Group Communication Specifications: A Comprehensive Study

GREGORY V. CHOCKLER

The Hebrew University of Jerusalem Computer Science Institute

IDIT KEIDAR

MIT Laboratory for Computer Science

AND

ROMAN VITENBERG

The Technion Department of Computer Science

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 survey 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 30 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. This survey is aimed at both system builders and theoretical researchers. The specification framework presented in this article will help builders of group communication systems understand and specify their service semantics; the extensive

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems; D.4.7 [**Operating Systems**]: Organization and Design—*distributed systems*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*specification techniques*; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design—*network communications*

General Terms: Algorithms, Reliability, Standardization

Additional Key Words and Phrases: Group communication systems, partitionable group membership, process group membership, specifications of group communication systems, view synchrony, virtual synchrony

---

This work is supported by Air Force Aerospace Research (OSR) grant F49620-00-1-0097, Nippon Telegraph and Telephone (NTT) grant MIT9904-12, and by NSF grants ACI-9876931, CCR-9909114, and EIA-9901592.

Authors' addresses: G. V. Chockler, Givat Ram, Jerusalem, 91904 Israel; email: grishac@cs.huji.ac.il; I. Keidar, 545 Technology Square, Cambridge, MA, 02139; email: idish@theory.lcs.mit.edu; R. Vitenberg, Technion City, Haifa 32000, Israel; email: romanv@cs.technion.ac.il.

ACM Computing Surveys, Vol. 33, No. 4, December 2001, pp. 427–469.

survey will allow them to compare their service to others. Application builders will find a guide here to the services provided by a large variety of GCSs, which could help them choose 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

1. INTRODUCTION

Group communication is a means for providing multipoint to multipoint communication, by organizing processes in groups. A *group* is a set of processes that are members of the group. For example, a group can consist of users playing an online game with each other. Another group can consist of participants in a multimedia conference. Each group is associated with a logical name. Processes communicate with group members by sending a message to the group name; the group communication service delivers the message to the group members.

In this survey, we focus on view-oriented group communication systems (GCSs). Such systems provide membership and reliable multicast services. The task of a *membership service* is to maintain a list of the currently active and connected processes in a group. 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 [Birman 1986]; it was followed by over a dozen others.

GCSs are powerful building blocks that facilitate the development of fault-tolerant distributed systems. Classical GCS applications include replication using a variant of the state machine/active replication approach [Lamport 1978; Schneider 1990] (e.g., Keidar and Dolev [1996], Amir et al. [1994] Fekete et al. [1997], Friedman and Vaysburg [1997], and Montresor et al. [2000]); primary-backup replication (e.g., Guerraoui and Schiper [1997b]); support for distributed and clustered operating systems (e.g., Kaashoek and Tanenbaum [1996], Goft and Yeger Lotem [1999], IBM [1996], and Cheriton and Zwaenepoel [1985]); distributed transactions and database replication (see Schiper and Raynal [1996], Guerraoui and Schiper [1995], Kemme and Alonso [1998], and Keidar [1994]); resource allocation (see Sussman and Marzulllo [1998] and Babaoğlu et al. [1998a]); load balancing (see Khazan et al. [1998] and Dolev et al. [1999]); system management (see Amir et al. [1996]) and monitoring (see Al-Shaer et al. [1999]); and highly available servers, for example, Mishra and Pang [1999] and Fekete and Keidar [2001] and the video-on-demand servers of Anker et al. [1999] and Vogels and van Renesse [1994].

More recently, GCSs have been exploited for collaborative computing (see Chockler et al. [1996], Rhee et al. [1997], Birman et al. [1998], and Anker et al. [1997]), for example, distance learning (see Al-Shaer et al. [1997]), drawing on a shared whiteboard (see Shamir [1996]), video and audio conferences (see Chodrow et al. [1997] and Valenci [1998]), application sharing (see Krantz et al. [1998, 1997]), and even distributed musical "jam sessions" over a network [Gang et al. 1997].

Currently, real-time GCSs such as RT-CAST [Abdelzaher et al. 1996] are being developed and are being exploited for real-time applications, for example, radar tracking; see Johnson et al. [2000]. Another emerging research direction focuses on the provision of object group services within the Common Object Request Broker Architecture (CORBA) framework; for examples, see Electra [Landis and Maffeis 1997], Orbix+Isis [IONA 1994], Eternal [Moser et al. 1998], and the Object Group Service [Felber et al. 1998]. Furthermore, GCS has been recently identified as a key tool for supporting fault tolerance in CORBA: the new fault-tolerant CORBA specification [OMG 2000] recommends that view-oriented GCSs be used to support active object replication in CORBA.

Traditionally, GCS developers concentrated primarily on performance, in order to make their systems useful for real-world distributed applications. In the past few years, the challenging task of specifying the semantics and services of GCSs has become an active research area (see Moser et al.[1994], Friedman and van Renesse [1995], Babaoğlu et al. [1998b], Fekete et al. [1997], De Prisco et al. [1998], Hickey et al. [1999], Keidar and Khazan [2000], Galleni and Powell [1996], and Lin and Hadzilacos [1999]). However, no comprehensive set of specifications covering the whole spectrum of useful GCS features has yet been established.

The task of defining a meaningful GCS is complicated by the fact that group communication services strive to have processes reach agreement about membership views, delivered messages, and the like, whereas many agreement problems are known to be unsolvable in failure-prone asynchronous environments. Many of the suggested specifications fail to capture the nontriviality of existing GCSs. In particular, many specifications are solvable by trivial algorithms (as shown in Anceaume et al. [1995]). Others are too strong to implement (as proven in Chandra et al. [1996]).

The main objective of this article is to present a comprehensive set of rigorously defined properties of GCSs that reflect the usefulness and nontriviality of numerous existing GCS implementations. We do not define new properties; rather, we rigorously formalize in a unified framework properties that have previously appeared in numerous sources in the literature in different forms.

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 Anceaume et al. [1995]. 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.

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 30 published specifications that cover over a dozen leading GCSs (including Consul [Mishra et al. 1993; Cristian and Schmuck 1995], the configurable service of Hiltunen and Schlichting [1998], Ensemble [Hayden and van Renesse 1996], Horus [van Renesse et al. 1996], Isis [Birman and van Renesse 1994], Newtop [Ezhilchelvan et al. 1995], Phoenix [Malloth et al. 1995], Relacs [Babaoğlu et al. 1998b], RMP [Whetten et al. 1995], Spread [Amir and Stanton 1998], Timewheel [Mishra et al. 1998], Totem [Amir et al. 1995], Transis [Dolev and Malkhi 1996; Amir et al. 1992b], and xAMp [Rodrigues and Verissimo 1992]). We correlate the terminology used in different papers with 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 Fekete et al. [1997], Keidar and Dolev [1996], Friedman and Vaysburg [1997], Amir et al. [1996, 1994, 1997], Anker et al. [1999], Vogels and van Renesse [1994], Sussman and Marzullo [1998], and Khazan et al. [1998]). We choose not to consider properties that are not exploited by applications, even if these properties are satisfied by many GCSs.

Nonetheless, not all the specifications are useful for all the applications. Experience with group communication systems and reliable distributed applications has shown that there is no "right" system semantics for all applications (see Birman [1996, Chapter 18]): Different GCSs are tailored to different applications that require different semantics and different qualities of service (QoS). Modern GCSs (e.g., Ensemble, Horus, and the configurable service of Hiltunen and Schlichting [1998]) 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 GCSs, 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 that 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 or micro-protocols in existing GCSs. This makes it possible to separately reason about the guarantees of each layer and the correctness of its implementation (see Hickey et al. [1999]). 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 [Lynch and Tuttle 1989]. We use logic formulae for stating the properties, to avoid ambiguity. 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.

Vitenberg [1998] presents a multisorted algebra of which the model herein is a possible interpretation. The axioms presented in this article also conform with Vitenberg's formalism. The benefit of using multisorted algebras is that axioms stated using this formalism can be checked with automated theorem proving tools, for example, the Larch Prover [Guttag et al. 1993].

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." For example, group membership algorithms usually use time-out-based failure detection in order to track the network situation. If a message from some process *q* to another process *p* is delayed longer than a certain time-out, then *p* will exclude *q* from its membership view. Theoretically, an adversary that knows the time-out and fully controls the communication can delay messages longer than this time-out, causing *p* to exclude *q* although *q* is correct. In general, an adversary 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, in practical networks, communication tends to be stable and timely during long periods. 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 as vicious adversaries.

Many formal specifications of group communication systems do not capture this notion of "best-effort." This results in specifications that can be implemented by trivial algorithms (as demonstrated in Anceaume et al. [1995]). Other specifications turn out to be too strong to implement (see Chandra et al. [1996]). However, since the "best-effort" principle is an important consideration of system builders, actual systems provide more than their specifications require.

In this survey, we address the nontriviality issues using external failure detectors and by reasoning about liveness guarantees at stable periods.

1.4. Roadmap to this Survey

This survey 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 needs to be preserved across groups (i.e., for messages multicast in separate groups).

Throughout the article 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 survey are meaningless unless certain basic properties hold.

The rest of this article is divided into two main parts: safety properties of group communication systems, and liveness properties. In order to state the liveness properties, we use the failure detector abstraction. Whereas safety properties are preserved in all runs, liveness

---

[1] Impossibility results to this effect may be found in Section 9 of this article and in Chandra et al. [1996].

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.

Each of the parts begins with a model section: Section 2 presents the model for all the properties presented in this article; 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 presents 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 survey; it contains tables that summarize all the properties presented here. In these tables, we also distinguish between basic and optional properties. In the Appendix, we prove a lemma 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 article as it can be proven by logical analysis of the properties themselves without considering GCS implementations.

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 article, 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 [Reiter 1996] and in Malkhi et al. [1997] and Malkhi and Reiter [1997].

2.1. The Specification Framework

We now overview the formal framework used to specify the group communication service. A system is modeled as a collection of components. The division into components is oriented towards the service model rather than describing an actual implementation: each component provides a service to other components. In practice, a single component can be implemented by a combination of hardware devices, programs, library modules, and so on. Furthermore, components are not necessarily local and can be distributed over a set of machines.

We model both the system and individual components as untimed I/O automata (see Lynch and Tuttle [1989] and Lynch [1996, Chapter 8]). In this model, each component has an internal state, invisible to other components. Components interact using shared actions that can affect the state of individual components. Specifically, an automaton interacts with its environment by two sets of external actions: input and output. These two sets of actions comprise the *external signature* of the automaton. A *trace* of an I/O automaton is the sequence of external actions it takes in an execution. Executions are assumed to be sequential; that is, actions are atomic, and no two actions can occur simultaneously. Roughly speaking, a *fair trace* is a trace of an execution in which enabled actions eventually become executed. For formal definitions, see Lynch [1996, Chapter 8].

In this survey, we only present service specifications; we do not discuss a specific implementation of the service and do not provide any proof of correctness. Therefore we are not concerned with the internal state of components but only with their external behavior, as reflected in their external signature and in their fair traces. A service specification is modeled as a set of acceptable fair traces. A system satisfies a service specification if the set of possible fair traces of the system is a subset of the set of acceptable fair traces defined by the specification. This is in contrast to specifications based on equality and bisimulation, which define the exact set of possible traces of a system rather than restricting this set.

We present the GCS service specification by defining its external signature in Section 2.2, and a collection of *trace properties* throughout the rest of this article. Each trace property is presented as an *axiom* in the set-theoretic mathematical model described in Section 2.3. 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.

2.2. The External Signature of the GCS Service

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

$\mathcal{P}$—the set of processes;

$\mathcal{M}$—the set of messages sent by the application; and

$\mathcal{VID}$—the set of view identifiers, partially ordered by the $<$ operator.

Each action of the GCS 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.

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.

