Agreeing on Processor Group Membership in Timed Asynchronous Distributed Systems

Flaviu Cristian[*]
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093−0114

Frank Schmuck
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120−6099

Abstract

We introduce the *timed asynchronous distributed system model* to describe existing asynchronous distributed systems subject to unbounded processing and communication delays, failures and recoveries. We then describe five increasingly strong specifications for *processor-group membership* services in timed asynchronous systems subject to partitioning. We also propose five distributed protocols that implement these specifications despite arbitrary numbers of crash/performance processor failures and omission/performance communication failures, and prove their correctness. Finally, we show how two of the protocols can be adapted to implement a *highly available processor leadership* service that ensures the existence of at most one leader at any point in real-time.

Keywords: Agreement, Asynchronous System, Distributed System, Failure Detection, Fault-Tolerance, Leader Election, Mutual Exclusion, Network Partition, Processor Membership, Replicated Data.

[*]Partially supported by grants from the Air Force Office of Scientific Research, Sun Microsystems, and the Microelectronics Innovation and Computer Research Opportunities in California.

UCSD Technical Report CSE95-428, 1995

1 Introduction

Most published distributed algorithms are described in terms of a fixed group of active processes that communicate through messages to achieve a certain goal. In real systems, however, such groups are *dynamic*, shrinking as failures occur, and growing as communication services and processes recover. An earlier paper [Cri91b] has specified and proposed several solutions to the *processor group membership* problem in synchronous distributed systems in which atomic broadcast guarantees timely communication between any pair of correct processors. Such communication *certainty* is achieved by using real-time executives to bound communication delays and massive redundancy to avoid physical partitioning. That paper has also shown how a *process group* membership service can be implemented on top of a processor group membership service. This paper investigates how one can specify and solve the processor group membership in asynchronous distributed systems in which unbounded delays and partition occurrences make communication *uncertain*. We do not deal with the problem of implementing a process group membership service on top of a processor group membership service, since the solution discussed in [Cri91b] can be adapted to an asynchronous communication environment in a straightforward way.

Our objective of specifying and solving the membership problem in the presence of communication uncertainty makes the problem much more complex than the synchronous case studied in [Cri91b]. Indeed, when processes cannot distinguish between peer processor failures and communication failures, two correct processes which lose communication with each other may take conflicting actions if each assumes the failure of the other, as illustrated by the membership protocol of [Car85]. In the Tandem protocol, a loss or delay of (at least) two successive "I am alive" messages between two processors can cause them to adopt diverging membership views, which in turn can lead to conflicting actions, such as the promotion of two servers of a primary/backup process group to the role of primary. The resulting system behavior, known as "split brains" behavior in Tandem terminology [Car90], can result in total system crashes.

We begin by defining what we mean by an asynchronous system and its likely failures. While the asynchronous system model used in the theory community [FLP85] has no notion of time, our system model assumes that processors have local clocks that allow them to measure the passage of time. To avoid confusion, we refer to the system model of [FLP85] as the *time-free asynchronous system model* and we call our model the *timed asynchronous system model*. Most existing distributed systems based on non-real-time operating systems and communication services, such as Unix and UDP, are timed asynchronous as captured by our definition.

The introduction of the timed asynchronous system model, and its comparison to the time-free asynchronous system model discussed in [FLP85] are key contributions of our paper. We then proceed by presenting a sequence of five increasingly strong specifications for processor membership in timed asynchronous systems and comment on how they can be used in practice by different higher level applications. This sequence of membership specifications and the discussion relating their use to the requirements placed on the membership service by the higher level applications is a second major contribution of our paper. Another key contribution is the description of a suite of five protocols for implementing the specified services and a description of how two of these services can be used to solve the highly available leadership problem [Cri91b]. All (necessarily ambiguous) English protocol descriptions are complemented by unambiguous formal protocol code in the appendix. The appendix also contains proofs that all five protocols satisfy their respective specifications and that the two solutions given for solving the highly available leadership problem are correct.

2 Related Work

According to [CHTCB96, ACBMT95], the membership problem was first defined for synchronous systems in [Cri91b]. Since then this problem has also been subject of intense investigation for asynchronous systems.

Two types of asynchronous membership services have been described: primary-partition membership [RB91, KT91, MPS92, MSPMA94, HS95, MFSW95, Nei96] and partitionable membership [ADKM92, JFR93, vRBM96, BBDG94, EMS95]. Some of these membership services explicitly assume a time-free underlying system model (e.g., [RB91, MPS92, HS95, BBDG94]), while for others it is not entirely clear whether the underlying system model is time-free or some variant of our timed system model.

The primary partition services are intended for systems that are not subject to partitions or for systems that allow the group membership to change in at most one physical network partition, the so-called primary partition. Because primary partition membership services attempt to ensure the existence of a single group at any point in time, they are similar to our 'three-round' and 'five-round' majority membership protocols, that also guarantee the existence of at most one completed majority group at any point in time. The main difference between our work and the primary partition approaches published so far is that 1) we assume that partitioning is possible and quite natural, especially in the presence of network congestion or long lasting physical failures and 2) we use a numeric majority criterion to ensure uniqueness of a distinguished group in time instead of assuming an operator designated 'primary' physical partition.

In contrast to primary partition membership services, partitionable membership services allow the existence of multiple, parallel groups at the same point in real-time. Such services allow groups to split into smaller groups as a consequence of failure occurrences and also enable smaller groups to merge into bigger groups when timely communication is restored. Thus, partitionable membership services resemble our 'one-round' and 'three-round with partition detection' membership services.

As pointed out in [ACBMT95], the most widely-cited papers [RB91] and [DMS94] that attempt to give formal specifications for the primary partition and partitionable asynchronous membership services, as well as their updated versions [Ric93, RB95, DMS95], contain specification faults: [RB91, Ric93, RB95] have flaws in their logical formalism and allow undesirable executions while the specification of [DMS94, DMS95] can be satisfied by trivial, useless protocols. [ACBMT95] also shows that the protocols of [RB95] allow certain executions that violate the safety and liveness requirements specified by the authors. The existence of such faults in widely cited research articles should be taken as an indication of how difficult the problem of specifying and implementing asynchronous membership services really is. We hope our specifications and protocols do not suffer from such shortcomings, partly because we assume a *timed* asynchronous system model. As pointed out in [CHTCB96] no useful membership service can be implemented in the time-free asynchronous system model, since a subproblem common to all such services cannot provably be solved in the time-free model.

3 Timed Asynchronous System Model

We consider a system consisting of a set of processors P linked by a datagram communication service. The specifications of the processor and communication services are *timed*: they prescribe not only what state transitions/outputs should occur in response to input events, such as clock ticks or message arrivals, but also the real-time intervals within which these state transitions/outputs should occur.

The processors do not share any storage: they gain information only by sending and receiving messages and by measuring the passage of time. We assume that all processors have access to private hardware clocks whose drift rate is *bounded*. For current technologies, this bound is of the order of 10^{-5} or 10^{-6} seconds per second. Since the errors introduced by such small drift in measuring the timeout durations discussed in this paper are insignificant, we will ignore them for simplicity. We assume a total order "$<$" on the set of processor names P. We let p, q, r, ... denote processors in P. We also assume that each processor has access to a private *stable storage* service. Processors have *crash/performance* failure semantics [Cri91c]. Performance failures include a processor being slow, as well as its clock running too slow or too fast. Crashes are partial-amnesia crashes: at restart, the state of stable storage is the same as before the crash, while the state of the volatile storage is reinitialized. In what follows, when we say that a processor fails we mean that it suffers either a crash or a performance failure. A correct processor is non-crashed and timely.

We do not make any assumption about the physical network topology. It can be an arbitrary point-to-point topology or a broadcast topology, a local area network or a wide area network. We assume the existence of a *datagram* communication service that allows any processor to send messages to any other processor. Message routing and forwarding is performed below the datagram service layer if the underlying physical network is not fully connected. Error detecting codes are used to detect message corruption, so the datagram service has *omission/performance* failure semantics. Thus, when we say that an (unqualified) communication failure occurs, we mean an omission or a performance failure. Omission failures occur mostly because of physical message corruption or buffer overflows, while performance failures occur whenever the actual delay experienced by a message exceeds a *timeout* delay δ specified for the datagram service. We assume the (one way) timeout value δ is an *end-to-end* delay, i.e., includes operating system overhead on the sending and on the receiving processor, including interrupt and scheduling delays, as well as message processing delays. Furthermore, the delay δ should be chosen so as to minimize the likelihood of "false" communication failure detections to an acceptably low risk level. To reduce the probability of "false" alarms, it is common practice to re-send a message that was timed out a finite number $k \geq 0$ of times before giving up. The number k of retries is most often determined empirically, so that, if after k attempts, a message is not known to have arrived at its destination, then sending it more than k times will not significantly increase the likelihood that it can be sent successfully. In what follows, we assume that the timeout delay δ used by our membership protocols is large enough to accommodate any such (lower level) retries. Since re-sending messages several times before giving up does not prevent them from being lost or late at the datagram service interface, we will not mention such lower level retries any more.

We call a system of processors and communication services that satisfies the hypotheses listed above a *timed asynchronous* system. Most existing distributed systems based on non-real-time operating systems and communication services, such as Unix and UDP, are timed asynchronous. In previous publications (e.g. [Cri89, Cri91c, Cri91a, CM94]) we called such systems simply asynchronous, as opposed to the synchronous system model investigated in [CASD85, Cri91b, Cri93]. This has unfortunately caused some confusion, since other authors (e.g. [FLP85]) use the term 'asynchronous system' with a different meaning. The difference comes from the fact that the processor and communication services of interest to us are *timed*, while the processor and communication services investigated in [FLP85] are *time-free*. Indeed, the service specifications considered in [FLP85] are independent of any notion of time: they specify, for each state and input only the next state and output to be produced *without imposing* any constraint on the real-time it will take that state transition and output to occur. Thus, a time-free processor is by definition "correct" even when it would take it an arbitrary amount of time (e.g. days, centuries) to react to an actual input. This very weak definition of correctness makes it impossible to distinguish between crashed and merely slow, but otherwise "correct" processors. Thus, in a time-free asynchronous system, it does not make sense to want to detect crash failures and one cannot logically talk about performance failures. In order to be able to detect such commonly observable failures, practical system services are timed[1]. Since in this paper (as in previous papers)

[1]While it is true that many of the services encountered in practice have incomplete and vague specifications

we are interested in timed systems, we will refer to timed asynchronous systems simply as asynchronous in what follows.

Due to the low processor and communication failure rates achievable with current technologies, it is reasonable to expect that the correct processors of an asynchronous system can achieve timely communication *most of the time*. However, due to congestion and other adverse phenomena, processors may become temporarily disconnected. For our model we assume that at any point in time, the communication between two processors p and q can be in one of three abstract states: *connected*, *disconnected*, or *partially connected*. These abstract states are not directly observable from within the system, but imply the following about the correctness of p and q and on messages sent between p and q: If p and q are *connected* throughout a time interval I=$[t, t']$, then p and q are correct (i.e. non-crashed and timely) throughout I, and every message sent by p or q in $[t, t'-\delta]$ to the other is delivered and processed within δ time units. If p and q are *disconnected* throughout a time interval I, then p is crashed throughout I, or q is crashed throughout I, or no messages sent between them in I are delivered in I. If p and q are *partially connected* throughout I, no conclusions can be made about processor correctness or message delivery in I. These abstract states are being introduced to allow a person who reasons about protocol properties in I to *predict* what happens if messages are actually being sent between p and q in I, rather than to allow p or q to *evaluate* what their communication mode is at a certain instant (such evaluation is impossible because processors cannot predict the future).

We call a distributed system *stable* in a time interval I=$[t, t']$ if, throughout I: (1) no processor fails or recovers, (2) all messages delivered in I are delivered within δ time units of their sending, (3) all pairs of processors in P are either connected or disconnected, and (4) the connected relation between processors is constant and transitive. If we call a set of processors that are pairwise connected a *physical partition*, a stable system will consist of one or more disjoint physical partitions. Because of the low processor and communication failure (and hence recovery) rates likely to be observed in practice, a well-tuned asynchronous system is likely to alternate between long stability periods and comparatively short instability intervals.

The above stability condition is similar to the global stabilization requirement of [DLS88] that was proven sufficient for solving consensus. While global stabilization requires that, beyond a certain global system stabilization time, a system should never experience omission/performance communication failures our stability condition is a weaker property, in that it assumes the absence of such failures only between certain pairs of processors for finite time intervals. Another way to add “synchrony” to the time-free system is to augment it with failure detectors [CHT96]. We find the timed model difficult to compare with

that often do not explicitly mention response-time promises, it is also true that all such services become “timed” whenever some higher level of abstraction that depends on them, in the worst case the human user, fixes a timeout delay for deciding of their failure.

the time-free model augmented with failure detectors. For example, one open question is which failure detectors are implementable in the timed model and which are not. So far, we have proven that a Perfect failure detector is not implementable in a timed system [FC96]. Another difference between the two models is more of system design philosophy. Failure detectors aim at hiding from higher levels of abstraction the time-related aspects of a fault-tolerant distributed computation, such as the detection of failures, so that these higher levels do not have to use timeouts. The idea is certainly appealing at first glance, but we do not believe it works for practical systems which have (usually) more than two levels of abstraction. Indeed, all the multi-layer systems we know employ timeouts at *each* abstraction level, because a level that *depends* [Cri91c] on another level must be able to detect and mask by its own means the possible failures of the lower level. In general the timeouts used at different levels are different: the higher the level the larger the timeout. For example, a connection service such as TCP/IP, might use timeouts of the orders of minutes to retransmit unacknowledged messages, while a mailing service that depends on the connection service typically uses timeouts of the orders of hours to re-transmit unacknowledged mail. The timed model, which allows timers to be used at any abstraction level, of course allows this kind of programming to be done naturally. Such programming is not possible in the time-free model augmented with failure detectors, since timers are unavailable above the failure detector layer.

Measurements done at the Dependable Systems Lab at UCSD [CF97] confirm that the timed asynchronous system model introduced here captures acuratelly what is going on in existing distributed systems built from networked workstations.

4 History-dependent Approach to Communication

To implement highly available distributed services, processors are structured into teams. By a processor *team*, we mean the set of all processors capable of implementing a certain service. For example, to implement a highly available file service, one might want to use three replicas of a file that reside on three distinct processors; these processors are structured as a three-member team. A service provided by servers residing on the processors of a team is inherently distributed if communication via messages among correct team members is an essential prerequisite for the provision of the service. For example, our replicated file service is distributed since a majority of team members is required for performing any update; when communication between a replica manager and the other two managers is impossible, that manager cannot provide any update service.

There exist in principle two approaches to managing communication among the team members that implement a distributed service. In the first, *history-independent* approach, when a team member p interprets a service request that requires it to communicate with the other members, p attempts to communicate with these members irrespective of the success it has had in the recent past at communicating with them. A well known example of a history independent approach is voting [Gif79]. To service an update request in a replicated implementation based on voting, p will attempt to communicate with both q and r, even when its last n consecutive attempts at communicating with q or r were unsuccessful. In the second, *history-dependent* approach, team members *adapt* to changing communication conditions by attempting to communicate only with the group of team members with which they could communicate successfully in the recent past. The idea is to let team members that have a recent history of successful communication form dynamic *communication and working groups*. When communication with a processor is lost, the processor is eliminated from the current group, and when communication is re-established, the processor is again admitted in a new current group. To provide their service, team members obey the rule of limiting communication and cooperation to members of their current group only. The group-based communication approach avoids useless attempts at communicating with members that are shut down, failed or disconnected for long time intervals. For example, in the replicated file service case, suppose that, because of successful communication in the recent past, p and q are members of a group g that does not contain r, and p receives an update request. In the history-dependent approach, p will only attempt to communicate with q to perform the update, thereby saving the cost of sending a message to r and waiting for a timeout expiration for that message. Thus, in general, group-communication based service implementations are characterized by lower message traffic and better response times. The main complication introduced by the history-dependent approach is the need for a *group membership* service. In addition to improving performance, such a service can achieve agreement on the history of groups that exist over time as well as on the membership of each group [Cri91b]. This allows users to order all failure and recoveries that occur in time, and contributes to simplify the programming of many higher level applications (see for example the synchronous and asynchronous availability management services described in [Cri93] and [CM94], respectively). In the remainder of this paper, we propose several possible group membership service specifications for asynchronous distributed systems and comment on their adequacy in being used by various higher level distributed services.

5 Detecting System Changes

A group membership management protocol can be thought of as consisting of two components: a protocol for detecting changes in the system (failures and recoveries) and a protocol for reporting this information consistently to correct processors. The various membership protocols we propose in this paper differ in how the membership information is updated after a failure or recovery, but all use essentially the same techniques for detecting such events. In this section we discuss methods for detecting failures and recoveries.

5.1 Detecting Failures

What makes fault-tolerant, asynchronous protocols especially difficult to design and understand is the fact that, in a (timed) asynchronous distributed system, it is impossible for a processor p to distinguish between the crash of another processor q, a communication failure that causes p and q to be disconnected for a long period of time, and a transient communication failure that causes a message (or a small number of messages) between p and q to be lost or late. In all three cases p simply observes that it cannot communicate with q within 2δ time units. Therefore, a minimum that a membership protocol should provide is

> If processor p is disconnected from processor q, and the system is stable for a sufficiently long time interval, and p is correct during this interval, then p will detect that it is disconnected from q.

During periods in which the system is unstable, a correct processor p will not be able to distinguish between scenarios such as: a) q has crashed, b) q suffers a performance failure, c) all messages from p to q get lost or are late, d) all messages from q to p get lost or delayed, even though q is correct and may even receive all messages from p. Since the main goal of using server groups is *to mask* processor failures [Cri91c], when p cannot communicate with q it will have to adopt the only decision that will allow it to mask a *real* failure of q, which is to proceed as if q had failed. Therefore, p will decide that q has failed if p expected to receive a message from q within some bounded time interval, but did in fact not receive the message by the end of that interval.

