Understanding Membership

Matti A. Hiltunen   Richard D. Schlichting

TR 95-07

Understanding Membership[1]

Matti A. Hiltunen Richard D. Schlichting

TR 95-07

Abstract

A membership service is used in a distributed system to maintain information about which sites are functioning and which have failed at any given time. Such services have proven to be fundamental for constructing distributed applications, with many example services and algorithms defined in the literature. Despite these efforts, however, little has been done on examining the abstract properties commonly guaranteed by membership services independent of a given implementation. Here, a number of these properties are identified and defined. These properties range from agreement among sites on membership changes, consistent ordering of change notifications, and timing properties to various ways for dealing with recoveries and partitions. Message ordering graphs, which are an abstract representation of the set of messages at each site in the system and their potential delivery order, are used to define the properties. Dependency graphs, which are a graphical representation expressing when a property is defined assuming the existance of another property or when a stronger property includes a weaker property, are used to illustrate the relationships between properties. These graphs help differentiate existing services, as well as facilitate the design of new configurable services in which only those properties actually required by an application are included. Finally, a number of existing membership services are characterized in terms of the properties identified.

July 19, 1995

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grants N00014-91-J-1015 and N00014-94-1-0015.

1 Introduction

A *membership service* is a service used in a distributed system to maintain information about which sites are functioning and which have failed at any given time. Membership services, and membership protocols that implement these services, have been studied extensively, both for synchronous systems where bounds are placed on network transmission time [Cri91, KGR91, KG94, EL90, LE90, vSCA94], and for asynchronous system where no such assumption is made [DMS94, EL95, AMMS$^+$93, MPS93, RB91, SR93, ADKM92a, GT92, RFJ93, Bir85, SM94]. The properties enforced by the protocols also vary, ranging from ones that offer very weak properties [RFJ93, GT92, Hil95] to others such as ISIS that guarantee strong properties [BSS91]. The tradeoff is the strength of the guarantee versus the execution cost. For example, the property called *virtual synchrony* [BJ87] guarantees that messages reflecting membership change events are delivered to the application by the membership layer at every site at precisely the same point in the message stream. Weaker properties provide less rigorous guarantees, but are correspondingly cheaper to implement. As might be expected, the semantics of the application has a strong influence on the type of membership protocol needed: some require strong properties, while others will execute correctly with something weaker.

Despite the above efforts, little has been done on examining the abstract properties important to membership independent of a given service. Here, we attempt to rectify this shortcoming by identifying these properties and characterizing them using *message ordering graphs* and *dependency graphs*. The former are used to specify the effect each property has on the flow of messages being exchanged, while the latter are used to characterize the relationships between properties. The way in which each property can be used by an applications is also discussed briefly.

The advantages of developing an understanding of membership's constituent properties are numerous. For example, it helps clarify the structure and semantics of such services, which by their very nature are one of the most complicated, but also one of the most important, services in a distributed system. It also helps differentiate existing services, thereby assisting the distributed system developer in the choice of which is most appropriate for a given situation. Perhaps most importantly, it also facilitates the design of new services in which only those properties actually required by an application are included. Such an approach can be realized by implementing each property as a separate software module and then combining the appropriate modules within a standard software infrastructure to produce a customized system [BS95, HS93].

This paper is organized as follows. Section 2 describes the details of message ordering graphs and other aspects of our computational model. Properties of membership services are then presented in section 3; these range from message ordering properties to properties related to handling partitions. A discussion of the way in which properties are related is found in section 4, along with a dependency graph that summarizes these relationships. Section 5 uses the framework constructed in previous sections to characterize a number of existing membership services, including those is Consul [MPS93], ISIS [BJ87], Mars [KGR91], and Totem [AMMS$^+$93]. Finally, section 6 offers some concluding remarks.

2 Message Ordering Graphs

The following notation is used in this and subsequent sections. Capital letters A, B, C, ... are used to denote sites.[1] Small letters with a subscript, for example a_i, are used to indicate the i^{th} message sent by site A. $\mathcal{S}$ denotes the set of all sites and $\mathcal{M}$ the set of all messages. In our model, each site's lifetime

[1] Membership can be characterized in terms of collections of processes or, if there is one member of the process group per site, in terms of sites. For simplicity, we choose the latter.

consists of initial startup, followed by any number of alternating failures and recoveries. Each period during which a site is operational is identified by an incarnation number, which is assumed to be unique over the lifetime of a site. When required, the incarnation r of site A is denoted by A_r.

In our model, the application interacts with the underlying layers—in particular, the communication and membership layers—only through messages that are passed in either direction across the interface. As a result, the only way to define properties for the underlying services is in terms of what messages are delivered to the application, in which order they are delivered, when they are delivered, how the messages and their order relate to events such as failure and recovery in the system, and how the set of messages, their order, and delivery time relate to one another on different sites. For example, a requirement that messages be ordered FIFO between two sites means that the messages sent by a site A are constrained to be delivered to the application at any other site B in the same order as they were sent. This property sets a constraint on the order in which messages are delivered to the application at the receiving site based on the order in which they were sent by the originating site. Similarly, a requirement that messages be delivered in total order at all sites in a group constrains the delivery order on A to be identical to the delivery order on any other site B. Finally, a requirement that message transmission time be bounded by some Δ constrains the delivery of a message sent at some global time t_1 to occur at some global time t_2 such that $t_2 - t_1 < \Delta$.

Let $\mathcal{P}$ be a set of properties implemented by a given service. As noted above, $\mathcal{P}$ can be stated in terms of constraints that must be true for the sequence of messages delivered to the application at the various sites involved in the computation. In this paper, we use *message ordering graphs* and temporal logic formulas to define and illustrate the various properties. A message ordering graph is an abstract representation of the set of messages to be delivered on a site and the ordering constraints between these messages. Formally, an ordering graph is a directed acyclic graph, $G = (N, E)$, where the set of vertices, N, is a set of messages and the set of edges, $E = \{(m_i, n_j) | m_i, n_j \in N\}$, is the set of ordering constraints between messages. If O is an ordering graph, $N(O)$ denotes the set of vertices and $E(O)$ denotes the set of edges. If $(m_i, n_j) \in E$, then m_i is called a *predecessor* of n_j, $m_i \in$ pred(n_j) for short, and n_j is called a *successor* of m_i, $n_j \in$ succ(m_i) for short. The meaning of the edges is that, if n_j is a successor of m_i, in order for $\mathcal{P}$ to be satisfied, n_j can only be delivered to the application after message m_i.

O_A denotes the ordering graph at site A and represents all the messages that are to be delivered to the application at A together with the corresponding ordering constraints. The set of ordering graphs from all sites for a given execution is denoted by Θ. Where clear from context, we also use O_A to denote the sequence of successive ordering graphs at A during a particular execution, and Θ to denote the global sequence of ordering graphs in the system. The state of O_A at real time t is denoted by $O_A(t)$. Similarly, Θ at time t is denoted by $\Theta(t)$.

The basic ordering graph only reflects the set of messages and their possible delivery orders. In some cases, the actual order in which messages are delivered to the application or the actual time when the delivery occurs is of interest. For this purpose, define event $del_A(m)$ to denote the event of message m being delivered at site A. Symmetrically, define $send_A(a_i)$, or $send(a_i)$ for short, to be the event of application on A sending message a_i. Furthermore, define $time(event_i)$ to be a function that returns the time t at which $event_i$ occurs. Note that t is not a timestamp generated by the system—i.e., the result of reading some real time clock—but rather a time as seen by an external observer that is used only for specification purposes. If an event has not occurred, $time$ is undefined.

Finally, define the *view* of the ordering graph at site A, $view_A$, to be the set of messages that have been delivered to the application on A. Thus, the relationship between message delivery and a view is:

$$\forall\ m \in \mathcal{M} : del_A(m) \Rightarrow m \in view_A$$

Note that in the above, $del_A(m)$ is used as a predicate. Such a predicate evaluates to false until the event occurs and to true afterwards. Similarly to ordering graphs, the view at some real time t, denoted $view_A(t)$, is defined to be the set of messages delivered on that site by time t.

Given these definitions, properties can be defined in terms of how they affect ordering graphs. Formally, a *property* is defined by a set of constraints on nodes and edges in a collection of ordering graphs, and how those nodes and edges relate to other system events. For example, a FIFO ordering property for a reliable multicast where every site is assumed to receive every message can be expressed as:

$$\forall\ A, B;\ i \in [1, \infty[\ : (b_i, b_{i+1} \in N(O_A)) \Rightarrow ((b_i, b_{i+1}) \in E(O_A))$$

This property specifies an ordering constraint for the graphs in Θ that must hold across all executions. While the FIFO property can be stated in terms of a single ordering graph, most other properties relate ordering graphs from multiple sites. Let $a_i \rightarrow b_j \in O_A$ indicate that there is a path of length ≥ 1 from message a_i to message b_j in ordering graph O_A. Then, for example, a consistent total order in a reliable multicast system can be stated as:

$$\forall\ A, m_i, n_j : (m_i, n_j \in N(O_A)) \Rightarrow (((m_i \rightarrow n_j) \in O_A)\ \vee\ ((n_j \rightarrow m_i) \in O_A))$$

and

$$\forall\ A, B, m_i, n_j : ((m_i \rightarrow n_j) \in O_A) \Rightarrow \Box((n_j \rightarrow m_i) \notin O_B)$$

where $\Box$ is the temporal operator denoting "henceforth". Again, these formula must hold for Θ across all executions.