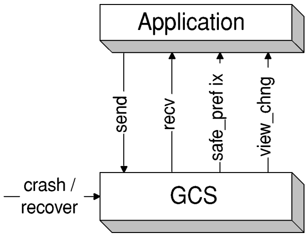

Fig. 1. External actions of the GCS.

—input **send**(p, m), $p \in \mathcal{P}, m \in \mathcal{M}$.

—output **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 **view_chng**$(p, \langle id, members\rangle, T)$, $p \in \mathcal{P}, id \in \mathcal{VID}$, $members \in 2^{\mathcal{P}}$, $T \in 2^{\mathcal{P}}$. 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) [Moser et al. 1994] model (cf. Section 4.3.1).

—output **safe_prefix**(p, m), $p \in \mathcal{P}, m \in \mathcal{M}$.

Interaction with the environment. The following actions model events that may occur in the environment and affect the GCS.

—input **crash**(p), $p \in \mathcal{P}$.

—input **recover**(p), $p \in \mathcal{P}$.

2.3. The Mathematical Model

We now present the mathematical model for stating trace properties of a GCS with the signature described in Section 2.2. We use *set theory* notation to state our axioms; we define the sets:

$\mathcal{P}$, $\mathcal{M}$, $\mathcal{VID}$—Basic sets as described above.

$\mathcal{V}$—The set of views delivered in **view_chng** actions is $\mathcal{VID} \times 2^{\mathcal{P}}$. Thus a view

$V \in \mathcal{V}$ is a pair. We refer to the elements in the pair as $V.id$ and $V.members$.

Events—Occurrences of actions.[2] The set of events is:
$\{\mathbf{send}(p, m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\mathbf{recv}(p, m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\mathbf{view\_chng}(p, V, T) \mid p \in \mathcal{P}, V \in \mathcal{V}, T \in 2^{\mathcal{P}}\} \cup$
$\{\mathbf{safe\_prefix}(p, m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup$
$\{\mathbf{crash}(p) \mid p \in \mathcal{P}\} \cup \{\mathbf{recover}(p) \mid p \in \mathcal{P}\}$.

Traces—Finite or infinite sequences of events.

The first parameter in each event is a process in $\mathcal{P}$. Thus, we can define the function: $pid : Events \rightarrow \mathcal{P}$ which returns the process at which an 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 (finite or infinite) 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.

2.4. Notation

With a view-oriented group communication service, events occur at processes within the context of views. The function $viewof$: $Events \rightarrow \mathcal{V} \cup \{\perp\}$ 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 (e.g., those of Fekete et al. [1997], De Prisco et al. [1998], and Chockler et al. [1998]) 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 2.1. (*viewof*). The *view of* an event t_i occurring at process p is the view delivered to p in a **view_chng** event t_j, which precedes t_i and such that no **view_chng** or **crash** events occur at p between t_j and t_i; the view is $\perp$ if there is no such t_j. Formally:

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

We define some general shorthand predicates in Table I. In all these predicates as well as throughout the rest of this survey, 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.

2.5. Assumptions About the Environment

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 event before a recovery is a crash. Formally:

$(next\_event(i, j, p) \wedge t_j = \mathbf{crash}(p) \Rightarrow t_i = \mathbf{recover}(p)) \wedge (t_j = \mathbf{recover}(p) \Rightarrow \exists i\ (prev\_event(i, j, p) \wedge t_i = \mathbf{crash}(p)))$.

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

---

[2] We use the term "events" in the context of specifications while using the term "actions" to define the automaton signatures.

Table I. General Shorthand Predicate Definitions

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

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 a view-oriented group communication system. The task of a membership service is to maintain a list of the currently active and connected processes. This list can change with new members joining and old ones departing or failing. When this list changes, the membership service reports the change to the members by installing a new view. The membership service strives to install the same view at mutually connected members.

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 3.1.2 we compare two approaches to group membership: partitionable and primary component.

3.1. Basic Properties

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

PROPERTY 3.1 (*Self Inclusion*). If process p installs view V, then p is a member of V. Formally:

$$\mathit{installs}(p,V) \Rightarrow p \in V.\mathit{members}.$$

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 Dolev et al. [1995], Friedman and van Renesse [1995], Ezhilchelvan et al. [1995], Babaoğlu et al. [1998b], Fekete et al. [1997], Keidar and Khazan [2000], and Galleni and Powell [1996].

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:

$$\begin{aligned} t_i &= \mathbf{view\_chng}(p, V, T) \wedge \\ t_j &= \mathbf{view\_chng}(p, V', T') \wedge i > j \\ &\quad \Rightarrow V.id > V'.id. \end{aligned}$$

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 both 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: Ricciardi and Birman [1991], Dolev et al. [1995], Amir et al. [1995], Ezhilchelvan et al. [1995], Malloth and Schiper [1995], and Keidar et al. [2000]); it is also required in all the group membership specifications (e.g., Neiger [1996], Fekete et al. [1997], and De Prisco et al. [1998]). Babaoğlu et al. [1998b] states an equivalent property: the order in which processes install views ensures 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 $\mathcal{VID}$ 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. Such violation of local monotonicity may cause an old message that has been traveling in the network since before the crash to be mistaken for a new one.

There are several ways to remedy this shortcoming. In Isis [Ricciardi and Birman 1991] 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 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 [Dolev et al. 1995] 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 view identifiers in the specifications of Fekete et al. [1997] and Neiger [1996] are taken from an ordered set. Hence, an integer counter is again a possible implementation. In Horus [Friedman and van Renesse 1995] and Cristian and Schmuck [1995], 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 Keidar et al. [2000] 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 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 Hiltunen and Schlichting [1995], and Friedman and Vaysburg [1997]. The protocol of Chockler et al. [1998] 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 Keidar and Dolev [1996, 2000], Amir et al. [1994], and Friedman and Vaysburg [1997].

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.

PROPERTY 3.3 (*Initial View Event*). Every **send**, **recv**, and **safe_prefix** event occurs within some view. Formally:

$$t_i = \mathbf{send}(p, m) \lor t_i = \mathbf{recv}(p, m) \lor t_i = \mathbf{safe\_prefix}(p, m) \Rightarrow \mathit{viewof}(t_i) \neq \bot.$$

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 predefined knowledge about processes in the system is required. Most GCSs adopt this option, for example, Isis and Ensemble.
- 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 [Dolev et al. 1995] and Consul [Mishra et al. 1993] take this approach.

  The initial view may be singleton or may consist of all possible processes in the system. In Hiltunen and Schlichting [1995] these two possibilities are called *individual startup* and *collective startup*, respectively. Transis is an example of a GCS that uses individual startup, and collective startup is deployed, for example, in Consul. Note that in order to install anything different from a singleton view, a process must possess a priori knowledge about other processes in the system. Such knowledge is assumed, for example, in Fekete et al. [1997] and Mishra et al. [1993].

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 Versus Primary Component Membership Services

A membership service may either be *primary component*[3] or *partitionable*. In a primary component membership service, views installed by all the processes in the system are totally ordered. In a partitionable one, views are only partially ordered (i.e., 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. Since the properties above do not enforce a total order on views, the specification presented thus far is 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) 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 f satisfies the following property.
For every view V with $f(V) > 1$ there exist a view V', such that $f(V) = f(V') + 1$, and a member p of V that installs V in V'

---

[3] A primary component was originally called a primary partition.

(i.e., V is the successor of V' at process p). Formally:

$$\begin{aligned}&\exists f : \{V \mid \exists p : \mathit{installs}(p, V)\} \rightarrow \mathcal{N} \ \mathit{such\ that} :\\ &\quad (f(V) = f(V') \Rightarrow V = V') \ \wedge\\ &\quad \forall V\, (f(V) > 1 \Rightarrow \exists V'\, (f(V) = f(V') + 1 \ \wedge\\ &\quad \exists p \in V.\mathit{members} : \mathit{installs\_in}(p, V, V'))).\end{aligned}$$

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 Malloth and Schiper [1995], Ricciardi and Birman [1991], Yeger Lotem et al. [1997], and De Prisco et al. [1998].

The first and best known group membership service is the primary component membership service of Isis [Birman and van Renesse 1994]. It was followed by many other primary component membership services, for example, those of Phoenix [Malloth and Schiper 1995], Consul, and xAMp. Primary component membership services are also specified in Chandra et al. [1996], Neiger [1996], Cristian [1991], Mishra et al. [1991], De Prisco et al. [1998], Lin and Hadzilacos [1999]. Consul, xAMp, and Cristian [1991] guarantee membership service properties only as long as no network partitions occur. In contrast, Isis [Ricciardi and Birman 1991] and Phoenix do assume the possibility of network partitions, but allow execution of the application to proceed only in a single component. In Isis detached processes "commit suicide," whereas in Phoenix they are blocked until the link is mended.

The first partitionable membership service was introduced as part of Transis [Amir et al. 1992a]. Since then, numerous new GCSs featuring a partitionable membership service have emerged, for example, those of Totem, Horus, RMP, Newtop, and Relacs. Partitionable membership services are discussed in the specifications of Moser et al. [1994], Fekete et al. [1997], Babaoğlu et al. [1996], Christian and Schmuck [1995], Jahanian et al. [1993], and Keidar and Khazan [2000]. Hiltunen and Schlichting [1995] present a specification of a primary component membership service and show how to extend it to a specification of a partitionable one.

Partitionable membership services have been used for a variety of applications, for example, resource allocation [Sussman and Marzullo 1998; Babaoğlu et al. 1998a], system management [Amir et al. 1996], monitoring [Al-Shaer et al. 1999], load balancing [Dolev et al. 1999], highly available servers [Mishra and Pang 1999; Anker et al. 1999; Fekete and Keidar 2001], and collaborative computing applications such as drawing on a shared whiteboard [Shamir 1996], video and audio conferences [Chodrow et al. 1997; Valenci 1998], application sharing [Krantz et al.1998, 1997], and even distributed musical "jam sessions" over a network [Gang et al. 1997].

In contrast, applications that maintain globally consistent shared state (e.g., Friedman and Vaysburg [1997], Keider and Dolev [1996, 2000], Amir et al. [1994], Fekete et al. [1997], Khazan et al. [1998], Schiper and Raynal [1996], Guerraoui and Schiper [1995, 1997b], Kemme and Alonso [1998], and Keidar [1994]), usually avoid inconsistencies by allowing only members of one view (the primary one) to update the shared state at a given time (see discussion in Hiltunen and Schlichting [1995]). For the benefit of such applications, some partitionable membership services (e.g., Friedman and van Renesse [1995] and Hiltunen and Schlichting [1995]) notify processes whether they are in a primary view, such that the primary views satisfy Property 3.4 (Primary Component Membership) above. The dynamic voting-based algorithm of Yeger Lotem et al. [1997] 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 nonprimary 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 [Chockler et al. 1996] extends Transis by providing a framework for synchronization of messages with different QoS properties; Maestro [Birman et al. 1998] extends Ensemble by coordinating several protocol stacks with different QoS guarantees, and the collaborative computing transport layer (CCTL) [Rhee et al. 1997] 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 a preceding **send** event of the same message:

$$t_i = \mathbf{receive}(p, m) \Rightarrow \exists q \exists j\, (j < i \wedge t_j = \mathbf{send}(q, m)).$$

This property is trivially implemented, and all GCSs support it; it is explicitly specified in Babaoğlu et al. [1998b], Rodrigues and Verissimo [1992], Fekete et al. [1997], De Prisco et al. [1998], and Keidar and Khazan [2000].

The following property states that messages are not duplicated by the GCS; 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:

$$t_i = \mathbf{recv}(p, m) \wedge t_j = \mathbf{recv}(p, m) \Rightarrow i = j.$$

Most GCSs eliminate duplication (some examples are: Babaoğlu et al. [1998b], Ezhilchelvan et al. [1995], Amir et al. [1992b], and Keidar and Khazan [2000]). 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.

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.[4] 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:

$$\mathit{receives\_in}(p, m, V) \wedge \mathit{sends\_in}(q, m, V') \Rightarrow V = V'.$$

---

[4] 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.