Let us first consider a system with only two processors p and q. If we want to ensure that p detects any *real failure* of q within a bounded time after p has become disconnected from q, it is sufficient for the two processors to periodically exchange messages. There are two ways of doing this:

1. Processor p could send an “Are-you-alive?” message to q within π time units after it learns that q is alive. A failure of q would lead to the absence of an answer to such a message within 2δ time units.
2. Processor q could send an “I-am-alive” message to p at least every π time units. A failure of q would result in p not receiving a new “I-am-alive” message from q within $\pi+\delta$ time units after receiving the last message from q.

The second approach requires fewer messages and leads to faster failure detection. We used this approach in all our earlier synchronous membership protocols [Cri91b] and we will also use it in all the protocols of this paper.

Now consider a system with n>2 processors. If every processor sent every other processor an "I-am-alive" message every π time units, there would be a total of $n(n-1)$ messages sent every π time units. This message traffic can be reduced by observing that it is not necessary for all processors to independently detect the failure of another processor. The most message-economical way of doing this is to use either an *attendance list* or a *neighbor surveillance* protocol [Cri91b].

The attendance list protocol, which we adopt for all (except the first) algorithms of this paper, works as follows. Assume the system is currently stable and consider a physical partition S with more than one processor. One can define a cyclic order that assigns a left and right neighbor to each processor in S by using the total order assumed on processor identifiers (or *ids*) as follows: the left and right neighbor of processor p are the processors with the next lower and higher id, respectively, in S (a special case are the processors with the lowest and highest id in S, which are defined to be right and left neighbors of each other). To check whether a failure of a processor in S occurs, all processors in S agree that one of them, say the one with the lowest identifier, will initiate the circulation of an "attendance list" message every π time units. If the message is relayed by all processors in S in the right neighbor cyclic order described above and arrives back at its originator in time, the processors in S know that none of them has failed. If any processor in S detects that the attendance list is lost or late, one of them may have failed. This failure detection method reduces the number of "I-am-alive" messages sent every π time units from $n(n-1)$ to n. In [Cri91b] this is proven to be the minimum number of messages that must be exchanged to detect member failures in a group with n members.

5.2 Detecting Recoveries

Detecting recoveries is simpler than detecting failures. A processor that restarts after a crash can simply send a message to all other processors letting them know about the event. Since some of the messages can get lost either because of transient communication failures or because the processors are disconnected for longer periods of time, such "probe" messages must in fact be sent periodically. In general, it is sufficient that one processor in each group (for example the one with the lowest id) periodically send probe messages. One further optimization is to limit the probe sending to minority groups only, i.e., groups containing no more than half of all processors as members. This will have the advantage that under normal situations when all processors are members of the same (majority) group no probes are sent at all[2]. This is the recovery detection protocol we will use in all algorithms presented in this paper. We will denote μ the period with which probe messages are sent.

[2]Majority and minority need not necessarily be defined as a simple numerical majority. The only requirement is that two majority groups, however defined, must always have at least one member in common.

Notice that the mechanisms for detecting failures and recoveries are not independent of the rest of the membership protocol, since they depend on some form of agreement on the current membership. For example, the attendance list protocol assumes agreement on which processors are neighbors (otherwise false failures may be detected), and the optimizations for sending probe messages depend on an agreement on which processor is responsible for sending such messages.

6 Independent Assessment Membership Protocol

As a minimum requirement, a membership service should let each processor p know, once the system stabilizes, which other processors are up and connected to p and which ones are not. We denote by

members: P $\rightarrow$ Set-of-P

the mapping that records each processor's local view of the current processor membership: members(p) yields the set of processors that p believes are correct and connected to it[3]. A membership service should at least satisfy the following timeliness requirements:

(Tjd) *Bounded Join Delays.* There exists a time constant JD such that, if two processors p and q are connected in I=[t,t+JD] and the system P is stable in I, then p and q are included in members(q) and members(p), respectively, sometime in I.

(Tpd) *Bounded Partition Detection Delays.* There exists a time constant PD such that, if two processors p and q are disconnected in I=[t,t+PD], the system P is stable in I, and p stays correct throughout I, then q is excluded from members(p) sometime in I.

The main difference between the above timeliness properties and those required from the synchronous membership protocols described in [Cri91b] is that synchronous protocols are required to detect failures and joins within a bounded delay *independently* of whether the system P is stable or not. In addition, a membership service should not report any "false" failures or recoveries. By this we mean that when the system has become stable and enough time has elapsed for the protocol to report any previous failures or recoveries (max(JD,PD)), no further failures or recoveries should be reported as long as the system remains stable. This is expressed in the following safety requirement:

(Sld) *Stability of Local Views.* If the system is stable in I=[t,t'], then for every correct processor p, members(p) will not change in [t+max(JD,PD),t'].

[3]The notation Set-of-P stands for the set of all subsets of the set P

Since we assume that the "connected" relationship is transitive when the system is stable, during periods of stability the set of processors P can be divided into disjoint subsets or *physical partitions* P_1, P_2, ..., such that two processors p and q are connected if and only if they belong to the same physical partition. This allows us to express the three requirements stated above in the following equivalent, but simpler form:

(Tcd) *View Convergence.* There exists a time constant C such that, if the system is stable in I=[t,t'], then for every correct processor p, members(p) = P_i throughout the interval [t+C,t'], where P_i is the physical partition to which p belongs in I.

It is straight forward to see that (Tcd) implies (Sld), (Tjd), and (Tpd) if we choose JD = PD = C, because P_i contains all processors connected to p (implies (Tjd)) and none of the processors disconnected from p (implies (Tpd)). Conversely, if we choose C=2*max(JD,PD), and apply (Tjd) and (Tpd) in the interval I'=[t+C/2, t+C], it follows that there must be a time t_1 ∈I' such that $P_i \subseteq$ members(p) (follows from (Tjd)) and a time t_2 ∈I' such that members(p) $\subseteq P_i$ (follows from (Tpd)). If we now apply (Sld) in the interval I, it follows that members(p) is constant in I', and therefore members(p) = P_i in I'. In fact, (Sld) implies that members(p) is constant in [t+C/2,t'], and hence members(p) must be equal to P_i throughout [t+C,t'].

The above requirements can be satisfied by the following simple protocol: when a processor p starts, it initializes members(p) to {p} and begins to send "I am alive" messages to every other processor in P every π time units. When a processor q receives a message from p it adds p to members(q); when q measures more than $\pi+\delta$ time units since it last received a message from p, it subtracts p from members(q). In other words, the "I am alive" messages play a double role of probe messages for detecting recoveries as well as "I-am-alive" messages for detecting failures. We call this protocol the "independent assessment" protocol because each processor q performs periodically an *independent assessment* of whether the other processors p are correct and connected to it or not.

The protocol satisfies requirements (Tjd) and (Tpd). To determine the JD delay, consider two connected processors p and q. After the system stabilizes, there can be at most π time units until p and q send their periodic "I am alive" messages, and at most δ time units until these messages are received by p and q, respectively. Thus, JD=$\pi+\delta$. Consider now the determination of the partition detection delay PD. The worst case scenario begins with a crash of p immediately after it sends an "I am alive" message and the system has stabilized. If q is a correct processor such that p $\in$ members(q), the "I am alive" message from p to q may take as long as δ to arrive at q. From that moment, q will monitor whether it receives another "I am alive" message from p within $\pi+\delta$ time units. It follows that it might take q as many as PD=$\pi+2\delta$ time units to exclude p from members(q) after p crashes. Thus, max(JD,PD) time units after the system stabilizes, any two connected processors p and q reach agreement on the identity of all processors connected to them, in the sense that the variables members(p) and members(q) will be equal and will contain the identity of all processors connected to p or q (recall that the "connected" relation is transitive during stability periods; in store and forward networks, this corresponds to a stabilization of the routing tables to a correct state after the network was perturbed by a failure or recovery). If the system remains stable a correct processors p will continue receiving "I am alive" messages from every processor it is connected to and will not receive any messages from processors it is disconnected from. Therefore, max(JD,PD) time units after the system stabilizes members(p) will not change as long as the system remains stable. Hence requirement (Sld) is satisfied.

The problem with the above protocol is its behavior during periods of system instability. During such periods, different processors can disagree on which processors are connected to them and which are not. For example, assume that p and q are two availability managers [CM94] responsible for enforcing a primary/back-up availability policy for a group of two database managers {d,d'} running on the same processors as p and q, respectively. Consider a stable state in which d plays the role of primary and d' plays the role of back-up, followed by a period of instability in which the "I am alive" message from p arrives late at q. Upon detecting what to q looks like a failure of p, q will promote d' to the rank of primary. This will violate the safety condition characteristic of the primary/back-up availability policy, namely that at no point in real-time there should be two primaries. It is precisely because the Tandem membership protocol described in [Car85] is an independent assessment protocol, that it is subject to the "split brain behavior" described above.

7 The One Round Protocol

In a system in which message delays are random, processor and communication failures occur at random times unknown in advance, and clocks can drift from real-time at actual rates that are unknown to processors, it is not realistic to require that connected processors agree on membership at the same real-time, since exact synchronization between processors is precluded [DHS86].

Even though such agreement is impossible, some form of agreement on which processors are up and connected is required for programming applications such as the availability manager example discussed previously. Therefore, the notion of a processor *group* is introduced as a means of grouping together those active processors that could communicate successfully in the recent past and agree to limit further communication to group members only. Instead of aiming at achieving the impossible goal of simultaneous agreement in real-time, we only require that processors that *join* the same group (possibly at different real-times) agree on the identity of the members of the group. At any point in real-time, we allow a processor to be joined to at most one group. To unambiguously designate the different processor-groups that exist in time, we will require a group membership protocol to identify all groups by a unique *processor-group identifier* g. We let G denote the set of all possible processor-group identifiers provided by the protocol. Furthermore, we will require a group membership protocol to define a total order "$<$" on G.

There exist times at which a processor might not be joined to any group, for example, between the moment it leaves a group and the moment it joins its next group. We denote by

joined: P $\rightarrow$ {true, false}

the (total) predicate that, for any processor p, is true when p is joined to some group and is false when p is not joined to any group.[4] A processor not joined to a group cannot take part in any coordinated group activity. We denote

group: P $\rightarrow$ G

the (partial) mapping that records the group to which a processor is joined, when it is joined, that is, if joined(p) is true, then group(p) yields the identifier of the group joined by p. As in the previous section,

members: P $\rightarrow$ Set-of-P

denotes the (partial) mapping that records a processor's view of the membership of the group to which it is joined, when it is joined, i.e., if joined(p) is true, then members(p) yields p's view of the membership of group(p). The value of the above mappings varies in fact with time, that is, they are higher order mappings from Time to P to Set-of-P, G, or {true, false}. However, to keep our presentation simple we will not mention the time domain explicitly in what follows. We say that "processor p *is joined* to group g" if joined(p)=true and group(p)=g. We say that "p *joins* g at t_1" and "p *leaves* g at t_2" if t_1 and t_2 are the earliest and latest real times, respectively, at which p is joined to g.

7.1 Requirements

To avoid the kind of behaviors that can be exhibited by our first protocol during instability periods, we require a group membership service to satisfy at least the following safety properties at all times:

[4] In our first group membership protocol, described in this section, a processor is always joined to some group, i.e., joined(p) is always true. This is not the case for our other protocols, however. We introduce the joined predicate here so we can use the same terminology throughout the rest of this paper.

(Sa) *Agreement on group membership.* If two processors p and q are joined to the same group g (not necessarily at the same time), then the two processors have the same view of the membership of that group: If at any time t, joined(p), group(p)=g, and members(p)=m, and at time t', joined(q), group(q)=g', and members(q)=m', then g=g' implies m=m'.

(Sr) *Reflexivity.* For any processor p, if joined(p) then p $\in$ members(p).

(Sm) *Monotonicity of groups.* Let g and g' be groups that processor p joins at times t and t', respectively. If t' < t then g' < g.

The first requirement implies that a group id uniquely determines the membership of a group. We will therefore often simply write members(g), for g $\in$ G, when we mean members(p) for some p that is a member of g. The second requirement says that a processor can only join groups it is a member of. The third requirement demands that groups are joined in a temporal order consistent with the order on group ids. As a consequence, two processors that join the same two groups g and g' will join them *in the same order.* As we will explain at the end of this section, requirement (Sm) is crucial to make a membership service useful for applications like an availability manager.

The following three requirements are analogous to the (Tjd), (Tpd), and (Sld) properties introduced for the Independent Assessment protocol:

(Tj) *Bounded Join Delays.* There exists a time constant J such that, if two processors p and q are connected in I=[t,t+J] and the system P is stable in I, then there must be a common group g, such that p is joined to g at some time in I and q is joined to g at some time in I (not necessarily at the same time as p).

(Td) *Bounded Partition Detection Delays.* There exists a time constant D such that, if two processors p and q are disconnected in I=[t,t+D], the system P is stable in I, and p stays correct throughout I, then there must be a group g with q $\notin$ members(g), such that p is joined to g at some time in I.

(Sl) *Stability of Local Views.* If the system is stable in I=[t,t'], then no processor will leave its group in [t+max(J,D),t'].

As in the case of the Independent Assessment protocol, these three requirements can also be expressed in the following equivalent, but simpler form:

(Tc) *View Convergence.* There exists a time constant C such that, if the system is stable in I=[t,t'], then all correct processors in each physical partition P_i, will be joined to the same group g_i throughout the interval [t+C,t'], and the membership of g_i, members(g_i), will be equal to P_i.

Finally, consider a group g such that p, q $\in$ members(g) that is proposed in response to instability detection by a processor. If the system is not stable, it is possible that p joins the proposed group g, but q never does (e.g., because a message to q is lost, or because an additional failure causes a new group g' to be created, which q joins instead of g). We will call g incomplete. In general, we say that a group g is *incomplete* if at least one processor q $\in$ members(g) never joins g. Incomplete groups are to be avoided, since no useful work (which requires the cooperation of all members) can be done in them. Requirements (Tj) and (Td) ensure that there will be no incomplete groups if the system remains stable for at least max(J,D) time units. We also require that there be a bound on the time a correct processor can be joined to an incomplete group *even when the system is unstable*:

(Ti) *Bounded Inconsistency Detection Delays.* There exists a time constant D' such that, if at time t, p is joined to a group g such that q $\in$ members(g), p stays correct in I=[t,t+D'], and at no time in I q joins g, then p leaves g by t+D'.

This requirement also implies that when a processor q that has joined a group g leaves g (for example because it detects system instability), all other correct members of g will leave the group no more than D' time units later. Thus, the time that a correct processor p believes it is joined to the same group g as processor q, when in fact q is not joined to g, is bounded by D'.

One can define a somewhat stronger version of requirement (Ti) that requires processor p in the example above to leave the common group g, not only if q does not join the group in a timely fashion, but also any time p and q are disconnected for a sufficiently long time:

(Ti') *Fail Awareness.* There exists a time constant D" such that, if at time t, p is joined to a group g such that q $\in$ members(g), p stays correct in I=[t,t+D"], and p and q are disconnected in I, then p leaves g by t+D".

Unfortunately, satisfying (Ti') would require exchanging I-am-alive messages between every pair of processors. This is because communication may not be transitive during instability. In particular, it is possible that all processors are correct and connected, except for p and q, which are correct but disconnected. Hence, unless p and q exchange messages, a membership protocol will not be able to detect that p and q are disconnected. Since the cost of regularly sending n^2 messages is prohibitive for reasonable large n, we do *not* include (Ti') in our formal requirements for a processor membership protocol. Note that during periods in which the system is stable, (Ti) and (Td) imply (Ti'). Since we assume that a system will be stable most of the time, dropping requirement (Ti') in favor of (Ti) is not a severe limitation.

In practice, a property close to (Ti') can be achieved if the communication subsystem includes an end-to-end acknowledgment protocol and has the ability to signal the membership protocol when messages are not acknowledged in a timely fashion. The fact that two processors are disconnected could then be detected by the membership protocol soon after the next message is sent between the two. For practical purposes such a protocol is a equivalent to one that would use n^2 messages to satisfy (Ti'), because the abstract state of being disconnected is not observable from within the system when no messages are sent.

7.2 The Protocol

We now describe the intuition behind the "One Round" protocol that we propose to satisfy requirements (Sa), (Sr), (Sm), (Sl) and (Tj), (Td), (Ti) introduced in the previous section.

Let us first describe the action that a processor p joined to group g will take in response to the occurrence of a single failure or recovery event, i.e., when another processor fails or recovers, or when p becomes disconnect from connected to another processor. When p detects the such an event through a timeout or the arrival of a probe message, instead of updating only its local membership view, as in the independent assessment protocol, p will propose the creation of a new group with id g', will join g', and will broadcast g' and its membership, members(g'), in a "new group" message to all processors, so they can also join g'. When another processor, say q, joins g', it sets members(q) to members(g'). Thus, property (Sa) will be satisfied, provided that the group identifier g' chosen by p is different from all other group ids generated by p or any other processor. In order to ensure global uniqueness, group identifiers are composed of an integer sequence number (chosen by the processor that created the group), and a processor identifier (the id of group creator). If g.n and g.p denote the sequence number and processor id fields of a group id g, we can define a total order on group ids as follows:

$$g < g' \Leftrightarrow (\, g.n < g'.n \,) \text{ or } (\, g.n = g'.n \text{ and } g.p < g'.p \,)$$

Processor p constructs the new group id g' from the id g of the group it is currently joined to by incrementing the group sequence number and appending its own processor id: g'=(g.n+1, p). This ensures that the monotonicity requirement (Sm) is satisfied (i.e. g'>g), and that g' differs from any group id that another processor might generate.

Since more than one processor may detect a failure or a recovery, a processor q may receive several "new group" proposals at about the same time. Therefore, a way is needed for q to decide which of these competing group proposals it should join. We adopt the rule of letting q join a new group g' only if g' is more "recent" than the id of its current group g", that is, g"<g'. Such a behavior will satisfy requirement (Sm), but it will also ensure that when more than one "new group" message is broadcast, all processors will eventually 'converge' and join a group with the same id (assuming no additional failures or recoveries occur).

