# Group Communication Specifications: A Comprehensive Study

**Roman Vitenberg**
Department of Computer Science
Technion – Israel Institute of Technology
romanv@cs.technion.ac.il
http://www.cs.technion.ac.il/~romanv

**Idit Keidar**
Lab for Computer Science
Massachusetts Institute of Technology
idish@theory.lcs.mit.edu
http://theory.lcs.mit.edu/~idish

**Gregory V. Chockler**
Computer Science Institute
The Hebrew University of Jerusalem, Israel
grishac@cs.huji.ac.il
http://www.cs.huji.ac.il/~grishac

**Danny Dolev**
Computer Science Institute
The Hebrew University of Jerusalem, Israel
dolev@cs.huji.ac.il
http://www.cs.huji.ac.il/~dolev

September 17, 1999

## Abstract

View-oriented group communication is an important and widely used building block for many distributed applications. Much current research has been dedicated to specifying the semantics and services of *view-oriented Group Communication Systems (GCSs)*. However, the guarantees of different GCSs are formulated using varying terminologies and modeling techniques, and the specifications vary in their rigor. This makes it difficult to analyze and compare the different systems.

This paper provides a comprehensive set of clear and rigorous specifications, which may be combined to represent the guarantees of most existing GCSs. In the light of these specifications, over thirty published GCS specifications are surveyed. Thus, the specifications serve as a unifying framework for the classification, analysis and comparison of group communication systems. The survey also discusses over a dozen different applications of group communication systems, shedding light on the usefulness of the presented specifications.

Defining meaningful GCSs is challenging; such systems typically run in asynchronous environments in which agreement problems that resemble the services provided by group communication services are not solvable. Therefore, many of the suggested specifications turned out to be too trivial, and in particular, solvable by weaker algorithms than the actual implementations. In this paper, the non-triviality issues are addressed by guaranteeing conditional liveness and by using external failure detectors. The resulting specifications are non-trivial on one hand, and allow implementation on the other.

This paper is aimed at both system builders and theoretical researchers. The specification framework presented in this paper will help builders of group communication systems understand and specify their service semantics; the extensive survey will allow them to compare their service to others. Application builders will find in this paper a guide to the services provided by a large variety of GCSs, which would help them chose the GCS appropriate for their needs. The formal framework may provide a basis for interesting theoretical work, for example, analyzing relative strengths of different properties and the costs of implementing them.

# Contents

## List of Figures

## List of Tables

# Part I
# Introduction

## 1 Introduction

*View-oriented group communication systems (GCSs)* are powerful building blocks that facilitate the development of fault-tolerant distributed systems. GCSs typically provide reliable multicast and group membership services. The task of the *membership service* is to maintain a listing of the currently active and connected processes and to deliver this information to the application whenever it changes. The output of the membership service is called a *view*. The reliable multicast services deliver messages to the current view members. The first and best known GCS was developed as part of the Isis toolkit [Bir86]; it was followed by over a dozen others.

Typical applications of view-oriented group communication systems include state-machine replication (for examples, please see [FV97a, FV97b, KD96, ADMSM94, Ami95, FLS97, KFL98]), distributed transactions and database replication (please see [SR96, GS95, KA98, Kei94]), resource allocation (please see [SM98, BDMS97]), load balancing (cf. [Kha98, KFL98]), system management (cf. [ABCD96]) and monitoring (cf. [ASAWM99]), and highly available servers for example, [MP99] and the video-on-demand servers of [ADK99, ACK$^+$97, VvR94].

Recently, GCSs have been exploited for collaborative computing (please see [CHKD96, RCHS97, BFHR98, ACDK97]), for example, distance learning (cf. [AWMY$^+$96, ASYAW$^+$97]), drawing on a shared white board (cf. [Sha96]), video and audio conferences (for examples, please see [CHRC97, Val98]), application sharing (cf. [KCH98, KRB$^+$97]) and even distributed musical "jam sessions" over a network (cf. [GCA$^+$97]).

Traditionally, GCS developers concentrated primarily on system performance, in order to make their systems useful for real-world distributed applications. Only recently, the challenging task of specifying the semantics and services of GCSs has become an active research area (for examples, please see [MAMSA94, FvR95, BDM95, BDM97, BDMS97, FLS97, DPFLS98, DPFLS99, HLvR99, KK99]). However, no comprehensive set of specifications covering all the spectrum of provably useful GCS features has yet been established.

The task of defining a meaningful GCS is complicated by the fact that group communication services resemble agreement problems which are unsolvable in failure-prone asynchronous environments. Many of the suggested specifications fail to capture the non-triviality of existing GCSs. In particular, they are solvable by weaker algorithms than the actual implementations, or even by trivial algorithms (as demonstrated in [ACBMT95]). Other specifications turned out to be too strong to implement (as proven in [CHTCB96]).

The main objective of this paper is to rigorously define a comprehensive set of properties of partitionable GCSs that reflect the usefulness and non-triviality of numerous existing GCS implementations.

### 1.1 Unifying the GCS properties

The guarantees of different GCSs are stated using different terminologies and modeling techniques, and the specifications vary greatly in their rigor. Moreover, many suggested specifications are complicated and difficult to understand, and some were shown to be ambiguous in [ACBMT95]. This makes it difficult to analyze and compare the different systems. Furthermore, it is often unclear whether a given specification is necessary or sufficient for a certain application.

In this paper we formulate a comprehensive set of specification "building blocks" which may be combined to represent the guarantees of most existing GCSs. In light of our properties, we survey and analyze over thirty published specifications which cover over a dozen leading GCSs (including Consul [MPS91b], the system of Cristian and Schmuck [CS95], Ensemble [HLvR99, HvR96], Horus [vRBM96], Isis [BvR94], Newtop [EMS95], Phoenix [MFSW95], Relacs [BDGB94], RMP [WMK95], Spread [AS98], Totem [AMMS$^+$95, MMSA$^+$96], Transis [DM96, ADKM92b] and xAMp [RV92]). We correlate the terminology used in different papers to our terminology. This yields a semantic comparison of the guarantees of existing systems.

Another important benefit of our approach is that it allows reasoning about the properties of applications that exploit group communication. We present here a set of specifications carefully compiled to satisfy the common requirements of many fault tolerant distributed applications. We justify these specifications with examples of applications that benefit from them and of services constructed to effectively exploit them (some examples are: [FLS97, KD96, ADMSM94, FV97a, ABCD96, ACDV97, ADK99, ACK$^+$97, VvR94, SM98, Kha98, KFL98]).

Nonetheless, not all the specifications are useful for all the applications. Experience with group communication systems and reliable distributed applications has shown that there are no "right" system semantics for all applications (cf. [Bir96]): Different GCSs are tailored to different applications that require different semantics and different *qualities of service (QoS)*. Modern GCSs (for example, Ensemble [HvR96]) are designed in a flexible fashion, which allows them to support a variety of semantics and QoS options. Such modular GCSs easily adapt to diverse application needs. When specifying group communication services, it is important to preserve this flexibility.

In order to account for the diverse requirements of different applications, we divide our specifications into independent properties which may be used as building blocks for the construction of a large variety of actual specifications. Individual specification properties may be matched by specific protocol layers in existing GCSs. This makes it possible to separately reason about the guarantees of each layer and the correctness of its implementation (examples of verification of individual layers may be found in [HLvR99, BH98]). Furthermore, the modularity of our specifications provides the flexibility to describe systems that incorporate a variety of QoS options with different semantics.

## 1.2 The specification style

We specify clear and rigorous properties formalized as *trace properties* of an *I/O automaton* [LT89]. The I/O automaton model is widely used for reasoning about distributed applications (please see examples in [Lyn96]), and has been recently exploited for specifying and reasoning about GCSs (for examples, please see [FLS97, Cho97a, DPFLS98, DPFLS99, Kha98, KFL98, HLvR99, BH98, KK99]); it yields specifications that are clear, intuitive and correspond naturally to actual implementations (for example, using the IOA programming language [GLV97, GL98a, GL99, GL98b]). Furthermore, the well-established theory of I/O automata promotes a modular design since it allows one to reason about compositions of automata.

Using logic formulae for stating properties allows us to avoid ambiguity. Furthermore, arbitrary combinations of properties may be derived as conjunctions of formulae that specify different properties. This provides system builders with the flexibility to construct modular systems in which different properties are fulfilled by different modules. The specifications of these modules may be combined using composition of I/O automata.

Furthermore, logic formulae as well as I/O automata-style specifications may be used in computer-assisted proofs: Vitenberg [Vit98] presents a multi-sorted algebra of which the model herein is a possible interpretation. The axioms presented in this paper also conform with Vitenberg's formalism. The benefit of using multi-sorted algebras is that axioms stated using this formalism can be checked with automated theorem proving tools. For example, the Larch Prover [GG91, GHG$^+$93] has been used to prove correctness of several algorithms modeled as I/O automata (please see examples in [SAGG$^+$93, PPG$^+$96, LSGL95]).

## 1.3 The difficulties of formally specifying GCSs

Defining meaningful group communication services is not a simple task; such systems typically run in asynchronous environments in which agreement problems that resemble the services provided by a GCS are not solvable.

Practical systems cannot do the impossible, they can only make their "best-effort". This concept is illustrated by the following example: No system builder can guarantee that his group membership service will be useful at all times. Theoretically, a powerful adversary that fully controls the communication can force every deterministic membership algorithm to either constantly change its mind or to reach inconsistent decisions that do not correctly reflect the network situation[1]. However, existing group communication systems make a "best-effort" attempt to reflect the network situation as much as possible, and indeed succeed most of the time. Note that the group communication systems we are concerned with are not intended for critical (real-time) applications; they run in environments in which such applications cannot be realized. The usefulness of these systems stems from the fact that real networks rarely behave like vicious adversaries.

Many formal specifications of group communication systems do not capture this notion of "best-effort". This results in specifications that can in fact be implemented by algorithms weaker than the actual implementations, or even by trivial algorithms (as demonstrated in [ACBMT95]). Other specifications turned out to be too strong to implement (please see [CHTCB96]). However, since the "best-effort" principle is an important consideration of system builders, actual systems provide more than their specifications require.

In this paper, we address the non-triviality issues using external failure detectors and by reasoning about liveness guarantees at stable periods.

## 1.4 Road-map to this paper

This paper presents specifications for view-oriented group communication systems. Such systems typically provide membership and multicast services within multicast groups. For simplicity's sake, we restrict our attention to the services provided within the context of a single group. This discussion can be easily generalized to multiple groups as long as the services are provided independently for each group. In Section 6.5 we discuss issues that arise when ordering semantics need to be preserved across groups (i.e., for messages multicast in separate groups).

Throughout the paper we make a distinction between *basic* properties and optional ones. Basic properties are satisfied by most group communication systems. In addition, many of the properties presented in this paper are meaningless unless certain basic properties hold.

The rest of this paper is divided into two main parts: the first presents safety properties of group communication systems, and the second, liveness properties. In order to state the liveness properties, we use the failure detector abstraction. While safety properties are preserved in all runs, liveness properties are conditional, that is, are required to be satisfied only if certain assumptions on the failure detector and the underlying network hold. In Section 9 we prove that this is inevitable: without such assumptions, the desired liveness guarantees are not attainable.

[1]Impossibility results to this effect may be found in Section 9 of this paper and in [CHTCB96].

Each of the parts begins with a model section: Section 2 presents the model for all the properties presented in this paper; Section 8 refines the model of Section 2, adding the failure detector abstraction and assumptions required to state the liveness properties.

The safety properties are divided into four sections: Section 3 presents properties of the group membership service; Section 4 – the properties of the reliable multicast service; properties of safe (stable) message indications appear in Section 5; and ordering and reliability properties of certain multicast service types are presented in Section 6. The liveness properties are presented in Section 10.

Finally, Section 11 concludes the paper; it contains tables that summarize all the properties presented in this paper. In these tables, we also distinguish between basic and optional properties.

Part II

# Safety Properties of Group Communication Services

## 2 The Model and Presentation Formalism

The system we consider contains a set $\mathcal{P}$ of processes that communicate via message passing. The underlying communication network provides unreliable datagram message delivery. There is no known bound on message transmission time, hence the system is *asynchronous*. The system model allows for the following changes: Sites may crash and recover; messages may be lost, failures may partition the network into disjoint components, and previously disjoint components may merge.

In this paper, we assume that no Byzantine failures occur, that is, processes do not behave in a malicious manner. (Most of the work on group communication does not address Byzantine failures. However, such failures are addressed in the Rampart system [Rei94, Rei95, Rei96b, Rei96a] and in [MMR97, MR97]).

Processes are modeled as untimed I/O automata (cf. [LT89] and [Lyn96], Chapter 8). We are concerned only with the external behavior of I/O automata, as reflected in their *external signature* and in their *fair traces*. The external signature of an automaton consists of two sets of actions by which the automaton interacts with its environment: input actions and output actions. A trace of an I/O automaton is the sequence of external actions it takes in an execution; actions are executed atomically. Roughly speaking, a *fair trace* is a trace of an execution in which enabled actions eventually become executed. For formal definitions, please see [Lyn96], Chapter 8.

We model the system as the *GCS service*. We present the GCS service specification by defining its external signature in Section 2.1 below, and a collection of *trace properties* throughout the rest of this paper. Each trace property is presented as an *axiom* in the set-theoretic mathematical model described in Section 2.2 below. A specification consists of an external signature and a set of such axioms. We say that an I/O automaton satisfies the specification if all of its fair traces satisfy the axioms that comprise the specification.

### 2.1 The external signature of the GCS service

The GCS specification models the the behavior of the entire system. In the specification, we use the following types:

$\mathcal{P}$ - the set of processes.

$\mathcal{M}$ - the set of messages sent by the application.

**VIDs** - the set of view identifiers; we assume that VIDs is partially ordered by the $<$ operator.

Each action in the GCS external signature is parameterized by a unique process $p \in \mathcal{P}$ at which this action occurs. The GCS interacts with the application as depicted in Figure 1. The external signature of the GCS consists of the following actions:

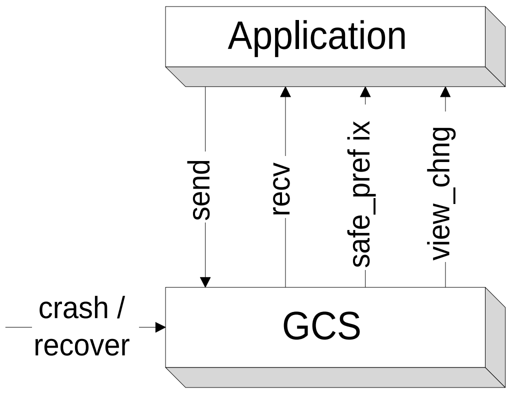

Figure 1: External actions of the GCS.

**Interaction with the application**

The application uses the GCS to send and receive messages, and also receives view change notifications and possibly safe prefix indications (cf. Section 5) from the GCS. Note: we include safe prefix indications in the signature, although not every interesting GCS will actually provide them.

- input $\mathbf{send}(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- output $\mathbf{recv}(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
  Note: The receive action does not contain the sender as an explicit parameter. In specific implementations of the automaton, the receiver may learn of the sender's identity by including the sender's identifier in the message text.
- output $\mathbf{view\_chng}(p, \langle id, members\rangle, T), p \in \mathcal{P}, id \in \mathit{VIDs}, members \in 2^{\mathcal{P}}, T \in 2^{\mathcal{P}}$
  $id$ is the view identifier, $members$ is the set of members in the new view and $T$ is the *transitional set* of the *Extended Virtual Synchrony (EVS)* [MAMSA94] model (cf. Section 4.3.1).
- output $\mathbf{safe\_prefix}(p, m), p \in \mathcal{P}, m \in \mathcal{M}$

**Interaction with the environment**

The following actions model actions that may occur in the environment and affect the GCS:

- input $\mathbf{crash}(p), p \in \mathcal{P}$
- input $\mathbf{recover}(p), p \in \mathcal{P}$

**Definition 2.1 (Event)** *An* event *is an occurrence of an* action *from the GCS external signature.*

**Definition 2.2 (Trace)** *A* trace *is a sequence of events.*

## 2.2 The mathematical model

We now present the mathematical model for stating trace properties of a GCS with the signature described in Section 2.1 above. We use *set theory* notation to state our axioms; we define the following sets:

$\mathcal{P}$, $\mathcal{M}$, **VIDs** - are basic sets as described above.