Among the group communication systems that support Sending View Delivery are Isis and Totem. In contrast, Newtop and RMP do not guarantee Property 4.3. Horus allows the user to choose whether this property should be satisfied; the programming model in which it is satisfied is called strong virtual synchrony (SVS) [Friedman and van Renesse 1995]. Property 4.3 also appears in various GCS specifications (examples include Moser et al. [1994], Fekete et al. [1997], Hiltunen and Schichting [1995], De Prisco et al. [1998], and Keidar and Khazan [2000]).

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 Amir et al. [1997], Sussman and Marzullo [1998], Hiltunen and Schlichting [1995], Friedman and Vaysburg [1997], Amir et al. [1997, 1993], Keidar and Dolev [1996, 2000], and Khazan et al. [1998]). 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 ith entry in the vector corresponds to the ith member in the current view (as explained in Friedman and van Renesse [1995]). Applications that exploit Sending View Delivery are called *view-aware*.

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 [1995] 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 Sending View Delivery, 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.

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, Relacs, and the GCSs that support Property 4.3 above. (Same View Delivery is called Uniqueness in Babaoğlu et al. [1998b]).

Same View Delivery is strictly weaker than Sending View Delivery. However, it is sufficient for applications that are not interested in knowing in which view messages are multicast; some examples are: Chockler et al. [1998], Keider and Dolev

[1996, 2000], Amir et al. [1996], and Anker et al. [1999].

Sussman and Marzullo [1998] 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, whereas 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. This identifies a tradeoff between the costs of totally ordered multicast and 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 (e.g., Transis and Spread). 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 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.

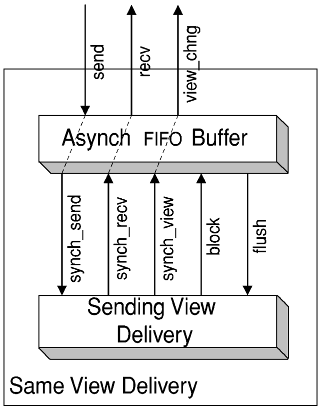

Fig. 2. Implementing Same View Delivery over Sending View Delivery.

4.2.3. The Weak Virtual Synchrony and Optimistic Virtual Synchrony models. The weak virtual synchrony (WVS) programming model [Friedman and van Renesse 1995] eliminates the need for blocking, and yet provides support for a certain type of view-aware applications. In WVS, every installation of a view V is preceded by at least one suggested view event. The membership of the suggested view is an ordered superset of V. Property 4.3 (Sending View Delivery) is replaced by the requirement that every message sent in the suggested view be 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 (e.g., Yeger Lotem et al. [1997]) 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 in Sussman et al. [2000]. 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 (e.g., 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 of Sussman and Marzullo [1998], algorithms that exploit WVS or OVS perform the same as those 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:

$$\begin{aligned}&\mathit{installs\_in}(p, V, V') \wedge \mathit{installs\_in}(q, V, V')\\&\quad\wedge \mathit{receives\_in}(p, m, V')\\&\quad\Rightarrow \mathit{receives\_in}(q, m, V').\end{aligned}$$

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 [Birman and Joseph 1987] in the context of a primary component membership service and later extended to a partitionable membership service [Friedman and van Renesse 1995; Dolev et al. 1995; Ezhilchelvan et al. 1995; Moser et al. 1994; Babaoğlu et al. 1998b]. In Moser et al. [1994] and Friedman and Vaysburg [1997] it is called "failure atomicity," and in Babaoğlu et al. [1998b] it is called "message agreement." Virtual synchrony is supported by nearly all group communication systems, either for all multicast services (e.g., in Ensemble, Horus, Isis, Newtop, Phoenix, Relacs, Totem, and Transis) or only for some multicast services, such as the totally ordered multicast of RMP. It also appears in specifications, for example, Hiltunen and Schlichting [1995], Hickey et al. [1999], Keidar and Khazan [2000], and Galleni and Powell [1996]. An exception is set by the specifications of Fekete et al. [1997] and De Prisco et al. [1998] which do not include this property.

Virtual synchrony is especially useful for applications that implement data replication using the state machine approach [Lamport 1978; Schneider 1990] (examples include Keidar and Dolev [1996, 2000], Amir et al. [1994, 1997, 1993], Friedman and Vaysburg [1997], Khazan et al. [1998], and Sussman and Marzullo [1998]). Such applications change their state when they receive application messages. In order to keep the replica in a consistent state, application

---

[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.

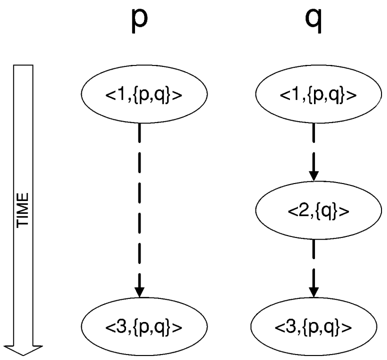

Fig. 3. A possible scenario with a partitionable GCS.

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 Amir et al. [1997]: 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 reconnects 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 $\langle 1, \{p, q\}\rangle$.

Thus, 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 from 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:

   $$t_i = \textbf{view\_chng}(p, V, T) \wedge \mathit{viewof}(t_i) = V' \;\Rightarrow\; T \subseteq V.\mathit{members} \cap V'.\mathit{members}.$$

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:

   $$t_i = \textbf{view\_chng}(p, V, T) \wedge \mathit{viewof}(t_i) = V' \wedge \mathit{installs\_in}(q, V, V'') \Rightarrow (q \in T \Leftrightarrow V' = V'').$$

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. If a process p in $V.members \cap V'.members$ does not install V', Property 4.6 does not specify whether p is included in transitional sets of other processes.

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 (see Amir et al. [1997] 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 [Moser et al. 1994]. This model is implemented in Transis and Totem. Later, Babaoğlu et al. [1996] [Babaoğlu et al. 1996] 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 Cristian and Schmuck [1995] also convey the previous view of every view member. The transitional set can be deduced from these views. The transitional set is also specified in Amir et al. [1997] and Keidar and Khazan [2000].

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:

$$installs\_in(p, V, V') \wedge installs(q, V) \wedge q \in V'.members \Rightarrow installs\_in(q, V, V').$$

Property 4.7 (Agreement on Successors) holds in Horus [Friedman and Vaysburg 1997], Ensemble [Hickey et al. 1999], and Relacs [Babaoğlu et al. 1998b].[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 may require processes to deliver extra views that exclude live and connected processes. Consider the example in Figure 3 above: p does not suspect q, but in order to satisfy the Agreement on Successors property, p would have to install a view without q before installing the correct view with q.

5. SAFE MESSAGES

Distributed applications often require "all or nothing" semantics; that is, either all the processes deliver a message or none of them do so. Unfortunately, "all or nothing" semantics is impossible to achieve in distributed systems in which messages may be lost. As an approximation to "all or nothing" semantics, the EVS model [Moser et al. 1994] 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. This "approximated" semantics is called Safe Delivery in Moser et al. [1994] and Total Resiliency in Whetten et al. [1995].

---

[6] In Hickey et al. [1999] and Babaoğlu et al. [1998b], 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.

Table II. Predicate Definitions for Safe Messages

Process p receives message m before message m':

$$recv\_before(p, m, m') \overset{\text{def}}{=} \exists i \exists j\ (t_i = \mathbf{recv}(p, m) \wedge t_j = \mathbf{recv}(\mathbf{p}, \mathbf{m'})\ \wedge\ i < j)$$

Process p receives message m before message m', both of them in view V:

$$recv\_before\_in(p, m, m', V) \overset{\text{def}}{=} \exists i \exists j\ (t_i = \mathbf{recv}(p, m) \wedge t_j = \mathbf{recv}(\mathbf{p}, \mathbf{m'}) \wedge viewof(t_i) = viewof(t_j) = V\ \wedge\ i < j)$$

A message m received in a view V is indicated as safe at process p:

$$indicated\_safe(p, m, V) \overset{\text{def}}{=} receives\_in(p, m, V)\ \wedge \exists i\ (t_i = \mathbf{safe\_prefix}(p, m) \vee \exists m'\ (t_i = \mathbf{safe\_prefix}(p, m')\ \wedge\ recv\_before\_in(p, m, m', V)))$$

Message m is stable in view V:

$$stable(m, V) \overset{\text{def}}{=} \forall p \in V.members(receives(p, m))$$

In this article we follow the approach of Fekete et al. [1997] 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 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 II.

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:

$$indicated\_safe(p, m, V) \Rightarrow stable(m, V).$$

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.

Consistent replication applications (e.g., Keidar and Dolev [1996] and Amir et al. [1994]) 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).

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:

$$indicated\_safe(p, m, V) \wedge recv\_before\_in(q, m', m, V) \Rightarrow stable(m', V).$$

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

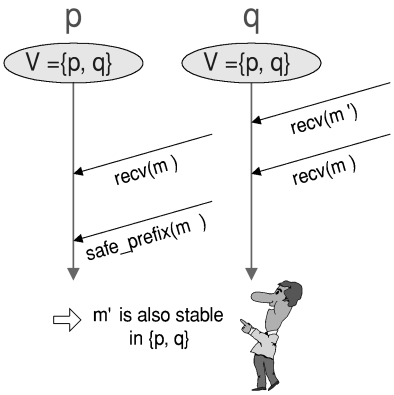

Fig. 4. The Safe Indication Reliable Prefix property.

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 [Amir et al. 1995; Moser et al. 1994] and Transis, the *Totally Resilient* QoS level of RMP, the *atomic, tight*, and *delta* QoS levels of xAMp, and the *Uniform* multicast of Phoenix [Malloth et al. 1995]. Safe delivery is also guaranteed by Horus if one uses the ORDER layer above the STABLE layer.

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. The *N resilient* QoS level of RMP and *atLeastN* QoS level of xAMp 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.

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 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, that is, when k members have received the message.

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

We note that the reliability properties do not imply the corresponding order properties. This is because the former properties apply only to messages that are sent within the same view, and the latter apply to all messages. For example, FIFO delivery requires that all messages sent by a single source be delivered in the order in which they were sent, whereas reliable FIFO prohibits gaps in the FIFO order only within a single view. Prohibiting gaps across views would require the GCS to log messages and retransmit them to new processes at view changes. GCSs generally do not log messages. Instead, services that provide gap-free communication

---

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

across views are often implemented atop GCSs (e.g., in Keidar and Dolev [1996] and Amir et al. [1994]).

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, for example, Property 4.3 (Sending View Delivery). Similar reliable ordering 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 (e.g., 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 (e.g., Isis) provide different primitives for sending messages of different service types; others (e.g., Transis) provide one *send* primitive and allow the application to tag the message sent with the requested service type; whereas in other systems (e.g., Horus and Ensemble), a different protocol stack is constructed for each service type, and a communication endpoint (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 through 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 before m'. Formally:

$$t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p, m') \\ \wedge\, i < j \wedge \mathit{viewof}(t_i) = \mathit{viewof}(t_j) \\ \wedge\, \mathit{receives}(q, m') \Rightarrow \mathit{recv\_before}(q, m, m').$$

Several group communication systems (e.g., Ensemble, Horus, and RMP) provide a reliable FIFO service type that satisfies Property 6.2 and does not impose additional ordering constraints. xAMp provides several service levels that satisfy Property 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 [Ezhilchelvan et al. 1995; Chockler et al. 1998] are often constructed over a reliable FIFO service.

6.2. Causal Multicast

The causal order (first defined in Lamport [1978]) extends the FIFO order by requiring that a response m' to a message m is always delivered after the delivery of m. The causal order of events is formally defined in Table III.

