INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Global flush communication primitive: Definition, implementations, and applications

Gahlot, Ashwani, Ph.D.
The Ohio State University, 1993

Copyright ©1993 by Gahlot, Ashwani. All rights reserved.
GLOBAL FLUSH COMMUNICATION PRIMITIVE: DEFINITION, IMPLEMENTATIONS, AND APPLICATIONS

DISSERTATION

Presented in Partial Fulfillment of the Requirements for the Degree Doctor of Philosophy in the Graduate School of The Ohio State University

By

Ashwani Gahlot, B.Tech., M.S.

* * * * *

The Ohio State University

1993

Dissertation Committee:

Dr. Mukesh Singhal
Dr. Phillip E. Krueger
Dr. Ten-Hwang Lai

Approved by

[Signature]
Adviser
Department of Computer and Information Science
To my Parents and my Wife
ACKNOWLEDGMENTS

I express my sincere appreciation to Dr. Mukesh Singhal for his guidance, support, and insight. Special thanks go to Drs. Steve Lai and Phil Krueger, members of my dissertation reading committee, for their suggestions and comments. Thanks also go to Dr. Mohan Ahuja. Work in Sections 2.2 and 3.1 was done when I was working with him.

Gratitude is expressed to James Chang, Bill Klingler, and other members of the NBROX development team at A T & T Bell Labs for their support and encouragement. Special thanks go to Amy Hsieh for her help.

I offer my sincere thanks to my wife, Sapna, to my parents, Surendra and Pushpa, my in-laws, Arun and Usha, and to the rest of my family for their motivation, support, and faith in me in reaching this point.
VITA

July 31, 1966 .................................................. Born - Ambala, India

Institute of Technology,
Banaras Hindu University,
Varanasi, India

1987-1992 ........................................................ Graduate Teaching/Research Associate,
at Dept. of Computer and Infr. Sc.,
The Ohio State University,
Columbus, Ohio

1988 ................................................................. M.S., Computer and Infr. Sc.
The Ohio State University,
Columbus, Ohio

1992-present ......................................... Senior Systems Analyst
at AT & T Bell Labs,
Middletown, New Jersey

Publications

Research Publications

TR-19, Department of Computer and Information Science, The Ohio State Univer-
sity, 1993.

A. Gahlot and M. Singhal. Bounding logical clocks in Distributed Systems. Technical
Report OSU-CISRC-5/93-TR-20, Department of Computer and Information Science,
The Ohio State University, 1993.

A. Gahlot and M. Singhal. Detecting Termination of a Distributed Computation us-
ing Global Flush Messages. Technical Report OSU-CISRC-10/92-TR-25, Department
of Computer and Information Science, The Ohio State University, 1992.


Fields of Study

Major Field: Computer and Information Science

Studies in:

- Distributed Systems Prof. Mukesh Singhal
- Database Systems Prof. Mohan Ahuja
- Computer Architecture Prof. D. N. Jayasimha
**TABLE OF CONTENTS**

| DEDICATION  | .......................................................... | ii |
| ACKNOWLEDGMENTS | .................................................. | iii |
| VITA  | .......................................................... | iv |
| LIST OF TABLES  | .................................................. | ix |
| LIST OF FIGURES  | .................................................. | x |

**CHAPTER**

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
</tbody>
</table>

| I Introduction | .......................................................... | 1 |
| 1.1 Background and Motivations | .......................................................... | 1 |
| 1.2 Objectives | .......................................................... | 8 |
| 1.3 Summary of Results | .......................................................... | 10 |
| 1.4 Thesis Organization | .......................................................... | 13 |

| II Global Flush Primitive | .......................................................... | 15 |
| 2.1 Preliminaries | .......................................................... | 15 |
| 2.2 Global Flush Property | .......................................................... | 18 |
| 2.3 Comparison to Related Work | .......................................................... | 23 |
| 2.4 Summary | .......................................................... | 27 |

<p>| III Implementing the Global Flush Primitive | .......................................................... | 29 |
| 3.1 A Counter-Based Implementation of GS Primitive | .......................................................... | 31 |
| 3.1.1 Encoding the Receive Before Set of a Message | .......................................................... | 32 |
| 3.1.2 Correctness Proof | .......................................................... | 35 |
| 3.1.3 Protocol Details | .......................................................... | 37 |</p>
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.4</td>
<td>Managing the Size of $R$</td>
<td>39</td>
</tr>
<tr>
<td>3.2</td>
<td>An F-channel Based Implementation of GS Primitive</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1</td>
<td>Basic Idea</td>
<td>43</td>
</tr>
<tr>
<td>3.2.2</td>
<td>Correctness Proof</td>
<td>47</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Protocol Details</td>
<td>51</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Bounding the Message Id</td>
<td>54</td>
</tr>
<tr>
<td>3.3</td>
<td>Comparison</td>
<td>56</td>
</tr>
<tr>
<td>3.4</td>
<td>A Fair and Efficient Implementation of F-channels</td>
<td>59</td>
</tr>
<tr>
<td>3.4.1</td>
<td>Motivation</td>
<td>59</td>
</tr>
<tr>
<td>3.4.2</td>
<td>KCA Implementation</td>
<td>62</td>
</tr>
<tr>
<td>3.4.3</td>
<td>$f$ and $o$</td>
<td>63</td>
</tr>
<tr>
<td>3.4.4</td>
<td>$b$ and $o$</td>
<td>66</td>
</tr>
<tr>
<td>3.4.5</td>
<td>$t$ and $o$</td>
<td>67</td>
</tr>
<tr>
<td>3.4.6</td>
<td>$f$, $b$, and $o$</td>
<td>68</td>
</tr>
<tr>
<td>3.4.7</td>
<td>$f$, $t$, and $o$</td>
<td>71</td>
</tr>
<tr>
<td>3.4.8</td>
<td>$b$, $t$, and $o$</td>
<td>72</td>
</tr>
<tr>
<td>3.4.9</td>
<td>$f$, $b$, $t$, and $o$</td>
<td>75</td>
</tr>
<tr>
<td>3.5</td>
<td>Summary</td>
<td>78</td>
</tr>
<tr>
<td>IV</td>
<td>Applications</td>
<td>80</td>
</tr>
<tr>
<td>4.1</td>
<td>Consistent Cuts, Snapshots, and Global Snapshots</td>
<td>83</td>
</tr>
<tr>
<td>4.2</td>
<td>Termination Detection</td>
<td>84</td>
</tr>
<tr>
<td>4.2.1</td>
<td>Basic Algorithm</td>
<td>85</td>
</tr>
<tr>
<td>4.2.2</td>
<td>Correctness</td>
<td>90</td>
</tr>
<tr>
<td>4.2.3</td>
<td>Enhancements and Implementation Issues</td>
<td>95</td>
</tr>
<tr>
<td>4.2.4</td>
<td>Comparison to Related Work</td>
<td>96</td>
</tr>
<tr>
<td>4.3</td>
<td>Bounding Logical Clocks</td>
<td>100</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Basic Idea</td>
<td>101</td>
</tr>
<tr>
<td>4.3.2</td>
<td>An Implementation</td>
<td>109</td>
</tr>
<tr>
<td>4.3.3</td>
<td>Correctness</td>
<td>112</td>
</tr>
<tr>
<td>4.3.4</td>
<td>Relaxing the Bounded Message Delay Constraint</td>
<td>114</td>
</tr>
<tr>
<td>4.3.5</td>
<td>Applications and Related Work</td>
<td>115</td>
</tr>
<tr>
<td>4.4</td>
<td>Shared Token</td>
<td>117</td>
</tr>
<tr>
<td>4.5</td>
<td>Updating Replicated Data with Mutual Exclusion</td>
<td>118</td>
</tr>
<tr>
<td>4.6</td>
<td>Summary</td>
<td>120</td>
</tr>
</tbody>
</table>


## List of Tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Information from which the receive before set at $p_i$ of a message can be identified.</td>
</tr>
<tr>
<td>2</td>
<td>Implementation rules for Global Flush primitive.</td>
</tr>
<tr>
<td>3</td>
<td>Summary of the data structures maintained by a sender, information carried by a message, and the data structures maintained by a receiver when only two flush message types are implemented.</td>
</tr>
<tr>
<td>4</td>
<td>Summary of the data structures maintained by a sender, information carried by a message, and the data structures maintained by a receiver when only three flush message types are implemented.</td>
</tr>
<tr>
<td>5</td>
<td>Comparison of hierarchical timestamps with vector timestamps and bit-matrix timestamps.</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

FIGURE PAGE

1 Messages exchanged in a three process distributed computation. (a) Even if $g$ is a backward flush or a two-way flush, the flush property cannot ensure that $g''$ will be received by $p_j$ after $g$ is received. (b) Even if $g$ is an forward flush or a two-way flush, the flush property cannot ensure that $g''$ is received by $p_j$ before $g$ is received. ............................................. 4

2 Naming a set of processes as a destination of a message is not the same as atomically, i.e., with no other intervening events, sending the global flush message to each destination. In (a), $g'' \notin \mathcal{F}(g)$, hence even if $g'$ is a $gf$, $g'$ may be received before $g'$ at $p_k$. In (b), $g'' \in \mathcal{F}(g)$ and hence if $g'$ is a $gf$ then the previous scenario is not possible. ............................................. 22

3 General architecture of the system: Process $p_i$ interacts with its underlying system, $S_i$, using $GS(g\text{.typ, g.dst, g.body})$ and $Rcv(g\text{.se, g.typ, g.body})$ primitives to send a message to a set of processes and to receive a message, respectively. $S_i$ sends $g$ to $S_j$. ............................................. 24

4 High level flowchart for the protocol. ............................................. 38

5 Protocol executed by $S_i$ to implement the $GS$ primitive. .............. 40

6 Figure (a) shows an F-channel network topology for a four process system. The solid directed lines denote F-channels. Figure (b) identifies the message exchange due to the diffusion process initiated by $S_i$, the min-cut, the max-cut, and the diffusion region on $S_i$. .............. 46

7 Protocol executed at $S_i$. ............................................. 55
Basic algorithm executed at each site $S_i$ to detect termination of the basic computation. ................................................. 91

The improved termination detection algorithm executed at site $S_i$. ........ 97

An illustration showing $p_i$'s windows and $p_i$'s shadow windows on $p_j$ and $p_k$. The broken arrows indicates the process time line. The solid arrows indicate $clng$ message being sent by augmented process $P_i$ to all augmented processes except itself. .......................................................... 105

An example showing $p_i$'s windows and $p_i$'s shadow windows on $p_j$ and $p_k$ satisfying conditions C1 and C2, when $W=2$. ................................. 106

Assume $W=3$. A simple (arithmetic) comparison of two clock values does not capture the relation $<_t'$ between events on different shadow windows. Events in windows $W_{i,0}$, $W_{i,1}$, and $W_{i,3}$ have $vc_i[i].\text{wid}$ value of 0, 1, and 0, respectively. Note that events in $W_{i,0}$ have $\leq_m$ relation with events in $W_{i,1}$ and events in $W_{i,3}$ do not have a $\leq_m$ with $W_{i,1}$. ............................ 108

Algorithm BVC to bound vector clocks. $(B>0, \ W>1)$. $vc[i]:=vc[i]+(0,1)$ is a short hand for $vc[i].\text{wid}:=vc[i].\text{wid}+0$, $vc[i].\text{vth}:=vc[i].\text{vth}+1$. 110

An execution illustrating that $p_j$ can see only windows $W_{i,c-1}$ and $W_{i,c}$ in $sw_{i,c}$ (Assume $W=2$). No message sent in the dotted region of the process time lines is received in $sw_{i,c}$. ................................. 111

An implementation of shared token using the $GS$ primitive. The function $\text{insert}(Q, \ gf.se)$ inserts $gf.se$ in the queue $Q$. The function $\text{empty}(Q)$ returns true iff $Q$ is empty. The function $\text{top}(Q)$ returns the value at the head of the $Q$ without removing it. The function $\text{remove-first}(Q, \ id)$ removes the first entry in the $Q$ with the value $id$. 119

A sample distributed computation. The dots represent events on a process, which itself is represented by a horizontal dotted line. The solid directed lines represent the $\prec$ relation. ......................................................... 128

$\phi_t(e_i)$ and $\phi_m(e_i)$ timestamps for each event $e_i$ in a distributed computation. 132
18 Vector timestamps associated with each event in a distributed computation. 137
19 Bit-matrix timestamps associated with events in a distributed computation. 139
20 A sample computation indicating synchronization delays. The solid directed lines indicate the ~ relation between events. 146
21 Sample computations with events having extreme values of the measure. (a) \( \gamma(e_{1,4})=0 \), i.e., \( e_{1,4} \) perceives no concurrency at all. (b) \( \gamma(e_{1,4})=1 \), i.e., \( e_{1,4} \) perceives that all events are concurrent to it and no time is spent in synchronization delay. 149
22 Values of \( \gamma \) for each event in the computation. 150
23 Values of \( \gamma \) for various executions of a distributed computation. 152
24 Even though event \( e_{1,5} \) has more events concurrent to it in (a) than in (b), the value of \( \alpha(e_{1,5}) \) is the same. 156
25 Even though the computation in (b) has more synchronization delay than the computation in (a), they have the same value for \( \beta \) for events \( e_{2,7} \) and \( e_{2,3} \). 159
CHAPTER I

Introduction

1.1 Background and Motivations

A distributed computation consists of a set of processes which cooperate to achieve a common goal. Processes in a distributed computation do not share common memory and communicate solely by message passing. A process is a set of events. An Event on a process could be an internal event or an event of sending and receiving a message (i.e., events for interprocess communication).

Communication Primitives

Interprocess communication mechanisms provide abstractions through which processes exchange messages. Typically, such communication abstractions support primitives to send and receive messages. Communication abstractions can be characterized by (a) the number of possible recipients of a message, (b) the communication level concurrency of the abstraction, and (c) the ease with which the abstractions can be used in the design and development of distributed algorithms. Given a set of messages that have been sent under an abstraction, the communication level concurrency of the abstraction characterizes the number of possible orders in which these messages can be received.
Traditionally, communication abstractions have been modeled as either FIFO or non-FIFO, one-to-one communication \([11, 14, 15, 16, 35, 43]\) abstractions. FIFO, one-to-one communication abstraction simplifies the design and the development of distributed algorithms by presenting a simplified view of the communication between processes. Messages sent under this abstraction can be received only in the order they are sent and thus their communication level concurrency is minimal. Non-FIFO, one-to-one communication abstraction on the other hand, provides maximum communication level concurrency since messages may be received in any order. The algorithm design gets complicated because the mechanisms for ordering the receipt of two messages are a part of the algorithm itself. In addition, the proofs of correctness of the algorithms designed using this abstraction are involved.

**F-channels:** F-channels \([3]\) have been proposed as an alternative to FIFO and non-FIFO, one-to-one communication abstractions. An F-channel is a non-FIFO channel that supports *flush property*, which is based on ordering the receipt of a message with respect to all messages sent before and/or after it along the same channel. It supports four message types, namely, *forward flush*, *backward flush*, *two-way flush*, and *ordinary*. A forward flush message sent along an F-channel is received after all messages sent along the F-channel before sending the forward flush message have been received. A backward flush message sent along an F-channel is received before any message sent along the F-channel after sending the backward flush message is received. A two-way flush message sent along an F-channel is received after all messages sent along the F-channel before sending it have been received but before any message sent
along the F-channel after sending the two-way flush message is received. An ordinary message sent along an F-channel can be received at any time as long as its receipt does not violate requirements specified by other messages. Depending on the flush message types used in the computation, F-channels support a spectrum of communication level concurrency ranging from that of FIFO to non-FIFO channels. F-channels simplify the algorithm design by pushing the mechanisms for message receipt ordering into the communication layer, and hence, also simplify the proof of correctness of the algorithm. However, F-channels do not provide adequate mechanisms for ordering the receipt of a messages sent on different channels and for one-to-many communication between processes.

The flush property [3] enforces receipt ordering restrictions only on messages sent along a single channel and cannot be used to order the receipt of messages sent by different processes. Thus, a message can be indirectly crossed by a sequence of messages. We explain this with an example. Consider a distributed computation consisting of three processes \( p_i, p_j, \) and \( p_k. \) Suppose it is desired (see Figure 1(a)) that no message sent by \( p_i \) after sending \( g \) directly or indirectly crosses \( g, \) i.e., it is desired that all messages, sent to \( p_j, \) whose send event is in the future \(^1\) of \( g \)'s send event be received only after \( g \) is received by \( p_j. \) Even if \( g \) is a backward flush or a two-way flush message, this cannot be enforced by flush property since it cannot ensure that message \( g' \) in the sequence \( (g', g') \) is received after \( g \) at \( p_j. \) Similarly, suppose that it is desired (see Figure 1(b)) that no message sent by \( p_i \) before sending

---

\(^1\) The terms "past" and "future" are based on Lamport's [44] "happened before" relation and will be formally defined later. The past of a message is the set of messages sent by any process in the system before this message was sent. Similarly, future is defined.
$g$ directly or indirectly crosses $g$, i.e., it is desired that all messages, sent to $p_j$, whose send event is in the past of $g$'s send event be received before $g$ is received by $p_j$. This cannot be enforced by the flush property, i.e., even if $g$ was a forward flush or a two-way flush message, the flush property cannot ensure that message $g''$ in the sequence $(g', g'')$ is received before $g$ at $p_j$. As shown in [26, 28], global message receipt ordering finds applications in the design of distributed algorithms. In both the cases above, we have shown that despite the flush property, a message can be indirectly crossed by a sequence of messages.

![Diagram](image)

Figure 1: Messages exchanged in a three process distributed computation. (a) Even if $g$ is a backward flush or a two-way flush, the flush property cannot ensure that $g''$ will be received by $p_j$ after $g$ is received. (b) Even if $g$ is an forward flush or a two-way flush, the flush property cannot ensure that $g''$ is received by $p_j$ before $g$ is received.

**One-to-Many Communication Primitives:** Many distributed applications need a message to be sent from one process to a group (possibly all) of processes in the system. These applications include commit protocols, mutual exclusion, replicated data update, and global property detection, among many others. Communication abstractions which support one-to-many and one-to-all communication have been
studied [10, 17, 9, 53, 54, 55, 60, 61]. These abstractions provide different message receipt ordering criteria. They also support different communication level concurrency varying from a total ordering on all message receipts (GBCAST messages of [10]) to no ordering at all. We propose a one-to-many communication abstraction which permits a global message receipt ordering and supports a higher communication level concurrency than most communication abstractions proposed in the past. Messages sent under this abstraction can be ordered with respect to messages sent in its past and/or its future.

**Viewing a Process as a Partially Ordered Set of Events**

Traditionally, a process is viewed as a single thread of flow control, i.e., as a sequence of events. In light of development of abstractions like threads [6, 66], a process can no longer be viewed as a sequence of events but must be viewed as a partially ordered set of events. A process may consist of multiple threads of flow control. When such a process is implemented in a multiprocessors environment (e.g., Solaris operating system of Sun Microsystems), events on two different threads may actually be executed simultaneously on different processors. Note that each thread could itself be viewed as a “process”. In this case however, the size of data structures in algorithms which order events on different processes (e.g., algorithms using vector clocks [22, 47]) becomes very large. We, thus, argue that it is more natural to view a process as a partially ordered set of events. This view of a process can also be used to model multiple processes of an application which execute at the same site.
and communicate via shared memory. By viewing a process as a partially ordered set of events, all processes at a site can be viewed as one big process at the site. Apart from providing a better representation of the system, this model simplifies the data structures kept at each site. A similar approach is taken in [58].

Viewing a process as a partially ordered set of events results in identification of all interprocess and intraprocess concurrency. It also presents a unified framework to reason about both sequential and parallel/distributed computation. The partial order between events in a distributed computation is described by a relation “affects” [4] (also referred to as “causality” relation).

**Clocks:** Bit-matrix clocks [5] have been proposed to assign timestamps to events to identify the “causality” relation between them. Bit-matrix clocks, however, have high cost: each timestamp is a vector of $N$ components (where $N$ is the number of processes in the distributed computation) where the $i^{th}$ component contains one bit for each event on the $i^{th}$ process. Inferring the “causality” relation has applications in debugging of distributed computations and in identifying potentially concurrent events. We propose hierarchical clocks to capture the “causality” relation. Hierarchical clocks have substantially lower storage and communication overhead when compared to bit-matrix clocks.

**Concurrency Measures:** Quantifying the number of events in a distributed computation that occur concurrently is an interesting characterization of a distributed computation. As opposed to traditional characterizations of distributed computa-
tions, which are based on the number of messages exchanged, total information exchanged by messages, or the total execution time of the computation, measures quantifying the number of potentially concurrent events in the distributed computation (or concurrency measures) provide a better characterization of the concurrency in the computation. Recently, measures have been proposed [18, 19, 56] which attempt to characterize a distributed computation based on the above criterion. They, however, assume that a process is a total order of events, i.e., two events on a process cannot be executed concurrently even if sufficient number of processors were available. Concurrency between events in a distributed computation, when a process is viewed as a sequence of events, is referred to as “explicit concurrency”. These measures, thus, attempt to quantify the “explicit concurrency” between events in a distributed computation; they do not characterize the number of events on the same process. Concurrency between events in a distributed computation, when a process is viewed as a partially ordered set of events, is referred to as “inherent concurrency”. The measures proposed in [18, 19, 56] do not capture the inherent concurrency in the computation. Fidge [23] attempts to capture inherent concurrency, but does so only for a restricted version of the “causality” relation. A measure which characterizes the inherent concurrency in a computation could aid in further distributing/consolidating the events in a distributed computation. We propose concurrency measures to capture the inherent concurrency in a distributed computation.
1.2 Objectives

In this dissertation, we investigate two related problems in the design, development, and analysis of distributed algorithms. First, we develop a one-to-many communication abstraction \( GS \), for Global Send, which allows the sender to order the receipt of a message with respect to messages sent in its past and/or future. (We also refer to \( GS \) as the Global Flush communication primitive.) We also discuss implementations and applications of this communication abstraction. Second, we model a process as a set of events and a partial order on them. We propose hierarchical clocks to capture this partial order. We also propose concurrency measures to capture inherent concurrency in a distributed computation.

Global Flush Communication Primitive

Analogous to F-channels, the proposed one-to-many communication abstraction \( GS \) supports four message types, namely, global backward flush, global forward flush, global two-way flush, global ordinary message type. The \( GS \) communication primitive has two properties: First, it ensures the global flush property (formally defined later) specified by the message type. The global flush property is an extension of the flush property to one-to-many communication and is based on ordering of the receipt of a message with respect to receipt of messages in its past and/or in its future. Second, it takes as a parameter a named group or a set of processes as the destination of the message. Use of this primitive provides an elegant way to reason about message receipt orderings (as it allows a process to deduce the message receipt orderings at other
processes), simplifies algorithm development, and permits higher communication level concurrency than many communication primitives proposed in the literature. Use of the $GS$ primitive simplifies algorithm development in a distributed environment because the receipt ordering is ensured by the communication layer as opposed to by the application itself. We discuss two implementations and two applications of the $GS$ primitive.

**Viewing a Process as a Partially Ordered Set of Events**

**Hierarchical Clocks:** Although bit-matrix clocks [5] can capture the causality relation in a distributed computation, they have a very high cost. We argue that the causality relation in a distributed computation should be viewed as a hierarchical partial order. We introduce the notion of hierarchical clocks to capture this hierarchical partial order. Hierarchical clocks provide the following advantages for applications such as debugging of distributed programs: First, hierarchical clocks can precisely capture causality. Second, the storage overhead at each process due to timestamps is greatly reduced. Third, the communication overhead for each message in terms of size of the timestamp carried by it is greatly reduced. This is specially advantageous because in a typical debugging session, the timestamps of relatively few events are ever compared. Thus, additional overhead of maintaining timestamps of events not considered in a debugging session are kept as low as possible.

**Concurrency Measures:** We further explore measures of concurrency in a distributed computation. We propose concurrency measures which characterizes the
inherent concurrency as observed by an event in a distributed computation and by the entire distributed computation. Along with traditional measures like total execution time, these measures give a better characterization of a distributed computation.

1.3 Summary of Results

Global Flush Communication Primitive: In this dissertation, we propose the Global Flush communication primitive which allows the sender to send a message to a group of receivers such that the receipt of the message is ordered with respect to messages sent in its past and/or future. We present two implementations of the GS primitive.

The first is a counter-based implementation [26] in which each process maintains counters and a message carries the value of the counters at the time it is sent. These counters contain enough information for the recipient to determine if a message can be received without violating the global flush property.

Since the GS primitive is motivated by F-channels, it is natural to see how the GS primitive can be implemented on top of F-channels. This is done in the second implementation [28]. This implementation uses a diffusion process [20] which results in a flush message to be sent along each F-channel in the system. In this implementation, each message carries just one bounded integer and the information kept at each site is bounded by the size of this integer. The overhead, both in terms of the amount of information carried by a message and the amount of information kept at each site, is independent of the number of processes in the system.

We propose an efficient and fair implementation of F-channels [3]. The imple-
mentation of F-channels proposed in the past penalize computations which use fewer flush messages because they increase the information that needs to be maintained by the receiver and they increase the size of information carried by each message. The implementations proposed here alleviates these problems.

The GS primitive has been found to be useful in a variety of applications [26, 28, 29, 30]. We discuss some applications exemplifying its use. Specifically, we discuss applications of the primitive in the following areas: obtaining consistent cuts, snapshots, and global snapshots; termination detection; bounding logical clocks; implementing shared token; and updating replicated data with mutual exclusion.

We propose an algorithm to detect termination of a distributed computation [30]. In this algorithm, the messages for termination detection are processed and sent out by a process only when it is idle. It is expected that these messages will not interfere much with the computation. Each message used to detect termination carries only a boolean value and thus, incurs very little communication overhead. The proposed algorithm is symmetric in the sense that any arbitrary process can start the algorithm and multiple invocations can be active in parallel. Marker based algorithms [25, 39, 42, 46] to detect termination in a distributed environment could potentially result in unbounded number of messages to be exchanged due to unsuccessful termination detection attempts. The proposed algorithm, on the other hand, uses bounded number of messages even in the worst case.

We define the notion of bounded clocks [29]. We then present an algorithm to bound vector clocks [22, 47], under the assumption of bounded message delay, in
a system that supports the global flush communication primitive GS. We discuss applications exemplifying the advantage of such bounded clocks. We modify this algorithm to bound vector clocks when the assumption of bounded message delay is removed. However, this requires inhibition on sending of some of the messages.

**Viewing a Process as a Partially Ordered Set of Events:** We propose hierarchical clocks [33] which precisely capture causality among events. The storage overhead at each process for the timestamps is greatly reduced when hierarchical clocks are used. The communication overhead for each message in terms of size of the timestamp carried by the message is also greatly reduced. This is specially advantageous to applications like debugging of distributed programs because in a typical debugging session, the timestamps of relatively few events are ever compared. Thus, additional overhead of maintaining timestamps of events not considered in a debugging session are kept as low as possible.

We also propose concurrency measures [32] which characterize the inherent concurrency as observed by an event in a distributed computation and by the entire distributed computation. Along with traditional measures like total execution time, these measures give a better characterization of a distributed computation. Such measures aid in further distributing/consolidating the events in a distributed computation by transforming a program to exploit the inherent concurrency. By observing the value of the concurrency observed by each event and the entire computation, the effectiveness of the program transformation can be measured.
1.4 Thesis Organization

This dissertation is organized as follows: In Chapter II, we describe the system model and give preliminary definitions. We describe the $GS$ primitive, which allows the sender to send a message to a group of receivers such that the receipt of the message is ordered with respect to receipt of other messages. Based on the communication level concurrency, we compare the $GS$ primitive with the one-to-many communication primitives described in ISIS [8, 10, 40], Psync in x-kernel [53], causally ordered computation class in [12], causal broadcasts in [9, 54], and reliable broadcast protocols [17].

In Chapter III, we describe two implementations [26, 28] of the global flush property. The first implementation is a counter-based implementation and it assumes that the underlying system is connected by reliable, non-FIFO communication channels. The second implementation is an F-channel based implementation and it assumes that the underlying system in connected by point-to-point F-channels. This implementation gives insight into the relation between point-to-point flush primitives and the global flush primitive. We compare the two implementations with each other and with implementations of other one-to-many primitives proposed in the past [8, 9, 10, 17, 40, 53, 54]. The comparison is in terms of the amount of additional information carried by a message, the size and the type of information maintained by each site, and the number of messages sent in the underlying system to enforce the permitted message receipt ordering. We also propose a fair and efficient counter-based implementation of the flush primitives which alleviates the problems in the
existing implementations. In this implementation, computations that use fewer flush messages have smaller overhead.

In Chapter IV, we discuss the following applications of the GS primitive: obtaining consistent cuts, snapshots, and global snapshots; termination detection; bounding logical clocks; implementing shared token; updating replicated data with mutual exclusion.

In the Chapters V and VI, we model a process as a partially ordered set of events.

In Chapter V, we propose hierarchical clocks which, like bit-matrix clocks, precisely capture causality between events. We show that maintaining hierarchical clocks results in reduced storage overhead at processes and reduced communication overhead for each message. We discuss the utility of such clocks in debugging of distributed programs.

In Chapter VI, we propose measures which capture the inherent concurrency perceived by an event and inherent concurrency perceived by the computation as a whole. We describe how to compute the measures at runtime. We compare them with other measures proposed in the past.

In Chapter VII, we summarize the results of this dissertation and discuss future extensions.
CHAPTER II

Global Flush Primitive

In this chapter, we describe the system model and give preliminary definitions. We define the global flush property which is satisfied when a global flush message is received by a process. Global flush messages are sent using the GS primitive. This primitive finds applications in the design and development of distributed algorithms. Some applications, illustrating the use of the GS primitive, are discussed in the next chapter. We give an example to show why naming a group as the destination of a global flush message is important and why it cannot be simply implemented by atomically sending a global flush message to each destination of the global flush message. We then discuss some one-to-many communication primitives proposed in the past. Specifically, based on the communication level concurrency, we compare the GS primitive with the one-to-many communication primitives described in ISIS [8, 10, 40], Psync in x-kernel [53], causally ordered computation class in [12], causal broadcasts in [9, 54], and reliable broadcast protocols [17].

2.1 Preliminaries

The system under consideration consists of a set of $N$ processes, $p_0, \ldots, p_{N-1}$, some of which are connected by non-FIFO channels (let $c_{i,j}$ denote a channel from $p_i$ to
such that there is a path from each process to every other process. The channels are reliable in that messages are not added, lost, or mutated in any way. It is also assumed that messages are delivered in arbitrary, but finite amount of time. A process is a sequence of events. An event on a process could either be an internal event or an event of sending or receiving a message. An internal event, denoted by \textit{Int}, does not directly affect events on other processes. Let \( gsend(g) \) denote the event of sending \( g \) to its specified group \(^1\) of destination processes in the system. We use \( g.se \) to denote the sender of \( g \). The event of \( p_j \) receiving \( g \) is denoted by \( grecv(g)_j \). For every \( grecv(g)_j \) event, there is a corresponding \( gsend(g) \) event. We concern ourselves only with events of type \( gsend(g) \) and \( grecv(g)_j \) because they directly affect events on other processes. We use \( e_i \) and \( e'_i \) to denote events on \( p_i \). Lamport’s “happened before” relation \([44]\), denoted by \( \xrightarrow{\text{lam}} \), describes a partial order on the set of events, denoted by \( \mathcal{E} \), in a distributed system.

**Definition 2.1.1 [\( \xrightarrow{\text{lam}} \), to be read as “happened before” ]**

The relation \( \xrightarrow{\text{lam}} \) on \( \mathcal{E} \) is the transitive closure of relation \( \rightarrow \), where relation \( \rightarrow \) is defined as follows: (a) If \( e_i \) and \( e'_i \) are events on the same process \( p_i \) such that \( e_i \) occurs before \( e'_i \), then \( e_i \rightarrow e'_i \). (b) If \( e_i \) is the event of sending a message and \( e_j \) is the event of receiving the same message, then \( e_i \rightarrow e_j \).

**Definition 2.1.2 [Message Chain]**

A sequence of \( n \) messages \( g_0, \ldots, g_{n-1} \), s.t. \( \forall k: 1 \leq k \leq n - 1 : grecv(g_{k-1})_{g_k} \rightarrow \)

\(^1\)The size of the group can vary from 1 to \( N \). When the group is of size 1, then the message is a one-to-one message. When this group contains all processes in the system, the message is a broadcast.
\( \text{gsend}(g_n) \), is defined as a message chain of length \( n \) between \( g_0 \) and \( g_{n-1} \).

We next define the past and future of a message. Past and future of a message identify a set of messages based on the \( \xrightarrow{\text{iam}} \) relation between their corresponding send event.

**Definition 2.1.3** ["\( \mathcal{P}(g) \)", to be read as “the past of \( g \)""]

For any message \( g \), define \( \mathcal{P}(g) = \{ g' : \text{gsend}(g') \xrightarrow{\text{iam}} \text{gsend}(g) \} \).

The past of a message \( g \) is the set of messages whose send events “happened before” the send event of \( g \). In other words, the past of \( g \) includes all messages that were sent/received by \( g.se \) before \( g \) was sent and all messages in their respective pasts.

Note that the past of \( g \) may include messages sent by processes other than \( g.se \).