$\mathcal{V}$ - the set of views delivered in **view_chng** actions is: VIDs $\times 2^{\mathcal{P}}$. Thus, a view $V \in \mathcal{V}$ is a pair. We refer to the first element in the pair as *V.id* and to the second element as *V.members*.

**Actions** The set of actions is:
$\{\textbf{send}(p,m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup \{\textbf{recv}(p,m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\textbf{view\_chng}(p,V,T) \mid p \in \mathcal{P}, V \in \mathcal{V}, T \in 2^{\mathcal{P}}\} \cup \{\textbf{safe\_prefix}(p,m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\textbf{crash}(p) \mid p \in \mathcal{P}\} \cup \{\textbf{recover}(p) \mid p \in \mathcal{P}\}$

**Traces** – sequences of actions.

**Events** – events which are members of traces.

Our modeling of a trace as a sequence of events captures the assumption that events are atomic.

Note that the first parameter in each event is a process in $\mathcal{P}$. Therefore, we can define the function: $pid : Events \rightarrow \mathcal{P}$ which returns the process at which each event occurs.

Since all of our axioms classify traces, they all take a trace as a parameter. For clarity of the presentation, we make the trace parameter implicit: We fix a trace $Trace = t_1, t_2, \ldots$, and all the axioms are stated with respect to this trace.

In our axioms, we omit universal quantifiers: when a variable is unbound it is understood to be universally quantified for the scope of the entire formula.

## 2.3 Notation

With a view-oriented group communication service, events occur at processes within the context of views. The function $viewof : Events \times \mathcal{P} \rightarrow \mathcal{V} \cup \{\perp\}$ returns the view in the context of which an event occurred at a specific process. Note that for a **view_chng** event, it is not the new view introduced, but rather the process' previous view. At startup time and following a crash, a process is not considered to be in any view (modeled by $\perp$). Some specifications (for example, those of [FLS97, DPFLS98, CHD98]) assume knowledge of a default view in which the process is considered to be at startup time. However, their specifications do not address the issue of recovery from crash and therefore do not specify a process' view following recovery. Actual GCSs, on the other hand, do not typically assume knowledge of default views. Therefore, we chose not to include default views in our specifications.

**Definition 2.3 (viewof)** *The view of an event $t_i$ at process p is the view delivered in a* **view_chng** *event, $t_j$, which precedes $t_i$ and such that there is no* **view_chng** *or* **crash** *events between $t_j$ and $t_i$; the view is $\perp$ if there is no such $t_j$. Formally:*

$$viewof(t_i, p) \overset{\text{def}}{=} \begin{cases} V & \text{if } \exists j \exists T : t_j = \textbf{view\_chng}(p,V,T) \ \wedge \ j < i \ \wedge \\ & \nexists k : j < k < i \ \wedge \ (t_k = \textbf{crash}(p) \ \vee \ (\exists T' \exists V' : t_k = \textbf{view\_chng}(p,V',T'))) \\ \perp & \text{otherwise} \end{cases}$$

We define some general shorthand predicates in Table 1 below. In all these predicates as well as throughout the rest of this paper, variables named $V$ and $V'$ are members of $\mathcal{V}$ (not $\perp$), variables named $p$ and $q$ are taken from $\mathcal{P}$, variables named $m$ and $m'$ are members of $\mathcal{M}$, variables named $T$, $T'$ and $S$ are in $2^{\mathcal{P}}$ and variables $i$, $j$ and $k$ are integers.

| | | | |
|---|---|---|---|
| Process $p$ receives message $m$: | | | |
| $\mathit{receives}(p,m)$ | $\stackrel{\text{def}}{=}$ | $\exists i:$ | $t_i = \mathbf{recv}(p,m)$ |
| Process $p$ receives message $m$ in view $V$: | | | |
| $\mathit{receives\_in}(p,m,V)$ | $\stackrel{\text{def}}{=}$ | $\exists i:$ | $t_i = \mathbf{recv}(p,m) \wedge \mathit{viewof}(t_i,p) = V$ |
| Process $p$ sends message $m$: | | | |
| $\mathit{sends}(p,m)$ | $\stackrel{\text{def}}{=}$ | $\exists i:$ | $t_i = \mathbf{send}(p,m)$ |
| Process $p$ sends message $m$ in view $V$: | | | |
| $\mathit{sends\_in}(p,m,V)$ | $\stackrel{\text{def}}{=}$ | $\exists i:$ | $t_i = \mathbf{send}(p,m) \wedge \mathit{viewof}(t_i,p) = V$ |
| Process $p$ installs view $V$: | | | |
| $\mathit{installs}(p,V)$ | $\stackrel{\text{def}}{=}$ | $\exists i \exists T:$ | $t_i = \mathbf{view\_chng}(p,V,T)$ |
| Process $p$ installs view $V$ in view $V'$: | | | |
| $\mathit{installs\_in}(p,V,V')$ | $\stackrel{\text{def}}{=}$ | $\exists i \exists T:$ | $t_i = \mathbf{view\_chng}(p,V,T) \wedge \mathit{viewof}(t_i,p) = V'$ |
| Event $t_i$ is the next event after $t_j$ at process p: | | | |
| $\mathit{next\_event}(i,j,p)$ | $\stackrel{\text{def}}{=}$ | $j < i \wedge$ | $\mathit{pid}(i) = \mathit{pid}(j) = p \wedge \nexists k : \mathit{pid}(k) = p \wedge j < k < i$ |
| Event $t_i$ is the previous event before $t_j$ at process p: | | | |
| $\mathit{prev\_event}(i,j,p)$ | $\stackrel{\text{def}}{=}$ | $j > i \wedge$ | $\mathit{pid}(i) = \mathit{pid}(j) = p \wedge \nexists k : \mathit{pid}(k) = p \wedge j > k > i$ |

Table 1: General shorthand predicate definitions.

### 2.4 Assumptions about the environment

In our model, we assume that no events occur at a process between crash and recovery.

**Assumption 2.1 (Execution Integrity)** *The next event that occurs at a process after a crash is recovery, and the previous event before a recovery is a crash. Formally:*
$(\mathit{next\_event}(i,j,p) \wedge t_j = \mathbf{crash}(p) \;\Rightarrow\; t_j = \mathbf{recover}(p)) \;\wedge$
$(\mathit{prev\_event}(i,j,p) \wedge t_j = \mathbf{recover}(p) \;\Rightarrow\; t_j = \mathbf{crash}(p))$

In order to distinguish between the messages sent in different send events, we assume that each message sent by the application is tagged with a unique message identifier, which may consist, for example, of the sender identifier and a sequence number or a timestamp. Thus, we can require that every message is sent at most once in the system. This assumption is not essential because a GCS can provide the same guarantees without it by adding a sequence number to distinguish between different instances of application messages. It does, however, simplify the presentation and the definitions of further requirements.

**Assumption 2.2 (Message Uniqueness)** *There are no two different send events with the same content. Formally:*
$t_i = \mathbf{send}(p,m) \;\wedge\; t_j = \mathbf{send}(q,m) \;\Rightarrow\; i = j$

## 3 Safety Properties of the Membership Service

A membership service is a vital part of every group communication system (for examples, please see [BJ87, ADKM92b, AMMS$^+$95, FvR95, WMK95, EMS95, BDGB94, MFSW95]). In this section we describe typical properties of membership services.

We begin, in Section 3.1, with some basic safety properties fulfilled by most group communication systems. In Section 3.2 we compare two approaches to group membership: partitionable and primary component.

## 3.1 Basic properties

Our first basic safety property requires that a process is always a member of its view.

**Property 3.1 (Self Inclusion)** *If process $p$ installs view $V$, then $p$ is a member of $V$. Formally:* $installs(p, V) \Rightarrow p \in V.members$

Since a membership of a view reflects the ability to communicate with the process and a process is always able to communicate with itself, this property holds in all group communication systems and specifications. It is explicitly specified in [DMS95, FvR95, EMS95, BDM95, FLS97].

### 3.1.1 View identifier order

Our next basic property requires that the view identifiers of the views that each process installs are monotonically increasing.

**Property 3.2 (Local Monotonicity)** *If a process $p$ installs view $V$ after installing view $V'$ then the identifier of $V$ is greater than that of $V'$. Formally:*
$t_i = \mathbf{view\_chng}(p, V, T) \ \wedge \ t_j = \mathbf{view\_chng}(p, V', T') \ \wedge \ i > j \ \Rightarrow \ V.id > V'.id$

Property 3.2 has two important consequences: it guarantees that a process does not install the same view more than once and that if two processes install the same two views, they install these views in the same order.

As long as there are no recoveries from crashes, Local Monotonicity is satisfied by virtually all group membership systems (examples include: [RB91, DMS95, AMMS$^+$95, FvR95, BDM97, EMS95, MS94, KSMD99]); it is also required in all the group membership specifications (some examples are: [Nei96, FLS97, DPFLS98, DPFLS99]). [BDM97] states an equivalent property: The order in which processes install views is such that the successor relation is a partial order. This is equivalent to the property herein, since the partial order derived by successors coincides with the partial order defined on the VIDs set.

However, some group communication systems may violate Local Monotonicity in case a process crashes and recovers with the same identity: when the process recovers, it installs its initial view, whose identifier is smaller than the last view it installed before crashing. There are several ways to remedy this shortcoming: In Isis [RB91] a process recovering after a crash is assigned a different identifier (using a new incarnation number). It is also possible to overcome this problem by saving information on a disk before each view installation.

RMP [WMK95] guarantees uniqueness of views, (although not monotonicity), even in the face of crashes by initializing a local counter to be the real clock value when a computer recovers from a crash.

There are different ways to generate view identifiers: In Transis [DMS95] the view identifier is a positive integer. This integer is computed based on the values of local counters, maintained by all processes. This local counter is increased by a process upon each installation. The counters in the specifications of [FLS97] and [Nei96] are taken from an ordered set. Hence, an integer counter is again a possible implementation. In Horus [FvR95] and [CS95], a view identifier is a pair $\langle p, c \rangle$ where $p$ is the process that created the view and $c$ is a value of a local counter on $p$. In Totem, a view identifier is a triple of integers, ordered lexicographically. In [KSMD99] the view identifier is a pair consisting of a vector that maps view members to integer counters and an integer, where the integer part of the view identifier is monotonically increasing. Newtop [EMS95] uses a logical timestamp to sign all messages. At the moment of the new view creation the maximum value among the timestamps of all view members satisfies all the properties of a view identifier.

The importance of view ordering properties is noted and emphasized in several works, for example in [HS95, FV97a]. The protocol of [CHD98] uses Local Monotonicity (Property 3.2) in order to implement a totally ordered multicast service. Other examples of applications that exploit view ordering can be found in [KD96, Ami95, FV97a].

### 3.1.2 Initial view event

We have already seen that with a view-oriented group communication system, events occur in the context of views. However, as per our definitions, this is not the case for all events: events that occur before the first view event are not considered to be occurring in any view. GCSs typically install an initial view at startup time and upon recovery from a crash (unless they crash before doing so), and thus *every* **send**, **recv** and **safe_prefix** event in these GCSs occurs in some view. This requirement is stated in Property 3.3 below.

**Property 3.3 (Initial View Event)** *Every* **send**, **recv** *and* **safe_prefix** *event occurs within some view. Formally:*
$t_i = \mathbf{send}(p,m) \ \vee \ t_i = \mathbf{recv}(p,m) \ \vee \ t_i = \mathbf{safe\_prefix}(p,m) \ \Rightarrow \ \mathit{viewof}(t_i,p) \neq \bot$

Note: In order to enforce this property, one has to restrict the behavior of the application, so that no **send** events occur before the first **view_chng** event.

The initial view can be determined in one of two ways:

- At startup, processes use the membership service to agree upon the view, as they do for any other view. Thus, no pre-defined knowledge about processes in the system is required. Most GCSs adopt this option, for example, Isis [BSS91] and Ensemble [HvR96].

- Each process unilaterally decides upon its initial view without communication with other processes. This approach is equivalent to having default views, but with an explicit initial view installation event. Transis [DMS95] and Consul [MPS91b] take this approach.

  The initial view may be singleton or may consist of all possible processes in the system. In [HS95] these two possibilities are called *individual startup* and *collective startup*, respectively. Transis is an example of a GCS which uses individual startup, and collective startup is deployed, for example, in Consul. Note that in order to install anything different than a singleton view, a process must possess *a priori* knowledge about other processes in the system. Such knowledge is assumed, for example, in [FLS97] and [MPS91b].

We do not provide a formal specification for each of these possibilities in this paper – Property 3.3 (Initial View Event) accounts for installing initial views in the most general way.

## 3.2 Partitionable vs. primary component membership services

A membership service may either be *primary component*[2] or *partitionable.* In a primary component membership service, views installed by all the processes in the system are totally ordered; in a

[2]A *primary component* was originally called a *primary partition.*

partitionable membership service, views are only partially ordered, that is, multiple disjoint views may exist concurrently. A GCS is partitionable if its membership service is partitionable; otherwise it is primary component.

All the safety properties presented above concern partitionable membership services as well as primary component ones. However, they do not enforce a total order on views, and thus, the specifications are partitionable. In order to specify a primary component membership service, we add a safety property that imposes a total order on views. Property 3.4 (Primary Component Membership) below requires that the set of views installed in a trace form a sequence such that every two consecutive views (in this sequence) intersect. The sequence is modeled as a function from the set of views installed in the trace to the natural numbers.

**Property 3.4 (Primary Component Membership)** *There is a one to one function $f$ from the set of views installed in the trace to the natural numbers such that every view $V$ with $f(V) > 1$ contains a member that installed $V$ in a view $V'$ such that $f(V') + 1 = f(V)$. Formally:*
$\exists f : \{V | \exists p : installs(p, V)\} \rightarrow \mathcal{N}$ *such that:*
$(f(V) = f(V') \Rightarrow V = V') \wedge$
$(\forall V : f(V) > 1 \;\Rightarrow\; \exists V' : f(V) = f(V') + 1 \;\wedge\; \exists p \in V.members : installs\_in(p, V, V'))$

This property implies that for every pair of consecutive views, there is a process that survives from the first view to the second (i.e., does not crash between the installations of these two views). Such a surviving process may convey information about message exchange in the first view to the members of the second. Similar properties appear in [MS94, RB91, YLKD97, DPFLS98].

The first and best known group membership service is the primary component membership service of Isis [BvR94]. It was followed by many other primary component membership services, for example, those of Phoenix [MS94], Consul [MPS91b] and xAMp [RV92]. Primary component membership services are also specified in [CHTCB96, Nei96, Cri91, MPS91a, DPFLS98, DPFLS99]. Consul [MPS91b], xAMp [RV92] and [Cri91] guarantee membership service properties only as long as no network partitions occur. In contrast, Isis [RB91] and Phoenix [MS94] do assume the possibility of network partitions, but allow execution of the application to proceed only in a single component. In Isis [RB91] detached processes “commit suicide”, whereas in Phoenix [MS94] they are blocked until the link is mended.

The first partitionable membership service was introduced as part of the Transis [ADKM92b, DM96, ADKM92a] group communication system. Since then, numerous new GCSs featuring a partitionable membership service have emerged, for example, those of Totem [AMMS$^+$95, MMSA$^+$96], Horus [vRBM96], RMP [WMK95], Newtop [EMS95] and RELACS [BDGB94]. Partitionable membership services are discussed in the specifications of [MAMSA94, FLS97, BBD96, CS95, JFR93, KK99]. [HS95] presents a specification of a primary component membership service and shows how to extend it to a specification of a partitionable one.

Partitionable membership services are useful for a variety of applications, for example, resource allocation (please see [SM98, BDMS97]), system management (cf. [ABCD96]), monitoring (cf. [ASAWM99]), highly available servers (cf. [MP99, ADK99, ACK$^+$97]) and collaborative computing applications such as drawing on a shared white board (cf. [Sha96]), video and audio conferences (cf. [CHRC97, Val98]), application sharing (cf. [KCH98, KRB$^+$97]) and even distributed musical “jam sessions” over a network (cf. [GCA$^+$97]).

In contrast, applications that maintain globally consistent shared state (for example, [FV97a, FV97b, KD96, ADMSM94, Ami95, FLS97, KFL98, SR96, GS95, KA98, Kei94]) usually avoid inconsistencies by allowing only members of one view (the primary one) to update the shared state at a given time (please see discussion in [HS95]). For the benefit of such applications, some partitionable membership services (for example, [FvR95, HS95]) notify processes whether they are in a *primary view* or not, such that the primary views satisfy Property 3.4 (Primary Component Membership) above. The dynamic-voting based algorithm of [YLKD97] runs atop a partitionable membership service and provides such notifications. The benefit of using a partitionable membership service for such applications is that members of non-primary views may access the data for reading purposes.