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

Table III. Causal Order, Recursive Definition

$$t_i \rightarrow t_j \overset{\text{def}}{=} (pid(t_i) = pid(t_j) \wedge j \geq i) \vee (t_i = \mathbf{send}(p, m) \wedge t_j = \mathbf{recv}(q, m)) \vee \exists k\, (t_i \rightarrow t_k \wedge t_k \rightarrow t_j)$$

Table IV. Timestamp (TS) Function Definition

A timestamp (TS) function is a one-to-one function from $\mathcal{M}$ to the set of natural numbers:

$$TS\_function(f) \overset{\text{def}}{=} f : \mathcal{M} \rightarrow \mathcal{N} \wedge f(m) = f(m') \Rightarrow m = m'$$

causally precedes m', then every process that receives both these messages, receives m before m'. Formally:

$$\begin{aligned} t_i &= \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p', m') \\ &\wedge t_i \rightarrow t_j \wedge t_k = \mathbf{recv}(q, m) \\ &\wedge t_l = \mathbf{recv}(q, m') \Rightarrow k < l. \end{aligned}$$

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 before m'. Formally:

$$\begin{aligned} t_i &= \mathbf{send}(p, m) \wedge t_j = \mathbf{send}(p', m') \\ &\wedge t_i \rightarrow t_j \wedge viewof(t_i) = viewof(t_j) \\ &\wedge receives(q, m') \Rightarrow recv\_before(q, m, m'). \end{aligned}$$

The CBCAST (causal broadcast) primitive of Isis [Birman and Joseph 1987] 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, Ensemble, Horus, Newtop, and xAMp.

6.3. Totally Ordered Multicast

Group communication systems usually provide a totally ordered (atomic, agreed) service type that 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). For a comprehensive survey of totally ordered multicast protocols and specifications, see Défago et al. [2000].

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). In Section 6.3.2 we discuss a stronger alternative (Reliable Total Order).

In Table IV we define a *timestamp* (*TS*) *function* to be a one-to-one function from $\mathcal{M}$ to the natural numbers. We use such functions to define a total order of messages.

6.3.1. Strong and Weak Total Order.

Wilhelm and Schiper [1995] introduce a classification of totally ordered multicast. In particular, they define *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 is a TS function f such that messages are received at all the processes in an order consistent with f. Formally:

$$\begin{aligned} &\exists f\, (TS\_function(f) \wedge \forall p \forall m \forall m' \\ &\quad (recv\_before(p, m, m') \Rightarrow f(m) < f(m'))). \end{aligned}$$

Note that the TS function merely exists: we do not require that the timestamp values be conveyed to the application. Some applications (e.g., the replication algorithm of Keidar and Dolev [1996]), do require that message timestamps be available to them. The ATOP algorithm [Chockler et al. 1998] which implements totally ordered multicast in Transis conveys timestamps to its application. These timestamps are unique and taken from a totally ordered set, but are not integers, and thus do not correspond to the timestamps given by f.

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 receive messages in the same order. The property has two parts: it specifies that processes that move together from a view V' to another view V receive messages in V' in the same order, and it specifies that processes that remain in the same view V forever, (i.e., V is their last view) receive the messages in this view in the same order. Like Strong Total Order, Weak Total Order is defined using timestamp functions. However, unlike Strong Total Order, there is no requirement for one universal timestamp function. Rather, there can be different timestamp functions for each pair of views V' and V, and for each last view V.

We use the following auxiliary shorthand definition.

Definition 6.1 (*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:

$$last\_view(p, V) \overset{\text{def}}{=} \exists i \exists T \; (t_i = \mathbf{view\_chng}(p, V, T) \wedge \nexists j > i \; \exists T' \exists V' \; t_j = \mathbf{view\_chng}(p, V', T')).$$

We now define Weak Total Order.

PROPERTY 6.6 (*Weak Total Order*).

1. For every pair of views V and V' there is a TS function f so that every process that installs V in V' receives messages in V' in an order consistent with f. Formally:

$$\forall V \forall V' \exists f \, (TS\_function(f) \wedge \forall p \forall m \forall m' \; (installs\_in(p, V, V') \wedge recv\_before\_in \; (p, m, m', V') \Rightarrow f(m) < f(m')).$$

2. For every view V there is a TS function f so that every process that has V as its last view receives messages in V in an order consistent with f. Formally:

$$\forall V \exists f \, (TS\_function(f) \wedge \forall p \forall m \forall m' \; (last\_view(p, V) \wedge recv\_before\_in \; (p, m, m', V) \Rightarrow f(m) < f(m')).$$

Applications that exploit GCSs for consistent replication require that processes agree upon the order of messages even if they disconnect from each other [Keidar and Dolev 1996; Amir et al. 1994; Fekete et al. 1997]; 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. For applications that do allow copies of the shared state to diverge while there are partitions, for example, Amir et al. [1997], Anker et al. [1999], and Fekete and Keidar [2001], Weak Total Order suffices.

Strong Total Order is provided by Totem and by some of the implementations of totally ordered multicast in Transis, Ensemble, Phoenix, RMP, and Horus. Many GCSs provide a weak totally ordered multicast service, for example, the ABCAST (atomic broadcast) primitive of Isis, similar primitives in Amoeba [Kaashoek and Tanenbaum 1996], Newtop, and xAMp, and certain implementations of totally ordered multicast in Transis, Ensemble, Phoenix, RMP, and Horus.

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 extends the Strong Total Order property to require 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', messages m and m' were sent in the same view, and $f(m) < f(m')$, then q receives m before m'. Formally:

$$\exists f\,(\mathit{TS\_function}(f) \wedge \forall V \forall m \forall m' \forall p \forall p' \forall q \; (\mathit{sends\_in}(p, m, V) \wedge \mathit{sends\_in}(p', m', V) \wedge \mathit{receives}(q, m') \wedge f(m) < f(m') \Rightarrow \mathit{recv\_before}(q, m, m'))).$$

In the Appendix, we prove Lemma A.1 which states that Property 6.7 (Reliable Total Order) along with Property 4.3 (Sending View Delivery) and the basic Property 4.1 (Delivery Integrity) imply Property 6.5 (Strong Total Order) for messages received in the same view. We also prove Lemma A.2 which asserts that Properties 6.7 (Reliable Total Order) and 6.2 (Reliable FIFO) 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 Property 6.4 (Reliable Causal).

Unfortunately, implementing Reliable Total Order imposes a performance penalty: in order to support Reliable Total Order, existing total order algorithms would be forced to deliberately discard messages from live and connected processes. Therefore, no GCS we are aware of guarantees Property 6.7. The only specifications that require Reliable Total Order are those of Fekete et al. [1997].

The Reliable Total Order property is exploited by the replication application in Fekete et al. [1997]; it guarantees that operations will be applied to the database in a consistent order without gaps. However, the application in Fekete et al. [1997] could have been satisfied with a weaker property: In Keidar and Dolev [1996, 2000] and Amir et al. [1994] 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 the aforementioned applications [Keidar and Dolev 1996, 2000; Fekete et al. 1997; Amir et al. 1994] 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 Friedman and Vaysburg [1997], 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 [1995] 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, and
—*strong incorporated*: m_1 and m_2 are delivered according to O_2.

For example, RMP supports weak incorporated semantics between any two messages of different service levels. Isis provides 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 may be configured to use one of several protocols providing totally ordered multicast. The more efficient ATOP protocol [Chockler et al. 1998] guarantees only weak incorporated semantics between a reliable causal message and a strong totally ordered message. A protocol based on Lamport's [1978] logical timestamps guarantees strong incorporated semantics between messages of these two types, but it incurs longer delivery latency. Highways [Ahuja 1993] 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 to which group (or groups) the message is being sent. 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 [Guerraoui and Schiper 2000] requires messages sent in different groups to be delivered in the same order at all their destinations. For example, assume that processes p and q are both members of two different multicast groups $g1$, $g2$. Assume also that message $m1$ is sent in group $g1$, 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 [2000] prove that fault-tolerant atomic multicast is costly: unless additional assumptions (such as reliable failure detection or reliable groups) are imposed on the model, solving atomic multicast requires sending messages to additional processes that are not members of the group to which the message is being sent. Protocols that solve atomic multicast without involving additional members other than those a message is being sent to (e.g., Fritzke et al. [1998] and Guerraoui and Schiper [2000]) do impose such additional assumptions and generally do not work in a partitionable environment.

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 lightweight 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 lightweight group in which the message was sent.

Horus provides users with the flexibility to choose whether atomic multicast will be provided by constructing different protocol stacks: If atomic multicast is desired, a lightweight group layer is used above the total order layer in the stack. Thus, messages are first sent to the members of the heavyweight group where they are totally ordered and then they are multiplexed to the different groups. If atomic multicast is not desired, the lightweight group layer is stacked below the total order layer, and messages are totally ordered in their destination groups.

GCSs that use a lightweight group structure typically allow users to send a message to multiple lightweight groups. This service is implemented by sending messages to the heavy-weight (or daemon) group, and then multiplexing messages to the appropriate lightweight group. Johnson et al. [1999] 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.

Virtually all group communication systems provide causally ordered multicast (see Kshemkalyani and Singhal [1998]), that is, preserve the causality of messages sent in different groups. However, recently, Kalantar and Birman [1999] 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 the latency for delivering messages without such order constraints.

LIVENESS PROPERTIES OF GROUP COMMUNICATION SERVICES

7. INTRODUCTION

In this part of the survey 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 useful.

In order to specify meaningful liveness properties, we envision an ideal GCS, and try to capture its ideal behavior. 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 “correct” 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.[8]

In Section 10, we present two types of liveness properties. The first kind of properties requires that the GCS behave as the ideal GCS envisioned above, but 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.) The second type of liveness properties complements the former by requiring a weaker form of liveness in unstable runs.

In executions in which the network does eventually stabilize, 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 8.4), which is known to be nonimplementable in our environment. Our impossibility result is not surprising. In fact, Chandra et al. [1996] 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 as an eventually perfect one. Similar assumptions were also proposed in Malloth and Schiper [1995] and Babaoğlu et al. [1998b]; see the detailed discussion in Section 10.

It is important to note that although conditional 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 such properties, a group membership implementation has to attempt to be precise in every execution as it can never know whether there is a stable component and whether the failure

---

[8] Conditional liveness specifications of GCSs also appear in Fekete et al. [1997], Cristian and Schmuck [1995], Keidar et al. [2000], Keidar and Khazan [2000], and Babaoğlu et al. [1998b].

detector behaves as an eventually perfect one. Moreover, conditional liveness properties are composable: they allow one to reason about application liveness under the same external conditions that the GCS is live.

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. We model the network and the failure detector together, as a single automaton. Although in reality these could be implemented as separate components, from the point of view of the GCS both comprise the environment, so it is convenient to reason about the composition of the two. Several GCSs are built atop layers that provide both network and failure detector functionalities, for example, the Multi-Send Layer of Babaoğlu et al. [1998b] and the MUTS layer of Horus. We discuss failure detector implementation issues in Section 8.4.1.

An automaton with the external signature presented in Section 2 satisfying 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.

The network is modeled as a set of unidirectional channels that connect every ordered pair of processes in the system. A channel between two processes represents the collection of all network paths between the processes. We assume that the underlying network provides an asynchronous datagram service. Messages may be delivered out of order, and may be duplicated; there is no bound on message transmission time. Furthermore, the communication channels can go down, in which case messages can be be lost. Channels can go up and down any number of times. However, if a channel is up and remains up from some point in an execution onward, then every message sent on this channel after this point eventually reaches its destination. We state this assumption formally below.

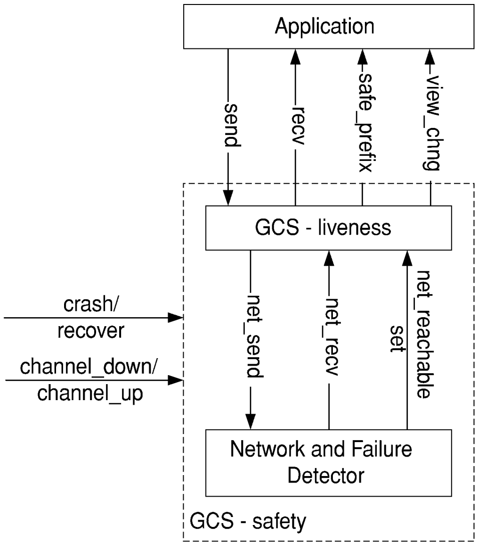

Fig. 5. Extending the external signature of the GCS to specify liveness.