**Definition 2.1.4** ["\( \mathcal{F}(g) \)", to be read as “the future of \( g \)""]

For any message \( g \), define \( \mathcal{F}(g) = \{ g' : \text{gsend}(g') \xrightarrow{\text{iam}} \text{gsend}(g) \} = \{ g' : g \in \mathcal{P}(g') \} \).

The future of a message \( g \) includes all messages \( g' \) such that the event of sending \( g \) “happened before” the event of sending \( g' \). Note that the future of \( g \) may include messages sent by processes other than \( g.se \). The future of \( g \), thus, includes all messages that are sent by \( g.se \) after sending \( g \) or are sent by \( p_j \) after receiving a message (which may be \( g \)) from \( g.se \) that was not sent before \( g.se \) sent \( g \) and messages in their respective future.

Given a set \( \mathcal{M} \) of global flush messages we define, for notational convenience, its senders projection and its receivers projection on a process. \( \mathcal{M} \)'s receivers projection on \( p_j \) is the set of all messages in \( \mathcal{M} \) which have \( p_j \) as a destination.
Definition 2.1.5 \([\mathcal{M}]_j\), to be read as “\(\mathcal{M}\)’s receiver projection on \(p_j\)”

Given any set \(\mathcal{M}\) of messages, define \([\mathcal{M}]_j = \{g: (g \in \mathcal{M}) \land (g.\text{dst}[j] = 1)\}\).

\(\mathcal{M}\)’s senders projection on \(p_i\) is the set of all messages in \(\mathcal{M}\) sent by \(p_i\).

Definition 2.1.6 \([\mathcal{M}]_i\), to be read as “\(\mathcal{M}\)’s senders projection on \(p_i\)”

Given any set \(\mathcal{M}\) of messages, define \([\mathcal{M}]_i = \{g: (g \in \mathcal{M}) \land (g.\text{src} = i)\}\).

2.2 Global Flush Property

Global flush messages are of following four types: GB, GF, GT, or GO (to be read as global backward flush, global forward flush, global two-way flush, and global ordinary messages, respectively). Global flush messages are sent using the Global Send primitive, \(\text{GS}(g.\text{typ}, g.\text{dst}, g.\text{body})\), where \(g.\text{typ}\), \(g.\text{dst}\), and \(g.\text{body}\) are defined as follows:

The message type \(g.\text{typ}\), which could be any one of GB, GF, GT, or GO, specifies that Properties \(\text{Gb}, \text{Gf}, \text{Gt}, \text{or Go}\) (defined later), respectively, should be ensured between receipt of \(g\) and receipt of messages in \(\mathcal{P}(g)\) and/or messages in \(\mathcal{F}(g)\). The message destination \(g.\text{dst}\) specifies the group of processes to which the body \(g.\text{body}\) of \(g\) is to be sent. The sender of a message can also name itself as one of the recipients of the message. Notationally, \(g.\text{dst}\) is represented as a vector of \(N\) integers such that \(g.\text{dst}[j] = 1\) if \(g.\text{body}\) is to be sent to \(p_j\) and \(g.\text{dst}[j] = 0\), otherwise. Thus, \(p_j\) is in \(g.\text{dst}\) iff \(g.\text{dst}[j] = 1\). Each invocation of \(\text{GS}(g.\text{typ}, g.\text{dst}, g.\text{body})\) by \(p_i\) corresponds to a \(\text{gsend}(g)\) event on \(p_i\).
We refer to $g$ as a $gb$, $gf$, $gt$, or $go$ message if $g.typ$ takes values of GB, GF, GT, or GO, respectively. We next define the four Global Flush properties [26].

**Property Gb** ["Global backward flush" property]

For each "gb", the following property holds at each process $p_j$ in $gb.dst$:

$$
\forall g: g \in F(gb) \land g.dst[j]=1: grecv(gb)_j \overset{\text{lam}}{\rightarrow} grecv(g)_j.
$$

A message $g'$, that is sent either by $p_{gb,te}$ after sending $gb$ or by a $p_j$ after $p_j$ has received $gb$ or a message that is in the future of $gb$, cannot be received by any process in $gb.dst$ before $gb$ is received. Thus, a $gb$ can be viewed as the fastest possible message in the system.

**Property Gf** ["Global forward flush" property]

For each "gf", the following property holds at each process $p_j$ in $gf.dst$:

$$
\forall g: g \in P(gf) \land g.dst[j]=1: grecv(gf)_j \overset{\text{lam}}{\rightarrow} grecv(gf)_j.
$$

A message $g'$, that is in the past of $gf$ will be received by all processes in $gf.dst$ before $gf$ is received. Thus a $gf$ can be viewed as the slowest possible message in the system.

**Property Gt** ["Global two-way flush" property]

For each "gt", the following holds at each process $p_j$ in $gt.dst$:

$$
Gt(a) \forall g: g \in F(gt) \land g.dst[j]=1: grecv(gt)_j \overset{\text{lam}}{\rightarrow} grecv(g)_j \text{ and }
$$

$$
Gt(b) \forall g: g \in P(gt) \land g.dst[j]=1: grecv(gt)_j \overset{\text{lam}}{\rightarrow} grecv(gt)_j.
$$

---

2In this thesis we use the following notational convention: A global backward flush message is denoted by $gb$ (both lower-case). It carries its type GB (both upper-case) and satisfies Property $Gb$ (upper-case G and lower-case b). Similarly, for $gf$, $gt$, and $go$. 
Thus, a $g_t$ satisfies both $G_b$ and $G_f$ properties.

**Property $G_o$ ["Global ordinary flush" property]

No ordering, with respect to any other global flush messages, is imposed on receipt of a "go".

A $g_o$ satisfies neither Property $G_b$ nor Property $G_f$ and it can be received in any order with respect to other messages, as long as the global flush properties dictated by other messages are not violated.

Properties $G_b$, $G_f$, $G_t$, and $G_o$ are collectively referred to as Property $G$ or the **Global Flush property**.

The primitive $Rcv(var\ g.se,\ g.typ,\ g.body)$ is used to receive a global flush message. $Rcv$ is a blocking primitive. On an invocation of $Rcv(g.se,\ g.typ,\ g.body)$ by $p_j$, the sender $g.se$, the type $g.typ$, and the body $g.body$ of $g$ are returned to $p_j$ whenever a $g$ exists at $p_j$ such that its receipt at $p_j$ does not violate Property $G$. Each invocation of $Rcv(g.se,\ g.typ,\ g.body)$ by $p_j$ corresponds to a $grecv(g)_j$ event.

The relation $\rightarrow$, defined next, describes the message receipt ordering permitted by the Property $G$.

**Definition 2.2.1 ["$\rightarrow$", to be read as “permitted message receipt ordering”]**

Define $g \rightarrow g'$ iff $\exists j \ g \xrightarrow{i} g'$, where $\xrightarrow{i}$ is defined as follows:

\[ g \xrightarrow{i} g' \Leftrightarrow gsend(g) \xrightarrow{l} gsend(g') \land g.dst[j]=g'.dst[j]=1 \land \]
\[ ((g.typ=GB \lor g.typ=GT) \lor (g'.typ=GF \lor g'.typ=GT)) \]

The relation $\rightarrow$ identifies the permitted message receipt orderings between various message types such that Property $G$ is not violated. $g \rightarrow g'$ means that $g$ must be
received before $g'$. Note that $\rightarrow$ is derived from \( \rightarrow_m \) relation between message sends and is a subset of \( \rightarrow_m \), i.e., $g \rightarrow g' \implies \text{gsend}(g) \rightarrow_m \text{gsend}(g')$ but not necessarily vice-versa. $\rightarrow$ is thus an acyclic partial order. It is irreflexive, asymmetric, and transitive. Note that $gb$ and $gt$ satisfy $\rightarrow$ relation with all messages in their future that have at least one common destination. Similarly, $gf$ and $gt$ satisfy $\rightarrow$ relation with all messages in their past that have at least one common destination.

**Power of Naming a Group as a Destination**

The ability to name a set of processes as the destination of a global flush message is important [10, 53] because it enables the sender and receiver(s) of a message to infer message receipt ordering at other receivers. As it will become clear later, sending a copy of the message to each member of a set of processes, with no other intervening sends/receipts, does not have the same effect as designating the set as a recipient of the message. We have found the latter property to be stronger and much more useful in reasoning about messages than the former. We illustrate the distinction between them in Figure 2 with an example. In this example, all messages satisfy the global flush property. Messages $g$ and $g'$ satisfy $Gb$.

Consider a system consisting of three processes $p_i$, $p_j$, and $p_k$ connected with each other by non-FIFO channels. Assume that both $p_j$ and $p_k$ keep a log of the messages sent to and received from each other. We use $S_j$, $S_k$ and $R_j$, $R_k$ to denote the logs kept by $p_j$ and $p_k$, respectively. Intuitively, at any global real time $R_k \subseteq S_j$ and $R_j$

---

3Both messages have an identical process as one of their destinations.

4Assuming the number of processes in this set is more than 1.
Periodically, $p_i$ sends requests to both $p_j$ and $p_k$ to collect the values of $S_j$, $R_j$, $S_k$, and $R_k$. Clearly, for the values of the logs to be consistent (i.e., logs should be such that $R_k \subseteq S_j$ and $R_j \subseteq S_k$), all messages sent by $p_j$ to $p_k$ after it has sent its logs to $p_i$ should be received after $p_k$ has sent $p_k$'s logs to $p_i$ and vice-versa.  

If the request from $p_i$, for the logs, to $p_j$ and $p_k$ is sent using two distinct messages (as with $g$ and $g'$ in Figure 2(a)) then even assuming the global flush property the logs collected may not be consistent. A message $g''$ sent by $p_j$ after receiving $g$ may be received by $p_k$ before receiving $g'$ because $g'' \notin \mathcal{F}(g)$.

Instead, if the same message is sent with $p_j$ and $p_k$ as destinations (as with $g$ in Figure 2(b)), then the requirement for consistent logs is satisfied. The logs collected by $p_i$ will always be consistent.

Figure 2: Naming a set of processes as a destination of a message is not the same as atomically, i.e., with no other intervening events, sending the global flush message to each destination. In (a), $g'' \notin \mathcal{F}(g)$, hence even if $g'$ is a $gf$, $g''$ may be received before $g'$ at $p_k$. In (b), $g'' \in \mathcal{F}(g)$ and hence if $g'$ is a $gf$ then the previous scenario is not possible.

---

5This is in fact the backward-flush property.
General Architecture of the System

Figure 3 describes the general architecture of the system considered in this thesis. Process $p_i$ communicates with process $p_j$ in the system by invoking the $GS(g\.typ, g\.dst, g\.body)$ and $Rcv(\text{g}.se, g\.typ, g\.body)$ primitives. To accomplish this, $p_i$ actually interacts with the underlying system, $S_i$, for $p_i$. The underlying system implements the global flush property. When $p_i$ invokes $GS(g\.typ, g\.dst, g\.body)$, $S_i$ sends $g$ to all $S_j$'s such that $p_j \in g\.dst$. A global flush message is said to be received by $p_j$ when its body is taken up for processing by $p_j$. A global flush message can be received by $p_j$ only when its receipt by $p_j$ will not violate Property $G$. Such a global flush message (one which has not been received yet by $p_j$ and its receipt will not violate Property $G$) is said to be receivable at $p_j$. A message that has arrived at $S_j$ from the communication medium is said to be delivered at $p_j$. A message delivered at $p_j$ may not yet be receivable at $p_j$. A global flush message $g$ can be received by a process only when $g$ satisfies the Property $G$ with respect to other global flush messages in the system.

2.3 Comparison to Related Work

In this section, we compare $GS$ with broadcast primitives proposed in the literature. The comparison is in terms of the communication level concurrency permitted by the primitives, i.e., in terms of number of possible message receipt orderings permitted by the primitive. Comparison of specific implementations of the $GS$ primitive with those of other primitives in terms of the amount of information carried by a message, the
Invokes $GS(g.\text{typ}, g.\text{dst}, g.\text{body})$ or $Rcv(g.\text{se}, g.\text{typ}, g.\text{body})$

On invocation of $Rcv$ by $p_i$, 
$<g.\text{se}, g.\text{typ}, g.\text{body}>$ is 
returned such that $g$ 
satisfies Property $G$

Figure 3: General architecture of the system: Process $p_i$ interacts with its underlying system, $S_i$, using $GS(g.\text{typ}, g.\text{dst}, g.\text{body})$ and $Rcv(g.\text{se}, g.\text{typ}, g.\text{body})$ primitives to send a message to a set of processes and to receive a message, respectively. $S_i$ sends $g$ to $S_j$. 
size and type of information maintained by each site to enforce the required message
receipt ordering, and the number of messages exchanged to enforce the message receipt
ordering is done later in Chapter III.

The \textit{Psync} primitive in x-kernel [53] generates a context relationship for each
message $g$, where $g$ is in the context of $g'$ if $g' \in \mathcal{P}(g)$. $g$ is received only after all
messages in $\mathcal{P}(g)$ have been received. The set of message receipt orderings permitted
by the \textit{Psync} primitive is a subset of message receipt orderings permitted by \textit{GS}. That
is, for any given sequence of message sends, all message receipt orderings permitted
when \textit{Psync} is used are also achievable when \textit{GS} primitive is used and there may be
some message receipt orderings permitted when \textit{GS} is used which are not achievable
under \textit{Psync}.

If it is desired that only the message receipt orderings permitted by \textit{Psync} be
allowed under \textit{GS}, then the following must be ensured:
$\forall g, g', j : g'.dst[j] = 1 \land g'$ is the last message received from $p_{g',sc}$ by $p_j$ before sending $g$:

$(g'.typ, g.typ) \notin \{(GO,GO), (GO,GB), (GF, GO), (GF,GB)\}$.

If $g' \in \mathcal{P}(g)$, then $g'$ must be received before $g$ at all common destinations. Since
$gf$, $gb$, and $go$ do not order this receipt with respect to receipt of all messages in
the system, allowing the above message type pairs will violate the message receipt
ordering permitted by \textit{Psync}.

Another difference between \textit{Psync} and \textit{GS} is that in \textit{Psync} each message is sent
to all processes in the system, as opposed to a subset of processes in \textit{GS}.

\textit{ISIS} [10] proposes a suite of three communication primitives: GBCAST, ABCAST,
and CBCAST. Unlike ABCAST and CBCAST, a message sent using the GBCAST
primitive is ordered with respect to all messages in the system and all failure events.
A Message sent using ABCAST is totally ordered with respect to all other ABCAST
messages. Concurrent ABCASTs\(^6\) are also ordered. The order of receipt of ABCASTs
is not known a priori [10]. The \(GS\) primitive is more closely related to the ISIS
CBCAST primitive. In CBCAST, a label "clabel" is associated with each message.
The ordering among message receipts is specified in terms of a "less than" relation,
denoted by \(\text{isis}\), based on these labels. The kernel at each site knows this "less than"
relation. Messages are received in an order that does not violate the \(\text{isis} \cap \text{lam}\) relation.
A message receipt ordering permitted by \(GS\) is also permitted using CBCAST by
appropriately defining the labels. Since the message receipt ordering is a subset of \(\text{lam}\)
relation, it can be captured by the bit-matrix timestamping scheme proposed in [5].
These bit-matrix timestamps can in turn be used as clabels.\(^7\) As it is described later,
the cost of implementing bit-matrix timestamps is very high.

In the new version of ISIS [9], the relation \(\text{isis}\) is the same as \(\text{lam}\). Thus, except
for the notion of process groups, this is similar to \(Psync\). The relation \(\text{isis}\) is encoded
using vector clocks [22, 47].

Chang and Maxemchuk [17] describe a reliable broadcast protocol which guaran-
tees that all processes receive all messages in the same order. All messages sent by
the same process are received in the order they are sent. The order between messages
sent by different processes is determined by the order of their arrival at a token site,

\(^6\)\(g\) and \(g'\) are concurrent iff \(g \not\in \mathcal{P}(g')\) and \(g' \not\in \mathcal{P}(g)\).

\(^7\)This assumes that the underlying kernel knows how to compare these label.
i.e., site carrying a token. All sites are arranged in a logical ring structure and the
token is circulated along the ring in a specified order. This is done to improve avail-
ability of the token site in view of failures. The communication level concurrency for
this protocol is less than that for Psync since it attempts to order even concurrent
messages. The order of receipt of messages using this protocol is one of the possible
orders permitted by GS, but since GS does not order concurrent messages, we cannot
enforce a specific message receipt ordering using GS.

Raynal et al. [54] and Schiper et al. [60] describe schemes to implement causally
ordered message receipt, i.e., schemes in which receipt of two messages is ordered only
if their send events have \( \rightarrow \) relation between them. The possible message receipt
orderings under this scheme is the same as that in Psync.

Charron-Bost et al. [12] characterize distributed computations based on message
receipt ordering. They define a causally ordered class of computation as a computation
in which message receipt ordering does not violate the “happened before” relation, i.e.,
if the event of sending of two messages have the “happened before” relation between
them then their receipt at a common destination will also have the “happened before”
relation. This ordering is same as that in Psync.

2.4 Summary

In this chapter, we described the system model that is used in this dissertation. We
defined the notion of past and future of a message. The past of a message \( g \) is the set
of all messages whose send event “happened before” the event of sending \( g \). Similarly,
the future of a message \( g \) is the set of all messages such that the event of sending of \( g \)
“happened before” their send respective send events. We proposed the one-to-many communication abstraction, the global flush primitive $GS$. Messages sent and received using this abstraction satisfy the *global flush property* $G$ which is based on the past and future of a global flush message. Global flush messages are of four types: *global backward flush*, *global forward flush*, *global two-way flush*, and *global ordinary*, which satisfies Properties $Gb$, $Gf$, $Gt$, and $Go$, respectively. Note that the four Properties $Gb$, $Gf$, $Gt$, and $Go$, are collectively referred to as Property $G$.

We compared the $GS$ primitive with other one-to-many communication primitives proposed in the past and found that $GS$ primitives provide higher communication level concurrency. Later in this dissertation, we show that the global flush property is very useful in the design and development of distributed algorithms.
CHAPTER III

Implementing the Global Flush Primitive

In the previous chapter, we proposed the Global Flush primitive which satisfies the global flush property. We now discuss two implementations [26, 28] of the Global Flush primitive.

The first implementation [26] (described in Section 3.1) is a counter-based implementation which assumes that the underlying system is connected by non-FIFO point-to-point communication channels. In this implementation, each message sent by a process along a channel is uniquely identified among all messages sent along the same channel. For each pair of processes $p_j$ and $p_k$ in the system, a site $S_i$ maintains two counters: one to store the identity of the last message sent from $p_j$ to $p_k$ and the other to store the identity of the last $gb$ or $gt$ sent from $p_j$ to $p_k$. Each message sent by $p_i$ carries with it the value of all these counters at the time it is sent. Each site also maintains some additional data structures. These are used to determine the receivability of a message.

The second implementation [28] (described in Section 3.2) is an F-channel based implementation which assumes that the underlying system is connected by F-channels [3]. This implementation gives insight into the relation between point-to-point flush primitives and the global flush primitive. It uses a diffusion process [20] which results
in a flush message to be sent along each F-channel in the system. In this implementa-
tion, each message carries just one bounded integer and the information kept at
each site is bounded by the size of this integer. The overhead, both in terms of the
amount of information carried by a message and the amount of information kept at
each site, is independent of the number of processes in the system.

In Section 3.3, we compare the two implementations of the GS primitive with
each other and with implementations of other one-to-many primitives proposed in the
past [8, 9, 10, 17, 40, 53, 54]. The comparison is in terms of the amount of additional
information carried by a message, the size and type of information maintained by
each site, and the number of messages sent in the underlying system to enforce the
permitted message receipt ordering.

Two implementations of F-channels have been proposed in [3, 41]. The imple-
mentation of F-channels proposed in [3] is based on selective flooding. It floods the
network with copies of flush messages by requiring that copies of forward, backward,
and two-way flush messages sent from process \( p_i \) to process \( p_j \) be sent along all the
paths of physical channels from \( p_i \) to \( p_j \). This may not be acceptable in communica-
tion intensive computations. The second [41] implementation of F-channels is a
counter-based implementation— to be referred to as KCA implementation —which
penalizes computations that use fewer flush messages by increasing the size of data
structures that need to be compared with, to determine the receivability of a mes-
sage. Additionally, it increases the burden on the ordinary messages, to enforce the
flush property of a message that might be sent in the future, by requiring that each
ordinary message carries its identity, which is an unbounded integer. Thus, for this implementation, even though the communication structure of a computation that uses only ordinary messages is equivalent to that of a non-FIFO channel, it is not so in this implementation since the former still requires each message to carry an unbounded integer as its identity. The implementation also requires each site to keep track of the set of all messages received so far. The size of this set is also unbounded.

In Section 3.4, we present a fair and efficient counter-based implementation of the flush primitives which alleviates the problems in the existing implementations. In this implementation, computations that use fewer flush messages have smaller overhead. The size of the data structures maintained at each site is bounded, irrespective of the order of delivery of messages. We compare the proposed implementation with the KCA implementation and discuss the trade-offs.

### 3.1 A Counter-Based Implementation of GS Primitive

In this section, we describe a counter-based implementation of the GS primitive which assumes that the underlying system is connected by non-FIFO point-to-point communication channels.

The partial order $\rightarrow$, (see Definition 2.2.1), identifies the permitted message receipt orderings under the global flush property. If messages are received by each process $p_j$ in an order that does not violate $\rightarrow$ partial order, i.e. $g \rightarrow g' \implies \text{recv}(g)_j \rightarrow \text{recv}(g')_j$, then Property $G$ is satisfied. The set of messages that must be received by $p_j$ before $p_j$ can receive a message $g$ is referred to as $g$'s receive before set at $p_j$. 
Definition 3.1.1 [g's "receive before set" at pj]

For any g such that g.dst[j]=1, define g's receive before set at pj as the receivers projection on pj of the set \{g': g' \rightarrow g\}.

The receive before set at pj for a g which is a gb or go contains all gb and gt in \[\mathcal{P}(g)\]j. The receive before set at pj for a gf and gt is \[\mathcal{P}(gf)\]j and \[\mathcal{P}(gt)\]j, respectively.

Lemma 3.1.1 In a system where for each pj in g.dst, g carries information encoding g's receive before set at pj, and where g is received by pj only after all messages in g's receive before set at pj have been received, a message when received will satisfy property G.

Proof: Follows from the fact that if for all g, g is received by pj only after all messages in its receive before set at pj have been received by pj then \rightarrow is not violated. □

To ensure that all messages in the system when received satisfy Property G, with each g Sg.se sends information, denoted by g.t (t stands for type information), needed to enforce Property G. g.t contains g.typ. It also contains information encoding g's receive before set at pj.

3.1.1 Encoding the Receive Before Set of a Message

In this section, we describe how the receive before set for each global flush message is encoded. We first describe how message matrix clocks can be used to encode the receive before set for global flush messages of type gf and gt. We then identify the encoding scheme of the receive before set of global flush messages of type go and gb.
Encoding Receive Before Set for $gf$ and $gt$

The receive before set at $p_j$ for a global flush message $g$, which is a $gf$ or a $gt$, is 
$[\mathcal{P}(g)]_j$. To encode this, each $S_i$ maintains a message matrix clock, $mc_i$, as a vector of $N$ elements, where each element is a vector of $N$ integers. $mc_i[j, k]$ is $S_i$’s knowledge of the number of messages sent by $p_j$ to $p_k$. Consecutive messages sent by process $p_j$ to process $p_k$ are given consecutive identifiers from a sequence of natural numbers starting from one. $mc_i[j, k]$ also denotes $S_i$’s knowledge of the identity of the last message sent by $p_j$ to $p_k$. Each $g$ carries the value of sender’s message matrix clock when $g$ is sent as its timestamp, denoted by $g.mt$. $S_i$ maintains $mc_i$ as follows:

- Initially, $\forall j, k:: mc_i[j, k] := 0$;
- On executing a $gsend(g)$ event at $p_i$, $g$ is timestamped with the value of $mc_i$ and for each $j$, s.t. $g.dst[j] = 1$, $S_i$ increments $mc_i[i, j]$ by 1 (i.e., $\forall j; g.dst[j] = 1:: mc_i[i, j] := mc_i[i, j] + 1$) after sending $g$ to all such $S_j$ but before executing the next event;
- On executing a $grecv(g)$; event $mc_i$ is maintained as follows:
  
  $\forall j, k:: j \neq g.se \land j \neq i:: mc_i[j, k] := \max(mc_i[j, k], g.mt[j, k])$,
  
  $\forall k:: g.dst[k] = 1:: mc_i[g.se, k] := \max(mc_i[g.se, k], g.mt[g.se, k] + 1)$,
  
  $\forall k:: g.dst[k] = 0:: mc_i[g.se, k] := \max(mc_i[g.se, k], g.mt[g.se, k])$, i.e., $S_i$ updates its knowledge of the last message exchanged between processes. All elements in all rows, except those corresponding to $g.se$ and $p_i$, are set to the maximum of the current value and the value carried by the message. The $k^{th}$ element in the row corresponding to $g.se$ is set to maximum of the current value and one more
than the value carried by the message, if \( p_k \) is a destination of the message. Otherwise, it is set to the maximum of the current value and the value carried by the message. Note that the \( i^{th} \) row is not updated on a message receipt.

Since \( S_i \) increments \( mc_i[s] \) only on \( gsend(g) \) events, as opposed to incrementing it on both \( gsend(g) \) and \( grecv(g) \) events for vector clocks [22, 47], only the \( gsend(g) \) events, and hence the associated messages, can be uniquely identified by using the message matrix clock timestamp.

Note that \( [\mathcal{P}(g)]_j \), the receive before set for \( g \) at \( p_j \), is precisely represented by the \( j^{th} \) column of the message matrix clock timestamp \( g.mt \). Thus for a \( gf \) or \( gt \), \( g.mt \) contains sufficient information to encode the receive before set at each destination.

A \( g \) sent by \( p_i \) to \( p_j \) is uniquely identified by the triplet \((i, j, g.mt[i, j]+1)\). Note that, \( g.mt[i, j]+1 \) uniquely identifies \( g \) among all messages sent by \( p_i \) to \( p_j \), and could be used as \( g \)'s message identifier. In the sequel, we use \( g \) and \((i, j, g.mt[i, j]+1)\) interchangeably to refer to a message.

**Encoding Receive Before Set for \( go \) and \( gb \)**

The receive before set at \( p_j \) for a \( g \) which is a \( gb \) or \( go \) contains all \( gb \) and \( gt \) in \( [\mathcal{P}(g)]_j \). If all \( gbs \) and \( gos \) are received only after the last \( gb \) and \( gt \) in \( [\mathcal{P}(g)]_k \), \( \forall k \), then by transitivity all \( gbs \) and \( gos \) will be received at \( p_j \) after all messages in the receive before set at \( p_j \) have been received. Each \( S_i \) maintains an \( N \times N \) matrix \( lbk_i \) (stands for last \( gb \) or \( gt \)) such that at the instant of sending \( g \), \( lbk_i[j, k] \) represents the last \( gb \) or \( gt \) in \( [\mathcal{P}(g)]_k \), i.e., \( lbk_i[j, k] \) is the message identifier of the last \( gb \) or \( gt \), in the
past of \( g \), sent from \( p_j \) to \( p_k \). Initially, \( \forall j, k \ lbt[j, k] = 0 \) to indicate that no \( gb \) or \( gt \) message has been sent.

\( g \) carries the value of \( lbt_i \), denoted by \( g.lbt \), at the time of sending. The receive before set at \( p_j \) for a \( g \) that is a \( gb \) or \( go \) is conveyed by the \( j^{th} \) column of \( g.lbt \). Even though, for a \( gf \) and \( gt \), messages represented by \( g.lbt \) are also represented by \( g.mt \) (i.e., in its past), \( gf \) and \( gt \) messages need to carry \( g.lbt \) so that \( lbt_i \) can be correctly maintained. \( lbt_i \) is updated as follows:

update \( lbt_i := \)
- After sending \( g \) but before the next event →
  \[ \forall j: g.dst[j] = 1: g.typ = GB \lor g.typ = GT \rightarrow lbt_i[j, j] := g.mt[j, j] + 1 \]
- On receipt of \( g \) →
  \[ \forall j, k: lbt_i[j, k] := \max(lbt_i[j, k], g.lbt[j, k]) \]
  \[ (g.typ = GB) \lor (g.typ = GT) \rightarrow \forall j: g.dst[j] = 1: lbt_i[g.se[j], j] := g.mt[g.se[j], j] + 1 \]

Based on the above discussion \( g.t = (g.typ, g.mt, g.lbt) \). The table below (Table 1) summarizes the information from which the receive before set at \( p_j \) of a message can be identified.

Table 1: Information from which the receive before set at \( p_j \) of a message can be identified.

<table>
<thead>
<tr>
<th>For ( g ) of type</th>
<th>Receive Before set at ( p_j ), ( p_j \in g.dst ), can be inferred from</th>
<th>Receive before set of ( p_j ) will contain, ( \forall k, l: (k, j, l) ) s.t.</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO/GB</td>
<td>( j^\text{th} ) column of ( g.lbt )</td>
<td>( l = g.lbt[k, j] )</td>
</tr>
<tr>
<td>GF/GT</td>
<td>( j^\text{th} ) column of ( g.mt )</td>
<td>( 0 &lt; l \leq g.mt[k, j] )</td>
</tr>
</tbody>
</table>

3.1.2 Correctness Proof

We now give a proof of Lemma 3.1.1 when each \( g \) carries \( g.t = (g.typ, g.mt, g.lbt) \).
Lemma 3.1.2 In a system where each $g$ carries $g.t=(g.typ, g.mt, g.lbt)$ and where a message $g$ is received at $p_j$ (for each $p_j$ in $g.dst$) only after all messages in its receive before set at $p_j$ have been received, a message when received will satisfy Property $G$.

Proof: Property $Go$ imposes no constraints on receipt ordering of a message. $Gf$ and $Gt(b)$ are trivially satisfied, since the receive before set at $p_j$ for a $gf$ or $gt$ message is the same as the receivers projection of its past on $p_j$, represented by the $j$th column of $g.mt$.

To prove that $Gb$ and $Gt(a)$ are satisfied we need to show that for each message pair $g$ and $g'$ such that $g$ is a $gb$ or $gt$, $p_j$ is in $g.dst$, and $g' \in [F(g)]_j$, $g'$ is received by $p_j$ after $g$. Consider any such $g$ and $g'$. There are two possible cases:

Case 1: $g'$ is a $gf$ or a $gt$.

$Gf$ and $Gt(b)$ for $g'$ ensure that at each such $p_j$ $g$ is received before $g'$. Hence $Gb$ and $Gt(a)$ are satisfied.

Case 2: $g'$ is a $go$ or a $gb$.

There are two sub-cases:

Case 2A: $g.se=g'.se$.

Let $r$ be the number of $gb$ and $gt$ messages in $[[F(g)]_j]^{g.se} \cap [[P(g)]_j]^{g.se}$. $^1$ We prove by induction on $r$ that $Gb$ and $Gt(a)$ hold.

For base case since $r=0$, $g.lbt[g.se,j]=g.mt[g.se,j]+1$. Therefore $g$ will be received before $g'$. Hence $Gb$ and $Gt(a)$ hold.

$^1$ i.e., number of $gb$ and $gt$ sent by $g.se$ after sending $g$ but before sending $g'$ with $p_j$ as their destination.
Assuming that $Gb$ and $Gt(a)$ hold for $r \leq k$. The proof that $Gb$ and $Gt(a)$ hold for $r = k + 1$ follows from the following facts: (1) $\exists g': g''$ is a $gb$ or $gt$ such that $g'' \in [[F(g)]]_{j}^{\text{gs} \cdot \text{se}} \cap [[P(g)]]_{j}^{\text{gs} \cdot \text{se}}$, (2) $g$ is received before $g'' (r \leq k)$, and (3) $g''$ is received before $g' (r \leq k)$. Hence the proof.

Case 2B: \( g.se \neq g'.se \).

Let $g''$ be the last message in $[F(g)]^{\text{gs} \cdot \text{se}} \cap [P(g)]^{\text{gs} \cdot \text{se}}$.

Since $g'' \in P(g)$, a message chain from $g'$ to $g$ will always exist (follows from definition). Also, all possible message chains from $g''$ to $g$ will contain exactly one message (i.e., $g''$) sent by $g.se$; otherwise $g''$ will not be the last message in $[F(g)]^{\text{gs} \cdot \text{se}} \cap [P(g)]^{\text{gs} \cdot \text{se}}$. Thus,

$$g''.lb[g.se,j]=g'.lb[g.se,j].$$

Now consider the last $gb$ or $gt$ message, denoted by $g'''$, in $[F(g)]^{\text{gs} \cdot \text{se}} \cap [P(g)]^{\text{gs} \cdot \text{se}} \cup \{g\}$. If $g''' = g$, then from (1) $g'.lb[i,j]=g.mt[i,j]+1$, i.e., $g$ is received after $g$. Otherwise, i.e., if $g''' \neq g$, then from case 2A, $g$ is received at $p_j$ before $g''$, and $g''$ is received at $p_j$ before $g'$. Thus $g'$ is received at $p_j$ after $g$. \(\square\)

### 3.1.3 Protocol Details

We now describe the protocol executed at $S_i$ to implement Property $G$. Figure 4 describes the high level flowchart of the protocol discussed below.