## 4 Safety Properties of the Multicast Service

We now discuss the multicast service, and its relationship with the group membership service.

GCSs typically provide various types of multicast services. Traditionally, GCSs provide reliable multicast services with different delivery ordering guarantees. Several modern group communication systems have incorporated a multicast paradigm that provides the QoS of the underlying communication, allowing a single application to exploit multiple QoS options. For example, in RMP, the *unreliable* QoS level provides the guarantees of the underlying communication. Similarly, the MMTS [CHKD96] extends Transis [ADKM92b, DM96] by providing a framework for *synchronization* of messages with different QoS properties; Maestro [BFHR98] extends the Ensemble [HvR96] GCS by coordinating several protocol stacks with different QoS guarantees and the Collaborative Computing Transport Layer (CCTL) [RCHS97] implements similar concepts, geared towards distributed collaborative multimedia applications.

Most of the multicast properties we formulate below are typically fulfilled only by reliable multicast paradigms, and not by multicast services that directly provide the QoS of the underlying communication layer.

### 4.1 Basic properties

Our first property requires that messages never be spontaneously generated by the group communication service.

**Property 4.1 (Delivery Integrity)** *For every* **recv** *event there is preceding* **send** *event of the same message:*
$t_i = \textbf{receive}(p, m) \;\Rightarrow\; \exists q \exists j : j < i \;\wedge\; t_j = \textbf{send}(q, m)$

This property is trivially implemented, so all GCSs support it; it is explicitly specified in [BDM95, RV92, FLS97, DPFLS98, DPFLS99, KK99].

The following property states that messages are not duplicated by the group communication service, that is, every message is received at most once by each process:

**Property 4.2 (No Duplication)** *Two different recv events with the same content cannot occur at the same process. Formally:*
$t_i = \textbf{recv}(p, m) \;\wedge\; t_j = \textbf{recv}(p, m) \;\Rightarrow\; i = j$

Most GCSs eliminate duplication (some examples are: [BDM95, EMS95, ADKM92b, KK99]). However, when a GCS directly provides the same QoS as the underlying communication layer, duplication is not eliminated, for example, in the Unreliable and Unordered QoS levels of RMP [WMK95].

## 4.2 Sending View Delivery and weaker alternatives

With a view-oriented group communication service, send and receive events occur within the context of views[3]. Several GCS specifications require that a message be delivered in the context of the same view as the one in which it was sent; other specifications weaken this requirement in a variety of ways. In this section we discuss this property and some of its weaker alternatives.

### 4.2.1 Sending View Delivery

Many GCSs guarantee that a message be delivered in the context of the view in which it was sent, as specified in the following property:

**Property 4.3 (Sending View Delivery)** *If a process $p$ receives message $m$ in view $V$, and some process $q$ (possibly $p = q$) sends $m$ in view $V'$, then $V = V'$. Formally:*
$\mathit{receives\_in}(p, m, V) \ \wedge \ \mathit{sends\_in}(q, m, V') \ \Rightarrow \ V = V'$

Among the group communication systems that support Sending View Delivery are Isis [BJ87] and Totem [AMMS+95]. In contrast, Newtop [EMS95] and RMP [WMK95] do not guarantee Property 4.3. Horus allows the user to chose whether this property should be satisfied or not; the programming model in which it is satisfied is called Strong Virtual Synchrony (SVS) [FvR95]. Property 4.3 also appears in various GCS specifications (for examples please see [MAMSA94, FLS97, HS95, DPFLS98, DPFLS99, KK99]).

Sending View Delivery is exploited by applications to minimize the amount of context information that needs to be sent with each message, and the amount of computation time needed to process messages. For example, there are cases in which applications are only interested in processing messages that arrive in the view in which they were sent. This is usually the case with *state transfer* messages sent when new views are installed (examples of applications that send state transfer messages include [ACDV97, SM98, HS95, FV97a, ACDV97, AAD93, KD96, KFL98, Kha98]). Using Sending View Delivery, such applications do not need to tag each state transfer message with the view in which it was sent. Sending View Delivery is also useful for applications that send vectors of data corresponding to view members. Such an application can send the vector without annotations, relying on the fact that the $i$th entry in the vector corresponds to the $i$th member in the current view (as explained in [FvR95]).

Unfortunately, in order to satisfy Sending View Delivery without discarding messages from live and connected processes, processes must *block* sending of messages for a certain time period before a new view is installed. In fact, Friedman and van Renesse [FvR95] prove that without such blocking, satisfying Sending View Delivery entails violating other useful properties such as Property 4.5 (Virtual Synchrony) and Property 10.1.3 (Self-delivery) below. Therefore, in order to fulfill Property 4.3, group communication systems block sending of messages while a view change is taking place. In order to notify the application that it needs to stop sending messages, the GCS sends a *block* request to the application. The application responds with a *flush* message which follows all the messages sent by the application in the old view. The application then refrains from sending messages until the new view is delivered.

An alternative way to satisfy Property 4.3 is by discarding certain messages that arrive in the course of a membership change or in later views, and thus violating at least one of Self-delivery and Virtual Synchrony, as well as the "best-effort" principle. We are not aware of any GCS that takes this approach.

[3]Note that if there is no initial view event, messages may be sent and received in the context of no view. The properties below only apply to those send and receive events that do occur in the context of some view.

### 4.2.2 Same View Delivery

In order to avoid blocking the application, some GCSs weaken the Sending View Delivery property and require only that a message be delivered at the same view at every process that delivers it. This is specified in the Same View Delivery property as follows:

**Property 4.4 (Same View Delivery)** *If processes p and q both receive message m , they receive m in the same view. Formally:*
$receives\_in(p, m, V) \wedge receives\_in(q, m, V') \Rightarrow V = V'$

Same View Delivery is a *basic* property. It holds in all the group communication systems and specifications surveyed herein, for example, in Transis [ADKM92b], Relacs [BDM95] and in all the GCSs that support Property 4.3 above. (Same View Delivery is called the View-Synchronous Communication Service M2 property in [BDM95]).

Same View Delivery is strictly weaker than Sending View Delivery. However, it is sufficient for applications that are not interested in knowing in which views messages are multicast, some examples are: [Cho97a, CHD98, KD96, ABCD96, ACK$^+$97, ADK99].

Sussman and Marzullo [SM98] compare the relative strengths of Same View Delivery and Sending View Delivery for solving a simple resource allocation problem in a partitionable environment. They define a metric specific to this application that captures the effects of the uncertainty of the global state caused by partitioning; this uncertainty is measured in terms of the quantity of resources that cannot be allocated. They show that when using *totally ordered multicast* (cf. Section 6.3), algorithms that use Same View Delivery and Sending View Delivery perform equally in terms of this metric, while if FIFO *multicast* is used (cf. Section 6.1), algorithms that use Sending View Delivery are superior with respect to this metric to those that use Same View Delivery. Thus, they identify a tradeoff between the cost of totally ordered multicast and the cost of Sending View Delivery.

There are two kinds of systems that provide Same View Delivery without Sending View Delivery: systems that provide stronger semantics than Same View Delivery (yet weaker than Sending View Delivery), as described in Section 4.2.3 below, and systems that are built around a small number of servers that provide group communication services to numerous application clients (for example Transis [ADKM92b, DM96] and Spread [AS98]). In the latter kind of systems, client membership is implemented as a “light-weight” layer that communicates with a “heavy-weight” Sending View Delivery layer asynchronously[4] using a FIFO buffer, as illustrated in Figure 2. The asynchrony may cause messages to arrive in later views than the ones in which they were sent. However, since the asynchronous buffer preserves the order of **recv** and **view_chng** events, messages are delivered in the same view at all destinations. Thus, at the client level, only Same View Delivery is supported. The benefit of using such a design is that the group membership service can proceed to agree upon the new view without waiting for flush messages indicating that all the clients are blocked.

### 4.2.3 The Weak Virtual Synchrony and Optimistic Virtual Synchrony models

In order to eliminate the need for blocking, and yet provide support for a certain type of view-aware applications, Friedman and van Renesse [FvR95] introduce the Weak Virtual Synchrony (WVS) programming model which replaces Sending View Delivery with a weaker alternative: In WVS every installation of a view $V$ is preceded by at least one *suggested view* event. The membership of the suggested view is guaranteed to be an ordered superset of $V$. Property 4.3 is replaced by

[4]Therefore, Same View Delivery is called *Asynchronous Virtually Synchronous Communication (AVSC)* in [SM98].

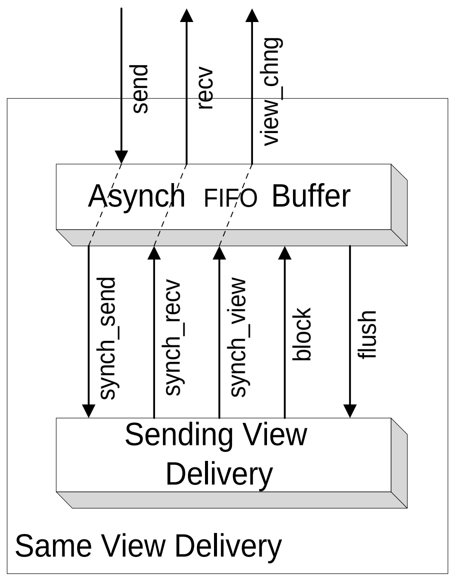

Figure 2: Implementing Same View Delivery over Sending View Delivery.

the requirement that every message sent in the suggested view is delivered in the next regular view. This allows processes to send messages while the membership change is taking place. The processes that use WVS maintain translation tables that map process ranks in the suggested view to process ranks in the new view. Thus, although messages are no longer guaranteed to be delivered in the view in which they were sent, an application may still send vectors of data corresponding to processes without annotations.

One shortcoming of the WVS model is that once a suggested view is delivered, it does not allow new processes to join the next regular view. If a new process joins while a view change is taking place, a protocol implementing WVS is forced to install an *obsolete* view, and then immediately start a new view change to add the joiner. This behavior violates the "best-effort" principle. A second shortcoming of WVS is that it is useful only for view-aware applications that are satisfied with knowledge of a superset of the actual view, and does not suffice for certain view-aware applications (for example, [YLKD97]) that require messages to be delivered in a view identical to the one in which they are sent.

These shortcomings are remedied by the *Optimistic Virtual Synchrony (OVS)* model, recently introduced by Sussman et al. [SKM99]. In OVS, each view installation is preceded by an *optimistic view* event, which provides the application with a "guess" what the next view will be. After this event, applications may optimistically send messages assuming that they will be delivered in a view identical to the optimistic view (note that this will be the case unless further changes in the system connectivity occur during the membership change). If the next view is not identical to the optimistic view, the application may still choose to use the messages (for example, if the new view is a subset of the optimistic view and WVS semantics are required) or roll-back the optimistic messages.

The WVS and OVS models both pose weaker alternatives to Sending View Delivery, and both imply Property 4.4 (Same View Delivery). Furthermore, according to the metric suggested in [SM98], algorithms that exploit WVS or OVS perform the same as algorithms that exploit Property 4.3 (Sending View Delivery).

## 4.3 The Virtual Synchrony property

We now present an important property of virtually synchronous communication that is often referred to as "Virtual Synchrony". This property requires two processes that participate in the same two consecutive views to deliver the same set of messages in the former.

**Property 4.5 (Virtual Synchrony)** *If processes $p$ and $q$ install the same new view $V$ in the same previous view $V'$, then any message received by $p$ in $V'$ is also received by $q$ in $V'$. Formally:* $installs\_in(p, V, V') \wedge installs\_in(q, V, V') \wedge receives\_in(p, m, V') \Rightarrow receives\_in(q, m, V')$

Virtual Synchrony is perhaps the best known property of GCSs, to the extent that it engendered the whole Virtual Synchrony model[5]. This property was first introduced in the Isis literature [BJ87, BvR94, BSS91, Bir93] in the context of a primary component membership service and later extended to a partitionable membership service [FvR95, DMS95, EMS95, MAMSA94, BDM95]. In [MAMSA94] and [FV97a] it is called "failure atomicity", and in [BDM95] it is called "view synchrony". Virtual Synchrony is supported by nearly all group communication systems, either for all multicast services (for example, in Ensemble [HvR96], Horus [FvR95], Isis [BJ87], Newtop [EMS95], Phoenix [SS93], Relacs [BDM97], Totem [AMMS$^+$95] and Transis [DMS95]) or only for some multicast services, like the totally ordered multicast of RMP [WMK95]. It also appears in specifications, for example, in [HS95, HLvR99, KK99]. An exception is set by the specifications of [FLS97, DPFLS98, DPFLS99] which do not include this property.

Virtual Synchrony is especially useful for applications that implement data replication using the state machine approach [Sch90], (for examples please see [SM98, HS95, FV97a, ACDV97, AAD93, KD96, KFL98, Kha98]). Such applications change their state when they receive application messages. In order to keep the replica in a consistent state, application messages are disseminated using totally ordered multicast.

Whenever the network partitions, the disconnected replica may diverge and reach different states. When previously disconnected replica reconnect, they perform a *state transfer*, that is, exchange special *state* messages in order to reach a common state. A group communication system that supports Virtual Synchrony allows processes to avoid state transfer among processes that "continue together" from one view to another, as explained in [ACDV97]: Whenever the membership service installs a new view $V$ (with the membership $V.members$) at a process $p$, $p$ should first determine the set $T$ of processes in $V.members$ that were also in p's previous view $V'$, and have proceeded directly from $V'$ to $V$ (i.e., installed view $V'$ and did not install any view after $V'$ and before $V$). If, for example, $T = V.members$, then according to the Virtual Synchrony property, each replica in $V.members$ has received the same set of messages in $V'$ and therefore has the same state upon installing view $V$. Hence, no state transfer is required.

Note that $T$ (as defined above) is not necessarily the intersection of the *members* sets of the new view and the previous one, as demonstrated in Figure 3. In this example, $p$ and $q$ are initially in the same connected component (both install $\langle 1, \{p, q\}\rangle$). Later, $p$ partitions from $q$. $q$ detects this partition first and delivers the view $\langle 2, \{q\}\rangle$. When the slower process $p$ also detects the fluctuation in the network connectivity and activates the membership protocol, the network re-connects and both processes deliver $\langle 3, \{p, q\}\rangle$. From $p$'s point of view, the intersection of $\langle 3, \{p, q\}\rangle$ and the preceding view is $\{p, q\}$, although Virtual Synchrony does not guarantee that they deliver the same set of messages in view $\langle 1, \{p, q\}\rangle$.

[5]The Virtual Synchrony property should not be confused with the Strong, Weak, Optimistic and Extended Virtual Synchrony Models, although all of these models include this property.

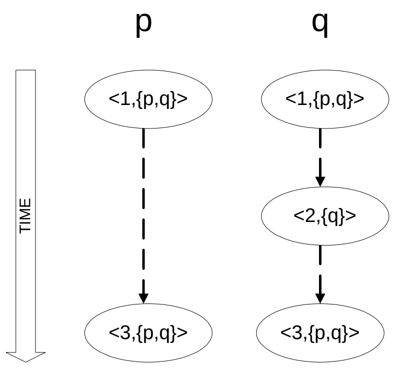

Figure 3: A possible scenario with a partitionable GCS.

Unfortunately, Virtual Synchrony is an "external observer" property. If the membership service at $p$ does not provide information about views installed at other processes in $V$, $p$ cannot deduce $T$ (as defined above) solely from $V$ and $V'$, and cannot always know whether the hypothesis of Virtual Synchrony holds. Additional information is required to allow processes to locally deduce when state transfer is indeed not needed. In the sections below, we present two possible solutions to this shortcoming.

### 4.3.1 Exploiting Virtual Synchrony using the Transitional Set

The *transitional set* contains information that allows processes to locally determine whether the hypothesis of Virtual Synchrony applies or a state transfer is required. Different transitional sets may be delivered with the same view at different processes.

