n$ we can reorder η to construct $\beta = \alpha\pi(m) \ldots$ □

5.3 Practical Heuristics

This general semantic ordering algorithm is rather impractical. Each process has a local view G_p of the causality graph and therefore can construct the set A_m only when m is fully acknowledged by all processors. Also even if A_m is known, we need to enumerate all possible (formal) operations and make sure that Lemma 5 is applicable. We can however design two heuristics to make this algorithm very effective.

The first heuristic is based on the premise that if we can place an upper bound on $ord(m)$ for a particular message and show that m will be included soon in the total order, we might be able to deliver it if the corresponding operation commutes forward with all $\pi \in L(A_m)$, *even if there might some messages concurrent to m not yet present in G_p.*

In the Top protocol, if a source message $m \in S_G$ has more than Φ votes ($nvt(m) > \Phi$), it is a stable source message and it will be delivered in the current wave. If lo denotes the ordinal position of the last message delivered by Top and n to be the number of processors in the group, $ord(m) < lo + n$.

Consider the BA object for which withdraw operations cannot exceed two hundred dollars (daily amount). If the current state of the account contains more than a thousand dollars, up to five successive withdraw

operations can be executed. If a withdraw operation wo is not yet totally ordered and if we can infer that at most four messages can be included in the total order before wo, then we can safely deliver wo (all four conditions of protocol $\mathcal{GS}$ are satisfied).

π_2 = BA:[withdraw(i), ok], $200 > i > 0$
Precondition:
$s \geq i$
Effects:
$s' \leftarrow s - i$
π_3 = BA:[withdraw(i), no], $i > 0$
Precondition:
$s < i \vee i > 200$

The second heuristic uses the commutativity property of reads operation, like in Psync, but apply it to the early delivery scheme of Top. The set of sources can be partitioned into read and write operations. Reads are delivered first in any order, and writes are delivered after all reads in lexicographic order. The protocol is fairly simple: when a read operation becomes a stable source message, it is delivered. Writes are delayed until the wave is complete (the sources are stable) and are delivered in lexicographic order. This scheme differs from the one in Psync by the fact that messages are usually delivered before they are fully acknowledged: as this optimization is applied *within* each wave, messages can also be ordered by the early delivery rule.

We explore both strategies in the next sections.

5.4 Revised Semantic Ordering Protocol

The revised semantic ordering protocol (Sop) is just a particular case of the $\mathcal{GS}$ protocol. We assume that M_G is not empty and that S_G is not internally stable (otherwise the Top protocol applies). $\mathcal{GS}$ relax the total order by delivering messages when Top cannot be applied in such a way that the new ordering is equivalent to the one defined by Top. The two heuristics presented informally above correspond to two different total orders. Recall that when the sources are delivered by Top, they need to be sorted first deterministically. We consider two total orders, one based on the origin, the other based on the content of a message:

Lexical order. A first solution is to sort messages in the source set by process id. This lexical ordering relation is noted $\rho_L : (m_{i,k}, m_{j,l}) \in \rho_L \Leftrightarrow i < j$.

Read first order. If operations consist of read and write operations, another solution is to deliver the reads first in lexical order, and then the writes also in lexical order. This linear ordering relation is noted $\rho_P : (m_{i,k}, m_{j,l}) \in \rho_L \Leftrightarrow [m_{i,k}$ is a read operation and $m_{j,l}$ is a write operation$] \vee i < j$

Let $\mathcal{S}$-deliverable(m) be the occurrence of the event corresponding to the delivery of message m by Sop. m is delivered when we can infer that m will be delivered in the current wave and that it commutes forward with all the operations placed before m in the total order. If ρ_L is used by Top, m needs to commute forward with all n such that $(n, m) \in \rho_L$.

let $Upp(m)$ the following predicate:

$$Upp(m) \Leftrightarrow \quad [S_G = \emptyset \wedge M_G = \{m\} \wedge u \leq \Phi] \vee \tag{8}$$
$$[m \in S_G \wedge nvt(m) > \Phi] \tag{9}$$

$Upp(m)$ hold when an upper bound on the ordinal position of message m in the total order can be defined:

```
* If Top uses a lexical order ρ_L to deliver the source set
RevisedSemanticDelivery(m)
{
      If Upp(m)
            Let A_m = {i | i ∈ M_G ∧ (i, m) ∈ ρ_L ∧ i ∈ Future(S_G)}
                  (A_m is the set of messages in G_p that might come earlier
                  than m in the total order and conflict with m).
            Let B_m be any set of messages such that ∀m_i ∈ G, m_i ∉ G_p ∧ (m_i, m) ∈ ρ_L.
                  (B_m is the set of unseen messages that might come earlier
                  than m in the total order and conflict with m).
            If    1) there exists only one possible π(m) corresponding to m
            and   2) all π_i ∈ L(A_m ∪ B_m) commute forward with π(m)
            and   3) all π_i ∈ L(A_m ∪ B_m) are legal in the current state
            Then m can be delivered by Sop: β ← βπ(m)
}
```

Figure 6: Sop for abstract data types

```
* If Top delivers read operations first by using ρ_P
RevisedSemanticDelivery(m)
{
      If Upp(m)
            if m is a read operation
            Then m can be delivered by Sop: β ← βπ(m)
}
```

Figure 7: Sop for read and write operations

Lemma 7 *for all $m \in G_p$, if $Upp(m)$ is satisfied, then m will be delivered in the current wave of the Top protocol.*

Proof: The first condition correspond to the case where there is only one candidate message m with not enough votes to become a source. As no unseen message will be able to vote against it, m will become a source message when all votes are cast. The second case is trivial with the definition of a source in mind: to be a source, a message needs Φ votes. □

The protocols for the two heuristics are shown in Figure 6 and 7.

This protocol is much more practical: A_m is defined on the local causality graph G_p, it can be constructed in linear time with a complexity $O(n)$ when n is the number of processors in the current group view.

Theorem 5 *The Sop protocol presented in figure 6 and 7 are correct according to definition 18*

Proof: These protocols are special cases of the $\mathcal{GS}$ protocol. □

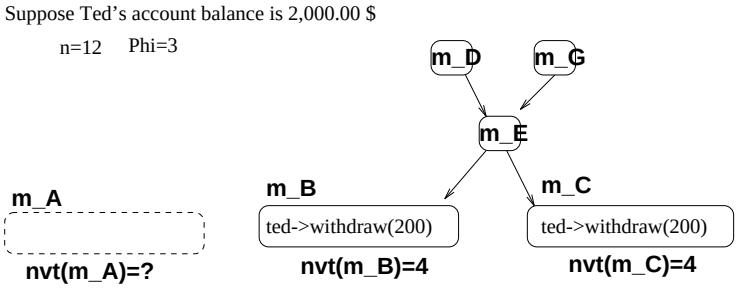


Figure 8: Messages from B and C get more than 3 votes, and not not conflict with an unseed message from A: there is enough money on the account

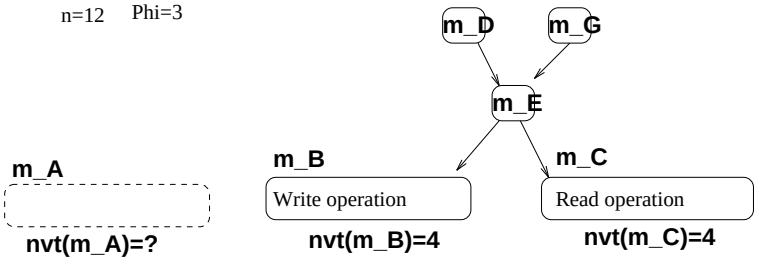


Figure 9: Messages G has just been received, and gives enough votes to C to be delivered immediately

5.5 Example

Consider the following example (Figure 8) with $n = 12$ processors, $\Phi = 3$. Suppose that messages are delivered in lexical order. The causality dag is depicted with 5 messages, two candidate messages m_B and m_C with 4 votes each (stable sources). Message m_A is not received yet. If m_A gets more than 3 votes, it will be placed in the total order before m_B and m_C. With the Sop protocol, it is possible to deliver m_B and m_C as these operations will not forward conflict with m_A even if m_A is a withdraw operation: there is enough money on the account. This is valid only with the modified bank account, where a limit is placed on the amount of a withdraw operation.

Now suppose (Figure 9) that messages consist of read and writes operations, and that read operations are delivered first. Suppose m_G has just been received, and votes for m_B and m_C. Again here, the set of sources is not stable, but as m_C is a stable source and commutes with all read operations that might come earlier in the total order (if m_A is a read operation), m_C can be delivered immediately. If m_A is a write operation, and becomes a source, it will be placed after m_C in the total order.

6 Performance

Our family of multicast protocols is geared towards reducing the average latency of message delivery. Two classes of message ordering are supported: linear and semantic order. To study the performance of our primitives, we opted for an approach based on discrete event simulation. Previous work on broadcast protocols relied on several different analytical models, making comparative evaluation difficult [63].