Whenever $p_i$ is ready to execute $GS(g.typ, g.dst, g.body)$, $S_i$ computes $g.t$, executes $send(j, i, g.t, g.dst, g.b)$ \(^2\) for each $S_j$ such that $p_j \in g.dst$, updates $lbt_i$ if needed, and

\(^2\)This primitive when executed by $S_i$ delivers the message $g$ to $S_j$. 

then updates its message matrix clock $mc_i$ as described earlier.

0: $p_i$ is ready to execute $GS(g\.typ, g\.dst, g\.body) \rightarrow$
   $g\.t:=(g\.typ, mt, lbt_i)$
   $\forall j: g\.dst[j]=l: send(j, i, g\.t, g\.dst, g\.body)$
   update $lbt_i$
   update $mc_i$

Each $S_i$ maintains a set of messages, denoted by $\mathcal{D}$, which contains messages that have been delivered at $S_i$ but do not satisfy Property $G$. Whenever $g$ is delivered at $S_i$, it is added to $\mathcal{D}$.

1: $g$ is delivered at $S_i \rightarrow \mathcal{D} := \mathcal{D} \cup \{g\}$

Let $S_g$ denote the set of messages delivered at $S_i$ that satisfy Property $G$ but have not yet been received by $p_i$. Whenever $g$ in $\mathcal{D}$ satisfies Property $G$ it is moved to $S_g$.

2: $\exists g \in \mathcal{D} \land Satisfies\.G(g) \rightarrow \mathcal{D} := \mathcal{D} - \{g\}$
   $S_g := S_g \cup \{g\}$

We postpone explaining, until a little later, the implementation of boolean function $Satisfies\.G(g)$ which returns true if the receipt of $g$ will not violate Property $G$; otherwise the function returns false.
Whenever $p_i$ is ready to execute $\text{Rcv}(g.se, g.type, g.body)$ and $S_g \neq \emptyset$, $S_i$ removes a
$g$ from $S_g$ and returns $(g.se, g.type, g.body)$ to $p_i$, updates $lb_{ti}$, and updates its clock.

3: $p_i$ is ready to execute $\text{Rcv}(\text{var } g.se, g.type, g.body) \land (\exists g \in S_g) \rightarrow$

$S_g := S_g - \{g\}$
update $lb_{ti}$
update $mc_i$
return $(g.se, g.type, g.body)$ to $p_i$

Implementing $\text{Satisfies-G}$

$S_i$ keeps track, in a set $\mathcal{R}$, of all messages $(g.se,i, g.mi[g.se][i]+1)$ that have been
received by $p_i$. Initially, $\mathcal{R} = \{ (j,i,0) : 0 \leq j \leq N-1 \}$ to indicate that $p_i$ has received
no message from any $p_j$. At $S_i$, $\text{Satisfies-G}(g)$ is defined as

$$\text{Satisfies-G}(g) \equiv \begin{cases} \bigwedge_{j=0}^{N-1} (\langle j, i, g.lb[j,i]\rangle \in \mathcal{R}) & \text{if } g.type = \text{GO} \text{ or } g.type = \text{GB} \\ \bigwedge_{j=0}^{N-1} (\bigwedge_{i=0}^{g.mi[i][j]} \langle j, i, k \rangle \in \mathcal{R} \rangle) & \text{if } g.type = \text{GF} \text{ or } g.type = \text{GT} \end{cases}$$

In light of this, statement 3 can be replaced by:

3': $p_i$ is ready to execute $\text{Rcv}(\text{var } g.se, g.type, g.body) \land (\exists g \in S_g) \rightarrow$

$S_g := S_g - \{g\}$
$\mathcal{R} := \mathcal{R} \cup \{ (g.se,i, g.mi[g.se][i]+1) \}$
update $lb_{ti}$
update $mc_i$
return $(g.se, g.type, g.body)$ to $p_i$

Figure 5 shows the complete protocol executed by $S_i$.

3.1.4 Managing the Size of $\mathcal{R}$

In the protocol presented earlier the set $\mathcal{R}$ at each $S_i$, used in $\text{Satisfies-G}$ and state-
ment 3', is unbounded. The set operations used in this function and in statement 3' are those of set membership, i.e., $(j,i,k) \in \mathcal{R}$, and adding a member, i.e., $\mathcal{R} = \mathcal{R} \cup$
0: $S_i \leftarrow \{ p \text{ is ready to execute } GS(g.ty, g.dst, g.body) \rightarrow$

$g.t = (g.ty, mt, lbt_i)$

$\forall j: g.dst[j] = 1: send(j, i, g.t, g.dst, g.body)$

update $lbt_i$

update $mc_i$

1: $g$ is delivered at $S_i \rightarrow D := D \cup \{ g \}$

2: $\exists g \in D \land Satisfies.G(g) \rightarrow D := D - \{ g \}$

$S_g := S_g \cup \{ g \}$

3': $p_i$ is ready to execute $Rcv(var \ g.se, g.ty, g.body) \land (\exists g \in S_g) \rightarrow$

$S_g := S_g - \{ g \}$

$R := R \cup \{(g.se, i, g.mt[g.se][i] + 1)\}$

update $lbt_i$

update $mc_i$

return $(g.se, g.ty, g.body)$ to $p_i$

Initially

$\forall j, k: mc[j][k] := 0$, $\forall j, k: lbt[j][k] := 0$

$D := S_g := 0$, $R := \{(j, 0) : 0 \leq j \leq N - 1\}$

Figure 5: Protocol executed by $S_i$ to implement the $GS$ primitive.

(j, i, k), respectively. In this section we discuss representations for $R$ to reduce the memory requirements and an implementation for these operations.

$R$ can be represented by two vectors $lft_i$ and $Rsi_i$, described next, of size $N$.

$S_i$ keeps track of the last $gf$ or $gt$ message received by $p_i$ from each process in the group. $lft_i[j]$ denotes the last $gf$ or $gt$ message received by $p_i$ from $p_j$. Initially, $\forall j$ $lft_i[j] := 0$ to indicate that no $gf$ or $gt$ message has been received by $p_i$. Properties $Gf$ and $Gt$ ensure that when a $gf$ or a $gt$ message $g$ is received by $p_i$ all messages in $[P(g)]_i$ have been received by $p_i$, i.e.,

$\forall j : 0 \leq j \leq N - 1 : (\forall k : 0 \leq k \leq g.mt[j,i] : (j, i, k) \in R)$
Each $S_i$ also keeps track, in the $j^{th}$ element of the vector $Rs_i$, of all message received by $p_i$ from $p_j$, for each $p_j$, since the message $(j, i, lft_i[j])$ was received by $p_i$.

Each $Rs_i[j]$ is a set of integers, s.t. $k \in Rs_i[j]$ implies that the message $(j, i, k)$ has been received by $p_i$ and it was received after the $lft_i[j]$.

Adding a set member: The statement $\mathcal{R} = \mathcal{R} \cup (g.se, i, g.mt[g.se,i]+1)$ in statement 3' can be implemented as:

- $g.typ = GT$ $\rightarrow$ $lft_i[g.se] := g.mt[g.se][i] + 1$
- $Rs_i[g.se] := \emptyset$

- $g.typ = GF$ $\rightarrow$ $lft_i[g.se] := g.mt[g.se][i] + 1$

- $\forall k: k \in Rs_i[g.se] \land k < (g.mt[g.se][i]+1)$: $Rs_i[g.se] := Rs_i[g.se] \setminus \{k\}$

- $g.typ = GB$ or $g.typ = GO$ $\rightarrow$ $Rs_i[g.se] := Rs_i[g.se] \cup \{g.mt[g.se][i]+1\}$

Set membership: The condition $(j, i, k) \in \mathcal{R}$ can now be implemented as:

$(j, i, k) \in \mathcal{R} \equiv (k \leq lft_i[j]) \lor (k \in Rs_i[j])$

As is obvious from the above discussion, receipt of a $gf$ or a $gt$ message by $p_i$ reduces the size of $Rs_i$. If an implementation requires a specific bound on the size of $Rs_i[j]$ the underlying system could implement such a bound by periodically sending a $gf$ or $gt$ message just for this purpose. This will, of course, limit the communication level concurrency in the system.

### 3.2 An F-channel Based Implementation of GS Primitive

We now describe an implementation of GS primitive that is based on F-channels. Unlike the previous implementation which assumes that the underlying system is connected by non-FIFO channels, this implementation assumes that $S_i$'s are connected by a network of F-channels such that there is a path consisting of F-channels
from each site to every other site. Execution of the underlying system at \( S_i \) can now be viewed as a sequence of events of sending and receiving flush messages. \( fsend(m) \) is the event of sending a flush message \( m \) an F-channel. \( frecv(m) \) is the event of receiving the flush message \( m \) from an F-channel. In this discussion, we use \( f, b, t \) and \( o \) to denote messages of forward flush, backward flush, two-way flush, and ordinary message. Next, we briefly describe the flush property \([3]\).

**Flush Property**

Properties \( F, B, T, \) and \( O \), defined next, together constitute the flush property.

**Property \( F \) [Forward flush property]**

An \( f \) sent by \( p_i \) to \( p_j \) along channel \( c_{i,j} \) must be received by \( p_j \) only after \( p_j \) has received all messages sent along \( c_{i,j} \) by \( p_i \) before sending the \( f \).

An \( f \) makes assertions only about the receipt of messages sent before it along the same channel. A message sent after the \( f \) along the same channel may be received before \( f \) is received, if it does not violate the conditions imposed by any other message.

**Property \( B \) [Backward flush property]**

A \( b \) sent by \( p_i \) to \( p_j \) along channel \( c_{i,j} \) must be received by \( p_j \) before any other message sent along \( c_{i,j} \) by \( p_i \) after sending the \( b \) is received.

A \( b \) makes assertions only about the receipt of messages sent after it along the same channel. It does not impose any receipt ordering restriction on any messages sent before it.

**Property \( T \) [Two-way flush property]**

A \( t \) sent by \( p_i \) to \( p_j \) along channel \( c_{i,j} \) must be received by \( p_j \) only (a) after all messages
sent along \( c_{i,j} \) by \( p_i \) to \( p_j \) before sending the \( t \) have been received, and (b) before any message sent along \( c_{i,j} \) by \( p_i \) to \( p_j \) after sending the \( t \) is received.

A \( t \) satisfies both Properties \( F \) and \( B \). Receipt of a \( t \) is ordered with respect to all messages sent before and after it.

Property \( O \) [Ordinary flush property]

No ordering, with respect to any other flush message, is imposed on receipt of an \( o \).

3.2.1 Basic Idea

The underlying principle for this implementation is as follows: “if \( g \) and \( g' \) are two global flush messages such that \( g \) must be received at some \( p_j \) before \( g' \) is received then

1. a flush message \( m \) corresponding to \( g \) must be sent on all F-channel paths in the F-channel network from \( S_{g,se} \) to \( S_j \) before the flush message \( m' \) corresponding to \( g' \) is sent on these paths and
2. this order must be maintained along all paths until they are received.”

This identifies the following constraint on the F-channel network.

Property TC [Topological Constraint]

Consider the set of all global flush messages in the distributed system. If there exists two global flush messages \( g \) and \( g' \) in this set that have \( p_j \) as one of their destinations such that \( g \overset{\rightarrow}{\longrightarrow} g' \), then there must be a path of F-channels in the F-channel network topology from \( S_{g,se} \) to \( S_j \) through \( S_{g',se} \).

This topological constraint follows directly from the global flush properties and the underlying principle of the implementation. Global flush properties state that \( gf \) and \( gb \) are ordered with respect to messages in their past and future, respectively. A
*g* is ordered with respect to messages in both its past and future. The conditions in the underlying principle state that if message *g* must be received before/after *g' at *p*_j then they must be inserted along F-channel paths to *p*_j in that order. The topological constraint, thus, states that a path of F-channels must exist to facilitate insertion of each message before/after messages in its future/past. Since the set of global flush messages in a distributed system may not be known apriori, TC can be stated as: if there is a path of F-channels from *S*_i to *S*_j, then there must be a path of F-channels from *S*_j to *S*_i. In the rest of the discussion we will assume that there is a path of F-channels from each site to every other site.

We now describe a diffusion process [20] which causes a flush message, *m*, carrying information *Inf*, to be sent exactly once along each F-channel in the underlying network. By appropriately defining the flush message type and information *Inf*, we will ensure the two underlying principles described in the beginning of this section.

**Diffusion process initiation rule:** When invoked as diffuse(*mtyp*, *Inf*), the initiating site sends a flush message of type *mtyp* carrying information *Inf* along each outgoing F-channel.

**Diffusion process propagation rule:** On the receipt of a flush message carrying *Inf*, the site forwards a copy of the flush message, if it has not already done so, along each outgoing channel.

Since each site forwards a flush message only the first time it receives the flush
message, a site receives exactly one copy of each flush message on each incoming channel.

The set of events of receiving the first flush message, one at each site, due to a diffusion process identifies a cut \(^3\) to be referred to as the min-cut for the diffusion process. Similarly, the set of events of receiving the last flush message due to the diffusion process identifies a cut referred as the max-cut for the diffusion process. Given any two cuts \(C_1\) and \(C_2\), we say that \(C_1\) crosses \(C_2\) iff \(\exists e_i \in C_1\) such that \(e_i \not\in C_2\). Note that the min-cut of a diffusion process never crosses its max-cut.

The interval between the first and the last event at site \(S_j\) due to a diffusion process is termed as the diffusion region on \(S_j\). Figure 6(b) identifies the messages exchanged due to a diffusion process, its min-cut, its max-cut, and its diffusion region on sites when an underlying F-channel network topology of Figure 6(a) is assumed.

Whenever \(p_i\) executes \(GS(g_{\text{typ}}, g_{\text{dst}}, g_{\text{body}})\), \(S_i\) initiates a diffusion process with \(Infr = (g_{\text{typ}}, g_{\text{dst}}, g_{\text{body}})\). When \(g_{\text{typ}}\) is a GF, GB, GT, or GO, then \(m_{\text{typ}}\) is a \(f, b, t,\) or \(o\), respectively. If \(g\) is a \(gf\) or \(gt\), then for each \(p_j\) in \(g_{\text{dst}}, g\) becomes receivable at max-cut of the diffusion process. Otherwise, for each \(p_j\) in \(g_{\text{dst}}, g\) becomes receivable at the min-cut. Additionally, if \(g\) is a \(gt\), then message sends by \(p_j\)'s are inhibited in \(g\)'s diffusion region on \(S_{g,se}\). Table 2 summarizes the implementation rules for the global flush messages.

We now prove the correctness of the implementation.

---

\(^3\)A cut \([14]\) is a set of events, containing an event from each site, such that if event \(e_j\) on \(S_j\) is in the cut, then all events on \(S_j\) before \(e_j\) are also in the cut.
Figure 6: Figure (a) shows an F-channel network topology for a four process system. The solid directed lines denote F-channels. Figure (b) identifies the message exchange due to the diffusion process initiated by $S_i$, the min-cut, the max-cut, and the diffusion region on $S_i$.

Table 2: Implementation rules for Global Flush primitive.

<table>
<thead>
<tr>
<th>To implement $GS(g.typ, g.dst, g.body)$ where $g.typ=$</th>
<th>Initiate diffuse process with $Infr=$ (g.typ, g.dst, g.body) and $mtyp=$</th>
<th>For each $p_j$ in $g.dst$, $g$ becomes receivable at</th>
<th>Inhibit initiation of diffuse process in the diffusion region on $S_{g,se}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>GF</td>
<td>$f$</td>
<td>max-cut</td>
<td>No</td>
</tr>
<tr>
<td>GB</td>
<td>$b$</td>
<td>min-cut</td>
<td>No</td>
</tr>
<tr>
<td>GT</td>
<td>$t$</td>
<td>max-cut</td>
<td>Yes</td>
</tr>
<tr>
<td>GO</td>
<td>$o$</td>
<td>min-cut</td>
<td>No</td>
</tr>
</tbody>
</table>
3.2.2 Correctness Proof

To prove the correctness of the proposed implementation of the GS primitive, we need to prove its safety and liveness properties. To prove its safety, we need to show that when a message is received according to rules summarized in Table 2, it satisfies Property $G$. This is shown in Lemmas 3.2.1 thru 3.2.4. To prove the liveness of the implementation, we need to show that every message sent will eventually become receivable. We show this in Lemma 3.2.5. In proving safety and liveness of the proposed implementation, we assume that the underlying system of flush channels [3] works correctly. That is, receipt of a flush message does not violate the flush property (safety) and a flush message sent eventually becomes receivable (liveness).

**Lemma 3.2.1 [Safety of $gf$]** In a system which supports flush messages only, where each global message send and receive is implemented according to rules summarized in Table 2, a $gf$ message when received satisfies Property $Gf$.

**Proof:** A $gf$ is received at its max-cut. To prove that each $g, g \in \mathcal{P}(gf)$, is received before the $gf$ is received, we show that max-cut of such a $g$ cannot cross the min-cut of $gf$ and hence cannot cross the max-cut of $gf$, i.e., any such $g$ cannot be received after $gf$. We do this by contradiction. Assume that max-cut of some $g, g \in \mathcal{P}(gf)$, crosses the min-cut of $gf$, i.e., $g$ is delivered at some site $S_i$ from $S_j$ after the first copy of $gf$ is delivered at $S_i$ from $S_k$. There are two cases.

- $j = k$: Both $g$ and $gf$ were forwarded from the same site. $g$ from $S_j$ is delivered after $gf$ from $S_j$. There are only two possible scenarios under which this can happen.
- $S_j$ sent $g$ before sending $gf$ but still $gf$ was delivered before $g$. Since $gf$ is sent using $f$ flush message type, the property for forward flush is not satisfied. A contradiction.

- $S_j$ sent $gf$ before sending $g$ and hence $gf$ was delivered before $g$. Since each site forwards messages as soon as they delivered at the site for the first time, $g$ must have been delivered at $S_j$ after $gf$. By recursively applying this argument, we can show that either $gf$ must have been delivered at $S_{g,se}$ before $g,se$ sent $g$ (a contradiction since $g \in P(gf)$) or $S_{gf,se}$ forwarded $g$ after it sent $gf$ (again a contradiction, since $g \in P(gf)$).

- $j \neq k$: $g$ and $gf$ were forwarded by different sites. In this case, arguments similar to those of previous case can be constructed between the copy of $g$ forwarded by $S_k$ and the copy of $gf$ forwarded by $S_k$ to show a contradiction. □

**Lemma 3.2.2 (Safety of gb)** In a system which supports flush messages only, where each global message send and receive is implemented according to rules summarized in Table 2, a gb message when received satisfies Property Gb.

**Proof:** A gb is received at its min-cut. To prove that each $g, g \in F(gb)$, is received after the gb is received, we show that min-cut of such a $g$ cannot cross the max-cut of gb and hence cannot cross the min-cut of gb, i.e., any such $g$ cannot be received before gb. We do this by contradiction. Assume that min-cut of some $g, g \in F(gb)$, crosses the max-cut of gb, i.e., $g$ is delivered at some site $S_i$ from $S_j$ before the last copy of gb is delivered at $S_i$ from $S_k$. There are two cases.
• $j = k$: Both $g$ and $gb$ were forwarded from the same site. $g$ from $S_j$ is delivered before $gb$ from $S_j$. There are only two possible scenarios under which this can happen.

  - $S_j$ sent $g$ after sending $gb$ but still $gb$ was delivered after $g$. Since $gb$ is sent using $b$ flush message type, the property for backward flush is not satisfied. A contradiction.

  - $S_j$ sent $gb$ after sending $g$ and hence $gb$ was delivered before $g$. Since each site forwards messages as soon as they delivered at the site for the first time, $g$ must have been delivered at $S_j$ before $gb$. By recursively applying this argument, we can show that either $gb$ must have been delivered at $S_{g, se}$ after $g.se$ sent $g$ (a contradiction since $g \in F(gb)$) or $S_{gb, se}$ forwarded $g$ before it sent $gb$ (again a contradiction, since $g \in P(gb)$).

• $j \neq k$: $g$ and $gb$ were forwarded by different sites. In this case, arguments similar to those of previous case can be constructed between the copy of $g$ forwarded by $S_k$ and the copy of $gb$ forwarded by $S_k$ to show a contradiction. □

Lemma 3.2.3 [Safety of $gt$] In a system which supports flush messages only, where each global message send and receive is implemented according to rules summarized in Table 2, a $gt$ message when received satisfies Property $Gt$.

Proof: Any $g'$ whose diffusion process is initiated by $S_{g', se}$ in this region will be in $F(g)$. Max-cut of $g$ may cross min-cut of $g'$, thus potentially permitting the receipt
of $g'$ before $g$. This will violate Property $Gt(b)$. \footnote{This is not the case with $gb$ since a $gb$ is received at its min-cut. The min-cut of $gb$ can never cross the min-cut of any such $g'$ (due to the backward flush property).} Initiation of diffusion process in $g'$'s diffusion region on $S_{g,se}$, if $g$ is a $gt$, is thus not allowed. The rest of the proof for $gt$ is similar to the proof for $gf$ and $gb$.

\square

\textbf{Lemma 3.2.4} [Safety of $G$] \textit{In a system which supports flush messages only, where each global message send and receive is implemented according to rules summarized in Table 2, a message when received satisfies Property $G$.}

\textbf{Proof:} Property $Go$ is trivially satisfied. The rest follows from Lemmas 3.2.1, 3.2.2, and 3.2.3. Hence, the implementation satisfies safety property.

\square

We now prove the liveness of the implementation.

\textbf{Lemma 3.2.5} [Liveness] \textit{In a system which supports flush messages only, where each global flush message send and receive is implemented according to rules summarized in Table 2, a global flush message sent will eventually become receivable.}

\textbf{Proof:} Since a global flush message becomes receivable at either the min-cut (for $go$ and $gb$) or at the max-cut (for $gf$ and $gt$), to prove the lemma for each destination $pj$, we need to show that for each message sent min-cut and max-cut are attained at $pj$ in finite time. The implementation forwards a flush message as soon as it is received for the first time (See diffusion process propagation rule). The implementation thus does not inhibit receipt of any flush message, i.e., a flush message is received as soon as
it becomes receivable. This along with the liveness of the underlying system ensures that for each diffusion process both min-cut and max-cut are eventually attained.

Hence, the implementation satisfies liveness property. □

3.2.3 Protocol Details

Recall that each $gf$ and $gt$ become receivable at the max-cut of their corresponding diffusion processes. Since for a diffusion process of a message, each $S_j$ receives exactly one copy of the message on each incoming channel, $S_j$ can count the number of copies of the flush message received for each diffusion process to determine the $S_j$’s event at the max-cut. To implement the message counter, we assume that each global flush message has a process-wide unique \(^5\) identity denoted by $g.id$. All flush messages, sent by a diffusion process invoked to send $g$, carry $g.id$ as a part of $Infr$. Thus, $Infr = (g.id, g.se, g.typ, g.dst, g.body)$.

Each $S_j$ maintains a set, $D$, which contains information about the messages whose min-cut has been attained at $S_j$ (i.e., at least one copy of the flush message has been received by $S_j$), but the max-cut has not been attained at $S_j$ (i.e., a copy of the flush message has not been received on all incoming channels). Each element in $D$ is a 6-tuple $(id, se, typ, dst, body, cnt)$. The $cnt$ field is used to count the number of copies received of the message which is uniquely identified by $(id, se)$.

The set of messages in $D$ which become receivable may not be received in any arbitrary order because such a receipt might violate Property $G$. If messages in this

\(^5\)Later we weaken this restriction to put a bound on $g.id$. 
set are received by \( p_j \) in the order they become receivable at \( p_j \), then Property \( G \) will not be violated\(^6\). To implement this receivable order, each \( S_j \) also maintains a FIFO queue \( Q \), which contains information about messages that have not yet been received but can be received, in the order specified by \( Q \), without violating Property \( G \). Each element in \( Q \) is a triplet: \((g.se, g.typ, g.body)\). Initially \( Q \) is empty.

To implement the inhibition of message sends as identified in Table 2, each site maintains a boolean \( inhibit \) initialized to false.

Whenever \( p_i \) is ready to execute \( GS(g.typ, g.dst, g.body) \), \( S_i \) initiates the diffusion process. This gives the 0\(^{th} \) statement of the algorithm.

\[
0: \quad \text{\( p_i \) is ready to execute \( GS(g.typ, g.dst, g.body) \) and \( inhibit \rightarrow \)} \\
\quad \text{[} \\
\quad \quad \text{Infr= (id, i, g.typ, g.dst, g.body)} \\
\quad \quad g.typ=GO \rightarrow mtyp= o \\
\quad \quad g.typ=GF \rightarrow mtyp= f \\
\quad \quad g.typ=GB \rightarrow mtyp= b \\
\quad \quad g.typ=GT \rightarrow mtyp= t; \ Inhibit= True \\
\quad \quad \text{diffuse}(mtyp, Infr) \\
\quad \quad id=id+1 \\
\quad \text{]} \\
\]

Initially, \( id=0; \ Inhibit = False. \)

Whenever \( p_i \) is ready to execute \( Rcv(g.se, g.typ, g.body) \) and \( Q \) is not empty, \( S_i \) returns the element at the top of \( Q \).

\[
1: \quad \text{\( p_i \) is ready to execute \( Rcv(g.se, g.typ, g.body) \) and \( \neg empty(Q) \rightarrow \text{return}(\text{pop}(Q)) \) to \( p_i \)} \\
\]

Whenever a flush message is delivered at \( S_i \), the following occurs: If an entry for this diffusion process does not already exist in \( D \), an entry is created with the \( cnt \)

\(^6\)Such a total ordering is sufficient but not necessary. A weaker ordering can be imposed, such that Property \( G \) is not violated, by using elaborate data structures. Such enhancements, although interesting, are not relevant to the central idea presented in this thesis and hence are not pursued.
field initialized to one. \( S_i \) then forwards the message on all outgoing channels. If the flush message is an \( o \) or \( b \) and \( p_i \) is a destination, then \( S_i \) inserts \( (g.se, g.typ, g.body) \) in \( Q \), i.e., \( g \) becomes receivable. As shown in Table 2, \( gb \) and \( go \) become receivable at the min-cut. If an entry for this diffusion process already exists, the \( cnt \) field is incremented. When \( cnt \) becomes \( inc_i \), where \( inc_i \) is the number of incoming F-channels for \( S_i \), the entry is deleted. (Note that the max-cut for a diffusion is attained when \( cnt = inc_i \).) If the flush message is an \( f \) or \( t \) and \( p_i \) is a destination, then \( S_i \) inserts \( (g.se, g.typ, g.body) \) in \( Q \), i.e., \( g \) becomes receivable. As indicated in Table 2, \( gf \) and \( gt \) become receivable at the max-cut of the diffusion process. Additionally, if \( g.se=i \) and \( g.typ=GT \), the Inhibit flag is set to false (this event marks the end of the diffusion region on \( S_i \)).

2: \( S_i \) receives \( m \) carrying \( Infr \) →

\[
\begin{align*}
\text{[} & \neg \text{InD}(Infr) \rightarrow \\
& \text{[} \text{InsertD}(Infr) \\
& \text{send } m \text{ carrying } Infr \text{ on all outgoing channels} \\
& (Infr.typ=GO \lor Infr.typ=GB) \land Infr.dst[i]=1 \rightarrow \\
& \text{InsertQ}((Infr.se, Infr.typ, Infr.body)) \\
\text{]} \end{align*}
\]

\[\text{InD}(Infr) \rightarrow \]

\[\text{[Increment}(Infr) \]

\[\exists d \in D \land d.cnt=inc_i \rightarrow \]

\[\begin{align*}
\text{[} & D=D - d \\
& (Infr.typ=GF \lor Infr.typ=GT) \land Infr.dst[i]=1 \rightarrow \\
& \text{InsertQ}((d.se, d.typ, d.body)) \\
& Infr.typ=GT \land Infr.se=i \rightarrow \text{Inhibit}=false \\
\text{]} \end{align*}
\]
The function $\text{InD}(\text{Inf}_r)$ returns true if a corresponding entry for $\text{Inf}_r$ already exists in $\mathcal{D}$. Otherwise, it returns false. The function $\text{InsertQ}(x)$ inserts entry $x$ in $\mathcal{Q}$. The function $\text{InsertD}(\text{Inf}_r)$ inserts the 6-tuple $\text{Inf}_r$ in $\mathcal{D}$. The function $\text{Increment}(\text{Inf}_r)$ increments the cnt field of the entry for $\text{Inf}_r$ in $\mathcal{D}$. The function $\text{pop}$ removes and returns the entry at the top of the $\mathcal{Q}$.

Figure 7 shows the complete protocol executed by $S_i$.

### 3.2.4 Bounding the Message Id

Note that in the above implementation, each global flush message is assumed to have a process wide unique message identifier. We now discuss ways to bound these identifiers.

**Lemma 3.2.6** Global flush messages $g$ and $g'$, sent by the same process, may have the same id iff their diffusion regions do not intersect at any process.

At each $S_j$, $g.id$ is used to count the number of incoming channels on which the message has been received by $S_j$. Once a message has been received on all incoming channels, the message identifier is no longer needed to identify the message. □

**Lemma 3.2.7** Given three global flush messages $g$, $gt$, $g'$ sent by $p_i$ such that $g \in \mathcal{P}(gt)$ and $g' \in \mathcal{F}(gt)$. The diffusion regions of $g$ and $g'$ do not intersect at any site.

**Proof:** We assume to the contrary and then show that Property $G$ is violated. Assume that three global flush messages $g$, $gt$, and $g'$, sent by $p_i$, exist such that $g \in \mathcal{P}(gt)$, $g' \in \mathcal{F}(gt)$, and the diffusion regions of $g$ and $g'$ intersect at $S_j$. 
0: $S_i$: $p_i$ is ready to execute $GS(g.typ, g.dst, g.body) \land \neg \text{Inhibit} \rightarrow$

$[\begin{array}{l}
\text{Infr} \leftarrow (id, 1, g.typ, g.dst, g.body) \\
g.typ=\text{GO} \rightarrow \text{mtyp}=o \\
g.typ=\text{GF} \rightarrow \text{mtyp}=f \\
g.typ=\text{GB} \rightarrow \text{mtyp}=b \\
g.typ=\text{GT} \rightarrow \text{mtyp}=t; \text{Inhibit}=\text{True} \\
\text{diffuse(mtyp, Infr)} \\
id=id+1
\end{array}]$

1: $p_i$ is ready to execute $\text{Rcv}(g.se, g.typ, g.body) \land \neg \text{empty(Q)} \rightarrow \text{return(pop(Q)) to } p_i$

2: $S_i$ receives $m$ carrying $\text{Infr} \rightarrow$

$[\begin{array}{l}
\text{InD(Infr)} \rightarrow \\
\text{[InsertD(Infr)]} \\
\text{send } m \text{ carrying } \text{Infr} \text{ on all outgoing channels} \\
(\text{Infr.typ}=\text{GO} \lor \text{Infr.typ}=\text{GB}) \land \text{Infr.dst[i]=1} \rightarrow \\
\text{InsertQ((Infr.se, Infr.typ, Infr.body))}
\end{array}]$

Initially, $id=0$, $\text{empty(Q)}=\text{True}$, $D=\{\}$, $\text{Inhibit}=\text{False}$

Figure 7: Protocol executed at $S_i$. 
For the diffusion regions of $g$ and $g'$ to intersect at $S_j$ either the max-cut of $g$ must cross the max-cut of $gt$ or the max-cut of $gt$ must cross the min-cut of $g'$. If the max-cut of $g$ crosses the max-cut of $gt$, then assume that $g$ is a $gf$. Since $gf$ is received at the max-cut, its receipt violates $Gt$. If max-cut of $gt$ crosses the min-cut of $g'$, then assume that $g'$ is a $go$. Since $go$ is received at the min-cut, its receipt violates $Gt$.

In both scenarios, for the diffusion regions of $g$ and $g'$ to intersect, we reach a contradiction.

Clearly, it follows from Lemmas 3.2.6 and 3.2.7 that the identifiers of messages, sent by $gt.se$, in $\mathcal{P}(gt)$ can be reused for messages sent by $gt.se$ in $\mathcal{F}(gt)$. Thus, to bound the range of values used by $S_i$ for message identifiers, $S_i$ should periodically initiate the diffusion process for $gt$ s.t. $\forall j \ (gt.dst[j]=0$. Since there is no process in this $gt.dst$, this $gt$ will not be received by any process and yet it can be used as a signal from $S_i$ that $S_i$ is resetting its message identifier $id$.

### 3.3 Comparison

In this section, we compare the two implementations of $GS$ with each other and with implementations of other broadcast primitives proposed [8, 9, 10, 17, 40, 53, 54]. The comparison is in terms of the amount of additional information carried by a message, the size and type of information maintained by each site, and the number of messages sent by the underlying system.

The counter-based implementation of the $GS$ primitive requires each site $S_i$ to assign a unique identifier to messages sent, from a sequence of integers starting at
one. For each pair of sites $S_j$ and $S_k$ in the system, $S_i$ maintains two counters: one ($mc_i[j, k]$) to store the identity of the last message sent from $S_j$ to $S_k$ and the other ($lbt_i[j, k]$) to store the identity of the last $gb$ or $gt$ sent from $S_j$ to $S_k$. Each message sent by $S_i$ carries the counter arrays $mc_i$ and $lbt_i$. These counters are updated on the receipt of a message. In addition to maintaining these $2N^2$ unbounded integer counters, each site also maintains three sets: the set of all messages that have been delivered at the site but are not yet receivable $D$, the set of messages that are receivable but have not yet been received $S_g$, and an unbounded set of messages that have been received so far by the process $R$.