The following property specifies the requirements imposed on the transitional set:

**Property 4.6 (Transitional Set)**

1. *If process $p$ installs a view $V$ in (previous) view $V'$, then the transitional set for view $V$ at process $p$ is a subset of the intersection between the member sets of $V$ and $V'$. Formally:*
   $t_i = \textbf{view\_chng}(p, V, T) \wedge \mathit{viewof}(t_i, p) = V' \;\Rightarrow\; T \subseteq V.\mathit{members} \cap V'.\mathit{members}$

2. *If two processes $p$ and $q$ install the same view, then $q$ is included in $p$'s transitional set for this view if and only if $p$'s previous view was also identical to $q$'s previous view. Formally:*
   $t_i = \textbf{view\_chng}(p, V, T) \wedge \mathit{viewof}(t_i, p) = V' \;\wedge\; \mathit{installs\_in}(q, V, V'') \;\Rightarrow\; (q \in T \Leftrightarrow\; V' = V'')$

Consider the example of Figure 3 above, there, $p$'s transitional set is $\{p\}$.

Note: The transitional set is not uniquely defined by Property 4.6, since if a process $p$ in $V.\mathit{members} \cap V'.\mathit{members}$ does not install $V'$, Property 4.6 does not specify whether $p$ is included in transitional sets of other processes in $V.\mathit{members} \cap V'.\mathit{members}$.

When used in conjunction with Virtual Synchrony, the transitional set delivered at a process $p$ reflects the set of processes whose states are identical to $p$'s state. Thus, applications can exploit this information in order to determine whether state transfer is needed as explained above (please see [ACDV97] for more details).

The transitional set is easily computed without additional communication over what is normally used for installing views: Since every membership protocol exchanges messages while agreeing on a new view, each process can piggyback its previous view on a membership protocol message. The transitional set is easily deduced from this information.

The transitional set was first introduced as part of the *transitional view* in the Extended Virtual Synchrony model [MAMSA94]. This model is implemented in Transis [ADKM92b, DM96] and Totem [AMMS$^+$95]. Later, Babaoğlu *et al.* [BBD96] introduced the notion of an *enriched view*, which, among other things, conveys information regarding the previous view of each of its members. Likewise, the views delivered by the membership service of [CS95] also convey the previous view of every view member. The transitional set can be deduced from these views. The transitional set also appears in the specifications of [ACDV97, KK99].

### 4.3.2 Exploiting Virtual Synchrony with Agreement on Successors

The following property provides an alternative to transitional sets:

**Property 4.7 (Agreement on Successors)** *If a process $p$ installs view $V$ in view $V'$, and if some process $q$ also installs $V$ and $q$ is a member of $V'$ then $q$ also installs $V$ in $V'$. Formally:* $\mathit{installs\_in}(p, V, V') \ \wedge\ \mathit{installs}(q, V)\ \wedge\ q \in V'.\mathit{members}\ \Rightarrow\ \mathit{installs\_in}(q, V, V')$

Property 4.7 (Agreement on Successors) holds in Horus [FV97a, FV97b], Ensemble [HLvR99] and Relacs [BDM95][6]. It guarantees that every member in the intersection of $p$'s current view and $p$'s previous view is also coming from the same previous view. Therefore, the hypothesis of Virtual Synchrony applies for all the members of this intersection.

Unfortunately, this property implies a deliberate exclusion of live and connected processes from the current view. Hence, it requires processing of an extra view. Though this exclusion does not violate our membership liveness property (cf. Property 10.1.1 (Membership Precision) in Section 10), it does contradict the "best-effort" principle discussed in Section 1.3.

## 5 Safe Messages

Distributed applications often require "all or nothing" semantics, that is, either all the processes deliver a message or that none of them do so. Unfortunately, "all or nothing" semantics are impossible to achieve in distributed systems in which messages may be lost. As an approximation to "all or nothing" semantics, the EVS model [MAMSA94] introduced the concept of *safe* messages. A safe message $m$ is received by the application at process $p$ only when $p$'s GCS knows that the message is *stable*, that is, all members of the current view have received this message from the network. In this case, each member of this view will deliver the message unless it crashes, even if the network partitions at that point. These "approximated" semantics are called *Safe Delivery* in [MAMSA94] and *Total Resiliency* in [WMK95].

In this paper we follow the approach of [FLS97] which decouples notification of message stability from its delivery. Thus, instead of deferring delivery until the message becomes stable, messages are delivered without additional delay. This delivery is augmented with a later delivery of *safe indications*. This approach also changes the semantics of safe indications to refer to application-level stability as opposed to network level. In other words, a message is stable when all members of the current view have delivered this message to the application (and not just received it from the network).

In our formalization, safe indications are conveyed using **safe_prefix** events which indicate that a prefix of the sequence of messages received in a certain view is stable: A **safe_prefix**$(p, m)$ event

[6]In [HLvR99] and [BDM95], a stronger property is stated – when two processes install the same view, their previous views are either identical or disjoint. The stronger property implies that Agreement on Successors holds.

indicates to $p$ that message $m$ is stable, as well as all the messages that $p$ received before $m$ in the same view as $m$. We define three new shorthand predicates in Table 2 below.

| | |
|---|---|
| Process $p$ receives message $m$ before message $m'$: | |
| $\mathit{recv\_before}(p, m, m')$ | $\overset{\text{def}}{=} \exists i \exists j : t_i = \mathbf{recv}(p, m) \wedge t_j = \mathbf{recv}(\mathbf{p}, \mathbf{m}') \wedge i < j$ |
| Process $p$ receives message $m$ before message $m'$, both of them in view $V$: | |
| $\mathit{recv\_before\_in}(p, m, m', V)$ | $\overset{\text{def}}{=} \exists i \exists j : t_i = \mathbf{recv}(p, m) \wedge t_j = \mathbf{recv}(\mathbf{p}, \mathbf{m}') \wedge$ $\mathit{viewof}(t_i, p) = \mathit{viewof}(t_j, p) = V \wedge i < j$ |
| A message $m$ received in a view $V$ is indicated as safe at process $p$: | |
| $\mathit{indicated\_safe}(p, m, V)$ | $\overset{\text{def}}{=} (\mathit{receives\_in}(p, m, V) \wedge \exists i : t_i = \mathbf{safe\_prefix}(p, m)) \vee$ $(\exists m' : t_i = \mathbf{safe\_prefix}(p, m') \wedge \mathit{recv\_before\_in}(p, m, m', V))$ |
| Message $m$ is stable in view $V$: | |
| $\mathit{stable}(m, V)$ | $\overset{\text{def}}{=} \forall p \in V.\mathit{members} : \mathit{receives}(p, m)$ |

Table 2: Predicate definitions for safe messages.

The next property requires that a message is *indicated* as safe only if it is stable, that is, delivered to all the members of the current view.

**Property 5.1 (Safe Indication Prefix)** *If a message is indicated as safe, then it is stable in the view in which it was received. Formally:*
$\mathit{indicated\_safe}(p, m, V) \Rightarrow \mathit{stable}(m, V)$

Note that Property 5.1 does not require that a message be stable *before* it is indicated as safe. However, since processes may crash at any point in the execution, there is no way for a system to guarantee that a message be delivered at all the members of the current view unless it was already delivered to them. Thus, any actual system that provides safe indications will be forced to wait until a message $m$ is stable before indicating $m$ to be safe.

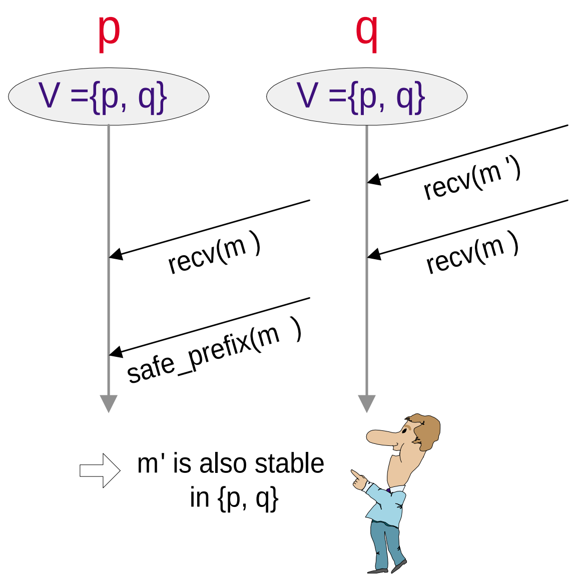

Figure 4: The Safe Indication Prefix property.

Consistent replication applications (for example, [KD96, Ami95]) often use safe indications in conjunction with a totally ordered multicast service that delivers messages in the same order at all the processes that deliver them (cf. Property 6.5 in Section 6.3). It is useful for such applications to receive safe indications that guarantee that all the members of a view $V$ receive the same prefix of messages in $V$ up to the indicated message. We state this requirement in Property 5.2 (Safe Indication Reliable Prefix) below.

**Property 5.2 (Safe Indication Reliable Prefix)** *If message $m$ is indicated as safe at some process $p$ and $m$ is also delivered by process $q$ in view $V$, then every message delivered at $q$ before $m$ in $V$ is also stable in $V$. Formally:*
$\mathit{indicated\_safe}(p, m, V) \wedge \mathit{recv\_before\_in}(q, m', m, V) \Rightarrow \mathit{stable}(m', V)$

This property is illustrated in Figure 4. In conjunction with totally ordered delivery it guarantees that all the members of $V$ receive the same sequence of messages in $V$ up to $m$.

Safe indications are closely related to garbage collection: if a message is stable, then a GCS will no longer need to keep it in its internal buffer. Since all GCSs attempt to recover from message losses and all GCSs perform garbage collection, they all internally keep track of message stability. However, some systems provide applications with safe indications or safe messages and some do not. Examples of systems that do provide this service include the *Safe* messages of Totem [AMMS$^+$95, MAMSA94] and Transis [ADKM92b], the *Totally Resilient* QoS level of RMP [WMK95], the *atomic, tight* and *delta* QoS levels of xAMp [RV92] and the *Uniform* multicast of Phoenix [MFSW95]. Safe delivery is also guaranteed by Horus if one uses the ORDER layer above the STABLE layer [vRHB94].

A process knows that a message is stable as soon as it learns that all other members of the view have acknowledged its reception. Usually such acknowledgments are given by the GCS level. However, in Horus [vRHB94] it is the responsibility of the application to acknowledge message reception. This approach may require extra communication and may be more complex, but it may yield more flexible and powerful semantics. Horus does not deliver safe prefix notifications. Instead, the Horus STABLE layer maintains a more general *stability matrix* at each process. The $(i, j)$ entry of the matrix stores the number of messages sent by $i$ that have been acknowledged by $j$. This matrix is accessible by the application, which then can deduce the information provided by safe prefix indications. The application can also learn about *k-stability* (cf. [vRHB94]), that is, $k$ members have received the message.

Some applications require a weaker degree of atomicity. For example, in quorum based systems it could be enough to defer delivery until the majority of the processes have the message. This is guaranteed by *Majority Resilient* QoS level of RMP [WMK95]. The *N resilient* QoS level of RMP [WMK95] and *atLeastN* QoS level of xAMp [RV92] guarantee that if a process receives a message, then at least N processes will also receive this message unless they crash. Here N is a service parameter.

## 6 Ordering and Reliability Properties

Group communication systems typically provide different group multicast services with a variety of ordering and reliability guarantees. Here we describe the service types most commonly provided by GCSs: FIFO, Causal and (several variants of) Totally ordered[7] multicast. These service types involve two kinds of guarantees: ordering and reliability. The ordering properties restrict the

[7]Totally ordered multicast is sometimes called atomic or agreed multicast.

order in which messages are delivered, and the reliability properties complement the corresponding ordering properties by prohibiting gaps or "holes" in the corresponding order within views.

Since reliability guarantees restrict message loss within a view, they are useful only when provided in conjunction with certain properties that synchronize view delivery with message delivery, e.g., Property 4.3 (Sending View Delivery). Similar properties may be stated for the OVS and WVS models (cf. Section 4.2.3). Systems that provide only Same View Delivery without Sending View Delivery, OVS or WVS (for example, Transis) typically implement a "heavy-weight" service that provides Sending View Delivery and the corresponding reliability property, and compose this service with an asynchronous FIFO buffer as demonstrated in Figure 2 in Section 4.2.2, thus yielding weaker semantics (satisfying only Same View Delivery).

Some GCSs (for example, Isis) provide different primitives for sending messages of different service types; others (for example, Transis) provide one *send* primitive and allow the application to tag the message sent with the requested service type; while in other systems (for example, Horus and Ensemble), a different protocol stack is constructed for each service type, and a communication end-point (associated with one such stack) provides exactly one service type.

In this section, we state all of the properties in terms of the *send* primitive. These properties are satisfied only for messages sent with some service types and not for other service types provided by the same GCS. In Sections 6.1, 6.2, and 6.3 we discuss the case that all the messages are sent with the same service type: FIFO in Section 6.1, Causal in Section 6.2, and Totally ordered in Section 6.3. In Section 6.4 we discuss the case that different messages are sent with different service types. In Section 6.5 we discuss issues that arise when ordering semantics need to be preserved across multicast groups.

## 6.1 FIFO multicast

The FIFO service type guarantees that messages from the same sender arrive in the order in which they were sent (Property 6.1), and that there are no gaps in the FIFO order within views (Property 6.2).

**Property 6.1 (FIFO Delivery)** *If a process p sends two messages, then these messages are received in the order in which they were sent at every process that receives both. Formally:*
$t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p, m') \ \wedge \ i < j \ \wedge \ t_k = \mathbf{recv}(q, m) \wedge t_l = \mathbf{recv}(q, m') \ \Rightarrow k < l$

**Property 6.2 (Reliable FIFO)** *If process p sends message m before message m′ in the same view V, then any process q that receives m′ receives m as well. Formally:*
$t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p, m') \ \wedge \ i < j \ \wedge \ \mathit{viewof}(t_i, p) = \mathit{viewof}(t_j, p) \ \wedge \ \mathit{receives}(q, m') \ \Rightarrow \mathit{receives}(q, m)$

Several group communication systems (for example, Ensemble [HvR96], Horus [FvR95] and RMP [WMK95]) provide a reliable FIFO service type which satisfies these two properties and does not impose additional ordering constraints. xAMp [RV92] provide several service levels that satisfy Requirement 6.1 but vary by their reliability guarantees.

This service type is a basic building block; it is useful for constructing higher level services, for example, Totally ordered multicast protocols [EMS95, CHD98] are often constructed over a reliable FIFO service.

## 6.2 Causal multicast

The Causal order (first defined in [Lam78]) extends the FIFO order by requiring that a response $m'$ to a message $m$ is always delivered after the delivery of $m$. Formally, the causal order of events is defined recursively as follows:

$$\boxed{\begin{aligned} t_i \to t_j \overset{\text{def}}{=} \; & (pid(t_i) = pid(t_j) \wedge j \geq i) \vee \;\; (t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{recv}(q, m)) \vee \\ & (\exists k : t_i \to t_k \wedge t_k \to t_j) \end{aligned}}$$

Table 3: Causal order definition.

The Causal service type guarantees that messages arrive in Causal order (Property 6.3), and that there are no "causal holes" within each view (Property 6.4).

**Property 6.3 (Causal Delivery)** *If two messages $m$ and $m'$ are sent so that $m$ causally precedes $m'$, then every process that receives both these messages, receives $m$ before $m'$. Formally:*
$t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p', m') \wedge t_i \to t_j \wedge t_k = \mathbf{recv}(q, m) \wedge t_l = \mathbf{recv}(q, m') \;\Rightarrow k < l$

**Property 6.4 (Reliable Causal)** *If message $m$ causally precedes a message $m'$, and both are sent in the same view, then any process $q$ that receives $m'$ receives $m$ as well. Formally:*
$t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p', m') \;\wedge\; t_i \to t_j \;\wedge\; viewof(t_i, p) = viewof(t_j, p') \;\wedge\; receives(q, m') \;\Rightarrow$ $receives(q, m)$

The CBCAST (Causal Broadcast) primitive of Isis [BJ87] was perhaps the first implementation of (Reliable) Causal multicast (satisfying Properties 6.3 and 6.4). Other GCSs that provide this service level include: Transis [DMS95, ADKM92b, DM96], Horus [vRBM96], Newtop [EMS95] and xAMp [RV92].

## 6.3 Totally ordered multicast

Group communication systems usually provide a Totally ordered (atomic, agreed) service type which extends the Causal service type. However, GCSs vary in the semantics that their Totally ordered multicast service provides. In Section 6.3.1 below, we discuss two possible ordering semantics: *Strong Total Order* (Property 6.5) and *Weak Total Order* (Property 6.6).