Simulation and experimental results with our implementation of the protocols are encouraging. We can not only increase concurrency, but also the protocols are more robust, as they can operate in a wide range of operating conditions.

6.1 Performance Goals

The objective of this study is to compare the performance of the protocols cited below. By performance, we mean the synchronization constraints placed on message delivery, and the latency of the resulting multicast primitives.

1. ToTo, the total ordering protocol of Transis;
2. G-Top, our generalized total ordering protocol;
3. LG-Top, the total ordering based on G-Top and lexical delivery;
4. H-Top, the linear ordering protocol based on hierarchical delivery rules;
5. Sop, the semantic ordering protocol.

We focus on several classes of operating conditions, by changing

1. Φ, the parameter of these protocols;
2. n, the size of the group;
3. T, the network topology.

6.2 System Model

Let S be a set of n sites interconnected by a communication network. Sites send and receive broadcast messages. A message sent by site i is delayed in the network, but is eventually received at all sites. Incoming messages at i are placed in a queue, and processed in sequence. A message is inserted in the causality graph when all its causal predecessors have been processed. The processing of each message delivers a set, possibly empty, of messages to the application. We characterize in the following sections the emission rate of messages for each site, the communication delays, and the processing time of each message.

6.3 Network topology

We choose to study three common types of network topology:

1. Star, or fully connected network: Any two sites are connected by a bi-directional link.
2. Ring network. The topology is a directed circular ring. To broadcast a message m, a site sends m to its right neighbor, who can retransmit it, and so on. All messages propagates in the same predefined direction. This network models local area networks like token-ring.
3. H-Lan (hierarchical network). This topology models wide-area networks, by connecting multiple Star networks. Each Star network is connected to a backbone by a repeater whose role is to retransmit the messages on the backbone. The backbone itself is composed of a sequence of bi-directional segments defined by the repeaters. Figure 10 depicts four star network connected to a backbone. A broadcast message sent by site a travels over one (respectively three) segment(s) of the backbone before it is received by the star network of site f (o).

Although the topology of networks could be made more accurate, these three types give us a qualitative idea of the behavior of the protocols under various conditions.

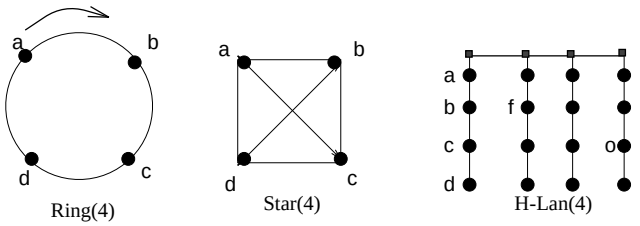


Figure 10: Network topologies

6.4 Workload

The workload of the system is characterized by the number of messages $N_i(t)$ sent by each i at time t in P. We assume that $N_i(t)$ is a continuous time renewal (Poisson) process with the following distribution:

$$P\{N_i(t) = n\} = e^{-\lambda t}\frac{(\lambda t)^n}{n!}$$

We assume that the all $N_i(t)$ are independent and identically distributed. This means that all sites send messages at rate λ, independently of each other. Let $\tau = 1/\lambda$ be the inter-emission time of messages (numerical values in milliseconds).

6.5 Communication delays

Communication delays are hard to model accurately if messages can get lost. The communication subsystem automatically recovers messages, but even if we know the frequency of omission failures, it is difficult to derive the resulting delay for causal messages: it might take several rounds of messages before one detects the loss of a message, and this message will be inserted in the causal order only when all its predecessors are also recovered. To solve this problem without assuming a constant propagation delay, we choose a uniform distribution for the sequence emission - recovery - reception.

Communication delays depend on the topology T of the network, and is uniformly distributed from 0 to $d_T(i,j)$. $d_T(i,j)$ corresponds roughly to the number of edges in the topology graph T from i to j. Note that it is possible to receive messages in a different order at different sites. The probability density function corresponding to communication delays between i and j is given by:

$$f(x) = \begin{cases} \frac{1}{d_T(i,j)} & \text{if } 0 < x < d_T(i,j) \\ 0 & \text{otherwise} \end{cases}$$

1. In a Star network $d_{STAR}(i,j)$ is constant. $d_{STAR}(i,j) = dy$.
2. In a directed circular ring $d_{RING}(i,j)$ depends on the relative position of i and j. Let $k = n + j - i$ if $j < i$ and $j - i$ otherwise. k represent the number of links that a message needs to travel on in the Ring network to go from i to j. Let ed the extra delay incurred for each of these links. We model the circular ring with $d_{RING}(i,j) = dy + ed * k$.
3. In a H-lan, or hierarchical Star network, the nodes are divided into H components connected to a backbone. The delay cost within a group is constant dy, but inter-group delays depend on their relative distance. Let $k = j\%H, l = i\%H$. The number of backbone segments that a message need to travel on to go from the Star network of i to the Star network of j is given by $abs(k - l)$. Let ed be the extra delay incurred by each backbone segment. The H-lan is modeled by $d_{H-LAN}(i,j) = dy + (l - k) * ed$ if $k < l$ and $dy + (k - l) * ed$ otherwise.

6.6 Service time

The processing time of an incoming message is given by an Erlang distribution with mean td and variance sd. This distribution is commonly used in queuing models as an extension of the exponential distribution. Its probability density function is given by:

$$f(x) = \frac{x^{m-1}e^{-x/a}}{(m-1)!a^m}$$

where $td = am$ and $sd = a^2m$.

6.7 Metrics

We measure the synchronization constraints placed on message delivery as being the number of processors one need to hear from before delivering a message. More precisely, when a message is delivered at processor i, the following parameters of the causality graph G_i and m are recorded:

- ntail(m): the number of processors that have a message causally following m in G_i.
- votes(m): the number of votes gathered by m.
- ntail(G): the number of votes in G_i.

Intuitively, the average values of these functions estimate the performance of the voting scheme. We refer to $ntail(G)$ as the index of synchrony.

The second performance metric is the average latency of message delivery. Latency(m) is defined as being the delay between the submission of m_i at processor i by the application layer and its delivery, also at i. This measure directly depends on the communication delays, processing costs and total ordering delays.

For the LG-Top protocol, we record also the number of messages delivered by:

- Lexicographic order rule (L-Delivery())
- Early delivery rule ($R(\Phi)$)
- Default delivery rule ($ntail(G) = n$)

6.8 Simulation

Performance evaluation was conducted using `simpack++`, a discrete event simulation package by P. Fishwick [43] . We introduced two system parameters, n, the number of sites in P and nm the number of messages for each simulation. We defined some standard numerical values for the simulations. Unless specified otherwise, we used these values in the simulations.

n = 20processors;
nm = 5000 messages;
τ = 5.0 milliseconds-seconds (ms);
dy = 0.6 ms;
td = 0.2 ms;
sd = 0.1 ms;
ed non applicable for Star

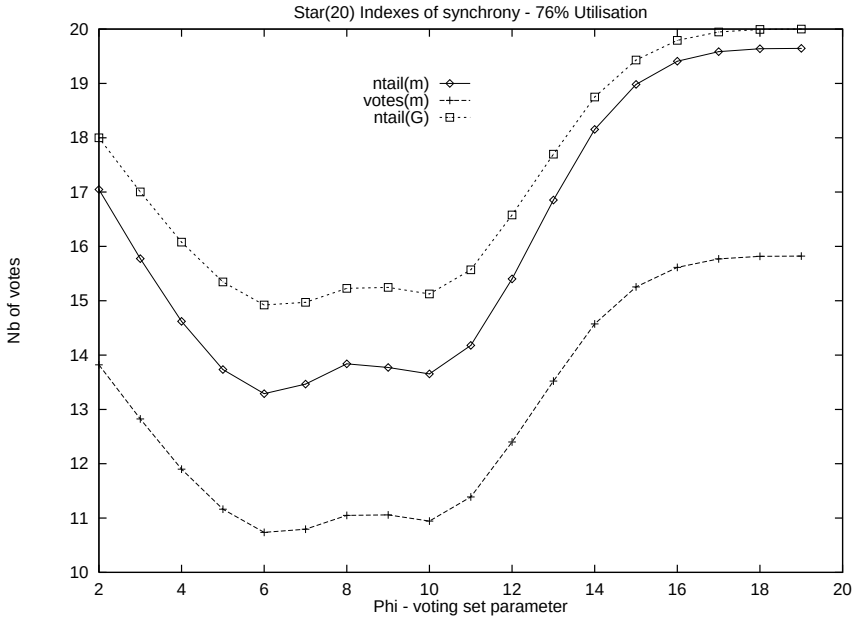


Figure 11: Indexes of synchrony

ed = `0.2 ms; (Ring)`
ed = `1.0 ms; (H-Lan)`
H = `4 groups of processors;` (H-Lan)

These values were validated by the simulations, by computing the utilization of the system and by studying the behavior of the protocols.

