LINEARIZABLE SHARED OBJECTS FOR ASYNCHRONOUS MESSAGE PASSING SYSTEMS

by

Andrew A. Wahbe

A thesis submitted in conformity with the requirements for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Copyright © 2000 by Andrew A. Wahbe
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-50378-X
Abstract

Linearizable Shared Objects for Asynchronous Message Passing Systems

Andrew A. Wahbe

Master of Science

Graduate Department of Computer Science

University of Toronto

2000

We consider linearizable implementations of shared objects in asynchronous message passing systems using failure detectors. Objects are replicated for reasons of fault-tolerance, and Atomic Multicast is used to ensure that the same sequence of operations is performed on each replica. However, for an object implementation to be linearizable, the ordering of this sequence must be consistent with the real-time ordering of operations. We identify a property called Linear Order that allows this constraint to be met, and show that this property is satisfied by every Atomic Multicast algorithm.

Operations that do not modify an object’s state do not require the ordering semantics of Atomic Multicast. We therefore define a new multicast primitive, called Linear Multicast, that satisfies Linear Order without totally ordering messages, and allows these operations to be implemented efficiently. We also show that Chandra and Toueg’s Atomic Multicast algorithm can be enhanced to also implement Linear Multicast.
Acknowledgements

I would like to thank my supervisor, Vassos Hadzilacos, for his advice and guidance during the completion of this work. I would also like to thank Ken Sevcik for reading a draft of this thesis, and suggesting corrections and improvements. I am grateful to my family and friends for their encouragement and moral support throughout this endeavor. Finally, I would like to thank the University of Toronto and the Natural Sciences and Engineering Research Council of Canada for the financial support that made this work possible.
# Contents

1 Introduction

1.1 Background ........................................ 1

1.2 Overview of Thesis .................................... 8

2 Conventions and System Model .......................... 9

2.1 System Model ........................................ 9

2.1.1 Processes ........................................ 9

2.1.2 Groups ........................................... 10

2.1.3 Time ............................................... 10

2.1.4 Configurations .................................... 10

2.1.5 Failures ........................................... 11

2.1.6 Failure Detectors ................................ 11

2.1.7 Algorithms and Runs .............................. 15

2.1.8 Causal Precedence ................................. 18

2.1.9 Removing an Event from a Run .................... 19

2.1.10 Fault-Tolerant Solutions to Problems .......... 20

2.1.11 Algorithms that Use Other Algorithms .......... 21

2.2 Analyzing the Performance of Algorithms .......... 22

2.3 Properties of Algorithms using Failure Detectors .. 23
3 Replicated Shared Objects

3.1 Problem Definition ........................................... 34
  3.1.1 Shared Objects ........................................... 34
  3.1.2 Operations .............................................. 36
  3.1.3 Histories .................................................. 37
  3.1.4 Replicated Objects ...................................... 38
  3.1.5 Consistency Protocols ................................... 39

3.2 Multicast Communication .................................... 40
  3.2.1 Reliable Multicasts .................................... 41
  3.2.2 Atomic Multicast ........................................ 42
  3.2.3 A Real-Time Ordering of Messages ....................... 44

3.3 A Simple Linearizable Consistency Protocol .................. 47
  3.3.1 Description .............................................. 47
  3.3.2 Fault-Tolerance ......................................... 49
  3.3.3 Proof of Correctness .................................... 51
  3.3.4 Performance Analysis ..................................... 60

4 Read/Update Shared Objects ..................................... 61
  4.1 Defining Read/Update Shared Objects ....................... 61
  4.2 Linear Multicast ........................................... 63
  4.3 Proving the Linearizability of Read/Update Objects ........ 64
  4.4 A Consistency Protocol for Read/Update Objects using Strong Linear Multicast ........................................... 69
    4.4.1 Description ............................................. 69
    4.4.2 Fault-Tolerance ....................................... 73
    4.4.3 Proof of Correctness .................................... 74
    4.4.4 Performance Analysis ..................................... 83
## A Consistency Protocol for Read/Update Objects using Weak Linear Multicast

A Consistency Protocol for Read/Update Objects using Weak Linear Multicast ................................................................. 84

### 5 Implementing Atomic and Linear Multicasts

5.1 Implementing Atomic Multicast ........................................... 87
  5.1.1 A Reliable Multicast Algorithm ..................................... 88
  5.1.2 The Repeated Group Consensus Problem ......................... 89
  5.1.3 An Atomic Multicast Algorithm .................................... 93
  5.1.4 Fault-Tolerance ..................................................... 94
  5.1.5 Proof of Correctness .............................................. 95
  5.1.6 Performance Analysis ............................................. 96

5.2 Implementing Linear Multicast ......................................... 97
  5.2.1 A Causal Analysis of Consensus ................................ 100
  5.2.2 Keeping Track of Consensus Rounds ............................. 106
  5.2.3 Adding Linear Multicast to the Atomic Multicast Algorithm 109
  5.2.4 Fault-Tolerance ..................................................... 111
  5.2.5 Proof of Correctness .............................................. 111
  5.2.6 Performance Analysis ............................................. 117

### 6 Conclusions and Discussion

6.1 Conclusions ............................................................... 119

6.2 Related Work ............................................................ 123
  6.2.1 Sequential Consistency versus Linearizability .................. 123
  6.2.2 Other Optimizations of Active Replication ....................... 131
  6.2.3 Other Methods of Replica Management ............................ 132

6.3 Future Work .............................................................. 136

### A Implementing Atomic Multicast

A.1 Proof of Correctness ................................................... 138
List of Figures

1.1 A sample execution of a consistency protocol based on active replication. 5

3.1 The problems of inconsistency and contamination. 44

3.2 An argument for the real-time ordering of messages. 45

3.3 A general algorithm for implementing linearizable replicated objects using Atomic Multicast. 48

4.1 A consistency protocol for read/update objects using Atomic Multicast and Strong Linear Multicast. 70

4.2 A Consistency Protocol for Read/Update Objects using Atomic Multicast and Weak Linear Multicast. Only the portions of the algorithm that differ from the algorithm in Figure 4.1 are shown. 84

5.1 A Reliable Multicast algorithm based on a Reliable Broadcast algorithm given by Hadzilacos and Toueg [29] that was adapted from the “Diffusion Algorithm” of Christian, et al. [11]. 88

5.2 An Atomic Multicast algorithm based on the Chandra and Toueg Atomic Broadcast algorithm [8]. 93

5.3 An algorithm that adds Linear Multicast to the Atomic Multicast algorithm in Figure 5.2. The type of Linear Multicast that is implemented depends on the failure detector that is used to solve Repeated Group Consensus. 110
6.1 A problem using (Pairwise) Atomic Multicast in a sequentially consistent implementation of read/update objects with fast reads. ................ 125
6.2 An argument for causally ordered read request messages when implementing sequentially consistent read/update objects. ......................... 128
6.3 An argument for Context Multicast when implementing sequentially consistent read/update objects. ................................................... 129
Chapter 1

Introduction

1.1 Background

Communication between processes in a distributed system is usually studied in the context of one of two system models: the shared memory model, and the message passing model. In the first model, processes share physical memory, and can therefore communicate through shared objects (e.g., registers) that reside in this memory. In the latter model, processes do not share physical memory and must instead communicate through the exchange of messages across network links.

In the design of distributed applications, the use of shared objects has been found to be a more effective communication paradigm than message passing since it allows processes to share a global view of the system [2]. However, a system may not always be afforded the luxury of shared physical memory in which to store these objects. It would therefore be advantageous to be able to emulate shared objects in message passing systems. Such an emulation would not only allow shared object algorithms to be used in message passing systems, but would also allow many distributed services to be implemented as shared objects. For instance, a name service can be modeled as a shared set of name-address pairs.
We investigate the issues of emulating shared objects in the context of an asynchronous message passing system. The system is asynchronous in that there is no bound on the relative speeds of processors or on message transmission times. The correctness of any algorithm that is designed for such a system model is independent of such bounds, and is therefore not "broken" if unexpectedly high loads slow down a process, or cause a message to be delayed. Algorithms designed for the asynchronous system model also have a wide applicability since they can be used in a variety of systems with different timing characteristics.

A simple implementation of a shared object $x$ in an asynchronous message passing system would be to store $x$ in the local memory of a particular process, called the server process for $x$. A client process that wishes to perform an operation on $x$ would simply communicate its intention to perform the operation to this server process. The server process would then perform the operation on behalf of the client process and then return the result of the operation to the client process.

An important concern in the area of distributed systems, however, is that of fault tolerance. Namely, it is not acceptable for a distributed service to become unavailable due to process failures. A process may fail in numerous ways. For instance, a process may fail by crashing, whereby the process stops executing, and sends and receives no more messages; or, a process may experience a malicious failure (sometimes called a Byzantine failure) by behaving in a way that is not consistent with its specification, such as by sending erroneous messages. In this thesis, we consider shared object implementations that tolerate crash failures. The implementation of object $x$ described above obviously cannot tolerate the failure of the server process for $x$.

Implementing services or objects that can tolerate crash failures requires replication, i.e., multiple replicas of an object are maintained by some subset of the processes in the system. Thus, the invocation of operations on an object $x$ is managed by a set of server processes rather than by a single server process. Replication provides fault tolerance
since the object is available despite the failure of a process that maintains a replica of the
object. The number of failures that are tolerated by a particular object implementation
varies from system to system but usually depends on the degree to which the object is
replicated. For example, an object that is replicated at \( n \) different servers can tolerate
\( n-1 \) failures in some systems but only \( \lceil \frac{n-1}{2} \rceil \) failures in others. In addition to facilitating
fault tolerance, replication can also be used to improve performance. For example, an
object replica that is stored in proximity to a client process can also act as a cache,
reducing the latency of operations on an object; however, in this thesis, fault tolerance
is our primary reason for replicating an object.

The behavior of a shared object when accessed concurrently by multiple client pro-
cesses is constrained by consistency criteria. These criteria allow a programmer to make
specific assumptions about the behavior of objects when using them to design distributed
applications. For instance, strong consistency criteria such as sequential consistency [36]
and linearizability [30] require that the results returned by operations allow the processes
to agree on a total ordering of those operations; this provides the illusion that the objects
are being accessed sequentially rather than concurrently.

For example, consider a read/write register object whose initial value is 0. If a process
\( p \) writes the value 1 to the register, and another process \( q \) reads 0 from the register, then
the read operation must precede the write operation in any total ordering of operations
(as long as no process writes 0 back to the register). In other words, it appears to the
processes executing the operations that the read operation preceded the write operation.
If either sequential consistency or linearizability are satisfied, then in any (concurrent)
execution, the participating processes should be able to agree on a sequential ordering of
the operations executed based on the results returned by those operations.

Sequential consistency not only requires that the processes agree on a total ordering
of the operations executed, but also that this ordering is consistent with the order of the
operations executed by the same process. Thus, if a process writes a value to a register
object and then reads that same register object, then if sequential consistency is satisfied, the value read must reflect the fact that the previous write had taken place, allowing the write operation to precede the read operation in any total ordering of operations.

Linearizability is a stronger consistency criterion than sequential consistency in that it has the added requirement that the total ordering of the operations executed be consistent with the real-time ordering of those operations. Specifically, if an operation \( \text{op}_1 \) is initiated by a process \( p \) after the execution of an operation \( \text{op}_0 \) by another process \( q \) has completed, then \( \text{op}_0 \) must precede \( \text{op}_1 \) in the total ordering of operations. For instance, if the operation \( \text{op}_0 \) in the preceding example writes 1 to a register object whose initial value was 0, and \( \text{op}_1 \) reads that same register object, then \( \text{op}_1 \) must return 1 (if no other writes to the register take place). In contrast, if only sequential consistency is guaranteed, then the read operation could return 0. Thus, linearizability provides the illusion that an operation takes effect at some discrete point during the execution of that operation. In this thesis, we will focus our attention on linearizable implementations of replicated shared objects.

A consistency protocol is an algorithm that is responsible for ensuring that the implementation of a set of replicated shared objects satisfies a particular consistency criterion. *Active replication* [34, 52], often called the *state-machine approach*, is a common method for maintaining the consistency of replicated objects. When using this approach, a client process that wishes to invoke an operation on an object forwards a *request message* to all of the server processes maintaining a replica of that object. Server processes handle the request messages one at a time. A server process executes each requested operation on its replica of the object, and returns the result of the operation to the requesting client in a *reply message*.

For example, if a client process \( c \) wishes to perform an operation \( \text{op} \) on an object \( x \), then the execution of a consistency protocol based on active replication would proceed as follows:
1. \( c \) forwards a request message to every server process that maintains a replica \( x \), specifying that it wishes to perform \( op \) on \( x \).

2. Upon receiving this message, a server process performs the operation \( op \) on its replica of \( x \), updating the state of the replica and storing the result in a reply message that is then sent to \( c \).

3. \( c \) waits until it receives a single reply to its request, and accepts the result contained in the reply message as its result to \( op \).

This execution is portrayed in Figure 1.1. The horizontal lines depict the execution of processes over time; the passage of time moves from left to right. The dots on the horizontal lines represent events in the execution of the processes; in this case, these events are the sending and receiving of messages. The arrows that connect the events represent messages sent from one process to another. The boxed segment of each horizontal line delimits the period of time devoted to the execution of the operation.

It is well known that if objects are deterministic, i.e., if the result of every operation is a deterministic function of the state of the object when the operation is performed, then a consistency protocol that is based on this approach ensures linearizability as long as the server processes that maintain a particular object process identical sequences of request messages in any execution [26]. This can be enforced by sending the request messages to the server processes using a communication primitive called Atomic Multicast. Informally,
Atomic Multicast ensures that server processes agree on the set of messages that are to be delivered, and that every server delivers these messages in the same order.

The properties of Atomic Multicast are obviously sufficient to ensure sequential consistency since all operations can be totally ordered in the order in which their respective request messages are delivered. The linearizability consistency criterion, however, requires that this total ordering preserves the real-time ordering of operations. This would imply that the communication primitive used to transmit the request messages also satisfies some property that ensures that the message ordering preserves the real-time ordering of messages.

In this thesis, we will formalize such a property, called Linear Order, and show that this property is satisfied by any Atomic Multicast algorithm. Informally, the Linear Order property requires that if a message \( m \) is multicast at some time \( t \), then no process can deliver \( m \) before it has delivered every message that was delivered by any process before \( t \). It is by satisfying this property that Atomic Multicast enforces the real-time ordering of messages that is required by linearizability.

We will also consider a class of shared objects called read/update objects. A read/update object defines two types of operations by which it can be accessed: update operations that modify the state of the object, and read operations which do not modify the state of the object. We assert that since read operations do not change the state of an object, the delivery semantics of Atomic Multicast are not necessary for sending the request messages for read operations to server processes.

We define a multicast primitive, called Linear Multicast, that has weaker delivery semantics than Atomic Multicast. Informally, Linear Multicast only requires that processes agree on the delivery of a message if the sender of the message is correct, and that messages are delivered in Linear Order. We will show that we can use Linear Multicast to implement read operations in an Active Replication scheme as long as the update operations are implemented using Atomic Multicast.
Unfortunately, there can be no fault-tolerant implementation of Atomic Multicast in an asynchronous system due to the relationship between the Atomic Multicast problem and the Consensus problem [19]. Roughly speaking, the Consensus problem allows a set of processes, each having an initial value, to agree on a common value. The Fischer-Lynch-Patterson (FLP) impossibility result [20] states that Consensus cannot be solved in an asynchronous system where even a single process can crash. This result stems from the fact that processes in an asynchronous system are unable to distinguish a crashed process from one that is only very slow. Since Consensus can be reduced to Atomic Multicast [13] (i.e., Consensus can be solved using Atomic Multicast), the result applies to Atomic Multicast as well.

Randomized solutions to Consensus for asynchronous systems have been presented [10]; however, in this thesis, we are focusing on deterministic algorithms. It has also been shown that, for various models of partial synchrony, fault-tolerant solutions to Consensus exist [13, 16]. However, if we study the Consensus problem in the context of a “less-general” partial synchrony model, we lose the features that made the study of Consensus in the asynchronous model attractive; namely, the tolerance of timing delays, and the wide applicability of solutions for the asynchronous model.

In order to circumvent the FLP impossibility result, Chandra and Toueg proposed the addition of unreliable failure detectors to the asynchronous system model [8], and showed that for various classes of unreliable failure detectors, Consensus can be solved. Chandra and Toueg also presented an algorithm that solved Atomic Multicast using Consensus, showing that Atomic Multicast can also be solved using unreliable failure detectors.

None of these unreliable failure detectors, however, can be implemented in an asynchronous system though it has been shown that all of the unreliable failure detectors presented by Chandra and Toueg can be implemented in some (reasonable) partially synchronous system [8]. Therefore, by incorporating unreliable failure detectors into our asynchronous system model, we are really exploring the solutions to problems in system
models, such as the partial synchrony model, that do make some timing assumptions; however, these assumptions are encapsulated in the properties satisfied by the failure detectors themselves.

We will show that if Consensus is solved using a deterministic algorithm that uses failure detectors, then Chandra and Toueg's Atomic Multicast algorithm can be augmented so that it solves Linear Multicast as well as Atomic Multicast. This implementation of Linear Multicast is less costly than the Atomic Multicast algorithm in terms of both latency of message delivery and the number of messages sent. We also define sufficient criteria which, if satisfied by any other Atomic Multicast algorithm, also allow Linear Multicast to be implemented efficiently.

1.2 Overview of Thesis

The remainder of this thesis is organized as follows: Chapter 2 introduces conventions used in this thesis and formally defines our system model. Chapter 3 formally defines the linearizable consistency protocol problem and the Linear Order property satisfied by Atomic Multicast; a general linearizable consistency protocol is then presented that uses Atomic Multicast. Chapter 4 defines read/update objects and Linear Multicast, and presents a consistency protocol for linearizable read/update objects using Atomic Multicast and Linear Multicast. Chapter 5 presents our implementation of Linear Multicast and discusses the addition of Linear Multicast to other Atomic Multicast algorithms. Finally, Chapter 6 presents our conclusions, and discusses related and future work.
Chapter 2

Conventions and System Model

2.1 System Model

In this section, we will present the system model that will be used in this thesis. This model is based upon system models used by Fischer, Lynch and Patterson [20], Chandra and Toueg [8], Chandra, Hadzilacos and Toueg [7] and, Lamport [35].

2.1.1 Processes

The system consists of a set $\Pi = \{p_1, p_2, p_3, \ldots, p_n\}$ of processes that are linked by a totally connected point-to-point network. Each process has a local state that cannot be accessed by any other process in the system. Processes must therefore communicate by passing messages to each other across the network links. Every message that is sent is a member of the set of valid messages $M$. Each message $m \in M$ contains a field $sender(m)$ which uniquely identifies the sender of the message, as well as a field denoted by $seqno(m)$ which contains the sequence number of the message (i.e., $1 +$ the number of messages previously sent by $sender(m)$). Together, these two fields uniquely identify a message. Every message that is sent is addressed to a single recipient process that is denoted by $dest(m)$. The network is reliable in that each message that is sent is guaranteed to
eventually be received by its recipient, though messages are not necessarily received in the order in which they are sent.

### 2.1.2 Groups

A group identifies a subset of the processes in the system. If a process $p$ is in the subset of processes that is identified by a group $G$, then we say that $p$ is a member of $G$, and write $p \in G$. Every group is a member of the set of groups $\mathcal{G}$. All groups are assumed to be static and finite. The membership of each group $G$ is available to every process in the system, including processes that are not members of $G$. Given a process $p$, we write $\mathcal{G}|p$ to denote the set of groups of which $p$ is a member.

The set of all groups $\mathcal{G}$ always contains a special group $G_\Pi$ called the system group. The membership of this group is $\Pi$.

### 2.1.3 Time

We assume the existence of a discrete global clock. Processes do not have access to the global clock; it is only used to simplify the model. The range of the clock’s ticks $\mathcal{T}$ is the set of natural numbers. The system is asynchronous as there is no bound on message delays (i.e., the time between the sending of a message and the receipt of that message) or on the time between the execution of events by a process (events are formally defined in Subsection 2.1.7).

### 2.1.4 Configurations

The state of the system at any time $t \in \mathcal{T}$ is characterized by a configuration. A configuration $C$ is a pair $(L, M)$ where $L$ is a function mapping each process to its local state, and $M$ is a message buffer — a set of messages that have been sent but have not yet been received. Given a configuration $C = (L, M)$ and a process $p$, $M|p$ denotes the set of
messages addressed to $p$ that have been sent but have not yet been received. Given any initial configuration $I = (L_0, M_0)$ (the configuration of the system at time 0), $M_0 = \emptyset$, and the local state that a process $p$ is mapped to by $L_0$ is referred to as an initial state of $p$.

2.1.5 Failures

Processes in the system may fail by crashing. If a process crashes at a time $t$, then that process' local state is set to a special crashed state, and that process executes no events at any time $t' \geq t$. A failure pattern $\mathcal{F}$ is a function from $\mathcal{T}$ to $2^\Pi$, where $\mathcal{F}(t)$ denotes the set of processes that have crashed through time $t$. We say that a process $p$ fails at time $t$ in a failure pattern $\mathcal{F}$ if and only if $p \notin \mathcal{F}(t - 1)$ and $p \in \mathcal{F}(t)$. We assume that a process cannot recover once it has failed, therefore $\forall t \in \mathcal{T} : \mathcal{F}(t) \subseteq \mathcal{F}(t + 1)$. We write $\text{faulty}(\mathcal{F})$ to denote the set of processes that crash in an entire failure pattern $\mathcal{F}$; thus, $\text{faulty}(\mathcal{F}) = \bigcup_{t \in \mathcal{T}} \mathcal{F}(t)$. We write $\text{correct}(\mathcal{F})$ to denote $\Pi - \text{faulty}(\mathcal{F})$.

2.1.6 Failure Detectors

Informally, every process in $\Pi$ has access to a failure detector module that outputs a set of processes that it currently suspects to have crashed. We say that a process $p$ suspects $q$ at time $t$, if the failure detector module owned by $p$ outputs $q$ at time $t$. To formally define failure detectors, we first define a failure detector history $\mathcal{H}$ as a function from $\Pi \times \mathcal{T}$ to $2^\Pi$. $\mathcal{H}(p, t)$ denotes the set of processes that process $p$ suspects at time $t$.

A failure detector $\mathcal{D}$ is a function that maps each failure pattern $\mathcal{F}$ to a set of failure detector histories $\mathcal{D}(\mathcal{F})$. This is the set of failure detector histories that are possible in an execution with failure detector $\mathcal{D}$ and failure pattern $\mathcal{F}$. 
Failure Detector Properties

Several classes of failure detectors were introduced by Chandra and Toueg [8]. These classes are defined by the completeness and accuracy properties that are satisfied by the failure detectors in each class. In this thesis, we will only be considering a subset of these failure detector classes. All of the failure detectors that we will consider satisfy the following completeness property:

**Strong Completeness** Eventually every process that crashes is permanently suspected by every correct process. Formally, a failure detector \( H \) satisfies Strong Completeness if:

\[
\forall F, \forall H \in D(F), \exists t \in T, \forall p \in \text{faulty}(F), \forall q \in \text{correct}(F), \forall t' \geq t : p \in H(q, t')
\]

The failure detectors considered can thus be characterized by the accuracy properties that they satisfy in addition to Strong Completeness. The strongest accuracy property that we will consider requires that a failure detector makes no mistakes:

**Strong Accuracy** No process is suspected before it crashes. Formally, a failure detector \( D \) satisfies Strong Accuracy if:

\[
\forall F, \forall H \in D(F), \forall t \in T, \forall p, q \in \Pi - F(t) : p \notin H(q, t)
\]

Since Strong Accuracy is difficult to achieve in practice, Chandra and Toueg also define weaker accuracy properties: Eventual Strong Accuracy, Weak Accuracy, and Eventual Weak Accuracy. The first of these properties only requires that eventually no mistakes are made:

**Eventual Strong Accuracy** There is a time after which each correct process is not suspected by any correct process. Formally, a failure detector \( D \) satisfies Eventual Strong Accuracy if:

\[
\forall F, \forall H \in D(F), \exists t \in T, \forall t' \geq t, \forall p, q \in \text{correct}(F) : p \notin H(q, t')
\]


The remaining two properties are weaker versions of Strong Accuracy and Eventual Strong Accuracy in which we are assured that one correct process is never suspected (or eventually never suspected). Specifically, these accuracy properties are defined as follows: Weak Accuracy – some correct process is never suspected; and Eventual Weak Accuracy – there is a time after which some correct process is never suspected by any correct process.

The definitions of Weak Accuracy and Eventual Weak Accuracy that we will use in this thesis are slightly different than those given by Chandra and Toueg since we will be using failure detectors to solve “group-wide” problems, whereas Chandra and Toueg explored the use of failure detectors to solve “system-wide” problems; their system model did not include groups and the problems studied were not defined in terms of groups.

The system-wide definitions of Weak Accuracy and Eventual Weak Accuracy are not conducive to the solution of group-wide problems since even though they guarantee that some process in the system will never be suspected, there are no guarantees on the group membership of this unsuspected process. Thus, the processes in every group may be required to interact with the unsuspected process, even though this process may not be a member of all of the groups in the system.

We will therefore specify Weak Accuracy (and Eventual Weak Accuracy) so that some process in each group is never suspected (or eventually never suspected). However, implicit to the definitions given by Chandra and Toueg is the assumption that there is always at least one correct process in the system. For instance, if we use the obvious adaptation of Eventual Weak Accuracy to our model – some correct process in each group is never suspected – we would need to assume that there is always at least one correct process in each group. This is a much stronger assumption than that made by Chandra and Toueg. Instead, we will use the following definition:

**Eventual Weak Accuracy** If there is a correct process in a group, then there is a time after which some correct process in that group is never suspected by any correct
process. Formally, a failure detector $D$ satisfies Eventual Weak Accuracy if:

$$\forall \mathcal{F}, \forall \mathcal{H} \in \mathcal{D}(\mathcal{F}), \forall G \in \mathcal{G} :$$

$$(\exists p \in G : p \in \text{correct}(\mathcal{F})) \implies$$

$$(\exists t \in T, \exists p \in \text{correct}(\mathcal{F}), \forall t' \geq t, \forall q \in \text{correct}(\mathcal{F}) : p \in G \land p \notin \mathcal{H}(q, t'))$$

However, if we used the analogous definition of Weak Accuracy – if there is a correct process in a group, then some correct process in that group is never suspected – then processes would be able to determine that all of the processes in a group were faulty before the processes in that group had failed. Specifically, if every process in a group $G$ is faulty then every process in $G$ can be suspected before it fails; furthermore, if every process in $G$ is suspected then all of the processes in $G$ must be faulty. This would make our version of Weak Accuracy somewhat more powerful than that given by Chandra and Toueg, since it would allow a failure detector to "predict" failures. For this reason, we will instead define Weak Accuracy as follows:

**Weak Accuracy** If, at any time, some process in a group has not crashed, then some process in that group that has not yet crashed has never been suspected by any process. Formally, a failure detector $D$ satisfies Weak Accuracy if:

$$\forall \mathcal{F}, \forall \mathcal{H} \in \mathcal{D}(\mathcal{F}), \forall t \in T, \forall G \in \mathcal{G} :$$

$$(\exists p \in G : p \in \Pi - \mathcal{F}(t)) \implies$$

$$(\exists p \in G, \forall t' \leq t, \forall q \in \Pi - \mathcal{F}(t') : p \in \Pi - \mathcal{F}(t') \land p \in \mathcal{H}(q, t'))$$

**Failure Detector Classes**

Using these completeness and accuracy properties, we now define the failure detector classes that we will consider:

**Perfect Failure Detectors** The set of failure detectors that satisfy Strong Completeness and Strong Accuracy. This set is denoted by $\mathcal{P}$. Any failure detector in $\mathcal{P}$ is called a *Perfect* failure detector.
Strong Failure Detectors The set of failure detectors that satisfy Strong Completeness and Weak Accuracy. This set is denoted by $S$. Any failure detector in $S$ is called a Strong failure detector.

Eventually Perfect Failure Detectors The set of failure detectors that satisfy Strong Completeness and Eventual Strong Accuracy. This set is denoted by $\diamond P$. Any failure detector in $\diamond P$ is called an Eventually Perfect failure detector.

Eventually Strong Failure Detectors The set of failure detectors that satisfy Strong Completeness and Eventual Weak Accuracy. This set is denoted by $\diamond S$. Any failure detector in $\diamond S$ is called an Eventually Strong failure detector.

We can see from these definitions that every Perfect failure detector is also a Strong failure detector; or $P \subseteq S$. Similarly, $P \subseteq \diamond P$, $S \subseteq \diamond S$ and $\diamond P \subseteq \diamond S$.

2.1.7 Algorithms and Runs

An algorithm $A$ is a collection of deterministic automata, one for each process in the system. An execution of an algorithm is modeled by a run. Informally, a run contains a sequence of events. Each event occurs at a specific time and is executed by a single process. The execution of an event by a process represents a state transition by that process' automaton. The local state of a process $p$ may therefore be changed by the execution of an event by $p$. This is the only way that the local state of a process can be changed.

Every event is of a specific type. Two types of events, send events and receive events, are associated with the exchange of messages by processes. A send event denotes the sending of a message by a process, and a receive event denotes the receipt of a message by a process. All other events are called internal events as they can only affect the local state of the process that executes them. Any event $e$ can be denoted by a tuple $(p, type, args)$ where $p$ is the process that executed the event, $type$ is the event type, and
args is a sequence arguments that are specific to the event's type. For instance, every send event is of the form \((p, \text{send}, m)\), where \(m\) is the message sent and \(p\) is the process that sent \(m\). Similarly, every receive event is of the form \((p, \text{recv}, m)\), where \(m\) is the message received and \(p\) is the process that received \(m\).

The automata that compose an algorithm define the events that processes may execute given their local state. Receive events have the added requirement that the received message must be in the message buffer when the event is executed. We say that an event \(e\) executed by a process \(p\) is applicable to a configuration \(C = (L, M)\), if \(e\) is not a receive event or if \(e = (p, \text{recv}, m)\) and \(m \in M\). The automaton associated with a process \(p\) also defines a transition function that maps a configuration \(C\) and an event \(e\) executed by \(p\) that is applicable to \(C\) to a new configuration that is denoted by \(e(C)\).

Let \(C = (L, M)\) be a configuration, let \(e\) be an event that is executed by a process \(p\) and is applicable to \(C\), and let \(e(C) = (L', M')\) be the new configuration that is mapped to by \(p\)'s automaton. By the definitions of events and configurations, the following hold:

1. \(\forall p' \in \Pi: p' \neq p \implies L'(p') = L(p')\);

2. if \(e\) is a send event where \(m\) is the message that is sent then \(M' = M \cup \{m\}\); and

3. if \(e\) is a receive event where \(m\) is the message that is received then \(M' = M - \{m\}\).

A schedule \(S\) of an algorithm \(A\) is a sequence of events that is executed by the processes whose automata compose \(A\). We say that a sequence of events \(S = e_1, e_2, \ldots\) is applicable to an initial configuration \(C_0\) if (1) \(S\) is an empty sequence, or (2) \(e_1\) is applicable to \(C_0\), \(e_2\) is applicable to \(e_1(C_0)\), etc.

An algorithm \(A\) using a failure detector \(D\) is an algorithm in which a process may query its failure detector module. The querying of a failure detector by a process \(p\) is modeled by an internal event of the form \((p, \text{query}, d)\) where \(d\) is the result returned by the process’ failure detector. These events are called query events.
A partial run $R$ of an algorithm $A$ using a failure detector $D$ is a tuple $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$ where $\mathcal{F}$ is a failure pattern, $\mathcal{H}_D \in D(\mathcal{F})$ is a history of failure detector $D$ for failure pattern $\mathcal{F}$, $I$ is an initial configuration, $S$ is a finite schedule of $A$ and $T$ is a finite sequence of increasing time values indicating the time at which each event in $S$ occurred. Given any process $p$ and a run $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$, if $p \in \text{faulty}(\mathcal{F})$ then we say that $p$ is faulty in $R$; otherwise, we say that $p$ is correct in $R$.

Any partial run $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$ must satisfy the following properties:

- $|S| = |T|$;
- $S$ is applicable to $I$;
- a process cannot take a step after it crashes;
- if a process $p$ executes a query event $(p, \text{query}, d)$ at time $t$, then $d = \mathcal{H}_D(p, t)$.

A run $R$ of an algorithm $A$ using a failure detector $D$ is a tuple $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$ where $\mathcal{F}$ is a failure pattern, $\mathcal{H}_D \in D(\mathcal{F})$ is a history of failure detector $D$ for failure pattern $\mathcal{F}$, $I$ is an initial configuration, $S$ is an infinite schedule of $A$ and $T$ is an infinite sequence of increasing time values indicating the time at which each event in $S$ occurred. A run must satisfy the above properties of a partial run as well as the following properties:

- every correct process executes an infinite number of steps; and
- every correct process eventually receives every message sent to it.

Given a partial run $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$ and a run (or partial run) $R' = (\mathcal{F}', \mathcal{H}_D', I', S', T')$, we say that $R$ is a prefix of $R'$ if $I = I'$, $S \preceq S'$, $T \preceq T'$, and $\forall i: 1 \leq i \leq |T|: \mathcal{F}(T[i]) = \mathcal{F}'(T[i])$ and $\forall p \in \Pi - \mathcal{F}(T[i]): \mathcal{H}_D(p, T[i]) = \mathcal{H}_D'(p, T'[i])$. If $R$ is a prefix of $R'$ then we say that $R'$ is an extension of $R$.

Given a run $R = (\mathcal{F}, \mathcal{H}_D, I, S, T)$ and a run $R' = (\mathcal{F}', \mathcal{H}_D', I', S', T')$, we say that $R'$ is identical to $R$ until time $t$ if:
• \( I' = I \);

• \( \forall i \) such that \( T[i] \leq t \): \( S'[i] = S[i] \) and \( T'[i] = T[i] \);

• If \( i \) is the largest integer such that \( T[i] \leq t \), then \( \forall j > i: T'[j] > t \);

• \( \forall t' \leq t \): \( \mathcal{F}'(t') = \mathcal{F}(t') \); and

• \( \forall t' \leq t, \forall p \in \Pi - \mathcal{F}(t') : \mathcal{H}'(p, t') = \mathcal{H}(p, t) \).

If \( R \) is identical to \( R' \) until time \( t \), then both \( R \) and \( R' \) are extensions of a partial run \( R^t \) that is the longest prefix of \( R \) (or \( R' \)) in which no events are executed after time \( t \); however, if the last event executed in \( R^t \) is not executed at time \( t \) then extensions of \( R^t \) exist that are not be identical to \( R \) until \( t \). Specifically, if the last event in \( R^t \) is executed at time \( t' < t \), then any extension of \( R' \) in which an event is executed at any time \( t'' \) such that \( t' < t'' \leq t \) is not identical to \( R \) until \( t \).

### 2.1.8 Causal Precedence

The causal precedence relation is an irreflexive relation on the events in a run \( R \) that was originally defined as the "happens before" relation by Lamport [35]. Given any two events \( e_0 \) and \( e_1 \) in a run \( R \), we say that \( e_0 \) causally precedes \( e_1 \), written \( e_0 \rightarrow_R e_1 \), if and only if:

1. \( e_0 \) and \( e_1 \) are events that are executed by the same process \( p \) and \( p \) executes \( e_0 \) before \( e_1 \); or

2. \( e_0 \) is the send event of a message \( m \), and \( e_1 \) is the receive event of \( m \); or

3. there exists an event \( e_2 \) in \( R \) such that \( e_0 \rightarrow_R e_2 \) and \( e_2 \rightarrow_R e_1 \).

The causal precedence relation is acyclic (since \( e_0 \rightarrow_R e_1 \) only if \( e_0 \) was executed before \( e_1 \) in \( R \)) and is transitive by Clause 3, and therefore induces an irreflexive partial
order on the events in a run. We say that \( e_0 \) and \( e_1 \) are concurrent in \( R \) if \( e_0 \not\rightarrow_R e_1 \) and \( e_1 \not\rightarrow_R e_0 \). Given a run \( R \), we say that a process \( p \) has an event \( e \) in its causal past at time \( t \) in \( R \) if the latest event executed by \( p \) at \( t \) in \( R \) is \( e' \) and \( e = e' \) or \( e \rightarrow_R e' \). If \( e_0 \rightarrow_R e_1 \) and it is clear from the context that \( e_0 \) and \( e_1 \) are events in run \( R \) then we will drop the subscript \( R \) and write: \( e_0 \rightarrow e_1 \).