In addition to the ordering semantics, Totally ordered multicast provides a reliability guarantee. In practically all existing GCSs (examples include: Transis, Horus, Newtop, xAMp, Totem, Phoenix and RMP), the reliability guarantee for Totally ordered multicast is Property 6.4 (Reliable Causal) above. In Section 6.3.2 below, we discuss a stronger alternative (Reliable Total Order).

In Table 4 we define an order on totally ordered messages using a one-to-one order function from $\mathcal{M}$ to the set of natural numbers; we call this function a *timestamp function*:

| A timestamp function is a one-to-one function from $\mathcal{M}$ to the set of natural numbers: |
|---|
| $TS\_function(f) \overset{\text{def}}{=} f : \mathcal{M} \to \mathcal{N} \;\wedge\; f(m) = f(m') \;\Rightarrow\; m = m'$ |

Table 4: Timestamp function definition.

### 6.3.1 Strong and Weak Total Order

Wilhelm and Schiper [WS95] introduce a classification of totally order multicast. In particular, this work defines *strong* and *weak* total order in the context of a primary component membership service. Here we extend these definitions to a partitionable environment.

Strong Total Order guarantees that messages are delivered in the same order at all the process that deliver them:

**Property 6.5 (Strong Total Order)** *There exists a timestamp function $f$ such that messages are received at all the processes in an order consistent with $f$. Formally:*
$\exists f : \mathit{TS\_function}(f) \ \wedge \ \forall p \forall m \forall m' : \mathit{recv\_before}(p, m, m') \ \Rightarrow \ f(m) < f(m')$

Note that the timestamp function is an abstract function that merely exists: we do not require that the timestamp values be conveyed to the application. However, some applications (for example, the replication algorithm of [KD96]) do require that timestamps be available to them. The ATOP algorithm [CHD98, Cho97a] which implements totally ordered multicast in Transis conveys such timestamps to its application.

Many group communication systems implement a weaker form of totally ordered multicast that allows processes to disagree upon the order of messages in case they disconnect from each other. Weak Total Order guarantees that processes that remain connected deliver messages in the same order.

**Property 6.6 (Weak Total Order)** *For every pair of views $V$ and $V'$ there is a timestamp function $f$ such that every process $p$ that installs $V$ in $V'$ receives messages in $V'$ in an order consistent with $f$ Formally:*

$$\begin{array}{ll} \forall V \forall V' \exists f : & \mathit{TS\_function}(f) \wedge \\ & (\forall p \forall m \forall m' : \mathit{installs\_in}(p, V, V') \ \wedge \ \mathit{recv\_before\_in}(p, m, m', V') \ \Rightarrow \ f(m) < f(m')) \end{array}$$

Applications that exploit GCSs for consistent replication require that processes agree upon the order of messages even in case they disconnect from each other [Ami95, KD96, FLS97]; otherwise, updates may be applied in a different order in replica that disconnect from each other, violating consistency. This feature is guaranteed only by Strong Total Order (Property 6.5) and not by Weak Total Order. Strong Total Order is provided by Totem [AMMS$^+$95, MMSA$^+$96] and by some of the implementations of totally ordered multicast in Transis ("all-ack" and ATOP [Cho97a, CHD98, DM95]), Phoenix [MFSW95], RMP [WMK95] (the totally resilient QoS level) and Horus [FvR97].

However, many GCSs provide a Weak totally ordered multicast service, for example, the ABCAST (Atomic Broadcast) primitive of Isis [BvR94], similar primitives in Amoeba [KT96], Newtop [EMS95] and xAMp [RV92], the ToTo [DKM93] protocol implemented in Transis and certain implementations of totally ordered multicast in Phoenix [MFSW95], RMP [WMK95] and Horus [FvR97].

The totally ordered multicast services, Strong or Weak, in all of the GCSs listed above guarantee that messages arrive in Causal order (Property 6.3), and that there are no "causal holes" within each view (Property 6.4).

### 6.3.2 Reliable Total Order

The Reliable Total Order Property requires processes to deliver a prefix of a common sequence of messages within each view:

**Property 6.7 (Reliable Total Order)** *There exists a timestamp function $f$ such that if a process $q$ receives a message $m'$, and messages $m$ and $m'$ were sent in the same view, and $f(m) < f(m')$, then $q$ receives $m$ as well. Formally:*

$$\begin{aligned}\exists f:\ & TS\_function(f) \wedge \\ & (\forall m \forall m' \forall p \forall p' \forall q : sends\_in(p, m, V) \wedge sends\_in(p', m', V) \wedge receives(q, m') \wedge f(m) < f(m') \\ & \ \Rightarrow\ receives(q, m))\end{aligned}$$

In the Appendix, we prove Lemma A.1 which asserts that Properties 6.7 (Reliable Total Order), 6.5 (Strong Total Order), 6.2 (Reliable FIFO) and 6.1 (FIFO delivery) along with Property 4.3 (Sending View Delivery) and the basic Properties 4.1 (Delivery Integrity), 3.2 (Local Monotonicity) and 3.3 (Initial View Event) imply Properties 6.4 (Reliable Causal) and 6.3 (Causal).

Unfortunately, implementing Reliable Total Order contradicts the "best-effort" principle since it forces the GCS to either deliberately discard messages or to prohibit concurrent sending of messages from different processes. Therefore, no GCS we are aware of guarantees Requirement 6.7. The only specifications that require Reliable Total Order are those of [FLS97].

The Reliable Total Order property is exploited by the replication application in [FLS97]; it guarantees that operations will be applied to the database in a consistent order without gaps. However, the application in [FLS97] could have been satisfied with a weaker property: In [KD96, Ami95] a similar application exploits Property 5.2 (Safe Indication Reliable Prefix) which uses *safe prefix indications* (presented in Section 5) to denote the end of the prefix in which there are no gaps in the total order. This property is weaker, since it does not preclude delivery of totally ordered messages with gaps, as long as these message will never become safe (or stable). Since in all of these applications [KD96, FLS97, Ami95] updates are not applied to the database before they are safe (stable), the weaker property is sufficient to guarantee consistency.

A similar approach was taken in [FV97a], which uses explicit *Reliable Totally Ordered Prefix Indications* to denote the end of the prefix in which there are no gaps in the total order.

## 6.4 Order constraints for messages of different types

Systems that provide more than one ordering type need to specify the delivery semantics (order constraints) of messages with different types. For example, should Causal messages be totally ordered with respect to totally ordered messages?

Wilhelm and Schiper [WS95] discuss three possible semantics in the context of weak and strong total order. However, these semantics can be generalized for the case of two messages $m_1$ and $m_2$ with any two different ordering semantics $O_1$ and $O_2$ such that $O_2$ implies $O_1$:

- *unordered*: there no ordering constraints on delivery of $m_1$ and $m_2$
- *weak incorporated*: $m_1$ and $m_2$ deliveries should satisfy $O_1$
- *strong incorporated*: $m_1$ and $m_2$ are delivered according to $O_2$

For example, RMP [WMK95] supports weak incorporated semantics between any two messages of different service levels. Isis [BJ87] gives weak incorporated semantics between messages sent by ABCAST and CBCAST multicast primitives. However, this system has another total order multicast primitive, GBCAST (Global Broadcast), so that messages sent by GBCAST and CBCAST primitives are ordered according to strong incorporated semantics. Isis' successors, Horus and Ensemble, do not allow messages of different types to be sent in the same group, hence they provide unordered semantics for messages of different types.

Transis [ADKM92b, DM96, Cho97b] may be configured to use one of several protocols providing totally ordered multicast. The more efficient ATOP protocol [CHD98, Cho97a] guarantees only weak incorporated semantics between a Reliable Causal message and a Strong Totally ordered message. On the other hand, the "all-ack" protocol [DM95, Cho97b] guarantees strong incorporated semantics between messages of these two types, but it incurs longer delivery latency[8]. Highways [Ahu93] defines different types of "incorporated" semantics for Causal delivery and shows how they can be efficiently combined in a GCS.

## 6.5 Order constraints for multiple groups

Group communication systems generally allow processes to join multiple groups. When a message is sent, the sender indicates which group (or groups) the message is being sent to. Messages sent in a given group are received only by the members of that group. Views are also associated with groups – a view reflects the set of processes that are currently members of a given group. The discussion above focuses on ordering semantics within a single multicast group. When multicast groups overlap, one has to determine the ordering semantics of messages that are sent in different groups.

Atomic Multicast (cf. [GS97b]) semantics require messages sent in different groups to be delivered in the same total order at all their destinations. For example, assume that processes $p$ and $q$ are both members of two different multicast groups $g1$ and $g2$. Assume also that message $m1$ is sent in group $g1$ and message $m2$ is sent in group $g2$, and that $p$ delivers $m1$ before $m2$. Atomic Multicast requires that $q$ also deliver $m1$ before $m2$. Guerraoui and Schiper [GS97b] prove that Atomic Multicast is costly: it requires sending messages to additional processes that are not members of the group the message is sent to.

The Isis system does not provide Atomic Multicast: totally ordered messages sent to different groups may be delivered in different orders at different recipients. Other GCSs (for example, Transis and Totem) provide Atomic Multicast by using a *light-weight* groups approach, in which all the messages are sent to a set of *daemons* which totally order messages of all the groups. The daemons forward each message to the members of the light-weight group in which the message was sent.

Horus and Ensemble provide users with the flexibility to chose whether Atomic Multicast will be provided by constructing different protocol stacks: If Atomic Multicast is desired, a light-weight group layer is used above the total order layer in the stack. Thus, messages are first sent to the members of the heavy-weight group where they are totally ordered and then they are multiplexed to the different groups. If Atomic Multicast is not desired, the light-weight group layer is stacked below the total order layer, and messages are totally ordered in their destination groups.

GCSs that use a light-weight group structure typically allow users to send a message to multiple light-weight groups. This service is implemented by sending messages to the heavy-weight (or daemon) group, and then multiplexing messages to the appropriate light-weight group. Johnson et al. [JJS99] suggest a different approach to sending a message to multiple groups. In their approach, messages are pipelined through a sequence of groups. Such pipelining preserves the order semantics across groups as long as groups do not overlap.

Unlike total order, virtually all group communication systems provide causally ordered multicast (cf. [BSS91, KS98]), that is, preserve the causality of messages sent in different groups. However, recently, Kalantar and Birman [KB99] have shown that causally ordered multicast is also costly. They show that such multicast leads to bursty behavior and to latencies three times longer than

[8]The totally ordered multicast service that complies with strong incorporated semantics is called "Global Agreed" in [Cho97b].

the latency for delivering messages without such order constraints.

# Part III
# Liveness Properties of Group Communication Services

## 7 Introduction

In this part of the paper we specify GCS liveness properties. Liveness is an important complement to safety, since without requiring liveness, safety properties can be satisfied by trivial implementations that do nothing. However, it is challenging to specify GCS liveness properties that are sufficiently weak to be implementable and yet are strong enough to be non-trivial.

In order to specify meaningful liveness properties, we envision an ideal GCS, and try to capture its ideal behavior in our liveness properties. Ideally, one would like a membership service to be *precise*, that is, to deliver a view that correctly reflects the network situation to all the live processes; likewise, one would want a multicast service to deliver all the messages sent in this "correct" view to all the view members. However, how can one argue about the "correct" network situation if this situation is constantly changing? We observe that the liveness of a GCS is bound to depend on the behavior of the underlying network. Therefore, unless we strengthen the model, it is not feasible to require that the GCS be live in every execution. The only way to specify useful liveness properties without strengthening the communication model is to make these properties *conditional* on the underlying network behavior[9].

In this paper, we require that the GCS be live only in executions in which the network eventually *stabilizes*. Intuitively, we say that the network eventually stabilizes if from some point onward no processes crash or recover, communication is symmetric and transitive, and no changes occur in the network connectivity. (This definition is made formal in Section 8). In such cases, we would like the membership service to be precise (i.e., to deliver a view that correctly reflects the network situation to all the live processes).

Unfortunately, it is impossible to implement such a precise membership service in purely asynchronous environments prone to failures. In Section 9 we prove Lemma 9.1 which asserts that a precise membership service is as strong as an *eventually perfect failure detector ($\diamond P$)* (formally defined in Section 8.3.1), which is known to be non-implementable in our environment. Our impossibility result is not surprising. In fact, [CHTCB96] prove that even a very weak definition of group membership is impossible to implement in asynchronous failure-prone environments.

In order to circumvent this impossibility result, we assume that the GCS uses an external failure detector and require the liveness properties to hold only in executions in which the failure detector behaves like an eventually perfect one. Similar assumptions were also proposed in [SR93, MS94, BDM97, HS95]; please see detailed discussion in Section 10.

It is important to note that although our liveness properties are guaranteed to hold only in certain executions, the conditions on these executions are *external* to the GCS implementation. Thus, in order to satisfy our liveness requirements, a group membership implementation has to attempt to be precise in every execution as it cannot know whether in a particular execution there is a stable component and whether the failure detector behaves like an eventually perfect one.

In order to specify conditional liveness properties, we need to refine the model described in Section 2. To this end, in Section 8, we extend the external signature of the GCS and specify the underlying network and failure detector as part of the environment. We also define what it means

[9]Conditional liveness specifications of GCSs also appear in [FLS97, CS95, KSMD99].

for a failure detector to "behave like $\diamond P$". In Section 9 we prove that in this model it is impossible to implement our desirable liveness properties unless we require that the failure detector behave like $\diamond P$. Finally, in Section 10, we state the liveness properties and survey related work.

## 8 Refining the Model to Reason about Liveness

In this section we extend the model described in Section 2. Since the liveness of a GCS depends on the network conditions and failure detector output, we extend the external signature presented in Section 2 by adding actions that represent the GCS' interaction with the network and failure detector. Thus, an automaton with the external signature presented in Section 2 that satisfies the GCS safety properties may be seen as a *composition* of two automata: a GCS-liveness automaton with the extended signature presented in this section, and a Network and Failure Detector automaton. This composition is depicted in Figure 5.

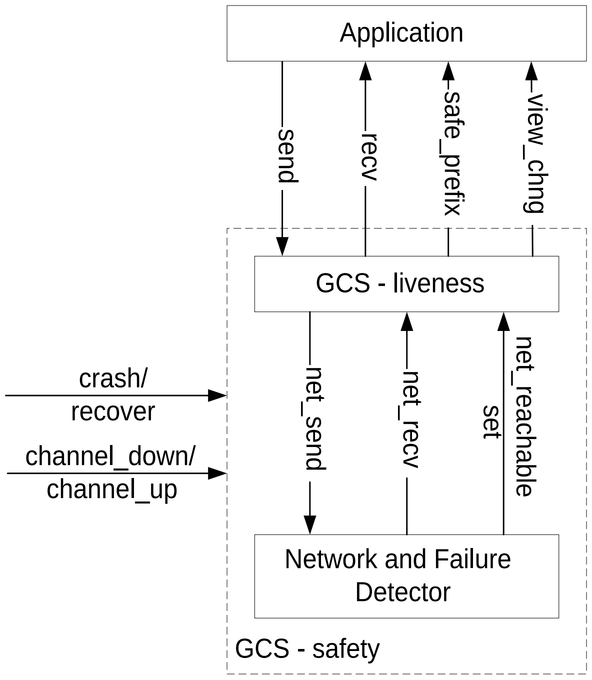

Figure 5: Extending the external signature of the GCS to specify liveness.

The network is modeled as a set of *channels* that connect every pair of processes in the system[10]. We assume that the underlying network provides an unreliable datagram service. Messages may be lost, delivered out of order, or duplicated.

In Section 8.1 we present the extension to the GCS signature and some auxiliary definitions. In Section 8.2 we specify our assumptions on the network behavior. In Section 8.3 we formally define the prerequisites for our liveness properties, namely stable components and eventually perfect failure detectors.

[10]Note that a channel between two processes is not necessarily a direct link; it can be any network path connecting these processes.

## 8.1 Extending the GCS external signature

### Interaction with the environment

We augment the GCS's interaction with the environment by adding communication channel up and down actions which model changes in the connectivity from every process $p$ to every process $q$:

- input **channel_down**$(p, q), p, q \in \mathcal{P}$
- input **channel_up**$(p, q), p, q \in \mathcal{P}$

### Interaction with the network and failure detector

The GCS sends and receives messages via the underlying communication network, and also receives failure detection information from it:

- output **net_send**$(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- input **net_recv**$(p, m), p \in \mathcal{P}, m \in \mathcal{M}$
- input **net_reachable_set**$(p, S), p \in \mathcal{P}, S \in 2^{\mathcal{P}}$
  This action denotes that the failure detector at $p$ believes that the set of processes in $S$ (and only these processes) are currently connected to $p$. Until the first **net_reachable_set** occurs at $p$, the set of processes $p$ believes to be connected to it is undefined.