6.9 Results

The first two figures present the general behavior of the protocols. The LG-Top protocol for a Star network is depicted in Figure 11: the various indexes of synchrony are functions of Φ, with standard numerical values. Under such conditions, the utilization of the system, as computed by `simpack++`, is 76%. The optimal Φ for this configuration lies between 6 and 10. In the following figures, we only refer to $ntail(G)$ when we compare synchrony (it represents the average number of processors that have a message in G when a message gets delivered).

When a message is delivered by LG-Top, it is delivered either by the lexical, early or default rule. Figure 12 shows their respective distribution. i.e for each Φ, the sum of the messages delivered by the three rules is exactly nm. With this configuration, most messages are delivered with the early delivery rule when $\Phi \leq \frac{2n}{3}$

Figure 13 shows the behavior of LG-Top for various types of network. The observed latency is proportional to the index of synchrony, and the optimal value for Φ seems to be around $n/3, n/4, n/5$ respectively for a Star, H-Lan, and Ring networks.

The parameter ed is the main variable for the Star and Ring network: it controls the communication delays between two sites. We looked first at its impact on synchrony and latency in figure 14 and 15. The results are consistent in the sense that the best Φ remains around $n/3$ when ed increases. A higher ed for a Star network means that messages take longer to go from a site to the next. A higher ed for a H-Lan network

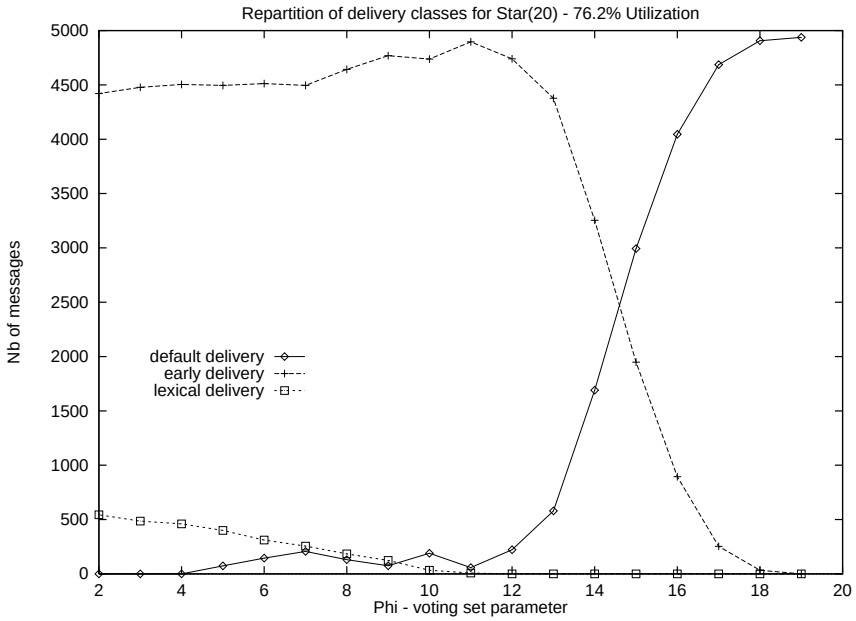
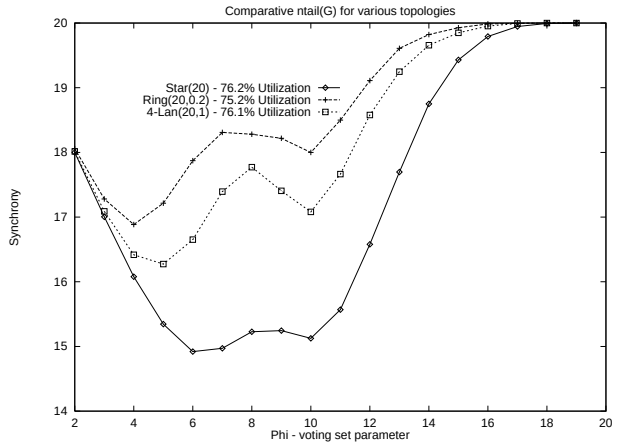
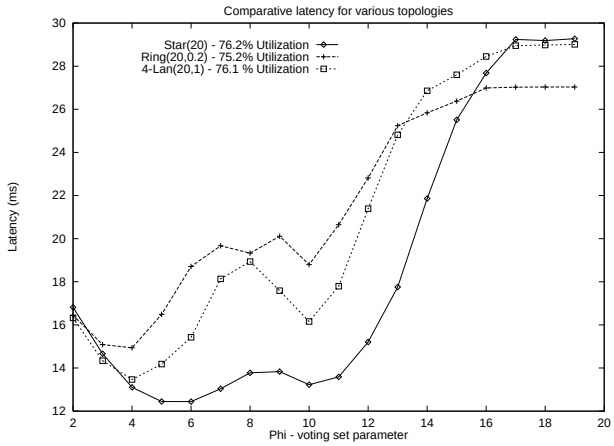


Figure 12: LG-Top delivery distribution

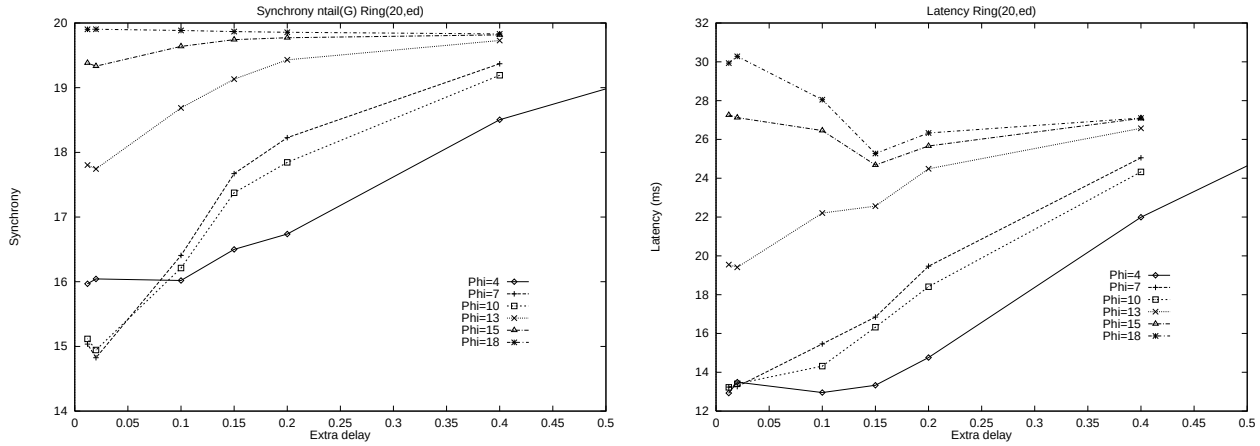


(a) Synchrony



(b) Latency

Figure 13: Comparative synchrony for the three types of networks



(a) Synchrony

(b) Latency

Figure 14: Impact of the communication delay on a Ring network (LG-Top)

means that it takes longer for a message to go from a local network to the next. The best performance is obtained for $\Phi = n/5 = 4$.

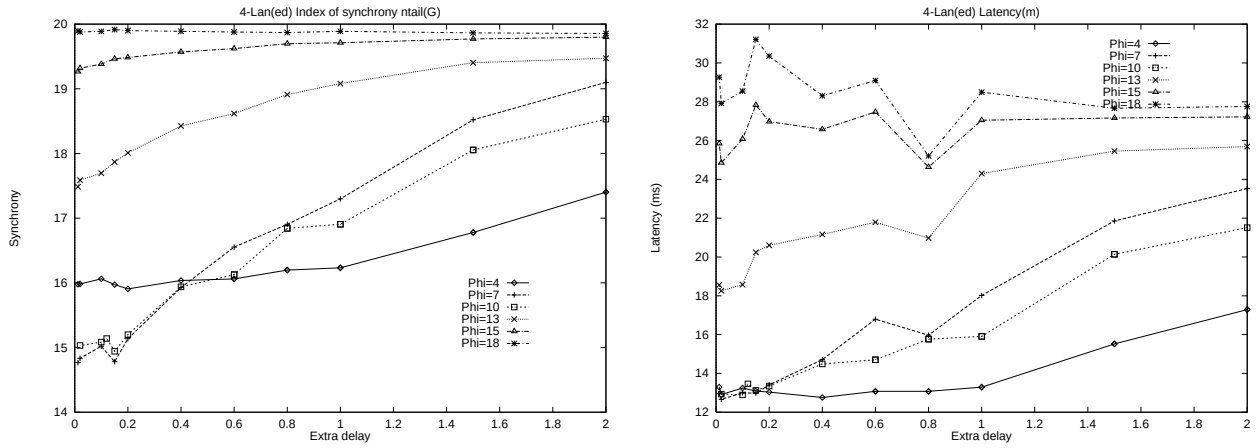
For each topology we then compared the behavior of the ToTo protocol, our early delivery protocol (G-Top), G-Top combined with lexical delivery (LG-Top), the hierarchical rule (H-rule) and the semantic delivery (Sop). We note in figures 17 and 18 that the Ring and H-lan networks emphasize the performance differences between ToTo and our family of protocol. A gain of 20% in concurrency is easily obtainable.