Let \( E = e_0, e_1, \ldots, e_n \) be a sequence of events in \( R \). We say that \( E \) is a path from an event \( e \) to a process \( p \) at time \( t \) in a run \( R \) if \( e_0 = e \), \( e_n \) is the latest event executed by \( p \) at \( t \) in \( R \), and \( e_0 \rightarrow e_1 \rightarrow \ldots \rightarrow e_n \). From the definition of a path and the causal past of a process \( p \) at time \( t \) in a run \( R \) we can see that an event \( e \) is in the causal past of \( p \) at time \( t \) in \( R \) if and only if there exists a path from \( e \) to \( p \) at time \( t \) in \( R \). We say that \( E \) is a path from an event \( e \) to a process \( p \) at time \( t \) through a group \( G \) in a run \( R \) if \( E \) is a path from \( e \) to \( p \) at time \( t \) in \( R \) and every event in \( E \) is executed by a process in \( G \).

### 2.1.9 Removing an Event from a Run

In many of the proofs in this paper, a run \( R' \) is constructed from a run \( R \) by removing an event from \( R \). Formally, let \( R = (\mathcal{F}, \mathcal{H}, I, S, T) \) be any (partial) run of an algorithm \( A \) and let \( e \) denote an event in \( R \). The tuple \( (\mathcal{F}, \mathcal{H}, I, S', T') \) is the result of removing \( e \) from \( R \) if \( S' \) and \( T' \) are the sequences resulting from removal of the \( i \)th element from \( S \) and \( T \) respectively, where \( e \) is the \( i \)th event in \( R \) (e.g., \( S' = S[1], S[2], \ldots, S[i-2], S[i-1], S[i+1], S[i+2], \ldots \) and \( T' = T[1], T[2], \ldots, T[i-2], T[i-1], T[i+1], T[i+2], \ldots \)).

However, if an event \( e \) is removed from a run \( R = (\mathcal{F}, \mathcal{H}, I, S, T) \) of an algorithm \( A \), the resulting tuple is not necessarily a run of \( A \), since the sequence of events in this new tuple may not be a schedule of \( A \). Specifically, it may be the case that some of the events in \( S \) that are causally preceded by \( e \) are executed in \( A \) only if \( e \) has previously been executed. We would therefore like to define a transformation of any run \( R \) in which an event \( e \) as well as all of the events that are causally preceded by \( e \) are removed from \( R \).
Before formalizing this transformation, we define what it means to remove multiple events from a run. Let \( e_1, e_2, \ldots \) be any subsequence of events in \( R \). The tuple \( (F, H, I, S', T') \) is the result of removing \( e_1, e_2, \ldots \) from \( R \) if \( S' \) and \( T' \) are the sequences resulting from the removal of the \( i_1 \)th, \( i_2 \)th, etc. elements from \( S \) and \( T \) respectively, where \( e_1 \) is the \( i_1 \)th event in \( S \), \( e_2 \) is the \( i_2 \)th event in \( S \), etc.

We now define what it means to causally remove an event from a (partial) run \( R \) of an algorithm \( A \). Let the sequence \( e_1, e_2, \ldots \) be the sequence of events in \( R \) that are causally preceded by \( e \), arranged in their order of appearance in \( R \). The tuple \( (F, H, I, S', T') \) is the result of causally removing \( e \) from \( R \) if that tuple is the result of removing \( e, e_1, e_2, \ldots \) from \( R \). Note that the events executed by each process \( p \) in \( S' \) is a prefix of the events executed by \( p \) in \( S \). Thus, if \( S \) is a schedule of an algorithm \( A \) then \( S' \) is a schedule of \( A \). Therefore, if \( R \) is a partial run of \( A \) then \( (F, H, I, S', T') \) is also a partial run of \( A \). However, if \( R \) is not a partial run (i.e., \( R \) is infinite) then \( (F, H, I, S', T') \) may not be a run of \( A \) since it may be the case that correct processes do not execute an infinite number of events in \( S' \).

2.1.10 Fault-Tolerant Solutions to Problems

The algorithms considered in this thesis are used to solve various problems. A problem is defined by a set of properties that runs must satisfy. An algorithm \( A \) is a failure-free solution to a problem \( P \) if every run \( R = (F, H, I, S, T) \) of \( A \) where faulty(\( F \)) = \( \emptyset \) satisfies \( P \).

In this thesis, however, we are interested in algorithms that solve problems despite process failures. Such algorithms are called fault-tolerant. The fault tolerance of an algorithm is usually measured by the number of system-wide process failures that the algorithm can tolerate before it no longer solves a problem. A system-wide measure of fault tolerance is not suitable for our purposes because the correctness of the algorithms considered in this thesis depends on the reliability of particular groups of processes rather
than on the reliability of the system as a whole. The fault tolerance of an algorithm is therefore measured by the number failures that may occur within each group.

The fault tolerance of an algorithm $A$ with respect to all groups in the system is represented by a tolerance function $\Gamma$ which is a function from $G$ to the set of non-negative integers where $\Gamma(G) \leq |G|$ for any group $G \in G$. The system formed by an algorithm $A$ and a tolerance function $\Gamma$ is a subset $\mathcal{R}$ of the runs of $A$ such that a run $R = (\mathcal{F}, \mathcal{H}, I, S, T)$ of $A$ is in $\mathcal{R}$ if and only if $\forall G \in G : |\text{faulty}(\mathcal{F}) \cap G| \leq \Gamma(G)$.

We say that an algorithm $A$ is a $\Gamma$-tolerant solution to a problem $P$ if every run $R = (\mathcal{F}, \mathcal{H}, I, S, T)$ of $A$, such that $\forall G \in G : |\text{faulty}(\mathcal{F}) \cap G| \leq \Gamma(G)$, satisfies $P$; in other words, every run in the system formed by $A$ and $\Gamma$ satisfies $P$.

When an algorithm $A$ uses a failure detector $\mathcal{D}$, we say that $A$ is a $\Gamma$-tolerant solution to $P$ using $\mathcal{D}$ if every run $R = (\mathcal{F}, \mathcal{H}, I, S, T)$ of $A$ using $\mathcal{D}$, such that $\forall G \in G : |\text{faulty}(\mathcal{F}) \cap G| \leq \Gamma(G)$, satisfies $P$. Given a class of failure detectors $C$, we say that an algorithm $A$ is a $\Gamma$-tolerant solution to $P$ using $C$ if $\forall \mathcal{D} \in C$, $A$ is an $\Gamma$-tolerant solution to $P$ using $\mathcal{D}$; thus, for every run $R$ in the system formed by $A$ and $\Gamma$, if $R$ uses any failure detector $\mathcal{D} \in C$ then $R$ satisfies $P$.

2.1.11 Algorithms that Use Other Algorithms

Many of the algorithms presented in this thesis use another algorithm as a subroutine in order to solve a problem. We formally define what it means for one algorithm to use another in terms of subruns. Given a run $R = (\mathcal{F}, \mathcal{H}, I, S, T)$ and a run $R = (\mathcal{F}', \mathcal{H}', I', S', T')$ we say that a run $R'$ is a subrun of $R$ if and only if $\mathcal{F}' = \mathcal{F}$, $I = I'$, $\mathcal{H}' = \mathcal{H}$, $S'$ is a subsequence of $S$, and $T'$ is a subsequence of $T$ such that for every event-time pair $(S'[i], T'[i])$, there exists an event-time pair $(S[j], T[j])$ such that $S'[i] = S[j]$ and $T'[i] = T[j]$. We say that an algorithm $A$ uses an algorithm $A'$ if for every run $R$ or $A$ there is a subrun $R'$ of $R$ that is a run of $A'$. 
2.2 Analyzing the Performance of Algorithms

The performance of the algorithms presented in this thesis will be analyzed based on two criteria: the number of messages sent, and the latency of the algorithm. The first criterion measures the total number of messages that must be sent by the processes executing the algorithm in order to solve a particular problem. The second criterion measures the time taken by the algorithm to solve a particular problem. The way in which this time is measured depends on how we determine the point in a run at which an algorithm begins to solve a problem, and the point in a run at which a problem has been solved. This will vary from problem to problem.

In analyzing an algorithm with respect to either criterion, we will assume that runs are failure free and that failure detectors make no mistakes since we consider this to be the most frequent case.

When analyzing the latency of an algorithm, we will make the additional assumption that messages have a bounded delay time of $d_m$. Since local processing time is often negligible compared to message delay times, we will also assume that all local processing is instantaneous. Thus, the latency of an algorithm reflects the number of “communication phases” required by that algorithm to solve a problem. We will measure the performance of an algorithm using latency rather than the number of communication phases since this term “phase” is used ambiguously in the literature, i.e., in some cases a “two phase algorithm” refers to an algorithm with a latency of $2d_m$ while in other cases it refers to an algorithm with a latency of $3d_m$, or even $4d_m$.

It is important to note that the timing assumptions that are made when calculating the latency of an algorithm cannot be used by the algorithms themselves. These assumptions are only made to facilitate the performance analysis; the algorithms must work correctly even if these timing assumptions do not hold. While our performance analysis is not rigorous and is based on many assumptions, it is sufficient to serve as a rough gauge of the relative performance of algorithms.
2.3 Properties of Algorithms using Failure Detectors

In this section, we introduce a set of theorems that are instrumental to the proofs of our multicast implementations in Chapter 5. These theorems are presented here, with the introduction of our system model, since they are a consequence of that model and do not depend on any specific problem or algorithm that we are considering in this thesis. Before stating and proving these theorems, we present a series of lemmata that we will use in our proofs.

The first lemma states that given any failure detector history \( H \), any failure pattern \( F \), and any failure pattern \( F' \) that is identical to \( F \) until some time \( t \), there is an Eventually Perfect failure detector \( D \) and a failure detector history \( H' \) such that \( H' \) is identical to \( H \) until \( t \) and \( H' \in D(F') \).

**Lemma 2.3.1** Let \( F \) and \( F' \) be any two failure patterns, let \( t \) be any time in \( T \), and let \( H \) be any failure detector history such that \( H \in D(F) \) for any failure detector \( D \). If \( \forall t' \leq t : F'(t') = F(t) \) then there exists a failure detector \( D' \in \Diamond \mathcal{P} \) and a failure detector history \( H' \in D'(F') \) such that \( \forall t' \leq t, \forall p \in \Pi - F'(t') : H'(p, t') = H(p, t') \).

**Proof:** Let \( \mathcal{D}_\Diamond \mathcal{P} \) be the weakest failure detector in \( \Diamond \mathcal{P} \): the failure detector consisting of all failure detector histories that satisfy Strong Completeness and Eventual Strong Accuracy. More formally, for every failure pattern \( F \), \( \mathcal{D}_\Diamond \mathcal{P}(F) \) is the set of failure detector histories such that \( \forall H \in \mathcal{D}_\Diamond \mathcal{P}(F), \exists t \in T, \forall t' \geq t, \forall p \notin \text{faulty}(F) : q \in \text{faulty}(F) \iff q \in H(p, t) \).

Let \( H' \) be any failure detector history that is identical to \( H \) until time \( t \), satisfies Strong Completeness, and after time \( t \), satisfies Strong Accuracy. Formally, \( H' \) is any failure detector history such that:

(a) \( \forall t' \leq t, \forall p \in \Pi - F'(t') : H'(p, t') = H(p, t') \);

(b) \( \exists t' \in T, \forall t'' \geq t', \forall p \in \text{faulty}(F), \forall q \in \text{correct}(F) : p \in H'(q, t'') \) and
(c) ∀t′ > t, ∀p ∈ Π − F′(t′) : q ∈ H′(p, t′) → q ∈ F′(t′).

By this definition, we can see that H' ∈ D_D(F'). □

We now give a similar lemma for Perfect failure detectors. We note that this lemma has the additional requirement that the given failure detector history H is the output of a Perfect failure detector for the given failure pattern F.

Lemma 2.3.2 Let F and F' be any two failure patterns, let t be any time in T, and let H be any failure detector history such that H ∈ D(F) for any failure detector D ∈ P. If ∀t' ≤ t : F'(t') = F(t) then there exists a failure detector D' ∈ P and a failure detector history H' ∈ D'(F') such that ∀t' ≤ t, ∀p ∈ Π − F'(t') : H'(p, t') = H(p, t').

Proof: Let D_P be the weakest failure detector in P: the failure detector consisting of all failure detector histories that satisfy Strong Completeness and Strong Accuracy. More formally, for every failure pattern F, D_P(F) is the set of failure detector histories such that ∀H ∈ D_P(F), ∃t ∈ T, ∀p ∈ faulty(F), ∀q ∈ correct(F), ∀t' ≥ t : p ∈ H(q, t) and ∀H ∈ D_P(F), ∀t ∈ T, ∀p, q ∈ Π − F(t) : p ∉ H(q, t).

Let H' be any failure detector history that is identical to H until time t, satisfies Strong Completeness, and after time t, satisfies Strong Accuracy. Formally, H' is any failure detector history such that:

(a) ∀t' ≤ t, ∀p ∈ Π − F'(t') : H'(p, t') = H(p, t');

(b) ∃t' ∈ T, ∀t'' > t', ∀p ∈ faulty(F), ∀q ∈ correct(F) : p ∈ H''(q, t''); and

(c) ∀t' > t, ∀p ∈ Π − F'(t') : q ∈ H'(p, t') → q ∈ F'(t').

Since D ∈ P, H ∈ D(F), and ∀t' ≤ t, ∀p ∈ Π − F'(t') : F'(t') = F(t') ∧ H'(p, t') = H(p, t'), H' satisfies Strong Accuracy with respect to F' until time t. By definition, H' satisfies Strong Accuracy with respect to F' after t, and also satisfies Strong Completeness. Therefore, H' ∈ D_P(F'). □
Next, we present another similar lemma for Strong failure detectors. Like the previous lemma, this lemma requires that the given failure detector history $\mathcal{H}$ is the output of a Strong failure detector for the given failure pattern $\mathcal{F}$. This lemma also has an additional requirement that the second failure pattern $\mathcal{F}'$ allows the new failure detector history $\mathcal{H}'$ to be identical to $\mathcal{H}$ until $t$ and still satisfy the Weak Accuracy property. Specifically, some correct process in each group (or the last process to fail if there is no correct process) must not be suspected before time $t$ in $\mathcal{H}$.

**Lemma 2.3.3** Let $\mathcal{F}$ and $\mathcal{F}'$ be any two failure patterns, let $t$ be any time in $\mathcal{T}$, and let $\mathcal{H}$ be any failure detector history such that $\mathcal{H} \in \mathcal{D}(\mathcal{F})$ for any failure detector $\mathcal{D} \in \mathcal{S}$. If:

1. $\forall t' \leq t : \mathcal{F}'(t') = \mathcal{F}(t')$, and

2. $\forall G \in \mathcal{G}, \forall t' > t : (\exists p \in G : p \in \Pi - \mathcal{F}'(t')) \implies (\exists p \in G, \forall t'' \leq t, \forall q \in \Pi - \mathcal{F}'(t'') : p \in \Pi - \mathcal{F}'(t') \land p \notin \mathcal{H}(q, t''))$

then there exists a failure detector $\mathcal{D}' \in \mathcal{S}$ and a failure detector history $\mathcal{H}' \in \mathcal{D}'(\mathcal{F}')$ such that $\forall t' \leq t, \forall p \in \Pi - \mathcal{F}'(t') : \mathcal{H}'(p, t') = \mathcal{H}(p, t')$.

**Proof:** Let $\mathcal{D}_S$ be the weakest failure detector in $\mathcal{S}$: the failure detector consisting of all failure detector histories that satisfy Strong Completeness and Weak Accuracy. More formally, for every failure pattern $\mathcal{F}$, $\mathcal{D}_S(\mathcal{F})$ is the set of failure detector histories such that $\forall \mathcal{H} \in \mathcal{D}_S, \exists t \in \mathcal{T}, \forall p \in \text{faulty}(\mathcal{F}), \forall q \in \text{correct}(\mathcal{F}), \forall t' \geq t : p \in \mathcal{H}(q, t)$ and $\forall \mathcal{H} \in \mathcal{D}_S, \forall G \in \mathcal{G}, \exists p \in \text{correct}(\mathcal{F}), \forall t \in \mathcal{T}, \forall q \in \Pi - \mathcal{F}(t) : p \notin \mathcal{H}(q, t)$.

Let $\mathcal{H}'$ be any failure detector history that is identical to $\mathcal{H}$ until time $t$, satisfies Strong Completeness, and after time $t$, satisfies Strong Accuracy. Formally, $\mathcal{H}'$ is any failure detector history such that:

(a) $\forall t' \leq t, \forall p \in \Pi - \mathcal{F}'(t') : \mathcal{H}'(p, t') = \mathcal{H}(p, t')$;

(b) $\exists t' \in \mathcal{T}, \forall t'' \geq t', \forall p \in \text{faulty}(\mathcal{F}), \forall q \in \text{correct}(\mathcal{F}) : p \in \mathcal{H}'(q, t'')$; and
(c) \( \forall t' > t, \forall p \in \Pi - \mathcal{F}'(t') : q \in \mathcal{H}'(p, t') \implies q \in \mathcal{F}'(t') \). 

Since \( D \) satisfies Weak Accuracy, \( \mathcal{H} \in D(\mathcal{F}) \), and \( \mathcal{F}' \) is identical to \( \mathcal{F} \) until \( t \), \( \mathcal{H}' \) satisfies Weak Accuracy with respect to \( \mathcal{F}' \) until \( t \); or more formally: \( \forall G \in G, \forall t' \leq t : (\exists p \in G : p \in \Pi - \mathcal{F}'(t')) \implies (\exists p \in G, \forall t'' \leq t', \forall q \in \Pi - \mathcal{F}'(t'') : p \in \Pi - \mathcal{F}'(t') \land p \notin \mathcal{H}'(q, t'') \).

By definition, \( \mathcal{H}' \) satisfies Strong Accuracy after \( t \). Thus, by 2, \( \mathcal{H}' \) satisfies Weak Accuracy with respect to \( \mathcal{F}' \) after \( t \); or more formally: \( \forall G \in G, \forall t' > t : (\exists p \in G : p \in \Pi - \mathcal{F}'(t')) \implies (\exists p \in G, \forall t'' \leq t', \forall q \in \Pi - \mathcal{F}'(t'') : p \in \Pi - \mathcal{F}'(t') \land p \notin \mathcal{H}'(q, t''). \)

\( \mathcal{H}' \) therefore satisfies Weak Accuracy with respect to \( \mathcal{F}' \). Since \( \mathcal{H}' \), by definition, also satisfies Strong Completeness, \( \mathcal{H}' \in D_S(\mathcal{F}') \). \( \square \)

The following lemma allows us to take the failure pattern and the failure detector history that were shown to exist by any of the three preceding lemmata and use them to construct a new run.

**Lemma 2.3.4** Let \( \mathcal{R} \) be the system that is formed by an algorithm \( A \) and a tolerance function \( \Gamma \), and let \( R = (F, H, I, S, T) \) be a run in \( \mathcal{R} \). Let \( F' \) be a failure pattern, let \( H' \) be a failure detector history, and let \( t \) be a time in \( T \). If \( \forall t' \leq t : F'(t') = F(t') \), \( \forall G \in G : |\text{faulty}(F') \cap G| \leq \Gamma(G) \), and \( \forall t' \leq t, \forall p \in \Pi - F'(t') : H'(p, t') = H(p, t') \) then there exists a run \( R' = (F', H', I', S', T') \) in \( \mathcal{R} \) that is identical to \( R \) until \( t \).

**Proof (sketch):** We must show that there exists an initial configuration \( I' \), a schedule \( S' \) of \( A \), and an increasing sequence of times \( T' \) such that \( R' = (F', H', I', S', T') \) is a run in \( \mathcal{R} \) and \( R' \) executes identically to \( R \) until \( t \).

Trivially, let \( I' = I \). Let \( j \) be the largest integer such that \( T[j] \leq t \), and let \( T' \) be any increasing sequence of times such that:

(a) \( \forall i \leq j : T'[i] = T[i] \), and \( \forall k > j : T'[k] > t \).

Let \( S' \) be any schedule of \( A \) such that:
(b) \( \forall i \leq j: S'[i] = S[i] \);

(c) \( |S| = |T| \);

(d) \( S' \) is applicable to \( I' \);

(e) \( \forall p \in \Pi, \forall t \in T, \forall i: T'[i] = t \) and \( S'[i] = (p, -, -) \) only if \( p \notin F(t) \);

(f) \( \forall p \in \text{correct}(F') \): \( p \) executes an infinite number of events in \( S' \);

(g) \( \forall p \in \text{correct}(F') \): \( p \) eventually receives every message that is sent to it;

(h) if for any integer \( i \), any process \( p \in \Pi \), and any set of processes \( d \subseteq \Pi \), \( S'[i] = (p, \text{query}, d) \) and \( T'[i] = t' \) then \( \mathcal{H}'(p, t') = d \).

The properties that define \( S' \), \( T' \), \( I' \), \( \mathcal{H}' \) and \( F' \) are trivially consistent with a few exceptions. In particular, properties (b) and (d) would be inconsistent if the first \( j \) events in \( S \) were not applicable to \( I' \). This is not the case, however, since \( S \) is applicable to \( I \) and \( I' = I \). Properties (a), (b) and (e) would also be inconsistent if there existed some process \( p \) such that \( p \) executed an event in \( S \) at some time \( t' \leq t \), and \( p \in F'(t') \). However, there can be no such process since \( \forall t' \leq t : F'(t') = F(t) \) and no process executes an event after it has crashed in \( R \). Finally, properties (b) and (h) would be inconsistent if at some time \( t' \leq t \), some process \( p \in \Pi - F(t) \) executed a query event \( (p, \text{query}, d) \) in \( R \) such that \( \mathcal{H}'(p, t') \neq d \). However, since \( R \) is a run, \( \forall p \in \Pi, \forall t' \in T: \) if \( p \) executes a query event \( (p, \text{query}, d) \) at time \( t' \) in \( R \) then \( \mathcal{H}(p, t') = d \). Therefore, since \( \forall t' \leq t, \forall p \in \Pi - F'(t) : \mathcal{H}'(p, t') = \mathcal{H}(p, t') \) and \( \forall t' \leq t : F'(t') = F(t') \), no such query event is executed in \( R \) at any time.

Properties (c) through (h) imply that the tuple \( R' = (F', \mathcal{H}', I', S', T') \) is a run of algorithm \( A \). Run \( R' \) is identical to \( R \) until time \( t \) by the definitions of \( F' \), \( \mathcal{H}' \), and \( I' \), as well as properties (a) and (b). Since for every group \( G \in \mathcal{G} \): \( |\text{faulty}(F') \cap G| \leq \Gamma(G) \), \( R' \) is in \( \mathcal{R} \). \( \square \)
We now present the theorems that we will later use in our proofs in Chapter 5. These theorems are defined in terms of two types of properties: *execution properties* and *process execution properties*. An execution property \( \phi \) is a predicate of a run \( R \) and a time \( t \), i.e., given a system of runs \( \mathcal{R} \), \( \phi \) is a function from \( \mathcal{R} \times \mathcal{T} \) to \{true, false\}. Given an execution property \( \psi \), a run \( R \), and a time \( t \), we say that \( \phi(R, t) \) holds if and only if \( \psi(R, t) = \text{true} \). Examples of execution properties are: "some process has sent message \( m \)," and "a correct process has received message \( m \)."

A process execution property is a predicate of a run \( R \), a time \( t \), and a process \( p \); or, given a system \( \mathcal{R} \), a process execution property \( \psi \) is a function from \( \mathcal{R} \times \mathcal{T} \times \Pi \) to \{true, false\}. Given a process execution property \( \psi \), a run \( R \), a time \( t \), and a process \( p \), we say that \( \psi(R, t, p) \) holds if and only if \( \psi(R, t, p) = \text{true} \). Examples of process execution properties are: "has received 3 messages," and "has not received a message from a faulty process."

Given a process execution predicate \( \psi \), a run \( R \) and a time \( t \), \( \text{ satisfy}(\psi, R, t) \) denotes the set of processes that satisfy \( \psi \) at time \( t \) in \( R \). More precisely, given any process \( p \in \Pi \), \( p \in \text{ satisfy}(\psi, R, t) \) if and only if \( \psi(R, t, p) \) holds.

An execution property \( \phi \) only depends on the past in a system \( \mathcal{R} \) if for every run \( R \) in \( \mathcal{R} \) and \( \forall t \in \mathcal{T} \), \( \phi(R, t) \) holds only if \( \phi(R', t) \) holds for every run \( R' \) that is identical to \( R \) until \( t \). Similarly, a process execution property \( \psi \) only depends on the past in a system \( \mathcal{R} \) if for any run \( R \), time \( t \), and process \( p \), \( \psi(R, t, p) \) holds only if \( \psi(R', t, p) \) holds for every run \( R' \) that is identical to \( R' \) until \( t \).

For example, the execution property "some process has sent message \( m \)" and the process execution property "\( p \) has received 3 messages" only depend on the past (in any system). However, the execution property "a correct process has received message \( m \)" and the process execution property "\( p \) has not received a message from a faulty process" do not only depend on the past since a process \( p \) that is correct in \( R \) may in fact be faulty in some run \( R' \) that is identical to \( R \) until \( t \), if \( p \) crashes after time \( t \) in \( R' \).
The theorems that we will now present apply to systems for which there is an execution predicate $\phi$ and a process execution property $\psi$ such that if $\phi(R, t)$ holds for any run $R$ of $A$ and any time $t$, then there exists a correct process $p$ such that $\psi(R, t, p)$. The theorems given below place constraints on the set of processes for which $\psi(R, t, p)$ holds. Specifically, for $\Gamma$-tolerant algorithms using Perfect failure detectors or Eventually Perfect failure detectors, there must be a group $G$ for which number of processes in $G$ that have crashed plus the number of processes in $G$ for which $\psi$ holds must be greater than $\Gamma(G)$.

For $\Gamma$-tolerant algorithms using Strong failure detectors the above constraint does not necessarily hold; however, we show that if this condition does not hold, then there exists a group $G$ such that $\Gamma(G) < |G|$ and there is a correct process in $G$ that is never suspected for which $\psi$ holds. This result is a consequence of the fact that Strong failure detectors can provide information about correct processes as well as faulty processes. Specifically, every process $p$ knows that for any group $G$ such that $\Gamma(G) < |G|$, some process that $p$ has never suspected is correct. Thus, if a process $p$ has suspected every process in a group $G$ but one, it can be deduced that this unsuspected process is correct, even if none of $p$'s suspicions are accurate. Thus, Strong failure detectors may reveal information about correct processes that cannot be provided by any of the other failure detector classes that we are considering.

**Theorem 2.3.1** Let $A$ be any algorithm that is a $\Gamma$-tolerant solution to a problem $P$ using $\mathcal{D}$. Let $R$ be the system formed by $A$ and $\Gamma$. Let $\phi$ be any execution predicate that only depends on the past in $R$, and let $\psi$ be any process execution predicate that only depends on the past in $R$. Further assume that for every run $R$ in $R$ that satisfies $P$ and every time $t$, if $\phi(R, t)$ holds then there exists a correct process $p$ such that $\psi(R, t, p)$ holds. For any run $R = (F, H, I, S, T)$ in $R$ and any time $t$, if $\phi(R, t)$ holds then for some group $G \in \mathcal{G}$ such that $\Gamma(G) < |G| : |(\text{sat}(\psi, R, t) \cup F(t)) \cap G| > \Gamma(G)$.

**Proof:** Assume, by way of contradiction, that for every group $G \in \mathcal{G}$ such that $\Gamma(G) <
\[ |G| : |(\text{satisfy}(\psi, R, t) \cup \mathcal{F}(t)) \cap G| \leq \Gamma(G). \] Trivially, the same property holds for every group \( G \in \mathcal{G} \) such that \( \Gamma(G) = |G| \).

Let \( \mathcal{F}' \) be a failure pattern such that satisfies the following properties:

(a) \( \forall t' \leq t : \mathcal{F}'(t') = \mathcal{F}(t') \);

(b) \( \forall G \in \mathcal{G} : \text{faulty}(\mathcal{F}') \cap G = (\text{satisfy}(\psi, R, t) \cup \mathcal{F}(t)) \cap G. \)

Note that (b) implies that \( \forall G \in \mathcal{G} : |\text{faulty}(\mathcal{F}') \cap G| \leq \Gamma(G). \)

By Lemma 2.3.1, there exists a failure detector history \( \mathcal{H}' \) such that \( \mathcal{H}' \in \mathcal{D}(\mathcal{F}') \) for some failure detector \( \mathcal{D} \in \Diamond \mathcal{P} \). By Lemma 2.3.4, there exists a run \( \mathcal{R}' \) in \( \mathcal{R} \) with failure pattern \( \mathcal{F}' \) and failure detector history \( \mathcal{H}' \) that is identical to \( \mathcal{R} \) until time \( t \). Since \( \phi \) and \( \psi \) only depend on the past, \( \phi(\mathcal{R}', t) \) holds and \( \text{satisfy}(\psi, \mathcal{R}', t) = \text{satisfy}(\psi, \mathcal{R}, t) \).

Therefore, by the definition of \( \mathcal{F}' \), and since \( G_{\Pi} \in \mathcal{G} \), if \( \psi(R, t, p) \) holds for any process \( p \) then \( p \) is faulty. Since \( \mathcal{R}' \) is in \( \mathcal{R} \) and uses a failure detector in \( \Diamond \mathcal{P} \), \( \mathcal{R}' \) satisfies \( \mathcal{P} \); therefore, since \( \phi(\mathcal{R}', t) \) holds, there exists a correct process \( p \) such that \( \psi(\mathcal{R}', t, p) \) holds. This is a contradiction. \( \square \)

Before presenting the remaining two theorems, we note that Theorem 2.3.1 not only applies to algorithms using \( \Diamond \mathcal{P} \), but also to algorithms using any failure detector class that includes \( \Diamond \mathcal{P} \), namely \( \Diamond \mathcal{S} \). This is because any any algorithm that is a \( \Gamma \)-tolerant solution to a problem \( \mathcal{P} \) using a failure detector class \( \mathcal{C} \), where \( \Diamond \mathcal{P} \subseteq \mathcal{C} \) is also a \( \Gamma \)-tolerant solution to \( \mathcal{P} \) using \( \Diamond \mathcal{P} \). Theorem 2.3.2 and Theorem 2.3.3 do not also apply to weaker failure detector classes since they require that the run \( \mathcal{R} \), for which the execution predicate \( \phi \) holds, uses \( \mathcal{S} \) or \( \mathcal{P} \) respectively.

**Theorem 2.3.2** Let \( A \) be any algorithm that is a \( \Gamma \)-tolerant solution to a problem \( \mathcal{P} \) using \( \mathcal{S} \). Let \( \mathcal{R} \) be the system formed by \( A \) and \( \Gamma \). Let \( \phi \) be any execution predicate that only depends on the past in \( \mathcal{R} \), and let \( \psi \) be any process execution predicate that only depends on the past in \( \mathcal{R} \). Further assume that for every run \( \mathcal{R} \) in \( \mathcal{R} \) that satisfies \( \mathcal{P} \) and every time \( t \),
if $\phi(R, t)$ holds then there exists a correct process $p$ such that $\psi(R, t, p)$ holds. For any run $R = (F, H, I, S, T)$ in $R$ using a failure detector $D \in S$ and any time $t$, if $\phi(R, t)$ holds then for some group $G \in \mathcal{G}$ such that $\Gamma(G) < |G| : |\{\text{satisfy}(\psi, R, t) \cup F(t)\} \cap G| > \Gamma(G)$ or $\psi(R, t, p)$ holds for some correct process $p \in G$ that is never suspected in $R$.

**Proof:** Let $\Pi_1$ be the set of all correct processes that are never suspected by any process in $R$. Formally, $\Pi_1 = \{p | p \in \text{correct}(R) \land \forall t \in T, \forall q \in F(t) : p \notin \mathcal{H}_D(q, t)\}$.

Assume, by way of contradiction, that for every group $G \in \mathcal{G}$ such that $\Gamma(G) < |G| : |\{\text{satisfy}(\psi, R, t) \cup F(t)\} \cap G| \leq \Gamma(G)$ and there is no correct process $p \in G$ such that $p$ is never suspected in $R$ and $\psi(R, t, p)$ holds (i.e., $\Pi_1 \cap G \cap \text{satisfy}(\psi, R, t) = \emptyset$). Trivially, $|\{\text{satisfy}(\psi, R, t) \cup F(t)\} \cap G| \leq \Gamma(G)$ for any group $G \in \mathcal{G}$ such that $\Gamma(G) = |G|$.

Let $F'$ be a failure pattern that satisfies the following properties:

(a) $\forall t' \leq t : F'(t') = F(t')$;

(b) $\forall G \in \mathcal{G} : \text{if } \Gamma(G) < |G| \text{ then } \text{faulty}(F') \cap G = (\{\text{satisfy}(\psi, R, t) \cup F(t)\} \cap G)$, otherwise $\text{faulty}(F') \cap G = G$;

(c) if at any time $t' > t$, a process in $G$ has not crashed then some process in $G$ that has not crashed was not suspected in $R$ before time $t$. Formally: $\forall G \in \mathcal{G}, \forall t' > t : (\exists p \in G : p \in \Pi - F'(t')) \implies (\exists p' \in G, \forall t'' \leq t, \forall q \in F(t'') : p \in F(t) \land p \notin \mathcal{H}(q, t''))$.

Note that (b) implies that $\forall G \in \mathcal{G} : |\text{faulty}(F') \cap G| \leq \Gamma(G)$. Property (a) trivially does not conflict with properties (b) and (c).

We now show that all three properties (a), (b) and (c) are consistent. Specifically, we must show that there is a run for which (a) and (b) hold such that for every group $G$, if at any time after $t$ there is a process in $G$ that has not crashed then there is a process in $G$ that has not crashed and was not suspected in $R$ before $t$.

Let $G$ be any group. If $\Gamma(G) < |G|$ then since $R$ uses a failure detector in $S$, there is a process $p \in G$ such that $p \in \Pi_1$. Since $\Pi_1 \cap G \cap \text{satisfy}(\psi, R, t) = \emptyset$, $p$ is correct in $F'$.
and \( p \) is not suspected in \( R \) before \( t \). If \( \Gamma(G) = |G| \) then every process in \( G \) is faulty by (b). However, as long as the last process to fail in \( G \) is a process that was not suspected before time \( t \) in \( R \), (c) holds. We can therefore assume that a failure pattern \( \mathcal{F}' \) exists that satisfies (a), (b), and (c).

By Lemma 2.3.3, there exists a failure detector history \( \mathcal{H}' \) such that \( \mathcal{H}' = \mathcal{D}'(\mathcal{F}') \) for some failure detector \( \mathcal{D}' \in \mathcal{S} \). By Lemma 2.3.4, there exists a run \( R' \) in \( \mathcal{R} \) with failure pattern \( \mathcal{F}' \) and failure detector history \( \mathcal{H}' \) that is identical to \( R \) until time \( t \). Since \( \phi \) and \( \psi \) only depend on the past, \( \phi(R', t) \) holds and \( \text{satisfy}(\psi, R', t) = \text{satisfy}(\psi, R, t) \).