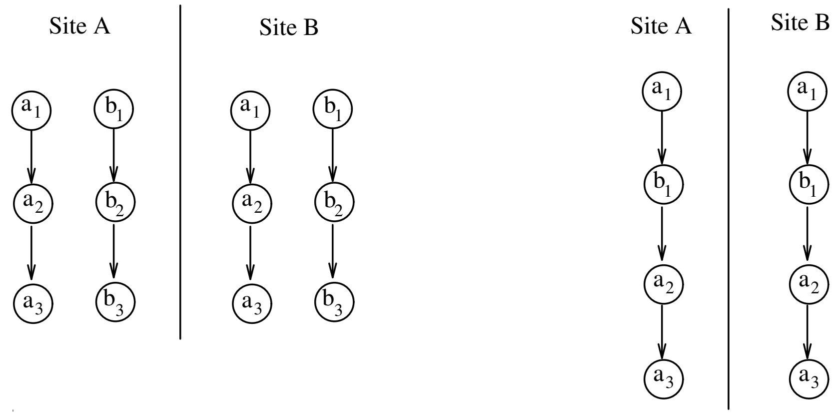

Figure 1: FIFO and Consistent Total Order Multicasts

Figure 1 illustrates these simple examples. In the figure we assume an underlying reliable multicast mechanism, so the ordering graphs are identical at each site. Note, however, that although the ordering graphs are identical for FIFO, the order in which messages are actually delivered to the application at sites A and B may differ while still satisfying the ordering constraints. For example, site A may deliver the messages in order a_1, a_2, a_3, b_1, b_2, b_3, while site B delivers them in order a_1, b_1, a_2, b_2, a_3, b_3. The only requirement is that a message be delivered after its predecessor(s), so messages between which there is no ordering constraint can be delivered in any order.

In this paper, message ordering graphs are used as a tool for specifying and illustrating properties of membership services. We assume that the system includes a communication service that implements whatever communication semantics are required by the application. The ordering constraints imposed by the communication service can be expressed as properties of ordering graphs as demonstrated above, where nodes are application messages. To specify the properties of membership services, then, messages that indicate a change in membership are inserted into such a graph with edges that represent the various ordering properties to be enforced. Figure 2 illustrates the notation used for ordering graphs in the following sections.

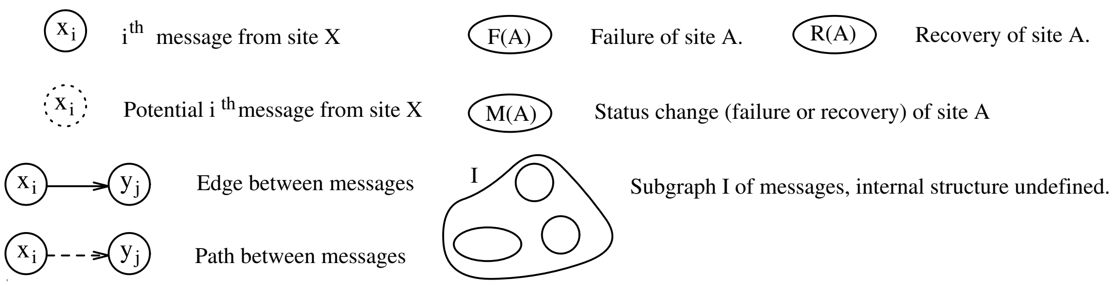

Figure 2: Ordering Graph Notation

Note that ordering graphs are only one method for defining, describing, or illustrating properties of distributed systems. Another approach, used for example in [RB91], is based on describing system behavior by *process histories*, where the history for process p is a sequence of events including send events, receive events, and internal events. A *system run* is a set of process histories, one for each process in the set of processes. Properties are defined in terms of temporal logic formulas over these histories. It would be possible to define the properties of membership services in this manner but in this paper we chose to use ordering graphs due to their more illustrative nature. Also, in contrast to process histories, ordering graphs model more closely the execution of the system and allow the expression of all legal orderings of message receptions in one graph instead of stating them as properties of linear histories. Numerous other methods are also possible. For example, in [RFJ93] membership properties are stated in terms of *membership runs*, which are defined as sequences of global membership states consisting of each site's view of the global membership. Transitions from one global membership state to the next occur whenever a site changes its view of the global membership. This approach is less comprehensive than ordering graphs, since it does not address the ordering of membership changes with respect to sending and receiving of application messages.

Finally, note that the concept of a graph of messages is very appealing as an implementation tool as well. For example, this technique is closely related to the *causality graphs* used in Psync [PBS89] and Transis [ADKM92b], which capture the causal ordering relationship between messages.

3 Properties of Membership Services

3.1 Overview

A membership service can be viewed as a protocol layer that generates messages indicating changes in membership and forwards them to higher levels. These *membership change messages* can report, for example, site failure, site recovery, or the joining of two partitions. The application can use the information in these messages in many ways. For example, it can be used to direct multicast messages to the current membership as seen by the application, to choose a leader of the group, or to make various decisions about the global state of the computation. Given this view, the properties of a membership service can be defined in terms of what membership change messages it generates and when they are delivered to the application [HS95].

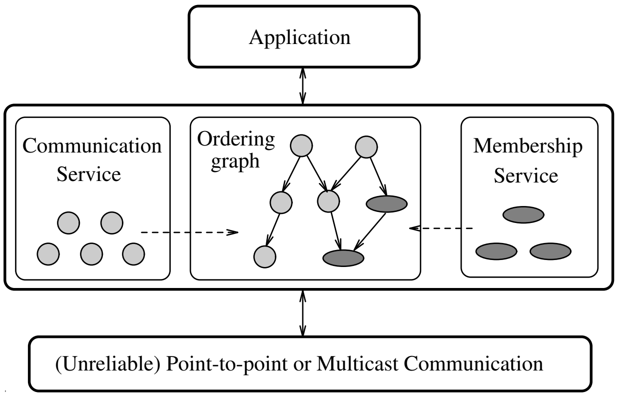

Figure 3: System Structure

Figure 3 illustrates the logical system structure. The *communication* and *membership* services add application and membership change messages, respectively, to the ordering graph. Although we separate them here logically, in practice the communication and membership component are often tightly bound to one another, and dependencies exist between them. The communication component is responsible for realizing the required properties of the application level communication between group members. Some properties of membership services can be implemented without considering the form of the communication service, but most, especially properties involving ordering membership change messages with respect to application messages, require that application communication be based on reliable ordered multicasts that guarantee that every message is delivered by all functioning sites in some consistent order. Since ordering properties like FIFO or causal ordering are defined relative to the sending site, these application messages typically encode in some way the information needed to realize these constraints. The responsibility of the membership service is to guarantee that membership change messages appear in the ordering graph when and where they are supposed to according to the properties specified.

Given this structure, then, we now define a number of properties of membership services based on how these properties are reflected in the ordering graphs.

3.2 Accuracy and Liveness

A membership service is *accurate* if the failure of a site is only reported to the application if the site has actually failed, and the recovery (or startup) of a site is only reported if the site has actually recovered (or started up). A membership service is *live* if any failure or recovery is eventually detected and reported. A special case of liveness is *bounded liveness*, where the failure or recovery is detected within a known bounded time. In an asynchronous system, a service can be either accurate or live, but not both [BG93], whereas in synchronous systems both properties can be guaranteed at the same time. An example of an accurate service that is not live is that of Mach, where the failed site notifies others about its own failure upon recovery [OIOP93]. However, most membership services for asynchronous systems have chosen to be live but not accurate, for example, ISIS [BSS91] and Consul [MPS93]. The lack of accuracy in such systems comes from the use of timeouts to suspect the failure of a site, a technique that may trigger false suspicions. To deal with potentially inaccurate decisions, suspected sites that have in fact not failed are often isolated from the group and forced to fail and then recover before continuing execution. Examples of synchronous membership protocols that are both accurate and live are the protocols proposed in [Cri91]. Note, however, that a synchronous system is an abstraction that is maintained only as long as the the bounded delivery time assumption is not violated. As a result, if this assumption is violated, a detection algorithm that is intended to be live and accurate will lose its accuracy characteristics. This scenario is acknowledged and handled, for example, in the design of Mars [KGR91].

Accuracy and liveness can be defined more formally in terms of ordering graphs. Let A and B be arbitrary sites and $F(B_i)$ and $R(B_j)$ be membership change messages indicating the failure (of incarnation i) and recovery (of incarnation j) of site B, respectively. Let $Failure(B_i)$ and $Recovery(B_j)$ signify events corresponding to site B failing or recovering, respectively. Then, a membership service is live if it guarantees that

$$Failure(B_i) \Rightarrow \Diamond(F(B_i) \in N(O_A)) \quad \wedge \quad Recovery(B_j) \Rightarrow \Diamond(R(B_j) \in N(O_A))$$

where $\Diamond$ is the temporal operator denoting "eventually". Note that in order for the definition to be satisfied, site A has to either not fail before receiving the membership change message or recover and upon recovery be notified of the change. The case of bounded liveness can be defined similarly, except that the membership change message for a change occurring at time t_1 must be delivered by some time t_2 such that $t_2 - t_1 < \Delta$, where Δ is a known constant. Likewise, a membership service is accurate if it guarantees that

$$\Diamond Failure(B_i) \vee \Box(F(B_i) \notin N(O_A)) \quad \wedge \quad \Diamond Recovery(B_j) \vee \Box(R(B_j) \notin N(O_A))$$

In membership services in which change detection is inaccurate, the level of *confidence* indicates how certain it is that the suspected change has actually occurred. The typical way to increase confidence is to compare information from different sites before making a final decision. Thus, different levels of confidence can be defined by specifying how many sites must agree that a suspected change has occurred before a change indication is forwarded to the application. The possibilities range from a single site [RB91, RFJ93, SM94] to all functioning sites [MPS92]. In the following, *single site suspicion* is used to specify the former and *consensus* the latter. Any option between these two extremes is referred to as a *voted decision*. In general, the use of voted decisions has not been explored widely in the context of membership services. One exception is [Rei94], where voting is used to handle Byzantine failures.

Note that detecting failure can be dealt with separately from detecting recovery. A typical solution in asynchronous systems is to have failure detection be live but not accurate, with recovery detection being accurate but not live. This approach is natural since the most practical approach for detecting recovery is the receipt of a message sent by the recovered site. Moreover, implementing live recovery detection is impossible in an asynchronous system, since any number of messages sent by the recovered site may be lost or delayed.