The protocol behavior is somewhat more complex when multiple failures and recoveries occur. For example, a processor p joined to a group g may receive a "new group" proposal for g' from a processor q that p knows was previously unreachable, i.e., q$\notin$members(p). If g' $<$ g, p cannot join the new group, but it would be a mistake if p simply ignored the information in the "new group" message, since it would then have evidence that q *is* reachable. Furthermore, q's message may contain information about other failure or recovery events detected by q or by any *other* processor r that has communicated with q recently. Therefore, p needs a way to extract any 'new' information that may be contained in q's "new group" message. To indicate how recent the information included in a "new group" message is, we adopt the solution of requiring its originator q to include in the message a set of *timestamps*, one per processor r $\in$ P, indicating how recent q's information about the status of r is. Whenever a processor sends out a message, it includes its local timestamp vector in the message. Every processor maintains a timestamp vector containing the most recent timestamp known to have been sent by each processor. The vector is updated each time an "I-am-alive" message is received or a "new group" message is received, as explained below. Every "new group" message includes not only its sender's timestamp, but the complete timestamp vector representing the last message known to have been sent by each processor.

The rule for processing "new group" messages is now as follows. Let p be a processor joined to group g with membership members(g) and having timestamp vector s: P$\rightarrow$Time when it receives from processor q a "new group" message carrying a proposed group g' with membership members(g') and a timestamp vector s': P$\rightarrow$Time. Processor p will combine its own knowledge of the system state with the information coming from q by using the following *state merge/group join rules*: For each processor r $\in$ P, p examines whether its own view of r's state (up or down), as represented by members(g), agrees with the r's state according to q, as represented by members(g'). If p and q disagree about the state of a processor r, p further examines the timestamp vector values s(r) and s'(r) to determine whether p or q has more recent information about r. If g' $>$ g and q has more recent information in all cases where p and q disagree, then p joins g'. If g' $<$ g and p has more recent information in all cases where p and q disagree, then p simply ignores g' and stays joined to g. In all other cases p creates a new group with an id g" greater than g and g' and a membership constructed by including exactly those processors that are up according to the most recent information from p and q. It then joins g" and broadcasts the corresponding "new group" message to other processors. In all the above cases, p updates its knowledge of the latest "I-am-alive" timestamp it knows for all processors r in P by assigning to s(r) the maximum of the values of s(r) and s'(r).

We can now describe the One Round protocol in its entirety. The detailed code for the protocol and its correctness proof are given in Appendix A and B.

When a processor p starts for the first time, it joins the singleton group (0,p) with membership p; the group id is stored in stable storage. When it restarts after a failure, it joins a similar singleton group with id (g.n+1, p), where g is last group id stored in stable storage before the failure. At each start, p initializes the timestamps s(r) of all other processors r to 0.

After joining a group, p follows the procedures described in Section 5 for detecting failure and recovery events: Whenever p is joined to a group with more than one member, if p is the *group leader*, that is, the processor with the lowest id in the group, it issues an attendance list message every π time units and monitors its timely return. Other processors that join the group just monitor the timely periodic passage of the attendance list message. If p is the leader of a minority group g (see Section 5.2), it also sends "probe" messages every μ time units to P$-$members(g). Each probe message includes the current group id g of p, members(g), and p's view of the mapping s$\in$P$\rightarrow$Nat that records for each r$\in$P p's knowledge of the latest timestamp s(r) sent by r.

While joined to a group in state (g,members(g),s), a processor p can detect one of the following events: its left neighbor has probably failed, a "new-group" message arrives or a "probe" message arrives. If p detects a possible failure of its left neighbor q, it creates a new group g'=(g.n+1,p) and sends a "new-group" message with arguments (g',members(g'),s') to all processors, where members(g')=members(g)$-$q and $\forall$r$\in$P: s'(r) = s(r). When p receives a "new-group" message with arguments (g',members(g'),s') it follows the state-merge rules described above. In order to reduce the number of messages involved in processing "probe" messages, only the group leader respond to such messages: when the leader receives a "probe" message it is processed in the same way as a "new-group" message; other group members ignore probe messages.

7.3 Advantages and Drawbacks of the One Round Protocol

The One Round protocol is very efficient in handling the case of a single processor failure or recovery: one processor is likely to detect the failure or recovery, create a new group, broadcast a "new group" message, and all members will join the new group when they receive that message. Hence a single processor failure or recovery results in a new group creation with a single round of n messages in a point-to-point network. As we will show in the next section, the protocol is also useful in implementing a service that is often needed to avoid the "split brain" behavior mentioned earlier: the *highly available leadership* service [Cri91b].

Despite its efficiency in handling single failures and recoveries, its possible use in avoiding the "split brain" behavior, and its power to solve the highly available leadership problem, the One Round Protocol has the following major disadvantages.

After multiple failures, recoveries, or message losses, the protocol may take a long time to stabilize. This is reflected in the relatively large value of the D constant for this protocol:

$D = \max(\pi + n\,\delta, \mu) + 5\delta + (n - 1)(\pi + (n{+}1)\delta)$ (see Appendix B). Notice that the value of D contains a term of order $n^2\delta$.

Furthermore, the membership and timestamp vector information included in "new-group" messages does not always allow a processor that stays correct to find out about *all* failures and recoveries that happen in a system. This may be a particularly vexing problem if the goal is to build an availability management service [CM94], which must enforce its specified service availability policies for all failures and recoveries that occur in the system. For example, if processor p_4 of group g such that members(g) = $\{p_1, p_2, p_3, p_4, p_5\}$ fails and then recovers quickly while processor p_5, which had crashed earlier, recovers, it is possible that processor p_1 will successively join the groups g' and g" such that members(g') = $\{p_1, p_2, p_3, p_4\}$ and members(g") = $\{p_1, p_2, p_3, p_4, p_5\}$. Thus, p_1 would be unaware that p_4 had failed and recovered.

The protocols presented in subsequent sections of this paper will address these issues.

8 The Highly Available Leadership Problem

As mentioned above, the One Round Protocol is also useful in implementing a *highly available leadership* service [Cri91b]. We can adapt the requirements proposed for this service in [Cri91b] to an asynchronous environment as follows.

(Su) *Uniqueness of Leader.* At any point in time there is at most one leader.

(Te) *Bounded Election Delays.* There exists a time constant E such that, if the system P is stable in I=[t,t+E] and a majority of processors is connected then there will be a leader by t+E.

In a synchronous system, a highly available leadership protocol can be derived in a straightforward way from a membership protocol, by adding a rule for choosing a leader from the members of a group, for example, "the leader should be the processor with the highest id among group members" [Cri91b]. Since all processors joined to the same group agree on its membership, they will all agree on who the leader of the group is. In asynchronous systems the problem is more complicated, because different processors leave and join groups at different times, and because, due to communication failures, more than one group can exist at a time. To ensure that there is at most one leader, one could restrict the groups that can produce leaders to be majority groups only. Thus, if there are communication failures and multiple different groups form at the same time, it would seem that at most one of these groups can elect a leader.

Unfortunately, the simple majority group rule suggested above for allowing a leader to exist is not strong enough to guarantee that there are never two processors that believe to be leaders at the same point in real-time. For example, consider the case where an "I-am-alive" message from the current leader p of majority group g is lost. If this leads to the creation of a new majority group g', such that p$\notin$members(g'), the processor p' with highest id in members(g') will be promoted to leader. However, it is possible that at time t when p' joins g' and learns that it must play the role of leader, p is still joined to g. Thus, until p leaves g, p and p' will both believe they are leaders. One might remark that property (Ti) (page 16) limits the time that p stays joined to the old group g to at most D' provided p stays correct in I=[t,t+D']. Thus, if p' were to wait D' time units before becoming leader, it would ensure that p has left g and relinquished its role as leader. However, if p suffers a performance failure, it may take an arbitrary amount of time before p leaves g.

In fact, the leadership problem is unsolvable in the timed asynchronous model unless we make the additional assumption that a processor can detect when it suffers a performance failure: if p becomes disconnected, it is impossible for the other processors to distinguish between p suffering a prolonged performance failure and p being crashed or correct. This makes it impossible for any correct protocol to elect a new leader within a bounded time, even if, in the latter case, the system remains stable after p initially becomes disconnected. It should also be noted that solving the leader election problem, i.e., guaranteeing that no two processors believe to be leader at the same point in real-time, is not very useful, unless the actions that a leader may take are guaranteed to complete within a bounded time. For example, if a leader election protocol is used to control access to a shared storage device, e.g., a dual-ported disk, the operating system must guarantee that an I/O operation to the device will complete within a bounded time. Otherwise, even though a processor may have resigned as a leader, an action it initiated before resigning may interfere with the operation of a newly elected leader (e.g., if an I/O operation initiated by the old leader can be delayed arbitrarily, it may overwrite data written by the new leader).

Given this observation, it is reasonable to assume that in a system in which leader election is useful, a processor can detect when it suffers a performance failure[5]. Under this assumption it would seem that waiting D' time units after a new majority group has formed before appointing a new leader would be safe, because by that time the old leader should either have left the old majority group, suffered a crash, or detected a performance failure. Unfortunately, this is not sufficient to prevent multiple leaders at the same time. The reason for this is that the requirements for a membership service presented so far allow multiple majority groups to be created at roughly the same time. In fact, this can happen in the One Round protocol if two or more messages are lost.

[5]The minimum requirement is that a processor must be able to determine whether it has suffered a performance failure at the time it initiates an action that requires it to be the leader. This could be done, for example by keeping track of the time by which the membership protocol must handle certain events (e.g., the time next I-am-alive message needs to be sent) and checking this time against the current hardware clock time before initiating a leader action. Note that this assumes that the hardware clock is reliable, or at least is never slow.

A waiting period of 2D' before becoming leader, however, is in fact sufficient to guarantee leader uniqueness in real-time. The intuition behind this is as follows. Suppose that two or more failures occur while a processor p is leader for a majority group g, that these failures lead to the parallel creation of majority groups $g_1 < g_2$, and let s be in members(g_1) $\cap$ members(g_2). The first waiting duration D' ensures that the process with highest id in g_1 will not attempt to become leader before p learns that it should step down. The second D' is sufficient for the process with highest id in g_2 to know that any process q who could have been leader in g_1 has learned that g_1 was incomplete because (at least the) process s has left g_1. A rigorous proof that the 2D' waiting period guarantees uniqueness in real-time is given in Appendix B.

From the (Tj) requirement (page 15) and the above waiting rule, it follows that E=J+2D' after a system becomes stable, a leader will exist. In the next section we will show how this bound on the leader election delay can be improved with a memberhip service that satisfies somewhat stronger ordering requirements than the One Round protocol (see page 24).

9 The Three Round Protocol with Partition Detection

One of the reasons why the One Round membership protocol may take a long time to stabilize, is the possibility of spurious timeout occurrences even after the system has become stable. Such timeouts are not caused by current failures, but are a consequence of earlier failures that occurred while the system was unstable. To illustrate such a spurious timeout, consider a processor p_1 joined to group g that expects to receive an "I am alive" attendance list message from its right neighbor p_3 in g. Due to earlier failures, however, p_3 may be joined to a group g' in which its left neighbor is p_2, not p_1. Thus, p_3 will send its "I am alive" messages to p_2 instead of p_1, and p_1 will (incorrectly) conclude that p_3 has crashed or is disconnected, and it will create a new group excluding p_3. The opposite problem — processor p_1 believing p_3 is alive and connected when in fact it is not — is also possible under the attendance list protocol.

Both problems can be avoided if the following rule is obeyed for new group creations: a processor p that attempts to create a new group g must first *check* that the members of g are indeed alive and not disconnected before any of the members of g can join g. Thus, the creation of a new group g becomes a three round process. In the first round, a group creator p first broadcasts a *proposal* to create a new group g. In a second round, all processors that learn of the proposal send their replies. In the third round, p includes in members(g) all processors that have replied to its proposal and sends them a "join" message containing g and members(g). Besides stabilizing faster after multiple failures and recoveries, a multi-round approach has other advantages: as we explain in more detail below, the approach gives faster solutions for the leader election problem and allows processors to detect events they have missed as soon as they join a new group.

9.1 Requirements for Logical Partition Notification

After multiple failures or recoveries a three round membership protocol as outlined above stabilizes faster than the One Round protocol, because a processor is included in (respectively excluded from) a new group, only if the group creator has *direct evidence* that the processor is alive (respectively failed / not connected). To state this property precisely, we introduce the notion of a *predecessor group.* Let pred: G $\times$ P $\rightarrow$ G, be a partial mapping with domain $\{(g,p)|p\in \text{members}(g)\}$ defined as follows:

pred(g,p) = max{ g' | g' < g and p joined g'}.

If p was never joined to a group g' < g, we define pred(g,p) = (0,0).[6]

We call pred(g,p) a *predecessor* group of g with respect to member p. For any p $\in$ members(g), pred(g,p) yields the previous group to which p was joined before g. Note that the definition does not require p to effectively join g, only that p be a member of g.

Given this definition, we will require our three round protocol to satisfy the following safety requirements in addition to those introduced in the previous section:

(Sd) *Legitimacy of processor deletions.* If processor p joins group g at time t such that q $\notin$ members(g) and q was a member of the previous group pred(g,p) joined by p, there must exist a processor r $\in$ members(g) that was not crashed and not connected to q at some time in [t',t], where t' is the time at which r left its predecessor group pred(g,r).

(Sj) *Legitimacy of processor additions.* If processor p joins group g at time t such that q $\in$ members(g) and q was not a member of the previous group pred(g,p) joined by p, there must exist a processor r $\in$ members(g) that was correct and not disconnected from q at some time in [t',t], where t' is the time at which r left its predecessor group pred(g,r).

By requiring a three round protocol to satisfy (Sd) and (Sj), such a protocol should stabilize faster than the One Round protocol. Another reason for introducing these two requirements is explained at the end of this section.

Another advantage of using three rounds for group creation is that it allows sequencing joining and leaving of groups in such a way that all members of a new group g will have

[6]The group id (0,0) is smaller than any "real" group id created during the execution of a membership protocol. It can be thought of as an initial group that every processor is implicitly joined to when it first enters the system.

left any previous group they were joined to before the first member of g joins g. This is expressed in the following requirement:

(Ss) *Join Synchronization.* If processor p joins group g at time t, then no member of g will be joined to a group g' < g after t.

If a membership protocol satisfies this additional property, a highly available leadership protocol with a waiting delay of D' instead of 2D' can be built from that protocol along the lines suggested in the previous section. Indeed, because of the join synchronization property, after D' time units, the leader l of a majority group g knows that any leader that could have existed before l has either left its group or is in g. A more rigorous proof that the D' waiting rule is correct is given in Appendix B. Thus, the Three Round protocol provides a basis for a faster implementation of a highly available leadership service.

Finally, during the proposal phase, the group creator has an opportunity to collect information about the previous history of the new group members. This can be exploited to solve the 'missing events' problem mentioned at the end of the previous section, where we gave an example in which a processor p_1 joins group g' with membership $\{p_1, p_2, p_3, p_4\}$ followed by group g" with membership $\{p_1, p_2, p_3, p_4, p_5\}$, being unaware that in the meantime, processor p_4 had crashed and recovered. Some systems (for example Isis [KB91]) solve this problem by associating an *incarnation number* with every member of a group. This number is incremented each time a processor recovers from a crash. Thus, processor p_1 would observe that p_4's incarnation numbers in g' and g" are different and conclude that p_4 had crashed and recovered. However, this approach does not generalize to the case where processor p_4 instead of crashing, is briefly disconnected from the other processors and joins a singleton group by itself, before becoming a member of the new group g". Incarnation numbers do not allow other members of g" to detect this fact, since p_4's incarnation number will not change from g' to g". Instead of incarnation numbers, we will require a multi-round membership protocol to notify all members of any newly created group g of the predecessor groups of all members of g.

(Sp) *Logical Partition Notification.* Let p be a processor that joins a group g. At join time the membership protocol should let p know the predecessor groups pred(g,q) of all q $\in$ members(g).

With this additional requirement, p_1 will be able to detect that it was logically partitioned from p_4 before joining g", because pred(g",p_4) $\neq$ pred(g",p_1) = g'.

We say of two members p and q of a group g that they were *logically partitioned* before joining g if their predecessor groups are different: pred(g,p) $\neq$ pred(g,q). Logical partitions can occur in asynchronous systems either because of the existence of "hard" physical partitions or simply because some messages get lost or arrive late. For example, in local area networks, it is likely that most logical partitions are caused by transient overloads and message losses. When two processors detect that they were logically partitioned, they might *disagree* on the events that happened in the system since the last time they were in a common group. These events could have lead to diverging group states in the distinct groups g' and g". Thus, when they rejoin a common group g, p and q may agree on a new state s= f(s',s") for g, where f is a "state merge" function, that determines the new group state derived from states s' and s" of g' and g", respectively. When the state of the service implemented by a team is represented as an instance of a data type with commutative operations, f can be as simple as a union between s' and s". For example, if the operations implemented by team members are inserts in a set, f can be the union of the sets s' and s". If the operations are not commutative, for example both inserts and deletes are allowed, then a conflict detection/resolution rule such as those described in [SSCA87] must be considered when defining the merge function f.

Note that for the logical partition notification mechanism to be useful, a membership protocol must create new groups only "when necessary". For example, one could imagine a protocol, in which a processor always first joins a singleton group (containing only itself as a member) when detecting any failure or recovery, before merging into a "real" group (i.e., one that reflects the current "is connected" relation). For such a protocol pred(g,p) would be different from pred(g,q) for all p $\neq$ q. In other words, two processors p and q joining a common group g would always believe they were logically partitioned, making the predecessor information introduced earlier useless. It was to exclude such trivial protocols that we introduced requirements (Sd) and (Sj).

9.2 The Protocol

The One Round protocol will now be changed into a new protocol, the "Three Round Protocol with Partition Detection". In addition to the safety properties (Sa), (Sr), (Sm), (Sl) and timeliness properties (Tj), (Td), and (Ti) (pages 15–16), this protocol also satisfies the safety requirements (Sd), (Sj), (Ss) and (Sp) introduced in the previous section. Like in the One Round protocol, groups are identified by unique ids consisting of a stable sequence number and a processor id. Also, similar to the One Round protocol, failures are detected by using the attendance list protocol and joins are detected via periodic probes. Since a group creator first "checks" the membership of any proposed group, the timestamp vectors and the somewhat cumbersome state merge/group join rule used by the One Round protocol are no longer needed. The detailed code of the protocol is given in Appendix A. Below we sketch only its main features.