In the Sop protocol, we assumed that each message corresponds to an operation, and that $X = 100\%$ represents the percentage of operations in the sources that forward commute. This assumption gives us an estimate of the concurrency possible with the semantic ordering protocol. As Sop reduces to Top when $X = 0\%$, and because of the strategy used by Sop, we can expect that with $0 < X < 100\%$, a linear interpolation between Top and Sop with $X = 100\%$ will give a good approximation of the level of concurrency and latency achievable.

As a result, Sop performs better than any other scheme. Even if a candidate message can be delivered as soon as its gets more than Φ votes, in average we observe in figure 16 that messages are delivered only when $ntail(G) \geq 14$ for a Star network. This is much higher then $\Phi = 6$, and suggests that messages are delayed *between* waves of the protocol: messages in G gather votes even before becoming candidates. When they become candidates, they are immediately delivered, but have already much more than Φ votes.

The H-Top protocol requires a sequence of delivery thresholds. We chose the sequence $\Phi, \Phi/2$. If $R(\Phi)$ cannot be satisfied, H-Top tries to deliver messages using $R(\Phi/2)$. This hierarchical rule performs better than the other protocols for high values of Φ. It corresponds to the intuition that if one needs almost all the votes to win an election, it becomes easier to detect a situation where no one is elected. By definition, H-Top performs better than G-Top. However, LG-Top seems to be superior to H-Top for most values of Φ.

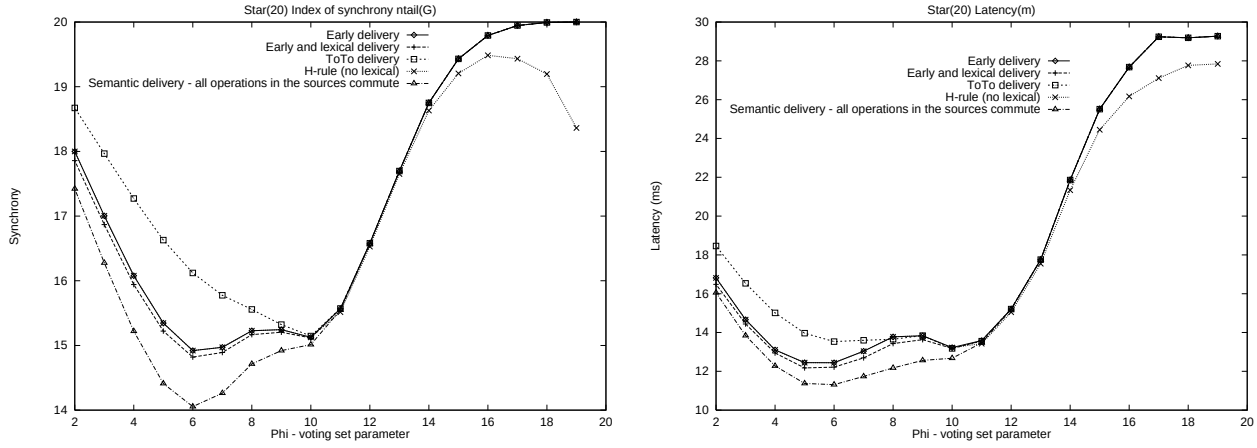
The next figures (19, 20 and 21) show the synchrony and latency of LG-Top for 5, 10, 15 and 20 processors. Again, we present three sets of figures for the three types of network. They confirm the fact that a low Φ offers best latency and show also that if Φ is carefully chosen, it takes about the same amount of time to place a linear order on messages for 5 or 20 processors (between 13 and 15 milliseconds on a Ring network). For the ring or H-lan network, LG-Top is more scalable than ToTo: as the number of sites increases, the best latency increases faster for ToTo than for LG-Top.



(a) Synchrony

(b) Latency

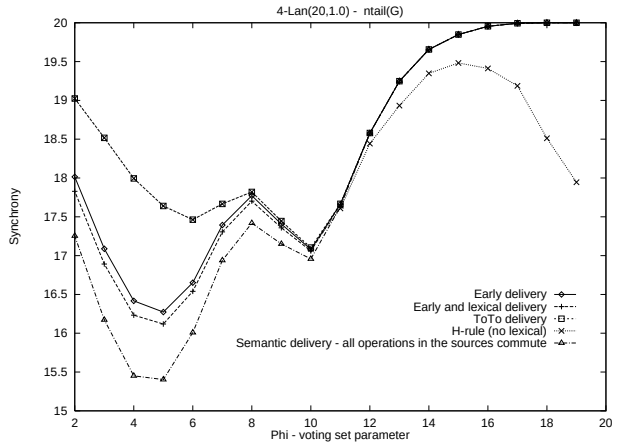
Figure 15: Impact of the communication delay on a H-Lan network (LG-Top)