3.3 Agreement

The *agreement* property requires that any membership change message delivered to the application at one site eventually be delivered at all other sites. Figure 4 illustrates this concept. The property can be stated in terms of the ordering graph as follows: if a membership change message $M(C)$ appears in the ordering graph of one site, it will eventually appear in the ordering graph of all other sites. More formally, let A and B be arbitrary sites and $M(C)$ an arbitrary membership change message. For agreement to be satisfied, the following must hold for O_A and O_B:

$$M(C) \in N(O_A) \Rightarrow \Diamond(M(C) \in N(O_B))$$

Note that there are no ordering requirements between membership change messages, or between membership change messages and application messages.

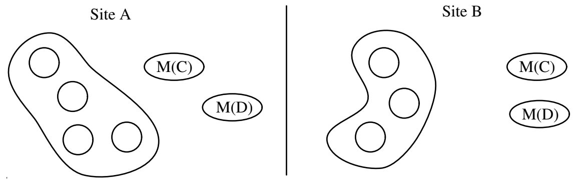

Figure 4: Agreement

Agreement specifies that the same membership change messages are delivered on all sites. A somewhat weaker variant, *eventual agreement on views*, only ensures that all sites eventually reach the same membership set (or view), assuming no additional failures occur for a long enough period of time. Unlike regular agreement, with this property, the actual changes made by different sites may vary—for example, a site A may be considered to have failed and recovered on one site and not failed at all on a second—as long as the end result is the same. This property, although implemented by certain weak protocols such as [RFJ93], is insufficient for implementing message ordering properties. Hence, for the remaining properties, we assume that regular agreement is guaranteed.

3.4 Ordering Properties

Ordering properties specify constraints on the order in which membership change and application messages are inserted into the ordering graph and hence, the order in which they are delivered to the application.

3.4.1 FIFO Ordering of Membership Messages

The *FIFO ordering* property requires that membership change messages concerning any given single site be delivered to the application at every site in the same order (Figure 5). In the ordering graph, this property requires that the membership changes of each individual site form a chain that is identical at every site. More formally, let A, B, and C be arbitrary sites and $M(C_i)$ and $M(C_j)$ be arbitrary membership change messages indicating a status change of site C. Then, for FIFO order to be satisfied, the following two properties must hold:

$$M(C_i), M(C_j) \in N(O_A) \Rightarrow ((M(C_i) \rightarrow M(C_j)) \in O_A) \ \vee \ ((M(C_j) \rightarrow M(C_i)) \in O_A)$$

and

$$(M(C_i) \rightarrow M(C_j)) \in O_A \Rightarrow \Box((M(C_j) \rightarrow M(C_i)) \notin O_B)$$

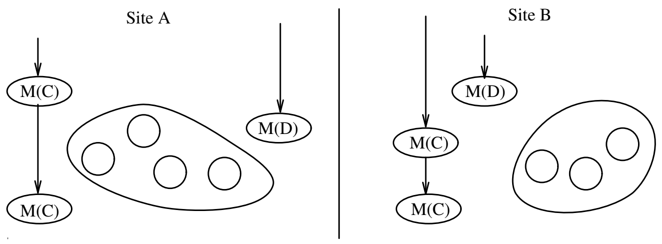

Figure 5: FIFO Order

3.4.2 Total Ordering of Membership Messages

Total ordering requires that membership change messages be delivered to the application at every site in the same total order (Figure 6). In the ordering graph, total order requires that membership change messages form a single chain that is identical at all sites. More formally, let A, B, C, and D be arbitrary sites, and $M(C)$ and $M(D)$ be arbitrary membership change messages. Then, for total order to be satisfied, the following two properties must hold:

$$M(C), M(D) \in N(O_A) \Rightarrow ((M(C) \rightarrow M(D)) \in O_A) \ \vee \ ((M(D) \rightarrow M(C)) \in O_A)$$

and

$$(M(C) \rightarrow M(D)) \in O_A \Rightarrow \Box((M(D) \rightarrow M(C)) \notin O_B)$$

Note that total order here only applies to membership change messages and hence, makes no statement about the relative ordering of membership and applications messages at the different sites.

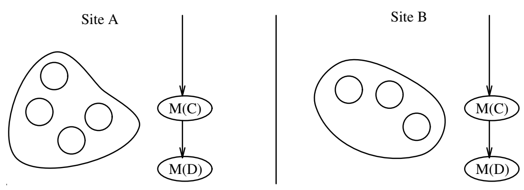

Figure 6: Total Order

Total order is a useful property for applications that rank processes or sites based on age, i.e., how long they have been members of the group, and then use that rank to reassign roles or tasks upon membership change. For example, an algorithm requiring a central coordinator could assign that role to the process with highest rank, with the second ranked taking over in case of failure. In order for the ranking seen on different sites to be identical, membership change messages must be processed in the same total order at each site.

The remaining ordering properties all order delivery of membership change messages with respect to application messages as well as other membership messages. As noted above, to establish such an ordering, application messages must be transmitted using an ordered reliable multicast service. Such a service guarantees that messages are delivered at all sites that remain functioning for the duration of the multicast and that are also within the same partition as the sending site. Sites that are in the process of joining the group may or may not receive the message. Furthermore, to simplify the presentation, we also assume that application messages are at least FIFO ordered.

3.4.3 Agreement on Last Message

The *agreement on last message* property requires that an agreed upon "final" message sent by a failed site be delivered to the application on each site prior to the membership change message announcing its failure (Figure 7). In the ordering graph, this property means that the membership change message indicating the failure of a site, say C, is a successor of the message c_i that all remaining sites agree is the last one to be delivered from the failed site. The notion that this is the last *agreed upon* message is important. There may, in fact, be a subsequent message, c_{i+1}, that was in transit when the agreement process was underway. Such a message will appear in the graph as a successor to the membership change message and be delivered to the application in the normal fashion. However, many applications will simply disregard this message as coming from a site that is no longer a valid member of the group.

More formally, for agreement on last message to be satisfied, there exists agreed last message c_i such that the following holds:

$$\exists\, c_i\ \forall\, A : (F(C) \in N(O_A)) \Rightarrow (c_i \rightarrow F(C)) \in O_A\ \wedge\ \Box(\forall j > i : (F(C) \rightarrow c_j) \in O_A)$$

If C did not send any messages before it failed, the agreed last message will be the membership change message $R(C)$.

This property is useful in applications where a consistent distributed state needs to be maintained. In such situations, the final message may cause a state change, so agreement ensures that the change is applied either at all sites or at no site.

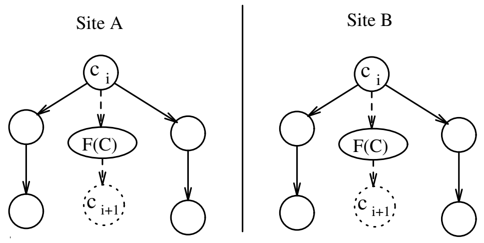

Figure 7: Agreement on Last Message

An analogous agreement on first message property can also be defined for recovering or joining sites.

3.4.4 Agreement on Successors

The *agreement on successors* property requires that all sites deliver a membership change message before any message in an agreed upon successor set is delivered (Figure 8). For example, if site C is recovering, then the successor set might be the cut in the message graph after which all messages are considered to have arrived after the recovery of C.

Formally, first define *cut*, $CUT(O_A)$, to be a set of nodes $S \in N(O_A)$ such that, for any node $m \notin S$, either $\exists s \in S : m \rightarrow s$ or $\exists s \in S : s \rightarrow m$, but not both. Let $IsCut(S, O_A)$ be a predicate that evaluates to true if the set of nodes S is a cut in ordering graph O_A, and false otherwise. Finally, let $succ_A(m)$ be the set of immediate successors of message m in the ordering graph of site A. Then, for agreement on successors to be satisfied, the following must hold for O_A and O_B:

$$IsCut(succ_A(M(C)), O_A) \ \wedge \ IsCut(succ_B(M(C)), O_B) \ \wedge \ succ_A(M(C)) = succ_B(M(C))$$

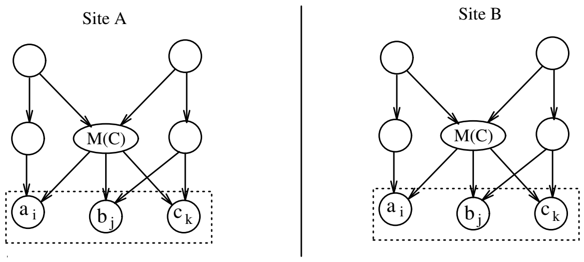

Figure 8: Agreement on Successors

Among other things, this property is useful for determining *message stability*, where a message is stable at the sending site once it has been acknowledged by every other operational site [PBS89]. If m is an agreed successor of $R(C)$, every site knows that m will have to be acknowledged by site C to be considered stable.

An analogous agreement on predecessors property can also be defined.

3.4.5 Virtual Synchrony

Virtual synchrony restricts the delivery order of application and membership change messages in such a way that it appears to the application as if events are occurring synchronously even though they are actually occurring on different sites at different times [BSS91]. Virtual synchrony is easy to explain in the ordering graph, as illustrated in Figure 9. Relative to membership, this property requires agreement among all operational sites on a division of the message stream such that each message is either in an agreed predecessor set to the membership change message or in an agreed successor set. In other words, virtual synchrony essentially creates an agreed cut in the message flow.

Let $\cap$ denote the intersection of two ordering graphs, i.e., the sets of vertices and edges that are common to both ordering graphs, and let $O_{A \cap B}$ denote an ordering graph that is the result of such an intersection. Then, for virtual synchrony to be satisfied, the following must hold for O_A and O_B:

$$\forall m_i : m_i, M(C) \in N(O_{A \cap B}) \Rightarrow ((M(C) \rightarrow m_i) \in O_{A \cap B}) \vee ((m_i \rightarrow M(C)) \in O_{A \cap B})$$

As extensively discussed in the ISIS literature, virtual synchrony makes it easy to write distributed applications. This is especially true for applications that can be viewed as replicated state machines that change their state when they receive application messages and membership changes [Sch90].