When a processor p in a group g detects a failure or recovery, instead of simply sending a "new group" message that orders all members to join the new group g'=(g.n+1,p), p will first send them an invitation to let him know whether they are willing to join g' or not. A processor q that receives such an invitation for group g' will reply with an "accept" message for g' if it does not know of any other attempt at creating a competing group g" such that g">g'. Otherwise, if q knows about a competing group g">g', it forwards the "new group" message for g" to r instead of replying with an "accept" message for g'. Since any processor that attempts to create a group can be in its turn invited to join a competing group by another processor, the above rule for accepting invitations ensures the following property: if several connected processors try to create competing groups, at most one of them will succeed. Moreover, if the system becomes stable, any successful group will be "maximal", in the sense that it will contain all processors connected to the successful group creator. To satisfy requirement (Sp), any "accept" message from a processor q also carries as a parameter the predecessor group that q left when it learned about the new group proposal.

After waiting for 2δ time units, the processor p that attempts to create group g', will go ahead with the creation if it did not learn in the mean time about any attempt to create a competing group with a higher id. In such a case, p computes the membership of the new group g' to be the set of all processors that it knows have accepted to join g', joins g' and then sends to all members of g' a "join" message containing g', members(g') and pred(g'), where pred(g') is the predecessor information gathered from the received "accept" messages. Upon receiving such a message, a processor q joins g', adopts members(g') as its view of the membership of g', and learns pred(g') if g' is still the highest group id it knows about. To handle the situation when the proposer of a new group g' fails, any processor q that accepts to join g' sets a timer for 3δ time units: if this timer expires, q attempts in its turn to create a new group (g'.n+1,q) that will make obvious the failure of the proposer of g' to all connected processors.

9.3 Advantages and Drawbacks

The new properties added for the Three Round Protocol with Partition Detection give it several advantages over the One Round Protocol. Due to (Sd) and (Sj) (page 23), this protocol creates fewer transient groups (i.e., groups that exist for only a short time) and stabilizes faster after multiple failures and recoveries than the One Round protocol. In Appendix B we derive that the maximum join and partition detection time constants for this protocol are $J = D = 9\delta + \max(\pi + (n+3)\delta, \mu)$. Property (Ss) (page 24) leads to a faster solution to the highly available leadership problem. Finally, the partition notification requirement (Sp) (page 24) allows members of a new group to detect inconsistencies among themselves.

For some applications, however, the properties of the Three Round membership protocol are still not strong enough. Consider the problem of implementing an availability management policy along the following lines: when a processor fails, the next available processor takes over the responsibility of the failed one, where "next available processor" is defined as the processor with the lowest id that has not yet taken on additional responsibilities. For example, if processor p_3 fails, p_1 will take over services previously provided by p_3; if later on p_5

fails, processor p_2 will take over for p_5 as the "next in line" after p_1. Notice that this policy depends on p_1 and p_2 agreeing that p_3 failed first before p_5 did. Otherwise, if for example, p_2 believed that p_5 failed before p_3, both p_1 and p_2 would attempt to take over services for p_3, each believing that the other one will handle services for p_5. In an asynchronous system it is impossible to determine exactly at what real time a processor failed or to decide the exact order in which two failures occurred in real time if they happen too close together. Fortunately, for an availability management policy as outlined above it is not necessary to know the order of two failures in real time as long as all processors responsible for enforcing the availability management policy *agree on the same order*. Unfortunately, the properties of the membership services presented so far are not strong enough to solve this problem. The following example illustrates this.

Assume that initially all five processors are joined to a group g_0 with membership $\{p_1, p_2, p_3, p_4, p_5\}$. The Three Round protocol with Partition Detection allows the following behavior to occur.

- Due to a lost "I am alive" message processor p_3 detects a failure and initiates the creation of group g_1 by sending a "new group" message to all other processors. All processors reply by sending an "accept" message back to p_3, but p_5's reply is lost. As a result, p_3 computes the membership of g_1 as $\{p_1, p_2, p_3, p_4\}$ and sends "join" message to the other members of g_1. Due to message loss, only p_2 receives the "join" message, and p_2 and p_3 are therefore the only processors that ever join g_1.
- Assume that p_4 detects the loss of the "join" message before p_1 does and initiates the creation of group $g_1' > g_1$ by sending a "new group" message to all other processors. Again, all processors reply, but this time p_3's "accept message" is lost. As a result, p_4 computes the membership of g_1' as $\{p_1, p_2, p_4, p_5\}$. Again, some of the "join" messages are lost; this time only p_1 receives a join message, so that g_1' is joined only p_1 and p_4.
- Now p_3 and p_5 crash. Processors p_2 detects that the "join" message for g_1' was lost and initiates the creation of a new group $g_2 > g_1'$. This time no messages are lost. As a result the surviving processors all join group g_2 with membership $\{p_1, p_2, p_4\}$.

Now consider the sequence of groups joined by p_1 and p_2 in this example:

p_1: $(g_0,\{p_1, p_2, p_3, p_4, p_5\})$, $(g_1,\{p_1, p_2, p_4, p_5\})$, $(g_2,\{p_1, p_2, p_4\})$

p_2: $(g_0,\{p_1, p_2, p_3, p_4, p_5\})$, $(g_1',\{p_1, p_2, p_3, p_4\})$, $(g_2,\{p_1, p_2, p_4\})$

This protocol behavior, which is consistent with all membership service requirements presented so far, will cause p_1 and p_2 to *disagree* on the order in which the processors p_3 and p_5 have failed: to p_1 it appears that p_3 failed before p_5, whereas to p_2 would believe that

p_5 failed first. Note that this behavior is possible even though p_1 and p_2 were correct and connected all along.

Our requirements, especially (Sm) (page 15), only ensure that for those groups joined by both p_1 and p_2 (i.e., g_0 and g_2) both processors join them in the same order (i.e., g_0 before g_2); they do not guarantee that both processors will join the *same set* of groups even though p_1 and p_2 are both members of all the groups that are created in this example. The partition notification mechanism allows such inconsistencies to be *detected* (in our example, when p_1 and p_2 join g_2: g_1=pred(g_2,p_1) $\neq$ g_1'=pred(g_2,p_2), thus p_1 and p_2 will detect that they were joined to different groups before joining g_1). However, the Three Round Protocol with Partition Detection does not *prevent* the occurrence of such inconsistencies. In the next two sections we will explore membership protocols that address this problem.

10 Majority Group Membership Protocols

To force agreement on the order in which failures and recoveries occur in a team, [Cri91b] has proposed to build protocols that guarantee that any two correct processors will join identical sequences (or histories) of groups. Since processors agree on the membership of each group, if they agree on a unique history, they can easily agree on the differences between the memberships of successive groups of the history. The synchronous membership protocols proposed in [Cri91b] all satisfy this property, which is named *agreement on group histories*. Unfortunately, in an asynchronous system two correct processors cannot always join the same groups, simply because there may be times when they are both correct but disconnected. The example presented at the end of the previous section shows that under the requirements for group membership services presented so far, it is possible that two processors observe different group histories even if they are correct and connected.

In an asynchronous system we can achieve a form of agreement on group histories nearly as strong as in the synchronous case if we consider only *majority group* histories, that is, groups that contain a majority of processors of P as their members (by definition, any two majority groups have intersecting memberships). In this section we present two membership protocols that achieve agreement on majority groups: a five round protocol that achieves agreement on a linear history of majority groups and a cheaper three round protocol that achieves a slightly weaker agreement on a linear history of completed majority groups, where a group is *completed* if it is joined by all its members. Since for many applications, any real work can only be done in completed groups, the cheaper three round protocol is often all that is needed.

10.1 Requirements for Majority Group Membership Protocols

At the end of the previous section an example illustrated how in an execution of the Three Round protocol with Partition Detection, two processors p_1 and p_2 could observe two different group histories leading to contradictory conclusions about the order of the failures of p_3 and p_5 at processors p_1 and p_2. Both histories contained only majority groups and ended in the same final group g_2, but containing different intermediate groups g_1 and g_1'. As was explained, requirement (Sp) (page 24) allows p_1 and p_2 to detect this inconsistency when they join g_2. We can strengthen (Sp) by requiring inconsistencies to be *resolved* by the group creation component of the membership protocol, instead of only being made detectable. What we mean by this is that, when the majority group g_2 is created, the membership protocol should *decide* whether g_1 or g_1' is the "official" majority group that precedes g_2. In this way, the membership protocol would tell p_1 and p_2 whether p_3 failed before p_5 or vice versa. It is natural to make the decision of which group to chose as "official" predecessor based on the order "<" given by group ids. Thus, among all candidates for official predecessor for a group g, we choose the previous majority group having the highest group id:

Pred(g) = max{g' | g' < g, g' is a majority group, *and* $\exists$ q $\in$ members(g): q joined g' }

If no member of g was ever joined to a majority group, we define Pred(g) = (0,0).[7]

We strengthen requirement (Sp) of a membership service to inform new majority group members of the predecessor of the current group, as defined above:

(Sp') *Predecessor Notification for Majority Groups.* Let p be a processor that joins a majority group g. At join time we require the membership protocol to let p know the majority group predecessor Pred(g) of g in addition to the individual predecessor groups pred(g,q) for all q$\in$members(g).

In the example from page 27 above, the predecessor of g_2 is g_1', the predecessor of g_1' is g_1, and the predecessor of g_1 is g_0. In other words, this is an example where the Pred relationship yields a linear order of groups. However, this is not the case in general. If we modify the example slightly by assuming that *all* of the "join" messages for group g_1 are lost, then p_3 will be the only processor that ever joins g_1. Then g_1 cannot be the predecessor of g_1', because according to our definition, at least one member of g_1' must have joined Pred(g_1'), but p_3 is not a member of g_1', and therefore none of the members of g_1' ever joined g_1. In this case, Pred(g_1') will be g_0, i.e., g_0 would be the predecessor of both g_1 and g_1'.

---

[7]See also the definition of pred(g,p) on Page 23.

Hence, although Pred defines a unique majority group predecessor for any majority group, a majority group may have *more than one* successor. Therefore, the Pred relation does, in general, not define a simple linear history among majority groups, but allows *branches*. We call a group g an *ancestor* of g', if g < g' and g and g' are on the same history branch. In other words, g is an ancestor of a group g' if g=Pred(g') or g is an ancestor of Pred(g'). By definition, the group (0,0) is an ancestor of all majority groups.

The reason we restrict our definition of the Pred relation to majority groups only, is that by doing so, we limit disagreements between two branches of the team history to short lived, incomplete majority groups. In our example, g_1 and g_1' are incomplete (p_1 never joins g_1, and p_2 never joins g_1'). While in the incomplete group, g_1', p_2 can temporarily see the failure of p_5 before it joins its next completed group g_2 to learn that the official team history is that p_3 failed before p_5. Thus, if agreement on the order of failures and joins is essential for the correctness of an application, only members who know that they are joined to completed majority groups can act on membership change information. Note that requirement (Ti) (page 16) limits the time that a processor can be joined to an incomplete group, so the delay in taking action on membership changes caused by incomplete groups is not likely to be very significant. We can now formalize our requirement that disagreement on majority group histories be limited to transient incomplete groups as follows:

(So) *Linear history of completed majority groups.* Let g < g' be two completed majority groups, i.e., majority groups that were joined by all their members. Then g is an ancestor of g' under the Pred relation.

Requirement (So) forbids the existence of completed majority groups in two parallel branches of the history of majority groups that may exist over time. This property is useful for applications where servers take actions *only* only after they learn that they are joined to a complete majority group. As an example of a higher level application that needs a membership service satisfying (So), consider an atomic broadcast service that maintains consistent copies of the state of a team of servers by using a train broadcast protocol as described in [Cri91a]. The two round train protocol works as follows: A message called the "group train" circulates among group members in a manner similar to the attendance list discussed earlier. When a processor wants to broadcast an update u to be applied to the team state, it waits for the train message, appends u to the end of the train; it removes u from the train the second time the train comes around again. Each processors applies u to its local copy of the team state only when it knows that all group members know u, that is, when it sees u coming around in the train for the second time. In this way updates are never applied in transient, incomplete groups, and branches in the group history are not harmful, because they cannot lead to diverging local views on what the team state is. The two round train protocol achieves "strict consistency" [Cri91a] among the local copies of the team state, in that any two team members have at any time prefixes of the history of team updates applied so far.

An alternative, stronger specification for a Majority Group Membership Service would not allow branching histories at all, and thus would avoid any inconsistency between members of majority groups:

(So') *Linear history of majority groups.* Let g < g' be two majority groups. Then g is an ancestor of g' under the Pred relation.

The reason we present both version, (So) and (So'), is that (So), although weaker than (So'), is still very useful and can be implemented by a more efficient protocol than (So').

10.2 The Three Round Protocol for Majority Groups

Requirement (Sp') (page 29) is satisfied by a straightforward modification of the Three Round Membership protocol described in the previous section: when a processor replies to the "new-group" message from the creator of a new group g, in addition to sending the id of the group it is currently joined to, it also sends the id of the most recent majority group it was joined to. The group creator can then compute the pred information useful for detecting logical partitions (see requirement (Sp')) and also the Pred(g) information as being the highest majority group predecessor id it receives. The creator then includes the value of Pred(g) in addition to pred in the "join" message it sends to the new members. The detailed code of this protocol is given in Appendix A. It is also not difficult to see that the otherwise unchanged Three Round Membership protocol satisfies requirement (Sp').

It is less obvious that the same protocol also satisfies (So) (page 30). The intuition behind this is the following. If g_1 is a completed majority group, then every majority group $g > g_1$ must have one member p that both joined g_1 and accepted to join g. By definition, the predecessor of g has an id equal to or larger than any majority group that p joined before accepting to join g. It follows that $g_1 \leq \mathrm{Pred}(g)$. Now consider two distinct, completed majority groups, g_1 and g_2. Without loss of generality, assume $g_1 < g_2$. To satisfy (So), g_1 must be shown to be an ancestor of g_2. By the argument given above, $g_1 \leq \mathrm{Pred}(g_2)$. Then either g_1 is equal to $\mathrm{Pred}(g_2)$, or $g_1 < \mathrm{Pred}(g_2)$. In the first case g_1 is already an ancestor of g_2. In the second case, we can repeat our argument to show that $g_1 \leq \mathrm{Pred}(\mathrm{Pred}(g_2))$, and so on. In general, it can be shown by induction, that if g_1 were not an ancestor of g_2, it would have to have an id smaller than any ancestor of g_2. But this is impossible, since (0,0) is also an ancestor of g_2. For a rigorous proof see Appendix B.

10.3 The Five Round Protocol

A protocol satisfying (So') (page 31) can be derived from the Three Round protocol by adding two additional rounds of message exchanges when majority groups are created.

Of the five rounds of message exchanges, the last three correspond to a commit protocol similar to two phase commit. The commit protocol is necessary to ensure agreement on the team history maintained by each member. A detailed description of the protocol is given in Appendix A. Below we just sketch the main features of this strongest protocol of our family.

Processors join minority groups as in a three round protocol. Five rounds of message exchanges are used only when creating majority groups. To illustrate how the protocol works in such situations, assume that a majority of processors is connected when a processor p attempts to create a new group g. As in the Three Round protocol, p broadcasts a "new group" message and collects "accept" messages as replies from all members of g. But then, instead of sending a "join" message immediately, p first broadcasts a "prepare" message to all group members, and only after having received a "prepared" message as a reply from all members of g, does p send the "join" message to the group members. The history of majority groups maintained by each processor consists of a stable history as well as any recently proposed extensions. A group becomes part of the stable history of a processor only if a majority of processors have learned of the group, i.e., have received a "prepare" message containing the id and membership of the group.

In response to a "new group" message, a member q that accepts to become a member of group g also sends what it knows to be the stable history as well as any recently proposed extensions to this history that are not yet stable at q. All stable histories that p receives are prefixes of each other, but there might exist some disagreement among the proposed extensions to histories that p collects. Among all of the histories it receives, processor p picks the one ending in the group with the highest group id. This new history is then sent in the "prepare" message along with other information such as the group id g and the group membership members(g) to each accepting processor q. The new history is adopted by each processor q∈members(g). At that time each member q joins any majority groups in the history that it is a member of but it has not joined before. Each member then replies with a "prepared" message to p. When p receives "prepared" messages from all processors in members(g) it knows that they all agree on the proposed history of majority groups, so p sends them a final "join" message that notifies them to effectively join g. In Appendix B we show that the Five Round Membership protocol satisfies (So').

10.4 Advantages and Drawbacks

By restricting groups to be majority groups only, the Three Round and Five Round Protocols provide stronger ordering guarantees on the observed history of group membership changes than the previous protocols. While the Three Round Protocol guarantees a linear history of majority groups only for *completed* majority groups, the Five Round Protocol guarantees that all processors observe the same history of majority groups. On the other hand, the Five Round Protocol requires five rounds of messages to create majority group, whereas the Three Round Protocol requires no more messages than the previous Three Round Protocol with Partition Notification.

11 Conclusion

This paper has discussed possible specifications for membership services in timed asynchronous distributed systems: from the simplest, nearly trivial "independent assessment" service to the strongest "agreement on a unique history of majority groups" service. The different services identified are suitable for supporting distributed applications with increasingly strong requirements at increasing costs in complexity and message traffic. For each service a protocol that implements it was also described and proven correct.

The Independent Assessment protocol is perhaps the protocol most frequently implemented in real systems (e.g. [Car85]) because of its simplicity. Unfortunately, the protocol exhibits the "split brain" behavior which can have serious consequences on system availability.

The One Round Protocol avoids split brain behavior, handles efficiently single failures and recoveries and can be used to solve the continuous leadership problem, all this with just n messages per group creation. Its major drawbacks are its slowness to stabilize after multiple failures or recoveries and at promoting new leaders, and the possibility that different processors see different failure and recovery event histories. Since the protocol can cause processors to miss certain events of interest and to disagree on the order in which events occur, the One Round protocol is not suitable for sophisticated applications such as system availability management [Cri93, CM94].