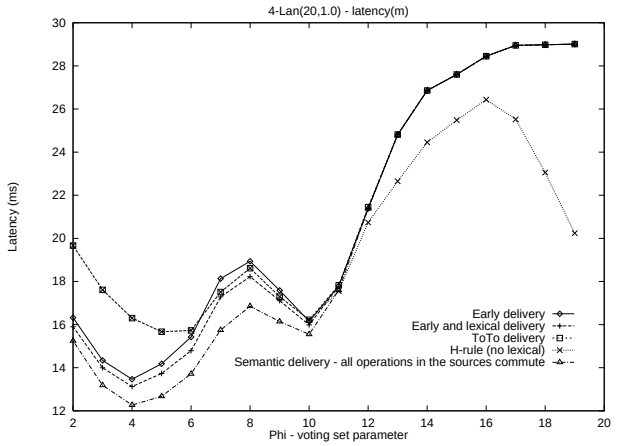
(a) Synchrony

(b) Latency

Figure 16: Evaluation of the protocols - Star topology

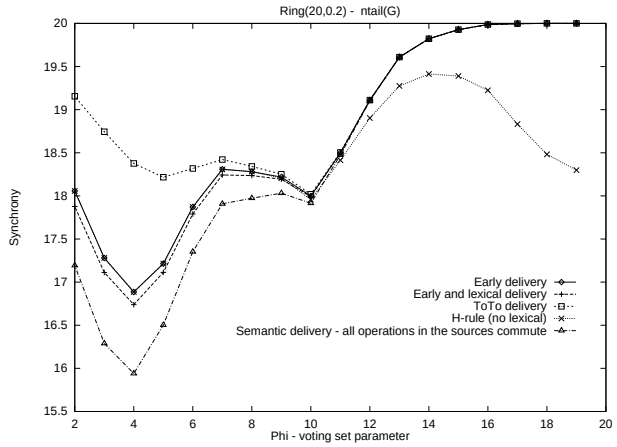


(a) Synchrony

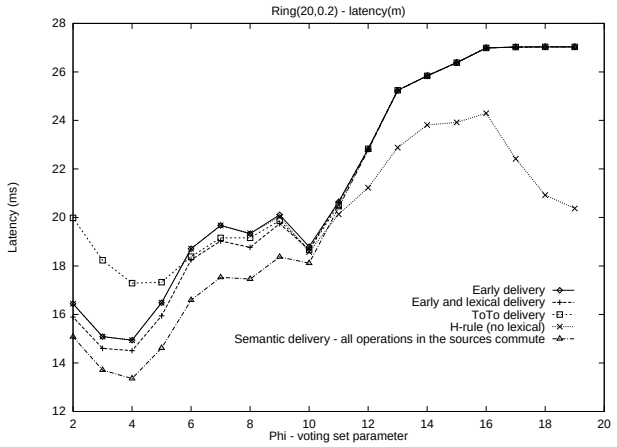


(b) Latency

Figure 17: Evaluation of the protocols - H-lan topology

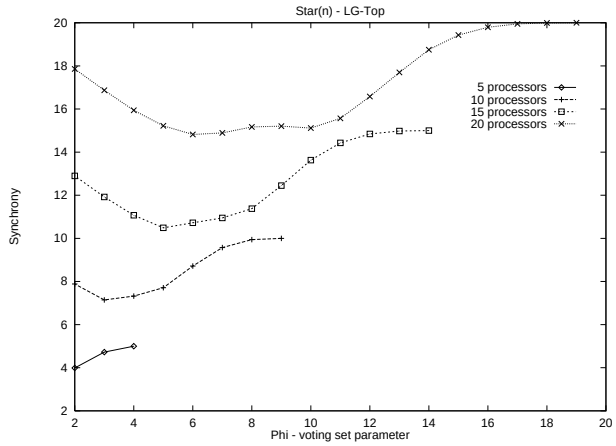


(a) Synchrony

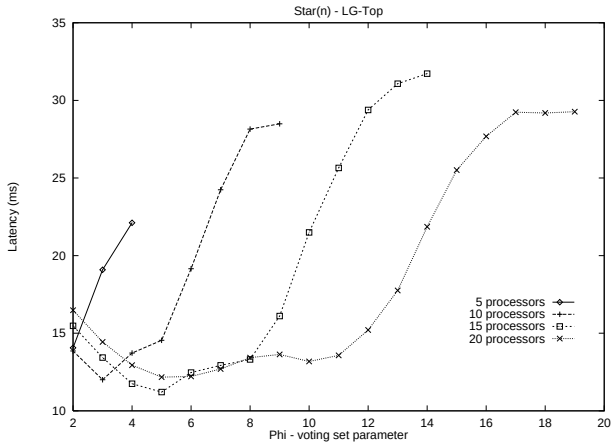


(b) Latency

Figure 18: Evaluation of the protocols - Ring topology

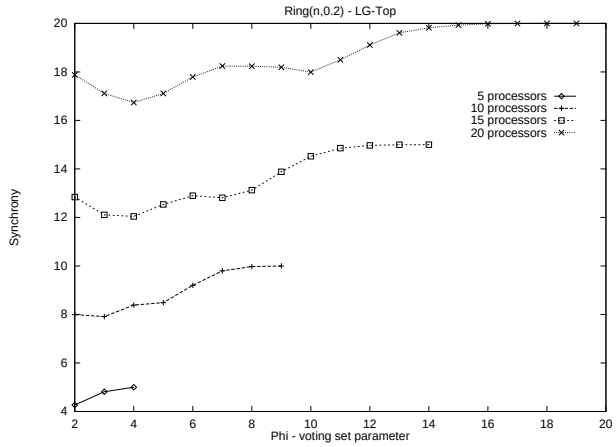


(a) Synchrony

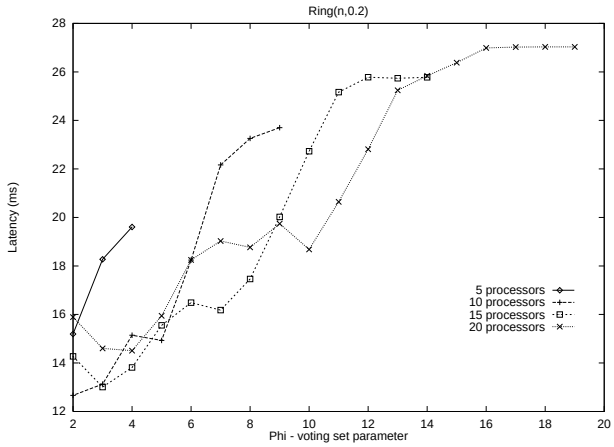


(b) Latency

Figure 19: Star - LG-Top for various group size

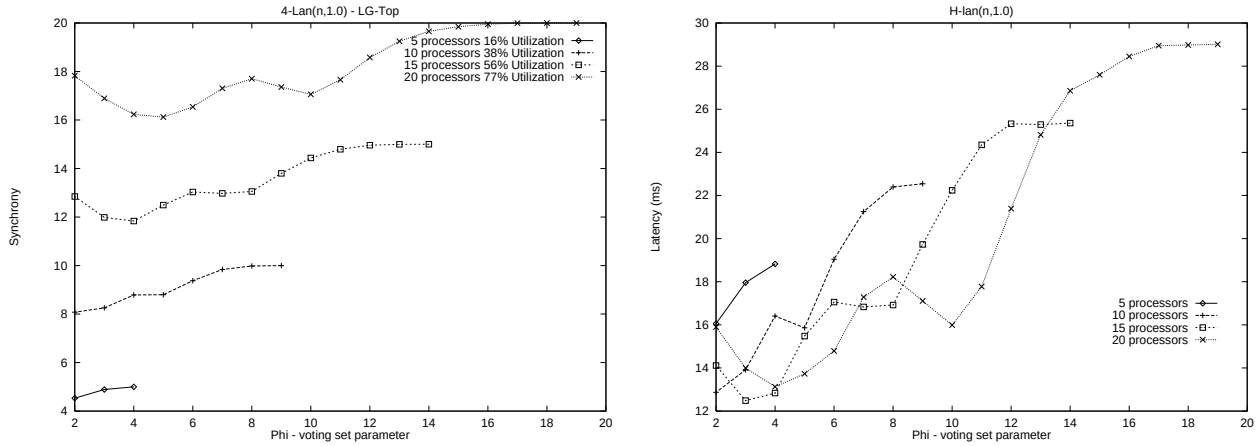


(a) Synchrony



(b) Latency

Figure 20: Ring - LG-Top for various group size



(a) Synchrony

(b) Latency

Figure 21: H-lan - LG-Top for various group size

Note: The workload used in the simulations is not geared towards presenting the best behavior of the new protocols. In a Star network, uniform request rates favor a majority threshold of $\Phi = n/2$ with GL-Top; that is, if delays and request rates are uniformly distributed, messages need half of the votes to be delivered efficiently. However real communication is more likely to be bursty, with non-uniform propagation delays, as in the ring or hierarchical networks. In these cases, our voting scheme improves performance with small majority thresholds $\Phi < n/2$; that is, if few processors start sending more messages, or if the network topology incurs various delays between sites, our heuristics allow the delivery of messages without necessarily waiting for the majority ($n/2$) to catch up.

To summarize, our family of protocols offers increased concurrency and better response time than previous work. Its superiority becomes even more noticeable under different types of communication networks, or disparate communication patterns. Remarkably, the cost of the protocols does not increase much with the number of sites. To build a total order LG-Top performs best, provided that low values of Φ are chosen. For high Φ, the H-rule is the method of choice.

To manage atomic data, improvements are possible by using a multicast primitive with delivery ordering semantics based on the commutativity of operations. It is however a difficult task: agreeing on a consistent semantic order still necessitates a rather high level of synchronization. We believe that the Sop protocol is close to the maximum level of concurrency obtainable under the current framework. Nevertheless, this semantic ordering protocol is a careful attempt in the search of a compromise between concurrency and consistency for abstract data types.

7 Approval Voting

The total ordering protocol (Top) uses a special voting system to deliver messages early. Processors "elect" some messages to be inserted in the total order, by giving a priority to the first messages they acknowledge. Messages are delivered "early" when a decision about the outcome of the election can be made before all votes are gathered (in a two candidates election, a candidate is declared winner as soon as it gets strictly more than half of the votes).

We discuss below various alternative voting strategies and study the properties of our voting scheme,

called approval voting. We also review fundamentals results of the theory of voting. Indeed, an impressive body of mathematical results have been gathered in the past few decades in the theory of social choice, and are important for the design of decision making procedures [86, 18]. Voting has many applications for computer science: it can be used for the allocation of resources, or like in distributed systems, to ensure that only one group of processes is allowed to perform some restricted operations. The choice of particular voting rules raises in this context, as in democracy, essential ethical questions: are these rules fair ? coherent ? are they vulnerable to manipulations ?

Early analytical work on the theory of voting, by the Marquis de Condorcet and Jean-Charles de Borda in the late eighteenth century, quickly revealed that seemingly straightforward voting methods could hide surprising logical subtleties[12]. We present the main issues in the following sections, and discuss their applicability to computer science.

7.1 Topics In Voting Theory

Formally, voting is a collective decision process where several individual agents must jointly choose one among several outcomes (henceafter called candidates), about which their opinion conflict. A finite set of N voters must pick a subset of candidates (ideally a singleton) within a finite set of A. Denote $L(A)$ to be the set of linear orders of A. A voter's opinion is an element of $L(A)$, that is, an ordering of A (indifferences are excluded). A voting rule is a (single valued) mapping from $L(A)^N$ into A. Elections that allow multiple candidates to be elected are defined with voting correspondences: (multi-valued) mappings from $L(A)^N$ into 2^A.

When only two candidates are on stage, ordinary majority voting is the unambiguously fair method of choice. This majority principle is the benchmark of democratic decision, and May proved in 1952 that majority voting is the only method that is anonymous (equal treatment of all voters), neutral (equal treatment of all candidates), monotonic (more support for a candidate does not jeopardize its election) and strategyproof (one cannot profit from misreporting his preferences).

7.2 Plurality voting and its critiques

When more than two candidates are at stakes, plurality voting comes first to mind. It is also by far the most popular voting rule.

> *Plurality rule:* Ask each voter to name his top candidate. Elect this candidate whose named most often.

Condorcet and Borda first noticed that plurality voting may elect a poor candidate, namely, one that may contradict the majority opinion. Consider a situation with 21 voters, 4 candidates, and the following preference profile:

This table (Figure 22) reads: 3 voters have the preference ordering $a > b > c > d$, 5 voters $a > c > b > d$, and so on. According to plurality, a wins with 8 votes, but says Condorcet, a is actually the *worst* candidate for a clear majority of voters (13). This majority prefers any candidate to a. Furthermore, another majority (of 14 voters) prefers c to d, Thus, argues Condorcet, c should actually be elected if we abide by the majority opinion. He suggested the following criteria:

[12] Lewis Carroll even wrote that *"the principles of voting makes an election more a game of skill than a real test of the wishes of the electors. My opinion is that it is better for elections to be decided according to the wish of the majority than of those who happen to have most skill at the game."*

voters	3	5	7	6
top	a	a	b	c
	b	c	d	b
	c	b	c	d
bottom	d	d	a	a

Figure 22: Preference profile

A Condorcet winner is a candidate that defeats every other candidate in majority comparisons. For all $b \neq a$, more voters prefer a to b than b to a. A voting rule is Condorcet consistent if it elects the Condorcet winner when it exists.