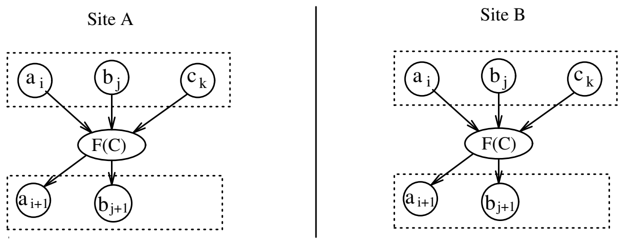

Figure 9: Virtual Synchrony

Note that, although virtual synchrony is closely related to agreement on successors and predecessors, it is not identical to combining these two properties. This follows because the combination does not require that every message be in one set or the other, whereas virtual synchrony does. Note also that, in contrast to the common definition of virtual synchrony, we choose here to define it without requiring that it also guarantee a total order of membership change messages.

3.4.6 Extended Virtual Synchrony

One drawback of virtual synchrony is that it does not necessarily relate the view of membership at the time a message is sent to the collection of sites that actually receive the message. For example, site A may multicast a message a_i when the membership is $\{A, B, C\}$, but before the message is received, another site D joins the group. Under virtual synchrony, the membership change message could be delivered to the application before a_i, thereby resulting in the delivery of the message to the application at D as well as A, B, and C. This is acceptable in cases where the actual membership of the destination group is not important, but stronger guarantees are useful in some cases. *Extended virtual synchrony* extends virtual synchrony by guaranteeing that all messages sent under the old membership are also delivered before the membership change message [AMMS+93].

Extended virtual synchrony can be defined more formally as follows. Let $event_1 < event_2$ denote $event_1$ happening before $event_2$ at the application on the same site. (Note that $<$ is a total ordering, assuming that the application is single threaded.) Then, for extended virtual synchrony to be satisfied, the following must hold in O_A and O_B:

$$\forall a_i : (send(a_i) < del_A(M(C))) \Rightarrow \Diamond((a_i \rightarrow M(C)) \in O_B)$$

and

$$\forall a_i : (del_A(M(C)) < send(a_i)) \Rightarrow \Diamond((M(C) \rightarrow a_i) \in O_B)$$

Note that the second rule is implicitly implemented by the communication subsystem provided that the communication is at least causally ordered. Figure 10 illustrates this property; the shaded circles represent messages that are sent before receiving the membership change message $M(C)$.

Extended virtual synchrony has been explored in a number of papers, especially [AMMS+93] and [MAMSA94]. Our definition only addresses the ordering aspects of the property as they relate to membership change messages, and as such, does not include the full functionality of extended virtual synchrony as defined in those papers.

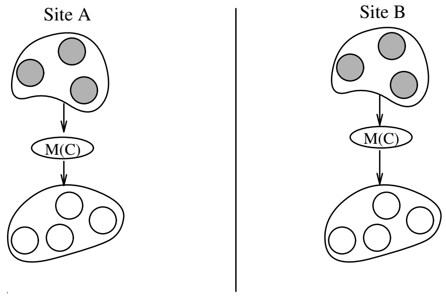

Figure 10: Extended Virtual Synchrony

3.5 Bounded Change Properties

Ordering properties constrain the order in which state changes related to membership occur at different sites, but leave unspecified any notion of when a site makes a change relative to the other sites. *Bounded change properties* extends ordering by adding bounds on when such changes must be applied.

3.5.1 External Synchrony

External synchrony guarantees that if a site delivers a given membership change message, all other sites have either already delivered the message or are in a transition state in which delivery is imminent [RFJ93]. Having this property ensures that sites move into a new membership state with some degree of coordination, and that all sites have consistent membership information, modulo sites undergoing a transition. The idea is, in fact, related to the concept of barrier synchronization in parallel programs, in which execution at all sites must reach the barrier—i.e., move into the transition state—before any site can proceed—i.e., make the membership change. Following [RFJ93], we denote this transition state as state 0. Since underlying layers interact with the application only through messages, any protocol implementing external synchrony requires an extra message to generate a transition into state 0. We call this the *prepare message* for membership change $M(C)$ and denote it as $Pre(M(C))$.

External synchrony can now be defined more formally as follows:

$$\forall A, B : del_A(M(C)) \Rightarrow \Box(Pre(M(C)) \in view_B)$$

This definition specifies that once a membership change message $M(C)$ is delivered on an arbitrary site A, the corresponding prepare message $Pre(M(C))$ has already been delivered on all sites, i.e., is in the view on all sites. Therefore, all sites have either delivered $M(C)$ or are in the transition state prior to delivering $M(C)$.

Figure 11 illustrates this property. Based on the definition, there is some global time t such that all sites deliver the prepare message prior to t and the membership change message after t. Note, however, that although this definition references t, implementation of this property does not require access to a global time source. For example, the barrier could be implemented by having each site multicast a message to the group after receiving the prepare message, and waiting until there is a message from all other sites before delivering the actual membership change message.

External synchrony is useful in a number of situations, especially in cases where an application must access an external device or send a message to a process outside the group. As an example, consider an application where a group leader is expected to take some action at a given time, where the action must

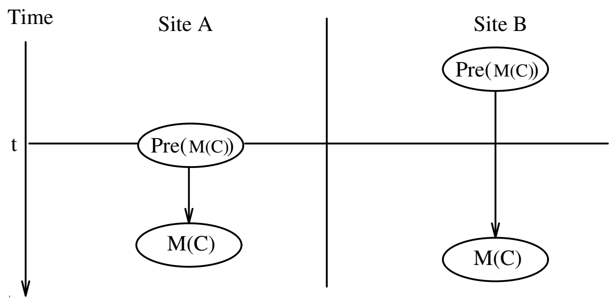

Figure 11: External Synchrony

be executed exactly once. Suppose further that a membership change that results in a leadership change happens to occur close to this time. Without external synchrony, there may be moments in real time when two different group members are designated as the leader on different sites, thereby potentially resulting in the external operation being executed more than once. External synchrony prevents this by guaranteeing that all sites share the same view of the membership or are knowingly in a transition state.

3.5.2 Timebound Synchrony

Timebound synchrony is a property of membership services in synchronous systems in which every site delivers a given membership change message within some known interval of real time [KGR91, Cri91]. The property has the same general applicability as external synchrony, but reduces the synchronization overhead by shrinking the window during which the membership is not identical on all sites. Also, it is important to have this property in most real-time systems, so that the system can respond in a predictable and timely manner to external events.

Timebound synchrony can be defined more formally as follows:

$$\forall A, B : (time(del_A(M(C))) = t_i \ \wedge \ time(del_B(M(C))) = t_j) \Rightarrow |t_i - t_j| < \Delta$$

where Δ is a known fixed constant. Figure 12 illustrates this concept. In the figure, dashed lines are used to represent the point in real time where the membership change message is delivered to the application.

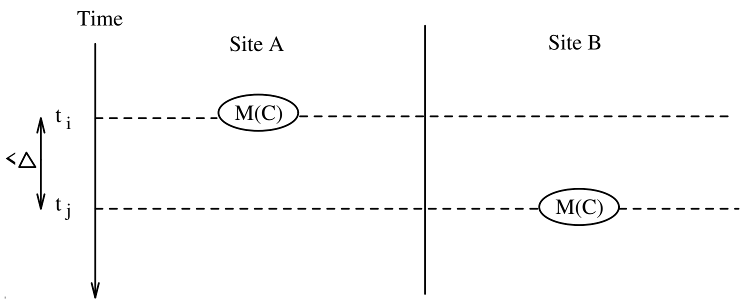

Figure 12: Timebound Synchrony

3.6 Startup and Recovery Properties

3.6.1 Startup

Two general approaches have been developed for coordinating the startup of a group: *collective startup* and *individual startup*. In the first case, the initial membership of the group is assumed to be known in advance, with all sites starting at approximately the same time. In the second case, each site starts with a membership consisting only of itself and sites merge membership views as they learn of one another. Collective startup is generally easier to handle since sites are known *a priori*, and can have implementation advantages if the initial membership is also assumed to be the maximum set of sites that might be group members [MPS93]. Individual startup is more general, but also more complex. For example, this approach requires some known external mechanism for locating other sites, such as a well-known communication port or a shared name server.

To ensure that the root of the ordering graph on each site is well-defined, a startup message S must be generated and delivered to the application before the application is allowed to send or receive other messages. This message carries with it the initial membership, which in the case of collective startup is a list of sites and in the case of individual startup is the identity of the site itself. More formally, if A is an arbitrary site, the following property is guaranteed for all messages in O_A:

$$\forall\ m \in \mathcal{M} : m \in N(O_A) \Rightarrow (S \rightarrow m) \in O_A$$

Startup is closely related to the way in which network partitions are handled, so further details on execution options are deferred until section 3.7 below.

3.6.2 Recovery

Recovery involves restarting a site and reintegrating it back into the group. The problem includes recovering the application process, of course, but here we focus exclusively on aspects of recovery that impact the membership layer and the properties that it guarantees to the application.

The basic recovery requirement for membership is that the membership information of the recovering site C be re-initialized to a valid state, where the details depend on the properties being guaranteed. For example, if no agreement or ordering properties are guaranteed, the recovering site can use the last membership view it had before failing or a view consisting only of itself. On the other hand, if agreement is being enforced, other techniques must be used to ensure that the membership information of the recovering site is consistent with other sites. Possibilities here include replay of missed messages [MPS93] or explicit state transfer from another site [Bir85, BJ87]. Once a new state has been established by the membership layer, the appropriate recovery message $R(C)$ is multicast and subsequently delivered to the application.

After $R(C)$ has been delivered, the recovering site C is considered to be a group member, and message delivery and ordering guarantees must begin to be enforced. In particular, if the membership service guarantees some ordering with respect to application messages, any message that is agreed to be after $R(C)$ must be delivered at C, while messages before $R(C)$ must not be delivered at C. Figure 13 illustrates these ordering graphs. Note that the shaded messages in the figure are not ordered with respect to $R(C)$, which means that they may or may not be delivered to the application on C.