The Three Round Protocol with Partition Notification stabilizes faster after multiple failures and recoveries and provides a faster promotion time for the continuous leadership problem, at the expense of using 3n messages instead of n for group creation. By providing the members of each newly formed group with the predecessor group information, it also enables processors to easily detect potential local state conflicts due to logical partitioning. However, the possibility that processors disagree on the order in which failures and recoveries occur, even they are correct and connected, still exists.

To solve this problem, we introduced the Three Round Protocol for Majority Groups that imposes a unique ordering on all events observed by members of completed majority groups at no extra message cost at group creation. We believe that this protocol is one of the most useful of the five presented, since it provides a good cost/performance compromise for implementing higher level services such as asynchronous atomic broadcast and availability management.

Our last Five Round Protocol ensures that all processors agree on a unique history of events observed by the successive majority groups that exist in a system. The cost is 5n messages to create new majority groups, which for many practical applications may be prohibitively high.

Acknowledgments

The timed asynchronous system model, the specifications and the code of the last four protocols presented in this paper were developed in 1991, while both authors worked at the IBM Almaden Research Center. The first author's move to the University of California at San Diego caused a four year delay in the resumption of work. The correctness proofs and the final text were finished in June of 1995. During this second phase of elaboration, the first author would like to acknowledge the support provided by the Air Force Office of Scientific Research, Sun Microsystems, and the Microelectronics Innovation and Computer Research Opportunities in California.

References

[ACBMT95] E. Anceaume, B. Charron-Bost, P. Minet, and S. Toueg. On the Formal Specification of Group Membership Services. Technical Report 95-1534, Computer Science Department, Cornell University, Ithaca, New York 14853, August 1995.

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership Algorithms in Broadcast Domains. Technical Report CS92-10, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1992.

[BBDG94] O. Babaoglu, M. Baker, R. Davoli, and L. Giachini. Relacs: a Communication Infrastructure for Constructing Reliable Applications. Technical Report CS94-15, Laboratory for Computer Science, The University of Bologna, 1994.

[Car85] R. Carr. The Tandem Global Update Protocol. *Tandem Systems Review*, Jun 1985.

[Car90] R. Carr. The Split Brain Syndrome. *Private Communication*, 1990.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic Broadcast: From Simple Message Diffusion to Byzantine Agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

[CF97] F. Cristian and C. Fetzer. The Timed Asynchronous System Model. Technical Report CS97-519, UCSD, Jan 1997.

[CHT96] T. Chandra, V. Hadzilacos, and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, 1996.

[CHTCB96] T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, May 1996.

[CM94] F. Cristian and S. Mishra. Automatic Service Availability Management in Asynchronous Distributed Systems. In *Proc. 2d Int. Workshop on Configurable Distributed Systems*, Pittsburgh, PA, March 1994.

[Cri89] F. Cristian. Probabilistic Clock Synchronization. *Distributed Computing*, 3:146–158, 1989.

[Cri91a] F. Cristian. Asynchronous Atomic Broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, Feb 1991. Also presented at the First IEEE Workshop on Management of Replicated Data, Houston, TX, (Nov 1990).

[Cri91b] F. Cristian. Reaching Agreement on Processor-group Membership in Synchronous Distributed Systems. *Distributed Computing*, 4:175–187, 1991. Early version: *FTCS-18*, 1988, Kyoto.

[Cri91c] F. Cristian. Understanding Fault-Tolerant Distributed Systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[Cri93] F. Cristian. Automatic Reconfiguration in the Presence of Failures. *Software Engineering Journal*, pages 53–60, Mar 1993.

[DHS86] D. Dolev, J. Y. Halpern, and R. Strong. On the Possibility and Impossibility of Achieving Clock Synchronization. *Journal of Computer and System Science*, 32(2):230–250, 1986.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeywer. Consensus in the Presence of Partial Synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[DMS94] D. Dolev, D. Malki, and R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report CS94-6, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1994.

[DMS95] D. Dolev, D. Malki, and R. Strong. A Framework for Partionable Membership Services. Technical Report CS95-4, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1995.

[EMS95] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a Fault-Tolerant Group Communication Protocol. In *Proceedings of the 15th International Conference on Distributed Systems*, Vancouver, Canada., May 1995.

[FC96] C. Fetzer and F. Cristian. Fail-Aware Failure Detectors. In *Proceedings of the 15th Symposium on Reliable Distributed Systems*, Niagara-on-the-Lake, Canada, Oct 1996.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[Gif79] D. K. Gifford. Weighted Voting for Replicated Data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Pacific Grove, CA, Dec 1979.

[HS95] M. Hiltunen and R. Schlichting. Properties of Membership Services. In *Proc. 2d Int. Symp. on Autonomous Decentralized Systems*, Phoeniz, AZ, Apr 1995.

[JFR93] F. Jahanian, S. Fakhouri, and R Rajkumar. Processor Group Membership Protocols: Specification, Design and Implementation. In *Proc. 12th Symposium on Reliable Distributed Systems*, Oct 1993.

[KB91] P. Stevenson K. Birman, A. Schiper. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.

[KT91] F. Kaashoek and A. Tanenbaum. Group Communication in the Amoeba Distributed System. In *Proc. 11th Int. Conf. on Distributed Computing Systems*, pages 222–230, May 1991.

[MFSW95] C. P. Malloth, A. Fehler, A. Schiper, and U. Wilhelm. Phoenix: a Toolkit for Building Fault-Tolerant Distributed Aplications in Large-Scale Networks. In *Proc. of IEEE Workshop on Parallel and Distributed Platforms in Industrial Products*, San Antonio, TX, Oct 1995.

[MPS92] S. Mishra, L. Peterson, and R. 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.

[MSPMA94] L. E. Melliar-Smith P. M., Moser and V. Agrawala. Processor Membership in Asynchronous Distributed Systems. *IEEE Trans. on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[Nei96] G. Neiger. A New Look at Membership Services. In *15th ACM Symposium on Principles of Distributed Computing*, pages 331–340, May 1996.

[RB91] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *10th ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1991.

[RB95] A. Ricciardi and K. Birman. Process Membership in Asynchronous Environments. Technical report, Computer Science Depmt, Cornell University, 1995. Available via anonymous ftp from ftp.sc.cornell.edu as /pub/team/asyncmembership.ps.Z.

[Ric93] A. Ricciardi. *The Group Membership Problem in Asynchronous Systems.* PhD Thesis, Department of Computer Science, Cornell University, 1993.

[SSCA87] R. Strong, D. Skeen, F. Cristian, and H. Aghili. Handshake Protocols. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 521–528, Berlin, Sep 1987.

[vRBM96] R. van Renesse, K. Birman, and S. Mafeis. Horus: a Flexible Group Communication System. *CACM*, 39(5):76–83, April 1996.

A Appendix: Protocol code

This appendix contains detailed Pascal-like pseudo code for our protocols. The Independent Assessment Protocol is omitted because of its simplicity. We assume the existence of subroutine libraries that support higher-level data structures, such as sets, lists, and queues, as well as primitives for sending a message to a set of processors and for setting timeouts. Messages are specified by a string label identifying the type of message and one or more parameters containing the message data. Timeouts are specified through variables of type **timer**, which supports the operations **set**(d), which schedules a timeout event to be generated d seconds into the future, and **cancel**(), which cancels a previously scheduled timeout.

The protocols are presented as event-driven programs, reacting to *message events* (a message was received) and *timer events* (a timeout has occurred). We use the following syntax to wait for the next one from a list of events and to process that event:

```
select event
    when receive ("x", a, b, ...) from p
        ...
    when y.timeout:
        ...
end select
```

This code fragment would wait for the arrival of the next message of type "x" or a timeout of timer variable y, whichever happens first. If a message of type "x" is received, its parameters (the message data) are assigned to the local variables a, b, ..., and the identity of the sender is assigned to local variable p. Our pseudo code also uses the following functions that operate on group membership sets m:

- Leader(m) returns the processor with the lowest id in m.
- RightNeighbor(p,m) and LeftNeighbor(p,m) return the left and right neighbor of processor p in m.
- Rank(p,m) returns an integer between 1 and |m| indicating the position of p in the cyclic neighbor relation; the leader has highest rank.
- Majority(m) is a boolean function that returns true if the set m contains a majority of processors in P. This predicate is assumed to satisfy the requirement: Majority(m) and Majority(m') $\Rightarrow$ m $\cap$ m' $\neq \{\}$.

A.1 The One Round Protocol

We describe the One Round Protocol as a single main task containing an event loop that handles all messages and timeout events. The main task calls subsequently defined procedures to handle processing of "new-group" messages, probe messages, and missing-neighbor timeouts.

const P = ProcessorSet;

type N = **integer**;
T = TimeValues;
G = **record** n: N; p: P **end**;

const π = NeighborPeriod;
μ = ProbePeriod;
δ = TimeoutDelay;

persistent var
CurrentGid: G **initially** (0, myid);

var members: **set of** P;
timestamps: **map** P$\rightarrow$T;
MissingNeighbor, SendIAmAlive, SendProbe: **timer**;

msg ("new-group", G, **set of** P, **map** P$\rightarrow$T);
("I-am-alive", G, T);
("probe", G, **set of** P, **map** P$\rightarrow$T);

```
task  Main;
var      p: processid;
         gid': G; members': set of P; timestamps': map P→T; t': T;
begin
     timestamps(myid)← LocalTime();
     for all  p ∈ P − {myid}  do
          timestamps(p)← 0;
     end for
     CurrentGid← (CurrentGid.n+1, myid);
     members← {myid};
     SendProbe.set(0);
     loop
          select event
               when SendIAmAlive.timeout:
                    % it is time to send "I am alive" message to right neighbor
                    p← RightNeighbor(myid, members)
                    send ("I-am-alive", CurrentGid, LocalTime())  to  p;
                    SendIAmAlive.set(π);
               when MissingNeighbor.timeout:
                    % "I am alive" message from left neighbor is missing
                    members← members − LeftNeighbor(myid, members);
                    CreateNewGroup(CurrentGid);
               when SendProbe.timeout:
                    % it is time to send a probe message
                    timestamps(myid)← LocalTime();
                    send ("probe", CurrentGid, members, timestamps)  to  P − members;
                    % Note that probes are sent only by the group leaders of minority
                         groups because in procedure Join, the SendProbe timer is set only
                         if this processor is the leader of a minority group
                    SendProbe.set(μ);
               when receive ("new-group", gid', members', timestamps')  from  p
                    NewGroupInfo(gid', members', timestamps');
               when receive ("I-am-alive", gid', t')  from  p:
                    if  gid' = CurrentGid  then
                         timestamps(p)← t';
                         MissingNeighbor.set(π + Rank(myid, members)*δ);
                         if  myid ≠ Leader(members)  then
                              p← RightNeighbor(myid, members)
                              send ("I-am-alive", CurrentGid, LocalTime())  to  p;
                         end if;
                    end if
               when receive ("probe", gid', members', timestamps')  from  p:
                    NewGroupInfo(gid', members', timestamps');
          end select
     end loop
end
```

```
procedure NewGroupInfo(gid': G; members': set of P; timestamps': map P→T)
var       MyInfoUptodate: boolean initially true;
          OtherInfoUptodate: boolean initially true;
          p: processid;
begin
     % Merge my knowledge of the state of the system with the new information
     for all p ∈ P do
          if p ∉ members and p ∈ members' then
                    if timestamps(p) < timestamps'(p) then
                         members← members ∪ {p};
                         MyInfoUptodate← false;
                    else
                         OtherInfoUptodate← false;
                    end if
          else if p ∈ members and p ∉ members' then
                    if timestamps(p) ≤ timestamps'(p) and p ≠ myid then
                         members← members − {p}
                         MyInfoUptodate← false;
                    else
                         OtherInfoUptodate← false;
                    end if
          end if
          timestamps(p)← max(timestamps(p), timestamps'(p));
     end for
     if CurrentGid < gid' then
          if OtherInfoUptodate then
               Join(gid');
          else
               CreateNewGroup(gid');
          end if
     else if CurrentGid > gid' then
          if MyInfoUptodate then
               % ignore
          else
               CreateNewGroup(CurrentGid);
          end if
     end if
end
```

```
procedure CreateNewGroup(oldid: G)
var     newid: G
begin
    newid← (oldid.no + 1, myid);
    timestamps(myid)← LocalTime();
    send ("new-group", newid, members, timestamps) to P − {myid};
    Join(newid);
end

procedure Join(newid: G)
begin
    % update current group id
    CurrentGid← newid;
    % cancel all timers
    SendProbe.cancel();
    SendIAmAlive.cancel();
    MissingNeighbor.cancel();
    % set new timers
    if Leader(members) = myid and ¬ Majority(members) then
        SendProbe.set(μ);
    end if
    if |members| > 1 then
        if Leader(members) = myid then
            SendIAmAlive.set(π − δ);
        end if;
        MissingNeighbor.set(π + Rank(myid, members)*δ);
    end if
end
```

A.2 The Three Round Protocols

Since Three Round for Majority Groups protocol is a superset of the one with Partition Detection, we only describe the former. The code for the latter can be derived from the former by simply removing all references to majority group predecessors (variables LastMajorityGroup, Pred and maxPred in the code).

The protocol is structured as three concurrent tasks, each responsible for handling a different set of related events. Instead of including explicit synchronization between these tasks, we simply assume that they are non-preemptable. Thus, the global protocol variables shared by all three tasks are only updated by one task at a time, just as if the protocol were written as a single task with one large event loop.

```
const   P = ProcessorSet;

type    N = integer;
        G = record n: N; p: P end;

const   π = NeighborPeriod;
        μ = ProbePeriod;
        δ = TimeoutDelay;

persistent var
        CurrentGid: G initially (0, 0);
        MaxKnownGid: G initially (0, 0);
        LastMajorityGroup: G initially (0, 0);

var     joined: Boolean;
        members: set of P;
        WinningGroupCreator: P;
        MissingNeighbor, SendIAmAlive, SendProbe: timer;

msg     ("new-group", G, P);
        ("accept", G, G, G);
        ("join", G, set of P, G, map P→G);
        ("probe", G);
        ("I-am-alive", G);

task Main;
begin
    % join initial group with myid as the only member
    Join((CurrentGid.n+1, myid), {myid}, LastMajorityGroup, {(myid, (0,0))});
    % start the three tasks that handle protocol events
    start(MonitorProbes);
    start(MonitorAttendance);
    start(MonitorGroupCreations);
end
```

```
task NewGroupCreation();
    var     newid: G;
            MissingAccept: timer;
            accepters: set of P initially {myid};
            maxPred: G initially LastMajorityGroup;
            pred: map P→G initially {(myid, CurrentGid)};
begin
    % propose a new group
    newid← (max(CurrentGid.n, MaxKnownGid.n) + 1, myid);
    MaxKnownGid← newid;
    WinningGroupCreator← myid;
    send ("new-group", newid, myid) to P − {myid};

    % collect replies
    MissingAccept.set(2δ);
    loop
        select event

            when receive ("accept", gid, prevMajorityGid, prevGid) from p:
                if gid = newid then
                    accepters← accepters ∪ {p};
                    maxPred← max(prevMajorityGid, maxPred);
                    pred(p)← prevGid;
                    if accepters = P then
                        MissingAccept.cancel();
                        exit loop;
                    end if;
                end if;

            when MissingAccept.timeout:
                exit loop;

        end select
    end loop

    % if I am a winner, tell all members to join
    if newid = MaxKnownGid then
        Join(newid, accepters, maxPred, pred);
        if |accepters| > 1 then
            send ("join", CurrentGid, accepters, maxPred, pred)
                to accepters − {myid};
        end if;
    end if;
end
```

```
task MonitorGroupCreations;
    var     MissingJoin: timer;
begin
    loop
        select event
            when receive ("new-group", newid, p) from q:
                if MaxKnownGid < newid then
                    MaxKnownGid← newid;
                    WinningGroupCreator← p;
                    LeaveCurrentGroup();
                    send ("accept", newid, LastMajorityGroup, CurrentGid) to p;
                    MissingJoin.set(3δ);
                else if (MaxKnownGid > newid) then
                    send ("new-group",MaxKnownGid,WinningGroupCreator) to q;
                end if;
            when receive ("join", newid, memb, Pred, pred):
                if newid = MaxKnownGid then
                    MissingJoin.cancel();
                    Join(newid, memb, Pred, pred);
                end if;
            when MissingJoin.timeout:
                start(NewGroupCreation);
        end select
    end loop
end
```

```
task MonitorProbes;
begin
    loop
        select event
            when receive ("probe", gid):
                if joined and Leader(members) = myid and
                            (Majority(members) or (gid < CurrentGid)) then
                    LeaveCurrentGroup;
                    start(NewGroupCreation);
                end if;
            when SendProbe.timeout:
                send ("probe", CurrentGid) to P − members;
                % Note that probes are sent only by the group leaders of minority
                      groups because in procedure Join, the SendProbe timer is set only
                      if this processor is the leader of a minority group
                SendProbe.set(μ);
        end select
    end loop
end

task MonitorAttendance;
begin
    loop
        select event
        when SendIAmAlive.timeout:
            send ("I-am-alive", CurrentGid) to RightNeighbor(myid, members);
            SendIAmAlive.set(π);
        when receive ("I-am-alive", g):
            if g = CurrentGid then
                MissingNeighbor.set(π + Rank(myid, members)*δ);
                if Leader(members) ≠ myid then
                    p← RightNeighbor(myid, members)
                    send ("I-am-alive", CurrentGid) to p;
                end if;
            end if
        when MissingNeighbor.timeout:
            LeaveCurrentGroup;
            start(NewGroupCreation);
        end select
    end loop
end
```

```
procedure Join(g: G, m: set of P, Pred: G, pred: P→G);
begin
    % notify interested parties of the group change
    signal ("Join-Notification", g, m, Pred, pred) to user;
    % update group id and related state
    if (Majority(members)) then LastMajorityGroup← CurrentGid; end if;
    CurrentGid← g;
    MaxKnownGid← g;
    members← m;
    joined← true;
    % set new timers
    if Leader(members) = myid and ¬ Majority(members) then
        SendProbe.set(μ);
    end if;
    if |members| > 1 then
        if Leader(members) = myid then
            SendIAmAlive.set(π − δ);
        end if;
        MissingNeighbor.set(π + Rank(myid, members)*δ);
    end if
end

procedure LeaveCurrentGroup;
begin
    if joined then
        % notify interested parties of the group change
        signal ("Leave-Notification") to user;
        % update the state to not joined
        joined← false;
        % cancel timers
        MissingNeighbor.cancel();
        SendIAmAlive.cancel();
        SendProbe.cancel();
    end if
end
```

A.3 The Five Round Protocol

The majority group history maintained by the Five Round Protocol is represented in the pseudo code below by the variable "hist", which is declared as a queue of records containing the group id and membership of groups in the majority group history. We assume that the data type "queue" supports the following operations: *q.append(e):* appends a new element e to the end of the queue q. *q.remove:* removes the first element from the front of the queue q. *q.last:* returns the last element that was appended to the end of the queue; does not alter the content of the queue. *q.n:* returns the number of append operations that have been applied to the queue, i.e., q.n is equal to the number of elements currently in the queue plus the number of elements that have been removed from the queue. *q.peek(i):* returns the i'th element that was appended to the queue, assuming the element has not yet been removed; the index i must be less than or equal to the q.n; does not alter the content of the queue.