The mathematical model described in Section 2.2 is extended by adding the following to the **Actions** set:
$\{\textbf{channel\_down}(p, q) \mid p, q \in \mathcal{P}\} \cup \{\textbf{channel\_up}(p, q) \mid p, q \in \mathcal{P}\} \cup$
$\{\textbf{net\_send}(p, m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup \{\textbf{net\_recv}(p, m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\textbf{net\_reachable\_set}(p, S) \mid p \in \mathcal{P}, S \in 2^{\mathcal{P}}\}$

### Notation

We define some shorthand predicates which describe the network situation in Table 5 below. Note that according to these definitions, processes are initially alive and links are initially up.

| | | |
|---|---|---|
| Process $p$ is alive after the $i$th event in the trace: | | |
| $alive\_after(p, i)$ | $\overset{\text{def}}{=}$ | $(\nexists j : t_j = \textbf{crash}(p)) \vee (\exists j \leq i : t_j = \textbf{recover}(p) \wedge \nexists k > j : t_k = \textbf{crash}(p))$ |
| Process $p$ is crashed after the $i$th event in the trace: | | |
| $crashed\_after(p, i)$ | $\overset{\text{def}}{=}$ | $\exists j \leq i : t_j = \textbf{crash}(p) \wedge \nexists k > j : t_k = \textbf{recover}(p)$ |
| The channel from $p$ to $q$ is up after the $i$th event in the trace: | | |
| $up\_after(p, q, i)$ | $\overset{\text{def}}{=}$ | $(\nexists j : t_j = \textbf{channel\_down}(p, q)) \vee$ $(\exists j \leq i : t_j = \textbf{channel\_up}(p, q) \wedge \nexists k > j : t_k = \textbf{channel\_down}(p, q))$ |
| The channel from $p$ to $q$ is down after the $i$th event in the trace: | | |
| $down\_after(p, q, i)$ | $\overset{\text{def}}{=}$ | $\exists j \leq i : t_j = \textbf{channel\_down}(p, q) \wedge \nexists k > j : t_k = \textbf{channel\_up}(p, q)$ |

Table 5: Predicates describing the network situation.

## 8.2 Assumption: Live Network

We now state a liveness assumption on the network.

**Assumption 8.1 (Live Network)** *If there is a point in the execution after which two processes, p and q are alive and the channel from p to q is up, then from this point onward, every message sent by p eventually arrives at q. Formally:*
$alive\_after(p, i) \ \wedge \ alive\_after(q, i) \ \wedge \ up\_after(p, q, i) \ \wedge \ t_i = \mathbf{net\_send}(p, m) \ \Rightarrow$
$\exists j : t_j = \mathbf{net\_receive}(q, m)$

## 8.3 Conditions for liveness

As explained above, our liveness guarantees are conditional: they require that the GCS be live only if a stable component eventually exists and the network behaves like an eventually perfect failure detector. We now formally define these conditions.

**Definition 8.1 (Stable Component)** *A stable component is a set of processes that are eventually alive and connected to each other and for which all the links to them from all other processes (that are not in the stable component) are down. Formally, $stable\_component(S), S \in 2^{\mathcal{P}}$ is defined as:*
$stable\_component(S) \overset{\text{def}}{=} \exists i \forall p \in S : (alive\_after(p, i) \ \wedge (\forall q \in S : up\_after(p, q, i)) \ \wedge$
$(\forall q \in \mathcal{P} \setminus S : down\_after(q, p, i) \ \vee \ crashed\_after(q, i)))$

Note that the existence of a stable component implies that within the stable component communication is eventually symmetric and transitive. We do not assume that the communication is always symmetric and transitive as part of the model. This is only a precondition for the liveness properties and for the failure detector's completeness and eventual accuracy properties stated in the next section. If the communication over the channels is not eventually stable, symmetric and transitive, the GCS is not required to be live and Definition 8.2 below imposes no restrictions on the failure detector's behavior.

It is common to assume transitivity, though it is not necessary. For example, Phoenix [MS94] does not assume transitivity, but instead, it ensures eventual transitivity of communication by relaying messages. It is more common to assume that communication is symmetric. However, in wide area networks prone to various types of failures, lack of symmetry may occasionally occur. Such absence of symmetry is difficult to overcome. Indeed, existing GCSs do not overcome absence of symmetry and all the specifications that we are aware of do not require membership to be precise in such cases.

### 8.3.1 Eventually perfect failure detectors

An eventually perfect failure detector is a failure detector that eventually stops making mistakes, that is, there is a time after which it correctly reflects the network situation. Since eventually perfect failure detectors are not implementable in asynchronous environments, we do not assume that our environment contains such a failure detector. Instead, we classify execution traces in which the failure detector *behaves like* an eventually perfect failure detector, and require that the GCS be live in such executions.

**Definition 8.2 (Eventually perfect-like trace)** *The failure detector behaves like $\diamond P$ in a given trace if for every stable component $S$, and for every process $p \in S$, the reachable set reported to $p$ by the failure detector is eventually $S$. Formally:*

$\Diamond P - like \overset{\text{def}}{=} \forall S : stable\_component(S) \Rightarrow \forall p \in S : \exists i : t_i = \mathbf{net\_reachable\_set}(p, S) \ \wedge$
$\not\exists S' \neq S : \exists j > i : t_j = \mathbf{net\_reachable\_set}(p, S')$

Note that if no stable component exists, Definition 8.2 imposes no restrictions on the failure detector's behavior.

**Definition 8.3 (Eventually perfect failure detector)** *An eventually perfect failure detector is a failure detector which behaves like $\Diamond P$ in every trace.*

Chandra and Toueg [CT96] define several classes of unreliable failure detectors which are strong enough to solve different agreement problems in fail-stop asynchronous environments. It is easy to see that, when restricted to a fail-stop model, our definition of $\Diamond P$ coincides with the one in [CT96], since in every execution in the fail-stop model all the correct processes form a stable component (once the last faulty process fails). Since it is impossible to implement $\Diamond P$ in the fail-stop model, it is also impossible to implement eventually perfect failure detectors as defined above in the asynchronous model of this paper.

Although it is impossible to implement eventually perfect failure detectors in truly asynchronous failure-prone environments, in practical networks, communication tends to be stable and timely during long periods. Time-out based failure detectors can be tuned to behave like eventually perfect ones during such periods. Hence, specifications that require liveness only in executions in which the failure detector behaves like an eventually perfect one are useful for practical systems.

The definition of eventually perfect failure detectors is extended to partitionable environments in [DFKM96, BDM97]. The definitions presented herein are similar to those of [DFKM96, BDM97] but not identical. The main difference is that our definition of stable components is stated explicitly in terms of **channel_down** and **channel_up** events, whereas the models in [DFKM96, BDM97] do not include such events, and connectivity (reachability) is defined in terms of whether the last messages sent on a channel reaches its destination or not.

## 9 Precise Membership is as Strong as $\Diamond P$

Having defined eventually perfect failure detectors in our models, we now justify their use as prerequisites for our liveness specifications. We focus on liveness of the membership service, since live membership is the basis for a live GCS. We show that a precise membership service is as strong as an eventually perfect failure detector. First, we have to define a precise membership service. We use the following auxiliary shorthand definition:

**Definition 9.1 (Last View)** *$V$ is the last view installed at process $p$ if $p$ installs view $V$ and does not install any views after $V$. Formally:*
$last\_view(p, V) \overset{\text{def}}{=} \exists i \exists T : t_i = \mathbf{view\_chng}(p, V, T) \wedge \not\exists j > i \ \exists T' \exists V' : t_j = \mathbf{view\_chng}(p, V', T')$

We now define a membership service to be precise if it delivers the same last view to all the members of a stable component. Note that this definition is *partitionable* as it requires members of all stable components to install views.

**Definition 9.2 (Precise Membership)** *A membership service is precise if it satisfies the following requirement: For every stable component $S$, there exists a view $V$ with the members set $S$ such that $V$ is the last view of every process $p$ in $S$. Formally:*
$stable\_component(S) \ \Rightarrow \ \exists V : V.members = S \ \wedge \ \forall p \in S : last\_view(p, V)$

We now prove that a precise membership service is as strong as an eventually perfect failure detector.

**Lemma 9.1** *Precise Membership is as strong as an eventually perfect failure detector.*

**Proof:** We provide a constructive proof of how an eventually perfect failure detector can be implemented using a Precise Membership service. Every process $p$ running the Precise Membership service generates **net_reachable_set** events as follows: Whenever a **view_chng**$(p, V, T)$ occurs, $p$ generates **net_reachable_set**$(p, V.members)$. We now show that if $p$'s membership service is precise, every generated trace is $\Diamond P - like$.

Note that if $p$ is not a member of a stable component, there are no restrictions on the failure detector's behavior. Therefore, we assume that there exists a stable component $S$ such that $p \in S$. In this case, Precise Membership guarantees that $p$ installs a last view $V$ with $V.members = S$. Thus, $p$ generates **net_reachable_set**$(p, S)$ and does not generate any **net_reachable_set** events afterwards, and thus satisfies the requirement for a $\Diamond P - like$ trace. ■

Note that the same result applies to the fail-stop model: In the fail-stop model, the set of correct processes forms a stable component in every execution. Thus, a precise membership service in the fail-stop model is required to deliver to all the correct processes a last view consisting of exactly the correct processes.

In the next section, we require the GCS liveness properties to hold only in executions in which the failure detector behaves like an eventually perfect one. Note that it is possible to implement a precise membership service using an eventually perfect failure detector, Section 10.1 surveys many examples of group communication systems that provide precise membership services when the failure detector they employ behaves like an eventually perfect one. GCS liveness is also specified using external failure detectors in [SR93, MS94, BDM97, HS95].

## 10 Liveness Properties

We now specify liveness properties for partitionable GCSs (cf. Section 3.2). Our liveness properties are partitionable in that they require a process $p$ to install a view in all traces in which $p$ is in a stable component and the failure detector behaves like $\Diamond P$, even if the stable component is not the primary one.

We state four partitionable liveness properties: Membership Precision, Multicast Liveness, Self Delivery and Safe Indication Liveness. Obviously, Safe Indication Liveness is only required if the system provides safe notifications (please see Section 5). All of these parts are conditional; they are required to hold in runs in which there exists a stable component $S$ and the failure detector behaves like $\Diamond P$.

**Property 10.1 (Liveness)** *If the failure detector behaves like $\Diamond P$, then for every stable component $S$, there exists a view $V$ with the members set $S$ such that the following four properties hold for every process $p$ in $S$. Formally:*
$\Diamond P - like \ \wedge \ stable\_component(S) \ \Rightarrow \ \exists V : V.members = S \ \wedge \ \forall p \in S :$

1. **Membership Precision** *$p$ installs view $V$ as its last view. Formally:*
   $last\_view(p, V)$

2. **Multicast Liveness** *Every message $p$ sends in $V$ is received by every process in $S$. Formally:*
   $sends\_in(p, m, V) \ \Rightarrow \ \forall q \in S : receives(q, m)$

3. **Self Delivery** *p delivers every message it sent in* any *view unless it crashed after sending it. Formally:*
   $t_i = \mathbf{send}(p,m) \wedge \nexists j > i : t_j = \mathbf{crash}(p) \Rightarrow \mathit{receives}(p,m)$

4. **Safe Indication Liveness** *Every message p sends in V is indicated as safe by every process in S. Formally:*
   $\mathit{sends\_in}(p,m,V) \Rightarrow \forall q \in S : \mathit{indicated\_safe}(q,m,V)$

Formally, stability of the connected component is required to last forever. Nevertheless, in practice, it only has to hold "long enough" for the membership protocol to execute and for the failure detector module to stabilize, as explained in [DLS88, GS97a]. However, we cannot explicitly introduce the bound on this time period in a fully asynchronous model, because its duration depends on external conditions such as message latency, process scheduling and processing time.

We do not include here a specification of liveness properties for a primary component GCS, since the liveness of a primary component membership service is dependent on the specific implementation: Note that primary component membership services block if they cannot form a primary view. For example, any primary component membership is bound to block if the network partitions into three minority components or if all the members of the latest view[11] crash. The exact scenarios in which a primary component does exist depends on the specific implementation and the policy it employs to guarantee Property 3.4 (Primary Component Membership).

## 10.1 Related work

### 10.1.1 Membership Precision

Precision is one of the most fundamental properties of a membership service. A group communication system is useless if its membership service is not precise at least to some extent.

GCSs typically exploit some failure detection mechanism based on time-outs or other methods (for example, please see [Vog96]) in order to detect conditions under which the membership protocol should be invoked. Furthermore, the failure detector provides an initial approximation of the view that the membership service would agree upon. If this approximation is precise, so is the output of the membership service. Thus, practically all of the existing GCSs satisfy Property 10.1.1 (Membership Precision), even if it does not explicitly appear in their specifications. A similar property explicitly appears in the specifications of [BDM97].

Phoenix [MS94] exploits a failure detector which is weaker than an eventually perfect one. Given the weaker failure detector, it guarantees progress but not precision: It guarantees that each invocation of the membership protocol will terminate. However, correct processes may be removed from the membership and forced to re-join infinitely many times, causing the membership to keep changing forever. We observe, however, that in executions in which the network eventually stabilizes and the underlying failure detector used by Phoenix behaves like an eventually perfect one, Phoenix also satisfies the Membership Precision property stated herein.

The specifications of [FLS97, CS95] guarantee precision of the membership service at periods during which the underlying network is stable and timely. These specifications guarantee the timeliness of the service and not just eventual termination. Of course, such guarantees can only be made when network message delivery and process scheduling are timely. The specifications are parameterized by timeouts suited for the underlying network and by constants that depend on the protocol implementation. Since in this paper we provide general specifications and do not focus on a specific protocol, we cannot provide such an analysis.

[11]Recall that in a primary component membership service views are totally ordered.

### 10.1.2 Multicast and Safe Indication Liveness

Like Membership Precision, Property 10.1.2 (Multicast Liveness) is satisfied by all existing GCS implementations, although it is not always explicitly formulated in the papers describing those systems. This property eliminates trivial GCS implementations that capriciously discard messages without delivering them. A similar multicast liveness property appears in [FLS97].

An alternative approach to formulating multicast liveness was undertaken in [FvR95, DMS95, BDM95], which require the following property:

**Property 10.2 (Termination of Delivery)** *If a process $p$ sends a message $m$ in a view $V$, then for each member $q$ of $V$, either $q$ delivers $m$, or $p$ installs a next view $V'$ in $V$. Formally:* $\mathit{sends\_in}(p, m, V) \land q \in V.\mathit{members} \;\Rightarrow\; \mathit{delivers}(q, m) \lor \exists V' : \mathit{installs\_in}(p, V', V)$

If Membership Precision holds, then Termination of Delivery implies Property 10.1.2 (Multicast Liveness). In addition, Property 10.2 (Termination of Delivery) requires that the membership service not block even when the network is unstable. However, we believe that this property is not particularly useful for applications: when the network is unstable, a membership service that satisfies this property will continuously install views without any guarantee to deliver messages in these views. Continuously installing new views at unstable times may only increase the load and lengthen the unstable period. Furthermore, any membership service that satisfies Property 10.2 is forced to deliver obsolete views, that is, views that are known to be changing soon, and thus violate the "best-effort" principle (cf. Section 1.3). However, most existing membership algorithms do satisfy Property 10.2 (Termination of Delivery). An exception is the membership service of [KSMD99] which does not install obsolete views.

In GCSs that provide primary component membership, message stability may be formulated as follows: If a process delivers a message in view $V$, then all *non-faulty* members of $V$ eventually deliver this message. This is called *Uniformity* in the Isis literature and in [SS93] and *Unanimity* in [RV92].

Property 10.1.3 (Self Delivery) requires that if the network eventually stabilizes, processes deliver all of their own messages unless they crash after sending them. Self Delivery complements Multicast Liveness by requiring that messages sent in any view be delivered (unless their sender crashes), and not just those sent in the last view.

All the GCSs that we are aware of satisfy Self Delivery, some examples are: Isis [BJ87], Transis [DMS95], Totem [AMMS$^+$95], Horus [FvR95] and Newtop [EMS95]. In RMP [WMK95] Self Delivery holds for all multicast services except the Unreliable one. However, this property does not hold in the specifications of [FLS97] which discard "left over" messages upon membership changes.