Therefore, by the definition of \( \mathcal{F}' \), and since \( G_\Pi \in \mathcal{G} \), if \( \psi(R, t, p) \) holds for any process \( p \) then \( p \) is faulty. Since \( R' \) is in \( \mathcal{R} \) and uses a failure detector in \( \mathcal{S} \), \( R' \) satisfies \( P \); therefore, since \( \phi(R', t) \) holds, there exists a correct process \( p \) such that \( \psi(R', t, p) \) holds. This is a contradiction. \( \square \)

**Theorem 2.3.3** Let \( A \) be any algorithm that is a \( \Gamma \)-tolerant solution to a problem \( P \) using \( \mathcal{P} \). Let \( \mathcal{R} \) be the system formed by \( A \) and \( \Gamma \). Let \( \phi \) be any execution predicate that only depends on the past in \( \mathcal{R} \), and let \( \psi \) be any process execution predicate that only depends on the past in \( \mathcal{R} \). Further assume that for every run \( R \) in \( \mathcal{R} \) that satisfies \( P \) and every time \( t \), if \( \phi(R, t) \) holds then there exists a correct process \( p \) such that \( \psi(R, t, p) \) holds. For any run \( R = (\mathcal{F}, \mathcal{H}, I, S, T) \) in \( \mathcal{R} \) using a failure detector \( \mathcal{D} \in \mathcal{P} \) and any time \( t \), if \( \phi(R, t) \) holds then for some group \( G \in \mathcal{G} \) such that \( \Gamma(G) < |G| : |(\text{satisfy}(\psi, R, t) \cup \mathcal{F}(t)) \cap G| > \Gamma(G) \).

**Proof:** Assume, by way of contradiction, that for every group \( G \in \mathcal{G} \) such that \( \Gamma(G) < |G| : |(\text{satisfy}(\psi, R, t) \cup \mathcal{F}(t)) \cap G| \leq \Gamma(G) \). Trivially, the same property holds for any group \( G \in \mathcal{G} \) such that \( \Gamma(G) = |G| \).

Let \( \mathcal{F}' \) be a failure pattern such that satisfies the following properties:

(a) \( \forall t' \leq t: \mathcal{F}'(t') = \mathcal{F}(t') \);

(b) \( \forall G \in \mathcal{G}: \text{faulty}(\mathcal{F}') \cap G = (\text{satisfy}(\psi, R, t) \cup \mathcal{F}(t)) \cap G \).
Note that (b) implies that $\forall G \in \mathcal{G}: |\text{faulty}(\mathcal{F}') \cap G| \leq \Gamma(G)$.

By Lemma 2.3.2, there exists a failure detector history $\mathcal{H}'$ such that $\mathcal{H}' = D(\mathcal{F}')$ for some failure detector $D \in \mathcal{P}$. By Lemma 2.3.4, there exists a run $R'$ in $\mathcal{R}$ with failure pattern $\mathcal{F}'$ and failure detector history $\mathcal{H}'$ that is identical to $R$ until time $t$. Since $\phi$ and $\psi$ only depend on the past, $\phi(R', t)$ holds and $\text{satisfy}(\psi, R', t) = \text{satisfy}(\psi, R, t)$. Therefore, by the definition of $\mathcal{F}'$, and since $G_{\Pi} \in \mathcal{G}$, if $\psi(R, t, p)$ holds for any process $p$ then $p$ is faulty. Since $R'$ is in $\mathcal{R}$ and uses a failure detector in $\mathcal{P}$, $R'$ satisfies $P$; therefore, since $\phi(R', t)$ holds, there exists a correct process $p$ such that $\psi(R', t, p)$ holds. This is a contradiction. $\Box$
Chapter 3

Replicated Shared Objects

3.1 Problem Definition

A replicated shared object system allows processes in a message passing system to communicate using typed shared objects. The set of objects that is provided by a replicated shared object system is denoted by $X$. For reasons of fault-tolerance, these objects are replicated. The processes that are used to maintain the replicas of objects are called server processes. A client process is any process that uses the shared objects to communicate. Every process in $\Pi$ belongs to either the subset $\Pi_C$ of client processes or the subset $\Pi_S$ of server processes. We note, however, that $\Pi_C \cap \Pi_S$ is not necessarily empty; a process may be a client as well as a server.

3.1.1 Shared Objects

Each object in the system is defined by its type $T$ and its initial state $v_{init}$. An object type defines a set of possible values for the object's internal state and a set object methods that can be invoked by client processes. Formally, an object type $T$ is a tuple $(V, F, RLT, Q)$ where $V$ is a set of possible internal states, $F$ is a set of methods, $RLT$ is a set of result values and $Q$ is the sequential specification of $T$ (see below).
Given an object \( x \) of type \( T = (V, F, RLT, Q) \), the only means by which the state of \( x \) can be changed is through the invocation of a method in \( F \) by a client. Each method invocation returns a result value from \( RLT \) to the invoking client.

Given an object type \( T = (V, F, RLT, Q) \), the sequential specification \( Q \) of \( T \), is defined as a directed finite or infinite multi-graph in which each vertex uniquely corresponds to a state in \( V \). The vertex in \( Q \) corresponding to state \( v \) is denoted by \( Q(v) \). Each edge in \( Q \) is labeled with a pair of the form \((f, rlt)\), where \( f \in F \), and \( rlt \in RLT \). Given any two states \( v, v' \in V \), an edge labeled \((f, rlt)\) from \( Q(v) \) to \( Q(v') \) indicates that if method \( f \) is applied to an object in state \( v \), then the operation returns \( rlt \) and leaves the object in state \( v' \).

An object type \( T \) is deterministic if for all states \( v \in V \), and for all methods \( f \in F \) there is at most one edge from \( Q(v) \) labeled \((f, rlt)\) for some \( rlt \in RLT \). A type \( T \) is total if for all states \( v \in V \) and for all methods \( f \in F \) there is at least one edge from \( Q(v) \) labeled \((f, rlt)\) for some \( rlt \in RLT \). All object types considered in this thesis are assumed to be deterministic and total.

In practice, object methods are often parameterized. For instance, in the case of a First-In-First-Out (FIFO) queue two methods would be defined: \( \text{enqueue} \) and \( \text{dequeue} \). The \( \text{enqueue} \) method would take a single argument: an item to be placed at the tail of the queue. Thus the result of invoking \( \text{enqueue} \) with an argument \( a \) would have a different effect than invoking \( \text{enqueue} \) with an argument \( b \). In our model, however, methods take no arguments. Instead, each "parameterized method" is represented by a set of methods – one for every possible argument combination that the method can accept. Therefore, if \( T = (V, F, Q) \) describes a FIFO queue object type, then the invocation of \( \text{enqueue} \) with argument \( a \), and the invocation of \( \text{enqueue} \) with argument \( b \) would constitute the invocation of two distinct methods in \( F \).
3.1.2 Operations

The execution of an object method is called an operation. An operation is invoked by a client process running an application. When an operation is completed, the operation's result is returned to the process that invoked the operation. The occurrence of an operation in any run is modeled by two events, an invocation event and a response event, which mark the beginning and the end of an operation respectively. Invocation events and response events are local events since they only affect the local state of the process that executed them.

Every invocation event is of the form \((c, inv, (x, f))\). As is the case with any event, \(c\) denotes the client process that is executing the invocation event and \(inv\) indicates the event type. In this case, \(inv\) denotes that the event is an invocation event. The event argument is a pair \((x, f)\), where \(x\) is the object that is the target of the operation, and \(f\) is the object method that is being executed. Every response event is of the form \((c, res, (x, rlt))\). Here, \(c\) is the client process executing the response event, \(res\) denotes that the event is a response event, \(x\) is the object delivering the response and \(rlt\) is the result being returned to \(c\).

A response event \(e_r = (p, res, (x, rlt))\) and an invocation event \(e_i = (p', inv, (x', f))\) that occur in a run \(R\) are said to match in \(R\) if \(e_i\) is the most recent invocation preceding \(e_r\) in \(R\) such that \(p = p'\) and \(x = x'\).

A complete operation in a run \(R\) is composed of a matching invocation event and response event in the schedule of \(R\). An incomplete operation in a run \(R\) is an invocation event in the schedule of \(R\) for which there is no matching response event. Given an operation \(op\) in a run \(R\), \(INV(op)\) is used to denote the operation's invocation event in \(R\), and, if the operation is complete, \(RES(op)\) is used to denote the operation's response event in \(R\). The applications executed by the client processes are assumed to be sequential; once a process has invoked an operation, it may not invoke another operation until it has executed a matching response to its initial invocation.
3.1.3 Histories

A history is a finite sequence of invocation and response events. The events in a history are ordered by the times at which they occur. A history is used to represent the operations that occur in a run. A history \( H \) respects a run \( R = \langle \mathcal{F}, \mathcal{H}, I, S, T \rangle \) if \( H \) is the subsequence of all invocation and response events in \( S \). Matching events, complete operations and incomplete operations are defined for histories in the same way that they are defined for runs. Likewise, given an operation \( op \) in a history \( H \), we write \( INV(op) \) to denote the operation's invocation event in \( H \), and write \( RES(op) \) to denote the operation's response event in \( H \).

Given a history \( H \), \( complete(H) \) is the maximal subsequence of \( H \) consisting only of invocations and matching responses. A history \( H \) is sequential if the first event in \( H \) is an invocation, and each invocation in \( H \) is immediately followed by a matching response.

A process subhistory \( H|p \) of a history \( H \) is the subsequence of all events in \( H \) that are executed by process \( p \). Since the applications executed by client processes are sequential, all process subhistories are sequential. Two histories, \( H \) and \( H' \), are said to be equivalent if for every process \( p \), \( H|p = H'|p \).

An object subhistory \( H|x \) of a history \( H \) and object \( x \) is the subsequence of all invocation and response events in \( H \) that occur to object \( x \). An object subhistory \( S|x \) of a sequential history \( S \) is legal if the methods and results of the operations in \( S|x \) form a sequence \((f_1, rlt_1), (f_2, rlt_2), \ldots, (f_i, rlt_i)\) that is a path in \( Q \) from the vertex \( Q(v) \) where \( x \) is of type \( T \), \( Q \) is the sequential specification of \( T \), and \( v \) is the initial state of \( x \). A sequential history \( S \) is legal if all object subhistories in \( S \) are legal.

Given a history \( H \), we can define an irreflexive partial order \( \prec_H \) on the operations in \( H \) as follows:

\[
op_0 \prec_H \op_1 \text{ if } RES(\op_0) \text{ precedes } INV(\op_1) \text{ in } H.\]

A history \( H \) is linearizable [30] if it can be extended to some history \( H' \) such that:
3. Replicated Shared Objects

Since processes do not share physical memory, they cannot share a single copy of an object. Instead, the shared objects in the system are replicated. Given any object \( x \), there is a non-empty set of server processes that forms the object group of \( x \). This set is denoted by \( \text{objGroup}(x) \). Each process in \( \text{objGroup}(x) \) has a copy, or replica, of \( x \) in its memory. An object replica is local to a process in that an object replica’s methods can only be invoked by a single process. This process is called the owner of the replica. Given an object \( x \) and a process \( p \in \text{objGroup}(x) \) the replica of \( x \) that is owned by \( p \) is denoted by \( x_p \).

Given an object \( x \) of type \( T = (V, F, RLT, Q) \) with initial state \( v \), every replica \( x_p \) of \( x \) is of type \( T \) and has an initial state of \( v \). Since each replica can only be accessed by a single process, the execution of a replica’s method is considered to be atomic and is modeled in a run by a single event called a replica operation event. Each replica operation event is of the form \( \langle p, \text{rep}, (x, f, rlt) \rangle \), where \( p \) denotes the process executing the event, and \( \text{rep} \) denotes that the event is a replica operation event. The event argument \( (x, f, rlt) \) denotes that \( p \) executed method \( f \) on its replica \( x_p \) of \( x \) and was returned the result \( rlt \).

The sequence of replica operation events executed by a process \( p \) on an object \( x \) in a run \( R \) is always legal. Formally, if the subsequence of all replica operation events in \( R \) that are executed by \( p \) on \( x \) is \( \langle p, \text{rep}, (x, f_1, rlt_1) \rangle, \langle p, \text{rep}, (x, f_2, rlt_2) \rangle, \ldots, \langle p, \text{rep}, (x, f_n, rlt_n) \rangle \), then the sequence \( (f_1, rlt_1), (f_2, rlt_2), \ldots, (f_n, rlt_n) \) forms a path from \( Q(v) \) through \( Q \), where \( x \) is of type \( T = (V, F, RLT, Q) \) and has the initial state \( v \).
3.1.5 Consistency Protocols

The multiple replicas of the shared objects are kept consistent by an algorithm that is run by the client and server processes. This algorithm is called a Consistency Protocol. A Consistency Protocol $A$ is an algorithm that uses message passing to implement some set of shared objects $X$.

We say that a client process $c$ invokes an operation $op$ on an object $x$ at time $t$ in a run $R$ if $c$ executes the invocation event $INV(op)$ at time $t$ in $R$, and $INV(op) = \langle c, \text{inv}, (x, -) \rangle$. A process $c$ returns from an operation $op$ on an object $x$ in a run $R$ at time $t$ (with a result $rlt$) if $c$ executes the response event $RES(op)$ at time $t$ in $R$, and $RES(op) = \langle c, \text{res}, (x, rlt) \rangle$. Formally, we say that an algorithm $A$ is a Consistency Protocol for a set of shared objects $X$ if every run $R$ of $A$ satisfies the following properties:

**Termination** If a correct client process $c$ invokes an operation $op$ on any object $x \in X$, then $c$ eventually returns from $op$.

**Uniform Integrity** There is exactly one matching invocation for every response of an operation on any object $x \in X$, and every invocation of an operation on any object $x \in X$ is matched by at most one response.

The term "Uniform" is used to indicate that a property does not only place constraints on the behavior of correct processes, but on the behavior of faulty processes as well. The Termination property is not uniform since it only requires that operations executed by correct processes eventually terminate. The Uniform Integrity property, however, applies to all operations, even those executed by faulty processes. The use of this term was introduced by Neiger and Toueg [43].

In addition to satisfying Termination and Uniform Integrity, a Consistency Protocol must also satisfy some consistency property. A consistency property affords an application that uses the implemented objects with some guarantees about the results returned by each operation, and to some degree, conceals the fact that the objects are replicated.
The consistency property that we will be considering in this thesis is linearizability.

A consistency protocol $A$ is a linearizable if, given any run $R$ of $A$, any history $H$ that respects $R$ is linearizable. Unless otherwise specified, every consistency protocol that we will consider in this thesis is linearizable.

A consistency property $\phi$ is local if a history $H$ satisfies $\phi$ whenever each individual object subhistory of $H$ satisfies $\phi$. Herlihy and Wing showed that linearizability is a local property [30]:

**Theorem 3.1.1** A history $H$ is linearizable if and only if, for each object $x$, $H|x$ is linearizable.

Therefore, we can show that an algorithm $A$ is a linearizable consistency protocol for a set of objects $X$ by showing that $A$ is a linearizable consistency protocol for every object $x \in X$.

### 3.2 Multicast Communication

In designing algorithms for distributed systems, it is often useful to address a single message to multiple processes. A multicast is a form of communication that allows messages to be addressed to a process group rather than to a single process. We will now briefly address the issue of fault-tolerant multicast communication since multicasts are used by the shared object consistency protocol defined in the next section.

In the literature, the term broadcast is sometimes used interchangeably with the term multicast; however, we will use the term broadcast to refer to a restricted form of multicast in which messages may only be addressed to the system group $G_H$. A multicast can be categorized as *internal* or *external*. If processes are communicating using an internal multicast then a process must be a member of a group $G$ in order to address a message to $G$. External multicast allows any process in the system to address a message to any group. We will only be considering external multicast in this thesis.
For a thorough examination of fault-tolerant broadcasts and multicasts, the reader is directed to Hadzilacos and Toueg [24].

The Multicast problem has many variations, each of which is defined by a slightly different set of properties. Every form of multicast is defined in terms of two primitives: `multicast` and `deliver`. These primitives act on messages from the set of valid messages $M$. Each message $m \in M$ that is multicast is addressed to a particular group, and carries the name of this group in a field denoted by $group(m)$. If a process $p$ invokes `multicast` with message $m$ as a parameter, we say that $p$ multicasts $m$ (to $group(m)$). When a process returns from the execution of `deliver` with message $m$ as a returned value, we say that $p$ delivers $m$.

### 3.2.1 Reliable Multicasts

Roughly speaking, a Reliable Multicast ensures that either all of the correct processes in the addressed group receive a sent message or none of them do. Formally, Reliable Multicast satisfies the following properties [24]:

**Validity** If a correct process multicasts a message $m$, then some correct process in $group(m)$ eventually delivers $m$, or no process in that group is correct.

**Uniform Agreement** If a process delivers a message $m$, then all correct processes in $group(m)$ eventually deliver $m$.

**Uniform Integrity** For any message $m$, every process $p$ delivers $m$ at most once, and only if $p$ is in $group(m)$ and $m$ was previously multicast by $sender(m)$.

A Reliable Broadcast is a Reliable Multicast that only allows messages to be addressed to the system group $G_N$.

We will also consider a weaker form of Reliable Multicast, called Weak Reliable Multicast, which ensures that either all of the processes in $group(m)$ receive a sent message
m or none of them do only if m was multicast by a correct process. Formally, Weak Reliable Multicast satisfies Validity, Uniform Integrity, and a weaker form of Uniform Agreement:

**Weak Uniform Agreement** If a process delivers a message m that was multicast by a correct process, then all correct processes in group(m) eventually deliver m.

For clarity, we will refer to the multicast and deliver primitives of Reliable Multicast as \textit{R-multicast} and \textit{R-deliver} respectively, and we will refer to the multicast and deliver primitives of Weak Reliable Multicast as \textit{WR-multicast} and \textit{WR-deliver} respectively.

### 3.2.2 Atomic Multicast

Even though Reliable Multicast requires that correct processes in a group deliver the same set of messages addressed to that group, it places no restrictions on the order in which messages are delivered. An Atomic Multicast requires that all processes in a group deliver messages in the same order. Various types of Atomic Multicast have been described that differ by the strength of their message delivery order requirements [24, 29].

**Local Atomic Multicast**

The form of Atomic Multicast that we will consider is \textit{Local Atomic Multicast (or LA-Multicast)}, which requires that all processes in a group deliver all messages that are multicast to that group in the same order. Local Atomic Multicast is usually defined as a Reliable Multicast that also satisfies the following property:

**Local Uniform Total Order** If processes \( p \) and \( q \) both deliver messages \( m \) and \( m' \) and 

\[
group(m) = \group(m') \text{ then } p \text{ delivers } m \text{ before } m' \text{ if and only if } q \text{ delivers } m \text{ before } m'.
\]

Stronger forms of Atomic Multicast are possible that order messages addressed to different groups. We will consider these multicasts in our discussion in Chapter 6; how-
ever, we will focus on Local Atomic Multicast in this thesis. In the interest of brevity, we will refer to Local Atomic Multicast simply as “Atomic Multicast” (or A-Multicast).

An *Atomic Broadcast* is a restricted form of Atomic Multicast in which messages can only be addressed to the system group $G_R$. For clarity, we will refer to the multicast and delivery primitives of Atomic Multicast as $A$-multicast and $A$-deliver respectively.

**Delivery Consistency**

Defining Atomic Multicast using the Local Uniform Total Order property introduces the problems of *inconsistency* and *contamination* [23, 24]. Consider the run of an Atomic Multicast algorithm depicted in Figure 3.1 in which messages $m_0$ and $m_1$ are multicast to a group $G = \{p, q, r\}$ by processes $c_0$ and $c_1$ respectively. If $p$ is faulty, and $q$ and $r$ are correct in this run, then the properties of Atomic Multicast allow $p$ to deliver $m_1$ (and later crash) without ever delivering $m_0$, even if both $q$ and $r$ deliver $m_0$ before $m_1$. If the state of a process in $G$ depends on the sequence of messages that that process has delivered (as is the case with active replication), then process $p$ reaches an “inconsistent” state after delivering $m$. Furthermore, if $p$ sends (or multicasts) a message to another process $p'$ while in this inconsistent state, it might “contaminate” $p'$ with faulty information, thereby forcing $p'$ into an inconsistent state. Our approach to the resolution of this issue is to require that Atomic Multicast does not allow inconsistency to result from the delivery of messages. We will therefore redefine Atomic Multicast so as to preclude the possibility of delivery inconsistency.

The sequence of messages delivered by a process $p$ up to time $t$ is called the *delivery sequence of process $p$ at time $t* and is written as $DLVD_p^t$. Given any delivery sequence $DLVD_p^t$, the subsequence of messages in $DLVD_p^t$ that are addressed to a particular group $G$ is written as $DLVD_p^t|G$. We shall define an Atomic Multicast as a Reliable Multicast that satisfies the following property:

**Local Delivery Consistency** If two processes $p$ and $q$ are in the same group $G$, then
at any time $t$, either $DLVD_p^t|G \preceq DLVD_q^t|G$, or $DLVD_q^t|G \preceq DLVD_p^t|G$.

This definition of delivery consistency differs from the one originally given by Gopal [24]. Gopal does show, however, that an Atomic Broadcast algorithm (and subsequently a Atomic Multicast algorithm) that satisfies the property given above is delivery consistent by his definition. Even though our definition is stronger than Gopal's, our version of the property is satisfied by the Atomic Multicast implementation that we will present in Chapter 5.

### 3.2.3 A Real-Time Ordering of Messages

Even though Atomic Multicast places a total order on the messages delivered by a process in a group, no explicit restrictions are made on how this ordering of messages is determined. However, the use of Atomic Multicast in a linearizable implementation of shared objects based on active replication requires that some \textit{real-time} constraints are placed on the ordering of messages.

For example, consider the run of a consistency protocol depicted in Figure 3.2. This protocol is based on active replication using Atomic Multicast; however, we will assume, for the purposes of this example, that the Atomic Multicast algorithm used does not satisfy Local Delivery Consistency. Two operations $op_0$ and $op_1$ are executed by client processes $c_0$ and $c_1$ respectively on an object $x$ such that $obj\text{Group}(x) = \{p, q, r\}$. Mess-
messages $m_0$ and $m_1$ are the request messages for $op_0$ and $op_1$ respectively. These messages are A-multicast to $objGroup(x)$ and have a similar delivery pattern as the one depicted in Figure 3.1. Processes $q$ and $r$ A-deliver $m_0$, perform operation $op_0$ on their replicas, and return reply messages to $c_0$ before $c_1$ invokes $op_1$. Thus, $RES(op_1)$ precedes $INV(op_2)$ in the history that respects this run. However, since process $p$ delivers $m_1$ without first delivering $m_0$, it may return a reply to $c_1$ that does not allow $op_0$ to be ordered before $op_1$ in any sequential ordering of operations. Thus, the history that respects this run is not linearizable.

This problem could be avoided if no process ever delivered a message $m$ without first delivering every message $m'$ that preceded $m$, where a message $m$ is preceded by a message $m'$, if $m'$ was delivered by any process before $m$ was multicast. More formally, we would like every run of our Atomic Multicast algorithm to satisfy the following property:

**Linear Order** If a process $p$ delivers a message $m$ that was multicast at time $t$, then $p$ has previously delivered every message $m'$ such that $\text{group}(m') = \text{group}(m)$ and $m'$ was delivered by some process in $\text{group}(m')$ before $t$. 
Let $m$ be a message that is A-multicast to a group $G$ at time $t$ by a process $p$. We say that $m$ is A-delivered in linear order by process $q$ if and only if $q$ has previously delivered every message $m'$ such that $\text{group}(m') = \text{group}(m)$ and $m'$ was A-delivered by some process in $\text{group}(m')$ before $t$. Thus, a run $R$ of any Atomic Multicast algorithm satisfies the Linear Order property if and only if every message that is A-delivered in $R$ is A-delivered in linear order. If a process $q$ A-delivers $m$ but does not A-deliver $m$ in linear order, then we say that $q$ A-delivers $m$ out of linear order.

In the example in Figure 3.2, message $m_1$ could not have been A-delivered out of linear order if the Atomic Multicast algorithm satisfied Local Delivery Consistency. We have already stated that in this thesis, we will define Atomic Multicast in terms of Local Delivery Consistency property instead of the weaker Local Total Order property. We will now show that by this definition, Atomic Multicast satisfies Linear Order.

**Lemma 3.2.1** Every run that satisfies the properties of Atomic Multicast also satisfies Linear Order.

**Proof:** Let $R$ be a run that satisfies the properties of Atomic Multicast, and let $m$ be a message that was A-multicast to a group $G$ at time $t$ in $R$. Assume, by way of contradiction, that a process $q \in G$ A-delivered some message $m'$ at time $t' \leq t$, where $\text{group}(m) = G$, and another process $p \in G$ A-delivered $m$ without previously delivering $m'$. Let $t_m$ be the time at which $p$ A-delivered $m$. By Local Delivery Consistency, either $DLVD_{p}^{m}|G \preceq DLVD_{q}^{m}|G$ or $DLVD_{q}^{m}|G \preceq DLVD_{p}^{m}|G$. If $DLVD_{p}^{m}|G \preceq DLVD_{q}^{m}|G$, then $m$ precedes $m'$ in $DLVD_{q}^{m}|G$ since $DLVD_{q}^{m}|G$ contains $m'$ and $m$ is the last element of $DLVD_{p}^{m}|G$. This means that $q$ A-delivered $m$ before $t'$ which violates Uniform Integrity since $m$ was A-multicast after $t'$. If $DLVD_{q}^{m}|G \preceq DLVD_{p}^{m}|G$, then $p$ must have delivered $m'$ by $t_m$, which is a contradiction. \(\square\)
3.3 A Simple Linearizable Consistency Protocol

3.3.1 Description

The algorithm in Figure 3.3 is a simple linearizable consistency protocol that is based on active replication using Atomic Multicast. Even though this algorithm is well known, it will be formally presented here in order to simplify the discussion and proof of a variation of the algorithm in Chapter 4.

Before discussing this algorithm, we will briefly discuss the format in which we will present algorithms in this thesis. Even though we formally model algorithms as a set of automata, we will specify algorithms rather informally using pseudo-code; however, these specifications are rigorous enough to allow the reader to translate them to more formal specifications, such as those used in the I/O automaton model [40], if required.

The algorithms presented in this thesis are written in terms of *tasks*—small "subroutines" that are executed when certain conditions are satisfied. Tasks should not be confused with processes; tasks are executed by the processes in the system. A process may execute multiple tasks concurrently. For example, the algorithm in Figure 3.3 consists of two tasks: *Server Task 1* and *Client Task 1*. Client Task 1 is executed by a client process whenever that process calls the `apply` function (this function is described below). Server Task 1 is executed by a server process whenever that process A-delivers a message of the form \((c, req, x, f)\). If a process is both a client and a server then it may execute both Client Task 1 and Server Task 1 concurrently. Unless otherwise noted, a task may be executed concurrently with any task other than itself.

Initialization code is also specified that must be executed by each process before it executes any task. Figure 3.3 contains two blocks of initialization code, one to be executed by client processes and one to be executed by server processes. A process that is both a server and a client executes both blocks of initialization code in any order.

We also note that throughout this thesis, we will write \(v_p\) to denote a process \(p\)'s copy
Every client process \( c \) executes the following:
Initialization:
\[
\text{req} \leftarrow 0
\]
To execute \( \text{apply}(x, f) \):
\[
\begin{align*}
\text{req} & \leftarrow \text{req} + 1 \\
\text{A-multicast}(c, \text{req}, x, f) & \text{ to } \text{objGroup}(\text{obj}) \\
\text{wait until} \ [\text{for some server process } p: \text{ received } (\text{req}, \text{res}) \text{ from } p] \\
\text{return } \text{res}
\end{align*}
\]
Every server process \( p \) executes the following:
Initialization:
\[
\text{for every object } z \text{ such that } p \in \text{objGroup}(z): \]
\[
\text{initializeReplica}(z) \\
\text{upd}[z] \leftarrow 0
\]
\[
\text{when } \text{A-deliver}(c, \text{req}, x, f): \\
\text{upd}[z] \leftarrow \text{upd}[z] + 1 \\
\text{res} \leftarrow \text{applyToReplica}(x, f) \\
\text{send } (\text{req}, \text{res}) \text{ to } c
\]

Figure 3.3: A general algorithm for implementing linearizable replicated objects using Atomic Multicast.

of a variable \( v \) used by an algorithm.

We now return our attention to the algorithm in Figure 3.3. The \( \text{apply} \) function is called whenever an application executed by a client process invokes an operation on an object. The function arguments, \( x \) and \( f \) denote the target object and the method to be invoked, respectively. The operation is complete when the client process returns from \( \text{apply} \) with the result of the operation.

Upon calling \( \text{apply}(x, f) \), the client process \( \text{A-multicast} \)s a request message to all processes in \( \text{objGroup}(x) \). The request message contains the client process’ id, \( x, f \), and a request id that is used to match replies to requests.

Each server process for an object initializes its own replica of the object to the object’s initial state. This is accomplished by calling the \text{initializeReplica} function. When a process \( p \) calls this function is called with an argument \( x \), the replica object \( x_p \) is set to
its initial state.

When a server process receives a request message, it performs the requested method on its own replica of the object. This replica operation is performed by a call to the `applyToReplica` function. If a process \( p \) calls `applyToReplica` with arguments \( x \) and \( f \), then the method \( f \) is executed on the replica object \( x_p \). The function returns the result of the replica operation that it performs. This result is then returned to the client in a reply message, along with the provided request id. Once the client has received a single reply that matches its request, it can return the result contained in the reply to the application.

Each server process \( p \) also maintains an update counter variable \( upd[x] \) for every object \( x \) such that \( p \in objGroup(x) \). This counter is initialized to \( 0 \) and is incremented by \( 1 \) before each replica operation is performed on \( x \). This variable is not necessary to the algorithm; it is an "auxiliary variable" — a device to simplify the algorithm’s proof of correctness. The use of auxiliary variables as a technique for proving properties of parallel programs was introduced by Owicki and Gries [44].

A detailed proof of the correctness of this algorithm can be found in Subsection 3.3.3. Intuitively, since all requests are sent to the servers using Atomic Multicast, all servers perform the same set replica operations in the same order. Furthermore, since Atomic Multicast satisfies Linear Order, any server that performs a replica operation for an operation \( op \) has already performed the replica operations for every operation \( op' \) that completed before \( op \) was invoked. Thus, the objects that are implemented by this algorithm are linearizable.

### 3.3.2 Fault-Tolerance

In order to ensure Termination, the algorithm requires that for each object \( x \in X \), no more than \( f < |objGroup(x)| \) processes fail in \( objGroup(x) \). The fault-tolerance of this algorithm is also limited by the fault-tolerance of the Atomic Multicast algorithm that it
employs. If we let $\Gamma'$ be the fault-tolerance function of the Atomic Multicast algorithm, then the algorithm in Figure 3.3 is a $\Gamma$-tolerant consistency protocol where:

$$
\forall G \in \mathcal{G} : \Gamma(G) = \begin{cases} 
\min(|G| - 1, \Gamma'(G)) & \text{if } G = \text{objGroup}(x) \text{ for some object } x \in X \\
\Gamma'(G) & \text{otherwise}
\end{cases}
$$

(3.1)

Defining the fault-tolerance function in this way, however, fails to capture the "modularity" of the algorithm. Specifically, for any two objects $x$ and $x'$, if all of the processes in $\text{objGroup}(x')$ fail and all of the processes in $\text{objGroup}(x)$ are correct, object $x$ will behave correctly: all invocations to $x$ will eventually terminate, invocation events and response events for operations on $x$ will match correctly and the object subhistory for $x$ will be linearizable.

By our definition of $\Gamma$, a run does not satisfy the properties of a linearizable consistency protocol for $X$ if all of the processes maintaining a particular object fail. However, as we have mentioned earlier, Herlihy and Wing have shown that an algorithm is a linearizable consistency protocol for a set of objects $X$ if it is a linearizable consistency protocol for every object in $X$. Thus, instead of being a fault-tolerant solution to a single problem, our algorithm is a fault-tolerant solution to a set of problems. It therefore makes sense to define a separate tolerance function for every problem in this set. Specifically, we will define a tolerance function $\Gamma_x$ for every object $x \in X$ where the algorithm in Figure 3.3 is a $\Gamma_x$-tolerant linearizable consistency protocol for $x$.

As we have already stated, the tolerance function of the algorithm in Figure 3.3 depends on the tolerance function of the Atomic Multicast algorithm that it uses. However, the fault-tolerance of the implementation of a particular object $x$, only depends on the ability of the Atomic Multicast algorithm to reliably A-deliver messages addressed to $\text{objGroup}(x)$. Therefore, in order to define a tolerance function for each object $x$, we must redefine the Atomic Multicast problem as a set of problems, one for each group
Given any group \( G \in \mathcal{G} \), we can define Atomic Multicast for \( G \) by restricting the Validity, Uniform Integrity and Uniform Agreement properties to messages addressed to \( G \), and by restricting the Local Delivery Consistency property to processes in \( G \). For example, Validity becomes: \text{If a correct process multicasts a message } m, \text{ where } \text{group}(m) = G, \text{ then some correct process in } G \text{ eventually delivers } m, \text{ or no process in } G \text{ is correct;} \text{ and Local Delivery Consistency becomes: If two processes } p \text{ and } q \text{ are in } G, \text{ then at any time } t, \text{ either } DLVD^t_p(G) \leq DLVD^t_q(G) \text{, or } DLVD^t_q(G) \leq DLVD^t_p(G).\)

We will show in Chapter 5 that Atomic Multicast algorithms exist that are \( \Gamma_G \)-tolerant solutions to Atomic Multicast for \( G \) for all \( G \in \mathcal{G} \) where for some \( f < |G| \):

\[
\forall G' \in \mathcal{G} : \Gamma_G'(G') = \begin{cases} f & \text{if } G' = G \\ |G'| & \text{otherwise} \end{cases} \tag{3.2}
\]

If the consistency protocol in Figure 3.3 uses such an Atomic Multicast algorithm, then it is a \( \Gamma_x \)-tolerant linearizable consistency protocol for any object \( x \) where:

\[
\forall G \in \mathcal{G} : \Gamma_x(G) = \begin{cases} \min(|G| - 1, \Gamma_G(G)) & \text{if } G = \text{objGroup}(x) \\ |G| & \text{otherwise} \end{cases} \tag{3.3}
\]

### 3.3.3 Proof of Correctness

Let \( x \in X \) be any object, let \( A \) be the algorithm in Figure 3.3 and let \( \Gamma_x \) be the tolerance function defined in Equation 3.3. In this subsection, we will show that \( A \) is a \( \Gamma_x \)-tolerant linearizable consistency protocol for \( x \). It then follows, as discussed in the previous subsection, that \( A \) is a \( \Gamma \)-tolerant linearizable consistency protocol where \( \Gamma \) is the tolerance function defined in Equation 3.1.

Let \( R = \langle \mathcal{F}, \mathcal{H}, I, S, T \rangle \) be any run in the system formed by \( A \) and \( \Gamma_x \). By the definition of \( \Gamma_x \), we can assume that there is at least one correct process in \( \text{objGroup}(x) \).
Calls to apply are modeled by invocation events and returns from apply are modeled by response events. A client \( c \) calls apply with arguments \( x \) and \( f \) at time \( t \) in \( R \) if and only if \( c \) executes an invocation event \( (c, \text{inv}, (x, f)) \) at time \( t \) in \( R \). Similarly, a client \( c \) returns from a call to apply with a result \( \text{rlt} \) if and only if \( c \) executes a response event \( (c, \text{res}, (x, \text{rlt})) \) at time \( t \) in \( R \), where \( x \) was the first argument to the call to apply.

Calls to applyToReplica are modeled by replica operation events. Note that, in contrast to calls to apply, calls to applyToReplica are modeled by a single event since they are considered to be atomic. A server \( p \) calls applyToReplica with arguments \( x \) and \( f \) and is returned \( \text{rlt} \) at time \( t \) in \( R \) if and only if \( p \) executes a replica operation event \( (p, \text{rep}, (x, f, \text{rlt})) \) at time \( t \) in \( R \).

Given an operation \( op \) invoked by a client \( c \) in \( R \), the replica operation event executed by \( p \) for \( op \) is the first replica operation executed by \( p \) after delivering the request message sent by \( c \) after executing \( \text{INV}(op) \). The replica operation event executed by \( p \) for \( op \) is written as \( \text{REP}_p(op) \). Throughout this proof, we will write \( e_0 <_R e_1 \) to denote that an event \( e_1 \) precedes an event \( e_2 \) in the schedule \( S \) of \( R \).