More formally, let A and B be arbitrary sites in the group and m be an arbitrary (application or membership) message. For recovery, the following must be guaranteed in the ordering graph O_C for the membership state on site C to be consistent with the other sites in the group:

$$((m \rightarrow R(C)) \in O_A) \Rightarrow \Box(m \notin N(O_C)) \ \wedge \ (R(C) \rightarrow m) \in O_{A\cap B} \Rightarrow \Diamond((R(C) \rightarrow m) \in O_C)$$

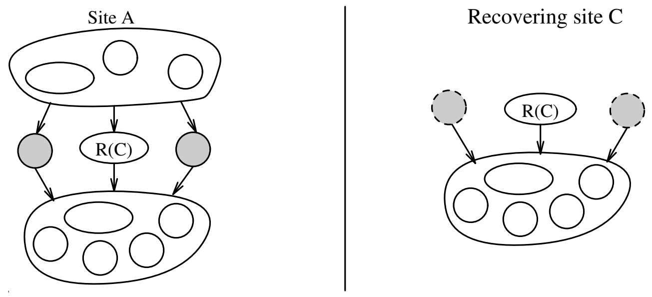

Figure 13: Ordering Guarantees at Recovery

Note, in particular, that there will be ordering guarantees at C only if similar ordering guarantees are being enforced on every other site as well.

3.7 Partition Handling

3.7.1 Overview

A network partition occurs when a subset of sites in a group is unable to communicate with the remainder of the sites. Partitions may be caused by disconnection of the underlying network or by problems such as network congestion or an overloaded gateway processor. A partition is impossible to distinguish from a site failure in a distributed system, so the membership service in each disconnected component of sites will forward failure notifications for the other sites to the application when this occurs. When communication is later reestablished, the service will generate a join message analogous to the recovery membership change message and forward that to the application. In this section, we describe properties that specify more precisely how these failure notification and join messages are generated and delivered.

A number of different approaches are used in membership services to deal with partitions. One common approach is simply to assume they will not occur [Cri91, KGR91, MPS92]. This can be justified by increasing the connectivity of the network or by using other architectural assumptions. In this case, membership properties such as the ordering properties described above will be guaranteed only as long as no partitions occur.

Another approach is to allow the possibility of partitions, but to simplify the problem by allowing execution of the application to proceed only in a single partition [RB91, SM94]. This is similar in intent to the way partitions are handled in some database systems, where techniques such as primary copy [AD76], tokens [MW82], and voting [Tho79] are used to ensure that only one partition remains active. The drawback of this approach, of course, is that it halts the application's execution in the rest of the system, thereby potentially affecting the progress or availability of the application. To address this problem, some services allow computation to proceed in all partitions [DMS94, Hil95, MAMSA94, RFJ93]. A problem with this approach is that it requires the application to deal with the difficult problem of merging states that potentially diverged once the partitioned sites rejoin. Variants of these approaches are also possible depending on the specifics of the application. For example, in some cases, it is feasible to allow continued execution in all partitions, but to avoid the problem of merging states by selecting a distinguished partition and forcing all other sites to adopt this state.

The operation of the membership layer is independent of issues regarding application state, so here we concentrate on describing guarantees or services that membership provides to the application.

Specifically, the focus in this section is on describing properties that are enforced when the partition occurs and when separated sites are subsequently rejoined to reform the original group. The properties discussed in previous sections (e.g., agreement, message ordering) remain relevant even when partitions occur, although each will now be enforced separately within each partition. We assume that membership operates continuously in all partitions, maintaining its own view of the membership and implementing the properties required by the application. It also forwards membership change messages to the application level on each site as usual, independent of policy choices made at that level.

3.7.2 Augmented Failure Notification

As noted, some application policies require that only one partition remains active following a partition. To identify this *active partition*, a predicate must be defined that, given any partition of sites into groups, evaluates to true in at most one of the groups. A typical example of such a predicate would be one that returns true if the group contains more than half the maximum number of sites and false otherwise. Note that this requires knowing *a priori* the maximum number of sites. Such a requirement is satisfied in systems that either start with a list of group members (e.g., [MPS93]), have a permanent list of possible sites that can participate (e.g., [Bir85]), or where the physical configuration implies that the identity of all sites is known and fixed (e.g., [KGR91]).

A majority predicate of this type is easy to implement in the membership layer, assuming that the service guarantees agreement and total order, and has knowledge about the maximum size of the group. To notify the application when a site is no longer a member of the majority group, a failure notification message $F(C)$ is augmented with an extra field indicating whether the (presumed) failure of C has caused the group size to shrink to the point that a majority can no longer be guaranteed. Note that such an indication only implies the *possibility* of a partition, not its actual existence; for example, so many sites may have actually failed that only a minority of the sites remain functioning. Similarly, a recovery message $R(C)$ is augmented to indicate whether the recovery of a site C has brought the group size back over the majority threshold.

3.7.3 Collective Failure Notification

As already noted, when a partition occurs, a basic membership protocol would forward a stream of membership change messages to the application, each indicating the failure of one of the sites in the other partition. *Collective failure notification* expands this notion by grouping together failure notifications for all the sites in a partition and forwarding them to the application in a single message. This message is ordered according to whatever criteria is being used to order individual membership change messages.

Collective failure notification is useful from the application's perspective, since it can be used to avoid a lengthy transition period in which the failures of multiple sites must be processed in succession. It is also straightforward to implement if the membership service already provides properties that require agreement of all sites, such as ordering with respect to application messages. In such cases, sites in the other partition will fail to participate in the required protocol and can therefore be collectively identified. This property allows agreement on the entire group of sites to be performed at once and forwarded to the application in a single message.

To realize the functionality of collective notification, we augment the failure notification message to indicate the failure of multiple sites $S_1, S_2, \ldots$, using the notation $F(S_1, S_2, \ldots)$. The analogous process occurs in both (or, in general, in all) partitions. Note that the group membership in the two partitions are non-overlapping after the failure notification messages are delivered.

Collective failure notification can be defined more formally as follows. Without loss of generality, assume that the partition results in the sites being divided into two sets of sites P and Q whose intersection is empty. Let $A, B \in P$, $C \in Q$, and D be an arbitrary site in either partition. Furthermore, let F_P and F_Q be the respective collective failure notification messages delivered to the sites in P and Q; $S \in F_P$ is used to denote that site S is included in the failure notification message F_P, and similarly for $S \in F_Q$. Then, collective failure notification can be characterized as ensuring the following for the the ordering graphs of A, B, and C:

$$F_P \in N(O_A) \Rightarrow \Diamond(F_P \in N(O_B))$$

and

$$(M(D) \to F_P) \in O_A \Rightarrow \Diamond((M(D) \to F_P) \in O_B) \ \wedge \ (F_P \to M(D)) \in O_A \Rightarrow \Diamond((F_P \to M(D)) \in O_B)$$

and

$$\forall D \in \mathcal{S} : (D \in P \Rightarrow D \in F_Q) \ \wedge \ (D \in Q \Rightarrow D \in F_P)$$

The first rule states that agreement is reached within each partition on collective failure notification messages, while the second guarantees that any message ordering constraints that apply are enforced within each partition. Since $P \cap Q = \emptyset$, the third rule ensures that the membership views in P and Q are non-overlapping after the failure notifications have been delivered.

Figure 14 illustrates this property, where shaded nodes represent membership change messages reporting changes that happened prior to the partition, the dashed line represents the time when the partition occurred, and F_P and F_Q are as above.

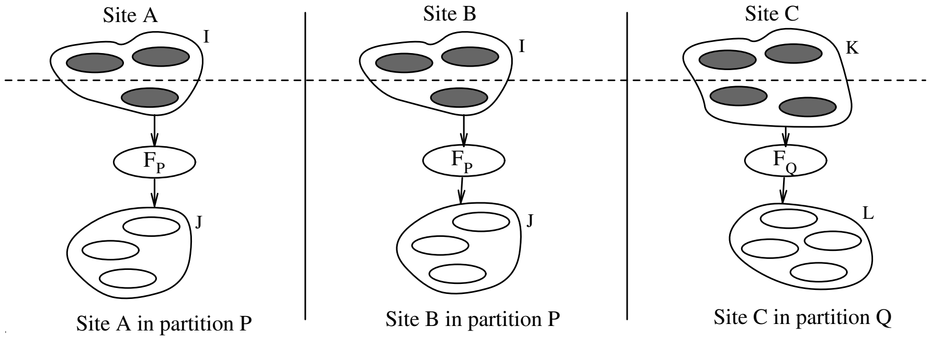

Figure 14: Collective Failure Notification

3.7.4 Ordered Collective Join

When a partition is repaired, a basic membership algorithm would integrate each site from the other partition into its group membership individually. *Ordered collective join* extends this notion to merge the sites in the two partitions into a single larger group collectively and in a way that is ordered consistently with respect to other membership change messages. The value of such a property is great. For example, without it, all ordering properties will be compromised and even properties like agreement are not well-defined if a site becomes a member of more than one overlapping view. A special *merge* message is used to implement the collective join. This message includes a list of the members of the new group and is totally ordered with respect to other membership change messages.

Figure 15 illustrates this property, where the shaded nodes represent membership change messages in the new membership after the merge and the larger circle in the middle is the merge message.