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. We then formally define the prerequisites for the liveness properties: in Section 8.3 we define stable components, and in Section 8.4, eventually perfect failure detectors.

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 that 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

Table V. Predicates Describing the Network Situation

Process p is alive after the ith event in the trace:

$$alive\_after(p,i) \stackrel{\text{def}}{=} \nexists j\,(t_j = \mathbf{crash}(p)) \vee \exists j \le i\,(t_j = \mathbf{recover}(p) \wedge \nexists k > j\,(t_k = \mathbf{crash}(p)))$$

Process p is crashed after the ith event in the trace:

$$crashed\_after(p,i) \stackrel{\text{def}}{=} \exists j \le i\,(t_j = \mathbf{crash}(p) \wedge \nexists k > j\,(t_k = \mathbf{recover}(p)))$$

The channel from p to q is up after the ith event in the trace:

$$up\_after(p,q,i) \stackrel{\text{def}}{=} \nexists j\,(t_j = \mathbf{channel\_down}(p,q)) \vee \exists j \le i\,(t_j = \mathbf{channel\_up}(p,q) \wedge \nexists k > j\,(t_k = \mathbf{channel\_down}(p,q)))$$

The channel from p to q is down after the ith event in the trace:

$$down\_after(p,q,i) \stackrel{\text{def}}{=} \exists j \le i\,(t_j = \mathbf{channel\_down}(p,q) \wedge \nexists k > j\,(t_k = \mathbf{channel\_up}(p,q)))$$

network, and also receives failure detection information from the failure detector:

—output $\mathbf{net\_send}(p, m)$, $p \in \mathcal{P}, m \in \mathcal{M}$

—input $\mathbf{net\_recv}(p, m)$, $p \in \mathcal{P}, m \in \mathcal{M}$

—input $\mathbf{net\_reachable\_set}(p, S)$, $p \in \mathcal{P}$, $S \in 2^{\mathcal{P}}$.

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 2.3 is extended by adding the following to the **Events** set.

$$\{\mathbf{channel\_down}(p,q) \mid p,q \in \mathcal{P}\} \cup \{\mathbf{channel\_up}(p,q) \mid p,q \in \mathcal{P}\} \cup \{\mathbf{net\_send}(p,m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup \{\mathbf{net\_recv}(p,m) \mid p \in \mathcal{P}, m \in \mathcal{M}\} \cup \{\mathbf{net\_reachable\_set}(p,S) \mid p \in \mathcal{P}, S \in 2^{\mathcal{P}}\}.$$

Notation. We define some shorthand predicates that describe the network situation in Table V. Note that according to these definitions, processes are initially alive and channels are initially up.

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. Stable Components

As explained above, our liveness properties require "ideal" behavior from the GCS only if a stable component eventually exists and the failure detector behaves as an eventually perfect one. We now formally define a stable component.

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 channels 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

$$stable\_component(S) \stackrel{\text{def}}{=} \exists i \forall p \in S\ (alive\_after(p,i) \wedge \forall q \in S\ up\_after(p,q,i) \wedge \forall\, q \in \mathcal{P} \setminus S\ (down\_after(q,p,i) \vee crashed\_after(q,i))).$$

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 1 below imposes no restrictions on the failure detector's behavior.

It is common to assume transitivity, although it is not necessary. For example, Phoenix [Malloth and Schiper 1995] 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. Although in wide area networks lack of symmetry may occasionally occur, all the specifications that we are aware of do not require membership to be precise in such cases.

8.4. 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. We now classify traces in which the failure detector behaves as an eventually perfect one. For the sake of specifying such traces, we examine the composition of the failure detector with the network, and classify traces in which the reachable set reported by the failure detector eventually corresponds to the network situation.

Definition 8.2 (*Eventually perfectlike trace*). The failure detector behaves as $\diamond P$ 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:

$$\begin{aligned}&\diamond P - like \overset{\text{def}}{=} \forall S\,(stable\_component(S) \Rightarrow\\ &\quad \forall p \in S\, \exists i\, (t_i = \mathbf{net\_reachable\_set}(p, S)\\ &\quad \wedge \neg(\exists S' \neq S\, \exists j > i\; t_j\\ &\quad = \mathbf{net\_reachable\_set}(p, S')))).\end{aligned}$$

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

We now define an eventually perfect failure detector to be a composition of a failure detector and a network, so that in all the traces of this composition, the failure detector behaves as $\diamond P$, with respect to the network situation.

Definition 8.3 (*Eventually perfect failure detector*). *An eventually perfect failure detector* is a network and failure detector automaton that behaves as $\diamond P$ in every trace.

Chandra and Toueg [1996] define several classes of unreliable failure detectors for the crash-failure model. It is easy to see that, when restricted to the crash-failure model, our definition of $\diamond P$ coincides with the one in Chandra and Toueg [1996], since in every execution in that model all the correct processes form a stable component (once the last faulty process fails).

The definition of eventually perfect failure detectors is extended to partitionable environments in Dolev et al. [1997] and Babaoğlu et al. [1998b], and the definitions presented herein are very similar to those. The main difference is in the modeling formalism, more specifically, in the definition of when a channel is considered to be up. Our definition of stable components is stated explicitly in terms of **channel_down** and **channel_up** events, whereas the models in Dolev et al. [1997] and Babaoğlu et al. [1998b] do not include such events, and connectivity (reachability) is defined in terms of whether the last message sent on a channel reaches its destination.

Another difference is that the definition of Babaoğlu et al. [1998b] requires the failure detector to eventually precisely detect pairwise reachability among two processes even if a stable component does not exist. It is easy to see that this definition is stronger than ours: an eventually perfect failure detector as defined by Babaoğlu et al. [1998b] is also an eventually perfect failure detector according to our definition. The stronger notion of failure detector as defined in Babaoğlu et al. [1998b] is required for implementing Property 10.2 (View Accuracy), which does not depend on stable components. For space limitations, we do not include this definition here.

The classical approach to failure detectors [Chandra and Toueg 1996] requires an oracle failure detector (e.g., an eventually perfect one) to exist as part of the system model. In contrast, we do not require an eventually perfect failure detector to exist. Rather, we assume an arbitrary failure detector and condition our liveness specification on the failure detector's behavior in a given trace. Note that the difference between the two approaches is small. Clearly, any algorithm that meets the specification in an environment where a failure detector of class $\diamond P$ exists, also meets our conditional specification. Thus our conditional specification is not weaker than a classical one.

8.4.1. On Implementing a Failure Detector. In general, since it is impossible to implement $\diamond P$ in an asynchronous model with process failures, it is also impossible to implement eventually perfect failure detectors as defined above in the asynchronous model of this article. However, in practical networks, communication tends to be stable and timely during long periods. Partial synchrony models [Dwork et al. 1988] capture such network behavior. In such models, processes can measure time, and a bound on communication latency eventually exits.

Eventually perfect failure detectors are easily implemented in these partial synchrony models, over a network that satisfies Assumption 8.1. Failure detector implementations use the network in order to send and receive messages,[9] and they generate **net_reachable_set** events whenever they change their mind about network connectivity. A network and failure detector automaton can be obtained as a composition of such a failure detector module with the underlying network, by hiding actions related to messages of the failure detector.

Chandra and Toueg [1996] present an algorithm implementing an eventually perfect failure detector in the crash-failure partial synchrony model where eventually there is a bound on message transmission time, but this bound is not known to the processes. Babaoğlu et al. [1998b] present a variant on this algorithm, adapted to the link failure model. It works roughly as follows.

ALGORITHM 8.1. *Each process has an approximated bound on round-trip latency* Δ_p. *Every process p periodically multicasts a* $\mathbf{ping_p}$ *message to all other processes. Every process q responds to such a message by sending an* $\mathbf{ack_q}$ *message to p. If p does not receive an* $\mathbf{ack_q}$ *message within* Δ_p *time of sending* $\mathbf{ping_p}$*, p suspects q (i.e., if q is in p's reachable set, p removes q from its reachable set). Once p receives such a response, if q is not in p's reachable set, then p adds q to the reachable set and increases* Δ_p *by one second.*

It is easy to see that if a stable component eventually exists and a bound on message latency eventually holds, then Δ_p can increase only a finite number of times, and p's reachable set eventually contains exactly the set of processes in p's connected component. Hence, the algorithm implements an eventually perfect failure detector.

This algorithm is not used in practice, however; failure detector implementations generally use smaller timeouts, at the risk of occasionally having false suspicions. Practical systems often do have an expected bound on latency, which holds at "stable" times. During "unstable" periods, messages can be delayed longer than this bound. This system behavior is captured by the timed asynchronous system model of Cristian and Fetzer [1999]. In this model, it is possible to build failure detectors that behave as eventually perfect ones during stable periods.

Note also that the network and failure detector automaton has certain functionalities: (1) an eventually reliable communication protocol that ensures Assumption 8.1, that is, that messages sent on channels that are up eventually reach their destinations; and (2) a failure detector. These functionalities can be implemented separately, as explained above. However, they are often implemented jointly by the same service, over an unreliable network. Examples of such

---

[9] Obviously, the failure detector implementation cannot see **channel_up** and **channel_down** events.

services include the MUTS layer of Horus, the Multi-Send Layer of Babaoğlu et al. [1998b], and the Core layer of Xpand [Anker et al. 2000]. TCP implements a similar service over the unreliable IP protocol: TCP uses retransmissions in order to guarantee that messages reach their destination while the channel is up. If the channel goes down, the TCP connection goes down, thus reporting the failure to the application. If a channel is up but slow, TCP can mistakenly report a failure where there is none.

9. PRECISE MEMBERSHIP IS AS STRONG AS $\diamond P$

We now justify the use of eventually perfect failure detectors as a prerequisite for liveness. 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 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 suitable only for partitionable membership services as it requires members of all stable components to install views.

Definition 9.1 (*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 *member* set S such that V is the last view of every process p in S. Formally:

$$stable\_component(S) \Rightarrow \exists V\ (V.members = S \wedge \forall p \in S\ last\_view(p, V)).$$

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. We begin with a group membership service implemented atop a network and failure detector automaton. We hide the **net_reachable_set** events, so that they will not appear in traces. Then, for each process p, we construct an automaton $\mathbf{MEMBtoFD}_p$ (see Figure 6). $\mathbf{MEMBtoFD}_p$ receives **view_chng** events from the group membership and generates **net_reachable_set** events as follows. Whenever a $\mathbf{view\_chng}(p, V, T)$ occurs, $\mathbf{net\_reachable\_set}(p, V.members)$ is generated. We compose the **MEMBtoFD** automata of all the processes with the group membership service.

We now show that if the membership service is precise, every generated trace of this composition is $\diamond P$-like. Let p be a process. If p is not a member of a stable component, there are no restrictions on the failure detector's behavior. Assume that there exists a stable component S such that $p \in S$; then by Precise Membership, p installs a last view V with $V.members = S$. Thus p generates $\mathbf{net\_reachable\_set}(p, S)$ 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 process failure model. In that model, the set of correct processes forms a stable component in every execution. Thus, a precise membership service in that model is required to deliver to all the correct processes a last view consisting of exactly the correct processes.

Note that it is possible to implement a precise membership service using an eventually perfect failure detector: Section 10.3 surveys many examples of group communication systems that provide precise membership services when the failure detector they employ behaves as an eventually perfect one. GCS liveness is also specified using external failure detectors in Schiper and Rocciardi [1993], Malloth and Schiper [1995], Babaoğlu et al. [1998b], and Hiltunen and Schlichting [1995].