In contrast, in the F-channel based implementation each site maintains only $D$ and $Q$, in addition to a bounded integer counter. The logic in determining the receivability of a message is also greatly simplified. Each global flush message is implemented by sending a flush message along each channel in the system, thus resulting in $|E|$ flush messages, where $|E|$ is the number of flush channels in the underlying system. In contrast, for each destination the number of messages required in the counter-based implementation is of the order of the number of hops in the shortest path between the source and the destination. For systems in which processes are sparsely connected by F-channels (as is the case in applications using a typical client-server model), increase in the number of messages is a reasonable price paid for the elegance and simplicity of the proposed implementation.

In the F-channel based implementation, message receipt ordering is enforced based on the message type and message identifier. The message body need not be available at the destination site until a message is ready to be received. Thus, if it can be ensured that a message body eventually arrives at the destination site, and hence is
available for a process to receive, each flush message in the underlying system need not carry the body. Each flush message now carries only a bounded integer. The body of the message can be sent separately. When compared to the counter-based implementation, the resulting overhead is now that of sending a bounded integer along each of |E| channels.

The bound on information carried by each message is not without cost. The number of message receipt orderings permitted by the proposed implementation is somewhat less than that permitted by the implementation proposed in [26]. It will become clear from the following example. Consider a system of three processes, \( p_i, p_j, \) and \( p_k \). Process \( p_i \) sends \( g \) which is a GO to \( p_j \) only and then sends a \( gt \) to \( p_k \) only. \( p_i \) then sends \( g' \) which is also a GO to \( p_j \) only. By definition, \( g \) and \( g' \) can be received in any order. Under the proposed implementation \( g' \) cannot be received before \( g \) at \( p_j \), because the min-cut of \( g \) cannot cross the max-cut of \( gt \) and the max-cut of \( gt \) cannot cross min-cut of \( g' \). Thus, the number of permissible message receipt orderings is limited to the order identified by the relation \( m^o \) (defined below). It must be noted that this is a minor limitation given other redeeming features such as reduced overhead (each message carries 1 bounded integer as opposed to \( 2N^2 \) unbounded integers) in the proposed implementation.

Definition 3.3.1 ["m^o", to be read as “actual message receipt ordering”]

Define \( g \rightarrow^o g' \) iff \( gsend(g) \stackrel{t}{\rightarrow} gsend(g') \land ((g.\text{typ}=\text{GB} \lor g.\text{typ}=\text{GT}) \lor (g'.\text{typ}=\text{GF} \lor g'.\text{typ}=\text{GT})) \).

Note that \( \rightarrow \) \( \subseteq \) \( m^o \).

The implementation of \( \text{Ps}ync \) proposed in [53] requires that each message be uniquely identified among all messages sent by a process. Each message carries \( N \)
unbounded integers to encode the context relationship.

In CBCAST [8, 10, 40] a label "clabel" is associated with each message. The ordering among message receipts is specified in terms of a "less than" relation, denoted by $\mapsto$, based on these labels. The kernel at each site knows this "less than" relation. Messages are received in an order that does not violate the $\mapsto \cap \xrightarrow{\text{lam}}$ relation. The size of labels needed to enforce the message receipt ordering permitted by GS is $|\mathcal{P}(g)|$ bits and is unbounded.

The new implementation of CBCAST [9] is similar to Psync except for process groups. The relation $\mapsto$ is encoded using vector clocks [22, 47]. Each message carries the value of the vector clock at the time of sending as its timestamp. This timestamp consists of $N$ unbounded integers. In both these implementations, the size of the set $\mathcal{R}$ is unbounded. Raynal et al. [54] describe an implementation of causally ordered message receipt. The possible message receipt orderings under this scheme is the same as that in Psync. The past of a message is represented by $N^2$ unbounded integers. Each message carries these integers with it. Additionally, the size of data structures maintained by each site is potentially unbounded.

3.4 A Fair and Efficient Implementation of F-channels

3.4.1 Motivation

Two implementations [3, 41] of F-channels have been proposed in the literature.

The first [3] is based on selective flooding. It requires that a flush message of type $f$, $b$, or $t$ sent from process $p_i$ to process $p_j$ be sent along all the paths of physical channels from $p_i$ to $p_j$. The implementation assumes that underlying physical channels are FIFO. The non-FIFO nature of the communication between processes is
due to multiple physical paths between them. Thus, this implementation floods the network with copies of flush messages. This may not be acceptable in communication intensive computations.

The second [41] is a counter-based implementation in which each message carries its identity, which is unique among all messages sent along the channel. It contains information about the set of messages that must be received before the message is received. This set of messages is referred to as flush receive before set. The flush receive before set of an $f$ and a $t$ sent along a channel is the set of all messages sent along the channel before them. The flush receive before set for a $b$ and an $o$ is the set of all backward flush and two way flush messages sent along the channel before them. The receiver maintains a set, called received set, which contains the identity of all messages that have been received so far. A message is received only when all messages in its flush receive before set are in the received set. This implementation penalizes computations that use fewer $f$, $b$, and $t$, messages by increasing the size of received set used to check receivability of a message. Additionally, it increases the burden on the ordinary messages, to enforce the flush property of a message that might be sent in the future, by requiring that an ordinary message carry its identity, which is an unbounded integer. Thus for this implementation, even though the communication structure of a computation that uses only ordinary messages is equivalent to that of a non-FIFO channel, it is not so in the implementation since the former still requires each message to carry an unbounded integer as its identity. Additionally, if these messages are received in a non FIFO order, the received set also grows unnecessarily.

We present a counter-based implementation for the flush primitives which allevi-
ates the problems in the existing implementations. In this implementation, computations that use fewer $f$, $b$, and $t$ messages have smaller overhead both in terms of the size of received set and the amount of information carried by each message, e.g., in a computation that uses only ordinary messages, each message carries a constant number and the size of the received set is also bounded irrespective of the order of delivery of messages.

To implement the flush primitives on channel $c_{i,j}$, the sender maintains certain data structures. The contents of some of these data structures at the time of sending of each message is appended to the message sent along $c_{i,j}$. It contains the information required to deduce the flush receive before set for the message, and is essential for the receiver $p_j$ to enforce the flush property. Messages may be delivered by $c_{i,j}$ at $p_j$ in any order. $p_j$ maintains information about the messages that have been delivered at $p_j$ and about the messages that have been received by $p_j$. We use $DLVD_j$ and $RCVD_j$ to denote the set of messages that have been delivered at $p_j$ but not yet received by $p_j$ and the set of messages that have been received by $p_j$ so far, respectively. This information, in conjunction with the information carried by a message, is used at $p_j$ to determine the receivability of a delivered message. Only a delivered message which is receivable can be received by $p_j$. In specifying the implementations below, we identify the following:

- **Senders data structure(s):** What data structure are maintained by the sender? How and when are these data structures updated and used?

- **Information carried by a message:** The amount and type of information carried by a message. At what rate does this information grow?
• Receivers data structure(s): What data structures are maintained by the receiver? How and when are these data structures updated and used?

• Receivers logic: Amount and type of computation performed by the receiver to detect the receivability of a message.

Apart from completely specifying the implementation, these also provide a criteria to compare different implementations.

We first describe the KCA implementation [41] of the flush messages. we then describe the proposed implementation of F-channels in detail. We also compare the proposed implementation with the KCA implementation and discuss the trade-offs.

3.4.2 KCA Implementation

In this section, we briefly describe the counter-based KCA implementation [41]. Specifically, we consider the channel $c_{i,j}$ from process $p_i$ to $p_j$.

In this implementation, the sender $p_i$ maintains a counter, $seqn_o_i$, which is incremented by one for every message sent along $c_{i,j}$. Each message sent along $c_{i,j}$ carries the value of $seqn_o_i$, at the time it is sent, as its identifier. We denote the identifier for message $g$ by $g.id$. $p_i$ also keeps track of the last backward flush or two way flush sent along $c_{i,j}$. This is used to denote the flush receive before set for $o$ and $b$ messages. Thus, each $o$ and $b$ message carries two numbers, its identity and the identity of the last backward flush or two way flush sent before it. We denote the latter by $g.bt$.

When a message is delivered at $p_j$, it is added to $DLV D_j$, the set of messages that have been delivered at $p_j$ but not yet received by $p_j$.

A message $g$ in $DLV D_j$, which is an $o$ or a $b$, becomes receivable when $g.bt \in$
A message \( g \) in \( DLVD_j \), which is an \( f \) or a \( t \), becomes receivable when 
\[
\forall i : 1 \leq i < g.id :: i \in RCVD_j, \quad \text{i.e., only after all messages, sent along } c_{i,j}, \text{ whose identifier is less than } g.id \text{ have been received. When a message is received, its identifier is added to } RCVD_j.
\]

It is interesting to note that in this implementation, \( \text{segno} \) increases by one for every message sent by \( p_i \) to \( p_j \). Also, the set \( RCVD_j \) is an unbounded set of non-negative integers. \( RCVD_j \) can be represented as an integer \( I_{rcvd_j} \) and a set \( RCVD_{small_j} \), such that 
\[
\forall i : 1 \leq i \leq I_{rcvd_j} \wedge i \in RCVD_{small_j} :: i \in RCVD_j \quad \text{and} \quad \forall i : i \in RCVD_j :: i < I_{rcvd_j} \vee i \in RCVD_{small_j}.
\]

Here \( I_{rcvd_j} \) represents the largest sequence of integers in \( RCVD_j \) and the sent of the integers in \( RCVD_j \) are now represented by \( RCVD_{small_j} \).

**Proposed F-channel Implementation**

We now discuss the proposed implementation of flush primitives. In most applications, not all flush primitives are used. Thus, we first propose independent implementation of each of these in Subsections 3.4.3, 3.4.4, and 3.4.5, respectively. We then give implementations of pairs of these flush primitives (along with \( o \)) in Subsections 3.4.6, 3.4.7, and 3.4.8. Finally in Subsection 3.4.9, we give an implementation supporting all four flush primitives.

### 3.4.3 \( f \) and \( o \)

The sender \( p_i \) counts the number of \( f \) messages sent so far and the number of \( o \) messages sent since the last \( f \) message. Initially, both the counters, denoted by \( fcnt_i \)

\( ^7 \)We assume, always \( 0 \in RCVD_j \).

\( ^8 \)Note that \( \lor \) is an exclusive OR.
and $ocnt_i$, respectively, are set to 0 to indicate that no message has been sent. The sender increments $ocnt_i$ after sending an $o$. The $fcnt_i$ is incremented by one and $ocnt_i$ is reset to 0 after an $f$ is sent.

Each $f$ carries the value of $fcnt_i$ and $ocnt_i$ at the time it is sent. We denote them by $f.fcnt$ and $f.ocnt$, respectively. An $o$ carries with it the value (denoted by $o.fcnt$) of $fcnt_i$ at the time it is sent.

The receiver $p_j$ maintains a count of the number of $f$ messages received so far (denoted by $fcnt_j$). It also maintains a set $RCVD_j$ of tuples. Each tuple $(x,y)$ in $RCVD_j$ denotes that $y$ ordinary messages with $o.ocnt=x$ have been received so far.

An $o$ can be received as soon as it is delivered. On receipt of an $o$, the entry $(o.fcnt, y)$ in $RCVD_j$ is replaced by $(o.fcnt, y+1)$, if such an entry exists. Otherwise, an entry $(o.fcnt, 1)$ is inserted into $RCVD_j$.

An $f$ can be received only when $f.fcnt=fcnt_j$ and when an entry $(f.fcnt,f.ocnt)$ exists \(^9\) in $RCVD_j$. After receipt of an $f$, $fcnt_j$ is incremented by one and the entry $(f.fcnt,f.ocnt)$ deleted from $RCVD_j$.

**Lemma 3.4.1** Messages received according to the above implementation satisfy Properties F and O.

**Proof:** In a channel which supports $f$ and $o$ message types only, the flush receive before set for an $o$ is empty. Thus, we need to show that the receipt of an $f$ does not violate Property F. This can be proved by induction on the number of $f$ received by $p_j$. Let $x$ denote the number of $f$ received so far.

**Base Case ($x=0$):** According to the implementation, the first $f$ can only be received when an entry $(0,f.fcnt)$ exists in $RCVD_j$. Such an entry will exist only when

\(^9\)We assume that $(x,y)\in RCVD_j$ is true if $y = 0$. 
the \( f.\text{ocnt} \) ordinary messages with \( o.\text{fcnt} = 0 \) have been received, i.e., the first \( f.\text{ocnt} \) ordinary messages have been received. Hence, the receipt of first \( f \) will not violate Property \( F \).

**Induction Step:** Assume that the receipt of first \( k \) forward flush messages does not violate Property \( F \). We now show that the receipt of \( k + 1^{\text{th}} \) forward flush message does not violate Property \( F \). Using an argument similar to the base case, we can show that all messages since the \( k^{\text{th}} f \) are received before the \( k + 1^{\text{th}} f \) is received. The implementation ensures that the \( k^{\text{th}} f \) will be received before the \( k + 1^{\text{th}} f \) is received. The receipt of the \( k^{\text{th}} f \) does not violate Property \( F \). Thus, the receipt of the \( k + 1^{\text{th}} f \) does not violate Property \( F \).

It is interesting to note that in this implementation, the fewer the number of \( f \) messages used by a computation, the smaller the amount of information carried by a message. Also the fewer the number of \( f \) messages used, the lesser the number of entries in \( RCVD_j \). Thus the size of information carried by a message and the size of data structures maintained by a receiver varies directly with the number of \( f \) messages used in the computation.

In the KCA implementation, an ordinary message carries two counters: its identity and the identity of the last \( f \) message sent before it. An \( f \) message carries its identity. Thus the size of information carried by an ordinary message depends on the total number of messages sent before it in the computation, instead of on the number of \( f \) messages sent before it. Also, the size of \( RCVD_j \) can be as high as the number of ordinary message sent in the computation.
3.4.4 $b$ and $o$

The sender $p_i$ counts the number of $b$ messages sent so far in a counter denoted by $bcnt_i$. Initially $bcnt_i$ is 0. $bcnt_i$ is incremented after a $b$ is sent.

Each $b$ and $o$ carries the value of $bcnt_i$ at the time it is sent, which are denoted by $b.bcnt$ and $o.bcnt$, respectively.

The receiver $p_j$ maintains a count of the number of $b$ messages received so far (denoted by $bcnt_j$).

A $b$ is received only when $b.bcnt = bcnt_j$ and an $o$ is received only when $o.bcnt = bcnt_j$.

Lemma 3.4.2 *Messages received according to the above implementation satisfy Properties B and O.*

**Proof:** The *flush receive before* set of a $b$ includes all backward flush messages that have been sent before it. It can be shown by induction on the number of $b$ messages received so far that all $b$ messages are received in the order they are sent. The *flush receive before* set of an $o$ includes all $b$ messages sent before it. Since an $o$ carries the count of number of $b$ messages sent before it, the implementation ensures that it is received only after $o.bcnt$ backward flush messages, i.e., the first $o.bcnt$ backward flush messages have been received. Thus, the receipt of an $o$ does not violate Properties $B$ and $O$. Thus, also the receipt of an $b$ does not violate the Property $B$. □

It must be noted that optimizing an implementation of $b$ and $o$ only, based on KCA implementation, yields an implementation similar to the above implementation.
3.4.5 \textit{t} and \textit{o}

The sender \( p_i \) counts the number of two way flush messages sent so far and the number of ordinary messages sent since the last \( t \). Initially, both the counters, denoted by \( tcnt_i \) and \( ocnt_i \), respectively, are 0. The sender increments \( ocnt_i \) after sending an \( o \). The \( tcnt_i \) is incremented by one and \( ocnt_i \) is reset to 0 after a \( t \) is sent but before any other action is taken.

Each \( t \) carries the value of \( tcnt_i \) and \( ocnt_i \) at the time it is sent which are denoted by \( t.tcnt \) and \( t.ocnt \), respectively. An \( o \) carries with it the value of \( tcnt_i \) at the time it is sent (denoted by \( o.tcnt \)).

The receiver \( p_j \) maintains a count of the number of two way flush messages received so far (denoted by \( tcnt_j \)) and the number of ordinary messages received since the last \( t \) (denoted by \( ocnt_j \)).

An \( o \) can be received only when \( o.tcnt = tcnt_j \). On receipt of an \( o \), the \( ocnt_j \) is incremented by one. A \( t \) can be received only when \( t.tcnt = tcnt_j \) and \( t.ocnt = ocnt_j \). After receipt of a \( t \), \( tcnt_j \) is incremented by one and \( ocnt_j \) is reset to 0.

\textbf{Lemma 3.4.3} Messages received according to the above implementation satisfy Properties \( T \) and \( O \).

\textbf{Proof:} Similar to Lemmas 3.4.1 and 3.4.2. \hfill \Box

In this implementation, the size of information carried by a message varies directly with the number of \( t \) messages used in the computation. Also, the receiver maintains only two counters. This simplifies the computation logic at the receiver since comparison of only two counters needs to be done to determine receivability of a message.
as opposed to checking set membership in the KCA implementation. The amount of information that needs to be maintained by the receiver is also greatly reduced in the proposed implementation.

In the KCA implementation, an ordinary message carries two counters: its identity and the identity of the last \( t \) sent. An \( t \) message carries its identity. Thus, the amount and size of information carried by an \( o \) does not depend on the number of \( t \) messages used; instead it depends on the total number of messages used in the computation. The size of \( RCVD_j \) is bounded only by the maximum number of ordinary messages between two successive two way flush messages.

Table 3: Summary of the data structures maintained by a sender, information carried by a message, and the data structures maintained by a receiver when only two flush message types are implemented.

<table>
<thead>
<tr>
<th>Msg Type Implemented</th>
<th>Counters at Sender</th>
<th>Information carried by message of type</th>
<th>Receivers Data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f, o )</td>
<td>( fcnt, ocnt )</td>
<td>( fcnt, ocnt )</td>
<td>( fcnt, RCVD_j )</td>
</tr>
<tr>
<td>( b, o )</td>
<td>( bcnt )</td>
<td>( bcnt )</td>
<td>( bcnt )</td>
</tr>
<tr>
<td>( t, o )</td>
<td>( tcnt, ocnt )</td>
<td>( tcnt, ocnt )</td>
<td>( tcnt, ocnt )</td>
</tr>
</tbody>
</table>

Table 3 summarizes the counters maintained by a sender, the information carried by a message, and the data structures maintained by a receiver in the three implementations discussed so far.

### 3.4.6 \( f, b, \) and \( o \)

We now describe an implementation of a channel which supports messages of type \( f, b, \) and \( o \) only. This implementation is a combination, as described next, of the implementations of \( f \) and \( o \), and \( b \) and \( o \), discussed in Subsections 3.4.3 and 3.4.4,
respectively.

The sender \( p_i \) maintains three counters, \( fcnt_i \), \( ocnt_i \), and \( bcnt_i \), to count the number of \( f \) flush messages sent so far, the number of ordinary messages sent since the last \( f \), and the number of \( b \) messages sent so far, respectively.

An \( f \) carries the value of \( fcnt_i \), \( bcnt_i \), and \( ocnt_i \) at the time it is sent. An \( o \) carries the value of \( fcnt_i \) and \( bcnt_i \) at the time it is sent. A \( b \) carries the value of \( bcnt_i \) at the time it is sent.

Along with the set \( RCVD_j \) the receiver \( p_j \) maintains two counters, \( fcnt_j \) and \( bcnt_j \), to count the number of \( f \) and \( b \) messages received so far.

Since the flush receive before set of \( f \) and \( o \) now include all the backward flush messages sent before them, the receivability condition for them is now different. An \( f \) can now be received only when \( f.fcnt = fcnt_j \), \( f.bcnt = bcnt_j \), and an entry \( (f.fcnt, f.ocnt) \) exists in \( RCVD_j \). An \( o \) can be received only when \( o.bcnt = bcnt_j \). An \( b \) can be received only when \( b.bcnt = bcnt_j \).

**Lemma 3.4.4** Messages received according to the above implementation satisfy Properties \( F \), \( B \), and \( O \).

**Proof:** A proof by induction, similar to that of Lemmas 3.4.1 and 3.4.2, can be easily constructed. We give a brief sketch of the proof. Use induction to show that all \( f \) and \( b \) messages are received in the order they are sent. Since both \( o \) and \( f \) carry the count of the number of backward flush messages sent before it, the implementation ensures that they are received only after all the backward flush messages sent before them have been received. Thus, the receipt of an \( o \) does not violate Properties \( F \), \( B \), and \( O \). A proof similar to that of Lemma 3.4.1 can be constructed to show that an \( f \) is received only after all messages sent before it have been received, hence received
after all backward flush messages sent before it have been received. Thus, the receipt of a $b$ does not violate Property $B$. □

**Lemma 3.4.5** The counters $fcnt$ and $bcnt$ cannot be combined.

**Proof:** Follows from the fact that the $\text{flush receive before}$ set for a $b$ is the set of all backward flush messages sent before it. This is different from the $\text{flush receive before}$ set of an $f$ which is the set of all $f$ messages, $b$ messages, and ordinary messages sent along the channels. □

In this implementation, the amount of information carried by a $b$ and an $o$ depends on the number of $b$ and $f$ messages sent so far. The fewer the number of flush messages sent the smaller the information. Also, the size of the set $RCVD_j$, maintained by the receiver, is also a function of the number of $f$ and backward flush messages sent. This contrasts with the KCA implementation, where both the $b$ and $o$ messages carry two counters, the size of which depends on the total number of messages exchanged so far. In applications where fewer number of flush messages are sent, the proposed implementation results in reduced overhead. Also, since the set $RCVD_j$ for KCA implementation is potentially larger that that for the proposed implementation, the former results in higher overhead in determining set membership. It must be noted that in the proposed implementation, an $f$ carries three counters as opposed to only two in KCA implementation.

\hspace{1em}^\text{Note that this lemma claims only that for the implementation approach taken in this chapter the number of counters is minimal.}
3.4.7 $f$, $t$, and $o$

In this section, we describe an implementation of a channel that supports messages of type $f$, $t$, and $o$ only. This implementation is based on the implementations described in Subsections 3.4.3 and 3.4.5.

A sender $p_i$ maintains three counters $tcnt_i$, $fcnt_i$, and $ocnt_i$ to count the number of two way flush messages sent so far, the number of $f$ messages sent so far, and the number of ordinary messages sent since the last $f$ message, respectively.

Messages of type $f$ and $t$ carry the value of $tcnt_i$, $fcnt_i$, and $ocnt_i$ at the time they are sent. An $o$ carries the value of $tcnt_i$ and $fcnt_i$ at the time it is sent.

Along with the set $RCVD_j$ a receiver $p_j$ maintains two counters $tcnt_j$ and $fcnt_j$, to count the number of $t$ and $f$ messages received so far, respectively.

An $f$ becomes receivable when $f.tcnt=tcnt_j$, $f.fcnt=fcnt_j$, and an entry $(f.fcnt, f.ocnt)$ exists in $RCVD_j$. After the receipt of the $f$, $fcnt_j$ is incremented by one and the entry $(f.fcnt, f.ocnt)$ deleted from $RCVD_j$.

An $o$ can be received only when $o.tcnt=tcnt_j$. On receipt of an $o$, the entry $(o.fcnt, y)$ in $RCVD_j$ is replaced by $(o.fcnt, y+1)$, if such an entry exists. Otherwise, an entry $(o.fcnt, 1)$ is inserted into $RCVD_j$.

A $t$ can be received when $t.tcnt=tcnt_j$, $t.fcnt=fcnt_j$, and an entry $(t.fcnt, t.ocnt)$ exists in $RCVD_j$. After the receipt of the $t$, $tcnt_j$ is incremented by one.

Lemma 3.4.6 Messages received according to the above implementation satisfy Properties $F$, $T$, and $O$.

Proof: It can be shown by induction on the number of two way flush messages received that all two way flush messages are received in the order they are sent. Since
an \( o \) carries the number of two way flush messages sent before it and is received only when \( o.tcnt=tcnt \), i.e., when \( o.tcnt \) two way flush messages have been received, it can be deduced that an \( o \) is received only when all messages in its flush receive before set has been received. Thus, the receipt of an \( o \) does not violate Properties \( F, T, \) and \( O \).

By using an argument similar to above and by constructing an induction proof similar to that of Lemma 3.4.1, it can be shown that an \( f \) is received only after all messages in its flush receive before set have been received. Thus, the receipt of an \( f \) does not violate Properties \( F, T, \) and \( O \).

Since the receipt of both \( f \) and \( o \) does not violate Properties \( F, T, \) and \( O \), and all two way flush messages are received in the order sent, we can infer that all messages sent after a \( t \) are received after it. An induction proof similar to that of Lemma 3.4.1 can be constructed to show that all messages sent before a \( t \) are received before it. Thus, receipt of \( t \) does not violate Properties \( F, T, \) and \( O \).

Observations similar to those made in the previous subsection can be made for this implementation.

### 3.4.8 \( b, t, \) and \( o \)

In this section, we present an implementation of a channel which supports messages of type \( b, t, \) and \( o \), only.

For such channels, the flush receive before set for a \( b \) or a \( o \) is the set of all \( b \) and two way flush channels sent before them. The flush receive before set for a \( t \) is the set of all messages \((t, b, \) and \( o)\) sent before it. Since unlike the previous two
channels, both $b$ and $t$ appear in the flush receive before set together, the sender needs to only maintain one counter to count the number of backward flush and two way flush messages sent so far.

A sender $p_i$ maintains a count of the number of backward and two way flush messages sent so far and the number of ordinary messages sent since the last $t$. These two counters are denoted by $btcnt_i$ and $ocnt_i$, respectively.

Both backward flush and ordinary messages carry the value of $btcnt_i$ at the time they are sent. A $t$ carries the value of both $btcnt_i$ and $ocnt_i$ at the time it is sent.

A receiver $p_j$ maintains two counters: $btcnt_j$ and $ocnt_j$. $btcnt_j$ is used to count the number of backward flush and two way flush messages received so far. The counter $ocnt_j$ is used to count the number of ordinary messages received since the last two way flush received.

An $b$ can be received only when $b.btcnt=btcnt_j$. An $o$ can be received only when $o.btcnt=btcnt_j$. A $t$ can be received only when $t.btcnt=btcnt_j$ and $t.ocnt=ocnt_j$.

After the receipt of a $b$ or $t$ the receiver increments $btcnt_j$ by one. Also, after the receipt of a $t$, $ocnt_j$ is reset to 0. $ocnt_j$ is incremented by one after the receipt of an $o$.

Lemma 3.4.7 Messages received according to the above implementation satisfy Properties $B$, $T$, and $O$.

Proof: Note that both backward flush and two way flush messages carry $btcnt$, the number of backward and two way flush messages sent so far. It can be shown by induction on the number of backward flush and two way flush messages received so far that all backward flush messages and two way flush messages are received in the order they are sent. Since an $o$ carries the value of $btcnt$ with it and is received only
when \( o_{btcnt} = btcnt \), it can be deduced that an \( o \) is received only when all messages in its \textit{flush receive before} set has been received. Thus, the receipt of an \( o \) does not violate Properties \( B, T, \) and \( O \).

Since the receipt of an \( o \) does not violate Properties \( B, T, \) and \( O \), and all backward flush and two way flush messages are received in the order they are sent, we can infer that all messages sent after a \( b \) or a \( t \) are received after the \( b \) or \( t \), respectively, is received. Hence, the receipt of an \( b \) does not violate Property \( B, T, \) and \( O \).

An induction proof similar to that of Lemma 3.4.1 can be constructed to show that all messages sent before a \( t \) are received before it. Thus, the receipt of an \( t \) does not violate Property \( B, T, \) and \( O \). \( \square \)

In this implementation, backward flush and ordinary messages carry only one integer, \( btcnt \). This contrasts with the KCA implementation where each backward flush and ordinary message carries two integers: its identity, which is a function of the total number of messages sent so far and the identity of the last backward or two way flush message sent so far. However, in the KCA implementation a two way flush message carries only one integer, its identity. In the proposed implementation, a two way flush message carries two integers. Since most applications use fewer two way flush messages than other types, such an overhead is not significant. In the proposed implementation, the burden of implementing the receipt order restrictions due to a message that is sent in the future is not passed on to ordinary messages by not requiring them to carry any additional data.

In the proposed implementation, a receiver \( p_j \) maintains only two counters \( btcnt_j \) and \( ocnt_j \). Thus, the receiver's logic is greatly reduced when compared to that in the
KCA implementation.

Table 4 summarizes the counters maintained by a sender, the information carried by a message, and the data structures maintained by a receiver in the implementations discussed in the previous three sections.

Table 4: Summary of the data structures maintained by a sender, information carried by a message, and the data structures maintained by a receiver when only three flush message types are implemented.

<table>
<thead>
<tr>
<th>Msg Type Implemented</th>
<th>Counters at Sender</th>
<th>Information carried by message of type</th>
<th>Receivers Data structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>f</td>
<td>b</td>
</tr>
<tr>
<td>$f, b, o$</td>
<td>$fcnt, bcnt, ocnt$</td>
<td>$fcnt, bcnt, ocnt$</td>
<td>$bcnt$</td>
</tr>
<tr>
<td>$f, t, o$</td>
<td>$fcnt, tcnt, ocnt$</td>
<td>$fcnt, tcnt, ocnt$</td>
<td>$tcnt, fcnt$</td>
</tr>
<tr>
<td>$b, t, o$</td>
<td>$btcnt, ocnt$</td>
<td>$btcnt$</td>
<td>$btcnt, ocnt$</td>
</tr>
</tbody>
</table>

3.4.9 $f$, $b$, $t$, and $o$

In this subsection, we describe an implementation of a channel that supports all four message types. This implementation is based on combining the implementations discussed in Subsections 3.4.7 and 3.4.8.

The sender $p_i$ maintains three counters, namely, $btcnt_i$, $fcnt_i$, and $ocnt_i$, to count the number of backward and two way flush messages sent so far, the number of $f$ messages sent so far, and the number of ordinary messages sent since the last $f$
message, respectively.

A \( t \) and an \( f \) carry the value of \( \textit{btcnt}_i, \textit{fcnt}_i \), and \( \textit{octnt}_i \) at the time they are sent. A \( b \) carries the value of \( \textit{btcnt}_i \) at the time it is sent, where as an \( o \) carries the value of \( \textit{btcnt}_i \) and \( \textit{fcnt}_i \) at the time it is sent.

The receiver \( p_j \) maintains two counters, \( \textit{btcnt}_j \) and \( \textit{fcnt}_j \). \( \textit{btcnt}_j \) is used to count the number of backward and two way flush messages received so far. \( \textit{fcnt}_j \) is used to count the number of \( f \) messages received so far. The receiver also maintains the set \( \textit{RCVD}_j \) as described in Subsection 3.4.3.

An \( f \) becomes \textit{receivable} when \( \textit{f.btcnt} = \textit{btcnt}_j \), and an entry \( \langle \textit{f.fcnt}, \textit{f.octnt} \rangle \) exists in \( \textit{RCVD}_j \). After the receipt of the \( f \), \( \textit{fcnt}_j \) is incremented by one and the entry \( \langle \textit{f.fcnt}, \textit{f.octnt} \rangle \) deleted from \( \textit{RCVD}_j \).

A \( t \) becomes \textit{receivable} when \( \textit{t.btcnt} = \textit{btcnt}_j \), and an entry \( \langle \textit{t.fcnt}, \textit{t.octnt} \rangle \) exists in \( \textit{RCVD}_j \). After the receipt of the \( t \), \( \textit{btcnt}_j \) is incremented by one.

A \( b \) becomes \textit{receivable} when \( \textit{b.btcnt} = \textit{btcnt}_j \). After the receipt of the \( b \), \( \textit{btcnt}_j \) is incremented by one.

An \( o \) becomes \textit{receivable} when \( \textit{o.btcnt} = \textit{btcnt}_j \). After the receipt of an \( o \), the entry \( \langle \textit{o.fcnt}, y \rangle \) in \( \textit{RCVD}_j \) is replaced by \( \langle \textit{o.fcnt}, y+1 \rangle \), if such an entry exists. Otherwise, an entry \( \langle \textit{o.fcnt}, 1 \rangle \) is inserted into \( \textit{RCVD}_j \).

\textbf{Lemma 3.4.8} Messages received according to the above implementation satisfy Properties \( F, B, T, \) and \( O \).

\textbf{Proof:} Each \( b \) and \( t \) carries the value of \( \textit{btcnt}_i \). Similar to Lemma 3.4.7, it can be shown that all backward flush and two way flush messages are received in the order they are sent.
Since the \textit{flush receive before} set of an \( o \) only includes messages of type \( b \) and \( t \) that were sent before it, it follows from Lemma 3.4.7 that an \( o \) will be received only after all messages in its \textit{flush receive before} set have been received. Thus the receipt of an \( o \) does not violate any flush property.

An argument similar to the one above can be made to show that an \( f \) is received only after all backward flush and two way flush messages sent before it have been received. A proof similar to Lemma 3.4.1 can be constructed to show that all ordinary messages sent before an \( f \) are received before the \( f \) is received. Thus receipt of an \( f \) does not violate any flush property.

Since the receipt of an \( o \) or \( f \) does not violate the flush property and all backward and \( f \) messages are received in the order they are sent, it can be deduced that all messages sent after a backward flush or a two way flush are received after the backward flush or the two way flush, respectively. Thus the receipt of a \( b \) does not violate any flush property.