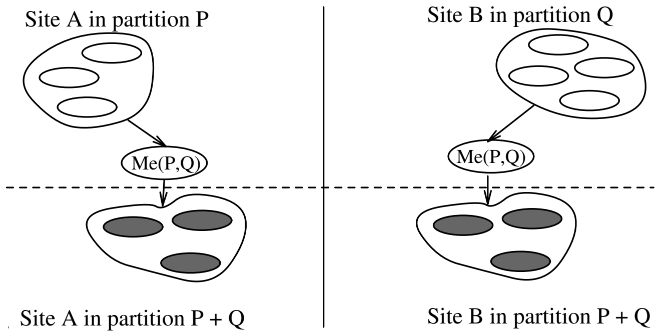

Figure 15: Ordered Collective Join

Ordered collective join can be defined more formally as follows. As above, let P and Q be the set of sites in each partition, $A \in P$, $B \in Q$, and C be an arbitrary site in either partition. Furthermore, let $Me(P,Q)$ be the merge message indicating the merging of P and Q. Then, ordered collective join can be characterized as ensuring the following for the the ordering graphs of A and B:

$$\forall M(C) : (M(C) \rightarrow Me(P,Q)) \in O_A \Rightarrow M(C) \notin N(O_B)$$

and

$$\forall M(C) : (Me(P,Q) \rightarrow M(C)) \in O_A \Rightarrow \diamond((Me(P,Q) \rightarrow M(C)) \in O_B)$$

3.7.5 Ordering Partition Handling Messages

When a partition occurs, the ordering graphs of the sites in the two separated subgroups will generally evolve differently. Subsequent membership changes in one partition will result in a membership change message being issued in one subgroup but not the other, while application messages will also appear in the ordering graphs of only one set of sites. As a result, when collective notification is used, the relevant failure notification and join messages—the $F(...)$ and $Me(...)$ messages from above—delineate boundaries in the ordering graph at which the graphs at different sites diverge and then reconverge, respectively. In some sense, then, a failure notification message can be viewed as creating two independent streams of messages to be delivered, one in each partition, while a merge message can be viewed as merging the two streams back into one. This property distinguishes such *partition handling messages* from normal membership change messages, which are delivered as part of a single stream.

An implication of the differences between the two types of membership change messages is that the ordering properties discussed above in section 3.4 cannot be used directly to argue about ordering properties of partition handling messages. For example, although messages sent after receiving a merge message can easily be ordered after that message, it is less clear how to order messages that were sent by members of the separate partitions before receiving the merge message, or even to which sites they should be delivered. Perhaps the simplest solution is to deliver such messages only to sites that were in the partition in which they were sent. This strategy is, however, contrary to the semantics implemented by systems such as Psync [PBS89], which automatically propagates messages of this type to all sites for recovery purposes by virtue of its negative acknowledgment scheme for retransmitting lost messages.

Extended virtual synchrony is an example of a stronger property that can be augmented to include partition handling messages. This *extended virtual synchrony with partitions* property requires that messages sent before receiving the merge message be delivered before that message, and analogously, that all messages sent before receiving the failure notification message be delivered before that message. Note, of course, that this guarantee only applies within each partition. Figure 16 illustrates this property for partition join.

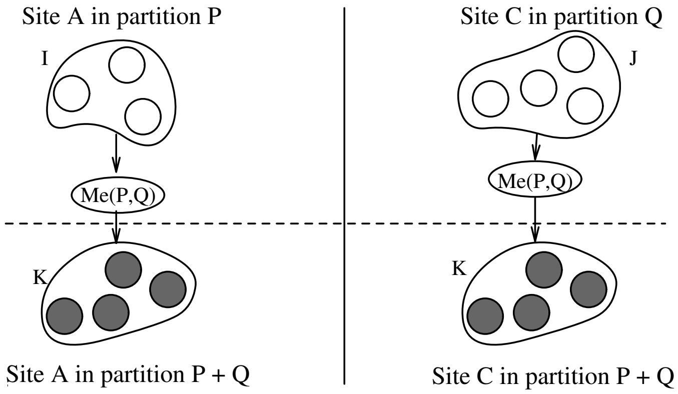

Figure 16: Extended Virtual Synchrony with Partitions

This property can be defined more formally as follows. As above, let P and Q be the set of sites in the partitions; F_P and F_Q be the failure notification messages delivered to sites in P and Q, respectively; and $Me(P,Q)$ be the merge message joining P and Q. Furthermore, assume A, B, and C are arbitrary sites such that $A, B \in P$ and $C \in Q$. Then, extended virtual synchrony with partitions can be characterized as ensuring the following for the ordering graphs of A and B at the time of partition:

$$\forall a_i : (send(a_i) < del_A(F_P)) \Rightarrow \Diamond((a_i \rightarrow F_P) \in O_B)$$

and

$$\forall a_i : (del_A(F_P) < send(a_i)) \Rightarrow \Diamond((F_P \rightarrow a_i) \in O_B)$$

An analogous property holds for sites in partition Q with respect to F_Q. Similarly, the following holds at the time of partition join:

$$\forall a_i : (send(a_i) < del_A(Me(P,Q))) \Rightarrow \Diamond((a_i \rightarrow Me(P,Q)) \in O_B) \ \wedge \ \Box(a_i \notin N(O_C))$$

and

$$\forall a_i : (del_A(Me(P,Q)) < send(a_i)) \Rightarrow \Diamond((Me(P,Q) \rightarrow a_i) \in O_{B \cap C})$$

Among other things, this property simplifies the problems associated with merging application states after a partition. In particular, since all application messages are either before or after the merge message, a consistent cut is created in the ordering graphs of all sites. To implement a merge of the application states, then, messages carrying the respective states can be sent from each site immediately after the merge message, with the assurance that they will be delivered after the merge and before any subsequent application messages. Of course, the difficult semantic problems associated with merging application states remain.

Our extended virtual synchrony with partitions property is derived from extended virtual synchrony as described in [MAMSA94]. The algorithm described in that paper provides the guarantees outlined above, plus additional ordering properties for messages that are sent around the time that the partition occurs.

4 Relating Properties

Even though we have defined membership properties in isolation, in reality, one property often builds on the functionality of another property or is otherwise related to it. In the above, two general types of relationships between properties can be identified: *dependency* and *inclusion*. A dependency occurs between properties addressing different aspects of membership when one property cannot be satisfied unless the second property is also satisfied. For example, totally ordering membership messages is impossible unless all membership messages are in the ordering graphs of all sites, so total ordering depends on agreement. An inclusion occurs between properties addressing the same aspect of membership when one property is strictly stronger than the other in the sense that satisfying the first also satisfies the second. For example, for properties that order membership change messages with respect to application messages, extended virtual synchrony includes virtual synchrony.

Formally, both types of relationships can be expressed as implication. For example, consider agreement (AG) and total ordering of membership change messages (TO). First, define Θ as in section 2 to be the sequence of all ordering graphs in the system for a given execution run, and S to be the set of Θ for all possible execution runs. Furthermore, define $sat(p, \Theta)$ to be true iff property p, which is expressed as a temporal formula, holds for Θ, and $sat(p, S)$ to be true iff p holds for all $\Theta \in S$. Then, the relationship between AG and TO can be expressed simply as:

$$\forall S : sat(TO, S) \rightarrow sat(AG, S)$$

In other words, any system that satisfies TO must also satisfy AG. Inclusion is argued similarly. Examples of formal dependency arguments using expanded property definitions can be found in [Hil96].

It is also worth noting that, although dependency and inclusion are expressed in the same way formally, the two concepts are worth separating for practical reasons. Dependencies are used to express relationships between properties that deal with different aspects of the membership problem; for example, agreement specifies in which ordering graphs a membership change message will appear, while total ordering specifies the order in which such messages will be delivered to the application. Inclusions, on the other hand, are used to express relationships between different variants addressing the same aspect of membership, such as different ways in message ordering can be performed. In a system such as [BS95] in which properties are implemented by separate software modules, dependencies dictate that a given module be included in any configuration if another that depends on it is also included, while inclusions indicate a choice of modules implementing variants of a property.

Dependency graphs are a method for expressing these types of relationships between properties. Such a graph is a directed graph, where each node represents a property, an edge from node A to node B indicates that A depends on B, and the nesting of node B in node A indicates that A includes B. Choice between two or more properties for which no inclusion relationship exists is represented by grouping them together within an unlabeled node. The figure is simplified by omitting edges that represent transitive dependencies. Also, if a property A includes B and both depend on C, only the edge from B to C is drawn. Finally, note that the graph will be cyclic if, for example, properties A and B depend on one another. This can be interpreted simply as indicating that a membership service must either guarantee all the properties in the cycle or none of them.

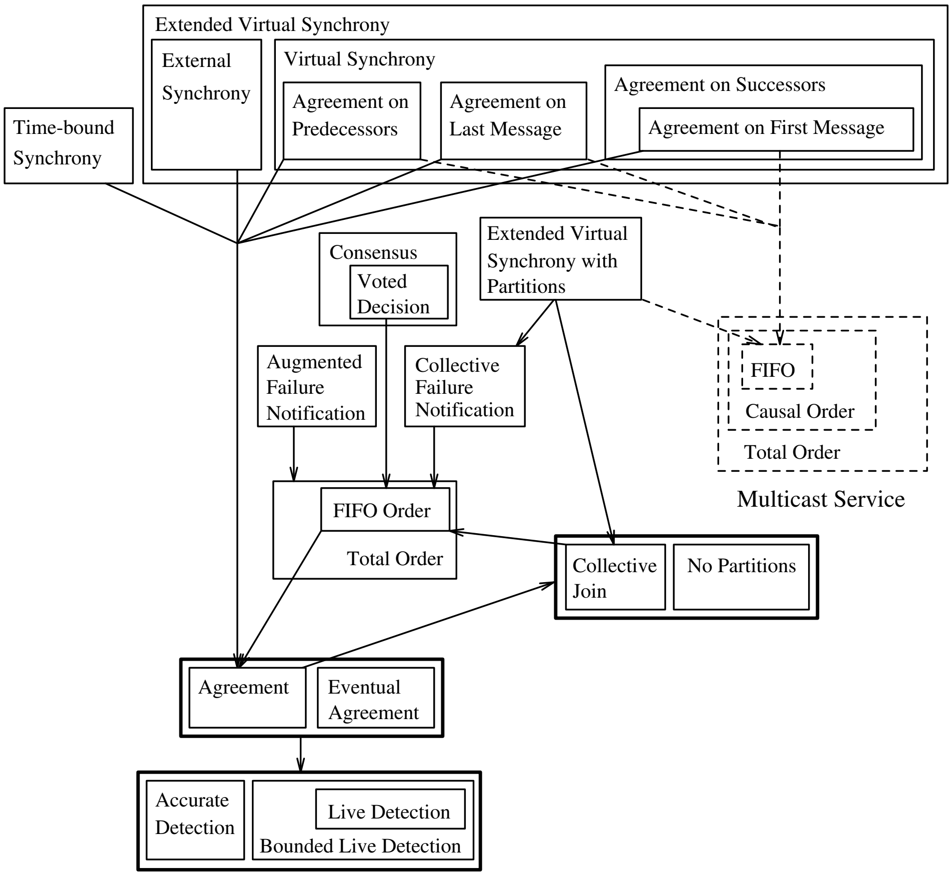

Figure 17: Membership Dependency Graph

Figure 17 gives the dependency graph containing the properties discussed in earlier sections. In the figure, the nodes formed from dashed lines represent properties implemented by the reliable multicast component of the system, which are needed for certain membership properties but are not addressed in detail in this paper.