Some specifications (for example, [MAMSA94]) require Self Delivery to hold in all executions, not just those in which the network eventually stabilizes. Since the GCS cannot deduce whether stability holds in a certain execution, these two formulations of Self Delivery are essentially equivalent.

Property 10.1.4 (Safe Indication Liveness) appears only in the specification of [FLS97] as this is the only work that explicitly introduces the notion of safe indications.

## Part IV

# Conclusions

## 11 Summary

We have presented a comprehensive set of specifications which may be combined to represent the guarantees of most existing GCSs. We have specified clear and rigorous properties formalized as trace properties of I/O automata. In light of these specifications, we have surveyed and analyzed over thirty published specifications which cover a dozen leading GCSs. We have correlated the terminology used in different papers to our terminology.

We have seen that the main components of a GCS are the membership and multicast services. In Table 6, we summarize the safety properties of the membership and multicast services, making a distinction between basic properties and optional ones.

| Basic Properties | | Optional Properties | |
|---|---|---|---|
| Property | Page | Property | Page |
| Property 3.1 Self Inclusion | 9 | Property 3.4 Primary Component Membership | 11 |
| Property 3.2 Local Monotonicity | 9 | Property 4.3 Sending View Delivery | 13 |
| Property 3.3 Initial View Event | 10 | Property 4.5 Virtual Synchrony | 16 |
| Property 4.1 Delivery Integrity | 12 | Property 4.6 Transitional Set | 17 |
| Property 4.2 No Duplication | 12 | Property 4.7 Agreement on Successors | 18 |
| Property 4.4 Same View Delivery | 14 | | |

Table 6: Summary of safety properties of the membership and multicast services.

In order to account for the diverse requirements of different applications, we followed a modular paradigm in this paper: Our specifications are divided into independent properties which may be used as building blocks for the construction of a large variety of actual specifications. Individual specification requirements may be matched by specific protocol layers in modular GCSs. This makes it possible to separately reason about the guarantees of each layer and the correctness of its implementation. Furthermore, the modularity of our specifications provides the flexibility to describe systems that incorporate a variety of QoS options with different semantics. Table 7 summarizes the properties of different ordering and reliability services (FIFO, causal and totally ordered) we have described in this paper, as well as safe message indications. In the future, our framework may be used for specifying additional qualities of service and semantics.

| FIFO Multicast | | Causal Multicast | |
|---|---|---|---|
| Property 6.1 FIFO Delivery | 21 | Property 6.3 Causal Delivery | 22 |
| Property 6.2 Reliable FIFO | 21 | Property 6.4 Reliable Causal | 22 |
| **Totally Ordered Multicast** | | **Safe Indications** | |
| Property 6.5 Strong Total Order | 23 | Property 5.1 Safe Indication Prefix | 19 |
| Property 6.6 Weak Total Order | 23 | Property 5.2 Safe Indication Reliable Prefix | 20 |
| Property 6.7 Reliable Total Order | 24 | | |

Table 7: Properties of different ordered multicast services and of safe message indications.

We have presented specifications of GCSs running in asynchronous failure-prone environments in which agreement problems that resemble group communication services are not solvable. We addressed the non-triviality issues and suggested ways to circumvent impossibility results by specifying conditional liveness guarantees and by using external failure detectors. We have argued that our specifications are non-trivial on one hand, and feasible to implement on the other. In Table 8 we summarize the liveness properties.

| Basic Properties | | Optional Properties | |
|---|---|---|---|
| Property | Page | Property | Page |
| Property 10.1.1 Membership Precision | 32 | Property 10.1.4 Safe Indication Liveness | 32 |
| Property 10.1.2 Multicast Liveness | 32 | Property 10.2 Termination of Delivery | 34 |
| Property 10.1.3 Self Delivery | 32 | | |

Table 8: Summary of liveness properties.

We would like to emphasize that the set of specifications presented herein has been carefully assembled to satisfy the common requirements of numerous fault tolerant distributed applications. Throughout the paper, the specifications are justified with examples of applications that benefit from them.

We hope that the specifications framework presented in this paper will help builders of group communication systems understand and specify their service semantics, and that the extensive survey will allow them to compare their service to others. Application builders will find in this paper a guide to the services provided by a large variety of GCSs, which would help them chose the GCS appropriate for their needs. Moreover, we hope that the formal framework will provide a basis for interesting theoretical work, analyzing relative strengths of different properties and the costs of implementing them.

In the Appendix, we present Lemma A.1 which implies that a certain combination of properties of a reliable totally ordered and FIFO ordered multicast service implies that the service also preserves the reliable causal order. We have included the lemma in this paper, as it can be proven by logical analysis of the properties themselves without considering GCS implementations. By reasoning about implementations, using arguments about when one execution of an algorithm "looks like" another execution to a certain instance of the algorithm, one can prove many other links between properties. For example, one can prove a "dual" assertion to Lemma A.1, showing that a non-reliable totally ordered and FIFO ordered multicast service implies that the service also preserves the causal order. An interesting research direction would be to explore additional relationships and tradeoffs between different properties.

## 12 Acknowledgments

We are grateful to Uri Abraham, Aviva Dayan, Roberto De Prisco, Alan Fekete, Shmuel Katz, Roger Khazan, Nancy Lynch, Sharon Or, Roberto Segala and Jeremy Sussman for many comments and helpful suggestions which helped us improve the quality of the presentation. We thank Ken Birman and Robbert van Renesse for inspiring us to undertake the project of writing this paper.

# Appendix

## A Proving a Relationship between Different Properties.

We prove that a certain combination of properties of a reliable totally ordered and FIFO ordered multicast service implies that the service also preserves the reliable causal order.

**Lemma A.1** *Properties 6.7 (Reliable Total Order), 6.5 (Strong Total Order), 6.2 (Reliable* FIFO*) and 6.1 (*FIFO *delivery) along with Property 4.3 (Sending View Delivery) and the basic Properties 4.1 (Delivery Integrity), 3.2 (Local Monotonicity) and 3.3 (Initial View Event) imply Properties 6.4 (Reliable Causal) and 6.3 (Causal).*

**Proof:** First, let us prove the following claims:

**Claim A.1.1** *If* $t_i = \mathbf{recv}(p,m)$, $t_k = \mathbf{send}(p,m')$, $i < k$, $viewof(t_i) = viewof(t_k)$ *and* $receives(q,m')$, *then* $ts(m) < ts(m')$

**Proof:** If $m = m'$, we get a contradiction to Delivery Integrity (Property 4.1) since every message can be sent only once (by Message Uniqueness, Assumption 2.2). Then, since $m \neq m'$, $ts(m) \neq ts(m')$. Now, assume the contrary, that is, $ts(m) > ts(m')$. Then, according to Reliable Total Order (Property 6.7), since there is $\mathbf{recv}(p,m)$ and $\mathbf{recv}(q,m')$, there is also $\mathbf{recv}(p,m')$. According to Strong Total Order, $\mathbf{recv}(p,m')$ is before $\mathbf{recv}(p,m)$. This means that $p$ receives its own message $m'$ before sending it. Since every message can be sent only once, this is a contradiction to the basic Delivery Integrity property 4.1. Thus, $ts(m) < ts(m')$. □

**Claim A.1.2** *If* $t_i$ *and* $t_k$ *are two events of types* **send** *or* **recv** *that occur at the same process* $p$, *such that* $i < k$, *then either* $viewof(t_i) = viewof(t_k)$ *or* $viewof(t_i).vid < viewof(t_k).vid$.

**Proof:** According to Initial View Event and Strong Local Monotonicity properties.
□

**Claim A.1.3** *If* $\mathbf{send}(p,m) \rightarrow \mathbf{send}(p',m')$, *then there is a sequence of events either* $S1 = \mathbf{send}(p_1 = p, m_1 = m) \rightarrow \mathbf{send}(p_1,m'_1) \rightarrow \mathbf{recv}(p_2,m'_1) \rightarrow \mathbf{send}(p_2,m_2) \rightarrow \mathbf{recv}(p_3,m_2) \rightarrow \mathbf{send}(p_3,m_3) \rightarrow \ldots \rightarrow \mathbf{recv}(p_n = p', m_{n-1}) \rightarrow \mathbf{send}(p_n = p', m_n = m')$ *or* $S2 = \mathbf{send}(p_1 = p, m_1 = m) \rightarrow \mathbf{recv}(p_2,m_1) \rightarrow \mathbf{send}(p_2,m_2) \rightarrow \ldots \rightarrow \mathbf{recv}(p_n = p', m_{n-1}) \rightarrow \mathbf{send}(p_n = p', m_n = m')$.

**Proof:** According to the transitive causality definition (Definition 6.2), there is a sequence $S$ of events starting with $\mathbf{send}(p,m)$ and ending with $\mathbf{send}(p',m')$. Each pair $t_i$ and $t_k$ of consecutive events in this sequence is either sending and receiving of the same message, or $pid(t_i) = pid(t_k)$ and $i < k$. Let us fix a process $q$ such that some event in $S$ occurred at $q$, and look at the first and the last event in $S$ that occurred at $q$. The last event is always a **send** event. The first event is a **send** event for $q = p$, and **recv** event for $q \neq p$. Therefore, if for each process $q$, we leave only the first and the last event in $S$ that occurred at $q$ and remove all the intermediate events from $S$, we obtain the required sequence. □

We now proceed to the proof of the lemma. Let us assume that $t_i = \mathbf{send}(p,m) \rightarrow t_k = \mathbf{send}(p',m')$, $viewof(t_i) = viewof(t_k)$ and there exists $\mathbf{recv}(q,m')$. We should prove that there is also $\mathbf{recv}(q,m)$, and $\mathbf{recv}(q,m)$ precedes $\mathbf{recv}(q,m')$. According to Claim A.1.3, there is a a sequence $S1$ of events $\mathbf{send}(p_1=p, m'_1=m) \rightarrow \mathbf{send}(p_1,m_1) \rightarrow \mathbf{recv}(p_2,m_1) \rightarrow \mathbf{send}(p_2,m_2) \rightarrow \ldots \rightarrow \mathbf{recv}(p_n=p',m_{n-1}) \rightarrow \mathbf{send}(p_n=p',m_n=m')$ [12]. For proof convenience we denote $q = p_{n+1}$.

First, let us prove that all events in this sequence occur in the same view. Assume the contrary. Then there is a pair of consecutive events $t_j$ and $t_l$ in $S$ such that $\mathit{viewof}(t_j) \neq \mathit{viewof}(t_l)$. If $t_j$ and $t_l$ are **send** and **recv** of the same message, then $\mathit{viewof}(t_j) = \mathit{viewof}(t_l)$ according to Sending View Delivery. Therefore, $t_j$ and $t_l$ occurred at the same process, and $j < l$. Using Claim A.1.2, we conclude that $\mathit{viewof}(t_j).\mathit{vid} < \mathit{viewof}(t_l).\mathit{vid}$. Hence, $\mathit{viewof}(\mathbf{send}(p_1,m'_1)).\mathit{vid} \leq \mathit{viewof}(\mathbf{send}(p_1,m_1)).\mathit{vid} = \mathit{viewof}(\mathbf{recv}(p_2,m_1)).\mathit{vid} \leq \mathit{viewof}(\mathbf{send}(p_2,m_2)).\mathit{vid} = \ldots = \mathit{viewof}(t_j).\mathit{vid} < \mathit{viewof}(t_l).\mathit{vid} = \ldots = \mathit{viewof}(\mathbf{recv}(p_n,m_{n-1})).\mathit{vid} \leq \mathit{viewof}(\mathbf{send}(p_n,m_n)).\mathit{vid}$. Summarizing, $\mathit{viewof}(t_i).\mathit{vid} < \mathit{viewof}(t_k).\mathit{vid}$. This is a contradiction to the lemma condition that $\mathit{viewof}(t_i) = \mathit{viewof}(t_k)$.

Since there are $\mathbf{send}(p_1,m'_1)$, later $\mathbf{send}(p_1,m_1)$ and $\mathbf{recv}(p_2,m_1)$ in the same view, there is also $\mathbf{recv}(p_2,m'_1)$ according to Property 6.2 (Reliable FIFO). From Property 6.1 (FIFO Delivery), $\mathbf{recv}(p_2,m'_1)$ precedes $\mathbf{recv}(p_2,m_1)$. Hence, according to Property 6.5 (Strong Total Order), $ts(m'_1) < ts(m_1)$. Applying Claim A.1.1 to $\mathbf{recv}(p_i,m_{i-1})$, $\mathbf{send}(p_i,m_i)$ and $\mathbf{recv}(p_{i+1},m_i)$ for $2 \leq i \leq n$, we conclude that $ts(m_{i-1}) < ts(m_i)$. Thus, $ts(m'_1=m) < ts(m_n=m')$. Since there is $\mathbf{recv}(q,m')$, then, according to Property 6.7 (Reliable Total Order), there is also $\mathbf{recv}(q,m)$, and according to Property 6.5 (Strong Total Order), $\mathbf{recv}(q,m)$ precedes $\mathbf{recv}(q,m')$. $\square$

[12]We do not give a separate proof for $S2$ since it can be considered as a private case of $S1$.

## References

[AAD93] O. Amir, Y. Amir, and D. Dolev. A highly available application in the Transis environment. In *Proceedings of the Hardware and Software Architectures for Fault Tolerance Workshop, at Le Mont Saint-Michel, France*, June 1993. LNCS 774.

[ABCD96] Y. Amir, D. Breitgand, G. Chockler, and D. Dolev. Group communication as an infrastructure for distributed system management. In *3rd International Workshop on Services in Distributed and Networked Environment (SDNE)*, pages 84–91, June 1996.

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the formal specification of group membership services. TR 95-1534, dept. of Computer Science, Cornell University, August 1995.

[ACDK97] T. Anker, G. V. Chockler, D. Dolev, and I. Keidar. The Caelum toolkit for CSCW: The sky is the limit. In *The Third International Workshop on Next Generation Information Technologies and Systems(NGITS 97)*, pages 69–76, June 1997.

[ACDV97] Y. Amir, G. V. Chokler, D. Dolev, and R. Vitenberg. Efficient state transfer in partitionable environments. In *2nd European Research Seminar on Advances in Distributed Systems (ERSADS'97)*, pages 183–192. BROADCAST (ESPRIT WG 22455), Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne, March 1997. Full version available as Technical Report CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

[ACK$^+$97] T. Anker, G. Chockler, I. Keidar, M. Rozman, and J. Wexler. Exploiting group communication for highly available video-on-demand services. In *Proceedings of the IEEE 13th International Conference on Advanced Science and Technology (ICAST 97) and the 2nd International Conference on Multimedia Information Systems (ICMIS 97)*, pages 265–270, April 1997.

[ADK99] T. Anker, D. Dolev, and I. Keidar. Fault tolerant video-on-demand services. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 244–252, June 1999.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *6th International Workshop on Distributed Algorithms (WDAG)*, pages 292–312. Springer Verlag, November 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *22nd IEEE Fault-Tolerant Computing Symposium (FTCS)*, July 1992.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[Ahu93] M. Ahuja. Assertions about past and future in highways: Global flush broadcast and flush-vector-time. *Information Processing Letters*, 48(1):21–28, October 1993.

[Ami95] Y. Amir. *Replication Using Group Communication Over a Partitioned Network.* PhD thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1995.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4), November 1995.

[AS98] Y. Amir and J. Stanton. The Spread Wide Area Group Communication System. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University, 1998.

[ASAWM99] Ehab S Al-Shaer, Hussein Abdel-Wahab, and Kurt Maly. HiFi: A new monitoring architecture for distributed system management. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 171–178, June 1999.

[ASYAW+97] Ehab Al-Shaer, Alaa Youssef, Hussein Abdel-Wahab, Kurt Maly, and C. Michael Overstreet. Reliability, scalability and robustness issues in IRI. In *IEEE Sixth Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'97)*, June 1997.

[AWMY+96] Hussein Abdel-Wahab, Kurt Maly, Alaa Youssef, E. Stoica, C. Michael Overstreet, C. Wild, and Ajay Gupta. The software architecture and interprocess communications of IRI: an internet-based interactive distance learning system. In *IEEE Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'96)*, June 1996.

[BBD96] Ö. Babaoğlu, A. Bartoli, and G. Dini. On programming with view synchrony. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, May 1996. Also available as technical report UBLCS95-15, Department of Computer Science, University of Bologna, 1995.