**Lemma 3.3.1 (Termination)** If a correct client process \( c \) executes an invocation on \( x \) then \( c \) eventually executes a matching response.

**Proof:** Let \( c \) be a correct client process that calls apply with arguments \( x \) and \( f \). By the algorithm, the only place where \( c \) may wait indefinitely before returning is when it waits until a reply message is received from a server process. Before waiting, \( c \) multicasts a request to all servers in \( \text{objGroup}(x) \). Since \( c \) is correct, and at least one server in \( \text{objGroup}(x) \) is correct, at least one correct server \( p \) in \( \text{objGroup}(x) \) receives the request by the Validity property of Atomic Multicast. By the algorithm, \( p \) will eventually send a reply to \( c \), and the call to apply will eventually return. \( \Box \)

**Lemma 3.3.2 (Uniform Integrity)** Every response to an operation on \( x \) matches exactly one invocation, and every invocation of an operation on \( x \) is matched by at most one
response.

**Proof:** Immediate from the algorithm. □

**Lemma 3.3.3** Given any complete operation $op$ on $x$ in $R$, there exists at least one replica operation $REP_p(op)$ executed by some process $p$ in $R$ such that $REP_p(op) < R \ RES(op)$.

**Proof:** By the algorithm, when a client process $c$ calls $apply(x,f)$ it A-multicasts a request message $m = (c, req, x, f)$ to all of the server processes in $objGroup(x)$, where $req$ is a request id used by $c$ to match replies with requests. $c$ then waits until it receives a matching reply message before returning. By the algorithm, a server process in $objGroup(x)$ will not send a reply message to $c$ with a request id $req$ until it has A-delivered a request message $m = (c, req, x, f)$ and called $applyToReplica(x,f)$. Thus, no operation $op$ can return before at least one replica operation $REP_p(op)$ has been executed by some server process $p$. □

**Lemma 3.3.4** Given any (complete or incomplete) operation $op$ on $x$ in $R$ and any replica operation $REP_p(op)$ performed by some process $p$ in $R$, $INV(op) < R \ REP_p(op)$.

**Proof:** By the algorithm, a replica operation $REP_p(op)$ is performed after a server process $p$ has A-delivered a request message $m$. By the Uniform Integrity property of A-Multicast, $m$ must have been multicast before it was delivered. Since request messages are multicast by client processes after an operation is invoked, $INV(op) < R \ REP_p(op)$. □

**Lemma 3.3.5** Given any operation $op$ in $R$ on $x$, a server process $p \in objGroup(x)$ executes at most one replica operation $REP_p(op)$ of $op$.

**Proof:** By the algorithm, a single request message $m$ is A-multicasted for any operation $op$. By Uniform Integrity, each server process $p \in objGroup(x)$ A-delivers $m$ at most
once. Since a replica operation $REP_p(op)$ is only performed when a request message for $op$ is delivered, $p$ executes $REP_p(op)$ at most once. □

Lemma 3.3.6 Given any two operations $op$ and $op'$ in $R$ on $x$, if $RES(op') <_R INV(op)$ then any process $p \in obj\text{Group}(x)$ that executes $REP_p(op)$ previously executed $REP_p(op')$.

Proof: Let $p$ be a process that performs $REP_p(op)$. By Lemma 3.3.4, $INV(op) < REP_p(op)$. By Lemma 3.3.3, some process $p'$ executed $REP_{p'}(op')$ before the client process invoking $op'$ executed $RES(op')$. Therefore, since $RES(op') <_R INV(op)$, $REP_{p'}(op') <_R REP_p(op)$. Let $m$ and $m'$ be the request messages for $op$ and $op'$ respectively. By the algorithm, $m$ was A-multicast after $INV(op)$, $p$ A-delivered $m$ before executing $REP_p(op)$ and $p'$ A-delivered $m'$ before executing $REP_{p'}(op')$. Since A-Multicast satisfies Linear Order by Lemma 3.2.1, $p$ A-delivered $m'$ before delivering $m$, and by the algorithm, executed $REP_p(op')$ before executing $REP_p(op)$. □

Lemma 3.3.7 Given an operation $op$ on $x$, if $p \in obj\text{Group}(x)$ executes $REP_p(op)$, and $q \in obj\text{Group}(x)$ executes $REP_q(op)$, then the value of $upd_p[x]$ when $p$ executes $REP_p(op)$ is equal to the value of $upd_q[x]$ when $q$ executes $REP_q(op)$.

Proof: Let $m_{op}$ be the request message for operation $op$, and let $t_p$ and $t_q$ be the times at which $m_{op}$ is A-delivered by $p$ and $q$ respectively. Without loss of generality, assume that $t_q \leq t_p$. By the Local Delivery Consistency property of A-Multicast, $DLVD^p_{t_p}\text{\text{objGroup}(x)} \preceq DLVD^q_{t_q}\text{\text{objGroup}(x)}$, and the number of messages addressed to $obj\text{Group}(x)$ that were A-delivered by $p$ and by $q$ prior to delivering $m_{op}$ are therefore the same. By the algorithm, every server process $p \in obj\text{Group}(x)$ increments its update counter $upd_p[x]$ each time it A-delivers a request message for an update operation. Since the update counter is incremented before the replica operation is performed, the value of $upd_p[x]$ when $p$ executes $REP_p(op)$ is equal to the value of $upd_q[x]$ when $q$ executes $REP_q(op)$. □
Given an operation \( op \) in \( R \), we say that \( op \) is \emph{completable} in \( R \) if and only if some server process executes \( REP_p(op) \) in \( R \). We note that by Lemma 3.3.3, every complete operation is a completable operation.

We can now assign a number to each completable operation in \( R \). Let \( T \) be a function from the set of all completable operations in \( R \) to the set of natural numbers that is defined as follows: given any operation \( op \) on \( x \) in \( R \), \( T(op) \) is the value of the update counter \( upd_p[x] \) of any server process \( p \in \text{objGroup}(x) \) when it executes \( REP_p(op) \). This value must exist and is unique by Lemmata 3.3.3, 3.3.5 and 3.3.7.

**Lemma 3.3.8** Let \( op \) and \( op' \) be any two completable operations on \( x \). If \( op \neq op' \) then \( T(op) \neq T(op') \).

**Proof:** Assume by way of contradiction that \( op \) and \( op' \) are two complete operations on an object \( x \) such that \( op \neq op' \) and \( T(op) = T(op') = i \). Let \( p \in \text{objGroup}(x) \) be any server that executed \( REP_p(op) \), and let \( p' \in \text{objGroup}(x) \) be any server that executed \( REP_{p'}(op') \). Processes \( p \) and \( p' \) must exist by Lemma 3.3.3. Let \( m_{op} \) and \( m_{op'} \) be the request messages that were A-multicast by the client processes invoking \( op \) and \( op' \) respectively. By the algorithm, \( p \) A-delivered \( m_{op} \) prior to executing \( REP_p(op) \), and \( p' \) A-delivered \( m_{op'} \) prior to executing \( REP_p(op') \). Thus, by the definition of \( T \), \( m_{op} \) is the \( i^{th} \) message A-delivered by \( p \) and \( m_{op'} \) is the \( i^{th} \) message A-delivered by \( p' \) which violates the Local Delivery Consistency property of A-Multicast. \( \square \)

**Lemma 3.3.9** Let \( op \) and \( op' \) be any two completable operations on \( x \), and let \( p \in \text{objGroup}(x) \) be any process that executes \( REP_p(op) \). \( T(op') < T(op) \) if and only if \( p \) executes \( REP_p(op') \) before \( REP_p(op) \).

**Proof:** We first prove the "only if" part of the lemma. Suppose that \( T(op') < T(op) \), and let \( t \) be the time at which \( REP_p(op) \) is executed by \( p \). By the definition of \( T \), \( T(op) \) is equal to the value of \( upd_p[x] \) at time \( t \). Since \( upd_p[x] \) is initialized to 0, and is
incremented by 1 every time a message is delivered, \( \text{upd}_p[x] \) was set to \( T(op') \) at some
time \( t' < t \). Since \( \text{upd}_p[x] \) is equal to the value of \( T(op) \) at time \( t \), \( \text{upd}_p[x] \) must have been
incremented at some time after \( t' \). Therefore, by the algorithm, \( p \) must have executed
some replica operation \( REP_p(op'') \) when \( \text{upd}_p[x] \) was set to \( T(op') \). By the definition of
\( T \), \( T(op') = T(op'') \), and so \( op' = op'' \) by Lemma 3.3.8. Thus \( p \) executed \( REP_p(op') \)
before executing \( REP_p(op) \).

We now prove the "if" part of the lemma. Suppose that \( p \) executes \( REP_p(op') \) before
executing \( REP_p(op) \), and let \( t \) be the time at which \( p \) executes \( REP_p(op') \). Assume, by
way of contradiction, that \( T(op) \leq T(op') \). By the definition of \( T \), the value of \( \text{upd}_p[x] \)
is \( T(op') \) at time \( t \). By the algorithm, \( \text{upd}_p[x] \) is never decremented, which implies that
\( T(op) = T(op') \). By Lemma 3.3.8, \( op = op' \), and so by lemma 3.3.5, \( p \) executed \( REP_p(op) \)
and \( REP_p(op') \) at the same time, which is a contradiction. \( \square \)

Let \( H \) be the history that respects \( R \). The history \( \text{extend}(H) \) is a transformation of
\( H \) that is defined as follows:

1. For every incomplete operation \( op \) in \( H \) such that \( INV(op) = \langle c, \text{inv}, (x, f) \rangle \), if a
   replica operation \( REP_p(op) = \langle c, \text{rep}, (x, f, rlt) \rangle \) is executed by any process \( p \) in \( R \)
then a response event \( \langle c, \text{res}, (x, rlt) \rangle \) is appended to \( H \).

2. Response events that are appended to \( H \) appear in the same order as their matching
   invocations.

We note that the particular ordering of appended response events specified above is not
critical to the correctness of our proof; any ordering could have been used. We specify
an arbitrary ordering solely for the convenience of defining \( \text{extend}(H) \) as a single history
rather than a set of histories.

We now define an irreflexive relation \( <_T \) on the operations in \( \text{complete}(\text{extend}(H|x)) \)
using the values assigned to each operation in \( R \) by the function \( T \). However, since
operations are formally defined in terms of the sequence in which they occur, \( R \) technically
contains an entirely different set of operations than \( \text{complete}(\text{extend}(H|x)) \), even though both sets of operations are modeling the same set of method executions. We therefore define a mapping, or correspondence, between operations in a run \( R \) and operations in a history \( H \) that respects \( R \), so that we may identify operations in \( R \) and \( H \) that refer to the same method execution. This mapping should also hold for any object subhistory \( H|x \), and should hold across the transformations \( \text{complete} \) and \( \text{extend} \).

Given any run \( R \), we uniquely identify each operation \( op \) in \( R \) with an identifier \( id_R(op) \) such that \( id_R(op) = (i, p, x) \) if and only if \( INV(op) \) is the \( i \)th event in \( R \) of the form \( (p, \text{inv}, (x, -)) \), i.e., \( op \) is the \( i \)th operation invoked by \( p \) on \( x \) in \( R \). Given any operation \( op \) in a history \( H \), the unique identifier \( id_H(op) \) is similarly defined. We say that an operation \( op \) in a run \( R \) and an operation \( op' \) in a history \( H \) correspond if and only if \( id_R(op) = id_H(op') \).

From the above definition, we can see that if \( H \) is the history that respects a run \( R \), then the corresponding operations in \( R \) and \( H \) model the same method executions, i.e., the correspondence relation is meaningful. This is also true for any object subhistory \( H|x \). In order to properly define \( \leq_{T_x} \), we must show that corresponding operations in \( R \) and \( \text{complete}(\text{extend}(H|x)) \) also model the same method executions.

Since the \( \text{extend} \) transformation does not affect invocations, the unique identifiers used to define correspondence are not affected either. On the other hand, the \( \text{complete} \) transformation does affect invocations. Given any history \( H \), \( \text{complete}(H) \) is identical to \( H \) except that all of the incomplete operations have been removed from \( H \). Recall, however, that process subhistories are sequential. This means that each process subhistory \( H|p \) contains at most one incomplete operation; furthermore, if \( H|p \) does contain an incomplete operation, this operation occurs at the end of \( H|p \). Thus, the only way that the correspondence relation is affected by the \( \text{complete} \) transformation is that some operations in \( R \) will not have a corresponding operation in \( \text{complete}(\text{extend}(H|x)) \).

We now define the relation \( \leq_{T_x} \) on the operations in \( \text{complete}(\text{extend}(H|x)) \). Given
any two operations $op_0$ and $op_1$ in $\text{complete}(\text{extend}(H|x))$, $op_0 <_{T_s} op_1$ if and only if $T(op_0') < T(op_1')$, where $op_0'$ and $op_1'$ are the operations in $R$ corresponding to $op_0$ and $op_1$ respectively.

**Lemma 3.3.10** $<_{T_s}$ is a well-founded, irreflexive total order on the operations in $\text{complete}(\text{extend}(H|x))$.

**Proof:** Follows from the definition of $T$, and Lemma 3.3.8. $\Box$

For the remainder of this proof we will not distinguish the operations in $R$ from their corresponding operations in $\text{complete}(\text{extend}(H|x))$ in the interest of brevity. Specifically, given an operation $op$ in $R$ and an operation $op'$ in $\text{complete}(\text{extend}(H|x))$, if $op$ and $op'$ correspond, then we will refer to $op'$ simply as $op$ if it is obvious from the context that we are referring to the operation in $\text{complete}(\text{extend}(H|x))$ that corresponds to $op$. Likewise we may refer to $op$ as $op'$ if it is obvious from the context that we are referring to the operation in $R$ that corresponds to $op'$.

**Lemma 3.3.11** Given an operation $op$ on $x$, let $op_0, op_1, \ldots, op_n$ be the sequence of operations that precede $op$ in $<_{T_s}$. If a process $p \in \text{objGroup}(x)$ executes a replica operation $\text{REP}_p(op)$, then the sequence of replica operations that $p$ has executed before executing $\text{REP}_p(op)$ is $\text{REP}_p(op_0), \text{REP}_p(op_1), \ldots, \text{REP}_p(op_n)$.

**Proof:** By Lemmata 3.3.5, 3.3.8 and 3.3.9, every server executes replica operations in the order given by $T$. The result follows from the construction of $<_{T_s}$. $\Box$

**Lemma 3.3.12** $H|x$ is linearizable.

**Proof:** Since $<_{T_s}$ is a well-founded total order by Lemma 3.3.10, we can order the operations in $\text{complete}(\text{extend}(H|x))$ in the order given by $<_{T_s}$ to yield a sequential history $S$. Thus $<_{S} <_{T_s}$. 
We now show that $\prec_S$ preserves the real-time order of operations. Let $\text{op}$ and $\text{op}'$ be any two operations in $\text{complete}(\text{extend}(H|x))$ such that $\text{op} \prec_H \text{op}'$. This implies that $\text{RES}(\text{op}) \prec_R \text{INV}(\text{op}')$. Now assume, by way of contradiction, that $\text{op}' \prec_S \text{op}$.

By Lemma 3.3.3, some process $p$ executed $\text{REP}_p(\text{op})$ before $\text{RES}(\text{op})$ was executed by the client invoking $\text{op}$. By the construction of $S$, $T(\text{op}') < T(\text{op})$, and so $p$ executed $\text{REP}_p(\text{op}')$ before executing $\text{REP}_p(\text{op})$, by Lemma 3.3.9. However, by Lemma 3.3.6, $p$ must execute $\text{REP}_p(\text{op})$ before executing $\text{REP}_p(\text{op}')$, which, by Lemma 3.3.5, is a contradiction. Therefore, since $\prec_S$ is a total order, $\text{op} \prec_S \text{op}'$.

Since $S$ preserves the real-time order of operations in $H$ and all process subhistories are sequential, $\text{complete}(\text{extend}(H|x))|_c = S|_c$ for any client process $c \in \Pi_c$. $S$ is therefore equivalent to $\text{complete}(\text{extend}(H|x))$.

It remains to show that $S$ is legal. By Lemma 3.3.11 and the fact that $\prec_S = \prec_T$, $\text{REP}_p(\text{op}_0), \text{REP}_p(\text{op}_1), \ldots, \text{REP}_p(\text{op}_n)$ is the sequence of replica operations executed by server $p$ in $R$, where $\text{op}_0, \text{op}_1, \ldots, \text{op}_n$ is the sequence of operations in $S$. We can write this sequence of replica operation events as follows:

$$\langle p, \text{rep}(x, f_0, rlt_0) \rangle, \langle p, \text{rep}(x, f_1, rlt_1) \rangle, \ldots, \langle p, \text{rep}(x, f_n, rlt_n) \rangle.$$

Let $(f_0, rlt_0), (f_1, rlt_1), \ldots, (f_n, rlt_n)$, be the sequence of method, result pairs that occur in this sequence of replica operation events. By the algorithm, $\text{REP}_p(\text{op}) = \langle p, \text{rep}(x, f, rlt) \rangle$ if and only if $\text{INV}(\text{op}) = \langle c, \text{inv}, (x, f) \rangle$ and $\text{RES}(\text{op}) = \langle c, \text{res}, (x, rlt) \rangle$ for some client $c$. Therefore, the sequence of method, result pairs formed by the sequence of operations in $S$ is equal to $(f_0, rlt_0), (f_1, rlt_1), \ldots, (f_n, rlt_n)$. Since the sequence of replica operations executed by a server on a replica object is always legal, $(f_0, rlt_0), (f_1, rlt_1), \ldots, (f_n, rlt_n)$ forms a path through the sequential specification of $x$ starting from the initial state of $x$. Thus, $S$ is legal. □

**Theorem 3.3.1** $A$ is a linearizable consistency protocol for $x$.

**Proof:** Follows from Lemmata 3.3.1, 3.3.2, and 3.3.12. □
3.3.4 Performance Analysis

Let the latency of an operation be the time between the invocation and response of that operation. In the algorithm in Figure 4.1, the latency of an operation depends on latency of an Atomic Multicast, i.e., the time between the A-multicast of any message \( m \) and the A-delivery of \( m \) by any process in \( \text{group}(m) \). If we make the assumptions outlined in Section 2.2 then the latency of any operation on an object \( x \) is \( d_A(\text{objGroup}(x)) + d_m \) where \( d_A(G) \) is the latency of a request message addressed to a group \( G \) and \( d_m \) is the fixed message delay defined in Section 2.2. The message cost of an operation is \( n_A(\text{objGroup}(x)) + |\text{objGroup}(x)| \) messages, where \( n_A(G) \) is the message cost of A-multicasting a message to a group \( G \). The values of \( d_A \) and \( n_A \) depend on the particular Atomic Multicast algorithm that is used.

The performance of the algorithm can be easily improved in the case where a process \( c \) is performing an operation \( op \) on an object \( x \) such that \( c \in \text{objGroup}(x) \). Since \( p \) will perform the replica operation for \( op \) on its copy of \( x \), the other processes need not send a reply message for \( op \) to \( p \). Thus, we can modify the algorithm so that a server process \( p \in \text{objGroup}(x) \) sends a reply message to a client \( c \) after performing a replica operation on \( x_p \) only if \( c \notin \text{objGroup}(x) \). \( c \) then returns the result of its own replica operation on \( x_c \). Using this modified algorithm, every operation by a client \( c \) on an object \( x \) such that \( c \in \text{objGroup}(x) \) has a latency of \( d_A \) and a message cost of \( n_A \) messages.
Chapter 4

Read/Update Shared Objects

4.1 Defining Read/Update Shared Objects

A read/update object is an object whose type defines methods that do not modify the object's state. Read/update objects allow processes to read the object's state (or part of it) without altering it. When referring to read/update objects, operations that do not modify the object's state are called read operations, while operations that do modify the object's state are called update operations.

A read/write object, e.g., a register, is a restricted form of read/update objects in which the update operations do not depend on the original state of the object. In other words, a write operation always leaves an object in the same state, regardless of the state of the object when the write operation was invoked. Therefore, the state of an object only depends on the last write operation that was invoked on that object.

An example of a read/update object that is not a read/write object is a FIFO queue for which an additional method head has been defined that returns the contents of the entry at the head of the queue, without removing that entry from the queue.

The consistency algorithm in Figure 3.3 is not efficient when used to implement read/update objects. Each time a read operation is invoked on an object z, a request
message is sent to all server processes in \( \text{objGroup}(x) \) using Atomic Multicast. Since read operations do not modify an object’s state, the request message for a read operation does not need to be sent using a form of multicast with agreement and ordering semantics as strong as those provided by Atomic Multicast.

For example, the multicast used to send a read request does not need to satisfy Uniform Agreement since the replica operation for a read may be executed by some servers and not others without consequence. This multicast does not need to satisfy Local Delivery Consistency either, since two read operations do not necessarily have to be executed in the same order by any two servers. Also, given any update operation \( u \) and any read operation \( r \), it doesn’t matter if one server executes the replica operation for \( u \) before the replica operation for \( r \) and another server executes the replica operations in the opposite order.

What is required of the multicast used to send read request messages is that it ensures that every read operation returns a result that allows it to be linearized, i.e., sequentially ordered with the rest of the operations in a run, in a manner that preserves the real-time order of those operations. If we assume that any result returned by a read operation \( r \) is the result of a replica operation for \( r \), then we can ensure that \( r \) can be linearized by making sure that the result returned for \( r \) is the result of a replica operation for \( r \) that was performed by a server that had previously executed a replica operation for every update operation that completed before \( r \) was invoked.

Therefore, we require that the multicast used to sent read request messages satisfies a property similar to the Linear Order property described in Subsection 3.2.3. Specifically, a read request message that is multicast at time \( t \) must be delivered after all update request messages that were delivered by any process before \( t \). If we assume that all update request messages are A-multicast then we require that a request message that is multicast at time \( t \) is only delivered by a process after that process has A-delivered every message that was A-delivered by any process before \( t \).
4.2 Linear Multicast

In order to adapt the consistency protocol in Figure 3.3 so that it efficiently implements read operations, we define a new type of multicast called Linear Multicast. The multicast and deliver primitives of a Linear Multicast shall be referred to as L-multicast and L-deliver respectively.

The semantics of Linear Multicast are weaker than those of an Atomic Multicast since Uniform Agreement and Local Delivery Consistency are not satisfied. Informally, a Linear Multicast is a Weak Reliable Multicast that places constraints on the set of processes that L-deliver any message in linear order.

For Atomic Multicast, we say that a message $m$ is A-delivered in linear order by a process $p$ if $p$ A-delivers $m$ only after A-delivering every message $m'$ that was A-delivered by any process before $m$ was multicast. In the case of Linear Multicast, we don’t require that L-multicast message are ordered with respect to other L-multicast messages, only that they are ordered with respect to A-multicast messages. Specifically, we say that a message $m$ that is L-multicast at time $t$ is \textit{L-delivered in linear order} by a process $p$ if $p$ L-delivers $m$, and before L-delivering $m$, $p$ A-delivered every message $m'$ such that $\text{group}(m') = \text{group}(m)$ and $m'$ was A-delivered by some process before $t$. If a process $p$ that L-delivers a message $m$ does not L-deliver $m$ in linear order, then we say that $p$ L-delivers $m$ \textit{out of linear order}.

Also, we do not require that \textit{all} processes in $\text{group}(m)$ L-deliver a message $m$ in linear order, but only that some subset of processes in $\text{group}(m)$ do not L-deliver $m$ out of linear order. We define two types of Linear Multicast that vary by the constraints that they impose on the set of processes that L-deliver messages out of linear order. We will show in Chapter 5 that the type of Linear Multicast that can be implemented depends on the type of failure detector that is available.

The properties that are used to define the two types of Linear Multicast are expressed in terms of a function $\beta$. This is referred to as the \textit{weakness function} of a Linear Multicast.
A weakness function $\beta$ is a function from $G$ to the set of non-negative integers such that for any group $G \in \mathcal{G}$, $0 \leq \beta(G) \leq |G| - 1$. This function determines the maximum number of processes in a group that may L-deliver a message out of linear order.

A **Strong Linear Multicast (with a weakness function $\beta$)** is a Weak Reliable Multicast that satisfies the following property:

**Strong Linear Order**: If a process L-multicasts a message $m$ at time $t$ then no more than $\beta(group(m))$ processes L-deliver $m$ out of linear order.

A **Weak Linear Multicast (with a weakness function $\beta$)** is a Weak Reliable Multicast that satisfies the following property:

**Weak Linear Order**: If a process L-multicasts a message $m$ at time $t$ then if more than $\beta(group(m))$ processes L-deliver $m$ out of linear order then there exists some correct process in $group(m)$ that is never suspected by any process and does not L-deliver $m$ out of linear order.

We will present two consistency protocols for read/update objects, one that uses Strong Linear Multicast and one that uses Weak Linear Multicast. In Chapter 5, we will show how to implement Strong Linear Multicast using Perfect, Eventually Perfect or Eventually Strong failure detectors, and how to implement Weak Linear Multicast using Strong failure detectors. For clarity, the multicast and delivery primitives of a Strong Linear Multicast shall be referred to as SL-multicast and SL-deliver, and a Weak Linear Multicast’s primitives will be referred to as WL-multicast and WL-deliver.

### 4.3 Proving the Linearizability of Read/Update Objects

Before presenting the consistency protocols and their proofs of correctness, we introduce a technique for proving the linearizability of read/update objects that uses a partial order
on the operations in a history to show that the history is linearizable. This technique is a modification of similar techniques used by Lynch [38], and Lynch and Shvartsman [39] to prove the linearizability of read/write object implementations, and by Fekete, Kaashoek and Lynch [18] to prove the sequential consistency of read/update object implementations. We now define the class of partial orders that we will use in our proof technique:

**Definition 4.3.1** Given a complete object subhistory \( H|x \) of an object \( x \) where \( x \) is of type \( T = (V, F, RLT, Q) \) and has an initial state of \( v_{init} \), an irreflexive partial order \( <_P \) of the operations in \( H|x \) is linearly supportive of \( H|x \) if:

1. \( <_P \) is well-founded (every operation in \( H|x \) has only a finite number of predecessors in \( <_P \)).

2. For any two operations \( op, op' \in H|x \), if \( op <_H op' \) then it cannot be the case that \( op' <_P op \).

3. If \( u \) is an update operation, and \( op \) is any operation in \( H|x \), then either \( u <_P op \) or \( op <_P u \).

4. For each operation \( op \in H|x \), let \( u_0, u_1, \ldots, u_n \) be the sequence of update operations that precede \( op \) in \( <_P \), in the order of appearance in \( <_P \). Let \( v_{op} \) be the state of \( x \) resulting if \( x \) is initialized to \( v_{init} \) and operations \( u_0, u_1, \ldots, u_n \) are applied to \( x \) in that order. If \( INV(op) = \{p, inv, (x, f)\} \) and \( RES(op) = \{p, res, (x, rlt)\} \), then there is an edge from \( Q(v_{op}) \), labeled \( (f, rlt) \), in the sequential specification \( Q \) of \( T \).

Before showing how linearly supportive partial orders can be used to prove the linearizability of objects, we prove the following lemma concerning the real-time ordering of operations in a history:

**Lemma 4.3.1** Given any history \( H \), and operations \( op_0, op_1, op_2, op_3 \in H \), if \( op_0 <_H op_1 \) and \( op_2 <_H op_3 \), then \( op_0 <_H op_3 \) or \( op_2 <_H op_1 \).
Proof: Let $H$ be a history with operations $op_0, op_1, op_2, op_3 \in H$ such that $op_0 <_H op_1$ and $op_2 <_H op_3$. By the definition of $<_H$, $RES(op_0)$ precedes $INV(op_1)$ and $RES(op_2)$ precedes $INV(op_3)$ in $H$. We now show that if $op_0 \notin_H op_3$ then $op_2 <_H op_1$. If $op_0 \notin_H op_3$, then $INV(op_3)$ precedes $RES(op_0)$ in $H$ by the definition of $<_H$. Therefore, since $RES(op_2)$ precedes $INV(op_3)$, and $RES(op_0)$ precedes $INV(op_1)$ in $H$, it follows that $RES(op_2)$ precedes $INV(op_1)$ in $H$. Thus, $op_2 <_H op_1$ by the definition of $<_H$. □

We can now show that given the union of a linearly supportive partial order and the partial order inferred by the real-time order of operations in a history, the transitive closure of this union is itself a linearly supportive partial order.

Lemma 4.3.2 Let $H_x = H|_x$ be a complete object subhistory of an object $x$, and let $<_P$ be an irreflexive partial order on the operations in $H_x$ that is linearly supportive of $H_x$. The transitive closure of the union of $<_P$ and $<_H$ is an irreflexive partial order that is linearly supportive of $H_x$.

Proof: Let $<_P'$ be the transitive closure of the union of $<_P$ and $<_H$. We begin the proof by showing that $<_P'$ is an acyclic relation. The proof is by contradiction. If $<_P'$ is not an acyclic relation, then there exists a set of operations $op_1, op_2, \ldots, op_n \in H_x$ such that $op_1 <_P' op_2 <_P' \ldots <_P' op_n <_P' op_1$, and each pair is directly related by some element of $<_P$ or $<_H$. We will choose such a cycle $c$ whose length is minimal.

Since $<_H$ and $<_P$ are both transitive, no two adjacent relations in $c$ are both elements of $<_H$ or $<_P$, as this would imply that $c$ was not of minimal length. The relations in $c$ therefore alternate between elements of $<_P$ and $<_H$ (i.e., $op_2 <_P op_3 <_H op_4 <_P \ldots$). This implies that $c$ is of even length. If $c$ consisted of only 2 operations, then we would have a situation where $op_0 <_H op_1 <_P op_0$, which violates the second condition of linearly supportive partial orders. Thus, $c$ must have a length of at least 4 operations.
Consider a string of operations $op_{-1}, op_0, op_1, op_2, op_3$ in $c$ that are numbered such that $op_{-1} <_{H_z} op_0 <_{P} op_1 <_{H_z} op_2 <_{P} op_3$. (In the case where $c$ is of length 4, $op_{-1} = op_3$). By Lemma 4.3.1, either $op_{-1} <_{H_z} op_2$ or $op_1 <_{H_z} op_0$. If $op_1 <_{H_z} op_0$, then the second condition of linearly supportive partial orders is violated since $op_0 <_{P} op_1$. If $op_{-1} <_{H_z} op_2$, then $c$ is not of minimal length since $op_0$ and $op_1$ can be removed from $c$ to yield a smaller cycle. $<_{P'}$ is therefore acyclic, and is transitive and irreflexive by definition. Thus, $<_{P'}$ is an irreflexive partial order on the operations in $H_z$.

We must now show that $<_{P'}$ is linearly supportive of $H_z$, i.e., that $<_{P'}$ satisfies all four conditions of linearly supportive partial orders. Condition one is trivially satisfied since both $<_{P}$ and $<_{H_z}$ are well-founded. Conditions two and three are also trivially satisfied since $<_{P}$ is linearly supportive of $H_z$. It remains to show that condition four holds for $<_{P'}$.

If we can show that, for any operation $op$ in $H_z$, the sequence of update operations that precede $op$ in $<_{P'}$ is identical to the sequence of update operations that precede $op$ in $<_{P}$, then condition four holds for $<_{P'}$ since it holds for $<_{P}$.

Let $op$ be any operation in $H_z$, and let $u$ be any update operation in $H_z$. Since $<_{P'}$ is the transitive closure of the union of $<_{P}$ and $<_{H_z}$, $u <_{P'} op$ if $u <_{P} op$. We now show that $u <_{P'} op$ only if $u <_{P} op$. Assume, by way of contradiction that $u <_{P'} op$ and $u <_{P} op$. Since $<_{P}$ satisfies condition three, $op <_{P} u$. Thus, by the construction $<_{P'}$, $op <_{P'} u$. This is a contradiction since $u <_{P'} op$ and $<_{P'}$ is acyclic. Therefore, $u <_{P'} op$ if and only if $u <_{P} op$. Thus, given any operation $op$ in $H_z$, the sequence of update operations that precede $op$ in $<_{P'}$ is identical to the sequence of update operations that precede $op$ in $<_{P}$. 

The following lemma is used to prove the correctness of linearizable read/update shared object consistency protocols. Its proof is largely based on the proof of a similar lemma presented by Fekete, Kaashoek and Lynch [18].
Lemma 4.3.3 If an object subhistory \( H_x = H|x \) of an object \( x \) and a history \( H \) can be extended to a history \( H'_x \) such that there is an irreflexive partial order \(<_p\) on the operations in \( \text{complete}(H'_x) \) that is linearly supportive of \( \text{complete}(H'_x) \), then \( H_x \) is linearizable.

Proof: Let \( H_x = H|x \) be an object subhistory of an object \( x \). Let \( H'_x \) be an extension of \( H_x \) such that there is an irreflexive partial order \(<_p\) on the operations in \( \text{complete}(H'_x) \) that is linearly supportive of \( \text{complete}(H'_x) \). Let \(<_{CH_x}\) be the partial order on the operations in \( \text{complete}(H'_x) \) that is inferred from the real time order of operations in \( \text{complete}(H'_x) \), and let \(<_{p'}\) be the transitive closure of the union of \(<_p\) and \(<_{CH_x}\). By Lemma 4.3.2, \(<_{p'}\) is an irreflexive partial order that is linearly supportive of \( H_x \). We note that since \(<_{CH_x}\subseteq<_{p'}\) and all process subhistories are sequential, \(<_{p'}\) totally orders all of the operations in \( \text{complete}(H'_x)|c \) for any client process \( c \in \Pi_C \). Therefore, no two operations that were invoked by the same client have the same number of predecessors in \(<_{p'}\).

We can extend \(<_{p'}\) to a total order \(<_{TO}\) as follows: \( \forall op_1, op_2 \in \text{complete}(H'_x) \), let \( op_1 <_{TO} op_2 \) provided that \( op_1 \) has fewer predecessors in \(<_{p'}\) than \( op_2 \), or else \( op_1 \) and \( op_2 \) have the same number of predecessors in \( P' \) and the process that invoked \( op_1 \) precedes the process that invoked \( op_2 \) in some fixed total ordering of \( \Pi_C \). By construction, \(<_{TO}\) is a total ordering of the operations in \( \text{complete}(H'_x) \), and \(<_{p'}\subseteq<_{TO}\).

We will now show that \(<_{TO}\) is linearly supportive of \( \text{complete}(H'_x) \). Since \(<_{p'}\subseteq<_{TO}\), the second and third conditions follow from the fact that \(<_{p'}\) is linearly supportive.

Since \(<_{p'}\) totally orders all of the operations invoked by a particular client, there are at most \( n(N + 1) \) operations that have \( \leq N \) predecessors in \( P \), where \( n = |\Pi_C| \). By the definition of \(<_{TO}\), if an operation has \( N \) predecessors in \(<_{p'}\), then each of its predecessors in \(<_{TO}\) must have at most \( N \) predecessors in \(<_{p'}\). Since there are at most \( n(N + 1) \) such operations, the operation has at most \( n(N + 1) \) predecessors in \(<Q\). Thus the first condition holds for \(<_{TO}\).
Because the third and fourth conditions hold for \( \prec_p \), the fourth condition holds for \( \prec_{TO} \), by the construction of \( \prec_{TO} \).

We now have a total ordering \( \prec_{TO} \) of the operations in \( complete(H'_x) \). Since every operation in \( \prec_{TO} \) has only a finite number of predecessors, we can create a sequence of operations by arranging the invocation and response events that constitute each operation in \( complete(H'_x) \) in the order given by \( \prec_{TO} \). This transformation yields a sequential history \( S \) of the operations in \( complete(H'_x) \) where \( \prec_S \), the total order inferred by the real-time order of operations in \( S \), is equal to \( \prec_{TO} \).

Since \( \prec_{CH_x} \subseteq \prec_p \subseteq \prec_S \), \( S \) preserves the real-time ordering in \( complete(H'_x) \). Since all process subhistories are sequential, \( S|c = complete(H'_x)|c \) for any client \( c \in \Pi_C \), thus \( S \) and \( complete(H'_x) \) are equivalent.