An induction proof similar to Lemma 3.4.1 can be constructed to show that all messages sent before a two way flush are received before it. Thus the receipt of a \( t \) does not violate any flush property. \( \square \)

When compared to the KCA implementation, in the proposed implementation backward flush messages carry fewer counters, ordinary flush messages carry the same number of counters, and both two way flush and \( f \) messages carry an extra counter. It is also interesting to compare the rate at which the counters increase; In the KCA implementation, since each message carries its identity, one counter carried by each message is a function of the total number of messages sent since the last two way
flush. In the proposed implementation, the counters (except $ocnt$) increase only when a flush message is sent. Thus, if a computation uses fewer flush messages then the information carried by an $o$ will be smaller than that in the KCA implementation.

When we compare the amount of information maintained by the receiver and its effect on the time spent while checking the receivability of a message, we find that the size of the set $RCVD_j$ in the proposed implementation is bounded by the number of $f$ messages sent so far. This compares favorably with the KCA implementation, where the size is bounded by the total number of flush messages sent. In a system in which the probability of messages being delivered out of order is high, as is the case with wide area networks, the size of set $RCVD_j$ affects the time spent in deciding the receivability of a message. Note that for the implementations proposed in this chapter, maintaining the $RCVD_j$ set as an ordered set greatly reduces the time spent in determining the set membership.

### 3.5 Summary

In this chapter we proposed two implementations of the global flush property $G$ which is satisfied when a global flush message is received by a process. We defined the notion of a receive before set of a global flush message $g$ as the set of global flush messages that must be received before $g$ can be received, so that the global flush property is satisfied.

The first implementation is a counter-based implementation which assumes that the underlying system is connected by non-FIFO channels. In this implementation, each message sent by a process along a point-to-point channel is uniquely identified among all messages sent by it along the same channel. For each pair of processes $p_j$
and $p_k$ in the system, each site $S_i$ uses counters to keep track of the last message sent from $p_j$ to $p_k$ and the last $gb$ or $gt$ sent from $p_j$ to $p_k$. Each message carries the value of these counters with it. These values are used to determine the receivability of a message (based on its receive before set).

The second implementation is a F-channel based implementation which assumes that the underlying system is connected by F-channels. This implementation uses a diffusion process, which results in a point-to-point flush message to be sent along each F-channel in the underlying system which is on the path to a destination. As shown, each message carries just one bounded integer and the information carried by each message is also bounded by the size of this integer. The overhead, both in terms of the size of information carried by a message and the size of information kept at each site, is bounded and independent of the number of processes in the system.

We also presented a fair and efficient implementation of F-channels. As opposed to implementation of F-channels proposed in the past, this implementation does not penalize computations which use fewer flush messages. In previous implementations, the size of data structures kept at each site varied inversely with the number of flush messages used in the computations. It has been our observation that in most applications not all flush messages are used. Thus, we proposed independent implementation of pairs and triples of flush message types. We also give an implementation supporting all four flush primitives.
CHAPTER IV
Applications

In this chapter we discuss the following applications of the GS primitive: obtaining consistent cuts, snapshots, and global snapshots; termination detection; bounding logical clocks; implementing shared token; and updating replicated data with mutual exclusion. Before considering each application in detail, we briefly describe them.  

Consistent cuts, snapshots, and global snapshots (uses go, and gb): In Section 4.1, we discuss algorithms to obtain a consistent cut, a snapshot, and a global snapshot of a distributed system. A consistent cut is defined [15, 52] as a set of instants, one on each process, such that if $g$ is received by $p_j$ before $p_i$'s instant in the set, then $g$ is sent by $p_{se}$ before $p_{se}$’s instant in the set. A snapshot is a set of states, one on each process, along a consistent cut. Recording snapshots has applications for setting breakpoints [49] and halting. A global snapshot is a set of states, one on each process, along a consistent cut and the states of each channel $c_{i,j}$ at the instant of recording the states of $p_i$ and $p_j$. Recording global snapshots has applications in debugging, distributed checkpointing [59], protocol verification [34], setting breakpoints in distributed programs [49], and in general stable property detection [14].

In all these applications, it is assumed that all messages are sent using the GS primitive.
Termination Detection (uses gf): In Section 4.2, we consider the problem of termination detection in a distributed system. A distributed computation is considered to be terminated if every process in the system is terminated and there are no messages in the system that have been sent but have not been received.

We propose an algorithm to detect termination of a distributed computation (to be referred to as basic computation). In this algorithm the messages for termination detection are processed and sent out by a process only when it is idle. Thus, it is expected that these messages will not interfere much with the computation. Each message used to detect termination carries only a boolean value and thus will incur very little communication overhead. The proposed algorithm is symmetric in the sense that any arbitrary process can start the algorithm and multiple invocations can be active in parallel. Marker based algorithms [25, 39, 42, 46] to detect termination in a distributed environment could potentially result in unbounded number of messages to be exchanged due to unsuccessful termination detection attempts. The proposed algorithm, on the other hand, uses bounded number of messages even in the worst case.

The beauty of this algorithm lies in its simplicity. It is based on identifying a cut at which all processes in the basic computation are idle and then detecting whether there were any messages in transit at that cut. We also discuss optimizations to this basic algorithm.

Due to the nature of the global flush messages, this algorithm does not suffer from the problem faced by other marker based algorithms, i.e., the number of control messages used, due to unsuccessful attempts at detection of termination, is not

\[2\text{The algorithm requires that at least the GF message type, described later, be supported.}\]
unbounded in the worst case. The algorithm ensure that in every phase, at least one basic message becomes receivable. Thus the number of control messages in the worst case is bounded by $O(M \times N)$, where $M$ is the number of basic messages exchanged after the detection algorithm has started and $N$ is the number of processes in the basic computation.

**Bounding Logical Clocks:** In Section 4.3, we define the notion of bounded clocks. We then present an algorithm BVC (stands for “Bounded Vector Clocks”) to bound vector clocks [22, 47], under the assumption of bounded message delay.

In a system with bounded clocks, many messages may have the same timestamp value. The proposed algorithm deals with this ambiguity by dividing the process time line into windows and shadow windows and by ensuring that the timestamp of a message received in a shadow window is unique with respect to other messages received within the same shadow window. This is achieved by limiting the set of sender’s windows from which a process can receive messages in the current shadow window to the last $W$ windows. We discuss applications exemplifying the advantage of such bounded clocks. Use of bounded clocks simplifies algorithm development in a distributed environment by bounding the logs and other information kept by a process. Bounded clocks result in saving of space in messages by limiting the size of the timestamp carried by each message. Such bounded clocks also help in trace-based debugging of distributed programs, checkpointing, controlling logs, and predicting average maximal parallelism. We modify the algorithm BVC to bound vector clocks when the assumption of bounded message delay is removed. However, this requires inhibition on sending of some of the messages.
Shared token (uses \(gf\) or \(gb\)): In Section 4.4, we discuss the implementation of a shared token supporting operations to request it, to pass it, and to determine its current holder.

Replicated data with mutual exclusion (uses \(gb\), \(gf\), \(gt\), and \(go\)): A reason for implementing a shared token is to enforce mutual exclusion in accessing a shared resource or a replicated data item. In Section 4.5 we show how a 1-copy behavior [10] can be obtained for a replicated data by using token passing and updating scheme implemented by the \(GS\) primitive.

4.1 Consistent Cuts, Snapshots, and Global Snapshots

Since the \(GS\) primitive can be used by a process to send a message to every process in the system, the system can be viewed as consisting of \(N\) processes connected by \(N^2\) logical channels, i.e., there is a logical channel between each pair of processes. Note that because the sender of \(g\) can also be in \(g.dst\), there is a logical channel from a process to itself.

The algorithm for recording a snapshot is super-imposed on the processes and does not directly affect the processes except for possibly affecting the order of receipt of messages. Messages sent by the snapshot algorithm will be referred to as a marker. More specifically a \(gb\) sent as a marker will be referred to as a \(gb\)-marker.

Algorithm Snapshot:

*The initiator sends a \(gb\)-marker to all processes including itself.* \(^3\) On receiving a \(gb\)-marker, the receiver records its state before executing the next event.

In the above algorithm, if recording the state of a process is replaced by just the

\[^3\text{Invoked as } GS(GB, gb\text{-marker}.dst, \text{ 'marker' }), \text{ where } \forall j::gb\text{-marker}.dst[j]=1.\]
identification of the event for the process, then it gives a consistent cut.

**Lemma 4.1.1** Algorithm Snapshot gives a snapshot.

**Proof:** To prove that the above algorithm gives a snapshot, we need to show that for every \( grecv(g)_j \) before \( p_j \)'s instant in the cut, \( gsend(g) \) is also before \( p_{g.se} \)'s instant in the cut, i.e., if \( gsend(g) \) is after \( p_{g.se} \)'s instant in the cut, then \( grecv(g)_j \), for each \( j \in g.dst \), is also after \( p_j \)'s instant in the cut.

Note that if \( gsend(g) \) is after \( p_{g.se} \)'s instant in the cut then \( g \in F(g) \). The proof, thus, follows from Property \( Gb \). \( \square \)

Lai and Yang [43] propose an algorithm to record a global snapshot. That algorithm can be adapted to a system supporting the \( OS \) primitive as follows: Each process records in its log all messages sent and received along each outgoing and incoming channel, respectively. The channel state is the set of messages sent along a channel minus the set of messages received along the same channel.

The initiator spontaneously sends a \( gb{-}marker \) to all processes. On receiving a \( gb{-}marker \), \( p_j \) records its state and the set of messages sent and received along each outgoing and incoming channel. \( p_j \) then sends \( p_j \)'s state and the log of messages sent and received to the initiator \( gb{-}marker.se \), using a \( go \) message.

### 4.2 Termination Detection

In this section, we consider the problem of termination detection in a distributed system. Messages exchanged by processes in the underlying distributed computation (also called basic computation) will be referred to as the basic messages. At any time during its execution, a process in the basic computation can be either active or
idle. Only an active process can send a basic message. An active process can also spontaneously become idle. An idle process, on the other hand, remains idle unless it receives a basic message. The basic computation is said to have been terminated at a cut when all processes in the basic computation are in idle state at the cut and there are no basic messages in transit at the cut. A global flush message $g$ is said to be "in transit" at a cut if the event of sending of $g$ is in the cut and at least at one of its destinations the event of its receipt is not in the cut, i.e., the global flush message $g$ has been sent but has not yet been received at all its destinations. We confine our discussion to computations in which initially at least one process is active and each process will eventually become idle.

The sites, for each process in the basic computation, run the termination detection algorithm and thus exchange messages, to be referred to as the control messages, which are transparent to processes, i.e., are not received by any $p_j$.

In Section 4.2.1, we describe the basic algorithm and prove its correctness. In Section 4.2.3, we describe enhancements to the basic algorithm and discuss implementation issues. Section 4.2.4 discusses related work.

4.2.1 Basic Algorithm

In this section, we describe the basic algorithm to detect termination in a distributed environment. Here, we stress more on the basic idea underlying the algorithm rather than on message efficiency. Later in Section 4.2.3, we improve the algorithm to reduce the number of control messages used.
Basic Idea

The termination detection algorithm has two phases. In both the phases, the initiator site, on becoming idle, sends a $gf$ marker to all other sites each of whom responds by sending a $gf$ marker to the initiator when it becomes idle. A phase terminates when the initiator receives a response from all other sites.

In phase 1, the algorithm identifies a cut, to be referred to as phase-1-cut, such that all processes of the basic computation are idle at this cut. On the termination of phase 1, the initiator knows that all processes in the basic computation were idle at phase-1-cut. This does not necessarily imply that the basic computation has terminated because there could be basic messages in transit at the cut which will eventually make some process in the basic computation active.

In phase 2, the algorithm determines whether there were any basic messages in transit at the phase-1-cut. Note that, each message $g$, in transit at phase-1-cut is in the "past of" the $gf$ marker sent by $S_{g,sc}$ to the initiator of the termination detection algorithm, and hence will be in the past of the marker sent by the initiator in phase 2. Since markers used in phase 2 are also $gf$, they will be received only after all messages in their past (including all messages that are in transit at phase-1-cut) have been received (Property $Gf$). Each site thus needs to keep a boolean variable (initialized to $false$ at phase-1-cut) to detect if there were any messages in transit at the phase-1-cut. Thus, if there were any messages in transit at phase-1-cut, they will be received before the markers for phase 2 are received and the termination detection algorithm will signal that the basic computation did not terminate at phase-1-cut. Otherwise, the termination detection algorithm signals that the basic computation has indeed terminated at phase-1-cut.
The Algorithm

Figure 8, shows the basic algorithm executed at site $S_i$. Lines 0.0, 1.0, and 2.0 describe the actions taken by $S_i$ for events in $p_i$.

$S_i$ initiates the algorithm only when $p_i$ becomes idle. Each site maintains a modulo $2N$ counter $cnt$ to count the number of responses received. For each invocation of the algorithm, $S_i$ will receive $2N - 2$ responses ($N - 1$ for each phase). Phase 2 of the algorithm starts only when phase 1 finishes and for each site at most one invocation can be in progress. Thus, a site can invoke the algorithm (i.e., phase 1) only when $cnt=0$. Phase 1 terminates when $cnt=N$. Phase 2 terminates when $cnt=2N$ ($cnt$ is then reset to zero).

Each site $S_j$ maintains a boolean variable $Intransit_i$ to detect messages in transit at the phase-1-cut for an invocation of the algorithm by $S_i$. $Intransit_i$ is set to $false$ at the phase-1-cut and is set to $true$ on the receipt of the first basic message after the phase-1-cut.

$S_i$ marks the start of phase 1 by sending a control message $init_1$ to all other sites when $p_i$ becomes idle. It also sets $Intransit_i$ to $false$. At any time a site can have at most one invocation of the algorithm in progress. Since one of the requirements for a computation to terminate is that there are no basic messages in transit and a message of type GF ensures that all messages in its past are received before it, $init_1$ should be of type GF. We will now denote $init_1$ by $gf-init_1$ to emphasize the fact that it is a $gf$ message.

3.0: \[ state=\text{idle} \land cnt=0 \land S_i \text{ wants to initiate the detection alg} \rightarrow \text{/*Initiate Phase-1*/} \]
\[ Intransit_i:=false \]
\[ cnt++ \quad \text{/*Sets guard to false*/} \]

\(^4\)One such variable is maintained for each site which can invoke the algorithm.
Send $gf$--init$_1$ by executing $GS(GF, AllButMe, "Init-1")$

Each site, $S_j$, responds to $gf$--init$_1$ by sending a $gf$--ack$_1$ (denotes a $gf$ message used as an acknowledgment in phase 1) to $S_i$ when $p_j$ becomes idle. On receipt of a $gf$--init$_1$ from $S_i$, $S_j$ sets repl$_1$ to true to indicate that $S_j$ has to send $gf$--ack$_1$ to $S_i$. The boolean variable repl$_1$ is used by $S_j$ to keep track of all un-responded $gf$--init$_1$'s. $S_j$ responds to them only when $p_j$ is idle. Note that at any time for any $S_i, S_j$ can have at most one unresponded $gf$--init.

4.1: $gf$--init$_1$ is receivable at $S_i \rightarrow repl_{gf$--init$_1$.se}: = True$

$S_j$ acknowledges all outstanding $gf$--init$_1$'s by sending $gf$--ack$_1$ when $p_j$ becomes idle.

4.2: $\exists j: repl_j \land \neg state= idle \rightarrow$

$In transit_{gf$--init$_1$.se} : = false$

$repl_j : = false$

send $gf$--ack$_1$ to $gf$--init$_1$.se by executing $GS(GF, gf$--init$_1$.se, "Ack-1")

The instant of sending of $gf$--init$_1$ and the instants of sending of $gf$--ack$_1$ identify a cut which will be referred to as phase-1-cut. Phase 1 terminates when $S_i$ has received $N - 1$ $gf$--ack$_1$'s.

5.1: $gf$--ack$_1$ is receivable at $S_i \rightarrow cnt++$

On termination of phase 1, $S_i$ knows that all processes in the basic computation were idle at phase-1-cut. Note that the algorithm does not require phase-1-cut to be consistent.

$S_i$ then initiates phase-2 to determine whether there were any messages in transit at the phase-1-cut. $S_i$ initiates phase 2 by sending a $gf$--init$_2$ to all sites.
5.2: \( \text{state=idle} \land \text{cnt=N} \rightarrow \) /*Initiate Phase-2*/
\( \text{cnt}++ \) /*Sets guard to false*/
\( \text{Done:= false} \)
send \( \text{gf-init}_2 \) by executing \( GS(GF, \text{AllButMe}, "\text{Init-2}" ) \)

Each site \( S_j \) responds to \( \text{gf-init}_2 \) by sending a \( \text{gf-ack}_2 \) to \( S_i \) when \( p_j \) becomes idle. On receipt of \( \text{gf-init}_2 \) from \( S_i \), \( S_j \) sets \( \text{rep}_2 \) to \( \text{true} \) to indicate that \( S_j \) has to send \( \text{gf-ack}_2 \) to \( S_i \). The boolean variable \( \text{rep}_2 \) is used by \( S_j \) to keep track of all un-responded \( \text{gf-init}_2 \)'s.

6.1: \( \text{gf-init}_2 \) is receiveable at \( S_i \) \( \rightarrow \) \( \text{rep}_2 \text{gf-init}_2 \text{.se} := \text{true} \)

\( S_j \) acknowledges all outstanding \( \text{gf-init}_2 \)'s by sending \( \text{gf-ack}_2 \) when \( p_j \) becomes idle. \( \text{gf-ack}_2 \) contains the value of \( \text{Intransit}_i \).

6.2: \( \exists j:: \text{rep}_2 \land \text{state=idle} \rightarrow \)
send \( \text{gf-ack}_2 \) to \( \text{gf-init}_2 \text{.se} \) by
executing \( GS(GF, \text{gf-init}_2 \text{.se}, \text{Intransit}_j \text{-ack}_2 \text{.se}) \)
\( \text{rep}_2 j \text{:= false} \)

Phase 2 terminates when \( S_i \) receives \( N-1 \) \( \text{gf-ack}_2 \), one for each \( S_j \). If at the end of phase 2, the value of all \( \text{Intransit}_i \)'s (including \( S_i \)'s) received by \( S_i \) is \( \text{false} \), then the basic computation has terminated; otherwise, it has not.

7.0: \( \text{gf-ack}_2 \) is receiveable at \( S_i \) \( \rightarrow \)
\( \text{Done:= Done} \lor \text{Intransit}_j \text{-ack}_2 \text{.se} \)
\( ++\text{cnt:=}2*N \rightarrow \) /*Phase-2 complete*/
\[ \) (Done \lor \text{Intransit}_i ) \rightarrow "\text{NO TERMINATION YET}" \( \text{cnt:=0} \)
\] \( \neg (\text{Done} \lor \text{Intransit}_i ) \rightarrow "\text{COMPUTATION TERMINATED}" \( \) ]
The boolean variable Done is initialized to \textit{false} at the \textit{phase-1-cut}. It is used to keep track of the boolean values \textit{Intransit}, (denoting, whether there were messages in transit at the \textit{phase-1-cut}) reported by other sites. If at the end of \textit{phase 2} Done is still \textit{false}, then the computation has terminated.

### 4.2.2 Correctness

To prove the correctness of the proposed algorithm, we need to show that (a) if the basic computation has terminated before \textit{phase-1-cut}, the termination is reported in the second phase, and (b) the algorithm does not report any false terminations.

**Lemma 4.2.1** If the basic computation has terminated before \textit{phase-1-cut}, it will be detected in phase 2.

**Proof:** Since the basic computation has terminated, there are no basic messages in transit at the \textit{phase-1-cut}. Thus, \textit{Intransit} will never be set to \textit{true} at any site. All \textit{gf-ack}_2 will report the value of \textit{Intransit} is \textit{false}. Hence, the termination will be detected in \textit{phase 2}. □

**Corollary 4.2.1** If there are no basic messages in transit at the \textit{phase-1-cut}, then \textit{phase 2} detects termination of the basic computation.

**Proof:** All processes are idle at \textit{phase-1-cut}. Since there are no basic messages in transit, the basic computation has terminated (follows from definition of termination). The rest follows from Lemma 4.2.1. □
$S_i:: \star$

0.0: $p_i$ sends a message $g \land state=active$ →
execute $GS(g,typ,g.dst,g.body)$

1.0: $p_i$ is ready to receive a message $\land \exists g :: g$ is a basic message receivable at $p_i$ →
execute $Rcv(g,typ,g.dst,g.body)$ to $p_i$
state:=active
\[ \forall j:: Intransit_j \rightarrow Intransit_j:= True \]

2.0: $state=active \rightarrow state:=idle$

3.0: $state=idle \land cnt=0 \land S_i$ wants to initiate the detection alg → /*Initiate Phase-1*/
\[ Intransit_i:=false \]
\[ cnt++ \]
Send $gf-init_i$ by executing $GS(GF, AllButMe, "Init-1")$

4.1: $gf-init_i$ is receivable at $S_i$ → $rep_{gf-init_i.se}=True$

4.2: $\forall j:: repl_j \land state=idle$ →
\[ Intransit_{gf-init_i.se}:= false \]
\[ rep_j:= false \]
send $gf-ack_j$ to $gf-init_i.se$ by executing $GS(GF, gf-init_i.se, "Ack-1")$

5.1: $gf-ack_j$ is receivable at $S_i$ → $cnt++$

5.2: $state=idle \land cnt=N$ → /*Initiate Phase-2*/
\[ cnt++ \]
\[ Done:= false \]
send $gf-init_2$ by executing $GS(GF, AllButMe, "Init-2")$

6.1: $gf-init_2$ is receivable at $S_i$ → $rep_{gf-init_2.se}=True$

6.2: $\forall j:: rep_{gf-init_2.se} \land state=idle$ →
send $gf-ack_2$ to $gf-init_2.se$ by
executing $GS(GF, gf-init_2.se, Intransit_{gf-ack_2.se})$
\[ rep_{gf-ack_2.se}:= false \]

7.0: $gf-ack_2$ is receivable at $S_i$ →
Done:= Done $\lor Intransit_{gf-ack_2.se}$
\[ ++cnt=2*N \rightarrow /*Phase-2 complete*/ \]
\[ \begin{cases} (Done \lor Intransit_i) \rightarrow "NO TERMINATION YET" \\
\quad cnt:=0 \\
\quad \neg (Done \lor Intransit_i) \rightarrow "COMPUTATION TERMINATED" \end{cases} \]

8.0: $True \rightarrow skip$

Initially $state=active \land state=idle, \forall j:: Intransit_j:= True, cnt:=0$
$AllButMe[i]:=0, \forall j \neq i: AllButMe[j]:=1 repl_j:=rep_{gf-init_i.se}:= false$

Figure 8: Basic algorithm executed at each site $S_i$ to detect termination of the basic computation.
Let \textit{phase-2-cut} denote the cut identified by the instant of sending of \textit{gf-init}_2 and the instants of sending of \textit{gf-ack}_2 for an invocation of the algorithm.

**Lemma 4.2.2** If there are basic messages in transit at the phase-1-cut, then they will be received at all their destinations before the phase-2-cut.

**Proof:** Since each process \( S_j \) sends a \textit{gf-ack}\(_1\) only when \( p_j \) is idle, all messages sent by \( p_j \) until then are in \( \mathcal{P}(\textit{gf-ack}\_1) \) where \( \textit{gf-ack}\_1.se=j \). Phase 2 is initiated only when \( S_i \) has received all \textit{gf-ack}\(_1\), thus \( \forall g, \forall \textit{gf-ack}\(_1\): g \in \mathcal{P}(\textit{gf-ack}\_1): g \in \mathcal{P}(\textit{gf-init}\_2) \). This implies that each message \( g \) sent by \( p_j \) before sending \textit{gf-ack}\(_1\) will be received by \( p_k, p_k \in g.dst \), before \( S_k \) receives \textit{gf-init}\(_2\). \( \square \)

**Lemma 4.2.3** If there are basic messages in transit at phase-1-cut, the algorithm reports in phase 2 that the basic computation has not terminated.

**Proof:** Proof follows from Lemma 4.2.2: If there are basic messages in transit at phase-1-cut, they will be received before the corresponding phase-2-cut, thereby setting \textit{Intransit} at receiver sites to \textit{true}.

Thus if there is a basic message in transit at phase-1-cut, termination of the basic computation will not be reported in phase 2. \( \square \)

**Lemma 4.2.4** The algorithm is correct.

**Proof:** Follows from Lemmas 4.2.1 and 4.2.3. \( \square \)
Corollary 4.2.2 For an invocation of the termination detection algorithm to report that the basic computation has not terminated, at least one basic message must be in transit at its phase-1-cut.

Proof: Follows from Corollary 4.2.1.

Lemma 4.2.5 If $M$ basic messages are sent in the basic computation after the termination detection algorithm is invoked for the first time, then a site $S_i$ will detect termination of the basic computation in at most $M + 1$ invocations of the algorithm.

Proof: Proof follows from the previous discussion that at least one message must be in transit at a phase-1-cut for the algorithm to report that the basic computation has not terminated (Corollary 4.2.2) and all messages in transit at a phase-1-cut are received before the corresponding phase-2-cut (Lemma 4.2.2).

Thus, in the worst case exactly one basic message will be in transit at the phase-1-cut for each invocation. $M$ invocations will be required for $M$ basic messages sent after the first invocation of the termination detection algorithm. For $M$ messages to be sent after the first invocation of the algorithm, there must be basic message in transit at the phase-1-cut of the first invocation, thus requiring an additional invocation in the worst case.

Discussion

A $gf$ marker, as used in the above algorithm provides a very powerful tool in a system which does not have FIFO channels. It can be used to "flush" channels, therefore,
all messages in the “past of” a gf can be claimed to have been delivered when a gf is delivered. This capability is not available in “marker” messages which are used by other existing algorithms because markers are of type go. This results in two problems: First, since markers can repeatedly overtake a basic message sent before them any number of times, for every basic message the potentially unbounded number of attempts for detecting termination may be required. Second, no claim can be made about messages sent in the “past of” a marker. To overcome these problems, such algorithms resort to techniques such as message counting [39], timestamping [37, 38], etc.

We take the first problem for further clarification. Consider a system of 5 processes numbered p₀ through p₄. Each pᵢ is connected to pᵢ₊₁ and pᵢ₋₁ by non-FIFO channels such that they are on a ring topology. For simplicity, we will assume that the termination detection algorithm is a four counter algorithm described in [46]. Each process counts the number of messages sent and received. A marker sent by pᵢ goes around the ring to collect Pᵢ and Cᵢ, the number of messages produced (sent) and consumed (received), respectively. If for two successive rounds of the ring, pᵢ finds the number of messages produced and consumed to be the same, then pᵢ can deduce that the basic computation has terminated. Now consider the following computation: All processes except p₀ are idle. p₀ sends a basic message g to p₂ and then becomes idle. This message takes very long time to reach p₂. p₀ sends a marker along the ring which crosses g. Since a marker can cross g any number of times before g reaches p₂, the number of control messages exchanged to detect termination in this case is unbounded.

Lemma 4.2.6 After the termination of the basic computation, the basic algorithm
takes $O(N)$ control messages to detect termination.

Proof: Follows from Lemma 4.2.1. \qed

4.2.3 Enhancements and Implementation Issues

In this section, we suggest some ideas for improving the message efficiency of the basic algorithm described in the previous section.

Note that the two phases of the basic algorithm are similar in both the type and the pattern of the global flush control messages exchanged by the sites. Phase 1 of the basic algorithm serves three purposes: First, to identify phase-1-cut; Second, to ensure that all basic messages sent before the phase-1-cut are in the past of, and hence are received before, $gf$–init in phase 2; Third, to initialize the boolean variables, Intransit, to false (to detect basic messages in transit at phase-1-cut). The first two are implemented by the message type $gf$ (used in both phases). Thus, by appropriately initializing and testing the variables, for any two successive invocations of the termination detection algorithm by a site, phase 2 of the first invocation can be treated as phase 1 of the second. This will save one phase in all subsequent invocations of the algorithm.

Note that the correctness proof of the basic algorithm does not require that phase 2 of the algorithm be initiated as soon as phase 1 of the invocation completes. By appropriately initializing (as described next) the algorithm variables the phase 1 of the very first invocation of the termination detection algorithm can be assumed to have happened before the basic computation was started. Figure 9 shows the algorithm with the required modifications.
Instead of resetting $\text{Intransit}_i$ to $\text{false}$ at phase-1-cut and testing whether or not it is $\text{true}$ (to detect messages in transit at phase-1-cut), we toggle the boolean and test it for a change. Specifically, each site maintains three sets of variables namely, $O_{\text{Intransit}}$, $N_{\text{Intransit}}$, and $\text{chkval}$. Initially, all of these variables are set to $\text{false}$. $O_{\text{Intransit}}$ and $N_{\text{Intransit}}$ are used to detect the receipt of any basic message after the receipt of a $gf$-$\text{init}$ from $S_i$. On receipt of a $gf$-$\text{init}$ from $S_i$, $O_{\text{Intransit}}$ is set to $N_{\text{Intransit}}$. On receipt of the first (detected by comparing $O_{\text{Intransit}}$ and $N_{\text{Intransit}}$) basic message by $p_j$ after $S_j$ has received $gf$-$\text{init}$ from $S_i$, $S_j$ toggles $N_{\text{Intransit}}$. For $S_i$, $\text{chkval}[j]$ is the last value, observed by $S_i$, of $N_{\text{Intransit}}$ at $S_j$. It is used by $S_i$ to detect any changes in $N_{\text{Intransit}}$ at $S_j$. It is updated on receipt of a $gf$-$\text{ack}$.

The termination of the basic computation is detected by $S_i$ when for each site $S_j$ the value of $N_{\text{Intransit}}$ carried by the last $gf$-$\text{ack}$ from $S_j$ matches $\text{chkval}[j]$ at $S_i$.

### 4.2.4 Comparison to Related Work

One of the earliest algorithms for detecting termination of a distributed computation was proposed by Dijkstra and Scholten [20]. This algorithm does not assume FIFO channels and detects termination within $N$ control message transmissions after the basic computation has terminated. However it requires one control message for every basic message communicated and this number can be very large.

A fault tolerant distributed termination detection algorithm is presented in [64].

Marker based algorithms [25, 39, 42, 46] usually do not require sending a control message for every basic message. In [46], several solutions to termination detection have been discussed. These counter-based algorithms detect termination of the ba-
$S_i:: *[
0.0: \quad p_i \text{ sends a message } g \land state=active \rightarrow \\
\quad \text{execute } GS(g\.typ, g\.dst, g\.body) \\
1.0: \quad p_i \text{ is ready to receive a message } \exists g :: g \text{ is a basic message receivable at } p_i \rightarrow \\
\quad \text{execute } Rcv(g\.typ, g\.dst, g\.body) \\
\quad \text{return } (g\.typ, g\.dst, g\.body) \text{ to } p_i \\
\quad state:=active \\
\quad \forall j:: O\.Intransit_j = N\.Intransit_j \rightarrow N\.Intransit_j := ~ N\.Intransit_j \\
2.0: \quad state=active \rightarrow state:=idle \\
3.0: \quad state=idle \land cnt=0 \land S_i \text{ wants to initiate the detection alg } \rightarrow \\
\quad O\.Intransit_i := N\.Intransit_i \\
\quad \text{send } gf-init \text{ by executing } GS(GF, AllButMe, "init") \\
\quad \text{Done := true} \\
4.1: \quad gf-init \text{ is receivable at } S_i \rightarrow \\
\quad \quad rep_{gf-init.se} := true \\
4.2: \quad \exists j:: rep_j = true \land state=idle \rightarrow \\
\quad \quad \text{send } gf-ack \text{ to } gf-init.se \text{ by executing } \\
\quad \quad \quad GS(GF, gf-init.se, N\.Intransit_{gf-init.se}) \\
\quad \quad O\.Intransit_{gf-init.se} := N\.Intransit_{gf-init.se} \\
\quad \quad rep_j := false \\
5.0: \quad gf-ack \text{ is receivable at } S_i \rightarrow \\
\quad \quad \text{Done:=Done } \land (\text{checkval}[gf-ack.se]=gf-ack.Intransit) \\
\quad \quad \text{checkval}[gf-ack.se]:=gf-ack.Intransit \\
\quad \quad cnt++ \\
\quad \quad cnt=N \land checkval[i]=N\.Intransit_i \land Done \rightarrow \\
\quad \quad \text{"COMPUTATION TERMINATED"} \\
\quad \quad cnt=N \land checkval[i]=N\.Intransit_i \land \neg Done \rightarrow \\
\quad \quad \text{"NO TERMINATION YET"} \\
\quad \quad cnt:=0 \\
\quad \quad checkval[i]:=N\.Intransit_i \\
\quad \quad true \rightarrow \text{skip} \\
]$

Initially Done:= true, cnt:=0, state=active \lor state=idle, \\
\forall j::O\.Intransit_j = N\.Intransit_j = checkval[j]=false

Figure 9: The improved termination detection algorithm executed at site $S_i$. 
sic computation within two rounds of communication between all processes \(O(N)\) after the computation has terminated. These algorithms require each process to count the number of messages sent and received. The number of unsuccessful invocation of the algorithm in these cases can be potentially unbounded, thus requiring potentially unbounded number of control messages. The number of rounds of communication needed to detect termination after the basic computation has terminated, can be reduced to one [63] by appropriately augmenting each basic message by logical timestamps [44]. In many algorithms using this technique, each process still counts messages sent and received.