Condorcet consistency is widely regarded as a compelling democratic principle. In most cases however, a Condorcet winner does not exist, as majority comparisons yield a cycle. In such situations, alternative voting rules have to be chosen. For example, consider the preference profile:

voters	8	7	6
top	a	b	c
	b	c	a
bottom	c	a	b

Here 14 voters (of 21) prefer a to b, 15 prefer b to c and 13 prefer c to a. From a mathematical standpoint, the frequency of such paradox can be estimated. It increases with the number of voters and candidates: for 3 voters and 3 candidates, there is a 5% chance that a Condorcet winner does *not* exist. For 7 voters and 7 candidates, it is already 30%.

Discussing the same profile (Figure 22), Borda agrees with Condorcet that a would be a poor candidate to elect, but he proposes a different winner - namely, b. He assigns points to candidates, linearly increasing with his ranking in a voter's opinion. This idea of point scoring according to ranks is generalized as:

Scoring voting rule: fix a nondecreasing sequence of real numbers $s_0 \leq s_1 \leq \ldots \leq s_{p-1}$ with $s_0 < s_{p-1}$. Voters rank the p candidates, thus giving s_0 points to the one ranked last, s_1 to the one ranked net to last, and so on. A candidate with maximal total score is elected.

Plurality voting is an example of scoring method: take $s_0 = s_1 = \ldots s_{p-2} < s_{p-1}$. Borda's method is defined by taking $s_i = i$. Scoring methods are in disagreement with Condorcet consistency as they may fail to elect the candidate approved by the majority:

Theorem 6 *(Fishburn 1973) There are profiles where the Condorcet winner is never elected by* **any** *scoring method.*

Proof:

voters	6	3	4	4
s_2	a	c	b	b
s_1	b	a	a	c
s_0	c	b	c	a

score of $a = 6s_2 + 7s_1$
score of $b = 8s_2 + 6s_1$
score of $b >$ score of a

Here a is the Condorcet winner. Yet b wins in any scoring method. To see this assume, without loss of generality that $s_0 = 0$ so that $0 \leq s_1 \leq s_2$ and $s_2 > 0$. Compare now the scores of a and b: b wins for all s_1, s_2 (above).

7.3 Normative properties of voting systems

To evaluate scoring rules and Condorcet consistent methods, several normative properties have been proposed:

Pareto optimality. If a candidate a is unanimously preferred to candidate b, then b should not be elected.

Anonymity. The name of the voter does not matter: if two voters exchange their votes, the outcome of the election is not affected.

Neutrality. The name of the candidate does not matter.

Monoticity. A voting rule is monotonic if a candidate remains elected when his support increases (i.e, when the relative position of this candidate improves in somebody's preferences while the relative position of the other candidates is unaffected).

Reinforcement. Two disjoint groups of voters N_1, N_2 face the same set A of candidates. Say the electorate N_i select subset B_i of A for each $i = 1, 2$. If B_1 and B_2 intersect, then electorate $N_1 \cup N_2$ should elect $B_1 \cup B_2$ as the set of equally best outcomes.

Continuity. Say that the electorate N_1 elects candidate a from A whereas a disjoint electorate N_2 elects a different candidate b. Then we can replicate the electorate N_1 sufficiently many times -say, m - so that the combined electorate $(mN_1) \cup N_2$ chooses a.

Strategyproofness. A voting rule is strategyproof if each individual voter has an incentive to report his opinion truthfully, namely if he does not benefit from misreporting his preferences.

The Borda rule is pareto optimal, anonymous and neutral. As any scoring scoring method, it is also monotic. As we have seen, it is not Condorcet consistent. Plurality voting is pareto optimal, monotic, anonymous, neutral, but is not Condorcet consistent. Indeed, it may even elect a Condorcet looser, namely a candidate that loose against any other candidate in pairwise contests (Figure 22). Another method, by Duncan Black, combines the virtues of Condorcet and scoring methods: it elects the Condorcet winner when it exists, otherwise apply the Borda count. This voting rule is pareto optimal, monotic, neutral, anonymous and Condorcet consistent.

Young proved the following results that give Borda's supporters strong arguments against Condorcet Consistency.

Theorem 7 *(Young 1975) There is no Condorcet voting correspondence satisfying reinforcement.*

Theorem 8 *(Young 1975) A voting correspondence is a scoring method if and only if it satisfies anonymity, neutrality, reinforcement and continuity.*

This result says that reinforcement is essentially enough to characterize the scoring methods.

Despite these problems, binary comparisons and Condorcet consistent voting rules are quite popular. One familiar method, by successive elimination, is used in the U. S Congress to vote upon a motion and its proposed amendments. Let a be an amendment, b the amendment to the amendment, c the original motion, and d the status quo. (Figure 23).

> Voting by successive elimination: first a majority vote decides to eliminate a or b, then a majority vote is called to eliminate the survivor from the first round or c, and so on.

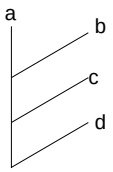


Figure 23: Sequential majority comparisons: order abcd

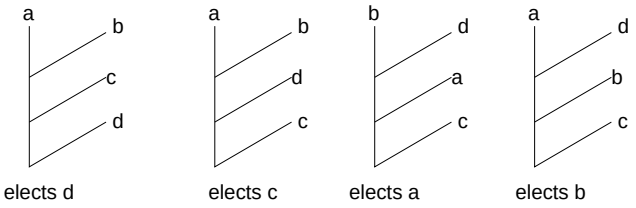


Figure 24: Amendment procedure

Even if this method elects the Condorcet candidate when it exists, it is inherently not neutral. The order of elimination obviously matters to the outcome. To see this, consider the following profile:

voters	1	2	1	1
top	d	a	d	b
	b	b	c	c
	a	c	a	d
bottom	c	d	b	a

With such profile (Figure 24), a beats b and c, b beats c and d, c beats d, and d beats a. Thus, by setting the agenda, the chairman can control the outcome of the election (above). In addition to the lack of neutrality, the successive elimination method may violate Pareto optimality. Consider the following profile:

voters	1	1	1
top	b	a	c
	a	d	b
	d	c	a
bottom	c	b	d

and the successive elimination ordering $abcd$. Then b beats a, next c beats b, next d beats c and so d is finally elected. Yet a is unanimously preferred to d!

Another concern with elections is their subjectivity to manipulations. Under plurality voting, it is sometimes rational to cast one's vote for another candidate than one's first-best choice: "if I know that my most preferred candidate a will not pass because b and c are both going to get more votes, then I had better help whomever of b, c I like more." Maybe the naive ballot suggested by the voter's true preferences does not serve his interest best. Rather than passively reporting his opinion about candidates, he should acts as a player in the game of election, trying the returns from his votes. To avoid the nuisance of dishonest votes, several analyst tried to design a voting method that avoids such strategic aspects, namely that is *strategyproof*. A dictatorial rule leaves all the decision power in the hands of a single voter. Surely, we dismiss such a rule as it is the most unfair possible. Yet for every nondictatorial voting rule, there exists some preference profile at which some agent benefits by not reporting truthfully his preferences:

Theorem 9 *(Gibbard-Satterthwaite 1975). If A contains at least three candidates, a voting rule S is strategyproof if and only if it is dictatorial.*

A related result by Kenneth Arrow is that it is impossible for a voting method to satisfy m reasonable criteria, like the properties cited earlier. This impossibility result can be relaxed for particular domain of preferences. For example, if the preference profile varies in a domain where the voting paradox does not exist, then electing the Condorcet winner with a majority voting is strategyproof.

This led us to our voting method, approval voting, for which the preference profiles are restricted to a particular domain.

7.4 Properties Of Approval Voting

Like plurality voting, approval voting [18] is a non-ranked voting system. A voter is not required to list the candidates his order of preference, but instead he lists the alternatives he "approves of." He can however, cast more than one vote:

> Approval voting: each voter votes for, or approves of, as many candidates as he wishes. He can give only one vote for each candidate. The candidate with the most approval votes wins.

Approval voting is very easy to use, and has the advantage of usually choosing the alternative which is acceptable to many voters. This method is also Pareto optimal, neutral, anonymous and monotic. It may be subject to manipulations if, for example, a voter prefers a tie of two candidates a, b to a victory of a. However, Brams and Fishburns have shown that approval voting is significantly less subject to strategic manipulations than any other non-ranked voting systems. If voters have dichotomous preferences (if candidates are either approved or disapproved), they have showed that

1. Approval voting discourages insincere voting; it is *never* advantageous for a voter to vote for candidates he disapproves of without voting for the candidates he approves of;
2. it is the only non-ranked voting system that is Condorcet consistent;
3. in a precise sense, approval voting is the *only* strategyproof method; there is *never* an incentive to vote for those not preferred.