Finally, to show that \( S \) is legal, we must show that the methods and results of the operations in \( S \) form a sequence \( (f_1, rlt_1), (f_2, rlt_2), \ldots, (f_n, rlt_n) \) that is a path in \( Q \) from the vertex \( Q(v) \), where \( x \) is of type \( T = (V, F, RLT, Q) \) and has an initial state \( v \). Since condition four holds for \( \prec_S \), the methods and results from the subsequence of update operations in \( S \) forms a path through \( Q \) from \( Q(v) \). Since read operations do modify the state of an object, \( (f_1, rlt_1), (f_2, rlt_2), \ldots, (f_n, rlt_n) \) must also form a path through \( Q \) from \( v \). Thus, \( S \) is legal and \( H_x \) is therefore linearizable. \( \square \)

### 4.4 A Consistency Protocol for Read/Update Objects using Strong Linear Multicast

#### 4.4.1 Description

A linearizable consistency protocol for read/update objects can be implemented using Atomic Multicast and Strong Linear Multicast as shown in Figure 4.1. This algorithm is an enhancement of the general algorithm in Figure 3.3. A client process must now
Every client process c executes the following:

Initialization:
\[ \text{req} \leftarrow 0 \]

To execute \text{apply-update}(x, f):
\[
\begin{align*}
\text{req} & \leftarrow \text{req} + 1 \\
\text{A-multicast}(c, \text{req}, x, f) & \text{to objGroup}(x) \\
\text{wait until} & \text{[received (req, res) from some server process p]} \\
\text{return res}
\end{align*}
\]

To execute \text{apply-read}(x, f):
\[
\begin{align*}
\text{req} & \leftarrow \text{req} + 1 \\
\text{SL-multicast}(c, \text{req}, x, f) & \text{to objGroup}(x) \\
\text{wait until} & \text{[for } \beta(\text{objGroup}(z)) + 1 \text{ processes } p: \text{ received (req, upd, res) from } p] \\
\text{replies} & \leftarrow \{(\text{req, upd, res}) | c \text{ received (req, upd, res) from } p\} \\
\text{lr} & \leftarrow \text{largest } \text{upd} \text{ such that } (\text{req, upd, res}) \in \text{replies} \\
\text{readRes} & \leftarrow \text{select one } \text{res} \text{ such that } (\text{req, lr, res}) \in \text{replies} \\
\text{return readRes}
\end{align*}
\]

Every server process p executes the following:

Initialization:
\[
\begin{align*}
\text{for every object } z \text{ such that } p \in \text{objGroup}(z): \\
\text{initializeReplica}(z) \\
\text{upd}[z] & \leftarrow 0
\end{align*}
\]

when A-deliver(c, req, x, f):
\[
\begin{align*}
\text{upd}[z] & \leftarrow \text{upd}[z] + 1 \\
\text{res} & \leftarrow \text{applyToReplica}(x, f) \\
\text{send } (\text{req, upd}, \text{res}) & \text{to } c
\end{align*}
\]

when SL-deliver(c, req, x, f):
\[
\begin{align*}
\text{res} & \leftarrow \text{applyToReplica}(x, f) \\
\text{send } (\text{req, upd}[x], \text{res}) & \text{to } c
\end{align*}
\]

Client Tasks 1 and 2 cannot be executed concurrently.
Server Tasks 1 and 2 cannot be executed concurrently.

Figure 4.1: A consistency protocol for read/update objects using Atomic Multicast and Strong Linear Multicast.
call two separate functions, `apply-update` and `apply-read`, to invoke update operations and read operations respectively. The `apply-update` function is identical to the `apply` function of the general algorithm. `apply-read` differs from `apply` in that read request messages are multicast using Strong Linear Multicast instead of Local Atomic Multicast, and, instead of returning the result contained in the first reply that is received, the result returned by the operation is chosen from the first \( \beta(objGroup(x)) + 1 \) replies that are received, where \( \beta \) is the weakness function of the Strong Linear Multicast algorithm that is used. We therefore require that for every object \( x \), \( \beta(objGroup(x)) < |objGroup(x)| \); otherwise, a client will never receive the \( \beta(objGroup(x)) + 1 \) replies it requires.

Update operations are handled by the server processes in the same way as in the general consistency protocol. The update counters, however, now serve a purpose in the algorithm, and are no longer merely auxiliary variables. When a server process \( p \) SL-delivers a read request message, it performs the replica operation on its replica of \( x \), and sends a reply message to the client. This reply message includes the process’ update counter \( upd[x] \) for \( x \) so that the client knows how many update replica operations were performed before the read replica operation was performed.

Strong Linear Multicast differs from Atomic Multicast in that it satisfies the weaker properties of Weak Uniform Agreement and Strong Linear Order, instead of Uniform Agreement and Local Delivery Consistency. Since SL-Multicast does not satisfy Local Delivery Consistency, the read request messages are not totally ordered with respect to each other or the update request messages. Therefore, when two servers execute the replica operation for the read, their respective replicas may be in different states. The reply messages sent to the client may therefore contain different result values. However, since read operations do not modify the object’s state, we only need to ensure that the client returns a result value that allows the read operation to be linearized.

The portion of the algorithm that is executed by client processes is composed of two tasks: Client Task 1 and Client Task 2. Client Task 1 is executed when `apply-update`
is called, and Client Task 2 is executed when apply-read is called. These tasks cannot be executed concurrently since client processes must execute operations sequentially. We are not concerned with how this constraint is enforced by a distributed system; we only require that it is enforced somehow.

The portion of the algorithm that is executed by server processes is also composed of two tasks: Server Task 1 and Server Task 2. Server Task 1 is executed when an update request message is A-delivered, and Server Task 2 is executed when a read request message is SL-delivered. These tasks also cannot be executed concurrently since calls to applyToReplica must be sequential.

A detailed proof of the correctness of the algorithm can be found in Subsection 4.4.3; however, the correctness of the algorithm intuitively follows from the properties of Strong Linear Multicast. After SL-multicasting the read request message, the client invoking the read operation waits until it receives $\beta(objGroup(x)) + 1$ reply messages. The Strong Linear Order property of SL-Multicast ensures that no more than $\beta(objGroup(x))$ processes deliver the request message out of linear order. At least one of the reply messages received is therefore from a server process that previously executed the replica operations for all updates that returned before the read was invoked. Furthermore, a server process that had performed all these update replica operations has an update counter value that is higher than the update counter value of a server process that has not performed these replica operations. Therefore, by choosing the result value that is included in the reply message containing the highest update counter value, the client can be sure that it is returning a result that allows the read operation to be linearized.

Finally, since SL-Multicast satisfies Weak Uniform Agreement instead of Uniform Agreement, a read request $m$ might not be SL-delivered by all correct processes in $\text{group}(m)$ if it is SL-multicast by a faulty client process. This means that if a process SL-delivers $m$ and then executes the replica operation for the read, all correct processes in $\text{group}(m)$ are not guaranteed to do so as well. This is acceptable, however, since read
operations do not modify an object’s state.

4.4.2 Fault-Tolerance

As with the algorithm in Figure 3.3, in order for update operation on an object \( x \) to terminate, at least one process in \( \text{objGroup}(x) \) must be correct. However, since \( \beta(\text{objGroup}(x)) + 1 \) replies must be received by a client process before it returns from a read operation on \( x \), no more than \( |\text{objGroup}(x)| - \beta(\text{objGroup}(x)) - 1 \) processes may fail in \( \text{objGroup}(x) \). The fault-tolerance of this algorithm depends on the weakness function \( \beta \) of the Strong Linear Multicast that is used.

The fault-tolerance of the algorithm in Figure 4.1 also depends on the fault-tolerance of the Atomic Multicast and Strong Linear Multicast algorithms it employs. Because the Strong Linear Order property of Strong Linear Multicast is defined in terms of Atomic Multicast, we will consider Atomic Multicast and Linear Multicast to be implemented by a single algorithm. We will refer to this algorithm simply as the “multicast algorithm.”

If we let \( \Gamma' \) be the tolerance function for the multicast algorithm, then the algorithm in Figure 4.1 is a \( \Gamma' \)-tolerant consistency protocol where:

\[
\forall G \in \mathcal{G} : \Gamma'(G) = \begin{cases} 
\min(|G| - \beta(G) - 1, \Gamma'(G)) & \text{if } G = \text{objGroup}(x) \text{ for some } x \in X \\
\Gamma'(G) & \text{otherwise}
\end{cases}
\]  

As before, we will parameterize the Atomic Multicast and Strong Linear Multicast problems in terms of a group \( G \). This will allow us to determine the fault-tolerance of the consistency protocol for each object \( x \in X \). We will assume that the multicast algorithm is a \( \Gamma'_G \)-tolerant solution to Atomic and Strong Linear Multicast for a group \( G \in \mathcal{G} \) where for some \( f < |G| \):

\[
\forall G' \in \mathcal{G} : \Gamma'_G(G') = \begin{cases} 
f & \text{if } G' = G \\
|G'| & \text{otherwise}
\end{cases}
\]  

(In Chapter 5, we will show that algorithms exist that meet this criterion.) Under this
assumption, the algorithm in Figure 4.1 is a $\Gamma_x$-tolerant consistency protocol for any object $x \in X$ where:

$$\forall G \in \mathcal{G} : \Gamma_x(G) = \begin{cases} 
\min(|G| - \beta(G) - 1, \Gamma'_G(G)) & \text{if } G = \text{objGroup}(x) \\
|G| & \text{otherwise}
\end{cases}$$

(4.3)

### 4.4.3 Proof of Correctness

Let $x \in X$ be any read/update object, let $A_r$ denote the algorithm in Figure 4.1 and let $\Gamma_x$ be the tolerance function defined in Equation 4.3. In this subsection, we will show that $A_r$ is a $\Gamma_x$-tolerant linearizable consistency protocol for $x$. It then follows, as discussed in the previous subsection, that $A_r$ is a $\Gamma$-tolerant linearizable consistency protocol for read/update objects where $\Gamma$ is the tolerance function defined in Equation 4.1.

Let $R = (\mathcal{F}, \mathcal{H}, I, S, T)$ be any run in the system formed by $A_r$ and $\Gamma_x$. By the definition of $\Gamma_x$ we can assume that at least $\beta(\text{objGroup}(x)) + 1$ processes are correct in $\text{objGroup}(x)$.

As in the proof of the algorithm in Figure 3.3, a call to either $\text{apply-read}$ or $\text{apply-update}$ is modeled by an invocation event, and a return from either of these functions is modeled by a response event. Also, a call to $\text{applyToReplica}$ is modeled by a replica operation event, and $\text{REP}_p(op)$ is used to denote the replica operation performed by a server process $p$ for an operation $op$. We will again write $e_0 <_R e_1$ to denote that event $e_0$ precedes event $e_1$ in the schedule $S$ of $R$.

Since the update operations in the $A_r$ are implemented in the same way as the operations in the previous algorithm $A$ in Figure 3.3, Lemmata 3.3.1 through 3.3.7 also apply to $A_r$, except that they only refer to update operations in the context of $A_r$ where they referred to all operations in the context of $A$.

**Lemma 4.4.1** If a correct process $p$ calls $\text{apply-read}(x, f)$ then eventually $p$ returns from this call.
Proof: By the algorithm, the only place where \( p \) may wait indefinitely before returning is when it waits for \( \beta(objGroup(x)) + 1 \) reply messages to be received. Before waiting, \( p \) SL-multicasts a request message \( m \) to all servers in \( objGroup(x) \). Since \( p \) is correct, and at least one server in \( objGroup(x) \) is correct, all correct servers will eventually receive \( m \) by the Validity and Weak Uniform Agreement properties of Strong Linear Multicast. Since \( \beta(objGroup(x)) < |objGroup(x)|-\Gamma_A(objGroup(x)) \), there are at least \( \beta(objGroup(x)) + 1 \) correct processes in \( objGroup(x) \) in any run \( R \) of \( A \). Since each correct server that receives \( m \) sends a reply message to \( p \), \( p \) will eventually receive \( \beta(objGroup(x)) + 1 \) reply messages, and the call to apply-read will eventually return. \( \square \)

Lemma 4.4.2 (Termination) If a correct client process \( c \) executes an invocation on \( x \) then \( c \) eventually executes a matching response.

Proof: Follows from Lemmata 3.3.1 and 4.4.1. \( \square \)

Lemma 4.4.3 (Uniform Integrity) Every response to an operation on \( x \) matches exactly one invocation, and every invocation of an operation on \( x \) is matched by at most one response.

Proof: Immediate from the algorithm. \( \square \)

Lemma 4.4.4 Given any complete operation \( op \) on \( x \) in \( R \), there exists at least one replica operation \( REP_p(op) \) executed by some process \( p \) in \( R \) such that \( REP_p(op) \leq_R RES(op) \).

Proof: By the algorithm, when a client process \( c \) calls apply-update\((x,f)\) (or apply-read\((x,f)\)) it A-multicasts (or SL-multicasts) a request message \( m = (c,req,x,f) \) to all of the server processes in \( objGroup(x) \), where \( req \) is a request id used by \( c \) to match replies with requests. \( c \) then waits until it receives one (or \( \beta(objGroup(x)) + 1 \)) matching reply message(s) before returning. Also by the algorithm, when a server process in
objGroup(x) A-delivers (or SL-delivers) a request message m = (c, req, x, f), it does not send a reply message to c until it has performed a replica operation \(\rho, rep, (x, f, rlt)\). Thus, no operation \(op\) can return before at least one replica operation \(REP_p(op)\) has been executed by some server process \(p\). □

**Lemma 4.4.5** Given any operation \(op\) on \(x\) and any replica operation \(REP_p(op)\) for \(op\) that is executed by some process \(p\), \(INV(op) <_R REP_p(op)\).

**Proof:** By the algorithm, a replica operation \(REP_p(op)\) is performed after a server process \(p\) has A-delivered (or SL-delivered) a request message \(m\). By the Uniform Integrity property of A-Multicast (or SL-multicast), \(m\) must have been multicast before it was delivered. Since request messages are multicast by client processes after an operation is invoked, \(INV(op) <_R REP_p(op)\). □

**Lemma 4.4.6** Given any operation \(op\) on \(x\), a server process \(p \in objGroup(x)\) executes at most one replica operation \(REP_p(op)\).

**Proof:** By the algorithm, a single request message \(m\) is A-multicast (or SL-multicast) for any operation \(op\). By Uniform Integrity, each server process \(p \in objGroup(x)\) A-delivers (or SL-delivers) \(m\) at most once. Since a replica operation \(REP_p(op)\) is only performed when a request message for \(op\) is delivered, \(p\) executes \(REP_p(op)\) at most once. □

We now redefine the function \(T\), originally defined in Subsection 3.3.3, in the context of this new consistency algorithm. \(T\) is a function from the set of all completable update operations and complete read operations on \(x\) in \(R\) to the set of natural numbers. Given any completable update operation \(u\) on \(x\) in \(R\), \(T(u)\) is the value of the update counter \(upd_p[x]\) of any server process \(p\) when it executes \(REP_p(u)\). This value must exist and is unique since Lemmata 3.3.3 and 3.3.7 apply to all update operations on \(x\). Given
any read operation \( r \) on \( x \) invoked by a client process \( c \) in \( R \), \( T(r) \) is the value of the variable \( lrg \) that is calculated before \( c \) returns from \( r \).

We note that the definition of \( T(u) \) for any update operation \( u \) is identical to the definition of \( T(u) \) in the proof of \( A \). Therefore, Lemmata 3.3.8, 3.3.9 and 3.3.11 apply here as well, except that they refer only to completable update operations in \( R \) where they referred to all completable operations in \( R \) in the proof of \( A \).

**Lemma 4.4.7** Let \( r \) be a complete read operation on \( x \) in \( R \). If \( T(r) \neq 0 \) then \( T(r) = T(u) \), where \( u \) is some completable update operation on \( x \) in \( R \) such that \( REP_p(u) <_R REP_p(r) <_R RES(r) \), for some process \( p \in objGroup(x) \).

**Proof:** Let \( r \) be a complete read operation \( r \) on an object \( x \) such that \( T(r) \neq 0 \), and let \( c \) be the client process that invoked \( r \). By the algorithm and the definition of \( T \), \( c \) received a reply message \( m = (req, upd, res) \) before executing \( RES(r) \), where \( upd = T(r) \). Also by the algorithm, some server \( p \in objGroup(x) \) sent \( m \) to \( c \) after executing \( REP_p(r) \), and the value of \( upd_p[x] \) was \( T(r) \) when \( m \) was sent. Thus, \( REP_p(r) <_R RES(r) \). Furthermore, since \( T(r) \neq 0 \), \( upd_p[x] \) was set to \( T(r) \) after \( p \) A-delivered a request message for some update operation \( u \). By the algorithm, \( T(u) = T(r) \) and \( p \) executed \( REP_p(u) \) before it executed \( REP_p(r) \). \( \square \)

**Lemma 4.4.8** Given a read operation \( r \) on \( x \) that was invoked by a client \( c \), if a process \( p \in objGroup(x) \) sends a reply message \( m = (req, upd, res) \) to \( c \) where \( upd = T(r) \), then no update operation \( u \) on \( x \) exists such that \( T(r) < T(u) \) and \( REP_p(u) <_R REP_p(r) \).

**Proof:** If \( p \) sends a reply message \( m \) to \( c \) for a read operation \( r \), then by the algorithm \( p \) previously executed \( REP_p(r) \). Also, if \( m = (req,upd,res) \) where \( upd = T(r) \), then \( upd_p[x] = T(r) \) when \( p \) executes \( REP_p(r) \). Assume, by way of contradiction, that some update operation \( u \) on \( x \) exists such that \( T(r) < T(u) \) and \( REP_p(u) <_R REP_p(r) \). By the definition of \( T \) and Lemma 3.3.7, the value of \( upd_p[x] \) when \( p \) executes \( REP_p(u) \) is
$T(u)$. The value of $\text{upd}_p[x]$ therefore decreased between the time $REP_p(u)$ was executed and the time $REP_p(r)$ was executed. We can see from the algorithm, however, that the value of $\text{upd}_p[x]$ is monotonic non-decreasing, since $\text{upd}_p[x]$ is only changed when it is incremented by 1 when a request message is A-delivered. This is a contradiction. □

Lemma 4.4.9 Given an update operation $u$ on $x$ and a complete read operation $r$ on $x$, if $REP_p(u) <_R INV(r)$ for some process $p \in \text{objGroup}(x)$ then $T(u) \leq T(r)$.

Proof: Since $r$ is complete, the client $c$ invoking $r$ SL-multicast a request message $m$ at some time $t$ after $INV(r)$. By the algorithm, $p$ must have A-delivered the request message $m'$ for $u$ before executing $REP_p(u)$. Since $REP_p(u) <_R INV(r)$, $p$ A-delivered $m'$ before $m$ was SL-multicast by $c$. Therefore, by the Strong Linear Order property of SL-Multicast, no more that $\beta(\text{objGroup}(x))$ processes SL-deliver $m$ without first A-delivering $m'$. By the algorithm, $c$ waits for $\beta(\text{objGroup}(x)) + 1$ replies before calculating $lrg$, so at least one of these replies contains a value of $\text{upd}$ such that $T(u) \leq \text{upd}$. Therefore, by the definition of $T$, $T(u) \leq T(r)$. □

Let $H$ be the history that respects $R$. We now redefine $\text{extend}(H)$ to be the history that is constructed by appending response events to $H$ for each completable update operation $u$ in $H$. Formally, $\text{extend}(H)$ is defined as follows:

1. For every incomplete update operation $op$ in $H$ such that $INV(op) = (c, \text{inv}, (x, f))$, if a replica operation $REP_p(op) = (c, \text{rep}, (x, f, rlt))$ is executed by any process $p$ in $R$ then a response event $(c, \text{res}, (x, rlt))$ is appended to $H$.

2. Response events that are appended to $H$ appear in the same order as their matching invocations.

We define the correspondence mapping between the operations in $R$ and the operations in $\text{complete}(\text{extend}(H|x))$ as in the previous chapter (see page 57).
We now define an irreflexive relation $<_T$ on the operations in $\text{complete}(\text{extend}(H|x))$. Given any two operations $o_p$ and $o_p'$ in $\text{complete}(\text{extend}(H|x))$ and their respective, corresponding operations $o_p'$ and $o_p'$ in $R$, $o_p <_T o_p'$ if and only if:

1. $T(o_p) < T(o_p')$; or

2. $o_p'$ is an update, $o_p'$ is a read and $T(o_p') = T(o_p')$.

**Lemma 4.4.10** $<_T$ is an irreflexive partial order.

**Proof:** We will first show that $<_T$ is acyclic. Assume, by way of contradiction, that $<_T$ contains a cycle $o_p, o_p', \ldots, o_p_n$. Let $o_p', o_p', \ldots, o_p'_n$ be the operations in $R$ that correspond to $o_p, o_p, \ldots, o_p_n$ respectively. By the construction of $<_T$, if $o_p_i <_T o_p_j$ then $T(o_p_i) \leq T(o_p_j)$. This implies that $T(o_p) = T(o_p') = \ldots = T(o_p_n)$. Given two operations $o_p_i$ and $o_p_j$ where $T(o_p_i) = T(o_p_j)$, then $o_p_i <_T o_p_j$ only if $o_p_i$ is an update and $o_p_j$ is a read. The cycle must therefore consist of both reads and updates. If we assume, without loss of generality, that $o_p_0$ is an update, then $o_p_1$ must be a read, and there is no operation $o_p_2$ such that $o_p_1 <_T o_p_2$ and $T(o_p_1) = T(o_p_2)$.

We now show that $<_T$ is transitive. Let $o_p_0, o_p_1$ and $o_p_2$ be any operations in $\text{complete}(\text{extend}(H|x))$ such that $o_p_0 <_T o_p_1$ and $o_p_1 <_T o_p_2$. Let $o_p_0', o_p'_1$ and $o_p'_2$ be the operations in $R$ that correspond to $o_p_0, o_p_1$ and $o_p_2$ respectively. By the definition of $<_T$, $T(o_p_0') < T(o_p'_1)$, or $T(o_p_0') = T(o_p'_1)$ and $o_p'_1$ is a read operation. Also by the definition of $<_T$, $T(o_p'_1) \leq T(o_p'_2)$. Therefore, if $T(o_p_0') < T(o_p'_1)$ then $T(o_p'_0) < T(o_p'_2)$. If $T(o_p_0') = T(o_p'_1)$ then since $o_p'_1$ is a read operation, $T(o_p'_1) < T(o_p'_2)$, and so $T(o_p'_0) < T(o_p'_2)$. Thus, $o_p_0 <_T o_p_2$. □

For the remainder of this proof we will not distinguish the operations in $R$ from their corresponding operations in $\text{complete}(\text{extend}(H|x))$ in the interest of brevity.

**Lemma 4.4.11** $<_T$ is linearly supportive of $\text{complete}(\text{extend}(H|x))$. 
Proof: To show that $<_{T_z}$ is linearly supportive of $complete(extend(H|x))$, we must show that $<_{T_z}$ satisfies all four conditions for linearly supportive partial orders (see Definition 4.3.1).

We first show that the third condition of linearly supportive partial orders is satisfied. Let $u$ be any update operation in $complete(extend(H|x))$. By Lemma 3.3.8, given any other update operation $u'$ in $complete(extend(H|x))$, $T(u) \neq T(u')$. Thus, by the definition of $<_{T_z}$, either $u <_{T_z} u'$ or $u' <_{T_z} u$. Given any read operation $r$ in $complete(extend(H|x))$, either $u <_{T_z} r$ or $r <_{T_z} u$, by the definition of $<_{T_z}$. The third condition is therefore satisfied by $<_{T_z}$.

We now show that $<_{T_z}$ satisfies the second condition. Let $op$ and $op'$ be any two operations in $complete(extend(H|x))$ such that $op <_H op'$. This implies that $RES(op) <_R INV(op')$. Now assume, by way of contradiction, that $op' <_{T_z} op$. We now consider the four cases that are possible depending on the type of operations that $op$ and $op'$ are:

- **Case 1:** Both operations are updates. By Lemma 4.4.4, some process $p$ executed $REP_p(op)$ before $RES(op)$ was executed by the client invoking $op$. By the definition of $<_{T_z}$, $T(op') < T(op)$, and so $p$ executed $REP_p(op')$ before executing $REP_p(op)$, by Lemma 3.3.9. However, by Lemma 3.3.6, $p$ must execute $REP_p(op)$ before executing $REP_p(op')$, which, by Lemma 4.4.6, is a contradiction.

- **Case 2:** $op$ is an update, and $op'$ is a read. By Lemma 4.4.4, some process $p$ executed $REP_p(op)$ before $RES(op)$ was executed by the client invoking $op$. Since $RES(op) <_R INV(op')$ and $op'$ is complete, $T(op) \leq T(op')$ by Lemma 4.4.9. However, since $op' <_{T_z} op$, $T(op') < T(op)$ by the construction of $<_{T_z}$, which is a contradiction.

- **Case 3:** $op$ is a read, and $op'$ is an update. By the construction of $<_{T_z}$, $T(op') \leq T(op)$. We can therefore assume that $T(op) \neq 0$, since this would imply that $T(op') < 0$ which is not possible since $op'$ is an update operation. Thus, by
Lemma 4.4.7, $T(op) = T(u)$, where $u$ is an update operation on $x$ such that $REP_p(u) <_R REP_p(op) <_R RES(op)$ for some process $p \in objGroup(x)$. Since $T(op') \leq T(op)$, $T(op') < T(u)$ by Lemma 3.3.8, and so $p$ must have executed $REP_p(op')$ before executing $REP_p(u)$ by Lemma 3.3.9. Thus, since $RES(op) <_R INV(op')$, $REP_p(op') <_R INV(op')$ which contradicts Lemma 4.4.5.

- Case 4: Both operations are reads. By the construction of $<_T$, $T(op') < T(op)$, which implies that $T(op) \neq 0$ since $T(op') \geq 0$. By Lemma 4.4.7, some update operation $u$ on $x$ exists such that $T(u) = T(op)$, and $REP_p(u) <_R REP_p(op) <_R RES(op)$ for some process $p \in objGroup(x)$. Since $RES(op) <_R INV(op')$, and $op'$ is complete, $T(u) \leq T(op')$ by Lemma 4.4.9. This implies that $T(op) \leq T(op')$, which is contradiction.

To show that the first condition is satisfied, we must show that given any operation $op$, there is a finite number of operations that precede $op$ in $<_T$. Let $op'$ be any operation such that $op' <_T op$. By condition two, $INV(op')$ precedes $RES(op)$ in $complete(extend(H[x]))$, and so, by the definition of $H$, $INV(op') <_R RES(op)$. By our model, there is only a finite number of operations $op'$ in $R$ such that $INV(op') <_R RES(op)$. This implies that $op$ has only a finite number of predecessors in $<_T$.

We now show that $<_T$ satisfies the fourth condition. Let $op$ be any operation in $complete(extend(H[x]))$, let $u_0, u_1, \ldots, u_n$ be the sequence of update operations that precede $op$ in $<_T$, in the order of appearance in $<_T$, and let $v_{op}$ be the state of $x$ resulting if $u_0, u_1, \ldots, u_n$ are applied to $x$, in that order, when $x$ is in its initial state.

By the algorithm, the result $rlt$ returned by $op$ is the result of a replica operation $REP_p(op) = \langle p, rep, (x, f, rlt) \rangle$ that is executed by some server $p \in objGroup(x)$. We must show that $REP_p(u_0), REP_p(u_1), \ldots, REP_p(u_n)$ was the sequence of update replica operations executed by $p$ prior to executing $REP_p(op)$. Since read operations do not change the state of an object and the sequence of replica operations executed by a server process is always legal, this will imply that the state of $x_p$ was $v_{op}$ when $p$ executed
REP_p(op), and that an edge from Q(v_op), labeled (f, rlt) in Q, where Q is the sequential specification of the type of object x.

First, we will assume that op is any update operation in complete(extend(H|x)). By Lemmata 3.3.11, the sequence of replica update operations executed by any server p ∈ objGroup(x), prior to executing REP_p(op), is REP_p(u_0), REP_p(u_1), ..., REP_p(u_n), where u_0, u_1, ..., u_n is the sequence of update operations that precede op in ≺_{T_x}. Thus, the fourth condition holds for all update operations.

Now, assume that op is any read operation in complete(extend(H|x)), let c be the client that invoked op, and let req be the request id in the request message sent by c after invoking op. Before executing RES(op) = (p, res, (x, rlt)), c must receive a message of the form (req, T(op), rlt) by the algorithm and the definition of T. Let p ∈ objGroup(x) be any process that sends a message m of this form. By the algorithm, p executed REP_p(op) before sending m, and the value of upd_p[x] when p executed REP_p(op) was T(op).

If T(op) = 0 then, by the algorithm, p did not execute the replica operation for any update operation before executing REP_p(op), and REP_p(op) is executed when x_p is in its initial state. Furthermore, no update operation can precede op in ≺_{T_x} since for every update operation u, T(u) > 0. Thus, p executed REP_p(op) after executing the replica operation for every update operation that preceded op in ≺_{T_x}.

If T(op) ≠ 0 then, by Lemma 4.4.7, T(op) = T(u) for some update operation u on x. By the algorithm and the definition of T, p must have executed REP_p(u) before executing REP_p(op). By Lemma 4.4.8, p executed no other update replica operations between REP_p(u) and REP_p(op). Therefore, by Lemma 3.3.11, the sequence of update replica operations that was executed by p prior to executing REP_p(op) is REP_p(u_0), REP_p(u_1), ..., REP_p(u), where u_0, u_1, ..., u is the sequence of updates preceding op in ≺_{T_x}. Thus, the fourth condition holds for all read operations as well. □

Lemma 4.4.12 H|x is linearizable.
Proof: Follows from Lemmata 4.3.3 and 4.4.11. □

Theorem 4.4.1 $A_r$ is a linearizable consistency protocol for read/update object $x$.

Proof: Follows from Lemmata 4.4.2, 4.4.3 and 4.4.12. □

4.4.4 Performance Analysis

To analyze the performance of the algorithm in Figure 4.1 we will make the assumptions outlined in Section 2.2. Again we will let the latency of an operation be the time between the invocation and response of that operation. The latency and message cost of an update operation depend on the latency and message cost of the Atomic Multicast algorithm, while the latency and message cost of a read operation depend on the latency and message cost of the Strong Linear Multicast algorithm.

Let $d_A(G)$ be the time between the A-multicast of a message $m$ to a group $G$ and the A-delivery of $m$ by any process in $G$. The latency of an update operation on an object $x$ is therefore $d_A(objGroup(x)) + d_m$. Let $d_{SL}(G)$ be the time between the SL-multicast of any message $m$ to a group $G$ and the SL-delivery of $m$ by any process in $G$. Because of our timing assumptions, we can assume that all processes in $group(m)$ SL-deliver $m$ at the same time. (If this is not the case, then we will instead let $d_{SL}(G)$ represent the time between the SL-multicast of any message $m$ to $G$ and the SL-delivery of $m$ by at least $\beta(G) + 1$ processes in $G$.) The latency of a read operation on an object $x$ is then $d_{SL}(objGroup(x)) + d_m$.

Let $n_A(G)$ and $n_{SL}(G)$ be the number of messages required to A-multicast a message and SL-multicast a message to a group $G$ respectively. The number of messages required by an update operation on an object $x$ is then $n_A(objGroup(x)) + |objGroup(x)|$, and the number of messages required by a read operation on an object $x$ is $n_{SL}(objGroup(x)) + |objGroup(x)|$. 
Every client process $c$ executes the following:

To execute $\text{apply-read}(z, f)$:

\[
\text{req} \leftarrow \text{req} + 1
\]

$\text{WL-multicast\ (c, req, z, f)}$ to $\text{objGroup}(x)$

wait until $\left[\left(\left(\text{for } p \in \text{objGroup}(x) \text{ processes } p: \text{ received } (\text{req, upd, res}) \text{ from } p\right)\right) \text{ and } \left(\forall p \in \text{objGroup}(x): \text{ received } (\text{req, upd, res}) \text{ from } p \text{ or } p \in D_c\right)\]$

\[
\text{replies} \leftarrow \{ (\text{req, upd, res}) | \text{c received } (\text{req, upd, res}) \text{ from } p\}
\]

\[
\text{lr}g \leftarrow \text{largest upd such that } (\text{req, upd, res}) \in \text{replies}
\]

\[
\text{readRes} \leftarrow \text{select one res such that } (\text{req, lr}, \text{res}) \in \text{replies}
\]

return $\text{readRes}$

Every server process $p$ executes the following:

when $\text{WL-deliver}(c, \text{req, z, f})$:

\[
\text{res} \leftarrow \text{applyToReplica}(x, f)
\]

send $(\text{req, upd}[z], \text{res})$ to $c$

Figure 4.2: A Consistency Protocol for Read/Update Objects using Atomic Multicast and Weak Linear Multicast. Only the portions of the algorithm that differ from the algorithm in Figure 4.1 are shown.

4.5 A Consistency Protocol for Read/Update Objects using Weak Linear Multicast

A linearizable consistency protocol for read/update objects using Atomic Multicast and Weak Linear Multicast is shown in Figure 4.2. Like the algorithm in Figure 4.1, this algorithm is an enhancement of the general algorithm in Figure 3.3. Update operations are implemented in the same way as in the previous two algorithms and so that portion of the algorithm is not shown.

The implementation of $\text{apply-read}$ differs from the implementation of the same function in Figure 4.1 in that read requests are multicast using Weak Linear Multicast instead of Strong Linear Multicast, and instead of just waiting for $\beta(\text{objGroup}(x)) + 1$ replies to be received, the client also waits until it receives a reply from every process that is not suspected. The output of a client process $c$'s failure detector module is written as $D_c$;
therefore, $c$ waits until for every process $p \in objGroup(x)$, $c$ has received a reply from $p$ or $p \in D_c$. The result is chosen from the replies as before. The server processes handle the WL-delivery of a request in exactly the same way as they handled the SL-delivery of a request in Figure 4.1.

As in the previous algorithm, we will assume that any number of system-wide process failures can occur as long as there are at least $\beta(objGroup(x)) + 1$ correct processes in $objGroup(x)$, for any object $x$. Thus, the number of faulty processes in $objGroup(x)$ must be less than $|objGroup(x)| - \beta(objGroup(x))$.

Since the algorithm is so similar to the algorithm in Figure 4.1, we do not provide a detailed proof of correctness. We will, however, provide an informal explanation as to why client processes do not wait indefinitely for replies to read requests, and why the result that is eventually returned allows the read operation to be linearized.

Since all failure detectors that we are considering satisfy Strong Completeness, every faulty process will eventually be suspected by every process. We have already mentioned that there are at least $\beta(objGroup(x)) + 1$ correct processes in $objGroup(x)$, for any object $x$. A client process is therefore guaranteed to receive at least $\beta(objGroup(x)) + 1$ replies, and receive a reply from every process that is not suspected, for every request that it WL-multicasts. A client process will therefore never wait indefinitely for reply messages.

Since Weak Linear Multicast satisfies Weak Linear Order, if the number of processes that deliver a read request message out of linear order is more than $\beta(objGroup(x))$ then some correct process that is never suspected does not deliver that message out of linear order. A client that WL-multicast a read request waits until it has received at least $\beta(objGroup(x)) + 1$ replies, and has received a reply from every process in $objGroup(x)$ that is not suspected. Therefore, at least one of those reply messages is from a server process that executed the replica operations for all updates that returned before the read was invoked. The result that is returned by the client is selected from a reply message that was sent by such a server; a reply received from a server that WL-delivered the
request message for the operation in linear order will have a \textit{upd} value that is higher than the \textit{upd} value in a reply message sent by a server that WL-delivered the read request out of linear order. By choosing the result included in the message with the highest \textit{upd} value, the client ensures that it will return a value that allows the operation to be linearized.