[BDGB94] Ö. Babaoğlu, R. Davoli, L. Giachini, and M. Baker. RELACS: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. TR UBLCS94-15, Laboratory of Computer Science, University of Bologna, 1994.

[BDM95] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems. TR UBLCS-95-18, Department of Conmputer Science, University of Bologna, November 1995.

[BDM97] Ö. Babaoğlu, R. Davoli, and A. Montresor. Partitionalbe Group Membership: Specification and Algorithms. TR UBLCS97-1, Department of Conmputer Science, University of Bologna, January 1997.

[BDMS97] Ö. Babaoğlu, R. Davoli, A. Montresor, and R. Segala. System Support for Partition-Aware Network Applications. TR UBLCS97-8, Department of Conmputer Science, University of Bologna, October 1997.

[BFHR98] K. Birman, R. Friedman, M. Hayden, and I. Rhee. Middleware support for distributed multimedia and collaborative computing. In *Multimedia Computing and Networking (MMCN98)*, 1998.

[BH98] Mark Bickford and Jason Hickey. An object-oriented approach to verifying group communication systems. Dept. of Computer Science, Cornell University, submitted to CADE-16, 1998.

[Bir86] Kenneth P. Birman. ISIS: A System for Fault-Tolerant Distributed Computing. Technical Report TR86-744, Cornell University, Department of Computer Science, April 1986.

[Bir93] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12), December 1993.

[Bir96] K. Birman. *Building Secure and Reliable Network Applications*, chapter 18. Manning, 1996.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *11th ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 123–138. ACM, Nov 1987.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, 1991.

[BvR94] K. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[CHD98] G. Chockler, N. Huleihel, and D. Dolev. An adaptive totally ordered multicast protocol that tolerates partitions. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 237–246, June 1998.

[CHKD96] G. Chockler, N. Huleihel, I. Keidar, and D. Dolev. Multimedia multicast transport service for groupware. In *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies*, September 1996. Full version available as Technical Report CS96-3, The Hebrew University, Jerusalem, Israel.

[Cho97a] G. V. Chockler. An Adaptive Totally Ordered Multicast Protocol that Tolerates Partitions. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1997.

[Cho97b] G. V. Chockler. An Implementation of Reliable Multicast Services in the Transis Group Communication System. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, October 1997. Available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[CHRC97] Sarah Chodrow, Michael Hircsh, Injong Rhee, and Shun Yan Cheung. Design and implementation of a multicast audio conferencing tool for a collaborative computing framework. In *JCIS*, March 1997.

[CHTCB96] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 322–330, May 1996.

[Cri91] Flaviu Cristian. Reaching agreement on processor group membership in synchronous distributed systems. *Distributed Computing*, 4(4):175–187, April 1991.

[CS95] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Computer Science and Engineering, University of California, San Diego, 1995.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[DFKM96] D. Dolev, R. Friedman, I. Keidar, and D. Malki. Failure Detectors in Omission Failure Environments. TR 96-13, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, September 1996. Also Technical Report 96-1608, Department of Computer Science, Cornell University.

[DKM93] D. Dolev, S. Kramer, and D. Malki. Early delivery totally ordered broadcast in asynchronous environments. In *23rd IEEE Fault-Tolerant Computing Symposium (FTCS)*, pages 544–553, June 1993.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DM95] D. Dolev and D. Malki. The design of the Transis system. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 83–98. Springer Verlag, 1995. LNCS 938.

[DM96] D. Dolev and D. Malkhi. The Transis approach to high availability cluster communication. *Communications of the ACM*, 39(4), April 1996.

[DMS95] D. Dolev, D. Malki, and H. R. Strong. A Framework for Partitionable Membership Service. TR 95-4, Institute of Computer Science, The Hebrew University of Jerusalem, March 1995.

[DPFLS98] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic view-oriented group communication service. In *17th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 227–236, June 1998.

[DPFLS99] R. De Prisco, A. Fekete, N. Lynch, and A. Shvartsman. A dynamic primary configuration group communication service. In *13th International Symposium on DIStributed Computing (DISC)*, page To appear, Bratislava, Slovak Republic, 1999.

[EMS95] P. D. Ezhilchelvan, A. Macedo, and S. K. Shrivastava. Newtop: a fault tolerant group communication protocol. In *15th International Conference on Distributed Computing Systems (ICDCS)*, June 1995.

[FLS97] A. Fekete, N. Lynch, and A. Shvartsman. Specifying and using a partionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 53–62, August 1997.

[FV97a] R. Friedman and A. Vaysburg. Fast replicated state machines over partitionable networks. In *16th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, October 1997.

[FV97b] R. Friedman and A. Vaysburg. High-performance replicated distributed objects in partitionable environments. Technical Report 97-1639, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, July 1997.

[FvR95] Roy Friedman and Robbert van Renesse. Strong and Weak Virtual Synchrony in Horus. TR 95-1537, dept. of Computer Science, Cornell University, August 1995.

[FvR97] Roy Friedman and Robbert van Renesse. Packing messages as a tool for boosting the performance of total ordering protocols. In *6th IEEE International Symposium on High Performance Distributed Computing*, 1997. Also available as Technical Report 95-1527, Department of Computer Science, Cornell University.

[GCA$^+$97] D. Gang, G. Chockler, T. Anker, A. Kremer, and T. Winkler. Conducting midi sessions over the network using the Transis group communication system. In *International Computer Music Conference (ICMC 97)*, September 1997.

[GG91] Stephen J. Garland and John V. Guttag. A guide to LP, the Larch Prover. Research Report 82, Digital Systems Research Center, 130 Lytton Avenue, Palo Alto, CA 94301, December 1991.

[GHG$^+$93] John V. Guttag, James J. Horning, Stephen J. Garland, Kevin D. Jones, Andrés Modet, and Jeannette M. Wing, editors. *Larch: Languages and Tools for Formal Specification.* Springer-Verlag Texts and Monographs in Computer Science, 1993.

[GL98a] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for designing, analyzing, and building distributed systems. Technical Report MIT/LCS/TR-762, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, August 1998. URL `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR-762.ps`.

[GL98b] Stephen J. Garland and Nancy A. Lynch. The IOA language and toolset: Support for mathematics-based distributed programming, 1998. Manuscript.

[GL99] Stephen J. Garland and Nancy A. Lynch. *Foundations of Component Based Systems*, chapter Using I/O Automata for Developing Distributed Systems. Cambridge University Press, USA, 1999. To appear.

[GLV97] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems.* Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, December 1997. URL `http://sds.lcs.mit.edu/~garland/ioaLanguage.html`.

[GS95] R. Guerraoui and A. Schiper. Transaction model vs virtual synchrony model: bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938, pages 121–132. Springer-Verlag, September 1995.

[GS97a] R. Guerraoui and A. Schiper. Consensus: the big misunderstanding. In *Proceedings of the 6th IEEE Computer Society Workshop on Future Trends in Distributed Computing Systems (FTDCS-6)*, pages 183–188, Tunis, Tunisia, October 1997. IEEE Computer Society Press.

[GS97b] R. Guerraoui and A. Schiper. Genuine atomic multicast. In *11th International Workshop on Distributed Algorithms (WDAG)*, Saarbrücken, Germany, September 1997.

[HLvR99] Jason Hickey, Nancy Lynch, and Robbert van Renesse. Specifications and proofs for ensemble layers. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems,(TACAS '99, Amsterdam, the Netherlands, March 1999)*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[HS95] M. Hiltunen and R. Schlichting. Properties of membership services. In *2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, 1995.

[HvR96] M. Hayden and R. van Renesse. Optimizing Layered Communication Protocols. Technical Report TR96-1613, Dept. of Computer Science, Cornell University, Ithaca, NY 14850, USA, November 1996.

[JFR93] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *12th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, pages 2–11. IEEE, October 1993.

[JJS99] Scott Johnson, Farnan Jahanian, and Jigney Shah. The inter-group router approach to scalable group composition. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 4–14, June 1999.

[KA98] Bettina Kemme and Gustavo Alonso. A suite of database replication protocols based on group communication primitives. In *18th International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[KB99] Michael Kalantar and Kenneth Birman. Causally ordered multicast: the conservative approach. In *19th International Conference on Distributed Computing Systems (ICDCS)*, pages 36–44, June 1999.

[KCH98] Alan Krantz, Sarah Chodrow, and Michael Hircsh. Design and implementation of a distributed x multiplexor. In *18th International Conference on Distributed Computing Systems (ICDCS)*, May 1998.

[KD96] I. Keidar and D. Dolev. Efficient message ordering in dynamic networks. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 68–76, May 1996.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also Institute of Computer Science, The Hebrew University of Jerusalem Technical Report CS95-5, and available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[KFL98] Roger Khazan, Alan Fekete, and Nancy Lynch. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing (DISC)*, pages 258–272, Andros, Greece, September 1998.

[Kha98] Roger Khazan. Group communication as a base for a load-balancing, replicated data service. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, June 1998.

[KK99] Idit Keidar and Roger Khazan. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. Technical report, MIT Laboratory for Computer Science, 1999. In preparation.

[KRB$^+$97] Alan Krantz, Injong Rhee, C. Breuker, Sarah Chodrow, and V. Sunderam. Supporting input multiplexing in a heterogenous environment. In *JCIS*, March 1997.

[KS98] A. D. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91–111, April 1998.

[KSMD99] I. Keidar, J. Sussman, K. Marzullo, and D. Dolev. A Client-Server Oriented Algorithm for Virtually Synchronous Group Membership in WANs. Technical Report CS99-623, Department of Conmputer Science and Engineering, University of California, San Diego, June 1999. Also MIT Technical Memorandum MIT-LCS-TM-593.

[KT96] M. F. Kaashoek and A. S. Tanenbaum. An evaluation of the Amoeba group communication system. In *16th International Conference on Distributed Computing Systems (ICDCS)*, pages 436–447, May 1996.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 78.

[LSGL95] Victor Luchangco, Ekrem Söylemez, Stephen Garland, and Nancy Lynch. Verifying timing properties of concurrent algorithms. In Dieter Hogrefe and Stefan Leue, editors, *Formal Description Techniques VII: Proceedings of the 7th IFIP WG6.1 International Conference on Formal Description Techniques* (FORTE'94, Berne, Switzerland, October 1994), pages 259–273. Chapman and Hall, 1995.

[LT89] N.A. Lynch and M.R. Tuttle. An introduction to Input/Output Automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Lyn96] N.A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers, 1996.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994. Full version: technical report ECE93-22, Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA.

[MFSW95] C. P. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant, distributed applications in large scale. In *Workshop on Parallel and Distributed Platforms in Industrial Products*, October 1995.

[MMR97] Dahlia Malkhi, Michael Merritt, and Ohad Rodeh. Secure multicast in a WAN. In *17th International Conference on Distributed Computing Systems (ICDCS)*, pages 87–94, May 1997.

[MMSA$^+$96] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4), April 1996.

[MP99] S. Mishra and G. Pang. Design and implementation of an availability management service. In *19th International Conference on Distributed Computing Systems (ICDCS) Workshop on Middleware*, pages 128–133, June 1999.

[MPS91a] S. Mishra, L. L. Peterson, and R. D. Schlichting. A Membership Protocol based on Partial Order. In *Proc. of the intl. working conf. on Dependable Computing for Critical Applications*, Feb 1991.

[MPS91b] S. Mishra, L. L. Peterson, and R. L. Schlichting. Consul: A Communication Substrate for Fault-Tolerant Distributed Programs. TR 91-32, dept. of Computer Science, University of Arizona, 1991.

[MR97] Dahlia Malkhi and Michael Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security*, 5:113–127, 1997.

[MS94] C. Malloth and A. Schiper. View synchronous communication in large scale networks. In *2nd Open Workshop of the ESPRIT project BROADCAST (Number 6360)*, July 1995 (also available as a Technical Report Nr. 94/84 at Ecole Polytechnique Fédérale de Lausanne (Switzerland), October 1994).

[Nei96] G. Neiger. A new look at membership services. In *15th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 331–340. ACM, 1996.

[PPG$^+$96] Tsvetomir P. Petrov, Anna Pogosyants, Stephen J. Garland, Victor Luchangco, and Nancy A. Lynch. Computer-assisted verification of an algorithm for concurrent timestamps. In Reinhard Gotzhein and Jan Bredereke, editors, *Formal Description Techniques IX: Theory, Applications, and Tools* (FORTE/PSTV'96: Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification, Kaiserslautern, Germany, October 1996), pages 29–44. Chapman & Hall, 1996.

[RB91] A. M. Ricciardi and K. P. Birman. Using process groups to implement failure detection in asynchronous environments. In *ACM Symposium on Principles of Distributed Computing (PODC)*, pages 341–352, August 1991.

[RCHS97] I. Rhee, S. Cheung, P. Hutto, and V. Sunderam. Group communication support for distributed multimedia and CSCW systems. In *17th International Conference on Distributed Computing Systems (ICDCS)*, 1997.

[Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *2nd ACM Conference on Computer and Communication Security*, pages 68–80, November 1994.

[Rei95] M. K. Reiter. The Rampart toolkit for building high-integrity services. In *Theory and Practice in Distributed Systems (LNCS 938)*, pages 99–110. Springer-Verlag, 1995.

[Rei96a] M. K. Reiter. Distributing trust with the Rampart toolkit. *Communications of the ACM*, 39:71–74, April 1996.

[Rei96b] M. K. Reiter. A secure group membership protocol. *IEEE Transactions on Software Engineering*, 22(1):31–42, January 1996.

[RV92] L. Rodrigues and P. Verissimo. $x$AMp, a protocol suite for group communication. RT /43-92, INESC, January 1992.

[SAGG$^+$93] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anna Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification* (5th International Conference, CAV'93, Elounda, Greece, June/July 1993), volume 697 of *Lecture Notes in Computer Science*, pages 305–319. Springer-Verlag, 1993.

[Sch90] F. B. Schneider. Implementing fault tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[Sha96] G. Shamir. Shared Whiteboard: A Java Application in the Transis Environment. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, October 1996. Available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[SKM99] J. Sussman, I. Keidar, and K. Marzullo. Optimistic virtual synchrony. Technical Report In preparation, Department of Conmputer Science and Engineering, University of California, San Diego, August 1999. Also MIT Technical Memorandum MIT-LCS-????.

[SM98] J. Sussman and K. Marzullo. The *bancomat* problem: An example of resource allocation in a partitionable asynchronous system. In *12th International Symposium on DIStributed Computing (DISC)*, September 1998. Full version: Tech Report 98-570 University of California, San Diego Department of Computer Science and Engineering.

[SR96] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, 39(4):84–87, April 1996.

[SR93] A. Schiper and A.M. Ricciardi. Virtually synchronous communication based on a weak failure suspector. *Digest of Papers, FTCS-23*, pages 534–543, June 93.

[SS93] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environment. In *13th International Conference on Distributed Computing Systems (ICDCS)*, pages 561–568, May 1993.

[Val98] M. Valenci. Audio Conferencing using Transis. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel, 1998. Available from: `http://www.cs.huji.ac.il/~transis/publications.html`.

[Vit98] R. Vitenberg. Properties of distributed group communication and their utilization. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, January 1998.

[Vog96] Werner Vogels. World wide failures. In *ACM SIGOPS 1996 European Workshop*, September 1996.

[vRBM96] R. van Renesse, K. P. Birman, and S. Maffeis. Horus: A flexible group communication system. *Communications of the ACM*, 39(4), April 1996.

[vRHB94] R. van Renesse, T. M. Hickey, and K. P. Birman. Design and Performance of Horus: A Lightweight Group Communications System. TR 94-1442, dept. of Computer Science, Cornell University, August 1994.

[VvR94] Werner Vogels and Robbert van Renesse. *Support for Complex Multi-Media Applications using the Horus system.* Ithaca, NY 14850, USA, December 1994. On-line html document: `http://www.cs.cornell.edu/Info/People/rvr/papers/rt/novsdav.html`.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A high perfomance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schipper, editors, *Theory and Practice in Distributed Systems: International Workshop*, pages 33–57. Springer Verlag, 1995. LNCS 938.

[WS95] U. G. Wilhelm and A. Schiper. A hierarchy of totally ordered multicasts. In *14th IEEE International Symposium on Reliable Distributed Systems (SRDS)*, September 1995.

[YLKD97] E. Yeger Lotem, I. Keidar, and D. Dolev. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 63–71, August 1997.

# Index