10. LIVENESS PROPERTIES

We now specify liveness properties for partitionable GCSs (cf. Section 3.1.2). These

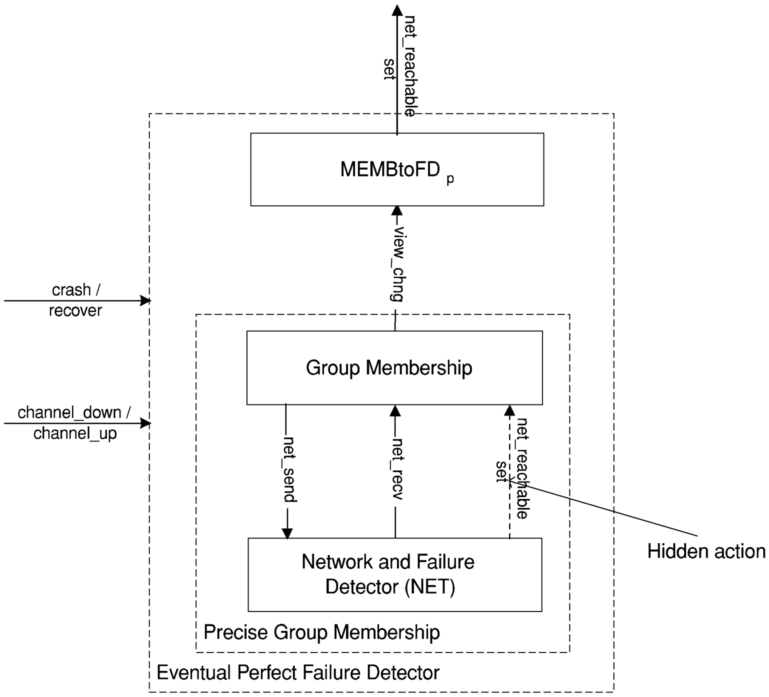

Fig. 6. Reducing precise membership to an eventually perfect failure detector.

properties are not suitable for primary component GCSs, as they require processes to install views in some situations even if they are not in a primary component. We do not specify liveness properties for a primary component GCS, since the liveness of such a service is dependent on the specific implementation and the policy it employs to guarantee Property 3.4 (Primary Component Membership). Note that primary component membership services block if they cannot form a primary view. For example, a primary component membership can block if the network partitions into three minority components or if all the members of the latest view[10] crash.

We define two kinds of liveness properties. In Section 10.1 we define liveness properties that are conditional on the existence of a stable component. In Section 10.2 we define complementary liveness properties in order to account for situations in which no stable component exists. In Section 10.3 we survey related work.

10.1. Liveness Properties for Stable Runs

In this section, we state four 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 (cf. Section 5). All of these properties are conditional; they are required to hold in runs in which there exists a stable component S and the failure detector behaves as $\diamond P$.

PROPERTY 10.1 (*Liveness*). If the failure detector behaves as $\diamond P$, 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:

$$\diamond P - like \wedge stable\_component(S)$$
$$\Rightarrow \exists V\ (V.members = S \wedge \forall p \in S$$

---

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

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:

$$t_i = \mathbf{send}(p, m) \wedge \nexists j > i\ t_j = \mathbf{crash}(p) \Rightarrow receives(p, m).$$

4. Safe Indication Liveness: Every message p sends in V is indicated as safe by every process in S. Formally:

$$sends\_in(p, m, V) \Rightarrow \forall q \in S\ indicated\_safe(q, m, V)).$$

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 Dwork et al. [1988] and Guerraoui and Schiper [1997a] However, we cannot explicitly bound this time period in an asynchronous model, because its duration depends on external conditions such as message latency, process scheduling, and processing time.

10.2. Additional Liveness Properties

10.2.1. Membership Accuracy. Property 10.1.1 (Membership Precision) guarantees that if a stable component eventually exists, the membership service installs a precise view at all the members in this component. When a stable component does not exist, most group communication systems still strive to provide meaningful views, even if these views may keep changing. This desirable behavior is captured by the following property, originally formulated in Babaoğlu et al. [1998b].

PROPERTY 10.2 (*Membership Accuracy*). If there is a time after which processes p and q are alive and the channel from q to p is up, then p eventually installs a view that includes q, and every view that p installs afterwards also includes q. Formally:

$$up\_after(q, p, i) \wedge alive\_after(p, i) \wedge alive\_after(q, i) \Rightarrow \exists j\ \exists V\ \exists T\ (t_j = \mathbf{view\_chng}(p, V, T) \wedge q \in V.members \wedge \forall k > j\ \forall V'\ \forall T' (t_k = \mathbf{view\_chng}(p, V', T') \Rightarrow q \in V'.members)).$$

Implementing this property requires a failure detector that eventually provides precise information about pairwise reachability between two processes, even when a stable component does not exist. Such a failure detector is defined in Babaoğlu et al. [1998b]. For space limitations, we do not repeat this definition here.

Membership Accuracy does not require processes to eventually stop installing views. Hence, it does not imply Property 10.1.1 (Membership Precision). Moreover, while no stable component exists, Membership Accuracy does not require processes that are connected to each other to install the same view or to deliver each other's messages. This diminishes the usefulness of this property for applications. Membership Accuracy is provided by most GCSs. However, it is not provided by membership services that do not install views while a stable component does not exist (e.g., Keidar et al. [2000]).

10.2.2. Termination of Delivery. The following alternative to Property 10.1.2 (Multicast Liveness) was suggested in Friedman and van Renesse [1995], Dolev et al. [1995], and Babaoğlu et al. [1998b].

PROPERTY 10.3 (*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:

$$\begin{aligned}&\mathit{sends\_in}(p, m, V) \wedge q \in V.\mathit{members}\\&\quad \Rightarrow \mathit{delivers}(q, m) \vee \mathit{delivers\_in}(p, V)\\&\quad \vee \exists V'\, \mathit{installs\_in}(p, V', V).\end{aligned}$$

Membership Precision and Termination of Delivery together imply Multicast Liveness (Property 10.1.2). In addition, Property 10.3 (Termination of Delivery) requires that the membership service not block even when the network is unstable. We believe that this property is not particularly useful for applications: when the network is unstable, a membership service satisfying this property will continuously install views without any guarantee to deliver messages in these views. Continuously installing new views at unstable times may increase the load and lengthen the unstable period. Furthermore, any membership service that satisfies Property 10.3 is forced to install obsolete views, that is, views that are known to be changing soon. However, most existing membership algorithms do satisfy Property 10.3 (Termination of Delivery). An exception is the membership service of Keidar et al. [2000] which does not install a view if it knows that this view is already obsolete.

10.3. Related Work

10.3.1. Membership Precision and Accuracy. 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 timeouts or other methods (e.g., Vogels [1996]) in order to detect conditions under which the membership protocol should be invoked. The failure detector also provides an initial approximation of the view upon which the membership service would agree. 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.

Property 10.1.1 (Membership Precision) is explicitly specified in Anker et al. [1998]. The specification of Keidar et al. [2000] summarizes the two preconditions for precision—stable component and eventually perfect failure detection—into a single one. It requires that if a connected set S of processes exists, such that the **net_reachable_set** at every member of S remains S forever, then all members of S eventually install the same last view. This latter precondition is weaker than the original two preconditions, since the actual connected component may contain additional members that are not included in S.

The specifications of Friedman and van Renesse [1995] and Lin and Hadzilacos [1999] are also conditional on the failure detector output. For example, they require that a process q be excluded from a view only if the failure detector module at some view member suspects q. The specifications of Lin and Hadzilacos [1999] also require that if all active processes almost always suspect (do not suspect) q, then their views almost always do not include (resp., do include) q. This property is different from Membership Accuracy in that it only applies when all processes agree on the inclusion of some member, not whenever pairwise reachability is established between two processes. Like Membership Accuracy and unlike Membership Precision, this property does not require processes to eventually stop installing views. Also unlike Membership Precision, it requires liveness whenever processes agree on the inclusion of some member, even if there is no agreement upon the entire connected component.

Phoenix [Malloth and Schiper 1995] exploits a failure detector that is weaker than an eventually perfect one. Given the weaker failure detector, Phoenix guarantees progress but not precision: it guarantees that each invocation of the membership protocol will terminate, but correct processes may be removed from the membership and forced to rejoin infinitely many times. We observe, however, that in executions in which the network eventually stabilizes and the underlying failure detector used by Phoenix behaves as an eventually perfect one, Phoenix also satisfies Membership Precision.

The specifications of Fekete et al. [1997], Cristian and Schmuck [1995], and Mishra et al. [1998] guarantee precision of the membership service at periods during which the underlying network is stable and timely. These specifications are formulated in the timed asynchronous system model; they 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 survey we do not focus on a specific protocol, we cannot provide such an analysis.

10.3.2. Multicast and Safe Indication Liveness. Like Membership Precision, Property 10.1.2 (Multicast Liveness) is satisfied by all the existing GCSs, although it does not always explicitly appear in their specifications. This property eliminates trivial GCS implementations that capriciously discard messages without delivering them. Similar properties appear in Fekete et al. [1997] and Keidar and Khazan [2000].

In primary component GCSs, message stability may be formulated as follows. If a process delivers a message in view V, then all nonfaulty members of V eventually deliver this message. This is called Uniformity in the Isis literature and in Schiper and Sandoz [1993] and Unanimity in Rodrigues and Verissimo [1992].

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 delivery of messages sent in any view, not just those sent in the last view.

All the GCSs that we are aware of satisfy Self Delivery; some examples are: Isis, Transis, Totem, Horus, and Newtop. In RMP, Self Delivery holds for all multicast services except for the Unreliable one. However, this property does not hold in the specifications of Fekete et al. [1997].

Some specifications that include Sending View Delivery (e.g., Moser et al. [1994] and Keidar and Khazan [2000]) define self-delivery as a safety property that holds between each pair of consecutive views installed by a process. Since a process cannot know whether there will eventually be a stable component, in both cases a process must deliver the messages it sent in the current view before it installs the next view. Other specifications (e.g., Babaoğlu et al. [1998b]) require a process to deliver its own messages in all executions, not just stable ones. Again, 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 Fekete et al. [1997] as this is the only work that explicitly introduces safe indications.

CONCLUSIONS

11. SUMMARY

We have presented a comprehensive set of specifications that 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 30 published specifications that cover a dozen leading GCSs. We have correlated the terminology used in different papers with our terminology.

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

In order to account for the diverse requirements of different applications, we followed a modular paradigm in this survey: our specifications are divided into independent properties that may be used as building blocks for the construction of a

Table VI. Summary of Safety Properties of the Membership and Multicast Services

Basic Properties	Optional Properties
Self Inclusion	Primary Component Membership
Local Monotonicity	Sending View Delivery
Initial View Event	Virtual Synchrony
Delivery Integrity	Transitional Set
No Duplication	Agreement on Successors
Same View Delivery	

Table VII. Properties of Different Ordered Multicast Services and of Safe Message Indications

FIFO Multicast	Causal Multicast
FIFO Delivery Reliable FIFO	Causal Delivery Reliable Causal
Totally Ordered Multicast	Safe Indications
Strong Total Order Weak Total Order Reliable Total Order	Safe Indication Prefix Safe Indication Reliable Prefix

Table VIII. Summary of Liveness Properties

Basic Properties	Optional Properties
Membership Precision	Safe Indication Liveness
Multicast Liveness	Termination of Delivery
Self Delivery	Membership Accuracy

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 VII summarizes the properties of different ordering and reliability services (FIFO, causal, and totally ordered) we have described in this article, as well as safe message indications. In the future, our framework may be used for specifying additional qualities of service and semantics.

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 nontriviality 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 nontrivial and feasible to implement. In Table VIII we summarize the liveness properties.

The set of specifications presented here has been carefully assembled to satisfy the common requirements of numerous fault-tolerant distributed applications. Throughout the article, the specifications are justified with examples of applications that benefit from them.