The fault-tolerance and performance of this algorithm are identical to the fault-tolerance of the algorithm in Figure 4.1. The performance of the algorithm is identical to the performance of the algorithm in Figure 4.1 except that $d_{SL}$ and $n_{SL}$ represent the latency and message cost, respectively, of the Weak Linear Multicast algorithm.
Chapter 5

Implementing Atomic and Linear Multicasts

5.1 Implementing Atomic Multicast

The implementation of Atomic Multicast in asynchronous systems has been widely studied [5, 8, 9, 14, 17, 21, 32, 50, 51]. However, many of these algorithms implement weaker delivery semantics, e.g., instead of Local Delivery Consistency, a weaker ordering property based on group views\(^1\) is satisfied, or make stronger assumptions about the system than those we are considering [14, 17, 21, 50]. Of those algorithms that fit our definition of Atomic Multicast, most of these (implicitly) assume that Perfect failure detectors are available [5, 9, 32], or that failure detectors with even stronger semantics than Perfect failure detectors are available [51]. Since Perfect failure detectors are often too strong to be implemented in practise, we would like to consider algorithms that solve our definition of Atomic Multicast using weaker, unreliable failure detectors.

Chandra and Toueg [8] presented an algorithm that solves Atomic Broadcast using

\(^{1}\)Informally, a process p's view of a group G at time t denotes the set of non-crashed processes that p believes to be part of group G at time t. Views are used to express the semantics of Group Membership systems [5, 15, 42, 53]. Group Membership is discussed in Chapter 6.
Every process $p$ executes the following:

To execute $R$-multicast($G,m$):
- send $m$ to every process in $G$

$R$-deliver($m$) occurs as follows:
- when receive $m$ for the first time
  - if sender($m$) $\neq p$ then send $m$ to all processes in $G$

$R$-deliver($m$)

Figure 5.1: A Reliable Multicast algorithm based on a Reliable Broadcast algorithm given by Hadzilacos and Toueg [29] that was adapted from the “Diffusion Algorithm” of Christian, et al. [11].

Any Consensus algorithm. They also showed that Consensus can be solved using any of the failure detector classes considered in this thesis. We will therefore give a modification of this Atomic Broadcast algorithm that solves Atomic Multicast. This algorithm makes use of a version of Consensus, called Repeated Group Consensus. The Atomic Multicast algorithm assumes the existence of a Reliable Multicast algorithm as well. Thus, we will present a Reliable Multicast algorithm and examine the Repeated Group Consensus problem before providing our implementation of Atomic Multicast.

5.1.1 A Reliable Multicast Algorithm

The Reliable Multicast algorithm in Figure 5.1 is based on a Reliable Broadcast algorithm given by Hadzilacos and Toueg [29] that was adapted from the “Diffusion Algorithm” of Christian, et al. [11]. Informally, when a process $R$-multcasts a message $m$ it sends $m$ to all of the processes in $\text{group}(m)$. When a process in $G$ receives $m$ for the first time, it relays $m$ to all of the other processes in $G$ before $R$-delivering $m$. Thus, $m$ is guaranteed to be $R$-delivered by all correct processes in $G$ if any process in $G$ $R$-delivers $m$. The proof of the correctness of this algorithm is obvious and is therefore omitted.

Every message $m$ that is $R$-multicast by a correct process is eventually $R$-delivered
as long as no more than \(|\text{group}(m) - 1|\) processes are faulty; however, the properties of Reliable Multicast only require that \(m\) is R-delivered by the processes in \(\text{group}(m)\) if every process in \(\text{group}(m)\) is correct. Therefore, the algorithm is a \(\Gamma\)-tolerant solution to Reliable Multicast where \(\Gamma(G) = |G|\) for every group \(G \in \mathcal{G}\).

We can parameterize the Reliable Multicast problem by group, yielding a set of problems. For each group \(G\), Reliable Multicast for \(G\) can be defined by restricting the properties of Reliable Multicast to messages addressed to \(G\). The algorithm is a \(\Gamma_{\mathcal{G}}\)-tolerant solution to Reliable Multicast to \(G\), for any group \(G\), where \(\Gamma_{\mathcal{G}}(G') = |G'|\) for every \(G' \in \mathcal{G}\).

Each message that is R-multicast is delivered with a latency of \(d_m\). The R-multicasting a message \(m\) has a message cost of \(|\text{group}(m)|^2\) messages if \(\text{sender}(m) \notin \text{group}(m)\). The message cost is \(|\text{group}(m)|^2 - |\text{group}(m)|\) if \(\text{sender}(m) \in \text{group}(m)\).

### 5.1.2 The Repeated Group Consensus Problem

In the Consensus problem, all processes in the system must come to a unanimous decision on a value. This decision is irrevocable, and the decided value must have been proposed by one of the participating processes. The Fisher-Lynch-Patterson (FLP) impossibility result [20] states that Consensus cannot be solved by a deterministic algorithm in asynchronous systems subject to crash failures. Since the Consensus Problem and the Atomic Multicast problem are equivalent [8, 13], this result applies to Atomic Multicast as well.

The FLP result stems from the fact that in an asynchronous system, processes cannot tell the difference between a process that has crashed and one that is only slow to respond. To circumvent this problem, the addition of unreliable failure detectors to the asynchronous system model was proposed by Chandra and Toueg [8], who also showed that Consensus can be solved in asynchronous systems subject to crash failures, using any of the failure detectors considered in this thesis. Chandra et al. [7] showed that \(\mathcal{W}\), a failure detector that satisfies Eventual Weak Accuracy and a completeness property
that is weaker than Strong Completeness, is the weakest failure detector that allows Consensus to be solved in asynchronous systems. This failure detector, however, has been shown to be equivalent to \( \diamond S \), the weakest failure detector that we are considering in this thesis.

In this thesis, we are interested in a variation of the Consensus problem, which we will call the \textit{Group Consensus} problem, in which all processes \textit{in a group} must come to unanimous decision on a value. The Group Consensus problem is formally defined in terms of two primitives: \texttt{propose}(G,v), and \texttt{decide}(G,v). For both primitives, the argument \( G \) is any group, and the argument \( v \) is taken from a set \( V \) of possible decision values. When a process executes \texttt{propose}(G,v), we say that it \textit{proposes} \( v \) \textit{for group} \( G \). Similarly, we say that a process \textit{decides} \( v \) \textit{for group} \( G \) when it executes \texttt{decide}(G,v).

Group Consensus is defined by the following properties:

**Termination** For every group \( G \), every correct process in \( G \) eventually decides some value for \( G \).

**Uniform Integrity** For every group \( G \), every process decides for \( G \) at most once.

**Uniform Agreement** For every group \( G \), no two processes decide differently for \( G \).

**Uniform Validity** For every group \( G \), if a process \( p \) decides \( v \) for \( G \), then \( p \) is a member of \( G \) and \( v \) was previously proposed for \( G \) by some process in \( G \).

Any of the Consensus algorithms in the literature can be easily converted to Group Consensus algorithms. Where the Consensus algorithms send and receive messages to and from all of the processes in the system, the Group Consensus versions of these algorithms only send and receive messages to and from processes in a particular group. Furthermore, a process \( p \) will have multiple copies of the variables used by the Consensus algorithm, one for every group of which \( p \) is a member.

Like the Atomic Multicast problem, we can parameterize the Group Consensus problem so that it is instead a set of problems, one for each group in \( G \). Specifically, we define
the problem *Group Consensus for a group G* by restricting the properties of Group Consensus to $G$. For example, Uniform Agreement becomes: *No two processes decide differently for group G*.

If we take any Consensus algorithm that tolerates $f$ failures in a system of $n$ processes and transform it into a Group Consensus algorithm as described above, then that algorithm tolerates $f$ failures in any group of size $n$. More precisely, any Consensus algorithm that is an $f$-tolerant solution to Consensus (using a failure detector class $C$) in a system of $n$ processes, when transformed into a Group Consensus algorithm, is a $\Gamma_G$-toleration solution to Group Consensus for a group $G$ of $n$ processes (using $C$) where:

$$\forall G' \in G : \Gamma_G(G') = \begin{cases} f & \text{if } G' = G \\ \lvert G' \rvert & \text{otherwise.} \end{cases}$$

That same Group Consensus algorithm is then a $\Gamma$-tolerant solution to Group Consensus (using $C$) where $\Gamma(G) = \Gamma_G(G)$ for all $G \in G$.

Chandra and Toueg's Atomic Broadcast algorithm actually uses a variation of Consensus in which multiple executions of Consensus are performed. These executions are completely independent even though they may be executed concurrently. Our Atomic Multicast algorithm uses a similar variation of the Group Consensus problem. We shall refer to this variation of the Consensus problem as *Repeated Group Consensus*, and each execution of Group Consensus shall be referred to as a *round* of Group Consensus.

Like Group Consensus, Repeated Group Consensus is defined in terms of the *propose* and *decide* primitives, however a round number $k$ is added as an argument to these primitives in order to specify the round to which the execution of a primitive pertains. When a process executes *propose*($G, k, v$) we say that it *proposes* $v$ *for round $k$ of group $G*$, and when a process executes *decide*($G, k, v$), we say that it *decides* $v$ *for round $k$ of group $G*$. We can adapt the properties that define Group Consensus to Repeated Group Consensus as follows:

**Weak Termination** For every group $G$, if a correct process in $G$ proposes a value for
round \( k \) of \( G \) or any correct process in \( G \) decides a value for round \( k \) of \( G \), then every correct process in \( G \) eventually decides some value for round \( k \) of \( G \).

**Uniform Integrity** For every group \( G \), every process decides for round \( k \) of \( G \) at most once.

**Uniform Agreement** For every group \( G \), no two processes decide differently for round \( k \) of \( G \).

**Uniform Validity** For every group \( G \), if a process \( p \) decides \( v \) for round \( k \) of \( G \), then \( p \) is in \( G \) and \( v \) was previously proposed by some process in \( G \) for round \( k \) of \( G \).

The defining properties of Repeated Group Consensus are obvious adaptations from those of Group Consensus, except that the Termination property has been weakened. The Weak Termination property only requires that all correct processes reach a decision for a particular round if a correct process proposes or decides a value for that round. This means that we are allowing runs in which faulty processes propose and decide values for a round but no correct processes decide for that round.

A Group Consensus algorithm can be easily transformed into a Repeated Consensus algorithm by tagging each message pertaining to the \( k \)th round of Group Consensus with its round number \( k \). When a process receives a message it uses this tag to identify the round of Group Consensus to which the message pertains.

We can again parameterize the Repeated Group Consensus problem, to define *Repeated Group Consensus for a group* \( G \), in the obvious way. An algorithm that is a \( \Gamma_{G} \)-tolerant solution to Group Consensus for a group \( G \) (using a failure detector class \( C \)), when transformed into a Repeated Group Consensus algorithm as described above, is a \( \Gamma_{G} \)-tolerant solution to Repeated Group Consensus for \( G \) (using \( C \)). Similarly, an algorithm that is a \( \Gamma \)-tolerant solution to Group Consensus (using \( C \)) becomes a \( \Gamma \)-tolerant solution to Repeated Group Consensus (using \( C \)) under this same transformation.
Every process \( p \) executes the following:

Initialization:

\[
\begin{align*}
R_{\text{delivered}} & \leftarrow 0, 0, \ldots, 0 \\
A_{\text{delivered}} & \leftarrow 0, 0, \ldots, 0 \\
k & \leftarrow 0, 0, \ldots, 0
\end{align*}
\]

To execute \( A\text{-multicast}(G, m) \):

\[
R\text{-multicast}(G, m)
\]

\( A\text{-deliver}(\cdot) \) occurs as follows:

\[
\text{when } R\text{-deliver}(m) \quad \{ \text{Task 1} \}
\]

\[
R\text{delivered}[\text{group}(m)] \leftarrow R\text{delivered}[\text{group}(m)] \cup \{ m \}
\]

\[
\text{when } \text{for any group } G: R\text{delivered}[G] - A\text{delivered}[G] \neq \emptyset \quad \{ \text{Task 2} \}
\]

\[
k[G] \leftarrow k[G] + 1 \\
A_{\text{undelivered}}[G] \leftarrow R\text{delivered}[G] - A\text{delivered}[G] \\
\text{propose}(G, k[G], A_{\text{undelivered}}[G]) \\
\text{wait until } \text{decide}(G, k[G], \text{msgSet}_k[G][G]) \\
A_{\text{deliver}}^{k[G]}[G] \leftarrow \text{msgSet}_k[G][G] - A_{\text{delivered}}[G] \\
A\text{ deliver all messages in } A_{\text{deliver}}^{k[G]}[G] \text{ in some deterministic order} \\
A_{\text{delivered}}[G] \leftarrow A_{\text{delivered}}[G] \cup A_{\text{deliver}}^{k[G]}[G]
\]

Figure 5.2: An Atomic Multicast algorithm based on the Chandra and Toueg Atomic Broadcast algorithm [8].

### 5.1.3 An Atomic Multicast Algorithm

The Atomic Multicast algorithm is presented in Figure 5.2. This algorithm is an adaptation of Chandra and Toueg's Atomic Broadcast algorithm [8]. Every variable has been replaced with an array of the same name that is indexed by a group. The Reliable Broadcast and Repeated Consensus algorithms have also been replaced with Reliable Multicast and Repeated Group Consensus algorithms respectively.

To \( A\text{-multicast} \) a message \( m \) to a group \( G \), a process \( R\text{-multicasts} \) \( m \) to all of the processes in \( G \). When a process \( p \in G \) R-delivers \( m \), \( p \) adds \( m \) to \( R_{\text{delivered}}_p[G] \) – the set of messages addressed to \( G \) that \( p \) has R-delivered. Process \( p \) also maintains a set \( A_{\text{delivered}}_p[G] \) of the messages addressed to \( G \) that it has A-delivered. The processes in
G use Repeated Group Consensus to agree upon batches of messages to be A-delivered; round $k$ of Repeated Group Consensus for $G$ determines the $k$th batch of messages to be A-delivered by processes in $G$.

The proposal, decision and delivery of each batch of messages occurs in Task 3 of the algorithm. When a process $p \in G$ executes Task 3, it calculates a set $A\text{-undelivered}_p[G]$ of messages addressed to $G$ that it has R-delivered but not A-delivered, and then proposes this set for the next, say the $i$th, round of Repeated Group Consensus for $G$. $p$ then waits until round $i$ for $G$ has been decided, and uses the set of decided messages $msgSet^i_p[G]$ to calculate $A\text{-deliver}^i_p[G]$ — the $i$th batch of messages addressed to $G$ to be delivered by $p$. Specifically, $A\text{-deliver}^i_p[G] = msgSet^i_p[G] - A\text{-delivered}_p[G]$. The messages in $A\text{-deliver}^i_p[G]$ are delivered in some deterministic order, e.g., a lexicographic order based on message ids. The same deterministic ordering is used by all of the processes in $G$. As long as $p$ has R-delivered messages addressed to $G$ that it has not A-delivered (i.e., $R\text{-delivered}_p[G] - A\text{-delivered}[G] \neq \emptyset$), $p$ executes Task 3.

### 5.1.4 Fault-Tolerance

The fault-tolerance of this algorithm depends on the fault-tolerance of the Reliable Multicast and Repeated Group Consensus algorithms that are used. However, if we assume that algorithm in Figure 5.1 is used, then since this Reliable Multicast algorithm tolerates any number of failures, it does not affect the fault-tolerance of the Atomic Multicast algorithm. We will assume that the Repeated Group Consensus algorithm used is a $\Gamma'$-tolerant solution to Repeated Group Consensus using a failure detector class $C$. The algorithm in Figure 5.2 is therefore a $\Gamma$-tolerant solution to Atomic Multicast using $C$ where $\Gamma(G) = \Gamma'(G)$ for all $G \in \mathcal{G}$.

However, in Subsection 3.3.2, we discussed parameterizing the Atomic Multicast problem by groups to allow for a more “modular” analysis of an algorithm’s fault-tolerance. Specifically, we desire to know the fault-tolerance of the algorithm in solving Atomic Mul-
multicast for a specific group. If we assume that the Repeated Group Consensus algorithm used is a $\Gamma_{G_i}$-tolerant solution to Repeated Group Consensus for a group $G_i$ where $G_i$ is any group in $G$, then the algorithm in Figure 5.2 is a $\Gamma_{G}$-tolerant solution to Atomic Multicast for $G$ where:

$$\forall G' \in G : \Gamma_{G}(G') = \begin{cases} 
\Gamma'_{G}(G') & \text{if } G' = G \\
|G'| & \text{otherwise}
\end{cases} \quad (5.2)$$

5.1.5 Proof of Correctness

We now present a proof of correctness for the Atomic Multicast algorithm presented in Figure 5.2. This proof is adapted from the proof of correctness of Chandra and Toueg's Atomic Broadcast algorithm [8] and so the proofs of the individual theorems and lemmata have been moved to Appendix A.

**Lemma 5.1.1** For any group $G$, any process $p \in G$, any correct process $q \in G$, and any message $m$ such that $\text{group}(m) = G$, if $m \in R\_\text{delivered}_p[G]$ then eventually $m \in R\_\text{delivered}_q[G]$.

**Lemma 5.1.2** For any group $G$, any two processes $p \in G$ and $q \in G$, and all $k \geq 1$, if $p$ A-delivers messages in $A\_\text{deliver}^k_p[G]$ and $q$ A-delivers messages in $A\_\text{deliver}^k_q[G]$ then $A\_\text{deliver}^k_q[G] = A\_\text{deliver}^k_p[G]$.

**Lemma 5.1.3** For any group $G$, any process $p \in G$ and correct process $q \in G$, and all $k \geq 1$:

1. If $p$ executes $\text{propose}(G, k, -)$, then $q$ eventually executes $\text{propose}(G, k, -)$.

2. If $p$ A-delivers messages in $A\_\text{deliver}^k_p[G]$ then $q$ eventually A-delivers messages in $A\_\text{deliver}^k_q[G]$.

**Lemma 5.1.4** The algorithm in Figure 5.2 satisfies the Uniform Agreement and Local Delivery Consistency properties of Atomic multicast.
**Lemma 5.1.5 (Validity)** If a correct process $A$-multcasts a message $m$, then some correct process in $\text{group}(m)$ eventually delivers $m$, or no process in that group is correct.

**Lemma 5.1.6 (Uniform integrity)** For any message $m$, every process $p$ $A$-delivers $m$ at most once, and only if $p$ is in $\text{group}(m)$ and $m$ was previously $A$-multicast by $\text{sender}(m)$.

**Theorem 5.1.1** The algorithm in Figure 5.2 is a solution to Atomic Multicast.

### 5.1.6 Performance Analysis

To analyze the performance of the Atomic Multicast algorithm we will make the assumptions outlined in Section 2.2. In addition, we will assume that the algorithm in Figure 5.1 is being used to solve Reliable Multicast. The performance of the Atomic Multicast algorithm depends on the performance of this Reliable Multicast algorithm as well as on the performance of the Repeated Group Consensus algorithm that is used. Let $n_C(G)$ be the number of messages required to execute one round of Group Consensus for a group $G$. (Note that we will assume that both the message cost and the latency of a round of Repeated Group Consensus for a group $G$ depends on $G$.) The number of messages required to $A$-multicast a message $m$ is therefore $|\text{group}(m)|^2 + n_C(G)$.

The latency required to $A$-multicast a message depends on the number of rounds of Consensus that are executed before a message is decided on. Since we are assuming that all messages that are sent have a fixed delay time of $d_m$, every process in $\text{group}(m)$ will R-deliver a message $m$ at the same time. We will therefore assume that after R-delivering $m$, every process in $\text{group}(m)$ will include $m$ in the set of messages that is proposed for the next round of Group Consensus. Therefore, if a round of Group Consensus is in progress when $m$ is R-delivered, two rounds of Group Consensus must be executed before $m$ is A-delivered. If a round of Group Consensus is not already in progress then one round
of Group Consensus must be executed before \( m \) is A-delivered. Let \( d_C(G) \) to denote the time between the proposal of a round of Repeated Group Consensus for a group \( G \) and the decision by any process in \( G \) for that round. Thus the best case latency for a message \( m \) to be A-delivered, i.e., the time between the A-multicast of \( m \), and the A-delivery of \( m \) by any process is \( d_C(\text{group}(m)) + d_m \) and the worst case latency is \( 2d_C(\text{group}(m)) + d_m \).

### 5.2 Implementing Linear Multicast

Various Atomic Multicast algorithms in the literature [5, 9, 32] ensure that a message \( m \) is properly A-delivered despite \( f \) failures in \( \text{group}(m) \) by propagating either \( m \) or a copy of \( m \)'s message id to \( f+1 \) processes in \( \text{group}(m) \) before any process A-delivers \( m \).

For example, the Atomic Multicast algorithm presented by Kaashoek and Tanenbaum [32] uses a special sequencer process to order the messages addressed to a particular group. To A-multicast a message \( m \) to a group \( G \), \( m \) is first sent to the sequencer process for \( G \). The sequencer process assigns a sequence number to each message that it receives, and then relays the messages (and their assigned sequence numbers) to the other processes in \( G \). Upon receiving such a sequenced message \( m \), the processes in \( G \) send an acknowledgement message back to the sequencer process indicating that \( m \) has been received, and do not A-deliver \( m \) until an accept message for \( m \) has been received back from the sequencer process. In order for the algorithm to tolerate \( f \) failures in \( G \), the sequencer process waits until \( f \) processes have acknowledged each message \( m \) before sending accept messages for \( m \) to the processes in \( G \). Thus, \( f+1 \) processes (including the sequencer process) in \( G \) have received a copy of a message \( m \) before any process in \( G \) A-delivers \( m \).

Kaashoek and Tanenbaum also describe a variation of the algorithm that is more efficient in networks that support hardware multicast. In this algorithm, a message \( m \) that is A-multicast to a group \( G \) is initially sent to all of the processes in \( G \) instead
of only being sent to the sequencer process for $G$. Then, when the sequencer process assigns a sequence number to $m$, it only relays $m$'s message id and sequence number to the processes in $G$. Thus, $m$ need only appear on the network once, while if the original algorithm was used, the message would appear on the network twice.

The rest of the algorithm proceeds as in the original algorithm; processes acknowledge the receipt of $m$'s id and sequence number, and wait until $m$ is accepted. A process $p$ delivers $m$ when $p$ has received a copy $m$ and an accept message for $m$. Again, the sequencer process waits until $f$ processes have acknowledged each message $m$ before sending accept messages for $m$; however, in this case, $f + 1$ processes are guaranteed to have a copy of $m$'s message id, rather than actual copy of $m$, before any process in $G$ A-delivers $m$.

Specifically, the Atomic Multicast algorithms that we are considering satisfy the following properties:

1. Before any process A-delivers a message $m$, at least $f + 1$ processes in $\text{group}(m)$ have either received a copy of $m$ or some message containing $m$'s message id $(\text{sender}(m), \text{seqno}(m))$.

2. If a correct process $p$ in $\text{group}(m)$ has received a copy of a message $m$ (or $m$'s message id) then $p$ will eventually A-deliver $m$.

If a fault-tolerant Atomic Multicast algorithm satisfies the above properties, then Linear Multicast can be efficiently implemented as follows:

- To L-multicast a message $m$, a process sends $m$ to every process in $\text{group}(m)$.

- When a process $p$ in $\text{group}(m)$ receives $m$ at time $t$, $p$ L-delivers $m$ as soon as $p$ has A-delivered every message $m_0$ that $p$ knows was A-multicast prior to $t$ (i.e., $p$ received a copy of $m_0$ or $m_0$'s message id at time $t' \leq t$).

Trivially, this algorithm satisfies Uniform Integrity. This algorithm also satisfies Validity and Weak Uniform Agreement since if the sender of some L-multicast message $m$ is
correct, every correct process in \( \text{group}(m) \) will eventually receive \( m \). If a correct process \( q \) in \( \text{group}(m) \) receives \( m \), it will L-deliver \( m \) only if it has A-delivered every message it “knew about” at time \( t \). By Property 2, \( q \) will eventually A-deliver all such messages.

The algorithm also satisfies Strong Linear Order, where \( \beta(G) = |G| - f - 1 \) for all \( G \in \mathcal{G} \). Let \( m \) be any message that is L-multicast at time \( t \), and let \( \mathcal{M}_0 \) be the set of messages that were A-delivered by any process before time \( t \). By Local Delivery Consistency, if \( \mathcal{M}_0 \neq \emptyset \) then there is a message \( m' \) that is A-delivered by a process \( p \) only after \( p \) has previously delivered all of the other messages in \( \mathcal{M}_0 \). By Property 1, \( f + 1 \) processes in \( \text{group}(m) \) have a copy of \( m' \) (or its message id) at time \( t \) and will therefore not L-deliver \( m \) until all of the messages in \( \mathcal{M}_0 \) have been A-delivered. Therefore, no more than \( |\text{group}(m)| - f - 1 \) processes will L-deliver \( m \) out of linear order.

This algorithm is very efficient since it only uses \( |\text{group}(m)| \) messages to L-multicast a message \( m \). Also, the algorithm allows a message to be L-delivered with a best case latency of \( d_m \). Furthermore, after receiving a message \( m \) that has been L-multicast, a process “waits” until it has A-delivered a set of messages before L-delivering \( m \). If we make the reasonable assumption that if \( m \) was instead A-multicast, \( m \) would not be ordered before any of the messages in this “waited-on” set, then the latency of L-multicasting a message can be no worse than the latency that would be incurred if we instead A-multicast that same message.

Thus, we can cheaply add Linear Multicasts to many Atomic Multicast algorithms by making use of methods used by those algorithms to ensure fault-tolerance. However, the algorithms in the literature that satisfy the properties that enable our Linear Multicast algorithm to be implemented usually assume the existence of Perfect Failure detectors. We would like to determine if Linear Multicasts can be implemented efficiently using any of the failure detector classes that we are considering in this thesis.

We can answer this question if we can show that the Atomic Multicast algorithm in Figure 5.2 can be enhanced so that it also cheaply implements a Linear Multicast. This
algorithm satisfies a property similar to Property 2; namely, if a correct process proposes a set of messages for a round of Consensus, it will eventually decide for that round and A-deliver all of the messages in the decided set. However, to implement Linear Multicast we must show that the Consensus algorithm that is used satisfies a property similar to Property 1.

Chandra and Toueg [8] have presented an algorithm that solves Consensus using $\diamond S$ that can easily be converted into a $\Gamma$-tolerant solution to Repeated Group Consensus for a group $G$ where $\Gamma(G) = \lceil \frac{|G|-1}{2} \rceil$ and $\Gamma(G') = |G'|$ for all $G' \neq G$. This algorithm, called the "Rotating Coordinator algorithm", has the property that $\Gamma(G) + 1$ processes in $G$ must receive an "estimate" message for a round of Consensus from a special "coordinator" process before any process decides for that round. Without going into the specific details of this algorithm, it should be easy to see that we can use this property to construct a Strong Linear Multicast using the methodology described above.

This property, however, could be viewed as a short-coming of the Rotating Coordinator algorithm. A process cannot decide for a round of Repeated Group Consensus for a group $G$ without (directly or indirectly) communicating with at least $\Gamma(G) + 1$ other processes in $G$ – an unfavorable property in the presence of communication delays. This raises the question of whether a Repeated Group Consensus algorithm exists that does not suffer from this deficiency; and if so, does it satisfy some other property that would allow a Linear Multicast to be implemented cheaply? The rest of this chapter is devoted to showing that an efficient implementation of Linear Multicast can be added to the Atomic Multicast algorithm in Figure 5.2, as long as the employed Repeated Group Consensus algorithm uses any of the failure detectors considered in this thesis.

### 5.2.1 A Causal Analysis of Consensus

In runs of Repeated Group Consensus algorithms, proposals and decisions are modeled by proposal events and decision events respectively. Each proposal event is an internal
event of the form \((p, \text{propose}, (G, k, v))\) where \(p\) is the process making the proposal, \(v\) is the value proposed, and \(k\) and \(G\) are the round and group, respectively, for which \(v\) is proposed. Each decision event is an internal event of the form \((p, \text{decide}, (G, k, v))\) where \(p\) is the process making the decision, \(v\) is the value decided, and \(k\) and \(G\) are the round and group, respectively, for which \(v\) is decided. We say that a process \(p\) has a proposal of \(v\) for round \(k\) of a group \(G\) in its causal past at time \(t\) in a run \(R\) if \(p\) has a proposal event \((p, \text{propose}, (G, k, v))\) in its causal past at time \(t\) in \(R\).

In this section we present a theorem that states that in order for any process to decide a value \(v\) for a round of Repeated Group Consensus, some correct process must have a proposal of \(v\) for that round in its causal past. The proof of this theorem makes the assumption that a Repeated Group Consensus algorithm has no way of predicting when a process will propose a value for a round of Consensus, and what value a process will propose. Specifically, we will assume that for every Repeated Group Consensus algorithm, the following property holds:

**Proposal Independence** If \(v \in V\) is any value and \(R\) is a run in which a process \(p\) in a group \(G\) does not have a proposal of \(v\) for round \(k\) of \(G\) in its causal past at time \(t\), then there exists a run \(R'\) that is identical to \(R\) until time \(t\) in which \(p\) proposes \(v'\) for round \(k\) of \(G\) at time \(t + 1\) where \(v'\) is any value in \(V\) such that \(v' \neq v\).

This property captures the idea that the Consensus algorithm is "modular" enough to be used by any other algorithm as a subroutine. Note that Proposal Independence does not prevent a process that has already proposed a value for a round \(k\) of Repeated Group Consensus from influencing another process' proposal for round \(k\).

Given any partial run \(R = (\mathcal{F}, \mathcal{H}, I, S, T)\) of a Repeated Group Consensus algorithm \(A\), we define \(\text{remove}(A, R, G, k, v)\) to be the partial run of \(A\) resulting from the causal removal (see Subsection 2.1.9) of every proposal of \(v\) for round \(k\) of group \(G\) from \(R\).

We now show that if a process decides a value for a round \(k\) of Repeated Group Consensus for a group \(G\), it must have a proposal of that value for round \(k\) of \(G\) in its
Chapter 5. Implementing Atomic and Linear Multicasts

Lemma 5.2.1 Given any Repeated Group Consensus algorithm A, if a process p in a group G decides v for round k of G at time t, then p has a proposal of v for round k of G in its causal past at time t.

Proof: Assume by way of contradiction that a run R of A exists in which at time t, a process p decides v for round k of G but does not have a proposal of v for round k of G in its causal past. By Uniform Validity, some process must have proposed v for round k of G at some time \( t_m \leq t \) in R. Since p does not have a proposal of v for round k of G in its causal past. We now show that the existence of run R allows us to construct a run of A that violates Uniform Validity.

Let \( R' \) be the shortest partial run that is a prefix of R and contains the execution of \( \text{decide}(G, k, v) \) by p at time t and let \( R' \) be any extension of \( \text{remove}(A, R', G, k, v) \). By the definition of \( \text{remove} \), p executes the same sequence of steps in \( \text{remove}(A, R', G, k, v) \) as in \( R' \) and decides v for round k of G at time t. By Uniform Validity, some process proposed v for round k of G at some time \( t' \leq t \) in \( R' \) which is a contradiction by the definition of \( R' \). □

Let \( \text{inPast}\_{G,k,v}(p, R, t) \) denote the following process execution predicate: “process p has a proposal of v for round k of group G in its causal past at time t in run R.”

Let \( \text{decided}\_{G,k,v}(R, t) \) denote the following execution predicate: “some process decided v for round k of group G at time t in run R.” Note that both \( \text{inPast}\_{G,k,v} \) and \( \text{delivered}_{G,k,v} \) only depend on the past.

Theorem 5.2.1 Let A be a \( \Gamma \)-tolerant solution to Repeated Group Consensus for a group G using some failure detector D, and let \( \mathcal{R} \) be the system formed by A and \( \Gamma \). Further assume that \( \mathcal{R} = (\mathcal{F}, \mathcal{H}_D, I, S, T) \) is a run in \( \mathcal{R} \) using D in which some process in G is correct. If a process decides a value v for round k of G at time t in \( \mathcal{R} \) then some
correct process has a proposal of \( v \) for round \( k \) of \( G \) in its causal past at time \( t \) in \( R \).

\[
(\text{decided}_{G,k,v}(R, t) \implies \exists p \in \text{correct}(R) : \text{inPast}_{G,k,v}(p, R, t)).
\]

**Proof:** Let \( R = (F, H_D, I, S, T) \) be any run in \( R \) using \( D \) in which some process in \( G \) is correct. Assume, by way of contradiction, that a process \( p \) decides \( v \) for round \( k \) of group \( G \) at time \( t \), and no correct process has a proposal of \( v \) for round \( k \) of group \( G \) in its causal past at time \( t \); or more formally: \( \text{decided}_{G,k,v}(R, t) \) and \( \forall q \in \text{satisfy}(\text{inPast}_{G,k,v}, R, t) : q \in \text{faulty}(F) \). By Lemma 5.2.1, \( \text{inPast}_{G,v,k}(p, R, t) \) holds; therefore, \( p \) is faulty in \( R \).

Let \( R' \) be the shortest partial run that is a prefix of \( R \) and contains the execution of \( \text{decide}(G, k, v) \) by \( p \), i.e., the last event in \( R' \) is executed at time \( t \). By the Uniform Validity property of Repeated Group Consensus, \( R' \) also contains at least one proposal of \( v \) for round \( k \) of \( G \).

Let \( R' = (F, H_D, I, S', T') \) be any run of \( A \) that is an extension of the partial run \( \text{remove}(A, R', G, k, v) \) after \( t \) in which no process in \( \text{faulty}(F) \) executes any events after \( t \). Further assume that the first event executed by each correct process after \( t \) is the proposal of any value other that \( v \) for round \( k \) of \( G \). This is possible by the Proposal Independence property.

Since both the failure pattern and the failure detector history of \( R' \) are identical to those of \( R \), \( R' \) is in \( R \) and uses \( D \), and must therefore satisfy the properties of Repeated Group Consensus. Thus, by Weak Termination and Uniform Validity, every correct process in \( G \) eventually decides some value \( v' \neq v \) for round \( k \) of \( G \). Let \( t' \) be the latest time at which a correct process in \( G \) decides for round \( k \) of \( G \) in \( R' \).

Let \( R'' = (F, H_D, I, S', T'') \) be an extension of \( R' \) after \( t \) in which no process in \( \text{faulty}(F) \) executes any events after \( t \). Furthermore, every process in \( \text{correct}(F) \) executes the same sequence of events as in \( R' \), and executes those events at the same time as in \( R' \), until \( t'' \). This is possible since no correct process has a proposal of \( v \) for round \( k \) of \( G \) in its causal past at time \( t \) so correct processes execute the same events in both \( R' \) and \( \text{remove}(A, R', G, k, v) \); and since the system is asynchronous, messages to correct
processes in $R'$ that were not sent in $\text{remove}(A, R', G, k, v)$ may be delayed until after $t''$ in $R''$.

As with $R'$, since $R''$ has the same failure pattern and failure detector history as $R$, $R''$ is in $\mathcal{R}$ and uses $\mathcal{D}$, and therefore must satisfy the properties of Repeated Group Consensus. Since all correct processes execute the same events in $R''$ as in $R'$ until $t$, all correct processes decide $v'$ for round $k$ of $G$. However, since $R''$ is an extension of $R'$ after $t$, $p$ decides $v$ for round $k$ of $G$ at time $t$ in $R''$ which violates Uniform Agreement.

We now present a corollary that follows from the above theorem and Theorem 2.3.1. This is the result that we will use to show that a Linear Multicast can be implemented efficiently from a Consensus algorithm that uses any failure detector class that includes $\diamond \mathcal{P}$.

**Corollary 5.2.1** Let $A$ be any algorithm that is an $\Gamma$-tolerant solution to Repeated Group Consensus for a group $G$ using $\diamond \mathcal{P}$, where $\forall G' \in G : G' \neq G \implies \Gamma(G') = |G'|$, let $\mathcal{R}$ be the system formed by $A$ and $\mathcal{I}$, and let $\mathcal{D}$ be a failure detector such that any run in $\mathcal{R}$ that uses $\mathcal{D}$ satisfies the properties of Repeated Group Consensus for $G$. Further assume that $R = \langle \mathcal{F}, \mathcal{H}, I, S, T \rangle$ is any run in $\mathcal{R}$ using $\mathcal{D}$. If a process $p \in G$ executes $\text{decide}(G, k, v)$ for some value $v$ at time $t$ in $R$ then $\min(\Gamma(G) - \mathcal{F}(t) + 1, |G|)$ processes in $G$ have a proposal of $v$ for round $k$ of $G$ in their causal past at time $t$ in $R$.