In [42], Kumar presents three classes of termination detection algorithms based on counting message sends and receipts along non-FIFO channels. These algorithms assume that there exists a cycle involving all the processes in the basic computation at least once. A marker traverses around the cycle to collect message counts. Two optimizations have also been suggested. These optimizations are based on reducing the number of counters (by keeping counters on a per process basis as opposed to on a per channel basis) or on parallel traversal of the cycle. After the basic computation has terminated, these algorithms require \(O(|C|)\) control message transmissions, where \(|C|\) is the number of channels in the cycle.\(^5\)

Termination detection algorithms in [37, 38] use distributed snapshots. In [37, 38], when a process becomes idle, it issues a request to all other processes to take a local snapshot. When an active process receives this request, it ignores the request. Thus, only the request of the last process to become idle will result in all processes to take a snapshot. The algorithm is based on ordering the requests using Lamport’s logical

\(^5\)For fairer comparison with other algorithms, we count the messages communicated at the same time along different channels distinctly.
clocks. As is clear from the above discussion, due to unsuccessful requests the number of control messages used in this case is also unbounded. Moreover, each process has to maintain a counter to count the number of messages sent/received.

In [13], Chandrasekaran et al. present a message optimal algorithm for distributed termination detection. This algorithm detects termination of the basic computation in $O(|E| + M)$ control messages, where $|E|$ is the number of channels in the system and $M$ is the number of messages exchanged in the basic computation after the the termination detection algorithm is initiated. This algorithm assumes FIFO communication between processes and uses a tree-based communication structure for control messages.

Performance Measures

In this section we compare the performance of the proposed algorithm with other algorithms proposed in the literature.

- **Total Number of Control Messages**: If $M$ is the number of basic messages sent after the first invocation of the algorithm, then the proposed algorithm requires at most $O(MN)$ control messages. Marker based termination detection algorithms [37, 38, 42, 46, 50] for non-FIFO channel may require unbounded control messages. On the other hand, algorithms which assume FIFO communication channels require $O(M)$ control messages.

- **Number of Control Messages After Basic Computation has Terminated**: Similar to other algorithms proposed in the literature for non-FIFO communication, our algorithm requires $O(N)$ control messages to detect termination after it has occurred.
• **Storage:** The basic algorithm requires \( N + \log(2N) \) bits of storage per site \((N\) bits per site for *Intransit*; and \( \log(2N) \) bits for *cnt*). Counter-based algorithms require potentially unbounded counters to count the number of messages sent and received.

• **Message Length:** The largest message used in our algorithm requires 1 bit for the *Intransit*; and 2 bits for distinguishing the four message types, namely \( gf\text{-init}_1, gf\text{-init}_2, gf\text{-ack}_1, gf\text{-ack}_2 \). Counter-based algorithms require potentially unbounded message lengths to pass the information regarding number of messages exchanged.

• **Detection Delay:** The detection delay is defined to be “the longest sequence of control messages needed to be sequentially sent to detect termination after the basic computation has terminated”. The detection delay in our algorithm is 4. One of the algorithms in [42] has a detection delay of 4. Algorithms proposed in [13, 46] have detection delays of at least \( \log(N) \) and \( N \), respectively.

### 4.3 Bounding Logical Clocks

Various models [4, 22, 44, 47] for logical time have been proposed in the past. For these models, clocks have been developed to assign timestamps to events so that causality or potential causality between events can be inferred from their timestamps. The values taken by these clocks increase monotonically. Thus, the clocks are unbounded, i.e., the timestamps assigned by these clocks can become arbitrarily large. In a typical distributed computation, messages carry clock values as their timestamps. Since the clocks are unbounded, the communication overhead due to these timestamps is also
unbounded. Bounding the clocks, and hence the timestamps, results in bounding the communication overhead due to the timestamps.

Use of bounded clocks simplifies algorithm development in a distributed environment by bounding the logs and other information kept by a process. Bounded clocks result in saving of space in messages by limiting the size of the timestamp carried by each message. Such bounded clocks also help in trace-based debugging of distributed programs, checkpointing, controlling logs, and predicting average maximal parallelism.

In Section 4.3.1, we define bounded clocks and present a general approach taken in this chapter to bound a clock. In Section 4.3.2, we describe an algorithm to bound vector clocks [22, 47] under the assumption of bounded message delay. In Section 4.3.4, we propose an algorithm to bound vector clocks when the assumption of bounded message delay is removed. In Section 4.3.5, we discuss some applications exemplifying the advantage of such clocks and describe some related work.

4.3.1 Basic Idea

Even though the ideas presented here can be easily applied to different models of logical time [4, 22, 44, 47] and other monotonically increasing counters, we will restrict our discussion to Vector Clocks [22, 44].

Vector Clocks

Vector clocks [22, 47] were proposed to assign vector timestamps to events to capture the $\leq$ relation between them. Each process maintains a vector clock of $N$ integers, one for each process in the system. The vector clock at $p_i$ is denoted by $vc_i$. A
message carries with it the value of the sender's clock at the time it is sent as its timestamp. The vector timestamp of \( g \) is denoted by \( g.mt \). In [22, 47], vector clock, \( vc_i \), at process \( p_i \) is maintained as follows:

- Initially, \( \forall j:: vc_i[j] := 0; \)

- On \( gsend(g) \) event, \( g \) is timestamped with the value of \( vc_i \) and the \( i^{th} \) component of \( vc_i \) is incremented by 1, i.e., \( vc_i[i] := vc_i[i] + 1 \), after sending \( g \) but before the next event;

- On \( grecv(g) \) event, after receiving \( g \) but before the next event,
  \( \forall j:: j \neq i : vc_i[j] := \max(vc_i[j], g.mt[j]) \) and \( vc_i[i] := vc_i[i] + 1. \)

**Definition 4.3.1 [Relation \( \prec \) on vector timestamps]**

Given two vector timestamps \( mt \) and \( mt' \), define \( mt \prec mt' \) iff \( \forall j:: (mt[j] \leq mt'[j]) \) and \( \exists k (mt[k] < mt'[k]) \).

When the timestamps components are unbounded integers, as in the above described clocks, then the relations \( \leq \) and \( \prec \) on vector timestamp components are the same as the relations \( \leq \) and \( < \), respectively, on integers. Later, when vector timestamp components are not integers, we will appropriately redefine \( \leq \) and \( < \).

The algorithms proposed in this chapter to bound vector clocks, map each process \( p_i \) to an augmented process \( P_i \). \( P_i \) contains all events of \( p_i \), in the same order of occurrence as in \( p_i \), interspersed with system events needed to bound a vector clock, \( vc_i \). The system events are the events of sending and receiving some special messages between the augmented processes to bound the values taken by \( vc_i \). These messages are transparent to the original processes, i.e., they do not have any effect on the computation in the original processes.
Bounded Clocks

**Definition 4.3.2 [Bounded Clock]**

A component of a clock is bounded if its value is reset to zero when it reaches some large value (which need not be pre-determined or may not be the same every time). A clock is bounded if all components of the clock are bounded.

As shown in [47], given any two events \( e_i \) and \( e_j \) with timestamps \( vt' \) and \( vt'' \), respectively, \( e_i \Rightarrow e_j \) iff \( vt'[i] \leq vt''[i] \). Since \( \Rightarrow \) between events can be deduced by comparing just one component of their timestamps, each component of the clock can be independently bounded without affecting the \( \Rightarrow \) relation. Also, note that \( \forall j : j \neq i : vc_j[i] \leq vc_i[i] \). In the rest of this section, we will discuss bounding the value of \( vc_i[i] \), i.e., how to bound the value of the \( i^{th} \) component of the vector clock. We then present an algorithm that a process \( p_i \) can use to bound its component of the clock, \( vc_i[i] \).

Each integer value \( vt[i] \) taken by a clock component \( vc_i[i] \) can be mapped to a pair \( \langle wid, vtb \rangle \) of integers, where \( vtb \) is bounded by some positive integer \( B \) and \( wid \) is unbounded. An example of such a mapping is \( wid=vt[i] \div B \), \( vtb=vt[i] \mod B \). This one-to-one mapping is similar to the one used in [4] and does not result in the loss of any information. Each clock component will now be a pair of values \( \langle vc_i[i].wid, vc_i[i].vtb \rangle \). The problem of bounding the range of values taken by \( vc_i[i] \) is thus transformed into the problem of bounding the range of values taken by \( vc_i[i].wid \).

By periodically resetting \( vc_i[i].vtb \) to zero, the process time line for \( p_i \) is broken into time-slots with boundaries at points when \( vc_i[i].vtb \) is reset. Each such time-slot will be referred to as a window. For ease of expression, the windows on a process time line are assigned an integer identifier starting from 0, called the window number, to
give the total order of occurrence of the windows on a process. We denote the \( c^{th} \)
window on \( p_i \) by \( w_{i,c} \). Note that \( vci[i].wid \) represents \( p_i \)’s current window number.
Whenever \( p_i \) transits from window \( w_{i,c} \) to \( w_{i,c+1} \), \( P_i \) sends a special message, called
\( chng \), to all other augmented processes.

Each \( p_j \)’s \(( j \neq i)\) process time line between two consecutive receipts\(^6\) of \( chng \) from
\( p_i \) by \( P_j \) is referred to as “\( p_i \)’s shadow window on \( p_j \)”. \( p_i \)’s \( c^{th} \) shadow window on \( p_j \),
which is assigned a shadow window number \( c \), is denoted by \( sw_{i,c} \). \( vc_j[i].wid \) represents
the number of \( p_i \)’s current shadow window on \( p_j \). Both window numbers and shadow
window numbers are just used to explain the concepts and are not a part of the
proposed algorithms. Figure 10 illustrates \( p_i \)’s windows and \( p_i \)’s shadow windows on
\( p_j \) and \( p_k \).

Definition 4.3.3 \([ p_j \text{ can see } w_{i,a} \text{ in } sw_{i,c} ]\)

\( p_j \) can see \( w_{i,a} \) in \( sw_{i,c} \) iff there exists a message chain \( g_0, \ldots, g_n \) such that \( g_0 \) is sent
by \( p_i \) in \( w_{i,a} \), \( g_n \) is received by \( p_j \) in \( sw_{i,c} \), and \( \forall m:1 \leq m \leq n : g_{m-1}.vt[i]=g_{m}.vt[i]. \)\(^7\)

Intuitively, “\( p_j \) can see \( w_{i,a} \) in \( sw_{i,c} \)” means \( p_j \) receives a message in \( sw_{i,c} \) such that
the \( i^{th} \) component of its timestamp was set due to an event in \( w_{i,a} \). Thus in \( sw_{i,c} \), \( p_j \)
is aware of window \( w_{i,a} \). If \( p_j \) is aware of \( W \) such windows in the timeline of process
\( p_i \), then the \( i^{th} \) component of each of timestamps in each of these windows will need
to be distinguished from each other by assigning a distinct value to \( vc_i[i].wid \). To
bound \( vc_i[i].wid \) to a specific \( W \), the system must ensure that in any \( sw_{i,c} \), \( p_j \) can see
at most \( W \) consecutive windows of \( p_i \).\(^8\) In \( sw_{i,c} \), \( p_j \) can see only windows \( w_{i,a} \), where

\(^6\)Receipt by \( P_j \) of two \( chng \) sent by \( P_i \) such that no \( chng \) sent by \( P_i \) is received between them.

\(^7\)That is, \( g_{m-1}.vt[i].wid=g_{m}.vt[i].wid \) and \( g_{m-1}.vt[i].vth=g_{m}.vt[i].vth \).

\(^8\)Initialized to 0, \( vc_i[i].wid \) is incremented by 1 in \textit{modulo} \( W \) for each new window.
max(0, c−W+1) ≤ a ≤ c (i.e., last W windows including wi,c), if

- (C1) all messages sent in wi,a and swi,a, 0 ≤ a ≤ c − W, are received before swi,c;
- (C2) all messages sent in wi,a and swi,a, a > c, are received after swi,c;
- (C3) P_i and P_j update their clocks ° to reflect timestamps from wi,c+1 when they transit from wi,c to wi,c+1 and swi,c to swi,c+1, respectively.

Figure 11 identifies p_i’s windows and p_i’s shadow windows on p_j that satisfy C1 and C2, when W=2. In swi,c, p_j cannot see windows in regions C1 and C2.

°By incrementing va[i].wid and vej[i].wid in modulo W.
Figure 11: An example showing $p_i$'s windows and $p_i$'s shadow windows on $p_j$ and $p_k$ satisfying conditions C1 and C2, when $W=2$.

What can be compared?

Note that since vector timestamps for events are no longer unique\textsuperscript{10} and monotonically increasing, timestamps of any two arbitrary events cannot be meaningfully compared. We say that the timestamp of an event $e_i$ can be compared with the timestamp of event $e_j$ iff $e_j \xrightarrow{\text{lam}} e_i$ can be inferred from some specified relation between their corresponding timestamps.

For an event $e_i$ in $w_{i,c}$, we now identify the events whose timestamps can be compared to the timestamp of $e_i$.

1. The timestamp of $e_i$ in $w_{i,c}$ can be compared with timestamp of other events in $w_{i,c}$ and $sw_{i,c}^j$ by using a relation $\leq$ defined next. This relation captures the $\xrightarrow{\text{lam}}$ relation between such events, i.e., if $e_i$ and $e_j$ are events in $w_{i,c}$ and $sw_{i,c}^j$, respectively,

\textsuperscript{10}Because both $vib$ and $wid$ are bounded.
with timestamps $vt$ and $vt''$, respectively, then $e_i \xrightarrow{\text{lam}} e_j$ iff $vt[i] \leq vt''[i]$.

**Definition 4.3.4 [Relation $\preceq$ on vector timestamp components]**

Given timestamp components $vt[i]$ and $vt''[i]$ of any two events in $w_{i,c}$ and $sw_{i,c}(\forall j)$, define $vt[i] \preceq vt''[i]$ iff $vt[i].vth \leq vt''[i].vth$.

(2) The timestamp of $e_i$ can be compared with the timestamp of events in the last $W$ windows and shadow windows, i.e., with timestamps of events in $sw_{i,a}$ or $w_{i,a}$, $\max(0, c - W + 1) \leq a < c$, using the relations $\prec$ and $\preceq$ defined earlier.

Observe that $vcj[i].wid$ for shadow window $sw_{i,c}$ has the same value as $vc[i].wid$ for window $w_{i,c}$. If $p_j$ receives a message $g$ in $sw_{i,c}$ such that $g.vt[i].wid \neq vcj[i].wid$, then $g$ must have been sent in $sw_{i,a}$ or $w_{i,a}$, $\max(0, c - W + 1) \leq a < c$, i.e., $g$ must have been sent in the last $W-1$ (shadow) windows, excluding the current (shadow) window.

(3) If $e_i$ is a receive event $grecv(g)$ in $w_{i,c}$ and $e_j$ is another receive event $grecv(g')$ in $sw_{i,c}$ (or $w_{i,c}$ if $j = i$), then the timestamp of the corresponding sends, i.e., the timestamps of $gsend(g)$ and $gsend(g')$ can also be compared using the relation $\prec'$ described next. This relation captures the $\xrightarrow{\text{lam}}$ relation between send events for any two messages received in the same (shadow) window.

**Definition 4.3.5 [Relation $\prec'$ on vector timestamp components]**

Given vector timestamp components $vt[i]$ and $vt''[i]$ of any two messages received in the same shadow window at $p_j$, define $vt[i] \prec' vt''[i]$ if

(1) $((vt[i].wid = vt''[i].wid) \text{ and } (vt[i].vth < vt''[i].vth)) \text{ OR }$

(2) $\exists a, b : (0 \leq a, b \leq W-1) \land ((d+a) \mod W = vt[i].wid) \land ((d+b) \mod W = vt''[i].wid) : a < b$. Where $d$ is the value of $vcj[i].wid$ in the shadow window in which the two messages are received.
Figure 12: Assume $W = 3$. A simple (arithmetic) comparison of two clock values does not capture the relation $\prec'$ between events on different shadow windows. Events in windows $w_{i,0}$, $w_{i,1}$, and $w_{i,3}$ have $vc_i[w].wid$ value of 0, 1, and 0, respectively. Note that events in $w_{i,0}$ have $\rightarrow_lam$ relation with events in $w_{i,1}$ and events in $w_{i,3}$ do not have $\rightarrow_lam$ with $w_{i,1}$.

The second condition in the previous definition infers $\prec'$ between events in different shadow windows. Since $vc_i[w].wid$ is a modulo $W$ number, a simple (i.e., arithmetic) comparison of two clock values will not give the correct answer. E.g., (refer to Figure 12). In current window, $w_{i,3}$, $vc_i[w].wid = 0$. $w_{i,3}$ can see only windows $w_{i,1}$, $w_{i,2}$, and $w_{i,3}$. Events in $w_{i,1}$ have $vc_i[w].wid = 1$. Even though, $0 < 1$, events in $w_{i,3}$ do not have $\rightarrow_lam$ relation with events in $w_{i,1}$. This anomaly is due to the fact that $vc_i[w].wid$ is incremented in a modulo fashion. The $\rightarrow_lam$ relation can be correctly deduced from the $<$ relation between the distances of the windows from the current window. In the second condition (in the definition 4.3.5), $a$ and $b$ represent the distance.

In general, $\preceq$ and $\prec'$ can be used by a process to deduce the $\rightarrow_lam$ relation between any two events that occur in the same window and shadow window and also between
events in all the windows that a process can see in the current (shadow) window.

### 4.3.2 An Implementation

The GS primitive is used to bound the clocks in the following way: Conditions C1 and C2 in Section 4.3.1 suggest using a \( gt \) for a \( chng \) message (we denote this by \( gt-chng \)). Additionally, to ensure C1, each \( P_j \) \((j \neq i)\) sends a special message \( gf-ack \) (this is \( gf \)) to \( P_i \) in response to \( gt-chng \). \( P_i \) sends \( gt-chng_c \) (the subscript \( c \) is the window number of the window at the end of which \( gt-chng \) is sent) only after it has received \( N-1 \) \( gf-ack_a^c, a = c - W + 1 \), one from each \( P_j \). This ensures that all messages sent in \( sw^i_{a,\omega} \), \( a = c - W + 1 \), are received before \( gt-chng \) is received, i.e., before any message from \( sw^i_{c+1} \) is received.\(^\text{11}\)

Figure 13 gives algorithm BVC executed at \( P_i \) to bound vector clocks for some \( B > 0 \) and \( W > 1 \). In BVC \( P_i \) sends a \( gt-chng \) to every other site when it transits to the next window and each site responds by sending a \( gf-ack \) message to \( P_i \) to acknowledge the transition. Each \( P_i \) maintains a vector, \( nack_i \), of \( N-1 \) “modulo \( W \)” counters, one for each \( P_j, j \neq i \), to count the number of \( gf-ack \) received from \( P_j \). Before proving the correctness of algorithm BVC, we briefly explain why each \( nack_i[j] \) need only be maintained as a modulo \( W \) counter.

Consider the \( gf-acks \) sent by \( P_j \) to \( P_i \). Observe that \( \forall c : c > 0 : gf-ack_{c-1} \in P(gf-ack_c) \), i.e., all \( gf-acks \) sent by \( P_j \) to \( P_i \) are received in the order sent. Thus, the shadow window number in which a \( gf-ack \) is sent can be deduced from the order of receipt of the \( gf-ack \).

When \( P_i \) is in \( w_{i,c} \), \( P_i \) has sent \( c \) \( gt-chng \) messages (notice that a window number

---

\(^{11}\)Due to properties \( Gf \) and \( Gt \) and the fact that all messages sent in \( sw^i_{a,\omega} \), \( a = c - W + 1 \), are in \( P(gf-ack_a) \) which is in \( P(gt-chng_a) \).
01: $P_i$: *[if OkToChange OR $P_i$ is ready to receive gt-chng or $gf$-ack then
02:     /*System events*/
03:     [OkToChange -> [send gt-chng /*change window*/
04:       vc$_i$[i] :=$((vc$_i$[i].wid+1)mod W, 0)
05:     ]
06:     ]
07:     $P_i$ is ready to receive gt-chng ->
08:     [ receive gt-chng /*change shadow window*/
09:       gf-ack.dst[gt-chng.se]:=1
10:       $\forall j: j \neq gt-chng.se:: gf-ack.dst[j]:=0$
11:       send $gf$-ack
12:       vc$_i$[gt-chng.se]:=$((vc$_i$[gt-chng.se].wid+1)mod W, 0)
13:     ]
14:     ]
15:     $P_i$ is ready to receive $gf$-ack ->
16:     [ receive $gf$-ack
17:       nack[$gf$-ack.se]:=(nack[$gf$-ack.se]+1)mod W
18:     ]
19:     ]
20:     else if $p_i$ is ready to send g OR receive g OR execute Int then /*$p_i$' events*/
21:     [ $p_i$ is ready to send g -> [ g.vt:= vc$_i$
22:       send g to g.dst
23:       vc$_i$[i] :=(vc$_i$[i]+(0, 1)
24:     ]
25:     ]
26:     $p_i$ is ready to receive g ->
27:     [receive g
28:       $\forall j: j \neq i:: vc$_i$[j] := max(vc$_i$[j], g.vt[j])
29:       vc$_i$[i] :=vc$_i$[i]+(0, 1)
30:       return g to $p_i$
31:     ]
32:     ]
33:     $p_i$ is ready to execute Int -> execute Int
34:     ]
35:     else skip /*null event*/
36:     ]
37: Always OkToChange= (vc$_i$[i].vtb $\geq B$) $\land$ ($\forall j: j \neq i::$nack$_i$[j] $\neq (vc$_i$[i].wid+1)mod W)
38: Initially $\forall j:: vc$_i$[j] := (0, 0), $\forall j: j \neq i::$nack$_i$[j] :=0, $nack$[i] not used.
39: $\forall j: j \neq i::$gt-chng.dst[j]=1, gt-chng.dst[i]=0

Figure 13: Algorithm BVC to bound vector clocks. ($B > 0$, $W > 1$). $vc_i$[i] :=vc$_i$[i]+(0, 1) is a short hand for vc$_i$[i].wid := vc$_i$[i].wid+0, vc$_i$[i].vtb := vc$_i$[i].vtb+1.
starts at 0). Also, $P_i$ has received at least $c - W + 1$ and at most $c$ $gf$-ack from each $P_j$. This follows from the facts that $P_i$ changes to $w_{i,c}$ only when it has received $N - 1$ $gf$-ack, $a = c - W + 1$, and all $gf$-acks are received in the order sent. The upper limit is due to the fact that a $gf$-ack is sent only in response to a $gt$-chng and so far only $c$ $gt$-chng have been sent. Thus, if in window $w_{i,c}$, $\forall j \neq i$: $nack[j] \neq (vc[i].wid+1) mod W$, then $P_i$ has received at least $c - W + 2$ $gf$-ack from each $P_j$, i.e., it can send $gt$-chng. This is represented by boolean variable $OkToChange$ (line 37) in Algorithm BVC. Hence, it is sufficient to maintain each $nack[j]$ as a modulo $W$ counter.

Figure 14: An execution illustrating that $P_j$ can see only windows $w_{i,c-1}$ and $w_{i,c}$ in $sw_{i,c}^j$ (Assume $W=2$). No message sent in the dotted region of the process time lines is received in $sw_{i,c}^j$. 
Bound on $vc_i[i].vtb$

In a window $w_{i,c}$, $P_i$ can see only the last $W$ windows and shadow windows. Each message $g$ sent in a window $w_{i,c}$ will be received in the shadow windows $sw_{i,c}^j$ because $gt-chng$ messages satisfy Property $Gt(b)$. Thus, in $w_{i,c}$ $P_i$ can receive from itself only the messages sent in $w_{i,c}$.

Let $d$ be the bound on the message delays and let $r$ be the maximum rate at which a processor can send messages. Let $x_s$ and $x_r$ denote the number of messages sent and received in a window.

In the worst case in $w_{i,c}$

$$\begin{align*}
x_r &= \text{Number of messages sent in } w_{i,c}\ (x_s) + \\
&\quad \text{Number of messages sent in the last } W \text{ shadow windows on all } P_j's \\
&= x_s + x_s \times W \times (N - 1) \\
\end{align*}$$

(1)

Also from the algorithm BVC

$$\begin{align*}
x_s &= B + d \times r \\
\end{align*}$$

(2)

Thus bound on $vc_i[i].vtb =$

$$\begin{align*}
&= x_s + x_r \\
&x_s + x_s \times W \times (N - 1) + x_s \\
&= (W \times (N - 1) + 2)(B + d \times r) \\
\end{align*}$$

Thus, in a system with large delays and large rate of message sends per process (i.e., $B << d \times r$), a smaller value of $W$ must be chosen to lower the bound on $vc_i[i].vtb$.

### 4.3.3 Correctness

**Lemma 4.3.1** In BVC, $p_j$ can see only windows $w_{i,a}$, $\max(c - W + 1, 0) \leq a \leq c$, in $sw_{i,c}^j$. 

Proof: The proof follows from the following facts (refer to Figure 14):

1. (\forall k) Each message sent in \( w_{i,a} \) or \( sw^k_{i,a} \), \( a > c \), is in the future of \( gt-chn_{c} \) and hence will be received by \( p_j \) after \( gt-chn_{c} \) is received by \( P_j \), i.e., after \( sw_{i,c}^j \). (Property \( Gt(a) \)).

2. Each message sent in \( w_{i,a} \), \( 0 \leq a \leq c - W \), is in the past of the \( gt-chn_{c-w} \) and hence is received by \( p_j \) before \( gt-chn_{c-w} \) is received by \( P_j \), i.e., before \( sw_{i,c}^j \). (Property \( Gt(b) \)).

3. (\forall k) Each \( g \) sent in \( sw^k_{i,a} \), \( 0 \leq a \leq c - W \), is in the past of \( gf-ack_{c-w} \) sent by \( P_k \) and hence is received by \( p_j \) before \( gt-chn_{c-w} \) is received by \( P_j \), i.e., before \( sw_{i,c}^j \). (Properties \( Gf \) and \( Gt(b) \)).

4. When \( P_j, J \neq i \), receives \( gt-chn \) message, \( a = c - W \), it updates \( vc_i[i] \) to reflect timestamps from \( w_{i,a+1} \). (Line 06 of BVC).

5. After sending \( gt-chn\), \( a = c - W \), \( P_i \) updates \( vc_i[i] \) to reflect timestamps from \( w_{i,a+1} \). (Line 03 of BVC).

Lemma 4.3.2 Algorithm BVC bounds \( vc_i[i] \).

Proof: From Lemma 4.3.1, in any shadow window \( sw_{i,c}^j \), \( p_j \) can see at most the last \( W \) windows of \( p_i \). These \( W \) windows can always be uniquely identified by assigning a distinct number in the range \( 0 \ldots W - 1 \). Thus, \( vc_i[i].wid \) needs to take values only in the range \( 0 \ldots W - 1 \). \( vc_i[i].vrb \) is already bounded to \( B \). Thus, \( vc_i[i] \) is bounded. □
4.3.4 Relaxing the Bounded Message Delay Constraint

If the assumption of bounded message delay is removed, then in Algorithm BVC an upper bound on \(v_{ci}[i].vtb\) cannot be determined. In this section we remove the assumption of bounded message delay and modify BVC to bound \(v_{ci}[i].vtb\) to a specified value. We do so by inhibiting message sends by other processes under some conditions.

In vector clocks, \(v_{ci}[i].vtb\) is incremented both on message sends and message receipts. Thus to bound \(v_{ci}[i].vtb\), message sends by a process will have to be inhibited under some conditions. Specifically, \(p_j\) inhibits message sends to \(p_i\) if it has sent \(B\) messages to \(p_i\) in \(p_i\)'s current shadow window until it receives \(gt-chng\) message from \(p_i\). Similarly \(p_i\) inhibits message sends in \(w_i,c\) if \(P_i\) has not yet received \(N-1 gf-ack\) in window \(w_i,e\), where \(a = c - W + 1\), messages and it has already sent \(B\) messages.

Each \(P_i\) maintains a vector of \(N\) counters denoted by \(ns_i\). \(ns_i[j]\), \(j \neq i\), denotes the number of messages sent by \(p_i\) to \(p_j\) in \(p_j\)'s current shadow window on \(p_i\). \(ns_i[i]\) denotes the number of messages sent by \(p_i\) in the current window. \(ns_i[j]\), \(j \neq i\), initialized to 0, is incremented on every message sent by \(p_i\) and is reset to zero on receipt of a \(gt-chng\) from \(p_j\). \(ns_i[i]\) is reset to zero on sending a \(gt-chng\).

An algorithm to bound vector clocks can thus be derived from algorithm BVC by adding the following lines:
and by changing the guard on line 21 to:

\[ \pi_i \text{ is ready to send } g \land (\forall j::ns_i[j]<B) \rightarrow [g.vt:=v_{ci}] \]

The additional condition on line 21 causes inhibition of message sends for such clocks. In this case, \(v_{ci}[i].vtb\) is bound to \(B(W(N-1)+2)\). \(^{12}\)

### 4.3.5 Applications and Related Work

**Applications**

**Snapshots:** Bounded clocks can be used to design efficient global snapshot recording algorithms. Lai and Yang [43] propose an algorithm to record a global snapshot. This algorithm requires each process to record all messages sent and received along each non-FIFO channel. Recording a complete message history might require a large amount of space.

Bounded clocks can be used to bound the message history kept at a process. According to Lemma 4.3.1, in \(sw_{i,c}^d\) all message sent in \(w_{i,a}\) and \(sw_{i,a}^k\), \(0 \leq a \leq c - W\), have already been received and no message sent in \(w_{i,a}\) and \(sw_{i,a}^k\), \(a > c\), will be received in \(sw_{i,c}^d\). Additionally, a message \(g\) sent by \(p_i\) in \(w_{i,c}\) to all \(p_j\) to take a global snapshot will be received by each \(p_j\) in \(sw_{i,c}^d\) because \(g \in \mathcal{P}(gt-chng_{c})\) and \(g \in \mathcal{F}(gt-chng_{c-1})\). Thus, to take a snapshot in \(sw_{i,c}^d\), each process needs to maintain message logs for the last \(W\) (shadow) windows only. If \(vtb\) bound by \(B\), then the space requirement to keep message history for a channel is \(O(B \times W)\).

\(^{12}\)\(B(W(N-1)+2)=2B\) (due to sends and receives in \(w_{i,a}\)) + \(B(W(N-1))\) (due to receipt of B messages each from the last \(W\) (including current) shadow window, one for each process.).
Potential Causality and Trace-Based Debugging: Relation $\rightarrow$ plays key role in determining causality among distributed events and consequently on trace-based debugging in distributed systems. The timestamps assigned by algorithm BVC to an event $e_i$ in $w_{i,c}$ can only be compared with timestamps of events in $w_{i,c}$ and $sw^j_{i,c}$ to infer $\rightarrow$, i.e., the timestamps generated precisely capture the relation $\rightarrow$ between events in the same (shadow) window. Note that an event $e_i$ in $w_{i,c}$ has $\rightarrow$ relation with all events in all subsequent windows and their shadow windows and events in all preceding windows and their shadow windows have $\rightarrow$ relation with $e_i$. An interesting application of bounded clocks is trace-based debugging of distributed programs where the notion of locality corresponds to the concept of windows. Locality suggests that sometimes after the occurrence of an event $e_i$, $e_i$ has a causal relation with each successive event in the system. By choosing an appropriate value of $B$ locality can be captured in a window.

Related Work

In [45], Lloyd and Kearns identify a class of algorithms in which unbounded counters can be replaced by bounded counters without compromising the correctness of the algorithms. The value of these counters is used for identification purposes only and not for deducing the temporal ordering of arbitrary events.

Ricart and Agarwala proposed a mutual exclusion algorithm [57] that involves use of messages with sequence numbers. A process requests entry into the critical section by sending a request to all other processes, setting the request’s sequence number to one higher than the highest sequence number known to the requester so far. It then waits until it has received a reply from every other process. A process receiving the
request responds to it in the following way: If its request has a lower sequence number than the received request or it is currently in the critical section, the reply is delayed until it has accessed the critical section. Otherwise, it sends a reply immediately. It has been shown that the sequence numbers can be bounded to $2N - 1$, where $N$ is the number of processes. This is true because at any time there can be at most $N$ outstanding requests, one from each process.

Sliding window protocols [65] use sequence numbers to detect loss and duplication of data frames in computer networks. The sequence numbers are implemented as a modulo counters. The sender appends to each data frame the value of its sequence counter. The receiver acknowledge either each data frame received or a block of data frames received. The sender, upon receipt of the acknowledgment, can reuse the sequence numbers associated with the data frames already acknowledged. Once the sender has used all available sequence numbers such that none of them have been acknowledged, the sender blocks until it receives an acknowledgment. In such protocols the sequence numbers are used to merely identify the data frames. The protocols do not deduce "causality" between events on sender and receiver.

### 4.4 Shared Token

We implement a shared token which supports operations to request it, to pass it, and to determine its current holder. A process wanting the token sends a request to other processes. The token is transferred by the current holder to the requesting process using a pass operation. If no process requests the token, the current holder of the token keeps the token until some process requests it.