The dependency graph represents the relationships between properties, and therefore all possible legitimate combinations of properties. The simplest possible membership services base the local view of the membership only on local live or accurate detection. More advanced services provide agreement augmented with various ordering and other properties. The inherently large number of combinations represented by the graph means that it can serve as a useful tool for configuring custom membership services for systems such as [BS95] mentioned above. In this case, the designer first chooses properties to be guaranteed, and then takes the transitive closure of these nodes in the graph to determine the set of properties that must be included. Using this approach, a large number of different membership services can be configured, each implementing a different collection of properties.

5 Characterizing Existing Services

5.1 Overview

The properties defined in previous sections can be used to characterize existing membership services. Doing so carries with it a number of caveats, however. For example, service properties defined in the literature are often properties of specific implementations that are difficult to relate to abstract properties as seen by the application. Furthermore, membership services are not uniform in how they interact with the application. In our approach, a service signals membership changes by forwarding membership change messages to the application in the regular message stream. We call services that follow this approach, including ISIS [BSS91], Consul [MPS92], and Transis [ADKM92a, ADKM92b], *delta-based services* since they deliver the changes ("deltas") to the application. Another approach is for the service to deliver the entire current membership set whenever a (possible) membership change occurs. We call services that follow this approach, including [Cri91, AMMS+93, RFJ93, SR93], *set-based services*. Although properties for the two types of services are typically stated differently, mappings between them can usually be constructed in a straightforward manner.

This section gives an overview of several existing membership services and characterizes their properties using the terminology defined in this paper. These services are then summarized in tabular form in section 5.8.

5.2 Consul

The membership service of the Consul system [MPS92, MPS93] assumes asynchronous communication and sites that experience crash or performance failures. The service is built using Psync [PBS89, MPS89], a multicast service that preserves the causal ordering of messages using a *context graph* abstraction. Psync guarantees reliable multicast communication, so that context graphs on various sites are identical except for transmission delays.

Failures are detected at each site by monitoring the message stream from all other sites in the group. If no message is received from some site within a specified interval, the site is suspected to have failed. When a site suspects the failure of another site, say S, it initiates an agreement process by multicasting an "S is down" message. Upon receiving such a message, a site decides based on the state of its context graph whether or not it agrees with the suspicion. If it agrees, it multicasts "Ack, S is down"; otherwise, it multicasts "Nack, S is down". Since messages are multicast using Psync, each message will eventually be received by every other site, including those that may fail and later recover. If all responses are Acks, S is removed from the membership set at all sites.

Consul deals with simultaneous failures—i.e., the failure of one or more sites during the execution of the agreement protocol for an initial suspicion—by means of *simultaneous failure groups* (sf–groups). The sites in an sf–group are removed from the membership simultaneously. However, sf–groups may vary from site to site, so the ordering property guaranteed is weaker than the total order of membership changes property defined above. sf–groups are transparent to the application, and are used primarily as a means of optimizing the agreement process.

Consul's membership service is live, with a level of confidence in failure detection that can be characterized as consensus. Although the system does not generate a singular membership change message in the sense used in this paper, it can be shown that it guarantees agreement and FIFO ordering of membership change messages. The service also guarantees agreement on first and last messages, as well as agreement on successors.

5.3 ISIS

The membership service of ISIS described in [BJ87] consists of a distributed site view management component and an ordered multicast primitive (GBCAST) that is used to multicast and order membership change messages to ensure virtual synchrony. Site failures are detected by sending "Hello" messages between sites. If an Hello message from a site is not received within a specified period, it is assumed to have failed.

Each site maintains a *site view*, which is the set of sites it deems to be operational. The view management algorithm ensures that each operational site goes through the same sequence of site views. The sites in a view are ordered uniquely according to the view in which they first became operational, with ties broken by site identifier. The "oldest site" in this ordering is called the *view manager* and is responsible for initiating the view management protocol when it detects a site failure or recovery. If a site detects that all sites older than itself have failed, it takes over as the new view manager.

View changes are done using a two-phase commit protocol. First, the manager multicasts the proposed site view. If this view is new—i.e., a newer site view transmitted by some other manager has not been received—a site sends a positive acknowledgement. Otherwise, it replies with a negative acknowledgement and the more recent view. If all acknowledgements are positive, the manager multicasts a commit message. If a negative acknowledgement is received, or if new site failures or recoveries occur, the protocol is restarted. This protocol guarantees that the view managers on all operational sites process site views in the same order.

GBCAST is used to multicast membership change messages among the sites comprising the group. GBCAST guarantees total ordering with respect to all messages; that is, no message is delivered before a message sent using GBCAST on one site and after it on another site. Furthermore, membership change messages sent by GBCAST are delivered after every message from the failed site. Because of these properties, informing the application about a membership change is just a matter of multicasting the appropriate message using GBCAST.

The ISIS membership service is live, with failure detection being based on single site suspicion. Since GBCAST messages are totally ordered, the service realizes both total ordering of membership changes and virtual synchrony. ISIS deals with partitions by allowing computation to continue in at most one partition, an approach supported by the augmented failure notification technique described in section 3.7.2.

5.4 Cristian's Synchronous Membership Protocols

In [Cri91], Cristian presents three group membership protocols built on the assumption that the underlying system provides synchronous reliable atomic broadcast primitives. The protocols handle faulty sites leaving the membership and fault-free or repaired processors joining. All three protocols provide the same service abstraction, but make slightly different tradeoffs in the implementation. For example, in the *periodic group creation* protocol, each site multicasts "Present" messages at agreed times, which allows a consistent membership set to be calculated at all sites based on the assumption of synchronous reliable communication. Another protocol, the *attendance list* protocol, reduces message overhead in the absence of joins and failures by circulating an attendance list through all sites once per period instead of using Present messages.

The protocols proposed by Cristian guarantee a number of properties. In the terminology of [Cri91], these include the following:

- *Agreement on group membership*: Any two sites in the same group have identical membership views.

- *Reflexivity*: A site that has joined the group belongs to the membership (excludes the trivial solution of an empty membership list).
- *Bounded join delay*: The time for a site to join a group is bounded by a constant.
- *Bounded departure detection delay*: The time to detect the departure of a site (e.g., because of a failure) is bounded by a constant.
- *Bounded group formation delay*: The time between the point when the first and last sites join the group during initial group formation is bounded by a constant.
- *Bounded group change delay*: The time elapsed between a site leaving one group and joining a new group is bounded by a constant.

Mapped into our abstract properties, the bounded join and departure detection delay properties combined with the reliability and synchrony assumptions guarantee our liveness and accuracy properties. Agreement on group membership corresponds to our agreement property. Reflexivity is guaranteed by liveness. Bounded group formation delay is equivalent to timebound synchrony. Bounded group change delay only makes sense in the context of this particular algorithm, since in our framework, a membership view never expires. Although not explicitly listed as a property, these protocols also guarantee total order of membership views for sites within the same partition, which corresponds to our total order of membership messages property. Note that these protocols do not attempt to order membership changes with respect to application messages.

5.5 Mars

The membership service in the Mars system is another example of a synchronous protocol [KGR91]. Mars builds on a physical ring architecture in which the network is accessed using a time division multiple access (TDMA) strategy based on common global time, i.e., access to the physical medium is divided into dedicated time slots that are allocated *a priori* to sites in a round-robin fashion. A *TDMA cycle* is defined as N consecutive slots, where N is the number of sites in the system. Based on these assumptions, the Mars membership protocol handles processor crash failures and sites failing to send or receive, under the assumption that at most one failure occurs in each TDMA cycle. Failure detection exploits the TDMA communication strategy: since each site is expected to send data in each of its slots, the lack of transmission is interpreted as an indication of failure. To reduce network failures, every message is transmitted twice in the same time slot. Agreement on membership changes is reached by forwarding information in each message about all messages received in the previous cycle. This essentially propagates the list of sites the sending site knows to have been alive in the previous TDMA cycle.

Due to the way in which membership is integrated with the communication system, the service realizes ordering with respect to application messages. Total order of membership changes is trivially guaranteed given the assumption of no more than one failure per TDMA cycle and since each processor observes the same membership changes in each cycle. Agreement on first and last messages is also guaranteed, since if one site receives a message, every non-failed site also receives the message, while if no site receives the message, the sending site is deemed to have failed. Although not guaranteed by the basic algorithm, agreement on successors and predecessors is easy to implement given the communication system and failure assumptions.

5.6 Totem

The Totem message ordering and membership protocol is described in [AMMS+93]. The protocol is based on a logical token passing scheme, where the token is used for total ordering of messages, reliable message transmission, flow control, and membership. All messages in Totem are totally ordered reliable multicasts.