**Proof:** First, we will show that the theorem holds if we assume that $\Gamma(G) < |G|$. This implies that for every run in $\mathcal{R}$ there is a correct process in $G$; thus, by Theorem 5.2.1, for every run $R$ in $\mathcal{R}$ using $\mathcal{D}$ and any time $t$, if $\text{decided}_{G,k,v}(R, t)$ holds then there exists a correct process $p$ in $R$ for which $\text{inPast}_{G,k,v}(p, R, t)$ holds. Therefore, the result follows from Theorem 2.3.1.

Now, assume that $\Gamma(G) = |G|$. We have already shown that the result holds for every run in $\mathcal{R}$ using $\mathcal{D}$ for which there is a correct process in $G$. We must show that it also
holds for any run in \( \mathcal{R} \) using \( \mathcal{D} \) where all processes in \( G \) are faulty.

Let \( R = (\mathcal{F}, \mathcal{H}, I, S, T) \) be a run in \( \mathcal{R} \) in which every process in \( G \) is faulty, let \( p \) be the last process in \( G \) to fail in \( \mathcal{F} \), and let \( t \) be the time at which \( p \) fails in \( \mathcal{F} \). Assume by way of contradiction that some process in \( G \) decides some value \( v \) for some round \( k \) of Repeated Group Consensus for \( G \) before all processes in \( G \) have either crashed or have a proposal of \( v \) for round \( k \) of \( G \) in their causal past.

Let \( \mathcal{F}' \) be a failure pattern such that \( \forall t' < t : \mathcal{F}'(t') = \mathcal{F}(t') \), and \( \text{correct}(\mathcal{F}') = \{ p \} \cup \text{correct}(\mathcal{F}) \). By this definition, \( \forall G \in G : |\text{faulty}(\mathcal{F}') \cap G | \leq \Gamma(G) \). By Lemma 2.3.1, there exists a failure detector \( \mathcal{D}' \in \Diamond \mathcal{P} \) and a failure detector history \( \mathcal{H}' \) such that \( \forall t' < t, \forall p \in \Pi - \mathcal{F}'(t') : \mathcal{H}'(p, t') = \mathcal{H}(p, t') \) and \( \mathcal{H}' \in \mathcal{D}'(\mathcal{F}') \). Thus, by Lemma 2.3.4, there exists a run \( R' = (\mathcal{F}', \mathcal{H}', I', S', T') \) that is identical to \( R \) until time \( t - 1 \). Since every process in \( G \) crashed by time \( t \) in \( R \), every process in \( G \) executes the same events in \( R' \) as in \( R \). Therefore, at some time in \( R' \) some process in \( G \) decides \( v \) for round \( k \) of \( G \) before all processes in \( G \) have either crashed of have a proposal of \( v \) for round \( k \) of \( G \) in their causal past. This is a contradiction, however, since we have already shown that our result holds for all runs in which a process in \( G \) is correct. \( \square \)

Similar corollaries for Consensus algorithms using Strong and Perfect failure detectors follow from Theorems 2.3.2 and 2.3.3 respectively. The proofs of these corollaries are omitted due to their similarity to the proof of Corollary 5.2.1.

**Corollary 5.2.2** Let \( A \) be any algorithm that is a \( \Gamma \)-tolerant solution to Repeated Group Consensus for a group \( G \) using \( \mathcal{S} \) where \( \forall G' \in G : G' \neq G \rightarrow \Gamma(G') = |G'| \), and let \( \mathcal{R} \) be the system formed by \( A \) and \( \Gamma \). Further assume that \( R = (\mathcal{F}, \mathcal{H}_D, I, S, T) \) is a run in \( \mathcal{R} \) using a failure detector \( \mathcal{D} \in \mathcal{S} \). If a process \( p \in G \) executes \( \text{decide}(G, k, v) \) for some value \( v \) at time \( t \) in \( R \) then \( \min(\Gamma(G) - \mathcal{F}(t) + 1, |G|) \) processes in \( G \) have a proposal for round \( k \) of \( G \) in their causal past at time \( t \) in \( R \) or some process \( p \in G \) has a proposal for round \( k \) of \( G \) in its causal past at time \( t \) in \( R \) where either \( p \) is correct and is never
suspected in $R$, or if all process in $G$ are faulty, then $p$ is the last process in $G$ to crash and is never suspected before crashing in $R$.

Corollary 5.2.3 Let $A$ be any algorithm that is a $\Gamma$-tolerant solution to Repeated Group Consensus for a group $G$ using $P$ where $\forall G' \in G : G' \neq G \implies \Gamma(G') = |G'|$, and let $R$ be the system formed by $A$ and $\Gamma$. Further assume that $R = \langle F, H_D, I, S, T \rangle$ is a run $A$ using a failure detector $D \in P$. If a process $p \in G$ executes $\text{decide}(G, k, v)$ for some value $v$ at time $t$ in $R$ then $\min(\Gamma(G) - F(t) + 1, |G|)$ processes in $G$ have a proposal for round $k$ of $G$ in their causal pasts at time $t$ in $R$.

5.2.2 Keeping Track of Consensus Rounds

The results of the previous section define constraints on the set of processes in a group that have a proposal for a round of Repeated Group Consensus in their causal pasts when any process in the group decides for that round. In order to make use of this fact to implement Linear Multicasts, we must make it possible for a process to “know” that it has a proposal in its causal past. This can be done by running a Round Tracking algorithm concurrently with any Repeated Group Consensus algorithm.

We say that a process $p$ is aware of a round $k$ of Repeated Group Consensus for group $G$ if $p$ has a proposal for round $k$ of Repeated Group Consensus for $G$ in its causal past. A solution to the Round Tracking problem gives each process $p$ in a group $G$ access to a variable $\text{round}_p[G]$ that identifies the latest round that $p$ is aware of. Formally, Round Tracking is defined by the following properties:

Validity If a process $p$ that is a member of a group $G$ is aware of round $k$ of Repeated Group Consensus for $G$ then $\text{round}_p[G] \geq k$.

Integrity If $\text{round}_p[G] > 0$ for a process $p$ and a group $G$, then some process in $G$ previously proposed for round $\text{round}_p[G]$ of group $G$. 
Monotonicity For every process \( p \) and every group \( G \), \( \text{round}_p[G] \) is monotonic non-decreasing.

A simple algorithm that solves Round Tracking can be implemented by making each process in the system keep track of the Consensus rounds of each group in the system. Each process \( p \in \Pi \) maintains an array \( \text{round}_p[G] \) indexed by \( G \). Each process \( p \) initializes \( \text{round}_p \) to \( \{0, 0, \ldots, 0\} \). Whenever a process \( p \) executes \( \text{propose}(G, k, -) \) it sets \( \text{round}_p[G] \) to \( k \). Each process \( p \) piggybacks \( \text{round}_p \) on each message it sends, and when a process \( q \) receives a message from \( p \) it sets \( \text{round}_q \) to the componentwise maximum of \( \text{round}_p \) and \( \text{round}_q \). It is not hard to see that this algorithm satisfies the Validity, Integrity and Monotonicity properties of Round Tracking.

The above algorithm is not very efficient if there is a large number of groups in the system since each process must maintain an array of size \( |G| \) and the size of every message in the system is increased by \( |G| \). Fortunately, if processes executing a round of Repeated Group Consensus for a group \( G \) only send and receive messages to and from other processes in \( G \) then the algorithm can be modified so that these storage space and message size requirements are reduced.

We say that a Repeated Group Consensus algorithm \( A \) is group closed for a group \( G \) if for every run \( R = \langle \mathcal{F}, \mathcal{H}, I, S, T \rangle \) of \( A \) there exists another run \( R' = \langle \mathcal{F}, \mathcal{H}, I, S', T' \rangle \) of \( A \) such that every process in \( G \) executes in \( S' \) a subsequence of the events it executed in \( S \) that includes every proposal and decision for every round of \( G \), and excludes the sending and receiving of messages that are not both sent by, and addressed to, processes in \( G \). We say that a Repeated Group Consensus algorithm is group-closed if it is group-closed for every group in \( G \).

If a process \( p \) is aware of a proposal at a time \( t \) then there is a chain of events from that proposal to \( p \) at time \( t \). The results of the previous section place constraints on the set of processes that are aware of a proposal of a value \( v \) for a round \( k \) of Repeated Group Consensus for a group \( G \) in the event that some process decides \( v \) for round \( k \).
of \( G \). If a Repeated Group Consensus algorithm is group-closed then these results are strengthened so that there is a chain of events through \( G \) from a proposal of \( v \) to any process that is aware of that proposal.

For instance, consider a group-closed algorithm \( A \) that is a \( \Gamma \)-tolerant solution to Repeated Group Consensus using \( \Diamond \mathcal{P} \), and let \( R \) be a run in the system formed by \( A \) and \( \Gamma \) that uses a failure detector \( D \in \Diamond \mathcal{P} \) in which a process \( p \) decides \( v \) for a round \( k \) of Repeated Group Consensus for a group \( G \) at time \( t \). By Corollary 5.2.1, \( \min(\Gamma(G)+1, |G|) \) processes have crashed or are aware of a proposal of \( v \) for round \( k \) of \( G \). Since \( A \) is group-closed, there exists a run \( R' \) in which the processes in \( G \) execute a subsequence of the events executed in \( R \) that includes the decision and proposal events for every round of \( G \) and excludes the sending and receiving of messages to and from processes outside of \( G \). Thus, \( p \) also decides at time \( t \) in \( R' \), again implying that \( \min(\Gamma(G)+1, |G|) \) processes have crashed or are aware of a proposal of \( v \) for round \( k \) of \( G \). By the definition of \( R' \), the chains of events from proposals to these aware processes are through \( G \). Since the events executed in \( R' \) are a subsequence of the events executed in \( R \), these chains of events through \( G \) must also be present in \( R \).

Many of the Consensus algorithms using failure detectors presented in the literature [8, 48] become group-closed when converted into Repeated Group Consensus algorithms using the methodology described in Subsection 5.1.2. If a group-closed Repeated Group Consensus algorithm is being used, the Round Tracking algorithm described above can be modified so that it is much more space efficient. Instead of storing an array \( \text{round} \) of size \( |G| \), each process \( p \) need only store an array entry \( \text{round}[G] \) if \( p \in G \). If process \( p \) sends a message \( m \) to a process \( q \) then \( p \) piggybacks \( \text{round}_p[G] \) on \( m \) only if \( p \in G \) and \( q \in G \). When \( q \) receives \( \text{round}_p[G] \) from \( p \) it sets \( \text{round}_q[G] \) to \( \max(\text{round}_p[G], \text{round}_q[G]) \). A process \( p \) that executes \( \text{propose}(G, k, -) \) sets \( \text{round}_p[G] \) to \( k \) as it did before. Again we can see that this algorithm satisfies the Validity, Integrity and Monotonicity properties of Round Tracking.
5.2.3 Adding Linear Multicast to the Atomic Multicast Algorithm

The algorithm in Figure 5.3 enhances the Atomic Multicast algorithm in Figure 5.2 so that it implements Linear Multicast as well. The particular type of Linear Multicast (Strong or Weak) that is implemented depends on the failure detector that is used by the Repeated Group Consensus algorithm. The algorithm assumes that the Repeated Group Consensus algorithm is executed concurrently with a Round Tracking algorithm that maintains the array \( \text{round}_p \) for each process \( p \).

To L-Multicast a message \( m \) to a group \( G \), a process simply sends \( m \) to all of the processes in \( G \). This is done in Task 2. Task 4 is executed when a process in \( G \) receives \( m \). This task ensures that no process \( p \in G \) L-delivers \( m \) until \( p \) has A-delivered every message in \( A_{\text{deliver}}^p \), where \( \ell \) is the latest round of Repeated Group Consensus for \( G \) that \( p \) is aware of when \( p \) receives \( m \).

When a process \( p \) receives \( m \), it reads \( \text{round}_p[G] \) to determine the latest round of Repeated Group Consensus for \( G \) of which it is aware. The value of \( \text{round}_p[G] \) is stored in a variable \( l_p \), and is compared to \( k_p[G] \) – the latest round of Repeated Group Consensus for \( G \) that \( p \) has executed. If \( l_p \leq k_p[G] \) then \( m \) is delivered right away (in Task 4) since every message in \( A_{\text{deliver}}^p \) has already been A-delivered by \( p \). If \( k_p[G] < l_p \) then \( m \) is added to the set \( L_{\text{deliver}}^m \). The messages in \( L_{\text{deliver}}^m \) are L-delivered in Task 5 after all of the messages in \( A_{\text{deliver}}^p \) have been A-delivered.

A detailed proof of the correctness of the algorithm can be found in Subsection 5.2.5. Intuitively, the linear ordering of messages is guaranteed by the results of Subsection 5.2.1 which place constraints on the set of processes that have a proposal for a round \( k \) of Repeated Group Consensus for a group \( G \) in their causal pasts when any process decides for round \( k \) of \( G \). The Round Tracking Algorithm then ensures that any process \( p \) in this set is aware of that round of Consensus for \( G \), i.e., that \( \text{round}_p[G] \geq k \).
Every process \( p \) executes the following:

**Initialization:**

\[
\begin{align*}
R._{\text{delivered}} & \leftarrow \{0,0,\ldots,0\} \\
A._{\text{delivered}} & \leftarrow \{0,0,\ldots,0\} \\
k & \leftarrow \{0,0,\ldots,0\}
\end{align*}
\]

To execute \( \text{A-multicast}(G, m) \): \hspace{2cm} \{ Task 1 \}
\[
\text{R-multicast}(G, m)
\]

To execute \( \text{L-multicast}(G, m) \): \hspace{2cm} \{ Task 2 \}
\[
\text{send } m \text{ to all processes in } G
\]

A-deliver(\( m \)) and L-deliver(\( m \)) occur as follows:

\[
\begin{align*}
\text{when R-deliver}(m) & \hspace{2cm} \{ Task 3 \} \\
R._{\text{delivered}}[\text{group}(m)] & \leftarrow R._{\text{delivered}}[\text{group}(m)] \cup \{m\}
\end{align*}
\]

\[
\begin{align*}
\text{when receive}(m) & \hspace{2cm} \{ Task 4 \} \\
G & \leftarrow \text{group}(m) \\
l & \leftarrow \text{round}[G] \\
\text{if } k[G] < l & \text{ then} \\
L._{\text{deliver}}^l[G] & \leftarrow L._{\text{deliver}}^l[G] \cup \{m\} \\
\text{else} \\
L._{\text{deliver}}(m)
\end{align*}
\]

\[
\begin{align*}
\text{when for any group } G: R._{\text{delivered}}[G] - A._{\text{delivered}}[G] & \neq \emptyset & \{ Task 5 \} \\
k[G] & \leftarrow k[G] + 1 \\
A._{\text{undelivered}}[G] & \leftarrow R._{\text{delivered}}[G] - A._{\text{delivered}}[G] \\
\text{propose}(G, k[G], A._{\text{undelivered}}[G]) \\
\text{wait until decide}(G, k[G], msgSet^{k[G]}[G]) \\
A._{\text{deliver}}^{k[G]}[G] & \leftarrow msgSet^{k[G]}[G] - A._{\text{delivered}}[G] \\
\text{A-deliver all messages in } A._{\text{deliver}}^{k[G]}[G] \text{ in some deterministic order} \\
A._{\text{delivered}}[G] & \leftarrow A._{\text{delivered}}[G] \cup A._{\text{deliver}}^{k[G]}[G] \\
L._{\text{deliver}} \text{ all messages in } L._{\text{deliver}}^{k[G]}[G] \text{ in any order}
\end{align*}
\]

Tasks 4 and 5 cannot be executed concurrently.

Figure 5.3: An algorithm that adds Linear Multicast to the Atomic Multicast algorithm in Figure 5.2. The type of Linear Multicast that is implemented depends on the failure detector that is used to solve Repeated Group Consensus.
Let \( m \) be any message that is \( L \)-multicast to \( G \) at time \( t \), and let \( l \) be the last round of Repeated Group Consensus for \( G \) to be decided before \( t \). Any process that \( A \)-delivers the messages that are decided upon in round \( l \) of \( G \) will have delivered every message \( m' \) addressed to \( G \) that was \( A \)-delivered by any process in \( G \) before time \( t \). Task 4 ensures that if \( m \) is received by a process \( p \) at time \( t_p \geq t \) then \( p \) delivers \( m \) only after deciding for every round of Consensus for \( G \) that it was aware of at time \( t_p \); thus, the processes that are aware of round \( l \) at time \( t \) will not deliver \( m \) out of linear order. Note that Task 4 and Task 5 cannot be executed concurrently; this is essential to the correctness of the algorithm and is discussed in Subsection 5.2.5. Again, we are not concerned with the mechanics of how these tasks are made to be mutually exclusive; we only require that this is done somehow.

Therefore, a set of processes will not deliver \( m \) out of linear order, and this set has certain properties, such as a minimum size, that are guaranteed by the results of Subsection 5.2.1. These properties depend on the type of failure detector that is used to solve Repeated Group Consensus. Specifically, if a Strong failure detector is used then the Linear Multicast algorithm satisfies Weak Linear order. Using any of the other failure detectors considered in this thesis allows Strong Linear Order to be satisfied.

### 5.2.4 Fault-Tolerance

The fault-tolerance of the algorithm in Figure 5.3 is identical to the fault-tolerance of the algorithm in Figure 5.2.

### 5.2.5 Proof of Correctness

Let \( A_L \) denote the algorithm in Figure 5.3, and let \( A_C \) denote the Repeated Group Consensus algorithm that is used by \( A_L \). The correctness of the Atomic Multicast portion of the algorithm is proven in Appendix A. We now prove the correctness of the \( L \)-Multicast portion of the algorithm.
Lemma 5.2.2 (Uniform Integrity) For any message $m$, every process $p$ L-delivers $m$ at most once, and only if $p$ is in $\text{group}(m)$ and $m$ was previously L-multicast by $\text{sender}(m)$.

Proof: Let $m$ be a message that is L-multicast to a group $G$. By the algorithm, a process $p$ L-delivers $m$ only if $m$ was received by $p$. By the system model, a message is received by a process only if the message was previously sent, i.e., the message is in the message buffer. Since $m$ is sent by a process $q$ only if it was L-multicast by $q$, and $q$ only sends $m$ to processes in $\text{group}(m)$, $p$ L-delivers $m$ only if it was previously L-multicast by $\text{sender}(m)$ and only if $p \in \text{group}(m)$. We must now show that every process L-delivers $m$ at most once.

From the algorithm, we can also see that $m$ is L-delivered by $p$ either immediately after being received, or with all of the messages in $L\_\text{deliver}^1_p[G]$. Since $m$ is not added to $L\_\text{deliver}^1_p[G]$ if it is delivered immediately, and $\forall k$, $p$ L-delivers the messages in $L\_\text{deliver}^k_p[G]$ at most once (namely, after it A-delivers the messages in $A\_\text{deliver}^k_p[G]$), $p$ L-delivers $m$ at most once. $\square$

Lemma 5.2.3 $A_L$ satisfies the Validity and Weak Uniform Agreement properties of Strong Linear Multicast.

Proof: We will show that the algorithm satisfies the following property: If a correct process L-multicasts a message $m$, then all correct processes in $\text{group}(m)$ L-deliver $m$. The result follows from this property.

Let $p$ be a correct process that L-multicasts a message $m$ to a group $G$. By the algorithm, $p$ sends $m$ to all processes in $G$. Since there are no communication failures in our system model, all correct processes in $G$ receive $m$. We now show that every correct process in $G$ will L-deliver $m$ after receiving $m$.

Let $q$ be any correct process in $G$ that receives $m$. Let $t$ be the time at which $q$ receives $m$ and let $t_q$ be the value of $\text{round}_q[G]$ at time $t$. Upon receiving $m$, $q$ L-delivers
m immediately if \( l_q \leq k_q[G] \), otherwise \( m \) is added to \( L_{\text{deliver}}^l[G] \). By the algorithm, \( q \) delivers the messages in \( L_{\text{deliver}}^l[G] \) at most once, and does so only after executing \( \text{decide}(G, l_q, -) \). Also by the algorithm, \( q \) will not execute \( \text{decide}(G, l_q, -) \) before adding \( m \) to \( L_{\text{deliver}}^l[G] \) since \( l_q > k_q[G] \), and Task 4 (in which \( m \) is added to \( L_{\text{deliver}}^l[G] \)) and Task 5 (in which \( \text{decide}(G, l_q, -) \) is executed) cannot be executed concurrently.

By the Integrity property of the Round Tracking algorithm, some process executed \( \text{propose}(G, l_q, -) \) before \( t \), which means, by Lemma 5.1.3 that \( q \) will eventually execute \( \text{propose}(G, l_q, -) \). By the Weak Termination property of Group Consensus, \( q \) will eventually execute \( \text{decide}(G, l_q, -) \), and therefore deliver the messages in \( L_{\text{deliver}}^l[G] \).

\[ \Box \]

**Lemma 5.2.4** Let \( p \) be any process, \( G \) be any group, and \( t \) be any time. If a message \( m \) is \( L \)-multicast to \( G \) at time \( t' \geq t \) then \( p \) will not \( L \)-deliver \( m \) before \( A \)-delivering all messages in \( A_{\text{deliver}}^p[G] \), \( \forall 1 \leq j \leq \ell \), where \( \ell \) is the value of \( \text{round}_p[G] \) at time \( t \).

**Proof:** By the algorithm, \( p \) will \( L \)-deliver \( m \) only if it receives \( m \). Since \( m \) is sent to all processes in \( G \) by \( \text{sender}(m) \) after \( t' \), \( p \) will not receive \( m \) at any time prior to \( t \). Therefore, by the Monotonicity property of the Round Tracking algorithm, \( \text{round}_p[G] \geq \ell \) when \( p \) receives \( m \).

Let \( k \) and \( l \) be the value of \( k_p[G] \) and \( \text{round}_p[G] \), respectively, when \( p \) receives \( m \), and let \( j \) be any round number such that \( 1 \leq j \leq l \). If \( j \leq k \) then by the algorithm, all of the messages in \( A_{\text{deliver}}^p[G] \) have already been \( A \)-delivered. If \( j > k \) then, since \( l \geq j \), \( m \) is added to \( L_{\text{deliver}}^l[G] \) and is not delivered until all of the messages in \( A_{\text{deliver}}^l[G] \) have been delivered. By the algorithm, the messages in \( A_{\text{deliver}}^p[G] \) are not delivered until all of the messages in \( A_{\text{deliver}}^l[G] \) have been delivered. \[ \Box \]

The following lemma states bounds on the set of processes that can \( L \)-deliver a message out of linear order; these bounds depend on the failure detector used by the Repeated Group Consensus algorithm.
Lemma 5.2.5 Let $R = (F, H, I, S, T)$ be any run of $A_L$ in which a message $m$ is $L$-multicast to a group $G$ at time $t$.

1. If $A_C$ is an $\Gamma$-tolerant solution to Repeated Group Consensus for $G$ using $\diamond \mathcal{P}$ where $\forall G' \in \mathcal{G} : G' \neq G \implies \Gamma(G') = |G'|$ then no more than $\max(|G| - \Gamma(G) - 1, 0)$ processes L-deliver $m$ out of linear order in $R$.

2. If $A_C$ is an $\Gamma$-tolerant solution to Repeated Group Consensus for $G$ using $S$ where $\forall G' \in \mathcal{G} : G' \neq G \implies \Gamma(G') = |G'|$ and $R$ uses a failure detector $\mathcal{D} \in S$ then one of the following is true: (a) no more than $\max(|G| - \Gamma(G) - 1, 0)$ processes L-deliver $m$ out of linear order in $R$, or (b) some process $p \in G$ does not L-deliver $m$ out of linear order in $R$, where either $p$ is correct and is never suspected in $R$, or, if all processes in $G$ are faulty, $p$ is the last process in $G$ to crash and is never suspected before crashing in $R$.

3. If $A_C$ is an $\Gamma$-tolerant solution to Repeated Group Consensus for $G$ using $\mathcal{P}$ where $\forall G' \in \mathcal{G} : G' \neq G \implies \Gamma(G') = |G'|$ and $R$ uses a failure detector $\mathcal{D} \in \mathcal{P}$ then no more than $\max(|G| - \Gamma(G) - 1, 0)$ processes L-deliver $m$ out of linear order in $R$.

Proof: Let $t$ be the time at which $m$ is L-multicast in $R$, and let $\mathcal{M}_1$ be the set of all messages $m'$ such that $\text{group}(m') = G$ and $m'$ is A-delivered by some process before $t$. If $\mathcal{M}_1 = \emptyset$ then $m$ cannot be delivered out of linear order. Assume instead that $\mathcal{M}_1 \neq \emptyset$. By the Local Delivery Consistency property of Atomic Multicast, there exists a process $p \in G$ such that $p$ has A-delivered all messages in $\mathcal{M}_1$ by $t$. Let $m'$ be the last message in $DLVD_p^t$ ($m'$ must exist since $\mathcal{M}_1 \neq \emptyset$). By the algorithm $m' \in A_{\text{deliver}}_p^t[G]$ for some round $k$, and all messages in $A_{\text{deliver}}_p^t[G]$ were members of the set $msgSet_p^k[G]$ that was decided in round $k$ of Group Consensus for $G$ by $p$.

Before continuing with our proof of the lemma, we prove the following claim:
Claim: Any process $q$ that has a proposal for round $k$ of Repeated Group Consensus for $G$ in its causal past at time $t$ will not L-deliver $m$ out of linear order.

Proof of Claim: By the Validity property of the Round Tracking algorithm, $\text{round}_q[G] \geq k$; therefore, by Lemma 5.2.4, $q$ will not L-deliver $m$ without previously A-delivering all of the messages in $A\_ \text{deliver}_q^k[G]$. Since $m' \in A\_ \text{deliver}_q^k[G]$, and $A\_ \text{deliver}_q^k[G] = A\_ \text{deliver}_q^k[G]$ by Lemma 5.1.2, $q$ must A-deliver $m'$ before it L-delivers $m$. By Local Delivery Consistency, any process that A-delivers $m'$ has previously delivered all messages in $M_1$; therefore, $q$ will not L-deliver $m$ out of linear order.

We will now prove the first part of the lemma. Assume that $A_C$ is a $\Gamma$-tolerant solution to Repeated Group Consensus for $G$ using $\mathcal{OP}$ where $\forall G' \neq G : \Gamma(G') = |G'|$. By Corollary 5.2.1, $\min(\Gamma(G) - F(t) + 1, |G|)$ processes in $G$ have crashed or have a proposal for round $k$ of Repeated Group Consensus for $G$ in their causal past at time $t$. By our claim, every process that has a proposal for round $k$ of $G$ in its causal past at time $t$ will not L-deliver $m$ out of linear order. Also, by the Uniform Integrity property of Linear Multicast (see Lemma 5.2.2), every process that has crashed by time $t$ will not L-deliver $m$ at all. Therefore, $\min(\Gamma(G) + 1, |G|)$ processes in $G$ will not L-deliver $m$ out of linear order. By the Uniform Integrity property of Linear Multicast, no more than $|G|$ processes may L-deliver $m$; therefore, no more than $\max(|G| - \Gamma(G) - 1, 0)$ processes deliver $m$ out of linear order.

Next we will prove the second part of the lemma. Assume that $A_C$ be an $\Gamma$-tolerant solution to Repeated Group Consensus for $G$ using $\mathcal{S}$ where $\forall G' \in G : G' \neq G \Rightarrow \Gamma(G') = |G'|$, and that $R$ uses a failure detector $\mathcal{D} \in \mathcal{S}$. Further assume that (a) does not hold; specifically, assume that more than $\max(|G| - \Gamma(G) - 1, 0)$ processes deliver $m$ out of linear order. Therefore, by our claim and by the Integrity property of Linear
Multicast, no more than \( \min(\Gamma(G), |G| - 1) \) processes have crashed by time \( t \) or have a proposal for round \( k \) of \( G \) in their causal past at time \( t \). By Corollary 5.2.2, some process \( p' \in G \) has a proposal for round \( k \) of \( G \) in its causal past at time \( t \), where either \( p \) is correct and is never suspected, or, if all processes in \( G \) are faulty, \( p \) is the last process in \( G \) to crash and is never suspected before crashing. By our claim, \( p' \) does not deliver \( m \) out of linear order; thus, (b) holds.

Finally, we will prove the third part of the lemma. Let \( A_C \) be an \( \Gamma \)-tolerant solution to Group Consensus for \( G \) only using \( P \) where \( \forall G' \in G : G' \neq G \implies \Gamma(G') = |G'| \), and assume that \( R \) uses a failure detector \( D \in P \). By Corollary 5.2.3, \( \Gamma(G) + 1 \) processes in \( G \) have crashed or have a proposal for round \( k \) of Repeated Group Consensus for \( G \) in their causal past at time \( t \). The remainder of the proof is identical to the proof of the first part of the lemma. \( \square \)

The following three theorems now follow from Theorem 5.1.1, and Lemmata 5.2.2, 5.2.3 and 5.2.5:

**Theorem 5.2.2** If \( A_C \) is a \( \Gamma \)-tolerant solution to Repeated Group Consensus for any group \( G \) using \( \Diamond P \) where \( \forall G' \in G : G' \neq G \implies \Gamma(G') = |G'| \), then \( A_L \) is a \( \Gamma \)-tolerant solution to \( A \)-Multicast for \( G \) and \( SL \)-Multicast for \( G \) using \( \Diamond P \) with a weakness function of \( \beta(G) = \max(|G| - \Gamma(G) - 1, 0) \).

**Theorem 5.2.3** If \( A_C \) is a \( \Gamma \)-tolerant solution to Repeated Group Consensus for any group \( G \) using \( S \) where \( \forall G' \in G : G' \neq G \implies \Gamma(G') = |G'| \), then \( A_L \) is a \( \Gamma \)-tolerant solution to \( A \)-Multicast for \( G \) and \( WL \)-Multicast for \( G \) using \( S \) with a weakness function of \( \beta(G) = \max(|G| - \Gamma(G) - 1, 0) \).

**Theorem 5.2.4** If \( A_C \) is a \( \Gamma \)-tolerant solution to Repeated Group Consensus for any group \( G \) using \( P \) where \( \forall G' \in G : G' \neq G \implies \Gamma(G') = |G'| \), then \( A_L \) is a \( \Gamma \)-tolerant solution to \( A \)-Multicast for \( G \) and \( SL \)-Multicast for \( G \) using \( P \) with a weakness function of \( \beta(G) = \max(|G| - \Gamma(G) - 1, 0) \).
5.2.6 Performance Analysis

The algorithm implements Atomic Multicast with the same message cost and latency as the algorithm in Figure 5.2. As with the Atomic Multicast algorithm, the latency of the Linear Multicast algorithm depends on the latency $d_C$ of the Repeated Group Consensus algorithm used. The message cost of the Linear Multicast algorithm, however, is independent of the message cost of the Repeated Group Consensus algorithm. Specifically, the message cost of L-multicasting a single message $m$ is $n_{SL} = |\text{group}(m)|$.

The best case latency to L-deliver a message is $d_m$, since a message can be delivered as soon as it is received if the delivering process has decided every round of Group Consensus that it is aware of. The worst case latency depends on how many Group Consensus rounds a process has to wait for before L-delivering a message. Thus, if a process $p$ is far "behind" other processes, i.e. $p$ is aware of round $k$ of Group Consensus but has only decided round $l \ll k$ of Group Consensus, then $p$ will have to wait until it has decided for many rounds of Group Consensus before L-delivering a message. However, by our timing assumptions, all messages have a fixed delay time of $d_m$, and local processing times are negligible; therefore, all processes should propose each round of Group Consensus at the same time, and no process will get “behind”. Thus, a process will wait at most one round of Group Consensus after receiving a message $m$ before L-delivering $m$. Therefore, the worst case latency of L-delivering a message $m$ is $d_{SL} = d_C(\text{group}(m)) + d_m$.

It is therefore more efficient, in terms of both message cost and latency, to L-multicast a message than to A-multicast that message using the algorithm in Figure 5.2. L-multicasting a message $m$ has a message cost of only $|\text{group}(m)|$ messages, while A-multicasting a message $m$ requires at least $|\text{group}(m)|^2$ messages. This message cost is not only due to the Reliable Multicast used – Consensus algorithms for asynchronous systems in the literature also require $O(|G|^2)$ messages to decide a round of Consensus for a group $G$. L-multicasting a message $m$ has a latency that is $d_C(\text{group}(m))$ less than the latency required to A-multicast $m$. Essentially, this is due to the fact that
even though both A-multicast and L-multicast messages cannot be delivered until the pending rounds of Group Consensus have been decided, A-multicast messages must then be decided upon in the next round of Consensus, while L-multicast messages can be L-delivered immediately.
Chapter 6

Conclusions and Discussion

6.1 Conclusions

We have studied the emulation of linearizable shared objects in asynchronous systems. For reasons of fault-tolerance, these objects are replicated across a particular group of processes. We have focused on algorithms, based on the state-machine approach [52], that use Atomic Multicast to ensure the linearizability of the object implementations. Even though Atomic Multicast ensures that all processes in a group agree on the set of messages to be delivered, and deliver those messages in the same total order, the requirements of ensuring linearizability using the state-machine approach demand that the order in which messages are delivered respects some real-time ordering constraints. We have shown that Atomic Multicast satisfies a property, called Linear Order, that allows it to provide the required real-time ordering constraints.

We then considered a class of shared objects, called read/update objects. A read/update object can be accessed using two types of operations: read operations that do not modify the state of the object, and update operations which can modify the state of the object. We argued that since read operations do not modify the state of an object, their implementations did not require the strong semantics of Atomic Multicast. Instead, we
showed that if the update operations are implemented using Atomic Multicast, the read operations of a read/update object can be implemented using a multicast with much weaker semantics, called Linear Multicast.

A Linear Multicast does not satisfy the agreement and total ordering semantics of Atomic Multicast; instead, it only requires that for every message $m$ that was Linearly multicast at any time $t$:

1. $m$ is delivered by all of the processes in the addressed group only if the sender of the message is correct; and

2. some subset of processes in the addressed group deliver the message in linear order, i.e., they deliver the message after delivering every message $m'$ that was atomically delivered by any process in the group before $t$.

We defined two types of Linear Multicast: Strong Linear Multicast and Weak Linear Multicast, which differ in the properties of the subset of processes that are guaranteed to deliver a message in linear order. Every implementation of Strong or Weak Linear Multicast is characterized by a weakness function $\beta$ which limits the number of processes that can deliver a message out of linear order.

Thus, when used to implement the read operations of a read/update object, Linear Multicast ensures that some set of processes will deliver the request message for any read operation after delivering and processing the request messages of all the update operations that completed before the read operation was invoked. Based upon the type and weakness function of the Linear Multicast used, a client process can select a return value from the reply messages for an operation that allows that operation to be linearized.

We then discussed the addition of Strong Linear Multicast to existing fault-tolerant Atomic Multicast algorithms. Criteria were presented which, if satisfied by an Atomic Multicast algorithm, were sufficient to allow that algorithm to be enhanced so that it also implemented Strong Linear Multicast. L-multicasting a message using an algorithm
that has been enhanced in this way is more efficient and faster than A-multicasting a message. Specifically, L-multicasting a message to a group \( G \) only has a message cost of \(|G|\) messages and a latency that is between the latency of simply sending a message to \( G \) and the latency of A-multicasting a message to \( G \).

However, most of the Atomic Multicast algorithms in the literature that satisfy these criteria rely on the existence of Perfect failure detectors. We presented an Atomic Multicast algorithm, based on the Chandra and Toueg Atomic Broadcast algorithm, that can use any Consensus algorithm to solve Atomic Multicast. Therefore, this algorithm can solve Atomic Multicast using any of the failure detector classes considered in this thesis. We showed that Linear multicast can also be implemented using any Consensus algorithm, by presenting an enhancement of this Atomic Multicast algorithm that also implements Linear Multicast. The weakness function of the Linear Multicast that was implemented by this algorithm depends on the fault-tolerance of the Consensus algorithm that is used; the type (Strong or Weak) of the Linear Multicast depends on the class of failure detector employed by the Consensus algorithm that is used.