We hope that the specifications framework presented in this article 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 here a guide to the services provided by a large variety of GCSs, which would help them choose 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.2, which states 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 article, 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.2, showing that a nonreliable totally ordered and FIFO ordered multicast service is also causally ordered. An interesting research direction would be to explore additional relationships and tradeoffs between different properties.

APPENDIX

A. PROVING A RELATIONSHIP BETWEEN DIFFERENT PROPERTIES

First, we prove that Property 6.7 (Reliable Total Order) implies Property 6.5 (Strong Total Order) for messages received in the same view.

LEMMA A.1 *Property* 6.7. (*Reliable Total Order*) *along with Property* 4.3 (*Sending View Delivery*) *and the basic Property* 4.1 (*Delivery Integrity*) *imply Property* 6.5 (*Strong Total Order*) *for messages received in the same view.*

PROOF. Let ts be the timestamp function f whose existence is given in Property 6.7 (Reliable Total Order). We now prove that $\forall p \forall m \forall m' (recv\_before\_in(p, m, m', V) \Rightarrow ts(m) < ts(m'))$, which will imply Property 6.5 (Strong Total Order).

First, $m \neq m'$; otherwise the same message is received twice which is a contradiction to Delivery Integrity (Property 4.1). Therefore $ts(m) \neq ts(m')$. Now assume by contradiction that $ts(m) > ts(m')$. Then, by Delivery Integrity (Property 4.1) there are $\mathbf{send}(q, m)$ and $\mathbf{send}(q', m')$, and by Sending View Delivery (Property 4.3) $viewof(\mathbf{send}(q, m)) = viewof(\mathbf{send}(q', m'))$. Hence, we can apply Reliable Total Order (Property 6.7) and conclude that $recv\_before(p, m', m)$. This contradicts the assumption that $recv\_before\_in(p, m, m', V)$. □

Similar proofs can be given to relate Property 6.2 (Reliable FIFO) with Property 6.1 (FIFO Delivery) and Property 6.4 (Reliable Causal) with Property 6.3 (Causal). We do not present these proofs here because they are trivial.

Now, 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.2 *Properties* 6.7 (*Reliable Total Order*) *and* 6.2 (*Reliable* FIFO) *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 Property* 6.4 (*Reliable Causal*).

PROOF. First, let us prove the following claims.

CLAIM A.2.1. *If* $t_i = \mathbf{recv}(p, m)$, $t_k = \mathbf{send}(p, m')$, $i < k$ *and* $viewof(t_i) = viewof(t_k)$, *then* $ts(m) < ts(m')$.

PROOF. First, $m \neq m'$, by Delivery Integrity (Property 4.1) since every message can be sent only once (by Message Uniqueness, Assumption 2.2). Since $m \neq m'$, $ts(m) \neq ts(m')$. Now, assume the contrary; that is, $ts(m) > ts(m')$. Then, by Reliable Total Order (Property 6.7), since there is $\mathbf{recv}(p, m)$, there is also $\mathbf{recv}(p, m')$ 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.2.2. *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. Immediate from Initial View Event and Strong Local Monotonicity. □

CLAIM A.2.3. *If* $\mathbf{send}(p, m) \to \mathbf{send}(p', m')$, *then there is a sequence of events either* $S1 = \mathbf{send}(p_1 = p, m_1 = m) \to \mathbf{send}(p_1, m'_1) \to \mathbf{recv}(p_2, m'_1) \to \mathbf{send}(p_2, m_2) \to \mathbf{recv}(p_3, m_2) \to \mathbf{send}(p_3, m_3) \to \cdots \to \mathbf{recv}(p_n = p', m_{n-1}) \to \mathbf{send}(p_n = p', m_n = m')$ *or* $S2 = \mathbf{send}(p_1 = p, m_1 = m) \to \mathbf{recv}(p_2, m_1) \to \mathbf{send}(p_2, m_2) \to \cdots \to \mathbf{recv}(p_n = p', m_{n-1}) \to \mathbf{send}(p_n = p', m_n = m')$.

PROOF. By the recursive definition of causal order (in Table III), 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 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 a **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 $t_i = \mathbf{send}(p, m) \to t_k = \mathbf{send}(p', m')$, $viewof(t_i) = viewof(t_k)$ and there exists $\mathbf{recv}(q, m')$. We should prove that there is also $\mathbf{recv}(q, m)$, and $\mathbf{recv}(q, m)$ precedes $\mathbf{recv}(q, m')$. By Claim A.2.3, there is a a sequence $S1$ of events $\mathbf{send}(p_1 = p, m'_1 = m) \to \mathbf{send}(p_1, m_1) \to \mathbf{recv}(p_2, m_1) \to \mathbf{send}(p_2, m_2) \to \cdots \to \mathbf{recv}(p_n = p', m_{n-1}) \to \mathbf{send}(p_n = p', m_n = m')$.[11]

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 $viewof(t_j) \neq viewof(t_l)$. If t_j and t_l are **send** and **recv** of the same message, then $viewof(t_j) = viewof(t_l)$, by Sending View Delivery. Therefore, t_j and t_l occurred at the same process, and $j < l$. Using Claim A.2.2, we conclude that $viewof(t_j).vid < viewof(t_l).vid$. Hence, $viewof(\mathbf{send}(p_1, m'_1)).vid \leq viewof(\mathbf{send}(p_1, m_1)).vid = viewof(\mathbf{recv}(p_2, m_1)).vid \leq viewof(\mathbf{send}(p_2, m_2)).vid = \cdots = viewof(t_j).vid < viewof(t_l).vid = \cdots = viewof(\mathbf{recv}(p_n, m_{n-1})).vid \leq viewof(\mathbf{send}(p_n, m_n)).vid$. Summarizing, $viewof(t_i).vid < viewof(t_k).vid$. This is a contradiction to the lemma condition that $viewof(t_i) = viewof(t_k)$.

Since there are $\mathbf{send}(p_1, m'_1)$, and later $\mathbf{send}(p_1, m_1)$ and $\mathbf{recv}(p_2, m_1)$ in the same view, there is also $\mathbf{recv}(p_2, m'_1)$ preceding $\mathbf{recv}(p_2, m_1)$, by Property 6.2 (Reliable FIFO). By Lemma A.1 we can apply Property 6.5 (Strong Total Order) and conclude that $ts(m'_1) < ts(m_1)$. Applying Claim A.2.1 to $\mathbf{recv}(p_i, m_{i-1})$ and $\mathbf{send}(p_i, m_i)$ 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, by Property 6.7 (Reliable Total Order), there is also $\mathbf{recv}(q, m)$ preceding $\mathbf{recv}(q, m')$. □

ACKNOWLEDGMENTS

We are grateful to Uri Abraham, Aviva Dayan, Roberto De Prisco, Danny Dolev, Alan Fekete, Shmuel Katz, Roger Khazan, Nancy Lynch, Alberto Montresor, Sharon Or, Michel Raynal, Ohad Rodeh, Andre Schiper, Roberto Segala, Jeremy Sussman, and the anonymous referees for many comments and helpful suggestions that 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 article.

REFERENCES

ABDELZAHER, T., SHAIKH, A., JAHANIAN, F., AND SHIN, K. 1996. RTCAST: Lightweight multicast for real-time process groups. In *IEEE Real-Time Technology and Applications Symposium* (*RTAS*) (June).

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

AL-SHAER, E., YOUSSEF, A., ABDEL-WAHAB, H., MALY, K., AND OVERSTREET, C. M. 1997. Reliability, scalability and robustness issues in IRI. In *IEEE Sixth Workshops on Enabling Technologies:*

---

[11] We do not give a separate proof for $S2$ since it is a special case of $S1$.

Infrastructure for Collaborative Enterprises (*WETICE'97*) (June).

Al-Shaer, E. S., Abdel-Wahab, H., and Maly, K. 1999. HiFi: A new monitoring architecture for distributed system management. In *19th International Conference on Distributed Computing Systems* (*ICDCS*) (June), pp. 171–178.

Amir, O., Amir, Y., and Dolev, D. 1993. 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). LNCS 774.

Amir, Y. and Stanton, J. 1998. The spread wide area group communication system. TR CNDS-98-4, The Center for Networking and Distributed Systems, The Johns Hopkins University.

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

Amir, Y., Chockler, G. V., Dolev, D., and Vitenberg, R. 1997. Efficient state transfer in partitionable environments. In *Second European Research Seminar on Advances in Distributed Systems* (*ERSADS'97*) (March), pp. 183–192. BROADCAST (ESPRIT WG 22455): Operating Systems Laboratory, Swiss Federal Institute of Technology, Lausanne. Full version: Tech. Rep. CS98-12, Institute of Computer Science, The Hebrew University, Jerusalem, Israel.

Amir, Y., Dolev, D., Kramer, S., and Malki, D. 1992a. Membership algorithms for multicast communication groups. In *Six International Workshop on Distributed Algorithms* (*WDAG*) (November), pp. 292–312. Springer Verlag.

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

Amir, Y., Dolev, D., Melliar-Smith, P. M., and Moser, L. E. 1994. Robust and efficient replication using group communication. Tech. Rep. CS94-20, Institute of Computer Science, Hebrew University, Jerusalem, Israel.

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

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

Anker, T., Chockler, G., Dolev, D., and Keidar, I. 1998. Scalable group membership services for novel applications. In M. Mavronicolas, M. Merritt, and N. Shavit, Eds., *Networks in Distributed Computing* (*DIMACS Workshop*), vol. 45 of *DIMACS*, pp. 23–42. American Mathematical Society.

Anker, T., Chockler, G., Shnaiderman, I., and Dolev, D. 2000. The design of Xpand: A group communication system for wide area networks. Tech. Rep. 2000-31 (July), Institute of Computer Science, Hebrew University, Jerusalem, Israel.

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

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

Babaoğlu, Ö., Bartoli, A., and Dini, G. 1996. On programming with view synchrony. In *16th International Conference on Distributed Computing Systems* (*ICDCS*) (May), pp. 3–10. Also Tech. Rep. UBLCS95-15, Department of Computer Science, University of Bologna, 1995.

Babaoğlu, Ö., Davoli, A., Montresor, A., and Segala, R. 1998a. System support for partition-aware network applications. In *18th International Conference on Distributed Computing Systems* (*ICDCS*) (May), pp. 184–191.

Babaoğlu, Ö., Davoli, R., and Montresor, A. 1998b. Group communication in partitionable systems: Specification and algorithms. TR UBLCS98-1 (April), Department of Computer Science, University of Bologna. In *IEEE Transactions on Software Engineering*, *27*, 4 (April 2001), 308–336.

Birman, K. 1996. *Building Secure and Reliable Network Applications*. Manning.

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

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

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

Birman, K. P. 1986. ISIS: A system for fault-tolerant distributed computing. Tech. Rep. TR86-744 (April), Cornell University, Department of Computer Science.

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

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

Cheriton, D. and Zwaenepoel, W. 1985. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems 3*, 2, 77–107.

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

Chockler, G., Huleihel, N., Keidar, I., and Dolev, D. 1996. Multimedia multicast transport service for groupware. In *TINA Conference on the Convergence of Telecommunications and Distributed Computing Technologies* (September).

Chodrow, S., Hircsh, M., Rhee, I., and Cheung, S. Y. 1997. Design and implementation of a multicast audio conferencing tool for a collaborative computing framework. In *JCIS* (March).

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

Cristian, F. and Fetzer, C. 1999. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 642–657.

Cristian, F. and Schmuck, F. 1995. Agreeing on process group membership in asynchronous distributed systems. Tech. Rep. CSE95-428, Department of Computer Science and Engineering, University of California, San Diego.

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

Défago, X., Schiper, A., and Urbán, P. 2000. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Tech. Rep. DSC/2000/036 (September), Swiss Federal Institute of Technology, Lausanne, Switzerland.