Membership in Totem is based on reforming the group whenever a group membership change is suspected, i.e., whenever a site failure or token loss is detected or a new message from a site outside the group is received. The site that initiates the membership change multicasts an "Attempt Join" message and then shifts to a "Gather" state while it waits for Attempt Join messages from other sites. After a specified time period has elapsed, it then multicasts a Join message that contains the identifiers of those sites from which it received messages, and shifts into a "Commit" state. In this state, sites reach agreement on the new membership, as follows. Each time a site receives a Join message containing sites of which it was previously unaware, it transmits a new Join message with the updated information. Once a site receives Join messages from all sites it included in its most recent transmission and the membership in all these messages agree, the protocol on that site terminates.

Based on our property definitions, the Totem membership service is live, with failure detection based on single site suspicion. It also guarantees FIFO ordering of membership messages, extended virtual synchrony, and extended virtual synchrony with partitions.

5.7 Weak, Strong, and Hybrid Membership Protocols

A family of three membership protocols labeled as *weak*, *strong*, and *hybrid* is described in [RFJ93]. All three deal only with ordering membership views and establishing agreement between views on different sites, with no attempt made to order membership changes with respect to application messages. The weak protocol simply guarantees that the views of all sites converge to a single consistent view if there are no failures for some period of time, More precisely, define $V_i \prec_s V_j$ if site S installs membership view V_i before installing view V_j. Let $\overset{*}{\prec}$ be the transitive closure of $\prec_s$, i.e.,

$$\forall i, j : V_i \overset{*}{\prec} V_j \Leftrightarrow \exists s : V_i \prec_s V_j$$

Then, the weak protocol ensures that $\overset{*}{\prec}$ is a partial ordering relation, i.e., it is irreflexive, transitive, and asymmetric. In particular, asymmetry guarantees that if one site installs V_i before V_j, then no other site will install them in the opposite order. The protocol also guarantees convergence: if site S has view V_i and site T has a different view V_j, then assuming no failures, there is eventually a state in which both sites have the same membership view V_k such that $V_i \overset{*}{\prec} V_k$ and $V_j \overset{*}{\prec} V_k$. Translated into our properties, the weak protocol is live, guarantees no agreement on membership changes but eventual agreement on membership views, and ensures none of our ordering properties. We have not defined a property that corresponds to the partial ordering of membership views, but it would be easy to define such a property within our framework.

Informally, the strong membership protocol ensures that membership changes are seen in the same order by all members. More precisely, it guarantees the properties of the weak protocol plus the following:

P_1: In any global state, if a site s has joined g locally, all other sites in g have either joined g locally or are in a transition state (i.e., members of no group).

P_2: In the presence of performance failures or network partitions, members of concurrently active groups are disjoint.

P_3: In the absence of performance failures or partitions, all active members go through the same sequence of groups (total order).

Based on the definitions in this paper, the strong membership protocol guarantees agreement and total order of membership changes. P_2 also implies that collective failure notification and ordered collective join notification are satisfied when partitions occur. Also, as discussed in section 3.5.1, P_1 is the source of our external synchrony property.

The hybrid membership protocol provides guarantees that are intermediate between those of the weak and strong protocols. In particular, it ensures the properties of the weak protocol, with the additional guarantee that there is a single leader for each group. Thus, an algorithm similar to strong membership is executed when the leader of a group changes; otherwise, an algorithm similar to weak membership is executed. Essentially, the hybrid protocol guarantees that in any global state, if a site S decides locally that site T is the leader of the group g, then all other sites in g either consider T the leader or are in a transition state (i.e., have not decided a leader). This property would be equivalent to having all membership changes that affect the leader have our external synchrony property.

5.8 Summary

Table 1 summarizes the properties guaranteed by each service discussed in this section. The properties

	Lv	Ac	Co	Ag	Fo	To	Af	Al	As	Ap	Vs	Ev	Es	Ts	Np	An	Cn	Oj	Ep
Consul	x		All	x	x		x	x	x						x				
ISIS	x		1	x	x	x	x	x	x	x	x					x			
Cristian	x	x	1	x	x	x								x	x				
Mars	x		2	x	x	x	x	x	x*	x*				x	x				
Totem	x		1	x	x	x*	x	x	x	x	x	x	x				x	x	x
Weak	x		1																
Strong	x		All	x	x	x							x				x	x	

Table 1: Properties Enforced by Existing Services

are abbreviated as follows:

- *Accuracy and Liveness*: Liveness (Lv), accuracy (Ac), level of confidence (Co) ranging from one site suspicion (1) to consensus (ALL).
- *Agreement*: Agreement (Ag).
- *Ordering Properties*: FIFO ordering (Fo), total ordering (To), agreement on first (Af), agreement on last (Al), agreement on successors (As), agreement on predecessors (Ap), virtual synchrony (Vs), extended virtual synchrony (Ev).
- *Bounded Change Properties*: External synchrony (Es), timebound synchrony (Ts).

- *Partition Handling*: No partitions (Np), augmented failure notification (An), collective failure notification (Cn), ordered collective join (Oj), extended virtual synchrony with partitions (Ep).

In the table, "x*" denotes a property that is not guaranteed by the service, but could easily be added. Recall also that a system that provides a given property P also provides all properties that satisfy the inclusion relationship with P, as discussed in section 4.

6 Conclusions

In this paper, we have defined abstract properties of membership services and described how they relate to one another. Ordering graphs were used to define the properties by illustrating their effect on the order in which messages are delivered to the application. For example, the agreement property guarantees that every membership change message is received at all sites, thereby providing the basis for consistent decision making. Ordering properties extend this notion to ensure consistent message ordering information as well, differing in the degree of consistent information provided. The tradeoff is that the stronger properties require more expensive algorithms and more synchronization between sites, which potentially results in less efficient execution. The way in which properties relate to one another was illustrated using dependency graphs.

As noted in section 5, existing membership services can be characterized in terms of these properties. With the exception of [SR93], however, membership papers concentrate on describing the properties of a particular algorithm or system, rather than providing a more global view. [SR93] gives a decomposition of membership services into three components: Failure Suspector, Multicast Component, and View Component. The Failure Suspector is responsible for detecting membership changes and propagating changes that it has detected to other Failure Suspectors. In our framework, this corresponds to change detection properties. The Multicast Component is responsible for implementing virtually synchronous communication. Compared to our approach where multicast is separate from membership, their Multicast Component combines these two functions. The View Component ensures that all sites have the same view of the membership. As such, it essentially implements our agreement property. However, our decomposition is much more extensive and identifies a large number of properties in contrast to just identifying more implementation oriented components of membership services.

Future work will include identifying additional properties of membership and specifying them using the ordering graph approach. Research will also continue on a system that allows membership services to be implemented by configuring software modules realizing individual properties in a common software infrastructure [HS94]. The current prototype runs on Mach-based DecStation 5000/240s and is based on extending the hierarchical model of the *x*-kernel [HP91] to support finer-grain composition [BS95].

Acknowledgements

The comments from Laura Sabel and Keith Marzullo on an earlier version of the paper greatly improved the presentation.

References

[AD76] P. Alsberg and J. Day. A principle for resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 562–570, Oct 1976.

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms (Lecture Notes in Computer Science 647)*, pages 292–312, Haifa, Israel, Nov 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Jul 1992.

[AMMS+93] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International IEEE Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[BG93] K. Birman and B. Glade. Consistent failure reporting in reliable communication systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[Bir85] K. Birman. Replication and fault-tolerance in the ISIS system. In *Proceedings of the 10th ACM Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, Dec 1985.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[BS95] N. T. Bhatti and R. D. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of SIGCOMM '95*, Cambridge, MA, Aug 1995.

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

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

[DMS94] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Mar 1994.

[EL90] P. D. Ezhilchelvan and R. Lemos. A robust group membership algorithm for distributed real-time system. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 173–179, Lake Buena Vista, Florida, Dec 1990.

[EL95] K. Echtle and M. Leu. Fault-detecting network membership protocols for unknown topologies. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications 4*, pages 69–90. Springer-Verlag, Wien, 1995.

[GT92] R. A. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, May 1992.

[Hil95] M. A. Hiltunen. Membership and system diagnosis. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, Bad Neuenahr, Germany, Sept 1995. To appear.

[Hil96] M. A. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1996. In preparation.

[HP91] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HS93] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, USA, Oct 1993.

[HS94] M. A. Hiltunen and R. D. Schlichting. A configurable membership service. Technical Report 94-37, Department of Computer Science, University of Arizona, Tucson, AZ, Dec 1994.

[HS95] M. A. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, USA, Apr 1995.

[KG94] H. Kopetz and G. Grunsteidl. TTP - A protocol for fault-tolerant real-time systems. *Computer*, 27(1):14–23, Jan 1994.

[KGR91] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[LE90] R. Lemos and P. Ezhilchelvan. Agreement on the group membership in synchronous distributed systems. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, Otranto, Italy, Sep 1990.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, Jun 1994.

[MPS89] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 42–52, Seattle, Washington, Oct 1989.

[MPS92] S. Mishra, L. L. Peterson, and R. D. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Wien, 1992.

[MPS93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed System Engineering*, 1:87–103, Dec 1993.

[MW82] T. Minoura and G. Wiederhold. Resilient extended true-cope token scheme for a distributed database system. *IEEE Transactions on Software Engineering*, SE-8(3):173–189, May 1982.

[OIOP93] H. Orman, E. Menze III, S. O'Malley, and L. Peterson. A fast and general implementation of Mach IPC in a network. In *Proceedings of the 3rd Usenix Mach Conference*, pages 75–88, Apr 1993.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, Aug 1991.

[Rei94] M. Reiter. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[RFJ93] R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specification, design, and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 2–11, Princeton, NJ, Oct 1993.

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

[SM94] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 138–147, Dana Point, CA, USA, Oct 1994.

[SR93] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd International Conference on Fault-Tolerant Computing*, pages 534–543, Jun 1993.

[Tho79] R. H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, Jun 1979.

[vSCA94] P. van der Stok, M. Claessen, and D. Alstein. A hierarchical membership protocol for synchronous distributed systems. In K. Echtle, D. Hammer, and D. Powell, editors, *Proceedings of the 1st European Dependable Computing Conference (Lecture Notes in Computer Science 852)*, pages 599–616, Berlin, Germany, Oct 1994.