The initial value of the history queue hist is set to contain only the initial group (0,0) that every processor is implicitly joined to when it first enters the system. As new majority groups are formed, they are appended to the end of the history queue; groups that are known to have been joined by all its members are removed from the front of the queue. In addition, each processor maintains a variable named "stable" that indicates which of the groups in the history are known by a majority of processors. Hence the majority group history can be divided into three sections: (1) completed majority groups: groups that are known to have been joined by all its members, and have been removed from the history queue. (2) stable majority groups: groups that are known to be part of the history queue at a majority of processors, and (3) recently proposed extensions to the history that may not yet have been seen by a majority of processors. The variable "stable" contains the index of the most recent group in the history queue that is known to be stable, i.e., stable groups are all groups up to and including hist.peek(stable). A group becomes stable when a majority of processors has received and processed a "prepare" message containing the group.

Removing completed majority groups from the history queue is not necessary for correctness, but prevents histories from growing indefinitely. The fastest way to prune histories would be through an extra round of acknowledgments after joining a majority group. An alternative is to rely on the Bounded Inconsistency Detection Delay (Td). This property implies that, if a processor remains joined to a group for a period of length D', it can assume that all other members have joined the group. We use the latter approach, since it requires fewer messages: after joining a majority group g, a processor sets a timer to go off D' seconds into the future. If the processor is still joined to g when the timer goes off, it can prune its history to contain only a single group g.

```
const   P = ProcessorSet;
type    N = integer;
        G = record n: N; p: P end;
const   π = NeighborPeriod;
        μ = ProbePeriod;
        δ = TimeoutDelay;
        D' = InconsistencyDetectionDelay;
type    MajorityGroupHist = queueof
            record
                gid: G;
                memb: set of P;
            end;
persistent var
        CurrentGid: G initially (0, 0);
        MaxKnownGid: G initially (0, 0);
        hist: MajorityGroupHist initially { ((0,0), P) };
        stable: integer initially 1
var     joined: Boolean;
        members: set of P;
        WinningGroupCreator: P;
        MissingNeighbor, SendIAmAlive, SendProbe, GroupComplete: timer;
msg     ("new-group", G, P);
        ("accept", G, MajorityGroupHist, integer);
        ("prepare", G, MajorityGroupHist, integer);
        ("prepared", G);
        ("join-majority", G);
        ("join-minority", G, set of P);
        ("probe", G);
        ("I-am-alive", G);

task Main;
begin
    % join initial group with myid as the only member
    JoinMinority((CurrentGid.n+1, myid), {myid});
    % start the three tasks that handle protocol events
    start(MonitorProbes);
    start(MonitorAttendance);
    start(MonitorGroupCreations);
    start(MonitorGroupCompletion);
end
```

```
task NewGroupCreation();
    var     newid: G;
            MissingReply: timer;
            memb: set of P;
            newhist: MajorityGroupHist;
            newstable: integer;
            replies: set of P;
            h: MajorityGroupHist; s: integer;
begin
    % propose a new group
    newid← (max(CurrentGid.n, MaxKnownGid.n) + 1, myid);
    MaxKnownGid← newid;
    WinningGroupCreator← myid;
    memb← {myid};
    newhist← hist;
    newstable← stable;
    send ("new-group", newid, myid) to P − {myid};

    % collect replies
    MissingReply.set(2 δ);
    loop
        select event
            when receive ("accept", newid, h, s) from p:
                memb← memb ∪ {p};
                % remember the history with the highest id
                if h.last.id > newhist.last.id then newhist← h end if;
                % remember the highest s
                if s > newstable then newstable← s end if;
                % quit loop when all procesors have replied
                if memb = P then
                    MissingReply.cancel();
                    exit loop;
                end if

            when MissingReply.timeout:
                exit loop;
        end select
    end loop

    if newid ≠ MaxKnownGid then return end if
```

```
    if ¬ Majority(memb) then
        JoinMinority(newid, memb);
        send ("join-minority", newid, memb) to memb − {myid}
        return;
    end if;
    % append the new group to the end of the history
    newhist.append((newid, memb));
    % send "prepare" message to new members
    Prepare(newid, newhist, newstable);
    send ("prepare", newid, newhist, newstable) to memb − {myid};
    MissingReply.set(2 δ);
    replies← ∅;
    loop
        select event
            when receive ("prepared", gid) from p:
                replies← replies ∪ {p};
                if replies = memb − {myid} then
                    MissingReply.cancel();
                    exit loop;
                end if
            when MissingReply.timeout:
                start(NewGroupCreation);
                return;
        end select
    end loop
    if newid ≠ MaxKnownGid then return end if
    % send "join" message
    JoinMajority(newid);
    send ("join-majority", newid) to memb − {myid};
end
```

```
task MonitorGroupCreations;
    var    newid: G; newhist: MajorityGroupHist; newstable: integer;
           memb: set of P;
           MissingJoin: timer;
begin
    loop
        select event
            when receive ("new-group", newid, p) from q:
                if MaxKnownGid < newid then
                    MaxKnownGid← newid;
                    WinningGroupCreator← p;
                    LeaveCurrentGroup;
                    send ("accept", newid, hist, stable) to p;
                    MissingJoin.set(3 δ)
                else if MaxKnownGid > newid then
                    send ("new-group",MaxKnownGid,WinningGroupCreator) to q;
                end if;
            when receive ("join-minority", newid, memb):
                if newid = MaxKnownGid then
                    MissingJoin.cancel();
                    JoinMinority(newid, memb);
                end if;
            when receive ("prepare", newid, newhist, newstable) from q:
                if newid = MaxKnownGid then
                    MissingJoin.cancel();
                    Prepare(newid, newhist, newstable);
                    send ("prepared", newid) to q:
                    MissingJoin.set(3 δ);
                end if;
            when receive ("join-majority", newid):
                if newid = MaxKnownGid then
                    MissingJoin.cancel();
                    JoinMajority(newid);
                end if;
            when MissingJoin.timeout:
                start(NewGroupCreation);
        end select
    end loop
end
```

```
task MonitorProbes;
begin
    loop
        select event
            when receive ("probe", gid):
                if joined and Leader(members) = myid and
                            (Majority(members) or (gid < CurrentGid)) then
                    LeaveCurrentGroup;
                    start(NewGroupCreation)
                end if;
            when SendProbe.timeout:
                send ("probe", CurrentGid) to P − members;
                % Note that probes are sent only by the group leaders of minority
                    groups because in procedure Join, the SendProbe timer is set only
                    if this processor is the leader of a minority group
                SendProbe.set(μ);
        end select
    end loop
end

task MonitorAttendance;
begin
    loop
        select event
        when SendIAmAlive.timeout:
            send ("I-am-alive", CurrentGid) to RightNeighbor(myid, members);
            SendIAmAlive.set(π)
        when receive ("I-am-alive", g):
            if g = CurrentGid then
                MissingNeighbor.set(π + Rank(myid, members)*δ);
                if Leader(members) ≠ myid then
                    p← RightNeighbor(myid, members)
                    send ("I-am-alive", CurrentGid) to p;
                end if;
            end if
        when MissingNeighbor.timeout:
            LeaveCurrentGroup;
            start(NewGroupCreation);
        end select
    end loop
end
```

```
task MonitorGroupCompletion;
begin
    loop
        select event
            when GroupComplete.timeout:
                % prune the history by removing all but the last element
                hist ← { hist.last }
        end select
    end loop
end

procedure JoinMinority(g: G, m: set of G);
begin
    % notify interested parties of the group change
    JoinMinorityNotify(g, m);

    % update group id and related state
    CurrentGid← g;
    MaxKnownGid← g;
    members← m;
    joined← true;

    % set new timers
    if Leader(members) = myid then
        send ("probe", g) to P − m;
        SendProbe.set(μ)
    end if;
    if |members| > 1 then
        if Leader(members) = myid then
            SendIAmAlive.set(π − δ);
        end if;
        MissingNeighbor.set(π + Rank(myid, members)*δ);
    end if
end
```

```
procedure Prepare(newid: G, newhist: MajorityGroupHist, newstable: integer);
    var     i: integer;
begin
    % join any new, stable majority group
    hist← hewhist;
    for i← stable + 1 to newstable do
        (g, m)← hist.peek(i);
        if myid ∈ m then
            signal ("Join-Notification", g, m) to user;
        end if
    end for
    stable← newstable;
    CurrentGid← newhist.last.id;
    MaxKnownGid← newhist.last.id;
    members← newhist.last.memb;
    % set new timers
    if Leader(members) = myid then
        send ("probe", g) to P − m;
        SendProbe.set(μ)
    end if;
    if |members| > 1 then
        if Leader(members) = myid then
            SendIAmAlive.set(π − δ);
        end if;
        MissingNeighbor.set(π + Rank(myid, members)*δ);
    end if
end
```

```
procedure JoinMajority(g: G);
    var     i: integer;
begin
    % join any new, stable majority group
    for i← stable + 1 to hist.n do
        (g, m)← hist.peek(i);
        if myid ∈ m then
            signal ("Join-Notification", g, m) to user;
        end if
    end for
    % update group id and related state
    stable← hist.n;
    joined← true;
    % set timer to go off when all members are known to have joined
    GroupComplete.set(D');
end

procedure LeaveCurrentGroup;
begin
    if joined then
        signal ("Leave-Notification") to user;
        joined← false;
        MissingNeighbor.cancel();
        SendIAmAlive.cancel();
        SendProbe.cancel();
        GroupComplete.cancel();
    end if
end
```

B Appendix: Correctness Proofs

To make it easier to follow the correctness proofs presented here we begin with a concise summary of our assumptions, definitions, requirements, and protocol properties.

B.1 System Model & Terminology

The system consists of a set of processors, P, linked by communication network. In the absence of failures, messages between two processors are delivered within δ time units. Each processor has a local clock and stable storage.

Failure assumptions:

- Crash or performance processor failures.
- Omission or performance communication failures.

Def: At any point in time, the communication between two processors p and q can be in one of three abstract states: *connected*, *disconnected*, or *partially connected*. These states imply the following about the correctness of p and q and on messages sent between p and q:

- If p and q are *connected* throughout a time interval I=$[t_1,t_2]$, then p and q are correct throughout I, and every message sent by p or q in $[t_1,t_2\text{-}\delta]$ to the other is delivered and processed within δ time units.
- If p and q are *disconnected* throughout a time interval I, then p is crashed throughout I, or q is crashed throughout I, or no messages sent between p or q in I are delivered in I.
- If p and q are *partially connected* throughout I, no conclusions can be made about processor correctness or message delivery in I.

Def: The system P is called *stable* in I if, throughout I:

(1) no processor fails or recovers,

(2) all messages delivered in I are delivered within δ time units of their sending

(3) all pairs of processors in P are either connected or disconnected, and

(4) the connected relation between processors is constant and transitive.

Groups and group ids:

members(p)	The set of processors that p believes are currently correct and connected.
G	The set of all group ids, ordered by "<".
joined(p)	True if processor p is currently joined to a group.
group(p)	Returns the id of the group p is currently joined to (Undefined if joined(p) is false).
members(g)	members(p) for some p joined to g. (Undefined if joined(p) is false).

B.2 Membership Service Requirements

B.2.1 Minimal Requirements

(Tjd) *Bounded Join Delays.* There exists a time constant JD such that, if two processors p and q are connected in I=[t,t+JD], and the system P is stable in I, then p and q are included in members(q) and members(p), respectively, sometime in I.

(Tpd) *Bounded Partition Detection Delays.* There exists a time constant PD such that, if two processors p and q are disconnected in I=[t,t+PD], the system P is stable in I, and p stays correct throughout I, then q is excluded from members(p) sometime in I.

(Sld) *Stability of Local Views.* If the system is stable in I=[t,t'], then for every correct processor p, members(p) will not change in [t+max(JD,PD),t'].

B.2.2 Minimum Requirements for a Group Membership Service

(Sa) *Agreement on group membership.* If two processors p and q are joined to the same group g (not necessarily at the same time), then the two processors have the same view of the membership of that group: If at any time t, joined(p), group(p)=g, and members(p)=m, and at time t', joined(q), group(q)=g', and members(q)=m', then g=g' implies m=m'.

(Sr) *Reflexivity.* For any processor p, if joined(p) then p $\in$ members(p).

(Sm) *Monotonicity of groups.* Let g and g' be groups that processor p joins at times t and t', respectively. If t' < t then g' < g.

(Tj) *Bounded Join Delays.* There exists a time constant J such that, if two processors p and q are connected in I=[t,t+J] and the system P is stable in I, then there must be a common group g, such that p is joined to g at some time in I and q is joined to g at some time in I (not necessarily at the same time as p).

(Td) *Bounded Partition Detection Delays.* There exists a time constant D such that, if two processors p and q are disconnected in I=[t,t+D], the system P is stable in I, and p stays correct throughout I, then there must be a group g with q $\notin$ members(g), such that p is joined to g at some time in I.

(Sl) *Stability of Local Views.* If the system is stable in I=[t,t'], then no processor will leave its group in [t+max(J,D),t'].

(Ti) *Bounded Inconsistency Detection Delays.* There exists a time constant D' such that, if at time t, p is joined to a group g such that q $\in$ members(g), p stays correct in I=[t,t+D'], and at no time in I q joins g, then p leaves g by t+D'.

B.2.3 Membership Service with Partition Notification

Def: For p $\in$ members(g) we define: pred(g,p) = max{ g' | g' < g and p joined g'}.
If p was never joined to a group g' < g, we define pred(g,p) = (0,0).

(Sa) (Sr) (Sm) (Tj) (Td) (Sl) (Ti): see B.2.2.

(Sd) *Legitimacy of processor deletions.* If processor p joins group g at time t such that q $\notin$ members(g) and q was a member of the previous group pred(g,p) joined by p, there must exist a processor r $\in$ members(g) that was not crashed and not connected to q at some time in [t',t], where t' is the time at which r left its predecessor group pred(g,r).

(Sj) *Legitimacy of processor additions.* If processor p joins group g at time t such that q $\in$ members(g) and q was not a member of the previous group pred(g,p) joined by p, there must exist a processor r $\in$ members(g) that was correct and not disconnected from q at some time in [t',t], where t' is the time at which r left its predecessor group pred(g,r).

(Ss) *Join Synchronization.* If processor p joins group g at time t, then no member of g will be joined to a group g' < g after t.

(Sp) *Logical Partition Notification.* Let p be a processor that joins a group g. At join time the membership protocol should let p know the predecessor groups pred(g,q) of all q $\in$ members(g).

B.2.4 Majority Group Membership Service, Weak Agreement

Def: For all g$\in$G we define:
Pred(g)=max{g' | g' < g, g' majority group, *and* $\exists$ q$\in$ members(g): q joined g'}.
If no member of g was ever joined to a majority group, we define Pred(g)=(0,0).

Def: Group g is called *completed* iff all members of g have joined g.

(Sa) (Sr) (Sm) (Tj) (Td) (Sl) (Ti) (Sd) (Sj) (Ss): see B.2.2 and B.2.3.

(Sp') *Predecessor Notification for Majority Groups.* Let p be a processor that joins a majority group g. At join time we require the membership protocol to let p know the majority group predecessor Pred(g) of g in addition to the individual predecessor groups pred(g,q) for all q$\in$members(g).

(So) *Linear history of completed majority groups.*
Let g $<$ g' be two *completed* majority groups, i.e., majority groups that were joined by all their members. Then g is an ancestor of g' under the Pred relation.

B.2.5 Majority Group Membership Service, Strong Agreement

(Sa) (Sr) (Sm) (Tj) (Td) (Sl) (Ti) (Sd) (Sj) (Ss) (Sp'): see B.2.2, B.2.3, and B.2.4.

(So') *Linear history of majority groups.*
Let g $<$ g' be two majority groups. Then g is an ancestor of g' under the Pred relation.

B.3 The Leader Election Problem

(Su) *Uniqueness of Leader.* At any point in time there is at most one leader.

(Te) *Bounded Election Delays.* There exists a time constant E such that, if the system P is stable in I=[t,t+E] and a majority of processors is connected then there will be a leader by t+E.