When this Atomic Multicast and Linear Multicast algorithm is used by our consistency protocol for read update objects, the message cost of an update operation on an object \( x \) is \( n_C(\text{objGroup}(x)) + 2|\text{objGroup}(x)| \) where \( n_C(\text{objGroup}(x)) \) is the number of messages required to decide a round of Consensus for group \( \text{objGroup}(x) \). The message cost of a read operation on \( x \) is \( 2|\text{objGroup}(x)| \). The latency of an update operation on \( x \) is between \( d_C(\text{objGroup}(x)) + 2d_m \) and \( 2d_C(\text{objGroup}(x)) + 2d_m \). The latency of a read operation on \( x \) is between \( 2d_m \) and \( d_C(\text{objGroup}(x)) + 2d_m \). This implementation of linear multicast therefore allows read operations to be implemented more efficiently than update operations in terms of both message cost and latency.

In order to prove the that our implementation of read/update shared objects is linearizable, we have developed a proof technique based on linearly supportive partial orders. This is a useful tool for proving the linearizability of other read/update object implement-
tations and is an extension of similar techniques found in the literature for proving the correctness of linearizable read/write object implementations and sequentially consistent read/update object implementations.

In order to prove the correctness of our Linear Multicast algorithm, we developed, in Section 2.3, a set of results dealing with the properties of algorithms using failure detectors. These results are not specific to the Consensus or Atomic Multicast problems, and apply to any algorithm for asynchronous systems using failure detectors. Essentially, these results show that if an algorithm must guarantee that a correct process satisfies a certain property, then that algorithm must guarantee that a sufficiently large set of processes satisfy that property. Specifically, for Perfect, Eventually Perfect, and Eventually Strong failure detectors, the algorithm must ensure that the number of processes that satisfy the property is more than the number of processes that may fail. This stems from the fact that most of the failure detector classes that we are considering do not allow a correct process to be identified until every faulty process (that is allowed to fail) has failed.

The same does not apply for Strong failure detectors since the properties of these failure detectors do allow a correct process to be identified before any processes have failed. A consequence of this result is that we cannot guarantee that a Consensus algorithm that uses Strong Failure detectors can be used to implement Strong Linear Multicast using the Linear Multicast algorithm presented in this thesis.

This limitation (or advantage, depending on your viewpoint) is, in fact, a practical one. Chandra and Toueg [8] presented an algorithm that uses S to solve Consensus. In any run of this algorithm where a process p suspects every other process in the system, p may decide for a round of Consensus before any other process has a proposal for that round in its causal past – even if every process is correct in that run. If this algorithm is used to solve Consensus (and Strong failure detectors are used), then our Linear Multicast algorithm does not satisfy Strong Linear Order; it does, however, satisfy Weak Linear
Order. It is interesting to note that this same Consensus algorithm can be used to implement Strong Linear Multicast if Perfect failure detectors are used instead.

6.2 Related Work

6.2.1 Sequential Consistency versus Linearizability

Linearizability is only one of many consistency criteria that have been defined for shared object implementations. Sequential consistency [36] is a consistency criterion that is similar to linearizability in that it guarantees that operations appear to be executed sequentially; although it does not guarantee that this sequential ordering preserves the real-time ordering of operations. Formally, a history $H$ is *sequentially consistent* [36] if it can be extended to some history $H'$ such that $\text{complete}(H')$ is equivalent to some legal sequential history $S$.

Attiya and Welch [3] showed that in message passing systems with imperfect clocks and variable (but not unbounded) message delays it is strictly more expensive to implement linearizable shared registers than to implement sequentially consistent shared registers. The consistency criteria were compared in the context of a *full replication scheme* where every process in the system maintains a replica of every object.

Two algorithms that ensure the sequential consistency of shared registers were presented by Attiya and Welch that implement writes by atomically broadcasting an update message to all processes; upon receiving this update message, each process performs the write on its local copy of the object. Also, reads are implemented in both algorithms by returning the state of the local copy of the register. However, in the first algorithm, write operations do not return until the invoking process has delivered the update message that it had Atomically Broadcast, while read operations return immediately. In the second algorithm, write operations return immediately after Atomically Broadcasting the update message, and read operations do not return with the state of the register until all
Pending update messages have been delivered. Thus, the first algorithm has fast reads (the reads had a worst case delay of 0), and the second algorithm has fast writes.

The fast-reads algorithm cannot only be used to implement sequentially consistent read/write objects, but it can also be used to implement sequentially consistent read/update objects. Furthermore, this algorithm does not make use of the time bounds of the system, and can therefore be used in an asynchronous system using failure detectors. However, we cannot use the fast-writes algorithm to implement sequentially consistent read/update objects with fast updates since, by our definition, updates return a value that may depend on the execution of the operation on an object replica.

Attiya and Welch also developed lower bounds for shared registers in their model that indicates that no implementation of a linearizable shared registers exists such that neither the read operation nor the write operation have a response time of 0. Since the model used by Attiya and Welch is a stronger model than the asynchronous model that we are considering, and read/write objects are a special case of read/update object, the lower bound also applies to read/update objects in an asynchronous system. Therefore, these results imply that the read operations of sequentially consistent read/update objects are less costly to implement than linearizable read/update objects in asynchronous systems. This makes sequential consistency a better consistency criterion than linearizability when replication is used for the purpose of caching, since sequential consistency allows a local copy of an object to be read instantly, while linearizability does not.

In this thesis, however, we have considered a partial replication scheme rather than a full replication scheme. Specifically, replicas of an object are only maintained by the processes in a particular group rather than by all of the processes in the system. For this case, it is not clear if providing sequential consistency is less costly than providing linearizability when implementing read/update objects. Specifically, if we wish to extend the fast-reads algorithm to the partial replication scheme then we must use a form of multicast with stronger semantics than those provided by Atomic Multicast to implement
Figure 6.1: A problem using (Pairwise) Atomic Multicast in a sequentially consistent implementation of read/update objects with fast reads.

For example, consider a set of processes $p_1$, $p_2$, $p_3$ and $p_4$, and a set of objects $x_1$, $x_2$, and $x_3$ such that $p_1$, $p_2$, and $p_4$ maintain replicas of $x_1$, $p_2$ and $p_3$ maintain replicas of $x_2$, and $p_3$ and $p_4$ maintain replicas of $x_3$. Figure 6.1 depicts a run in which process $p_1$ performs an update $u_1$ on $x_1$, $p_2$ performs an update $u_2$ on $x_2$, and $p_3$ performs an update $u_3$ on $x_3$. Furthermore, $p_2$ reads $x_1$, $p_3$ reads $x_2$ and $p_4$ reads $x_3$, and each process does so after processing the updates of the respective objects. Process $p_4$ also reads $x_1$ after reading $x_3$ but before processing the update of $x_1$. It follows that in the history that respects this run: $u_1$ precedes $u_2$, $u_2$ precedes $u_3$ and $u_3$ precedes $u_1$; thus, the history that respects this run is not sequentially consistent.

The delivery of update request messages, however, not only satisfies the properties of Atomic Multicast, but also satisfies the properties of Pairwise Atomic Multicast [29]. This is a stronger form of multicast that ensures that no two processes deliver any two messages in a different order. Formally, Pairwise Atomic Multicast is a Reliable Multicast that satisfies the following property:

**Pairwise Uniform Total Order** If processes $p$ and $q$ both deliver messages $m$ and $m'$, then $p$ delivers $m$ before $m'$ if and only if $q$ delivers $m$ before $m'$.

Thus, in order to implement read/updates with fast reads, we must use an even
stronger form of multicast to implement the updates. Specifically, we must use a form of Atomic Multicast that does not allow cycles in the message delivery pattern. This multicast is called a \textit{Global Atomic Multicast} \cite{29}, and is defined in terms of a relation \( <_M \) on the set of all messages delivered in a run, where \( m <_M m' \) if and only if any process delivers \( m \) before it delivers \( m' \). Formally, Global Atomic Multicast is a Reliable Multicast that satisfies the following property:

\textbf{Global Uniform Total Order} The relation \( <_M \) is acyclic.

Another factor that needs to be considered in a partial replication scheme is that an operation can be invoked on an object by a client that does not maintain a replica of that object. Clearly, if a process does not maintain a copy of an object, fast reads cannot be implemented for this process. We can reasonably assume that in order to implement a read operation in this case, a request message must be sent to one or more server processes and at least one reply message must be received by the client invoking the operation. This raises the following question: Can the read operations of sequentially consistent read/update objects be implemented using multicsats (or point-to-point messages) whose ordering semantics are cheaper to implement than those required by linearizable read/update objects?

Our implementation of read/update objects ensures that at least one correct server process receives each request message in linear order. Linear Order not only ensures that request messages respect real-time ordering constraints, but also ensures that these messages are delivered causally. Specifically, a multicast that satisfies Linear Order also satisfies an ordering property called \textit{Local Context Order} (this property is called \textit{Local Order} by Hadzilacos and Toueg \cite{29}). This property is formally defined as follows:

\textbf{Local Context Order} If a process delivers a message \( m' \) before multicasting a message \( m \) and \( \text{group}(m') = \text{group}(m) \) then no process delivers \( m \) unless it has previously delivered \( m' \).
It is easy to see that any message $m$ that is delivered in Linear Order is also delivered in Local Context Order, i.e., after delivering every message $m'$ that was addressed to $\text{group}(m)$ and was delivered by $\text{sender}(m)$ before $m$ was multicast.

Local Context Order can be seen as a weakened version of Local Causal Order [29] – an ordering property that is defined by the causal precedence of messages in a particular group. Specifically, causal precedence in a group $G$ is a causal precedence relation induced by the multicascts and deliveries of messages $m$ such that $\text{group}(m) = G$. Thus, the causality relation is defined in such a way that messages that are not addressed to $G$ are ignored. Local Causal Order is defined as follows [29]:

**Local Causal Order** If the multicast of a message $m$ causally precedes in $\text{group}(m)$ the multicast of a message $m'$, then no correct process delivers $m'$ unless it has previously delivered $m$.

Hadzilacos and Toueg showed that Local Causal Order is equivalent to Local Context Order and Local FIFO Order (which requires that messages multicast by the same process to the same group are delivered in the order in which they are sent)\(^1\) [29]. However, Linear Order implies FIFO Order in our model since it is assumed that processes execute operations sequentially. Specifically, once a process has multicast a request message $m$, it does not multicast another request message $m'$ until it has either delivered $m$ or a reply message from some process that has delivered $m$. Thus, any process that delivers $m'$ in linear order will have previously delivered $m$. Thus, messages that are delivered in linear order are delivered in local causal order.

Even if we relax the linear order requirement for request messages in a sequentially consistent implementation of read/update objects, it is important that the request messages be delivered in a global causal ordering (wherein causal precedence is not restricted to a particular group); otherwise, the problem depicted in Figure 6.2 may occur. This

\(^1\)Hadzilacos and Toueg's result is actually given for broadcast ordering properties; however, it is straightforward to translate the result and its proof to these multicast ordering properties.
Figure 6.2: An argument for causally ordered read request messages when implementing sequentially consistent read/update objects.

The figure portrays a run of a system containing six processes: \( p_1, p_2, q_1, q_2, \) and \( c_1 \); and two read/update objects: \( x_1 \) and \( x_2 \). Replicas of \( x_1 \) are maintained by \( p_1 \) and \( p_2 \); replicas of \( x_2 \) are maintained by \( q_1 \) and \( q_2 \). We assume that update request messages are sent using a form of multicast that is at least as strong as Global Atomic Multicast, and that read request messages are sent to one of the processes maintaining replicas of the invoked object. We further assume that a process \( p \) executing an operation \( op \) on an object \( x \) returns from \( op \) if and only if \( p \) has performed \( op \) on its replica of \( x \) (if such a replica exists) or after receiving a single reply to its request message for \( op \).

In the run shown in Figure 6.2, \( p_1 \) executes update \( u_1 \) then executes update \( u_2 \) on objects \( x_1 \) and \( x_2 \) respectively; meanwhile, \( c_1 \) executes a read \( r_2 \) then executes read \( r_1 \) on objects \( x_2 \) and \( x_1 \) respectively. By order of message delivery, the values read by \( c_1 \) indicate that \( u_2 \) precedes \( u_1 \); thus, the history that respects this run is also not sequentially consistent. Even though the delivery pattern of request messages \( <_M \) is free of cycles, process \( p_1 \) delivered the request message to read \( x_1 \) before delivering the update request message that causally preceded it, causing the problem.

Unfortunately, even if the delivery order \( <_M \) of all request and reply messages is acyclic, and all request and reply messages are delivered in causal order, the implemented objects are not sequentially consistent. Figure 6.3 portrays a run of the same system.
Figure 6.3: An argument for Context Multicast when implementing sequentially consistent read/update objects.

shown in Figure 6.2, except that it includes another client process $c_2$. In this run, two updates $u_1$ and $u_2$ are performed on $x_1$ and $x_2$ respectively. $c_1$ performs a read on $x_1$ followed by a read on $x_2$, while $c_2$ reads the objects in the opposite order. Every message in this run is delivered in causal order, and the delivery order $<_M$ is acyclic. However, by the order of message delivery that is depicted, the values read by $c_1$ indicate that $u_1$ precedes $u_2$ and the values read by $c_2$ indicate that $u_2$ precedes $u_1$; therefore, the history that respects this run is not sequentially consistent.

The Orca distributed programming language [4] implements shared read/update objects over the Amoeba distributed operating system [53]. Fekete et al. [18] give a formal definition of the consistency algorithm used by Orca, and prove that this algorithm implements sequentially consistent read/update objects. The algorithm allows a client to perform fast reads on an object when the client maintains a replica of that object. To overcome the problems we have outlined above, all request and reply messages are multicast using a Context Multicast. Like Global Atomic Multicast, this multicast is defined in terms of the relation $<_M$. Context Multicast is also defined in terms of the Context Order relation $<_C$ on the set of messages multicast in a run, where $m <_C m'$ if and only if:
1. \textit{sender}(m') delivered \( m \) before multicasting \( m' \); or

2. there exists a message \( m'' \) such that \( m <_G m'' \) and \( m'' <_G m' \).

Formally, a Context Multicast is a Reliable Multicast that satisfies the additional property that the union of \( <_M \) and \( <_G \) is acyclic. (Note that as in our model, the FIFO ordering of the messages need not be explicitly enforced by the properties of the multicast primitive.)

We are now left with the following question: Are Atomic Multicast and Linear Multicast cheaper to implement than Context Multicast? The answer to this question is not definitive however, Atomic Multicast and Linear Multicast have local semantics while Context Multicast has Global semantics. This is, perhaps, related to the fact that linearizability is a local property, while sequential consistency is not [30]. Thus, a consistency protocol for linearizable shared objects only need to ensure that each individual object implementation is linearizable while a consistency protocol for sequentially consistent shared objects must ensure that the implementation of the entire set of objects is sequentially consistent.

Guerraoui and Schiper [25] have shown that in any asynchronous system using unreliable failures no Pairwise Atomic Multicast algorithm (and subsequently no Global Atomic Multicast algorithm) exists that allows a message \( m \) to be multicast to a group \( G \) by exchanging messages only between \( \text{sender}(m) \) and \( G \). Therefore, a Context Multicast, which has even stronger semantics than Global Atomic Multicast, places a higher load on a system than an Atomic Multicast.

Orca was designed for local-area networks in which Atomic Broadcast could be efficiently implemented with hardware assistance. Context Multicast was then implemented on top of this Atomic Broadcast mechanism. However, in asynchronous systems with point-to-point communication the extra load required by Context Multicast might make it no less efficient to implement linearizable shared objects, even if sequential consistency is all that is required by the applications using those objects.
Thus, despite the fact that sequentially consistent read/update objects can be implemented with fast reads, this implementation requires a stronger and more expensive form of multicast than our implementation of linearizable read/update objects in the context of partial replication. Thus there seems to be a trade-off between the cost of reads and the cost of writes when comparing sequential consistency to linearizability for read/update objects.

6.2.2 Other Optimizations of Active Replication

Many other systems define communication primitives with message delivery semantics that are weaker than those provided by Atomic Multicast, allowing for the optimization of operations that do not require request messages to be totally ordered. We have already discussed Orca, which allows request messages for read operations to respect only a causal ordering while update request messages are totally ordered. The use of causal messages to implement replicated services has also been explored in the Lazy Replication system of Ladin et al. [33]. Many group communication systems such as ISIS [5, 50], Horus [54], Transis [15], Psync [45] and Totem [42] also provide a form of causal multicast. However, as we have already seen, some form of real-time ordering semantics are required to implement linearizable shared objects using Active Replication.

Consul [41], an extension of Psync [45], provides a communication primitive that delivers messages in semantic order. This message ordering is based on the commutativity of the operations on an object, allowing processes to relax the total ordering of request messages for operations that commute. Unlike our implementation of read/update objects, all request messages are multicast using the same multicast primitive. The server processes for each object run a protocol that totally orders all request messages unless a batch of messages are received such that the operations that are requested by those messages commute. In this case the messages can be delivered by the servers in any order. The servers must, however, agree upon the subsequence of totally ordered messages that
precedes each batch of "unordered" request messages. Object implementations using semantically ordered request messages are therefore linearizable. Johnson and Maugis [31] proposed a similar protocol that implement semantic ordering on top of the ToTo total ordering protocol for the Transis system.

Linear Multicast only allows read operations to be efficiently implemented while semantic ordering also allows for the efficient implementation of update operations that commute. Therefore, a shared object implementation such as Consul can optimize the implementation of a wider class of operations than an implementation based on Linear Multicasts. However, Linear Multicast can be used to implement read operations more efficiently than semantic ordering since Linear Multicast does not require that servers agree on the set of update request messages that are to be delivered before a read request message; it is only required that some correct process delivers a read request message in linear order. Also, existing semantic ordering protocols assume that a failure detection (or group membership) service is available that guarantees that no messages from a crashed process will be delivered after the failure of that process has been detected. Our Linear Multicast algorithm has no such requirement.

6.2.3 Other Methods of Replica Management

We will briefly compare Active Replication to other methods of replica management. For a more in depth comparison of these techniques, the reader is referred to Guerraoui and Schiper [26].

Passive Replication

Passive Replication, also called the Primary-Backup Approach [6], is a centralized form of replica management that does not require the use of multicast communication. One of the servers that maintain a replica of an object \( x \) fills a special role and is called the \textit{primary}; the remaining servers for \( x \) are called \textit{backups}. Instead of multicasting a
request message to all of the processes in objGroup(x), a client only sends a request to the primary process when invoking an operation on an object. Upon receiving the request message, the primary process performs the requested operation on its replica of the object. The primary then sends an update message containing the state of x to all of the backup processes in objGroup(x), and then sends a reply message to the client. When a backup server receives an update message, it changes the state of its replica of x to the value contained in the message. The backup processes must process the messages received from the primary in the order that they are sent. This can be accomplished by sending the update messages using a multicast primitive that enforces FIFO delivery of messages. When implementing read/update objects using Passive Replication, the read operations can be optimized. Specifically, once the primary has performed the read, update messages do not need to be sent to the backups since the state of the object has not changed.

Even though the primary process may act as a bottleneck, Passive Replication has many advantages over Active Replication. When using Passive Replication, the cost of each operation tends to be cheaper than when using Active Replication since request messages are only sent to the primary server do not need to be sent using an expensive communication primitive such as Atomic Multicast. Since replica operations are only performed by a single process, the implemented objects need not be deterministic. Also, if it is more efficient to update the state of an object replica than to perform a replica operation on that replica, then Passive Replication uses less system resources than Active Replication.

It is easy to see that a consistency protocol that is based on Passive Replication can easily tolerate the failure of any number of backup servers. Tolerating the failure of the primary server, however, is more difficult. If the primary server fails, then one of the backup processes must take over and become the new primary. This is called the election of a primary (sometimes called leader election), and requires that processes detect the
failure of the primary and eventually agree on the identity of the new primary.

Once a new process has taken on the role of the primary, the primary and backup processes must come to an agreement on the latest update to the object that occurred before the original primary failed. This is necessary because the original primary may have failed while sending an update message and so only a subset of the backup processes may have received that message. If the new primary was one of the processes that did not receive this update message, then future operations on the object may be incorrect.

Thus, if the primary process crashes, the remaining processes must identify that the primary has crashed, and agree on both the identity of the new primary, and on the latest update to the object. This kind of agreement can be costly and is, of course, not possible without the aid of failure detectors due to the FLP impossibility result. It is also not clear whether this agreement can be implemented more efficiently, or less efficiently than the agreement necessary for Active Replication; namely, the agreement by the servers on the sequence of request messages to be delivered.

Quorum Techniques

Gifford [22] introduced the use of quorums to manage transactions on replicated files in message passing systems. When using this technique, each replica of an object \( x \) is assigned a number of votes, and read and write quorums are defined so that all write quorums intersect, and each read quorum intersects with every write quorum. To write a file, a transaction writes to at least a write quorum of file replicas. Similarly, to read a file, read results are gathered from at least a read quorum of file replicas. Since read and write quorums overlap, at least one of the file replicas that are read must contain the most recently written version of the file. The algorithm tolerates both process crashes and network link failures, since a client process can properly execute transactions as long it can communicate with a write quorum and a read quorum of non-crashed servers. The algorithm assumes the existence of some underlying transactional system that allows
operations to be ordered.

Lynch and Shvartsman [39] presented an algorithm that uses quorum techniques to implement linearizable read/write shared objects for asynchronous message passing systems. The algorithm provides a similar fault-tolerance to Gifford's algorithm, but does not assume the existence of either an underlying transactional system, or a communication primitive such as Atomic Multicast. The algorithm does not need to explicitly order operations since the outcome of a write operation does not depend on the state of an object. Thus, unlike read/update objects, read/write objects can be emulated in an asynchronous message passing system without the use of failure detectors (arbitrary read/update objects cannot be implemented without the use of failure detectors since these objects could be used to solve Consensus).

The algorithm presented by Lynch and Shvartsman elaborated on a previous emulation of single-writer/multiple-reader shared registers developed by Attiya et al. [2]. The previous algorithm was based on processor majorities rather than quorums, and tolerated executions in which a minority of processes maintaining replicas of an object became unable to communicate either due to crashes or link failures. A majority, of course, can be viewed as a restricted form of quorum, where all of the object replicas are assigned an equal number of votes.

Our implementation of read/update objects using Atomic and Linear Multicast can also be seen as an application of quorum techniques. Essentially, our implementation of Linear Multicast require that an "update quorum" of processes are aware that an update operation has been invoked before that operation completes. The Linear Multicast algorithm then ensures that a process does not deliver read request messages until it has delivered and processed every update request message that it is aware of. When executing a read operation, a client waits until it has received replies from a set of processes, i.e., a "read quorum", that includes at least one process in any update quorum.
6.3 Future Work

We have presented sufficient criteria for enhancing an Atomic Multicast algorithm so that it also implements Linear Multicast. It remains to show what are the necessary criteria for implementing Linear Multicast. Many Atomic Multicast algorithms for asynchronous systems do not satisfy the sufficient criteria that we have outlined [51, 50, 21, 14, 17], however many of these algorithms either satisfy weaker definitions of Atomic Multicast or make stronger assumptions about the systems in which they are executed. The question of whether Linear Multicast can be added to any Atomic Multicast algorithm for asynchronous systems using failure detectors remains to be answered.

We also examined an algorithm, based on Chandra and Toueg's Atomic Broadcast algorithm, which uses Consensus to solve Atomic Multicast, and showed that a Linear Multicast can be added to the algorithm if the Consensus algorithm used any of the failure detectors considered in this thesis. Specifically, we showed that Weak Linear Multicast can be implemented if the Consensus algorithm uses $S$, and Strong Linear Multicast can be implemented if the Consensus algorithm uses $P$ or any failure detector class that includes $\Diamond P$. However, other failure detector classes can be used to solve Consensus [8]. It also would be interesting to know what type of Linear Multicast (if any) can be implemented given any failure detector class.

In our treatment of replicated shared objects, we have assumed that the replicas of each shared object are maintained by a static group of server processes. Even though some of the servers that are maintaining replicas of an object may crash, new processes cannot join a group, nor can processes be removed from a group. Thus, after sustaining a large number of failures, a group may reach a point where it can no longer properly maintain an object.

Algorithms that use *dynamic quorums* have been presented [1, 12] which ensure that a quorum can be obtained despite process failures. Essentially, when a process $p$ crashes, $p$'s votes are assigned to other processes. Also, these reconfigurations are transparent to the
clients, i.e., concurrently executing read and write operations are not interrupted. The read/write object implementation presented by Lynch and Shvartsman [39] also allows quorum configurations to be dynamically changed. The questions of when a quorum system should be reconfigured, and what the new configuration should be are usually not addressed by dynamic quorum algorithms.

It would be advantageous to be able to reconfigure a group in a similar manner, i.e., to remove crashed processes from a group, or add new processes to a group. Basically, we would like groups to be dynamic rather than static. The issue of maintaining the membership of dynamic groups is called the Group Membership problem. Unlike dynamic quorum systems, the specification of this problem usually includes determining when the membership of a group should change and what the new membership should be, as well as requiring that all processes in a group agree on the new membership.

It would be interesting to see if the techniques used to implement linearizable read/update objects can be adapted to dynamic groups. Unfortunately, even though many group membership systems have been implemented [5, 15, 42, 53], no satisfactory, or widely accepted, formal specification of the group membership problem for asynchronous systems has been presented. The existing specifications [47, 46, 27] are either incomplete or flawed. Lin and Hadzilacos [37] have recently proposed a model of group membership that does not suffer from these drawbacks. The implementation of read/update shared objects using Linear Multicast in the context of such a group membership model is an attractive topic for future research.
Appendix A

Implementing Atomic Multicast

A.1 Proof of Correctness

We now present a proof of correctness for the Atomic Multicast algorithm presented in Figure 5.2. This proof is adapted from the proof of Chandra and Toueg’s Atomic Broadcast algorithm [8]. The original Atomic Broadcast algorithm was proven to satisfy Agreement and Pairwise Total Order. We must show that this Atomic Multicast algorithm actually solves stronger properties; namely, Uniform Agreement and Local Delivery Consistency.

Lemma 5.1.1 For any group $G$, any process $p \in G$, any correct process $q \in G$, and any message $m$ such that $\text{group}(m) = G$, if $m \in \text{R.delivered}_p[G]$ then eventually $m \in \text{R.delivered}_q[G]$.

Proof: If $m \in \text{R.delivered}_p[G]$ then $p$ R-delivered $m$ (in Task 3). By the uniform agreement property of Reliable Broadcast $q$ eventually R-delivers $m$, and inserts $m$ into $\text{R.delivered}_q[G]$. □

Lemma 5.1.2 For any group $G$, any two processes $p \in G$ and $q \in G$, and all $k \geq 1$, if $p$ A-delivers messages in $\text{A.deliver}^k_p[G]$ and $q$ A-delivers messages in $\text{A.deliver}^k_q[G]$ then $\text{A.deliver}^k_q[G] = \text{A.deliver}^k_p[G]$.  

138
The proof is by induction. For \( k = 1 \), if a process \( A \) delivers messages in \( A_{\text{deliver}}^1[G] \) then, by the algorithm, it previously executed \( \text{decide}(G,1,\text{msgSet}^1[G]) \). By the uniform agreement property of Repeated Group Consensus, \( \text{msgSet}^1[G] \) is the same for all processes that decide. Since \( A_{\text{deliver}}^p[G] \) and \( A_{\text{deliver}}^q[G] \) are initially empty, and \( \text{msgSet}^1_q[G] = \text{msgSet}^1_q[G] \), we have \( A_{\text{deliver}}^p[G] = A_{\text{deliver}}^q[G] \).

Now assume that the lemma holds \( \forall k, 1 \leq k < l \). By the algorithm, if \( p \) delivers messages in \( A_{\text{deliver}}^k[G] \) then \( p \) delivered messages in \( A_{\text{deliver}}^p[G] \) for all \( k, 1 \leq k < l \). The same applies for \( q \). The algorithm also ensures that \( p \) and \( q \) previously executed \( \text{decide}(G,1,\text{msgSet}^k[G]) \). By the uniform agreement property of Repeated Group Consensus, \( \text{msgSet}^k_p[G] = \text{msgSet}^k_q[G] \). By the induction hypothesis, \( A_{\text{deliver}}^k_p[G] = A_{\text{deliver}}^k_q[G] \) for all \( k, 1 \leq k < l \). Since \( A_{\text{deliver}}^1_p[G] = \text{msgSet}^1_p[G] = \bigcup_{k=1}^{l-1} A_{\text{deliver}}^k_p[G] \) and \( A_{\text{deliver}}^1_q[G] = \text{msgSet}^1_q[G] = \bigcup_{k=1}^{l-1} A_{\text{deliver}}^k_q[G] \), it follows that \( A_{\text{deliver}}^1_q[G] = A_{\text{deliver}}^1_p[G] \). □

**Lemma 5.1.3** For any group \( G \), any process \( p \in G \) and correct process \( q \in G \), and all \( k \geq 1 \):

1. If \( p \) executes \( \text{propose}(G,k,-) \), then \( q \) eventually executes \( \text{propose}(G,k,-) \).

2. If \( p \) A-delivers messages in \( A_{\text{deliver}}^k_p[G] \) then \( q \) eventually A-delivers messages in \( A_{\text{deliver}}^k_q[G] \).

**Proof:** The proof is by simultaneous induction on (1) and (2). For \( k = 1 \), we first show that if \( p \) executes \( \text{propose}(G,1,-) \), then \( q \) eventually executes \( \text{propose}(G,1,-) \).

When \( p \) executes \( \text{propose}(G,1,-) \), \( R_{\text{deliver}}^p[G] \) must contain some message \( m \). By Lemma 5.1.1, eventually \( m \in R_{\text{deliver}}^q[G] \). Since \( A_{\text{deliver}}^1_q[G] \) is initially empty, eventually \( R_{\text{deliver}}^q[G] - A_{\text{deliver}}^q[G] \neq \emptyset \). Thus, \( q \) eventually executes Task 3 and \( \text{propose}(G,1,-) \).

We now show that if \( p \) A-delivers messages in \( A_{\text{deliver}}^k_p[G] \), then \( q \) eventually A-delivers messages in \( A_{\text{deliver}}^k_q[G] = A_{\text{deliver}}^q[G] \). From the algorithm, if \( p \) A-
delivers messages in $A_{\text{deliver}}[G]$, it previously executed $\text{propose}(G, 1, -)$. From part (1) of the lemma, all correct processes eventually execute $\text{propose}(G, 1, -)$. By Weak Termination and Uniform Integrity of Repeated Group Consensus, every correct process eventually executes $\text{decide}(G, 1, -)$ and it does so exactly once. Thus, by the algorithm, $q$ eventually A-delivers messages in $A_{\text{deliver}}[G]$.

Now assume that the lemma holds for all $k$, $1 \leq k < l$. We first show that if $p$ executes $\text{propose}(G, l, -)$, then $q$ eventually executes $\text{propose}(G, l, -)$. When $p$ executes $\text{propose}(G, l, -)$, $R_{\text{delivered}}[G]$ must contain some message $m$ that is not in $A_{\text{delivered}}[G]$. Thus, $m$ is not in $\bigcup_{k=1}^{l-1} A_{\text{delivered}}[G]$. By the induction hypothesis and Lemma 5.1.2, $A_{\text{delivered}}[G] = A_{\text{delivered}}[G]$ for all $1 \leq k \leq l - 1$. So $m$ is not in $\bigcup_{k=1}^{l-1} A_{\text{delivered}}[G]$. Since $m$ is in $R_{\text{delivered}}[G]$, by Lemma 5.1.1, $m$ is eventually in $R_{\text{delivered}}[G]$. Thus, there is a time after $q$ A-delivers $A_{\text{deliver}}[l-1][G]$ such that there is a message in $R_{\text{delivered}}[G] - A_{\text{delivered}}[G]$. So $q$ eventually executes Task 3 and $\text{propose}(G, l, -)$.

We now show that if $p$ A-delivers messages in $A_{\text{deliver}}[G]$ then $q$ A-delivers messages in $A_{\text{deliver}}[G]$. Since $p$ A-delivers messages in $A_{\text{deliver}}[G]$, it must have executed $\text{propose}(G, l, -)$. By part (1) of this lemma, all correct processes eventually execute $\text{propose}(G, l, -)$. By Weak Termination and Uniform Integrity of Repeated Group Consensus, every correct process eventually executes $\text{decide}(G, l, -)$ and does so exactly once. Thus, by the algorithm, $q$ eventually A-delivers messages in $A_{\text{deliver}}[G]$.

Lemma 5.1.4 The algorithm in Figure 5.2 satisfies the Uniform Agreement and Local Delivery Consistency properties of Atomic Multicast.

Proof: Immediate from Lemmata 5.1.3 and 5.1.2, and the fact that correct processes A-deliver messages in each batch in the same deterministic order.  

Lemma 5.1.5 (Validity) If a correct process $A$-multicasts a message $m$, then some correct process in $\text{group}(m)$ eventually delivers $m$, or no process in that group is correct.
Proof: The proof is by contradiction. Suppose a correct process \( p \) A-multicasts \( m \) to a group \( G \) and a correct process exists in \( G \) but no correct process in \( G \) ever A-delivers \( m \). By the algorithm, \( p \) R-multicasts \( m \) to \( G \) when executing Task 1. By the Validity and Uniform Agreement properties of Reliable Multicast, every correct process \( q \in G \) eventually R-delivers \( m \), and by the algorithm, inserts \( m \) in \( R_{\text{delivered}}_q[G] \). Since correct processes never A-deliver \( m \), they never insert \( m \) in \( A_{\text{delivered}}[G] \). Thus, for every correct process \( q \in G \), there is a time after which \( m \) is permanently in \( R_{\text{delivered}}_q[G] - A_{\text{delivered}}_q[G] \). From the algorithm and Lemma 5.1.3, there is a \( k_1 \), such that for all \( l \geq k_1 \), all correct processes in \( G \) execute \( \text{propose}(G, l, -) \), and they do so with sets that always include \( m \).

Since all faulty processes in \( G \) eventually crash, there is a \( k_2 \) such that no faulty process executes \( \text{propose}(G, l, -) \) with \( l \geq k_2 \). Let \( k = \max(k_1, k_2) \). Since all correct processes execute \( \text{propose}(G, k, -) \), by Termination and Uniform Agreement of Repeated Group Consensus, all correct processes execute \( \text{decide}([G]k, \text{msgSet}^k[G]) \) with the same \( \text{msgSet}^k[G] \). By Uniform Validity of Repeated Group Consensus, some process \( q \in G \) executed \( \text{propose}(G, k, \text{msgSet}^k[G]) \). From our definition of \( k \), \( q \) is correct and \( \text{msgSet}^k[G] \) contains \( m \). Thus all correct processes in \( G \) deliver \( m \). Since there is a correct process in \( G \), this is a contradiction. \( \Box \)

**Lemma 5.1.6 (Uniform integrity)** For any message \( m \), every process \( p \) A-delivers \( m \) at most once, and only if \( p \) is in \( \text{group}(m) \) and \( m \) was previously A-multicast by \( \text{sender}(m) \).

Proof: Let \( G = \text{group}(m) \). Suppose a process \( p \) A-delivers \( m \). After \( p \) A-delivers \( m \), it inserts \( m \) in \( A_{\text{delivered}}_p[G] \). From the algorithm, it is clear that \( p \) cannot A-deliver \( m \) again. From the algorithm, \( p \) executed \( \text{decide}(G, k, \text{msgSet}^k[G]) \) for some \( k \) and some \( \text{msgSet}^k[G] \) that contains \( m \). By Uniform Validity of Repeated Group Consensus, some process \( q \) must have executed \( \text{propose}(G, k, \text{msgSet}^k[G]) \). So \( q \) previously R-delivered all the messages in \( \text{msgSet}^k[G] \), including \( m \). By the Uniform Integrity property of Reliable Multicast, process \( \text{sender}(m) \) R-multicast \( m \). So, \( \text{sender}(m) \) A-multicast \( m \). \( \Box \)
Theorem 5.1.1  The algorithm in Figure 5.2 is a solution to Local Atomic Multicast.

Proof: Immediate from Lemmata 5.1.4, 5.1.5 and 5.1.6. □
Bibliography


BIBLIOGRAPHY