Each process maintains a queue of outstanding request in the order of arrival, i.e.,
those requests for which a corresponding pass has not yet been issued. The process wanting the token sends the request to all processes, including itself, in the system. On receipt of a request, the requesting process is added to the queue. The holder of the token sends a pass message, if its queue is not empty, to give the token to the process at the head of the queue. The pass message, which contains the identity id of the new holder, is sent to all processes in the system, including the sender. On receipt of a pass message the first entry for the process whose identity is contained in the pass message, is removed from the queue. Note that this entry may not be at the head of the queue. All processes must receive a pass message, to pass the message to some process pj, only after it has received the associated request from pj. Note that the pass is issued by pi only after pi has received a request from pj. Thus, if a pass message is a gf, all request and their associated passes are received in order. Additionally, all requests from a process must be received in the order they were sent. Hence, all requests must be a gf. Note that for both pass and request, g, ∀j: pj ∈ g.dst. Figure 15 gives a pseudo-code for this implementation.

In the above implementation, either both pass and request messages need to be gf or both need to be gb. Henceforth, we will assume that both pass and request messages are gb.

4.5 Updating Replicated Data with Mutual Exclusion

Consider a system where some shared data is completely replicated at each processes site and a process must access the data in mutually exclusive manner. Each process wishing to access the replicated data first requests the shared token using the request operation, as described in Section 4.4. Upon acquiring the token, i.e., upon
01: * \[ p_i \text{ wants the token} \rightarrow GS(GF, \text{token.dst, 'REQUEST'}) \]
02: \[
03: p_i \text{ receives a REQUEST} \rightarrow insert(Q, gf.se)
04: \]
05: \[ p_i \text{ wants to pass the token} \land \text{have.token} \land \neg empty(Q) \rightarrow
\]
06: \[ GS(GF, \text{token.dst}, ('PASS'\text{,top(Q)}))
07: \]
08: \[ \text{have.token} := \text{False}
\]
09: \[ p_i \text{ receives ('PASS',id)} \rightarrow \]
10: \[ \text{Curr.Holder} := \text{remove.first}(Q, id)
11: \]
12: \[ \text{Curr.Holder} = i \rightarrow \text{have.token} := \text{True}
13: \]
14: \]
15: \[ \text{true} \rightarrow \text{skip}
16: \]
17: \]
18: \[
\]
Initially, \[ \text{Curr.Holder} := 0, \forall j::\text{token.dst}[j] := 1, \]
\[ \text{have.token} := (\text{Curr.Holder} = i), Q = \{\}. \]

Figure 15: An implementation of shared token using the \textit{GS} primitive. The function \textit{insert}(Q, gf.se) inserts \textit{gf.se} in the queue \textit{Q}. The function \textit{empty}(Q) returns true iff \textit{Q} is empty. The function \textit{top}(Q) returns the value at the head of the \textit{Q} without removing it. The function \textit{remove.first}(Q, id) removes the first entry in the \textit{Q} with the value \textit{id}.
receiving the pass message corresponding to the request, the process reads and writes the replicated data as if it were reading and writing a private read-write data. After finishing its accesses the process passes the token to another process if such a process has requested the token, using the pass operation.

A request is made to gain access to the shared token. Each process must receive the corresponding pass message only after it has received the request. Thus a request is gb.

pi sends a pass to (1) give access to the shared token, and (2) ensure that all accesses, by pi, to the replicated data are received by a process before the pass is received. Therefore, a pass message must be gf.

To maintain consistency, all replicas should be updated in the same order. Additionally, a read access should return the value written by the last write. Consecutive read accesses may be performed in any order. Thus, a write access must be gt and a read access may be go.

4.6 Summary

In this chapter, we described several applications to illustrate the usefulness of the Global Flush messages in the design and development of distributed algorithms. Specifically, we considered its applications in the following areas: getting consistent cuts, snapshots, and global snapshots; termination detection; bounding logical clocks; implementing shared token; and updating replicated data with mutual exclusion. We discussed its application in the areas of termination detection and bounding logical clocks in detail.

13Since updates are done by one process only, the “last update” is precisely defined.
We proposed a new termination detection algorithm which does not assume FIFO channels between processes. This algorithm is based on global flush messages. The beauty of this algorithm lies in its simplicity. It is based on identifying a cut at which all processes in the basic computation are idle and then detecting whether there were any messages in transit at that cut. We also discussed optimizations to this basic algorithm. Due to the nature of the global flush messages, this algorithm does not suffer from the problem faced by other marker based algorithms, i.e., the number of control messages used, due to unsuccessful attempts at detection of termination, is not unbounded in the worst case. The algorithm ensure that in every phase, at least one basic message becomes receivable. Thus the number of control messages in the worst case is bounded by $O(M \times N)$, where $M$ is the number of basic messages exchanged after the detection algorithm has started and $N$ is the number of processes in the basic computation.

We defined the notion of bounded clocks and propose an algorithm to bound vector clocks. The algorithm assumes bounded message delay. In a system with bounded clocks, many messages may have the same timestamp value. The proposed algorithm deals with this ambiguity by dividing the process time line into windows and shadow windows and by ensuring that the timestamp of a message received in a shadow window is unique with respect to other messages received within the same shadow window. This is achieved by limiting the set of sender’s windows a process can see in the current shadow window to the last $W$ windows. *Global Flush* messages are used to implement this ordering. We also extended the algorithm to bound the vector clocks when the assumption of bounded message delay is removed. Use of bounded clocks simplifies algorithm development in a distributed environment by bounding
the logs and other information kept by a process. Bounded clocks result in saving of space in messages by limiting the size of the timestamp carried by each message. Such bounded clocks also help in trace-based debugging of distributed programs, checkpointing, controlling logs, and predicting average maximal parallelism.
CHAPTER V
Hierarchical Clocks

5.1 Motivation

Logical clocks have been developed [22, 44, 47] to assign timestamps to events such that causality between events can be inferred from their timestamps. Applications of such clocks are widely known. Lamport [44] defines the partial temporal order “happened before” and presents timestamping mechanism for inferring potential causality. In this mechanism, ties can be broken arbitrarily (by using unique process i.d., for example) to form a total ordering. Neiger and Toueg in [51] suggest a modified timestamping mechanism to infer potential causality. Mattern [47] and Fidge [22] independently suggest a vector clock with which more concurrency can be identified than with previously suggested clocks. All these clocks, however, assume that no two events on a process can be concurrent even if sufficient number of processors were available, i.e., a process is a total order of events. ¹

In light of development of abstractions like threads [6, 66], a process can no longer be viewed as a sequence of events but must be viewed as a partially ordered set of events. A process may consist of multiple threads of flow control. When such a process is implemented in a multiprocessors environment (e.g., Solaris operating system

¹Two events which are not required by the computation to be executed in a specific order are referred to as concurrent events.
of Sun Microsystems), events on two different threads may actually be executed simultaneously on different processors. Note that each thread could itself be viewed as a "process". In this case however, the size of data structures in algorithms which order events on different processes (e.g., algorithms using vector clocks [22, 47]) becomes very large. We, thus, argue that it is more natural to view a process as a partially ordered set of events. This view of a process can also be used to model multiple processes of an application which execute at the same site and communicate via shared memory. By viewing a process as a partially ordered set of events, all processes at a site can be viewed as one big process at the site. Apart from providing a better representation of the system, this model simplifies the data structures kept at each site. A similar approach is taken in [58].

Viewing a process as a partially ordered set of events results in identification of all interprocess and intraprocess concurrency. It also presents a unified framework to reason about both sequential and parallel/distributed computation. The partial order between events in a distributed computation is described by a relation "affects" [4] (also referred to as "causality" relation). We denote the "affects" relation by $\sim$. If $e_1$ and $e_2$ are events such that $e_1 \sim e_2$, then $e_1$ and $e_2$ cannot happen simultaneously, even if sufficient number of processes were available. Two events are concurrent iff they do not have $\sim$ relation between them. Vector clocks cannot capture the $\sim$ relation. Bit-matrix clocks [4] were proposed to assign bit-matrix timestamps to events in a distributed computation. To recognize a $\sim$ relationship between two events, bit-matrix timestamps for the events must be compared to each other. Inferring $\sim$ relationship has applications in debugging of concurrent or sequential programs and in identifying events which do not affect each other, i.e., are potentially concurrent.
For debugging of distributed programs, given an event which indicates an error, a set of events can be determined which caused this event, from the causality relationship between it and and other events that happened before it. Clearly, a subset of these events must have caused the error. To determine causality between two events their timestamps can be compared. For debugging purposes, the smaller the set of events to be looked into, the easier it is to find the event that caused the error. By capturing $\leadsto$, bit-matrix clocks aid in reducing the set of events that need to be considered while debugging because $\leadsto$ is a subset of $\text{lan}$. If two events have $\leadsto$ relation between them, then they also have $\text{lan}$ relation between them, but the converse need not be true.

Bit-matrix clocks, however, have high cost in terms of both storage and communication overheads: A timestamp assigned to an event is a vector of $N$ components, one for each process in the system. Each component is a vector of bits. The $i^{th}$ component of the timestamps contains one bit for each event on the process $i$. The $i^{th}$ component for $X^{th}$ event on process $i$ is a binary string of length $X$. Since each message carries the value of the clock at the time it is sent, the communication overhead due to large timestamps in messages is also very high and may not be acceptable in many applications.

**Objectives**

In this chapter, we propose hierarchical clocks to capture the relation $\leadsto$ among events in a system where each process is a partially ordered set of events. Hierarchical clocks are more efficient than bit-matrix clocks as they substantially reduce both the storage and communication overheads. A distributed computation is thus viewed as a set of
events and a hierarchical partial order among them. This hierarchical partial order consists of two partial orders: $\sim$ between events on the same process (due to data dependencies and control flow in the process) and $\preceq$ between events on different processes (due to message sends and receives). The transitive closure of the relation $\sim \cup \preceq$ is the same as the $\leadsto$. This hierarchical partial order precisely captures the causality relation between all the events in a distributed system and provides an alternative way to view the $\leadsto$ relation.

Hierarchical clocks provide the following advantages for applications such as debugging of distributed programs: First, since hierarchical clocks can precisely capture causality, the set of events which cause an event indicating an error is minimal. The concurrency identified is the same as that identified by bit-matrix clocks. Second, the storage overhead at each process due to timestamps is greatly reduced. Third, the communication overhead for each message in terms of size of the timestamp carried by it is greatly reduced. This is specially advantageous because in a typical debugging session, the timestamp of relatively few events are ever compared. Thus, additional overhead of maintaining timestamps of events not considered in a debugging session are kept as low as possible.

The rest of the chapter is organized as follows: In Section 5.2, we describe the system model assumed in this and the next chapter. In Section 5.3, we present hierarchical clocks and provide ordering criteria for their associated timestamps. In Section 5.4, we discuss application of these clocks in debugging distributed programs. In Section 5.5, we discuss related work. We briefly describe vector clocks [22, 47] and bit-matrix clocks [4], and compare hierarchical clocks with them. We also discuss techniques [24, 48, 61, 62] proposed in the past for reducing the communication
overhead in implementing vector clocks.

5.2 Computation Model

A process is modeled as a partially ordered set of events. An event on a process is either an internal event or an event of sending or receipt of a message. An internal event does not directly affect events on other processes. A send event is an event which results in a message to be sent along a channel. A receive event is an event which results in a message to be received from a channel. Only a message that has been sent can be received, i.e., for every receive event there is a corresponding send event. The causality between events in a distributed computation is represented hierarchically as two relations: \( \preceq \) and \( \succcurlyeq \). The relation \( \preceq \) captures the causality between events on the same process. We assume that each process knows the \( \preceq \) relation between its events. We also assume that the \( \preceq \) relation is minimal, i.e., the transitive reduction of \( \preceq \) is the relation \( \preceq \) itself. The relation \( \succcurlyeq \) captures causality between send and receive events due to message exchange. Thus \( e \succcurlyeq e' \) iff \( e \) is the event of sending a message and \( e' \) is the event of receipt of the same message. Clearly, the transitive closure of \( \preceq \cup \succcurlyeq \) is the \( \succeq \) relation itself.

Definition 5.2.1 [\( \succeq \), to be read as “affects”]

The relation \( \succeq \) on the set of events in the distributed system is the transitive closure of relation \( \preceq \), where relation \( \preceq \) is defined by the following two conditions: (1) If \( e \) and \( e' \) are events on the same process such that \( e \preceq e' \), then \( e \succeq e' \). (2) If \( e \) and \( e' \) are events such that \( e \succcurlyeq e' \), then \( e \succeq e' \).

Lamport’s “happened before” relation [44] describes a partial order between events in a distributed system.
Definition 5.2.2 \([\xrightarrow{\text{lam}}]\), to be read as "happened before"

The relation \(\xrightarrow{\text{lam}}\) on the set of events in a distributed system is the transitive closure of relation \(\rightarrow\), where relation \(\rightarrow\) is defined as follows: (1) If \(e\) and \(e'\) are events on the same process such that \(e\) occurs before \(e'\), then \(e \rightarrow e'\); (2) If \(e\) and \(e'\) are events such that \(e \xrightarrow{m} e'\), then \(e \rightarrow e'\).

![Diagram](image)

Figure 16: A sample distributed computation. The dots represent events on a process, which itself is represented by a horizontal dotted line. The solid directed lines represent the \(\xrightarrow{\sim}\) relation.

The difference between \(\xrightarrow{\sim}\) and \(\xrightarrow{\text{lam}}\) lies in the first condition of their definition, i.e., while \(\xrightarrow{\sim}\) states that the events on a process are partially ordered, \(\xrightarrow{\text{lam}}\) totally orders them. For ease of illustration, we identify a process by a single time line (see Figure 16). The \(\xrightarrow{\sim}\) relation between events (events are denoted by a dark dot) is identified by a solid directed line connecting them. Events which do not have \(\xrightarrow{\sim}\) relation between them, are arbitrarily ordered on the time line. Thus the (total) order in which events occur on the time line identifies the order due to \(\xrightarrow{\text{lam}}\) and denotes one
possible execution order of events on a process.

To each event on a process, we assign an integer event identifier for identification purposes only. The order between event identifiers for events on the same process is the same as that of "happened before" relation between them. The first event has an identifier 1, and the \( X^{th} \) event has an identifier \( X \), and so on. Let \( e_i \) denote the \( X^{th} \) event on process \( p_i \).

Let \( \mathcal{E} \) denote the set of all events in the distributed computation under consideration and let \( \mathcal{E}' \) denote an arbitrary subset of \( \mathcal{E} \).

**Definition 5.2.3 [Projection]**

Define \( [\mathcal{E}']_i \) as the subset of events in \( \mathcal{E}' \) which occur on process \( p_i \).

Messages exchanged between processes are denoted by \( g, g' \), etc. \( gsend(g) \) denotes the event of sending \( g \) and \( grecv(g) \) denotes the event of receipt of \( g \). For every \( grecv(g) \), there is a corresponding \( gsend(g) \) in the computation.

**Definition 5.2.4 [\( \mathcal{P}[e_i] \), to be read as "the past events of \( e_i \)"]**

Define \( \mathcal{P}[e_i] = \{ e_j : e_j \xrightarrow{\text{lam}} e_i \} \cup \{ e_i \} \), i.e., the past of \( e_i \) is the set of events containing \( e_i \) and all events which happened before \( e_i \) according to \( \xrightarrow{\text{lam}} \).

### 5.3 Hierarchical Clocks

The use of bit-matrix clocks to capture \( \rightsquigarrow \) relations in systems where a process is a partial order of events, as in [4], results in excessive communication and storage overheads. Instead, we propose that the causality relation in such systems should be captured hierarchically by a pair of clocks: local bit-vector clock and a global vector clock denoted by \( lc_i \) and \( gc_i \) at \( p_i \), respectively. The two clocks are collectively referred
to as hierarchical clocks. Hierarchical clocks assign two timestamps $\phi^i$ (due to $lc_i$) and $\phi^m$ (due to $gc_i$) to each event to capture the hierarchical partial order. $\phi^i$ for an event is used to capture the set of events on the same process that have $\prec_i$ relation with it. The timestamp $\phi^i$ is a variable length bit-vector containing one bit for every event on the process. This timestamp is not communicated between processes as a part of the message timestamp. $\phi^m$ is a vector of $N$ integers, one for each process in the system. The $k^{th}$ component of $\phi^m$ timestamp for an event uniquely identifies the last event on process $p_k$ which causally affects it. By storing and forwarding only the identity of the last event on a process which causally affects an event as opposed to the identity of all events on the process which causally affect the event, considerable savings in storage and communication is achieved. Each message carries only the $\phi^m$ timestamp of its send event. Consequently, the communication overhead due to message timestamp is greatly reduced. Causality between events on the same process can be deduced by comparing their $\phi^i$ timestamps. Causality between events on different processes can be deduced by reconstructing the bit-matrix timestamps, when needed (this is described later).

For an event $e_i$, the timestamp due to $lc_i$ will have $X$ bits and the $k^{th}$ component of $\phi^m(e_i)$ will be an integer with a value $\|\mathcal{P}[e_i]_k\|$. 

Since the proposed system of clocks requires that each message carry only the timestamp due to $gc_i$, the communication overhead to maintain hierarchical clocks is comparable to that of vector clocks. This overhead is much less than that for bit-matrix clocks. The storage overhead at each process is also reduced. For event $e_i$, $p_i$ needs to store $X+\sum_k \lceil \log(\|\mathcal{P}[e_i]_k\|) \rceil$ bits.

Each process $p_i$ generates the timestamp, due to $lc_i$, of $e_i$ denoted by $\phi^i(e_i)$, by
bit-wise ORing the timestamp due to Ici of all events e'_i such that e'_i \xrightarrow{i} e_i. Also, the X^{th} bit of the \( \phi^i(e_i) \) is set to 1 to indicate that it is the timestamp of \( e_i \).

**Definition 5.3.1 [\prec \text{ on } \phi^i(e_i)]**

For any two events \( e_i \) and \( e'_i \) on \( p_i \), define \( \phi^i(e'_i) \prec \phi^i(e_i) \) if \( \phi^i(e_i)[Y]=1 \), i.e., the \( Y^{th} \) bit of \( \phi^i(e_i) \) is 1.

**Observation 1** \( e'_i \xrightarrow{i} e_i \) iff \( e'_i \xrightarrow{i} e_i \) iff \( \phi^i(e'_i) < \phi^i(e_i) \).

The gc_i clock is a vector of \( N \) integers, one for each process in the system. Initially, \( \forall k \ gc_i[k]=0 \). The timestamp \( \phi^m(e_i) \) due to clock gc_i for an event \( e_i \) is computed as follows:

- For a gsend or an internal event \( e_i \), the \( k^{th} \), \( k \neq i \), component of the \( \phi^m(e_i) \) is the maximum of the \( k^{th} \) component of the \( \phi^m(e'_i) \) such that \( e'_i \xrightarrow{i} e_i \). The \( i^{th} \) component of \( \phi^m(e_i) \) is set to \( X \). For a gsend event \( e_i \), the message carries the value of \( \phi^m(e_i) \) as its timestamp.

- For a grecv event \( e_i \), the \( k^{th} \), \( k \neq i \), component of \( \phi^m(e_i) \) is the maximum of the \( k^{th} \) components of all \( \phi^m(e'_i) \) such that \( e'_i \xrightarrow{i} e_i \) and \( k^{th} \) component of the timestamp carried by the message. The \( i^{th} \) component of \( \phi^m(e_i) \) is set to \( X \).

Figure 17 shows \( \phi^i(e_i) \) and \( \phi^m(e_i) \) timestamps for each event \( e_i \) in the sample distributed computation in Figure 16. It is interesting to note that the vector timestamps are a special case of \( \phi^m(e_i) \), i.e., when the relation \( \xrightarrow{i} \) is a total order, the

\[ \phi^m(e_i)[i]=2^{X^{th}+1} - 1. \]

\[ \text{This can be optimized by requiring that a process maintain only } N-1 \text{ component, one for each process in the system except itself. This results in saving of space required for storage of a timestamp. Note that all } N \text{ components still need to be sent as a message timestamp. For an event } e_i, \text{ the } X^{th} \text{ bit is already set to } 1 \text{ and all bits } Y, Y > X \text{ are } 0 \text{ in } \phi^i(e_i). \text{ Thus,} \]

\[ \phi^m(e_i)[i]=2^{X^{th}+1} - 1. \]
timestamps assigned due to $g_c$ become the same as vector timestamps \([22, 47]\). Note that in this case, relation $\rightsquigarrow$ is the same as $\triangleleft$ relation.

![Diagram showing timestamps for events in a distributed computation](image)

**Figure 17**: $\phi^i(e_i)$ and $\phi^m(e_i)$ timestamps for each event $e_i$ in a distributed computation.

While timestamps $\phi^i(e_i)$ of events captures the causality between events on the same process, timestamps $\phi^m(e_i)$ of events do not alone completely capture the causality between events on different processes. Instead, $k^{th}$, $k \neq i$, component of the $\phi^m(e_i)$ timestamp identifies the last event on $p_k$ which has the causality relation with $e_i$, i.e., the $k^{th}$ component is an index to the last event on $p_k$ to causally affect $e_i$. This is analogous to storing an index to information instead of copying the information itself. Such an indirection saves storage space at the cost of traversal of the indirection chain to access the information when it is needed. Such a representation is quite useful in
applications where information is accessed less often.

**Observation 2** For event $e_i$, $\phi^m(e_i)[i] = X$, i.e., $i^{th}$ component of $\phi^m(e_i)$ stores the identity of the event $e_i$.

**Observation 3** If $e_i \leadsto e_j$, then $\phi^m(e_i)[j] = X$.

The previous two observations follow directly from the way the clock $gc_i$ is maintained.

**Definition 5.3.2** [Event-Chain]

Define $e_1, \ldots, e_n$ to be an event-chain starting at $e_1$ and ending at $e_n$ iff $\forall k: 1 \leq k < n: e_k \overset{i}{\rightarrow} e_{k+1} \lor e_k \overset{m}{\rightarrow} e_{k+1}$.³

Note that from definition of $\overset{i}{\rightarrow}$ and $\overset{m}{\rightarrow}$, $\lor$ is an exclusive OR.

**Lemma 5.3.1** For any two events $e_i$ and $e_j$, $e_i \leadsto e_j$ iff an event-chain starting at $e_i$ and ending at $e_j$ exists.

**Proof:** Follows from definition of $\leadsto$, $\rightarrow$, and $\Rightarrow$. □

**Observation 4** If $\phi^m(e_j)[i] = X$, then there exists an event-chain starting at $e_i$ and ending at $e_j$.

This observation follows from the definition of $gc_i$.

**Definition 5.3.3** [Closed Event Set]

Define $\mathcal{E}'$ as a closed event set iff for any two events $e_i$ and $e_j$ in $\mathcal{E}'$, all event-chains starting at $e_i$ and ending at $e_j$ are in $\mathcal{E}'$.

³The subscripts in this definition identify order between events in the sequence only. Processes on which these events occur are irrelevant and hence are omitted from the notation.
Past of any event, past of any set of concurrent events, and the set $E$ itself are examples of closed event sets.

**Lemma 5.3.2** If $E'$ is a closed event set, then the causality between any two events in $E'$ can be determined based solely on timestamps, due to $lc_i$ and $gc_i$, of events in $E'$.

**Proof:** We prove this by contradiction. Assume that two events $e_i$ and $e_j \in E'$ such that $e_i \rightarrow e_j$, and $e_i \rightarrow e_j$ is not captured by timestamps due to $lc_i$ and $gc_i$, of events in $E'$.

Since $e_i \rightarrow e_j$, there exists an event-chain $E'$ starting at $e_i$ and ending at $e_j$. There are three possible cases:

- **Case 1:** $e_i \rightarrow e_j$. Since this relation is not captured by the timestamps, \( \phi^i(e_j)[X] = 0 \). This contradicts Observation 1.

- **Case 2:** $e_i \rightarrow e_j$. Since this relation is not captured by the timestamps, \( \phi^m(e_j)[i] \neq X \). This contradicts Observation 4.

- **Case 3:** $e_i \rightarrow e_k \rightarrow e_j$. Either $e_i \rightarrow e_{k,Z}$ is not captured by the timestamps or $e_{k,Z} \rightarrow e_j$ is not captured by the timestamps. Using this reasoning inductively, we arrive at the contradictions as in Cases 1 and 2. □

Thus, the \( \rightarrow \) relation between events in a closed event set can be inferred from the hierarchical timestamps associated with the events in the set. Later, we use the result of this lemma to compute the bit-matrix timestamps of events in any closed event set.
5.4 Applications in Debugging of Distributed Programs

We discuss the use of hierarchical clocks in debugging of distributed programs. We consider debugging of a distributed program based on the trace of the program execution. A trace is generated such that events of interest are assigned the timestamps from hierarchical clocks. The approach we discuss here uses Balzer's flow back analysis [7] which attempts to help the programmer in locating bugs by showing the past flow of program execution and by showing the causal relation between events. However, the approach is general and can be used with debugging techniques.

Bugs in a program manifest themselves into some symptoms that are 'visible' to the user, e.g., incorrect balance after a deposit into an account or simply a system core dump. A symptom is caused by an erroneous internal state of the program due to some fault in the program. The programmer has some notion of 'correctness' of the program to which the program execution does not correspond. From the programmer's point of view, detecting a bug in the program requires looking at the last action which caused the effect to become visible. If this action is 'correct' (according to the semantics of the program known to the programmer), then the programmer proceeds to look at the past of this event. To aid in the debugging effort, the set of events that the programmer needs to consider should be kept as small as possible.

The strategy adopted here is to generate a trace of events using hierarchical clocks. This keeps the overhead in terms of communication and storage small when compared to using bit-matrix clocks. At debugging time, these clock values can be used to deduce the events that directly caused the event which made the symptom visible. Since the hierarchical timestamps of events contain enough information to precisely capture the causality relation $\rightarrow$, a bit-matrix timestamp can be associated with a set
of events of interest to reflect the \( \rightarrow \) relation among them. We do this in Section 5.5.3. At debugging-time this can be represented as a causality graph. Since this is done only with a subset of events and at debugging time, the run-time penalty of excess communication and storage overhead is avoided.

5.5 Related Work

We now briefly describe vector clocks [22, 47] and bit-matrix clocks [4], and compare hierarchical clocks with them. We also discuss some techniques [24, 48, 61, 62] proposed in the past for reducing the communication overhead in implementing vector clocks. These techniques can be applied to reduce the communication overhead in implementing \( gc_i \).

5.5.1 Vector Clocks

Vector clocks [22, 47] were proposed to assign vector timestamps to events to capture the \( \xrightarrow{\lambda m} \) relation between them. Each process maintains a vector clock of \( N \) integers, one for each process in the system. Each message carries with it the value of the sender's clock at the time it is sent as its timestamp. The vector timestamp of message \( g \) is denoted by \( \phi(g) \). The vector clock \( mc \) at \( p_i \) is maintained as follows:

- Initially, \( \forall k:: mc[k]=0 \).

- On \( gsend(g) \) event at \( p_i \), \( g \) is timestamped with the value of \( mc \) and the \( i^{th} \) component of \( mc \) is incremented, i.e., \( vc_i[i]:=vc_i[i]+1 \), after sending \( g \).

- On \( grecv(g) \) event at \( p_i \), after receiving \( g \),

\[ \forall k: k \neq i:: vc_i[k]:=\max(vc_i[k], \phi(g)[k]) \text{ and } vc_i[i]:=vc_i[i]+1. \]
On an internal event at \( p_i \), the \( i^{th} \) component of \( m_c \) is incremented.

The relation \( \prec \) on vector timestamps, defined next [47], describes a partial order on the events.

**Definition 5.5.1 [Relation \( \prec \) on vector timestamps]**

Given two vector timestamps \( \phi(e_i) \) and \( \phi(e_j) \), define \( \phi(e_i) \prec \phi(e_j) \) iff \( \forall l :: (\phi(e_i)[l] \leq \phi(e_j)[l]) \) and \( \exists k :: (\phi(e_i)[k] < \phi(e_j)[k]) \).

As shown in [22, 47], \( e_i \xrightarrow{\text{lam}} e_j \) iff \( \phi(e_i)[i] < \phi(e_j)[i] \); i.e., only one component of the vector timestamp needs to be compared to infer \( \text{lam} \).

![Diagram](image.png)

Figure 18: Vector timestamps associated with each event in a distributed computation.

Figure 18 shows vector timestamps assigned to events in the distributed computation shown in Figure 16. As shown in the figure, even though events with timestamps
and $|\frac{1}{2}|$ do not have $\preceq$ relation between them, their vector timestamps have $\prec$ relation between them. Thus, vector timestamps do not capture the $\sim$ relation. The $k^{th}$ component of $\phi(e_i)$ is an integer with value $|\mathcal{P}[e_i]|$.

### 5.5.2 Bit-matrix Clocks

Bit-matrix clocks [4] were proposed to assign timestamps to events to capture the $\sim$ relation between events. Each process maintains a vector clock of $N$ component, one for each process in the system, where each component itself is a variable size vector of bits. As with the vector clocks, each messages carries the value of the sender’s clock at the time it is sent. The bit-matrix clock $\text{bmc}_i$ at $p_i$ is maintained as follows:

- Initially, $\forall k:: \text{bmc}_i[k]=0$;

- For a $\text{gsend}$ or an internal event, $e_i$, the timestamp is computed as follows: The $k^{th}$ component ($\forall k$) of the timestamp is computed by bit-wise ORing the $k^{th}$ component of timestamps of all events $e'_i$, such that $e'_i \rightarrow e_i$. Also, the $X^{th}$ bit of the $i^{th}$ component of the timestamp is set to 1 to indicate that it is the timestamp of event $e_i$. For a $\text{gsend}$ event, the message carries this timestamp.

- For a $\text{grecv}$ event, $e_i$, the timestamp is computed as follows: The $k^{th}$ component of the timestamp is computed by bit-wise ORing the $k^{th}$ component of timestamps of all events $e'_i$ such that $e'_i \leftarrow e_i$ and the $k^{th}$ component of the timestamp carried by the message. Also, the $X^{th}$ bit of the $i^{th}$ component is set to 1, to indicate that it is the timestamp of event $e_i$.

Figure 19 shows bit-matrix timestamps assigned to events in the distributed computation shown in Figure 16.
Figure 19: Bit-matrix timestamps associated with events in a distributed computation.

The \( k \)th component of the bit-matrix timestamp of \( e_i \) is a bit-string of \(|\mathcal{P}[e_i]|_k\) bits, one bit for each event in \([\mathcal{P}[e_i]]_k\). For computations with large number of events, the number of bits required to represent a timestamp grows very rapidly and so does the communication overhead. In applications like debugging, where the timestamps of only a few events may ever be compared, this may not be acceptable. Apart from the communication costs, the storage costs involved in storing each timestamp is also large. For each event \( e_i \), \(|\mathcal{P}[e_i]|\) bits need to be stored at each process.

### 5.5.3 Comparison to Hierarchical Clocks

Table 5 compares the proposed hierarchical timestamping scheme with vector clocks and bit-matrix clocks based on storage cost and communication overhead for an event
e_i. The storage overhead due to hierarchical timestamps is slightly more than that of vector clocks, but is considerably less than bit-matrix clocks. The communication overhead for maintaining hierarchical timestamps is the same as that for vector timestamps, but is considerably less than that for bit-matrix timestamps.

Table 5: Comparison of hierarchical timestamps with vector timestamps and bit-matrix timestamps.

<table>
<thead>
<tr>
<th></th>
<th>Hierarchical</th>
<th>Vector</th>
<th>Bit-Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>lc_i</td>
<td>gc_i</td>
<td>mc</td>
</tr>
<tr>
<td>Storage</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Constructing bit-matrix timestamps: We now briefly describe how to compute the bit-matrix timestamp φ(e_i) of an event e_i in a closed event set E'. We assume the existence of a function trunc() which takes a timestamp and a closed event set E' as an argument and returns a timestamp from which entries for all events not in E' have been eliminated.

The i^th component of the bit-matrix timestamp of e_i can simply be computed by truncating φ^i(e_i), the timestamp due to lc_i:

\[ φ(e_i)[i] = \text{trunc}(φ^i(e_i), E') \]

The j^th, j ≠ i, component of the bit-matrix timestamp of e_i can be computed by the following recurrence relation.

\[ φ(e_i)[j] = \text{trunc}( (\forall k \neq i, φ^k(e_i)[k]) = Y \ (φ(e_k,Y)[j]) \lor (\forall φ^i(e_i,x) \lor \phi(e_i,z)[j]), E') \]
Assume $\forall k, j \phi(e_{k,0})[j] = 0$.

The first component $\phi_{k \neq \phi^m(e_i, x)}[k] = \phi(e_{k, y})[j]$ is the $j^{th}$ component of the bit-matrix timestamp of the events on other processes pointed to by the $k^{th}$ component of $\phi^m(e_i)$. The second component $\phi_{e_i, x}[z] = \phi(e_{i, z})[j]$ on the other hand, is the $j^{th}$ component of the bit-matrix timestamp of all events $e'_i$ on $p_i$ such that $e'_i \sim e_i$.

### 5.5.4 Efficiency Considerations for $\phi^m$

Recently, several techniques [24, 48, 61, 62] have been proposed to reduce the communication overhead when implementing vector clocks [22, 47]. These techniques can be used to efficiently implement $\phi^m$. Fowler et al. [24] reduce the communication overhead by requiring each message to carry only the event identifier of its send event instead of the vector timestamp of its send event. The storage overhead at each site is not reduced. The vector timestamps are later constructed based on these event identifiers. Schwarz and Mattern [61] improve the algorithm to compute the vector timestamps by observing that a process is a total order of events.