B.4 Protocols Properties

Parameters:

π = period for exchanging "I-am-alive" messages
μ = period for sending "probe" messages

B.4.1 Independent Assessment Protocol

Satisfies (Tjd), (Tpd) with

$JD = \pi + \delta$
$PD = \pi + 2\delta$

B.4.2 One Round Protocol

Satisfies (Sa), (Sr), (Sm), (Tj), (Td), (Sl), (Ti) with

$J = \max(\pi+n\delta, \mu) + 5\delta$
$D = \max(\pi+n\delta, \mu) + 5\delta + (n - 1)(\pi+(n+1)\delta)$
$D' = 2(\pi+n\delta)$

B.4.3 Three Round Protocols

Satisfies (Sa), (Sr), (Sm), (Tj), (Td), (Sl), (Ti), (Sd), (Sj), (Ss), and (Sp) or (Sp'), (So) with

$J = \max(\pi+(n+3)\delta, \mu) + 9\delta$

$D = \max(\pi+(n+3)\delta,\ \mu) + 9\delta$
$D' = 2(\pi+n\delta)$

B.4.4 Five Round Protocol

Satisfies (Sa), (Sr), (Sm), (Tj), (Td), (Sl), (Ti), (Sd), (Sj), (Ss), (Sp'), (So') with

$J = \max(\pi+(n+3)\delta,\ \mu) + 11\delta$
$D = \max(\pi+(n+3)\delta,\ \mu) + 11\delta$
$D' = 2(\pi+n\delta) + 2\delta$

B.4.5 Leader Election Protocols

Using One Round Membership Protocol: E = J + 2D'
Using Three Round Protocol: E = J + D'

This concludes our summary of protocol properties and requirements. The remainder of this appendix section will prove that our protocols indeed satisfy the requirements listed above.

B.5 Common protocol properties

All four non-trivial protocols have some features in common, for example, the way group ids are created and the order defined on them, as well as the way in which failures and recoveries are detected (attendance list and probe messages). In this section we will present proofs for protocol properties that depend only on these common features and hence apply to all four protocols.

B.5.1 Common Safety Properties

(Sa) *Agreement on group membership*: In all four protocols group ids consist of a processor id (the id of the group creator) and a monotonically increasing counter. Thus, two groups created by different processors will have different group ids. In all four protocols, a processor p joins a new group g=(c,i) only if it has either created the group itself (c=p), or received a message (g,members(g)) from the group creator c. Therefore, all processors that join a group g=(c,i), must either be c or must have received a message from c containing the same membership set members(g). Hence, all processors that join g adopt the same membership for g.

(Sr) *Reflexivity*: In the two three-round protocols and in the five-round protocol "Join" messages are only sent to group members. Hence only members will join a group.

In the one-round protocol, there are no "Join" messages, and "New-group" messages are sent to every processor. However, if a processor receives a "New-group" message, it specifically checks whether the membership list it received includes itself. If it does not, it will not join the new group. [8]

(Sm) *Monotonicity of groups*: In all four protocols, a processor p only joins a group g if g is higher than the id of the group p is currently joined to. Therefore the sequence of group ids joined by a processor is strictly increasing. Hence two processors that join the same two groups must join them in the same order, namely the order given by the group ids.

(Sl) *Stability of Local Views*: Assume that after a number of failures and recoveries the system has stabilized: at time t_0 any pair of processors is either connected or disconnected and the "connected" relation is transitive. This means that all processors that are not crashed form *communication cliques* or *physical partitions* $P_1, \ldots, P_k \subseteq P$. Furthermore, if the system is stable, no late messages arrive after t_0. If the protocol satisfies the bounded join and partition detection delay requirements, (Tj) and (Td), then all processors in partition

[8] See procedure NewGroupInfo: because of the statement "if ... p $\neq$ myid" in the for loop, the variable OtherInfoUptodate will always be set to false in this case; if OtherInfoUptodate is false, the processor will not join the group.

P_i will have joined a common group g_i with membership equal to P_i by time t_0+max(J,D). There are only two reasons why a member of g_i would leave the group if the system remains stable after t_0+max(J,D):

1. A member of g_i receives a message containing a higher group id.

   Such message must have been sent from a processor in another physical partition, because all processors in P_i eventually join g_i. Since the system is stable, all such messages are delivered no later than $t_0+\delta$.

2. A member of g_i times out waiting for an I-am-alive message from its neighbor.

   In all protocols the SendIAmAlive timer and MissingNeighbor timers are set so that if all members join a group within δ time units of each other, and the system is stable, then all I-am-alive messages will be sent such that they arrive before the recipient's MissingNeighbor timer can run out. Therefore, this is possible only if group g_i was created before time t_0 when the system was unstable. In this case, however, if a MissingNeighbor timeout occurs after time t_0, this must happen 'soon' after t_0. To see why this is so, consider what happens the first time that the leader of g_i sends a new I-am-alive message after time t_0. Since the system is stable, the message will reach the k^{th} member of g_i no more the k δ time units later; at that time that member will reset its MissingNeighbor timer so that it will *not* run out before the next I-am-alive message from the group leader will reach that processor. Therefore, if any MissingNeighbor timeouts occur after t_0, this must happen before the first I-am-alive message sent after t_0 has reached all members of g_i.

In either case, new groups may be created even after all processors in a physical partition P_i have joined a common group g_i, but, as we have argued, this can happen only for a limited time after t_0. If the system is stable for another max(J,D) time units after that, all processors in P_i must again have joined a common group g_i', and will not leave this group as long as the system remains stable.

Of course, to complete this proof, we still have to show that requirements (Tj) and (Td) are satisfied. We will do so for each protocol individually in subsequent sections.

B.5.2 Common Timeliness Properties

(Ti) *Bounded Inconsistency Detection Delays*: We claim that all four protocols satisfy requirement (Ti) for D' = $2(\pi + n\ \delta)$, where n is the number of processors in the system.

To prove our claim, we show that the following, equivalent statement is true: if a processor p is joined to a group g for a time interval I of length D', then every member of g $q \in$ members(g) must be joined to g at some time $t \in$ I. The reason is that in an interval of length $2(\pi + n\ \delta)$, p must have received at least two I-am-alive messages; otherwise, it would have left the group g. For p to receive two I-am-alive messages, the first one must have been received and forwarded by all members of g; otherwise, the leader of g would would have left g before sending the second I-am-alive message. Therefore every member of g must have been joined to g at some time in I. [9]

B.6 Properties of the One-Round Protocol

In the previous sections we showed that all four protocols satisfy the safety requirements (Sa), (Sr), (Sm), and (Sl), as well as the timeliness requirement (Ti). We prove here that the 2D' waiting rule for electing a leader in a majority group is correct, and that the protocol specific (Tj) and (Td) requirements also hold.

The proof that the 2D' waiting rule is correct is by contradiction. Assume that, at time t, there are two leaders p_1 and p_2 joined to majority groups g_1 and g_2. Let t_1=t-2D' and t_2=t-D'. By the waiting rule, p_1 and p_2 were joined to g_1 and g_2 respectively throughout the interval $[t_1,t]$. Without loss of generality, suppose $g_1 < g_2$. Since both g_1 and g_2 are majority groups, their membership must intersect, so let $p \in$ members(g_1) $\cap$ members(g_2). Processor p must have been joined to g_1 at some time in $[t_2,t]$, otherwise by (Ti) processor p_1 would either have crashed, detected a performance failure, or have left g_1 by t. Similarly, p must have been joined to g_2 at some time in $[t_1,t_2]$, otherwise p_2 would have failed or left g_2 by t_2. It follows that p must have been joined to g_2 before being joined to g_1, which together with (Sm) contradicts our working hypothesis that $g_1 < g_2$.

We now proceed to prove the (Tj) and (Td) requirements. Assume that the system stabilizes at time t_0. Below we show that if the system remains stable for a sufficiently long time, all processors in one physical partition eventually join a common group with a membership equal to the set of processors in that partition. Thus, (Tj) and (Td) will be satisfied.

Intuitively, the reason why these properties hold is as follows. After the system stabilizes, there could be some messages broadcast by processors that decide to create some new groups. Recall that, according to the rules of the protocol, such messages are broadcast to all processors, not only to the group membership. With each such message, the "collective

[9] Note that this argument assumes that the leader does not send a new I-am-alive message before it has received a copy of its previous I-am-alive message from its left neighbor. This assumption implies that π can be no smaller than 2 δ. Although the protocol would still remain correct if π is chosen smaller, this would not improve the bound D'. In fact, it would make the inconsistency detection delay worse! For example, if π were chosen just smaller than n δ, then when p receives the first I-am-alive message, it cannot infer that this first I-am-alive was received by all members. This is so because a leader sends the second I-am-alive before its MissingNeighbor timer for the previous I-am-alive message runs out. Hence, for such a choice of π, processor p would need to wait for at least three I-am-alive messages before it could be sure that every member has joined g. This would lead to a bound of D' = $3(\pi + n\,\delta)$. As π is made smaller, the term $(\pi + n\,\delta)$ decreases, but the factor in front of it increases, because p needs to wait for more I-am-alive messages. The net effect will be a larger bound D'.

knowledge" of all processors about the state of the system increases, because every processor that receives such a message combines its previously recorded knowledge with the information it receives in the message.

Unfortunately, matters are complicated by the possibility of spurious timeout occurrences (see Section 7). We will define this notion more precisely here. Assume processors p and q are in the same physical partition and p is joined to a group g with q $\in$ members(g). Furthermore, assume that in g, q is p's left neighbor, such that p expects to receive I-am-alive messages from q. It is possible that, due to failures that occurred before the system stabilized, q is joined to a different group and sends its I-am-alive messages to a processor other than p. Processor p will time out waiting for I-am-alive messages from q and conclude that q has crashed or become disconnected. We say that a *spurious timeout* occurs if after the system has stabilized, a processor p creates a group excluding processor q, although p and q are in the same physical partition. Spurious timeouts may actually decrease the "collective knowledge": p will create a new group g' excluding q and send out new-group messages that may convince other processors in its partition that q is crashed or disconnected. Below we prove a number of Lemmas that show that spurious timeouts can occur only for a bounded time after a system stabilizes.

Let $t_1 = t_0 + \delta$. Note that any broadcast message delivered in P_i after t_1 must have been sent after t_0, since no late messages are delivered in a stable system. Hence such a broadcast was sent by a processor in P_i and will be delivered to all its recipients in P_i, provided the system remains stable δ time units after the broadcast was sent.

Let g_0 be the group with the highest id among the groups joined by processors in P_i at time t_1; let S be the set of processors in P_i joined to g_0 at t_1. Let g_0' be the group with the second highest id, among the groups joined by processors in P_i at time t_1, and let S' be the set of processors in P_i joined to g_0' at t_1. If no such g_0' exists (all processors in P_i are joined to g_0 at t_1) we define S' to be the empty set.

Lemma 1:

Let $t_2 = t_1 + \max(\pi + (n+1)\delta, \mu+\delta)$. If the system is stable in $I = [t_0, t_2]$ then either

(1) at time t_1 all processors in P_i are joined to g_0 and all members of g_0 are in P_i (i.e., $S = P_i = \text{members}(g_0)$), or

(2) a group $g_1 \geq g_0$ is created between t_0 and t_2.

Proof:

Case 1: $S \neq \text{members}(g_0)$

Then there must be a processor p $\in$ S with its left neighbor q $\notin$ S.

Case 1a: p receives an I-am-alive message with group id g_0 from q between t_1 and t_2.

Then q must be in P_i and q must have joined g_0 after t_1. Then q must have received a broadcast message about g_0. Then g_0 was created after t_0. In this case Condition (2) of the Lemma is satisfied for g_1=g_0.

Case 1b: p does not receive any I-am-alive message with group id g_0 from q between t_1 and t_2.

Then p will time out waiting for an I-am-alive message from q and create a new group $g_1 > g_0$ by time $t_1 + \pi + n\ \delta$. In this case Condition (2) of the Lemma is satisfied.

Case 2: S = members(g_0), S' = $\emptyset$

Then S = P_i. In this case Condition (1) of the Lemma is satisfied.

Case 3: S = members(g_0), S' $\neq \emptyset$, S' $\neq$ members(g_0')

Then there must be a processor p $\in$ S' with its left neighbor q $\notin$ S'.

Case 3a: p times out waiting for an I-am-alive message from q.

Then p will create a new group by $t_1 + \pi + n\ \delta$. Every processor in P_i will receive the new-group message from p by $t_1 + \pi + (n+1)\delta$. The first member of g_0 receiving the message will create a new group g_1, since it learned new information (p is alive). In this case Condition (2) of the Lemma is satisfied.

Case 3b: q in P_i and q joins g_0' after t_1.

Then q must have received a broadcast message about g_0'. Then g_0' was created by some p' in P_i after t_0. Then every processor in P_i will receive the new-group message from p' by t_1+δ. The first member of g_0 receiving the message will create a new group g_1, since it learned new information (p' is alive). Again, Condition (2) of the Lemma is satisfied.

Case 4: S = members(g_0), S' $\neq \emptyset$, S' = members(g_0').

This is possible only if g_0 and g_0' are disjoint. Hence at least one of the two must be a minority group.

Case 4a: The leader of g_0' is still joined to g_0' at time t_1+μ.

The leader of the minority group will send a probe message by time t_1+μ. This message is received by the leader of the other group by t_1+μ+δ. At this time the other leader will create a new group g_1.

Case 4b: The leader of g_0' joins some other group by time t_1+μ.

Analogous to case 3b above.

In case 2, condition (1) of the lemma is satisfied. In all other cases condition (2) is satisfied. □

If condition (1) of Lemma 1 is satisfied our timeliness conditions are already satisfied at t_1 and we are done. Therefore, we only need to consider the case where condition (2) is satisfied. In what follows we will therefore assume that there is a group g_1 created between t_0 and t_2 with an id at least as high as any group joined by a processor in P_i at t_1.

Lemma 2:

Let $t_3 = t_2 + \delta$. By t_3 all processors are joined to a group created after t_0.

Proof:

Let p be in P_i. By t_3 p must have received the new-group message for g_1. Let g be the group p is joined to when it receives this message. If $g > g_1$ then $g > g_0$; hence g must have been created after t_1. In this case p is already joined to a group created after t_0. Otherwise, p will either join g_1 or create and join a new group with a higher id. □

Lemma 3:

There will be no spurious timeouts after t_3.

Proof:

Assume otherwise, i.e., p learns that q is not connected. Let g be the group p is joined to when it "detects" q's failure. By Lemma 2 the group g was created after t_0. Hence q must have received the new-group message for g no more than δ time units later than p. There are only two possible reasons why q would not send the I-am-alive message that p expects:

1. q did not join g because it was joined to a group g' > g when it received the new-group message for g. By Lemma 2, g' was created after t_0. Hence p must have received the new-group message for g' before its MissingNeighbor timer ran out. Then p would not have detected that it was not connected to q.
2. q was joined to g, but left g and joined a different group g' before its SendIAmAlive timer ran out. Then g' > g; hence g' was created after t_0; Again, p must have received the new-group message for g' before its MissingNeighbor timer ran out. □

Lemma 4:

Let p, q be in P_i. If processor p creates or joins a group g after t_3 and q is not a member of g, then there must be a group g' $\geq g_1$ created between t_0 and t_3, such that q is not a member of g'.

Proof:

Assume there is no such group g', i.e., all groups created between t_0 and t_3 contain q as a member. By Lemma 2 this implies that at t_3 all processors are joined to some group containing q. There are only two reasons why p would create or join a group not containing p:

1. p times out waiting for an I-am-alive message from q. By Lemma 3 that is not possible after t_3.
2. p receives a new-group or probe message with a group g" that excludes q. By assumption, g" must have been created before t_0. A new-group message would

have reached p by time $t_0 + \delta$; a probe message would have reached p by time $t_0+\delta+\mu+\delta$. Since $t_0+\delta+\mu+\delta < t_3$ this is not possible. □

Lemma 5:

Let $t_4 = t_3 + 2\delta$. At t_4 every processors in P_i is joined to some group with a membership that is a superset of P_i.

Proof:

Assume otherwise, i.e., there are p, q in P_i such that at time t_4 p is joined to a group not containing q. Lemma 4 implies that there must be a group created between t_0 and t_3 that does not contain q. Among all such groups, let g be the one with the highest value of timestamps(q). Processor q will receive the new-group message for g by time $t_3 + \delta$. At this time q will create a new group g' containing q with a higher value for ticks(q). By time $t_3 + 2\delta$ processor p will receive the new-group message for g' from q. After this point p would create or join a group excluding q only if

1. p times out waiting for an I-am-alive message from q. By lemma 3 this is not possible.
2. p receives a new-group or probe message with a group not containing q and a value of timestamps(q) greater than or equal to the one associated with g'. But since q is the only processor that increments ticks(q) this is not possible. □

Corollary:

The one phase protocol satisfies the condition for *Bounded Join Delays* for $J = t_4 - t_0 = \max(\pi + n\ \delta, \mu) + 5\delta$.

Lemma 6:

Let $t_5 = t_4 + (n - 1)(\pi + (n+1)\delta)$. At t_5 every processor in P_i is joined to a group with a membership equal to P_i.

Proof:

There can be at most (n − 1) processors not in P_i. It takes at most (n − 1) timeout periods plus new group creations until all of these have been detected to be disconnected from P_i. After that a group with membership equal to P_i must have been created. □

Corollary:

The one phase protocol satisfies the condition for *Bounded Partition Detection Delays* for $D = t_5 - t_0 = \max(\pi + n\ \delta, \mu) + 5\delta + (n - 1)(\pi + (n+1)\delta)$.

B.7 Properties of the Three-Round Protocols

B.7.1 Common Safety Properties

(Sd), (Sj) *Legitimacy of Processor Deletions and Additions*: In the three-round protocols, the membership of any proposed group g is determined by the group creator r: r sends a new-group message to all processors and includes in members(g) all processors that reply within 2 δ time units. Therefore, a processor q that was disconnected from r during this time interval cannot be included in members(g). Similarly, if a processor q is connected to r, q will be included in members(g) whenever r does not abort the creation of g (a group g is aborted if no processor joins it). To see why this is so, consider that r and q are connected and that q receives a (new-group,g) message from r. If the highest group proposal wgid seen so far by q is smaller than g, q replies positively to r's invitation, so r will insert q in members(g). If wgid at q is higher than g (because some processor r' attempts to create a group with id wgid and q has already learned about this proposal), q replies to the new-group message of r by relaying the (new-group,wgid) message from r'. Since q and r are connected, this message will be received by r before its MissingAccept timer expires, so r will abort the creation of g. Therefore, requirements (Sd) and (Sj) are satisfied by choosing processor r to be the creator of g.