Approval voting was introduced by Brams and Fishburn in 1977. Since then, it has received a lot of attention, and is currently being considered for use in the presidential elections.

7.5 Discussion

In the light of these different voting schemes, we choose approval voting for its adequacy to the problem of ordering messages linearly, given causal dependencies. A processor votes possibly for more than one message, the ones he acknowledges first. He does not need to list all candidates in some particular order.

In this context, we were concerned with determining the outcome of the election "early", before all votes are cast, and approval voting seemed to be good method of choice. On the other hand, we were not so concerned with electing a small number of candidates, but rather we tried to choose a maximum number of candidates possible as soon as more than half of the votes are cast. This requirement was implicit in our definition of *source* messages.

Monoticity and reinforcement were also major advocates for the use of approval voting. These two properties unsure that in some cases, it is possible to infer the outcome of the election early. With causally ordered messages, we do not know in advance all candidates, as they might correspond to unreceived messages. As the candidates are the roots of the causality dag, the maximal number of candidates if the number

of processors in the group. Under these assumptions, even if all votes are not known, it is sometimes possible to decide early on a set of elected candidates.

To break ties between elected candidates, we introduced a non-neutral rule: for the total ordering, messages are delivered using a fixed (lexical) ordering of the messages. We also pointed out another non-anonymous rule for semantic ordering: messages are classified first into read and write operations. We acknowledged the fact that this method is Pareto optimal and Condorcet consistent. It is nice to learn that our voting scheme is fair.

From a more general perspective, it is clear that the issue of fairness and strategic reasoning in voting has been studied carefully by political scientists[13]. This work is also important for computer science, especially in distributed systems. Consensus is required for numerous problems including atomic commitment, election, linear ordering of messages, or mutual exclusion. Indeed, the main problems in distributed computing involve the design of decision making procedures that allow sites to cooperate to solve a common problem. Fairness is a basic requirement for such decision procedures, and the study of the properties of voting techniques are important in that respect.

This (small) digression is based on the premise that not just one, but several approaches are possible to solve problems in distributed computing. Current formalisms promote reasoning about events (Lamport [56, 21]), states (Chandy and Misra [22]), or knowledge (Halpern [70]). However, we have shown that recent results in social choice theory (Moulin [72, 71]) can also prove to be useful, as they provide a better understanding about voting procedures. Similarly, advances in game theory (Conway [27]) might also be adequate for distributed applications where decisions are made in a non-cooperative (competitive) manner, for the allocation of resources for example. We left these issues as future research.

8 Conclusion

Technological progress in computing systems generates new challenging demands. Where it was sufficient to store, query and process data efficiently, it is now necessary to provide such services without interruption, 24 hours a day, 7 days a week (7x24). The speed of modern processors, the size of memory chips and even the capacity of disk arrays are increasing at a much faster pace than the load of most transaction processing systems[14]. As a result, future platforms will easily sustain the workload of most database applications, and popular systems will be required to offer non-stop (7x24) operations, that is, to be available and fault-tolerant.

Services can be made fault-tolerant by using replication. One approach, based on state machines [83], ensures that every update is propagated to all sites, and that all sites process updates in unique order. This can be achieved using a reliable and totally ordered communication primitive. Our protocols implement such primitive, and provide a consistent order for operation processing on replicated objects. The algorithms are fully distributed and utilize available hardware multicast. Moreover, they exhibit low response time and improved concurrency.

- We designed a decision procedure based on a variation of "approval" voting to fasten the construction of a linear order for causal messages. Our protocols perform better than previous work, and are more scalable: they adapt better to various network topologies, and the latency of message delivery increases slowly with the number of processors. In some situations, the synchronization constraints are reduced by 20%.

[13] Surprisingly, most of the research results in voting theory have been gathered during the past fifty years.

[14] Consider a 200,00$ platform of 1995, as planned by Dec, Intel, Hp or Sun. In [85], it is suggested that one of them is likely to offer a four way shared memory multiprocessor with 700 MIPS, 1 GBytes of main memory and a disk array (RAID) of 500 GBytes. Under these assumptions, such configuration is capable of performing about 3,500 TP1 benchmarks per second. Most current transaction processing systems need to run less than 100 transactions per second, and loads are going up slowly, that is, at less than 10 percent per year.

- We proposed a framework that allow operations on objects to be initiated in different orders as long the consistency of the data is preserved. The semantic based ordering multicast can deliver a message even if its ordinal position in the total order is unknown at that time. This is done by taking advantage of the commutativity property of operations. If no optimization based on commutativity is possible, this primitive reduces to our linear ordering primitive. In other words, it is always advantageous to use the semantic ordering protocol for replicated abstract data types.

- Furthermore, we used a notion of commutativity, termed forward commutativity, that depends on the state of objects. It was proposed by Weihl in [90] to reduce the constraints imposed by concurrency control protocols. He describes also a protocol to implement atomic actions with this type of commutativity. If concurrency control is based forward commutativity, then a deferred update recovery strategy (no-undo/redo) is required. This means that a transaction's updates need to be saved while the transaction runs, and the database is updated only when the transaction commits. Our semantic ordering protocol (Sop) is primarily defined for this algorithm. We found it adequate for databases that resides in main memory or for short-lived transactions, as all updates have to be kept in memory until commit time. This makes Sop suitable for transactional applications that necessitate high concurrency and low response time.

8.1 Future Areas Of Research

The focus of our work is on the development of a highly performance server for clusters of workstations. We target applications that require high available main-memory databases. In this report, we have investigated one facet of this puzzle, namely the construction of a communication substrate with strong semantics. However much more efforts lie head of us:

1. We believe that our protocols are very close to the maximum level of concurrency achievable. However with clever optimizations, it might be possible to develop algorithms that provide a higher level of concurrency. Although some theoretical work exist on broadcast consensus protocols [69, 73], we do not have a characterization of an optimal algorithm.

2. The above remark is especially true for the semantic ordering protocol: sites can relax a total order if their sequence of operations invocations is equivalent to the one defined by the total order. However, agreeing on an equivalent order still places strong constraints on message delivery. It might be possible to improve on our scheme.

3. Other aspects of a replicated main memory database (MMDB) need further research: when a site s recovers, state information has to be transferred to bring the database of s up-to-date. Processing should continue at other sites and s should recover fast. Incremental recovery algorithms [57] have been developed for the centralized case, but could change with when multiple copies are available.

4. Kumar and Burger have evaluated in [19] the performance of main memory recovery algorithms based on update in place and shadow approaches, but not for deferred update. However, this recovery strategy seems to place fewer restraints on concurrency and might be more adequate when a database is replicated.

References

[1] A. El Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant algorithm for replicated data management. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 215–229, Portland, OR, March 1985.

[2] D. Agrawal and A. El Abbadi. The generalized tree quorum protocol: An efficient approach for managing replicated data. *ACM Transactions on Database Systems*, 17(4):689–717, December 1992.

[3] D. Agrawal, A. El Abbadi, and A. K. Singh. Consistency and orderability: Semantics-based correctness criteria for databases. *ACM Transactions on Database Systems*, 18(3):460, September 1993.

[4] M. Ahamad and M. Ammar. Performance characterization of quorum-consensus algorithms for replicated data. *IEEE Transactions on Software Engineering*, 15(4):492–501, April 1989.

[5] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *6th International Workshop on Distributed Algorithms, Springer-Verlag*, pages 292–312, November 1992.

[6] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, Los Alamitos, CA, July 1992.

[7] Y. Amir, L. E. Moser, P.M. Melliar-Smith, V. Agrawala, and P. Ciarfella. Fast message ordering and membership using a logical token passing ring. In *International Conference on Distributed Computing Systems*, Pittsburg, PA, May 1993.

[8] E. Arjomandi, M. J. Fischer, and N. A. Lynch. Efficiency of synchronous versus asynchronous distributed systems. *Journal of the Association for Computing Machinery*, 30(3):449–456, July 1983.

[9] D. Barbara and H. Garcia-Molina. The vulnerability of vote assignments. *ACM Transactions on Computer Systems*, 4(3):187, August 1986.

[10] D. Barbara and H. Garcia-Molina. The reliability of voting mechanisms. *IEEE Transactions on Computers*, 36(10):1197–1208, October 1987.

[11] D. Barbara, H. Garcia-Molina, and Spauster. Protocols for dynamic vote reassignment. In *Proceedings of the 5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pages 195–205, New York, 1986.

[12] P. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Computing Surveys*, 13(2), June 1981.

[13] P. A. Bernstein and N. Goodman. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions on Database Systems*, 9(4):596, December 1984.

[14] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-wesley, Reading, MA, 1987.

[15] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[16] K. P. Birman. The process group approach to reliable distributed computing. Technical report, Cornell University, July 1991.

[17] R. Braden. Extending {TCP} for transactions. Technical Report RFC 1379, University of Southern California, November 1992.

[18] Steven J. Brams and Peter C. Fishburn. *Approval Voting.* Birkauser, Boston, 1983.