Singhal and Kshemkalyani [62] propose an efficient implementation of vector clocks. This implementation is based on the observation that in a typical distributed computation, many components of a vector clock do not change very frequently. Efficiency is achieved by sending only the components of the vector clocks and the associated process indices that have changed since the last time a message was sent. This implementation requires each process to maintain additional data structures and assumes that communication channels between processes is FIFO.

Meldel et al. [48] propose a scheme to reduce the communication and storage overheads in maintaining vector clocks by assuming that communication links be-
tween processes are static and are known apriori. This scheme restricts the set of events which can be compared to deduce $\textsf{lam}$ to events on a process and the send events of messages received on that process. In this scheme, a process only maintains those components of vector clock for processes with which it directly or indirectly communicates.

5.6 Summary

In this chapter, we viewed a distributed computation as a set of events with a hierarchical partial order on them. We represented this hierarchical partial order as two partial orders, $\sim$ and $\triangleright$. The $\sim$ is a partial order between events on the same process, whereas $\triangleright$ is the partial order between events on different processes due to message sends and receives. We proposed a timestamping scheme which assigns two timestamps $\phi^i(e_i)$ and $\phi^m(e_i)$ to each event $e_i$ in the system. Each message sent by an event $e_i$ carries only $\phi^m(e_i)$. We showed that the two timestamps assigned to events can be used to deduce the causality relation between the events in a distributed computation. The proposed timestamping scheme reduces the communication and storage overheads when compared to bit-matrix clocks while capturing the causality relation. We also showed how hierarchical clocks can be helpful in trace-debugging techniques of distributed programs.
CHAPTER VI

A Concurrency Measure

Performance of distributed computation have traditionally been characterized by measures based on number of messages exchanged, total information exchanged by messages, or the total execution time of the computation. Though important, these measures do not completely characterize a distributed computation. Recently, various measures characterizing the synchronization delays and concurrency in the computation have been proposed. These measure assume that a process is a sequence of events. Thus, irrespective of the number of processors available, any two events on a process cannot execute simultaneously.

In this chapter, we model a process as a partially ordered set of events. Thus, if more than one processor is available, then some events on a process may execute simultaneously. Viewing a process as a partial order results in identification of all interprocess and intraprocess concurrency in the computation and it presents a unified framework to model both sequential and parallel/distributed computation. It also provides a convenient framework to model processes with multiple threads of flow control and systems with multiple processes at each site by simplifying the data structures kept at each site. In this chapter, we propose concurrency measures which capture the concurrency perceived by an event and concurrency perceived by the entire computation, when a process is viewed as a partially ordered set of events.
Such measures can be used in exploiting the inherent concurrency in a computation by further distributing/consolidating the events in a distributed computation. Along with traditional measures like total execution time, these measures give a better characterization of a distributed computation.

6.1 Motivation

Distributed computations have been traditionally compared using measures such as the number of messages exchanged, total information exchanged by the messages, or the total execution time. Though useful, such measures do not completely characterize a distributed computation and hence are not a good basis for comparison of distributed computations. For example, the number of messages exchanged does not quantify how many of these messages were sent concurrently, and the total execution time of a computation does not quantify the time spent by a process waiting to receive a message.

An important and interesting characterization of a distributed computation is one which quantifies the amount of synchronization needed between events in the computation. Synchronization between events can be measured in terms of the set of events that must be executed before an event in the computation can be executed or the time spent by a process waiting for an event to execute (e.g., waiting to receive a message). Recently measures have been proposed \[18, 19, 56\] which attempt to characterize a distributed computation based on the above criteria. They, however, assume that a process is a total order of events \(^1\), i.e., two events on a process cannot be simultaneously executed even if sufficient number of processors were available.

\(^1\)Two events which are not required by the computation to be executed in a specific order are referred to as concurrent events.
Lamport's "happened before" relation \[^2\] identifies the order of execution of events. In these measures, two events are concurrent iff they do not "happen before" each other. Concurrency between events in a distributed computation, when a process is viewed as a sequence of events, is referred to as "explicit concurrency". These measures [18, 19, 56] attempt to quantify the "explicit concurrency" in a distributed computation.

Even though events in a process occur in some temporal order they need not be causally related [4]. If more number of processors were available to execute the distributed computation then some more events in the distributed computation might occur simultaneously. Causality relation between events is not a total order on events. The "affects relation" \(\rightsquigarrow\) [4] identifies the causality between events. We refer to the concurrency identified due to \(\rightsquigarrow\) as the inherent concurrency in the computation.

The measures proposed in [18, 19, 56] do not capture the inherent concurrency in the computation, i.e., they do not characterize the number of events which happened before an event on the same process that are actually concurrent to it. A measure which characterizes the inherent concurrency in a computation could aid in further distributing/consolidating the events in a distributed computation. Fidge [23] attempts to capture inherent concurrency, but does so only for a restricted version of \(\rightsquigarrow\).

In this chapter we propose a concurrency measure which characterizes the inherent concurrency as observed by an event in a distributed computation and by the distributed computation itself. Along with traditional measures like total execution time, these measures will give a better characterization of a distributed computation.

\[^2\]Defined later.
The model of distributed computation assumed in this chapter is the same as that in Chapter V.

To aid in the analysis, we assume that all events take one time unit to execute. To avoid the affect of system and network load and processor speed, we assume that message transfer is instantaneous. By appropriately adjusting the event granularity, these assumptions can be reasonably achieved. Similar assumptions have been made previously [23, 56].

Figure 20: A sample computation indicating synchronization delays. The solid directed lines indicate the relation between events.

Execution of an event at a process (e.g., $e_{2,1}$ on $p_2$ in Figure 20.) may be delayed and no events may be executed at the process during this period. This time spent is referred to as the synchronization delay [18, 56]. For example, a synchronization delay of 2 time units is introduced on $p_2$ before $e_{2,1}$ is executed. The synchronization
delay in a computation has been used [56, 18] to inversely measure the concurrency in a computation. Intuitively, it represents the time units that could have been used to execute concurrent events. We next describe a concurrency measure $\gamma$ for an event which measures concurrency observed by the event with respect to events that happened before it. $\gamma$ measures the number of events that are concurrent to it as a fraction of the maximum number of events that could be concurrent to it. Clearly, the higher the value of this measure, the more concurrency perceived by the event. Based on this measure, we later propose concurrency measures for the entire computation. Later in Section 6.2.2, based on this measure, we propose concurrency measures for the entire computation. Section 6.3 compares the proposed measure with other concurrency measures proposed in the literature [18, 19, 56, 23]. In Section 6.4, we describe how to compute the proposed measures.

6.2 Concurrency Measure for an Event

Before describing the proposed concurrency measure, we define three different metrics, mass, weight, and volume, associated with an event.

The mass of an event $e_i$ is the number of events that must execute before $e_i$ can be executed at $p_i$. The mass of an event $e_i$ represents the set of events that causally precede $e_i$ (according to $\prec$) and thus must execute (may be on a different processes) before $e_i$. That is,

$$\text{mass}(e_i) = |\{ e_j : e_j \prec e_i \}|$$

The weight of an event $e_i$ is the number of events that happened before $e_i$ (according to $\xrightarrow{\text{lam}}$) in the computation. Thus the weight of an event includes events that
happened before $e_i$ and are concurrent to $e_i$. That is,

$$wt(e_i) = |\{e_j : e_j \xrightarrow{lam} e_i \}|$$

Note that neither mass nor weight of an event counts the event itself. Also, since $\sim$ is a subset of $\xrightarrow{lam}$ (i.e., $e_i \sim e_j \implies e_i \xrightarrow{lam} e_j$ and $e_i \xrightarrow{lam} e_j \not\implies e_i \sim e_j$), $mass(e_i) \leq wt(e_i)$.

The *volume* of an event $e_i$ is a measure of the total time spent on each process before $e_i$ could be executed, i.e., it measures the maximum number of events that could be executed before $e_i$. Intuitively, the volume of an event is the sum of its weight and the total, over all processes, time units spent on synchronization delay. Clearly, $wt(e_i) \leq vol(e_i)$.

**Concurrency Measure:** For any event $e_i$, $wt(e_i) - mass(e_i)$ denotes the number of events that occurred before $e_i$ and are concurrent to $e_i$. The $vol(e_i)$ of an event denotes the maximum number of events that could have occurred before $e_i$ and be concurrent to $e_i$. The ratio of the two defines a concurrency measure $\gamma(e_i)$, to be read as concurrency perceived by $e_i$:

$$\gamma(e_i) = \frac{wt(e_i) - mass(e_i)}{vol(e_i)}$$

### 6.2.1 Rationale

The value of measure $\gamma(e_i)$ for any event $e_i$ lies between 0 and 1. The higher the value of this measure for an event, the more the perceived concurrency by the event. If for an event $e_i$, $wt(e_i) = mass(e_i)$, i.e., $e_i$ perceives that no event is concurrent to it, then $\gamma(e_i) = 0$. On the other hand, if $mass(e_i) = 0$ and $wt(e_i) = vol(e_i)$, then $\gamma(e_i) = 1$. 
Figure 21: Sample computations with events having extreme values of the measure. (a) $\gamma(e_{1,4})=0$, i.e., $e_{1,4}$ perceives no concurrency at all. (b) $\gamma(e_{1,4})=1$, i.e., $e_{1,4}$ perceives that all events are concurrent to it and no time is spent in synchronization delay.
i.e., an event $e_i$ perceives the highest concurrency only when all events are concurrent to it and no time units have been spent in synchronization delay. Figure 21 illustrates these two extreme cases. In Figure 21(a), $\gamma(e_{1,4})=0$, i.e., $e_{1,4}$ perceives that no event is concurrent to it. In Figure 21(b), $\gamma(e_{1,4})=1$, i.e., $e_{1,4}$ perceives that all events are concurrent to it and no time is spent in synchronization delay.

![Diagram](image-url)

Figure 22: Values of $\gamma$ for each event in the computation.

Figure 22 shows the value of $\gamma$ for events in the sample computation shown in Figure 20. (We take $\frac{0}{0}$ as 1.) No event happened before events $e_{1,1}$ and $e_{3,1}$ and hence the perceived concurrency for them is 1. For event $e_{1,2}$ the maximum possible number of events that could happen before it are concurrent to it ($\gamma(e_{1,2})=1$), i.e., $\text{vol}(e_{1,2})=\text{wt}(e_{1,2})$ and $\text{mass}(e_{1,2})=0$. Event $e_{3,2}$ perceives no concurrency at all because none of the possible events that could happen before it can be concurrent to it, i.e., $\text{vol}(e_{3,2})=\text{wt}(e_{3,2})=\text{mass}(e_{3,2})=1$, and $\gamma(e_{3,2})=0$. For event $e_{2,1}$, $\gamma(e_{2,1})=1/4$, because only 1
out of possible 4 events occurred concurrently to it. The value of $\gamma(e_{2,1})$ accounts for the time spent in synchronization delay. Similarly, for event $e_{3,3}$ $\gamma(e_{3,3})=3/8$.

Figure 23 shows various executions of the same distributed computation. Of interest to this discussion is the concurrency perceived by the event indicated by $e$. Observe that the concurrency perceived by $e$ decreases from $\frac{1}{4}$ to $\frac{1}{5}$ to $\frac{1}{6}$ as we go from execution (a) to (b) to (c). In all three cases, the number of events that occur before $e$ and are concurrent to $e$ remains the same (i.e., $=1$) but the total number of time units spent (i.e., volume) increases from 4 to 5 to 6, respectively. Also note that the concurrency perceived by $e$ increases from $\frac{1}{5}$ to $\frac{2}{5}$ as we go from execution (b) to (d). In both the cases, the volume remains the same but the number of events that happen before $e$ and are concurrent to $e$ increases by one.

### 6.2.2 Concurrency Measure for a Computation

In this section, we describe concurrency measures for a computation. We first give some definitions.

In a computation consisting of event set $\mathcal{E}$, we say that an event $e_i$ is a terminating event in $\mathcal{E}$ iff $\mathcal{E}$ does not contain any event $e_j$ such that $e_i \prec e_j$. Intuitively, $e_i$ marks the end of a subset of threads of computation and does not causally affect any other event in the computation. A computation may contain many terminating events.

The measure of concurrency for a computation $C$ is a function of $\gamma$ for all terminating events in $\mathcal{E}$, the set of all events in $C$. Two concurrency measures $\delta$ and $\eta$ for a computation are defined next.

The first measure $\delta(C)$ for a computation $C$ is the arithmetic mean of $\gamma(e_i)$ of all terminating events $e_i$ in the computation.
Figure 23: Values of $\gamma$ for various executions of a distributed computation.
\[ \delta(C) = \frac{\sum e \gamma(e)}{\text{Card}(te)} \]

where \( e \) is a terminating event and \( \text{Card}(te) \) is the cardinality of the set of terminating events, \( te \), in \( C \).

For example, for the computation in Figure 22, \( \delta = \frac{1+\frac{3}{2}+0}{3} = \frac{11}{24} \). Similarly, for computations in Figures 21(a) and 21(b), \( \delta \) is 0 and 1, respectively. Clearly, \( \delta \) for any computation takes values between 0 and 1. A value of 0 indicates that none of the terminating events perceives any concurrency. A value of 1 indicates that all events in the computation are concurrent and no time is spent as synchronization delay.

**Definition 6.2.1 [Event Chain]** Define \( e_1, \ldots, e_{n+1} \) to be an event chain of length \( n \) ending at \( e_{n+1} \) iff \( \forall j: 1 \leq j \leq n: e_j \leadsto e_{j+1} \).

Before describing the second measure for a computation, we define the height of an event. The height of an event \( e_i \) is the length of the longest event chain ending at \( e_i \). Intuitively, the height of an event denotes the earliest time on \( p_i \) at which event \( e_i \) can be executed. The height of an event \( e_i \) can be an important criterion [56] for measuring concurrency.

The second measure \( \eta(C) \) for a computation is the mean (over all terminating events \( e_i \)) of \( \gamma(e_i) \) weighted inversely by \( h\ell(e_i) \).

\[ \eta(C) = \frac{\sum e \frac{\gamma(e_i)}{h\ell(e_i)}}{\text{Card}(te)} \]

The justification for weighting each \( \gamma \) inversely by the height of the corresponding event is that an event with greater height contributes less to the total concurrency.

\(^3\)The subscripts in this definition identify order between events in the sequence only. Processes on which these events occur are irrelevant and hence are omitted from the notation.
of a computation. Clearly, \( \eta \) for any computation takes values between 0 and 1. For the computation in Figure 22, \( \eta = \frac{1+1+\frac{1}{3}+0+\frac{1}{3}}{3} = \frac{3}{5} \). Similarly, for computations shown in Figures 21(a) and 21(b), \( \eta \) is 0 and 1, respectively. The second measure \( \eta \) is a better measure than \( \delta \) since it also considers the height of terminating events. Thus, computations containing terminating events with greater height perceive less concurrency.

### 6.3 Related Work

**Charron-Bost’s Measure 1**

In [19], Charron-Bost proposes two measures of concurrency, \( \rho_i \) and \( \rho \) (called coefficient of coupling), in terms of synchronization delays. These measures are calculated at the end of a computation for each process and for the entire computation, respectively.

\[
\rho_i = \frac{T_i - P_i \delta_i}{\sum_{j \in \{0,...,n\}/i} P_j \delta_j}
\]

\[
\rho = \frac{\sum_{i=1}^{n} (T_i - P_i \delta_i)}{\sum_{i=1}^{n} (\sum_{j \in \{0,...,n\}/i} P_j \delta_j)}
\]

where \( T_i \) is the total execution time of process \( i \) (including synchronization delays), \( P_i \delta_i \) is the time spent by the \( i^{th} \) process on executing events. \( \rho_i \) is the ratio of the time spent by \( p_i \) in synchronization delays to the total computation time. \( \rho_i \) thus takes into account only the synchronization delays at process \( p_i \) rather than in the entire computation. Our measure is computed for each event and it takes into account all synchronization delays that happened before an event is executed.
Charron-Bost’s Measure 2

Charron-Bost proposed another concurrency measure \( m \) [18] for a computation \( C \), to characterize the effects of stopping a process on the computation. In a system with more concurrency, stopping a process would have little effect on other processes. It has been observed [18] that this behavior is reflected by the number of consistent cuts in the distributed computation. The more the number of consistent cuts the more the concurrency in the computation.

\[
m(C) = \frac{\mu - \mu^s}{\mu^c - \mu^s}
\]

where \( \mu \) is the number of consistent cuts in the computation, \( \mu^s \) and \( \mu^c \) are the number of consistent cuts in a computation with the same number of events for a sequential computation and a totally concurrent computation, respectively. \( m(C) \) is computed for the entire computation and not for each event. Computing \( \mu \), however, is not feasible.

The concurrency measures \( \gamma, \delta, \) and \( \eta \), proposed in this chapter are more closely related to the measures proposed by Raynal et al. [56] and Fidge [23].

Raynal’s Measure

Raynal et al. [56] propose a concurrency measure \( \alpha \) which is defined for both individual events and entire computation. For an event \( e_i \), \( \alpha(e_i) \) is defined as a ratio of the actual synchronization delay to the maximum possible synchronization delay.

\[
\alpha(e_i) = 1 - \frac{\text{vol}(e_i) - \text{wt}(e_i)}{\text{vol}(e_i) - \text{ht}(e_i)}
\]

In this measure, the volume of a computation is extended to include the termination delays too (termination delay in a computation is the time spent by processes
Figure 24: Even though event $e_{1,5}$ has more events concurrent to it in (a) than in (b), the value of $\alpha(e_{1,5})$ is the same.
after termination waiting for other processes to terminate.) The measure is defined with the assumption that a process is a total order of events and thus does not account for all intraprocess concurrency and interprocess concurrency. The numerator denotes the time spent in synchronization delay and the denominator denotes the maximum possible synchronization delay. The primary difference between $\alpha$ and $\gamma$ is that $\alpha$ assumes a process to be a totally ordered set of events an thus does not take into account any intraprocess and some interprocess concurrency. $\gamma$ on the other hand, assumes a process to be a partially ordered set of events.

Consider the two computations shown in Figure 24. As stated earlier, we arbitrarily order events on a process that do not have $\rightarrow$ relation between them to deduce an ordering consistent with the happened before relation. Since $\alpha$ models a process as a sequence of events, it cannot distinguish between the two executions. Hence, the value of $\alpha(e_{1,5})$ in both the cases is same. This is not the case for $\gamma$.

**Fidge’s Measure**

Fidge [23] identifies a concurrency measure $\beta$ which is defined for both individual events in a computation and the entire computation itself. For an event $e_i$, $\beta(e_i)$ is defines as:

$$\beta(e_i) = \frac{\rho - \tau}{\rho - 1}$$

where, $\rho$ is the minimum number of events that must be executed before $e_i$ is executed and $\tau$ is the minimum logical time at which $e_i$ can be executed. By definition, $\rho$ is the same as the mass of an event. Note that $\rho$ for $e_i$ is the same as $mass(e_i)$.

This measure attempts to capture the intraprocess concurrency by using the notion of nested subprocesses. $\gamma$ generalizes this by permitting a process to be an
arbitrary partial order on events. The primary difference between $\beta$ and $\gamma$ is that $\beta$ captures synchronization delays only partially. Consider two computations in Figure 25. Fidge’s measure, $\beta$, gives the same value for events $e_{2,3}$ and $e_{2,7}$ in the two computations. However, $\gamma$ gives different values for both of them by taking into account the increased synchronization delay in the computation in Figure 25(b).

6.4 Computing the Measure

In this section, we explain how to compute the concurrency measures proposed in this chapter. Specifically, we describe how to compute the mass, $wt$, $vol$, and $ht$ of an event. We also describe how to determine the set of terminating events in a computation. We use bit-matrix clocks [4] and vector clocks [47, 22] to compute the mass, $wt$, $ht$, and $vol$ of an event.

Computation of $\gamma$

Bit-matrix clocks can be used to compute mass and weight of an event. Specifically, the mass of an event is one less than the sum of all the bits in the bit-matrix timestamp of an event, i.e., it is the count of number of 1’s in the bit-matrix timestamp of the event (excluding itself).

The weight of an event is one less than the sum of the indices of the last 1 in each component of the bit-matrix timestamp of the event.

To compute the volume of an event, we use the scheme of Raynal et al. [56] which is based on vector clocks [22, 47]. Each process $p_i$ maintains a vector $V$ of $N$ integers. $V[i]$ contains the Lamports logical clock [44] value for $p_i$. $V[j], j \neq i$, contains the logical clock value for the last send event on $p_j$ which happens before $e_i$. Each $p_i$
Figure 25: Even though the computation in (b) has more synchronization delay than the computation in (a), they have the same value for $\beta$ for events $e_{2,7}$ and $e_{2,3}$. 
maintains $V$ as follows: On a send or an internal event $p_i$, increments $V[i]$. Each message carries with it the value of $V$ of its send event. On a receive event, when $p_i$ receives from $p_j$ a message carrying the value $m.V$, $p_i$ updates $V$ as follows:

- $V[k] = \max(V[k], m.V[k])$, $k \neq i$
- $V[i] = \max(V[i], m.V[j])$.

The volume of an event then is one less than the sum of all components of $V$.

**Computation of $\delta$**

Computing $\delta$ for a computation requires computing the set of terminating event. We first note that a send event cannot be a terminating events because when a computation terminated all messages would have been received and hence all event chains ending at a send event can be extended to the corresponding receive event. An event $e_i$ which is either a receive event or internal event would be a terminating event iff no other event $e'_i$ (on the same process) exists such that $e_i \sim e'_i$. Since the relation $\sim$ between events can be determined within a process the set of terminating events can be independently computed for each process. Each process maintains a bit-vector $T$ with one bit for each event on the process. Initially all bits are 0. While computing the bit-matrix timestamp of an event $e_i$, $p_i$ performs a logical OR operation on $T$ and the $i^{\text{th}}$ component of its bit-vector timestamp. The $X^{\text{th}}$ bit is then set to 0. At the end of the computation, all events corresponding to bits which are 0 are terminating events.
Computation of $\eta$

Computing $\eta$ for a computation requires computing the height of all terminating events. The height of a terminating event $e_i$ can be computed recursively by setting the height of an event to be one more than the maximum height of events $e_i'$ such that $e_i' \overset{i}{\rightarrow} e_i$ or $e_i' \overset{m}{\rightarrow} e_i$.

6.5 Summary

In this chapter we proposed a new measure of concurrency, $\gamma$, for each event in a distributed computation in which a process itself is a partially ordered set of events. $\gamma$ for an event measures the concurrency observed by the event with respect to events that happened before it. It measures the number of events that are concurrent to it as a fraction of the maximum number of events that could be concurrent to it. $\gamma$ thus takes into consideration the time spent in synchronization delays and all interprocess and intraprocess concurrency. We compared $\gamma$ with measures proposed in the past and showed that $\gamma$ gives a better characterization of a distributed computation.

We defined the set of terminating events in a computation and extended the proposed measure to two measures for an entire computation. The two new measures $\delta$ and $\eta$ are functions of $\gamma$ for terminating events in the computation. We have compared them to other concurrency measures proposed in the past and described methods to compute them.
CHAPTER VII

Conclusions and Future Work

In this dissertation, we investigated two related problems in the design, development, and analysis of distributed algorithms. First, we developed a one-to-many communication abstraction GS which allows the sender to send a message to a group of receivers such that the receipt of the message can be ordered with respect to messages sent in its past and/or its future. Second, we modeled a process as a set of events and a partial order on them. We proposed hierarchical clocks to capture this partial order.

7.1 The GS Communication Abstraction

We proposed a one-to-many communication primitive GS which has two properties: First it ensures global flush property, specified by the message type. Second, it provides the ability to name a group of processes as a destination of a message. The GS primitive supports four message types, namely, global backward flush, global forward flush, global two-way flush, global ordinary message type. The global flush property is based on ordering of the receipt of a message with respect to receipt of messages in its past and/or in its future.

Use of the GS primitive simplifies algorithm design in a distributed environment by providing abstractions to aid in reasoning about message receipt ordering at other
recipients. By using GS primitive, mechanisms to implement message receipt ordering required in an application are subsumed by the underlying system as opposed to being ensured by the application itself. Many interesting algorithms in distributed systems require reasoning about messages in terms of past and future of a message. The GS primitive provides a clear and convenient way to reason about the past and future of messages. It also permits higher communication level concurrency, when compared to primitives proposed in the literature.

**Implementation Techniques**

We presented two implementation techniques of the GS primitive, namely, a counter-based and an F-channel based implementations.

**Counter-based Implementation:** In the counter-based implementation of the GS primitive each site $S_i$ assigns a unique identifier to messages sent, from a sequence of integers starting at one. For each pair of sites $S_j$ and $S_k$ in the system, $S_i$ maintains two counters: one to store the identity of the last message sent from $S_j$ to $S_k$ and the other to store the identity of the last $gb$ or $gt$ sent from $S_j$ to $S_k$. These counters are denoted by $mc_{i[j,k]}$ and $lbt_{i[j,k]}$, respectively. Each message sent by $S_i$ carries the counter arrays $mc_i$ and $lbt_i$. These counters are updated on the receipt of a message. In addition to maintaining these $2N^2$ integer counters, each site also maintains three sets: the set of all messages that have been delivered at the site but are not yet receivable $D$, the set of messages that are receivable but have not yet been received $S_g$, and the set of messages that have been received so far by the process, $R$. A $go$ or a $gb$ becomes receivable at $S_j$ only when for each site $S_k$, the message from $S_k$ to $S_j$ identified by $g.lbt[k,j]$ is in the set $R$. Similarly a $gf$ or a $gt$ becomes receivable at $S_j$. 


only when for each site $S_k$, all messages sent from $S_k$ to $S_j$ with identifier less than or equal to $g.mc[k,j]$ are in $\mathcal{R}$.

**F-channel Based Implementation:** The second implementation of the GS primitive assumes that the underlying system is connected by F-channels and uses a diffusion process which results in a flush message to be sent along each F-channel in the system. We identified min-cut and max-cut for each diffusion process. We also showed that if all $gf$ and $gt$ messages are received at the max-cut and all $go$ and $gb$ messages are received at the min-cut, then the global flush property will not be violated. As opposed to the counter-based implementation in which each message carries $2N^2$ integers, in the F-channel based implementation each message carries just one bounded integer. The information kept at each site is also bounded by the size of the integer carried by the message. The overhead, both in terms of the amount of information carried by a message and the amount of information kept at each site, is independent of the number of processes in the system. It must be noted that implementations of causal broadcast in related works, e.g. ISIS and Psync, also have unbounded overhead in terms of information carried by each message. This is due to the unbounded clocks used to encode this information. The reduced overhead in the F-channel based implementation is at the cost of increased message traffic in the system.

**An Implementation of F-channels:** We also presented a fair and efficient implementation of F-channels which alleviates the problems in existing implementations. Counter-based implementations of the F-channels presented in the literature penalize computations that use fewer $f$, $b$, or $t$ messages because they increase the size
of information that needs to be maintained by the receiver. These implementations also increase the size of information carried by each message. In the proposed implementation, computations that use fewer \( f \), \( b \), and \( t \) messages, have smaller overheads both in terms of the size of received set and the amount of information carried by each message. For example, in a computation that uses only ordinary messages, each message carries a constant number and the size of the received set is also bounded irrespective of the order of the delivery of messages. Since most applications do not use all flush message types, we find that it is useful to provide implementations that only provide a subset of the four flush message types. We compared the proposed implementation with the implementations proposed in the literature and discussed the trade-offs. The comparison is based on the sender’s and receiver’s data structures, the information carried by each message, and the receiver’s logic.

Applications

The \( GS \) primitive finds uses in a variety of applications. To illustrate the usefulness of the \( GS \) primitive in the design and development of distributed algorithms, we discussed a few applications. Specifically, we describe termination detection and bounding logical clocks.

**Termination Detection:** We proposed an algorithm to detect termination of a distributed computation which does not assume FIFO channels between processes. The beauty of this algorithm lies in its simplicity. It is based on identifying a cut at which all processes in the basic computation are idle and then detecting whether there were any messages in transit at that cut. We also discussed optimizations to this basic algorithm.
Due to the nature of the global flush messages, this algorithm does not suffer from the problem faced by other marker based algorithms, i.e., the number of control messages used due to unsuccessful attempts at the detection of termination, is not unbounded in the worst case. The algorithm ensures that in every phase, at least one basic message becomes receivable. Thus the number of control messages in the worst case is bounded by $O(M \times N)$, where $M$ is the number of basic messages exchanged after the detection algorithm has started and $N$ is the number of processes in the basic computation.

**Bounding Logical Clocks:** We defined the notion of bounded clocks and proposed an algorithm to bound vector clocks. The algorithm assumes bounded message delay. In a system with bounded clocks, many messages may have the same timestamp value. The proposed algorithm deals with this ambiguity by dividing the process time line into windows and shadow windows and by ensuring that the timestamp of a message received in a shadow window is unique with respect to other messages received within the same shadow window. This is achieved by limiting the set of sender's windows a process can see (see Definition 4.3.3) in the current shadow window to the last $W$ windows. We also extended the algorithm to bound the vector clocks when the assumption of bounded message delay is removed.

Use of bounded clocks simplifies algorithm development in a distributed environment by bounding the logs and other information kept by a process. Bounded clocks result in saving of space in messages by limiting the size of the timestamp carried by each message. Such bounded clocks also help in trace-based debugging of distributed programs, checkpointing, controlling logs, and predicting average maximal parallelism.
7.2 Viewing a Process as a Partially Ordered Set of Events

We model a process as a set of events and a partial order on the events. Apart from providing a better model for processes with multiple threads of flow control and sites with multiple processes, modeling a process as a partially ordered set of events provides a unified framework to reason about both sequential and parallel/distributed computations. It also results in identification of all interprocess and intraprocess concurrency. The partial order between events in a distributed computation under this model is described by the "affects" relation.

Hierarchical Clocks: We proposed that a computation should be viewed as a set of events and a hierarchical partial order among them. This hierarchical partial order consists of two partial orders: \( \preceq^i \) between events on the same process (due to data dependencies and flow control) and \( \preceq^m \) between events on different processes (due to message sends and receipts). The transitive closure of \( \preceq^i \cup \preceq^m \) is the "affects" relation itself. We proposed hierarchical clocks which precisely capture the hierarchical partial order between events in a distributed computation. Hierarchical clocks assign two timestamps \( \phi^i \) and \( \phi^m \). \( \phi^i \) for an event is used to capture the set of events which happened before it and have the \( \preceq^i \) relation with it. This timestamp is not communicated between processes. The \( \phi^m \) for an event is a vector of \( N \) components. The \( k^{th} \) component of \( \phi^m \) for an event uniquely identifies the last event on process \( p_k \) which causally affects it. Hierarchical clocks provide the following advantages: First, hierarchical clocks can precisely capture causality. Second, the storage overhead at each process due to timestamps is greatly reduced. Third, the communication overhead for each message in terms of size of the timestamp carried by it is greatly reduced.
This is specially advantageous in applications like debugging of distributed programs because in a typical debugging session, the timestamps of relatively few events are ever compared.

**Concurrency Measures:** We proposed a concurrency measure $\gamma$ which characterizes the inherent concurrency as observed by an event in a distributed computation. The concurrency measure, $\gamma$, for an event captures the number of events that happened before the event and are concurrent to the event as a fraction of the maximum number of events that could have happened before it and be concurrent to it. The measure $\gamma$, thus, takes into consideration the time spent in synchronization delays. We defined the set of terminating events in a computation and extended the proposed measure to two measures, for the computation, which measure the inherent concurrency in the entire computation. The two new measures $\delta$ and $\eta$ are functions of $\gamma$ for terminating events in the computation. We described mechanisms to compute these measures. Along with traditional measures like total execution time, these measures give a better characterization of a distributed computation. Such measures aid in further distributing/consolidating the events in a distributed computation.

### 7.3 Future Work

This work can be extended in a number of directions. First, the implementations of the $GS$ primitive proposed in Chapter III assume that the underlying system is connected by reliable point-to-point channels. In reality, this need not be true. Although, this assumption can be independently dealt with at a lower level of implementation detail, it will be interesting to incorporate the ideas of failures and reliability in these implementations. Implementing the global flush property in presence of process fail-
ure is also interesting. It has implications in terms of the amount, duration, and kind of information kept at a site.

The idea of naming a group of processes as a destination of a message needs to be further explored. The global flush property that a process can deduce when a message is received, needs to be redefined in light of processes dynamically joining and leaving a process group.

Distributed shared memory (DSM) allows processes to uniformly access both local and remote information. This uniformity of access simplifies program development. A simple owner based protocol for implementing causal distributed shared memory can be implemented based on our application example of shared token. Under this protocol, attempts to read a location not currently owned by a process would result in generating a request message for its ownership. The requesting process blocks until it receives the ownership. Similarly, attempts to write to a location not currently owned by a process would result in a request message to be generated. Similar approach is taken in the implementation of DSM in [2]. Since implementing a DSM requires that all read and write request be causally ordered and hence requires reasoning about past and future of events, \(GS\) primitive seems to be quite useful. Cheaper implementations of DSM using the \(GS\) primitive and implementing other weaker models of consistency [1, 21, 36, 27] needs to be further explored.

Viewing a process as a partial order of events instead of as a total order of events has been found to be useful. Hierarchical clocks present an interesting timestamping scheme to capture the \(\sim\) relation. We explored its applications in debugging of distributed programs. Its application in other areas needs to be further explored.
BIBLIOGRAPHY