(Ss) *Join Synchronization*: In the three-round protocol, the creator c of a group g joins g only after having received accept messages from all members of g. These leave their previous groups when they receive a new-group message for g from c. Thus, no member of g can be joined to a group with a lower id at the time the first member (i.e. c) joins g.

Correctness of the D' waiting rule for unique leadership:

The proof is by contradiction. Assume that at time t there are two leaders p_1 and p_2 joined to majority groups g_1 and g_2. Since for a majority group there can be at most one leader, $g_1 \neq g_2$. Without loss of generality, assume $g_1 < g_2$. By the waiting rule, p_1 and p_2 were joined to g_1 and g_2, respectively, at t' = t − D'. Let q be a member of both g_1 and g_2. By t', q must have accepted to join g_2 (q must have received the new-group message for g_2 and must have replied "accept", before p_2 could join g_2). According to the three round protocol, once a process accepts a group g, it can only accept or join groups with higher id. Thus q cannot join g_1 after t'. If q was joined to g_1 earlier, it must have left g_1 by t'. It follows that q is not joined to g_1 throughout I = [t', t]. But then p_1 must either have crashed, detected a performance failure, or, by property (Ti), have left g_1 by t.

B.7.2 Safety Properties of the Three-Round Protocol with Partition Notification

(Sp) *Partition Notification*: The three-round protocol with partition notification satisfies this requirement, because each member q of a group g includes its predecessor group pred(g,q) in the Accept message it sends to the group creator, and the group creator includes all of the collected values in the Join message it sends to the members of g.

B.7.3 Safety Properties of the Three-Round Majority Protocol

(Sp') *Predecessor Notification for Majority Groups*: Each processor q joining a majority group saves the id of that group in the stable variable Pred(q). When a new majority group creator p receives the accepts for some new majority group g, p computes Pred(g) as the maximum of all Pred(q) for q $\in$ members(g) and then sends Pred(g) in the join message to all members of g. Thus, requirement (Sp') is satisfied.

(So) *Linear history of completed majority groups*:

We need to prove that for any distinct, completed majority groups g_1 and g_2 one must be an ancestor of the other. Without loss of generality, assume that $g_1 < g_2$. We will show that g_1 must be an ancestor of g_2. We begin with the following technical lemma.

Lemma 1:

Let g' be a majority group joined by at least a member such that $g_1 <$ g'. Then $g_1 \leq$ Pred(g').

Proof:

Let g" = Pred(g') and q$\in$members(g')$\cap$memb(g_1). Since g' was joined by at least a member, all members (including q) must have accepted to join g'. Let h be the last majority group q was joined to when it accepted to join g'. Then h was one of the candidates among which the creator of g' chose g" = Pred(g'). By definition of Pred: h$\leq$ Pred(g') (1). Since q is a member of g_1 and g_1 is complete, q must have joined g_1. Since we assumed $g_1 <$ g', q must have joined g_1 before accepting to join g'. Therefore $g_1 \leq$ h (2). Now (1) and (2) imply $g_1 \leq$ Pred(g'). $\Box$

We proceed to prove our (So) claim by contradiction: we will assume that g_1 is *not* and ancestor of g_2 and derive a contradiction.

Lemma 2:

If $g_1 < g_2$ and g_1 is not an ancestor of g_2, then for every g: if g is an ancestor(g_2) then $g_1 <$ g.

Proof: By induction on the distance between g and g_1 in the graph of the Pred relation:

- Induction Base: distance = 1, i.e., g = Pred(g_2). By Lemma 1 with g' = g_2: $g_1 \leq$ g. By hypothesis g_1 is not an ancestor of g_2; hence g $\neq g_1$; it follows that $g_1 <$ g.
- Induction Step: distance = n, where n>1. Let g'=Pred$^{(n-1)}(g_2)$, i.e., g' is the the

ancestor of g_2 at distance (n-1) in the graph of the Pred relation. By induction hypotheses, our claim holds for g', that is, $g_1 < g'$. Let g=Pred(g'). By Lemma 1: $g_1 \leq g$, but since g_1 is not an ancestor of g_2, g_1 cannot be equal to g. Therefore, $g_1 < g$. □

Now, the initial group with id (0,0) is by definition an ancestor of all majority groups, including g_2. Thus, by Lemma 2, it follows that $g_1 < (0,0)$. Since this is impossible, we conclude that the assumption that g_1 is *not* an ancestor of g_2 is false, i.e., g_1 must be an ancestor of g_2.

B.7.4 Timeliness Properties of the Three-Round Protocol

(Tj), (Td) *Bounded Join and Partition Detection Delay*: Assume that after a number of failures the system has stabilized at time t_0. We will show that if the system remains stable for sufficiently long time, all processors in one physical partition will eventually join a common group with a membership equal to the set of processors in that partition. Thus, (Tj) and (Td) will be satisfied.

We prove our claim in three steps. First, we show that if, after the system has stabilized, all processors in physical partition P_i do not already belong to a common group with membership equal to P_i, then a new group will be created (Lemma 1). Next we prove that every group that was successfully created after the system has stabilized will have membership equal to P_i (Lemma 2). Finally, we show that one of these groups will actually be joined by all its members.

Let $t_1 = t_0 + 4\delta$. Let g_0 be the group with the highest id among the groups joined by processors in P_i at time t_1; let S be the set of processors in P_i joined to g_0 at t_1. Let g_0' be the group with the second highest id, among the groups joined by processors in P_i at time t_1, and let S' be the set of processors in P_i joined to g_0' at t_1. If no such g_0' exists (all processors in P_i are joined to g_0 at t_1) we define S' to be the empty set.

Lemma 1:

Let $t_2 = t_1 + \max(\pi + (n+4)\delta, \mu + \delta)$. If the system is stable in $I = [t_0, t_2]$ then either

(1) at time t_1 all processors in P_i are joined to g_0 and all members of g_0 are in P_i: $S = P_i$ and P_i = members(g_0), or

(2) a group $g_1 \geq g_0$ is created between t_0 and t_2.

Proof:

Case 1: $S \neq$ members(g_0)

Then there must be a processor $p \in S$ with its left neighbor $q \notin S$.

Case 1a: p receives an I-am-alive message from q between t_1 and t_2.

Then q must be in P_i and q must have joined g_0 after t_1. If the system is stable, no more than 3δ time units can elapse between the decision to create a group and the moment the last member receives the Join message for this group. Thus, if q joins g_0 after t_1, g_0 must have been created after $t_1 - 3\delta$, which is greater than t_0. In this case Condition (2) of the Lemma is satisfied for g_1=g_0.

Case 1b: p does not receive any I-am-alive message from q between t_1 and t_2.

Then p will time out and will initiate the creation of a new group $g_1 > g_0$ by $t_1 + \pi + n\ \delta$. In this case Condition (2) of the Lemma is satisfied.

Case 2: S = members(g_0), S' = $\emptyset$

Then S = P_i. In this case Condition (1) of the Lemma is satisfied.

Case 3: S = members(g_0), S' $\neq \emptyset$, S' $\neq$ members(g_0')

Then there must be a processor p $\in$ S' with its left neighbor q $\notin$ S'.

Case 3a: p times out waiting for an I-am-alive message from q.

Then p will initiate the creation of a new group g by $t_1' = t_1 + \pi + n\delta$. If g $> g_0$, then Condition (2) of the Lemma is satisfied. If g $< g_0$, and if no other processor in P_i creates a group g' $> g_0$ between t_1' and $t_1' + 4\delta$, the following will happen: the first member of g_0 receiving the NewGroup message for g from p will, instead of accepting g, forward the NewGroup message for g_0 to p. Processor p will receive the forwarded message by time $t_1'+2\delta$. At this time, p will abort the creation of g, and instead send an accept message for g_0 to the creator of g_0. Since p $\notin$ members(g_0), p will never receive a Join message for g_0. By time $t_1'+4\delta$, p's MissingJoin timer will expire, and p will create another group g' $> g_0$. In either case Condition (2) of the Lemma is satisfied.

Case 3b: q is in P_i and q joins g_0' after t_1.

If the system is stable, no more then 3δ time units can elapse between the decision to create a group g and the moment the last member of g receives the Join message for g. Hence g_0' must have been created by some processor r $\in P_i$ after time $t_1 - 3\delta = t_0 + \delta$. But then every processor in P_i must have received the NewGroup message for g_0' from r. Any processor in P_i that received a NewGroup message for g_0 must have received this message after receiving the NewGroup message for g_0' from r. Otherwise, it would, instead of accepting g_0', have forwarded a NewGroup message for g_0 to r, and r would have aborted the creation of g_0'. Therefore, g_0 could not have been created more than δ time units before g_0'. Hence g_0 must have been created after t_0, and Condition (2) of the Lemma is satisfied for g_1=g_0.

In either case Condition (2) of the Lemma is satisfied.

Case 4: S = members(g_0), S' $\neq \emptyset$, S' = members(g_0').

This is possible only if g_0 and g_0' are disjoint. Hence at least one of the two must be a minority group.

Case 4a: The leader of g_0' is still joined to g_0' at time $t_1+\mu$.

The leader of the minority group will send a probe message by time $t_1+\mu$. This message is received by the leader of the other group by $t_1+\mu+\delta$. At this time the other leader will create a new group g_1.

Case 4b: The leader of g_0' joins some other group by time $t_1+\mu$.

Analogous to case 3b above.

In case 2, Condition (1) of the lemma is satisfied. In all other cases Condition (2) is satisfied. □

If Condition (1) of Lemma 1 is satisfied, our timeliness conditions are already satisfied at t_1 and we are done. Therefore, we only need to consider the case where condition (2) is satisfied. In what follows we will therefore assume that there is a group g_1 created between t_0 and t_2 with an id at least as high as any group joined by a processor in P_i at t_1. Next we will show that any such group g_1 must include every member in P_i, if at least one processor in P_i joins this group.

Lemma 2: Let g be a group created after t_0 by a processor r in physical partition P_i. If at least one processor joins g, and if the system is stable for 2δ time units after r decided to create the group, then members(g) = P_i.

Proof:

If the system is stable for 2δ time units, then every processor in P_i will receive the NewGroup message, and r will receive a reply from every processor in P_i before its MissingAccept timer expires. Furthermore, no processor outside of P_i will receive or reply to the NewGroup message. If all processors in P_i reply with an "accept" message, then the membership of g computed by r will be P_i. If one or more processors reply by relaying a NewGroup message with a group id greater than g, then r will abort the group creation, and no processor will join g. □

Finally, we have to show that one of the groups created within a certain time after t_0 will actually be joined by all its members.

Let g be the group with the largest id created between t_0 and t_2, let r be the processor that created g, and let t $\in [t_0, t_2]$ be the time at which r did so. By time $t+\delta$ all processors in P_i will have received the NewGroup message for g_1 from r. At that time, every processor in P_i that was joined to a group with id less than g will have left that group, and if any processor in P_i is joined to any group at that time, it must be joined to a group created between t and $t+\delta$ (either to g or to a group with id greater than g created after t). Therefore, no MissingNeighbor timeouts will occur in P_i between $t+\delta$ and $t + \pi + n\ \delta$. Also, there will be no probe messages causing new groups to be created in P_i after t+δ, because probe messages are not sent to group members, and by Lemma 2, all processors in P_i are members of groups created after t. Therefore, no new groups will be created in P_i between t+δ and t+π+δ.

Let g' be the group with the largest id known to processors in P_i at time t+δ. This group must have been created between t and t+δ. Therefore, all processors in P_i will have received the NewGroup message for g' by time t'=t+2δ. Since no new groups are created between t+δ and t + π + n δ, g' will still be the group with the largest id at time t'. Therefore, every processor in P_i will accept to join g'. It takes at most δ more time units until the creator of g' has received all accept messages and sends out Join messages. By time t'+2δ every processor in P_i will have received the Join message and joined the group.

We conclude that all processors in P_i must be joined to a common group with membership equal to P_i by time $t'+2\delta = t+4\delta \leq t_2+4\delta = t_0 + 9\delta+\max(\pi+(n+3)\delta, \mu)$. Therefore, the two three-round protocols satisfy requirements (Tj) and (Td) for

$J = D = 9\delta + \max(\pi + (n+3)\delta, \mu)$.

B.8 Properties of the Five-Round Protocol

B.8.1 Common Safety and Timeliness Properties

The five-round protocol has essentially the same structure as the three-round protocols; the difference is that when a majority group is created, an additional round of "prepare" messages are sent after the group creator has collected "accept" messages from all members, and the creator waits for "prepared" replies from all members before sending out Join messages.

In fact, we can express this similarity more clearly, using the following terminology. We say that a newly created majority group g becomes *preliminary group* when the creator of g has collected all accept messages and begins sending out prepare messages for g; we say that a processor p "joins a preliminary group g", if p is the creator of g and p is just sending out prepare message for g, or if p is a member of g and p has just received a prepare message for g. We call a group *committed* if the creator has joined the group (i.e., after collecting "prepared" replies from all members) and is sending out Join messages to the other members.

For preliminary groups the five-round protocol behaves exactly like the two three-round protocols.[10] If follows that all of the common safety and timeliness properties of the three-

[10] In fact, this is the reason why the protocol calls for a processor to start its group related timers (IAmAlive,

round protocols apply directly to the preliminary groups created by the five-round protocol. From this observation it is straightforward to see that the same properties also hold for the committed groups created by the five-round protocol, if an additional term of 2δ is added to the time constants J, D, and D' to take into account the time required for the extra two rounds of messages before a preliminary group becomes a committed group.

All that remains is to show that the five-round protocol achieves agreement on a unique history of majority groups (So'), which is the one property that distinguishes the five-round protocol from the three-round protocol for majority groups.

B.8.2 Agreement on Group History

We first examine the version of the protocol that relies on infinite storage to keep a history of all majority groups.

Observation 1:

All processors joined to the same preliminary majority group g have the same local history (i.e., the value of local variable hist is identical) until they join another preliminary majority group.

Proof:

A processor updates its local history (variable hist) only when it sends or receives a "prepare" message for a new majority group g. According to our definition, that is exactly the time at which it joins the preliminary group g. Therefore, the value of the variable hist does not change as long as a processor is joined to the same preliminary group. Furthermore, at the time a processor joins a preliminary group g, i.e., when it sends or receives a "prepare" message for g, it replaces its local history by the one contained in the "prepare" message. Therefore, all processors joined to the same preliminary group g, must have the same value of hist. □

Observation 2:

Whenever a processor p joins a new preliminary majority group g it updates its local history with a history h containing group g as its *last* element.

Proof:

By our definition, p joins majority group g when it receives a prepare message from the creator of g. At that time, p replaces its local history with the history h sent by the

MissingNeighbor, and SendProbe timer) as soon as it receives the Prepare message for a new majority group, rather than waiting for the Join message, because that is the time when it joins the preliminary group (see procedure Prepare in the pseudo code).

creator of g in the "prepare" message. But the creator of g constructed h by appending g to some other history. □

Lemma 1:

Let h = (g_0, g_1, g_2, ..., g_n) be the local history of processor p at some time t (i.e., h is the value of the variable hist at processor p at time t). Then all of the g_i are preliminary majority groups, and g_i is the predecessor of g_{i+1} under the definition of Pred for the three round majority protocol applied to the preliminary groups formed during the execution of the five round protocol.

Proof:

In the five round protocol, the only way in which histories grow is during group creation, when the creator q of a new group g appends g to one of the histories it received in "accept" messages from the members of the new group g. By observation 2, each history that q receives in an "accept" message has as its last element the last majority group g' that the sender of the message was joined to when it received the NewGroup message for g. Among those histories, q picks the history h with the largest g' as its last element. But then g' satisfies the definition of the predecessor of g. Therefore, whenever the group creator adds a new group g to a history, it adds it to a history ending with the predecessor of g. □

Lemma 2:

For all p, q: the stable group history at p is a prefix of the stable group history at q, or vice versa.

Proof:

Proof by contradiction: assume the stable histories of processors p and q are identical up to g_0, but then p's stable history continues with group g_1, and q's stable history continues with g_2.

Groups become part of the stable history of processor p when it receives a "join" message following a "prepare" message for some new group r. At that time all of the groups in p's history ending with r become stable. Let r_1 be the group id contained in the join message that caused p to make g_1 part of its stable history. At that time, p's history must have looked like this:

$$H_p = (\ldots, g_0, g_1, \ldots, r_1),$$

where r_1 is possibly — but not necessarily — equal to g_1. Similarly, at the time when q made g_2 part of its stable history, it must have received a "join" message for some group r_2, and its history must have looked like this:

$$H_q = (\ldots, g_0, g_2, \ldots, r_2)$$

By Lemma 1, each group in H_p is the predecessor of the group immediately to the right of it. By definition, a group cannot have more than one predecessor. Therefore, H_p contains exactly all of the ancestors of r_1. Similarly, H_q contains all ancestors of r_2. Since $r_1 \notin H_q$ and $r_2 \notin H_p$, it follows that r_1 cannot be a predecessor of r_2 and that r_2 cannot be a predecessor of r_1.

However, remember that processor p received a join message from the creator of r_1. That implies that the creator of r_1 must have received "prepared" messages from all members of r_1, which means that all members of r_1 must have received a "prepare" message from the creator of r_1. But by our definition of *preliminary* groups, this means that every member of r_1 must have joined r_1 as a preliminary group. Similarly, every member of r_2 must have joined r_2 as a preliminary group. Now we can apply property (So) of the Three Round Protocol (linear history of completed majority groups) to the preliminary groups r_1 and r_2. It follows that r_1 must be an ancestor of r_2. But this contradicts our previous observation that r_1 and r_2 cannot be ancestors of each other.

We conclude that our original assumption is false, i.e., the stable histories at p and q cannot diverge. □

Since a processor joins a majority group only after it becomes part of its stable history, it follows that all committed majority groups must be part of the same linearly ordered history. Hence the protocol satisfies requirement (So').

The version of the five round protocol analyzed above requires infinite storage, because groups are added to the history, but never removed. To avoid this problem, histories can periodically be pruned by removing groups from the history as shown in the pseudo code. Since a group is removed only after it is known that it has been joined by all its members, this change does not affect the correctness of the protocol.