[19] A. Burger and V. Kumar. Performance measurement of main memory database recovery algorithms based on update-in-place and shadow approaches. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):567–571, December 1992.

[20] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. In *Proceedings of the 10th Annual Symposium on Principles of Distributed Computing*, pages 325–340. ACM, August 1991.

[21] M. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *Transactions on Computer Systems*, 3(1):63–75, 1985.

[22] M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[23] J. M. Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[24] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77, May 1985.

[25] S. Y. Cheung, M. Ahamad, and M. H. Ammar. Optimizing vote and quorum assignments for reading and writing replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):387, September 1989.

[26] S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):582–59, December 1992.

[27] J. H. Conway. *On Numbers and Games.* Academic Press, London, 1976.

[28] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real-Time Systems*, 2(1):57–94, 1990.

[29] F. Cristian. Fault-tolerance in the advanced automation system. *ACM SIGOPS Operating Systems Review*, 25(2):117, April 1991.

[30] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–188, 1991.

[31] F. Cristian, H. Aghili, and R. Strong. Clock synchronization in the presence of omission and performance failures, and processor joins. Technical report, IBM Almaden Research Center, 1992.

[32] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. Research Report RJ 4540, IBM Research Laboratory, San Jose, CA, December 1984.

[33] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM Computing Surveys*, 17(3):341, September 1985.

[34] S. Deering. Host extensions for IP multicasting. Technical Report RFC 1112, Stanford University, August 1989.

[35] A. Demers, D. Greene, C. Hauser, W. I., J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pages 1–12. ACM, August 1987.

[36] D. Dolev, S. Kramer, and D. Malki. Total ordering of messages in broadcast domains. Technical report, The Hebrew University of Jerusalem, November 1992.

[37] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered broadcast in asynchronous environments. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, June 1993.

[38] D. Dolev, L. Lamport, M. Pease, and R. Shostak. The Byzantine generals. In *'Concurrency Control and Reliability in Distributed Systems', Bhargava(ed)*. Van Nostrand Reinhold, New York, NY, 1987.

[39] D. Dolev, D. Malki, and Ray Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS TR94-6, The Hebrew University of Jerusalem, November 1993.

[40] A. R. Downing, I. G. Greenberg, and J. M. Peha. OSCAR: A system for weak-consistency replication. In *IEEE Workshop on Management of Replicated Data*, Houston, TX, November 1990.

[41] D. L. Eager and K. C. Sevcik. Achieving robustness in distributed database systems. *ACM Transactions on Database Systems*, 8(3), September 1983.

[42] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[43] P.A. Fishwick. *Simulation Model Design and Execution: Building Digital Worlds.* Prentice Hall, 1994. Simpack is available via anonymous ftp at ftp.cis.ufl.edu:pub/simdigest/tools/simpack-2.23.tar.Z.

[44] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, 31(1):12–46, January 1982.

[45] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the Association for Computing Machinery*, 34(4):841–860, October 1985.

[46] D. K. Gifford. Weighted voting for replicated data. In *Proceeding of the 7th ACM Symposium on Operating Systems Principles, Pacific Grove CA*, December 1979.

[47] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32, February 1986.

[48] M. Herlihy. Dynamic quorum adjustment for partitioned data. *ACM Transactions on Database Systems*, 12(2):170–175, June 1987.

[49] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[50] M. Herlihy, L. Zahn, T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant. Concurrency and availability as dual properties of replicated atomic data network computing architecture. *Journal of the Association for Computing Machinery*, 37(2):257–278, April 1990.

[51] S. Jajodia and D. Mutchler. Dynamic voting algorithms for maintaining the consistency of a replicated database. *ACM Transactions on Database Systems*, 15(2):230–253, June 1990.

[52] M. F. Kaashoek and A. S. Tanenbaum. Efficient reliable group communication for distributed systems. submitted for publication, 1994.

[53] A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Transactions on Computers*, 40(9):996–1004, September 1991.

[54] A. Kumar and A. Segev. Cost and availability tradeoffs in replicated data concurrency control. *ACM Transactions on Database Systems*, 18(1):102–131, March 1993.

[55] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transaction on Computer Systems*, 10(4):360, November 1992.

[56] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[57] E. Levy and A. Silberschatz. Incremental recovery in main memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):529–540, December 1992.

[58] B. Liskov, S. Ghemawat, R. Gruber, P. Johnson, L. Shrira, and M. Williams. Replication in the Harp file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 226–38. Association for Computing Machinery SIGOPS, October 1991.

[59] S. Luan and V. Gligor. A fault-tolerant protocol for atomic broadcast. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):271–285, 1990.

[60] N. A. Lynch, M. Merritt, W. Weihl, and A. Fekete. *Atomic Transactions*. Morgan Kaufmann Publishers, San Mateo, CA, 1994.

[61] P. M. Melliar-Smith, L. Moser, and V. Agrawala. Broadcast protocols for distributed systems. *IEEE transactions on parallel and distributed systems*, 1(1), 1990.

[62] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the 9th International Conference on Distributed Computing Systems*, pages 129–135, Newport Beach, CA, 1989. IEEE Computer Society , Washington, DC.

[63] P. M. Melliar-Smith and L. E. Moser. Analyzing the performance of complex systems. In *Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems – Performance Evaluation Review*, pages 1–10, San Diego, CA, May 1991. University of California at Santa Barbara.

[64] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing fault-tolerant replicated objects using psync. In *Proceedings of the 8th Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, WA, October 1989.

[65] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, University of Arizona, 1991.

[66] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on a partial order. In *Proceedings of the International Working Conference on Dependable Computing for Critical Applications*, New York, NY, May 1991.

[67] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *Proceedings of the 11th International Conference on Distributed Computing Systems*, pages 480–489, Arlington, TX USA, 1991. IEEE Computer Society , Washington, DC.

[68] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Asynchronous fault-tolerant total ordering algorithms. *SIAM Journal on Computing*, 22(4):727–750, 1993.

[69] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Necessary and sufficient conditions for broadcast consensus protocols. *Journal of Distributed Computing*, 7(2):12–27, December 1993.

[70] Y. Moses, D. Dolev, and J. H. Halpern. Cheating husbands and other stories: A case study of knowledge, action, and communication (preliminary version). In *Proceedings of the Fourth Annual ACM Symposium on Principles of Distributed Computing*, pages 215–223, August 1985.

[71] Hervé Moulin. Fairness and strategy in voting. In *Proceedings of the 33th Symposia in Applied Mathematics on Fair Allocation*, Cambridge, MA, 1985. American Mathematical Society.

[72] Hervé Moulin. *Axioms of Cooperative Decision Making*. Cambridge University Press, Cambridge, MA, 1988.

[73] Sape Mullender. *Distributed Systems*. ACM, New York, 1993.

[74] B. M. Oki and B. Liskov. Viewstamped replication: A general primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, Toronto, Ontario, August 1988.

[75] J.-F. Paris. Voting with witnesses: A consistency scheme for replicated files. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, page 36, Cambridge, MA, 1986. IEEE Computer Society, Washington, DC.

[76] J.-F. Paris and D. E. Long. Efficient dynamic voting algorithms. In *Proceedings of the IEEE International Conference on Data Engineering*, page 268, Los Angeles, CA, February 1988.

[77] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[78] C. Pu, J. D. Noe, and A. Proudfoot. Regeneration of replicated objects: a technique and its eden implementation. *IEEE Transactions on Software Engineering*, 14(7):936–945, July 1988.

[79] R. Van Renesse, K. Birman, R. Cooper, B. Glade, and P. Stephenson. Reliable multicast between micro-kernels. In *USENIX Workshop on Micro-Kernels and Other Kernel Architectures*, pages 269–284, Seattle, WA, April 27-28 1992. USENIX.

[80] R. Van Renesse and A. S. Tanenbaum. Voting with ghosts. In *The 8th International Conference on Distributed Computing Systems*, pages 456–462. IEEE, June 1988.

[81] A. Ricciardi. The group membership problem in asynchronous distributed systems. submitted for publication, 1994.

[82] K. Salem and H. Garcia-Molina. System M: A transaction procressing testbed for memory resident data. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):161–172, March 1990.

[83] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(3):299, December 1990.

[84] M. Spasojevic and P. Berman. Voting as the optimal static pessimistic scheme for managing replicated data. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):64–73, January 1994.

[85] M. Stonebraker. *Readings in Database Systems, 2nd ed.* Morgan Kaufmann, San Mateo, CA, 1993.

[86] Philip D. Straffin. *Topics in the Theory of Voting.* Birkhauser, Boston, 1980.

[87] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[88] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers*, 37(12):1488–1505, December 1988.

[89] W. E. Weihl. The impact of recovery on concurrency control. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 259–269. ACM Press , Philadelphia, PA, March 1989.

[90] W. E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Transactions on Programming Languages and Systems*, 11(2):249–282, April 1989.

[91] W. E. Weihl. Linguistic support for atomic data types. *ACM Transactions on Programming Languages and Systems*, 12(2):178–202, April 1990.