Dolev, D. and Malkhi, D. 1996. The Transis approach to high availability cluster communication. *Communications of the ACM 39*, 4 (April), 64–70.

Dolev, D., Friedman, R., Keidar, I., and Malki, D. 1997. Failure detectors in omission failure environments. In *16th ACM Symposium on Principles of Distributed Computing* (*PODC*) (August), pp. 286. Brief announcement. Full version: Tech. Rep. 96-1608, Department of Computer Science, Cornell University.

Dolev, D., Malki, D., and Strong, H. R. 1995. A framework for partitionable membership service. TR 95-4 (March), Institute of Computer Science, Hebrew University.

Dolev, S., Segala, R., and Shvartsman, A. 1999. Dynamic load balancing with group communication. In *Six International Colloquium on Structural Information and Communication Complexity* (*SIROCCO'99*), pp. 111–125.

Dwork, C., Lynch, N., and Stockmeyer, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35*, 2 (April), 288–323.

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

Fekete, A. and Keidar, I. 2001. A framework for highly available services based on group communication. In *IEEE 21st International Conference on Distributed Computing Systems Workshops* (*ICDCS-21W 2001*)*; the International Workshop on Applied Reliable Group Communication* (*WARGC*) (April), 57–62.

Fekete, A., Lynch, N., and Shvartsman, A. 1997. Specifying and using a partitionable group communication service. In *16th ACM Symposium on Principles of Distributed Computing* (*PODC*) (August), pp. 53–62. Full version to appear in *ACM Transactions on Computer Systems* 19, 2 (May 2001), 171–216.

Felber, P., Guerraoui, R., and Schiper, A. 1998. The implementation of a CORBA object group service. *Theory and Practice of Object Systems 4*, 2, 93–105.

Friedman, R. and van Renesse, R. 1995. Strong and weak virtual synchrony in Horus. TR 95-1537 (August), Dept. of Computer Science, Cornell University.

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

Fritzke, U. J., Ingels, P., Mostefaoui, A., and Raynal, M. 1998. Fault-tolerant total order multicast to asynchronous groups. In *17th IEEE International Symposium on Reliable Distributed Systems* (*SRDS*) (October), pp. 228–234.

Galleni, A. and Powell, D. 1996. Consensus and membership in synchronous and asynchronous distributed systems. Tech. Rep. 96104 (April), LAAS-CNRS. Revised January 1997.

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

Goft, G. and Yeger Lotem, E. 1999. The AS/400 cluster engine: A case study. In *IGCC 1999, in conjunction with ICPP 1999*.

Guerraoui, R. and Schiper, A. 1995. Transaction model vs. virtual synchrony model: Bridging the gap. In *Theory and Practice in Distributed Systems*, LNCS 938 (September), pp. 121–132. Springer-Verlag.

Guerraoui, R. and Schiper, A. 1997a. Consensus: The big misunderstanding. In *Proceedings of the Six IEEE Computer Society Workshop on*

Future Trends in Distributed Computing Systems (*FTDCS-6*) (Tunis, Tunisia, October), pp. 183–188. IEEE Computer Society Press.

Guerraoui, R. and Schiper, A. 1997b. Software-based replication for fault tolerance. *IEEE Computer 30*, 4 (April), 68–74.

Guerraoui, R. and Schiper, A. 2000. Genuine atomic multicast in asynchronous distributed systems. *Theoretical Computer Science 254*, 1–2 (March 2001), 297–316. Also Tech. Rep. 98/273, Swiss Federal Institute of Technology.

Guttag, J. V., Horning, J. J., Garland, S. J., Jones, K. D., Modet, A., and Wing, J. M. 1993. *Larch: Languages and Tools for Formal Specification*. Texts and Monographs in Computer Science. Springer-Verlag.

Hayden, M. and van Renesse, R. 1996. optimizing layered communication protocols. Tech. Rep. TR96-1613 (November), Dept. of Computer Science, Cornell University, Ithaca, NY.

Hickey, J., Lynch, N., and van Renesse, R. 1999. Specifications and proofs for ensemble layers. In *Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (*TACAS*), LNCS (March). Springer-Verlag.

Hiltunen, M. and Schlichting, R. 1995. Properties of membership services. In *Second International Symposium on Autonomous Decentralized Systems*, pp. 200–207.

Hiltunen, M. A. and Schlichting, R. D. 1998. A configurable membership service. *IEEE Transactions on Computers 47*, 5 (May), 573–586.

IBM 1996. *RS/6000 SP High Availability Infrastructure*. IBM. SG24-4838, available online at: http://www.redbooks.ibm.com/abstracts/sg2448-38.html.

IONA 1994. *IONA and Isis. An Introduction to Orbix+ISIS*. IONA Technologies and Isis Distributed Systems.

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

Johnson, S., Jahanian, F., Ghosh, S., Vanvoorst, B., and Weininger, N. 2000. Experiences with group communication middleware. In *The International Conference on Dependable Systems and Networks* (*DSN*). Practical Experience Report.

Johnson, S., Jahanian, F., and Shah, J. 1999. The inter-group router approach to scalable group composition. In *19th International Conference on Distributed Computing Systems* (*ICDCS*) (June), pp. 4–14.

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

Kalantar, M. and Birman, K. 1999. Causally ordered multicast: The conservative approach. In *19th International Conference on Distributed Computing Systems* (*ICDCS*) (June), pp. 36–44.

Keidar, I. 1994. A highly available paradigm for consistent object replication. Master's thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel. Also Institute of Computer Science, Hebrew University Tech. Rep. CS95-5, and available from: http://www.cs.huji.ac.il/~transis/publications.html.

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

Keidar, I. and Dolev, D. 2000. Totally ordered broadcast in the face of network partitions. Exploiting group communication for replication in partitionable networks. In D. Avresky, Ed., Chapter 3 of *Dependable Network Computing*, pp. 51–75. Kluwer Academic.

Keidar, I. and Khazan, R. 2000. A client-server approach to virtually synchronous group multicast: Specifications and algorithms. In *20th International Conference on Distributed Computing Systems* (*ICDCS*) (April), pp. 344–355. Full version: MIT Tech. Rep. MIT-LCS-TR-794.

Keidar, I., Sussman, J., Marzullo, K., and Dolev, D. 2000. A client-server oriented algorithm for virtually synchronous group membership in WANs. In *20th International Conference on Distributed Computing Systems* (*ICDCS*) (April), pp. 356–365. Full version: MIT Technical Memorandum MIT-LCS-TM-593a, June 1999, revised September 2000.

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

Khazan, R., Fekete, A., and Lynch, N. 1998. Multicast group communication as a base for a load-balancing replicated data service. In *12th International Symposium on DIStributed Computing* (*DISC*) (Andros, Greece, September), pp. 258–272.

Krantz, A., Chodrow, S., and Hircsh, M. 1998. Design and implementation of a distributed x multiplexor. In *18th International Conference on Distributed Computing Systems* (*ICDCS*) (May).

Krantz, A., Rhee, I., Breuker, C., Chodrow, S., and Sunderam, V. 1997. Supporting input multiplexing in a heterogenous environment. In *JCIS* (March).

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

Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (July), 558–565.

Landis, S. and Maffeis, S. 1997. Building reliable distributed systems with CORBA. *Theory and Practice of Object Systems 3*, 1.

Lin, K. and Hadzilacos, V. 1999. Asynchronous group membership with oracles. In *13th International Symposium on DIStributed Computing* (*DISC*) (Bratislava, Slovak Republic), pp. 79–93.

Lynch, N. 1996. *Distributed Algorithms*. Morgan Kaufmann.

Lynch, N. and Tuttle, M. 1989. An introduction to input/output automata. *CWI Quarterly 2*, 3, 219–246.

Malkhi, D. and Reiter, M. 1997. A high-throughput secure reliable multicast protocol. *Journal of Computer Security 5*, 113–127.

Malkhi, D., Merritt, M., and Rodeh, O. 1997. Secure multicast in a WAN. In *17th International Conference on Distributed Computing Systems* (*ICDCS*) (May), pp. 87–94.

Malloth, C. and Schiper, A. 1995. View synchronous communication in large scale networks. In *Second Open Workshop of the ESPRIT project BROADCAST (Number 6360)* (July).

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

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

Mishra, S., Fetzer, C., and Cristian, F. 1998. The Timewheel group membership protocol. In *Third IEEE Workshop on Fault-tolerant Parallel and Distributed Systems* (*FTPDS*) (April).

Mishra, S., Peterson, L. L., and Schlichting, R. D. 1991. A membership protocol based on partial order. In *International Working Conference on Dependable Computing for Critical Applications* (February).

Mishra, S., Peterson, L. L., and Schlichting, R. L. 1993. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal 1*, 2 (Dec.), 87–103.

Montresor, A., Davoli, R., and Babaoglu, O. 2000. Middleware for dependable network services in partitionable distributed systems. In *PODC Middleware Symposium* (July). Also Tech. Rep. UBLCS 99-19, October 1999 (Revised April 2000).

Moser, L. E., Amir, Y., Melliar-Smith, P. M., and Agarwal, D. A. 1994. Extended virtual synchrony. In *14th International Conference on Distributed Computing Systems* (*ICDCS*) (June), pp. 56–65.

Moser, L. E., Melliar-Smith, P. M., and Narasimhan, P. 1998. Consistent object replication in the Eternal system. *Theory and Practice of Object Systems 4*, 2, 81–92.

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

OMG. 2000. *Fault Tolerant CORBA Specification*. OMG (Object Management Group) Document ptc/2000-04-04.

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

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

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

Rodrigues, L. and Verissimo, P. 1992. *x*AMp, a protocol suite for group communication. RT /43-92 (January), INESC.

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

Schiper, A. and Ricciardi, A. 1993. Virtually synchronous communication based on a weak failure suspector. In *23rd IEEE Fault-Tolerant Computing Symposium* (*FTCS*) (June), pp. 534–543.

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

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

Shamir, G. 1996. Shared whiteboard: A Java application in the Transis environment. Lab project, High Availability lab, The Hebrew University of Jerusalem, Jerusalem, Israel. Available from: http://www.cs.huji.ac.il/~transis/publications.html.

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

SUSSMAN, J., KEIDAR, I., AND MARZULLO, K. 2000. Optimistic virtual synchrony. In *19th IEEE International Symposium on Reliable Distributed Systems* (*SRDS*) (October), pp. 42–51.

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

VAN RENESSE, R., BIRMAN, K. P., AND MAFFEIS, S. 1996. Horus: A flexible group communication system. *Communications of the ACM 39*, 4 (April), 76–83.

VITENBERG, R. 1998. Properties of distributed group communication and their utilization. Master's thesis, Institute of Computer Science, Hebrew University, Jerusalem, Israel.

VOGELS, W. 1996. World wide failures. In *ACM SIGOPS 1996 European Workshop* (September).

VOGELS, W. AND VAN RENESSE, R. 1994. Support for complex multi-media applications using the Horus system. Ithaca, NY, Dept. of Computer Science, Cornell University. On-line html document: http://www.cs.cornell.edu/Info/People/rvr/papers/rt/novsdav.html.

WHETTEN, B., MONTGOMERY, T., AND KAPLAN, S. 1995. A high performance totally ordered multicast protocol. In K. P. Birman, F. Mattern, and A. Schiper, Eds., *Theory and Practice in Distributed Systems: International Workshop*, pp. 33–57. Springer Verlag. LNCS 938.

WILHELM, U. G. AND SCHIPER, A. 1995. A hierarchy of totally ordered multicasts. In *14th IEEE International Symposium on Reliable Distributed Systems* (*SRDS*) (September).

YEGER LOTEM, E., KEIDAR, I., AND DOLEV, D. 1997. Dynamic voting for consistent primary components. In *16th ACM Symposium on Principles of Distributed Computing* (*PODC*) (August), pp. 63–71.

Received December 1999; revised March 2001; accepted March 2001