Reaching Agreement on Processor Group Membership in Synchronous Distributed Systems[*]

Flaviu Cristian
Computer Science and Engineering
University of California, San Diego
La Jolla, California 92093-0114
flaviu@cs.ucsd.edu

Abstract

Reaching agreement on the identity of correctly functioning processors of a distributed system in the presence of random communication delays, failures and processor joins is a fundamental problem in fault-tolerant distributed systems. Assuming a synchronous communication network that is not subject to partition occurrences, we specify the processor-group membership problem and we propose three simple protocols for solving it. The protocols provide all correct processors with consistent views of the processor-group membership and guarantee bounded processor failure detection and join delays.

Key words: Communication network – Distributed system – Failure detection – Fault tolerance – Real time system – Replicated data

1 Introduction

When designing a computing service that must remain available despite component failures, a key idea is to *replicate* service state information at several servers running on distinct processors. The service state typically consists of the server-group membership, that is, the set of all correctly functioning servers that cooperate to provide the service, and service specific state information, such as the queue of service requests accepted and not yet completed, the current assignment of work to various active servers, and the state of the physical resources used to provide the service. Server replication lets a service be *highly available* despite processor or server failures. Indeed, once the surviving servers detect the failure of

[*]published in Distributed Computing, Vol. 4, pp. 175-187, 1991

some of their peers, they have enough state information to redistribute among themselves the workload handled by the failed servers. However, replication creates problems that do not exist in non-redundant systems. Perhaps the most difficult new problem is *achieving agreement* among replicated servers on a global service state despite random information propagation delays, component failures, and server joins. Such agreement is necessary if the goal is to make a replicated server-group behave as a single logical server rather than as a group of autonomous processes.

Earlier papers have investigated fault-tolerant protocols for agreeing on service specific global states in replicated server-groups whose membership can decrease but not increase [CASD85, Cri90]. This paper focuses on the problem of reaching agreement on the membership of dynamic server-groups that can shrink with failures and grow with joins. We break down this goal into two sub-goals. First, we show how to achieve agreement on the identity of all correctly functioning processors that can execute server processes. We refer to this first problem as the *processor-group membership problem.* Second, assuming the processor-group membership problem solved, we show how to solve the *server-group membership problem.* The solution to the latter problem shows how to maintain agreement on the global state of a server-group when server joins cause the group membership to increase.

We begin by describing our system model and failure assumptions and by specifying the processor-group membership problem. The clear isolation and precise specification of this problem is a key contribution of the paper. Another contribution is the description of three protocols for solving the problem. The first protocol provides fast processor failure detection but can require significant message traffic overhead, even when no failures occur. To reduce this overhead, we derive two other protocols which have a (provably) minimal message overhead in the absence of failures, but that provide longer failure detection delays. The simplicity of these three protocols compares favorably with the complexity of earlier known solutions to the membership problem [BJ87, Car85, CM84, ASC85, Wal82].

The membership problem is a fundamental problem of distributed computing, like routing, clock synchronization, atomic broadcast, or atomic commit, in the sense that once solved, it allows easy solutions to other important problems encountered when designing fault-tolerant distributed applications. To illustrate this point, we show how a processor membership service helps solve the server-group membership problem, the problem of ensuring high availability of computing services in a distributed system, and the problem of precisely defining the scope of server-group communication. We also examine some optimizations and possible extensions to our three algorithms. We conclude by comparing our approach with other published approaches.

2 System model and failure hypotheses

We consider a system consisting of distributed server processes running on processors linked by a physical network. There is total order "$\prec$" on the set P of processor identifiers. Each processor consists of hardware (CPUs, I/O controllers, storage, clock, and so on) and software (operating system, communication subsystem, etc). The operating system supports process execution. The communication subsystem accepts messages from, and delivers messages to, processes, manages message queues, and drives the physical network links. Collectively, the communication subsystems provide the distributed processes that run at various processors the abstraction of a *communication network*. The communication subsystems are the *nodes* of this network.

2.1 Synchronous Communication Network

Among the communication services that we assume the network provides, the simplest is a *datagram* service. This service provides a cheap means for any process to send a message m to any other process. This works as follows: the source process entrusts m to a network node p, the node p sends m along a single physical route to another node q, and q delivers (that is, makes available) m to the target process. We do not make any assumption about the topology of the network. It can be point-to-point or based on broadcast channels, such as token rings or busses. If it is not fully connected, we assume the nodes implement message forwarding. Neither do we make any assumption on datagram message transmission protocols. For instance, a datagram message might be broken down into, and transmitted as, a sequence of datagram packets, and lost packets might be retransmitted a finite, a priori known, number of times before a message is considered lost. Between the acceptance of a datagram message m by a source node p and the moment m is delivered to the destination process by the target node q, there is an arbitrary random delay. To avoid waiting forever for packets that will never arrive, there is a need to decide on a timeout delay **d**, such that a datagram message that travels more than **d** time units between node p and node q is considered lost. Such timeout delays are introduced by system designers to prevent situations in which a process A waits forever for a message from another process B that will never arrive (for example, because of B's failure). Datagram timeout delays are established by studying statistics about network behavior under various load patterns, so as to ensure that node-to-node datagram message transmission delays are smaller than d with very high probability.

The adoption of a bound d on datagram message delays divides datagram service behaviors in two classes: correct behaviors and failures. The datagram service provided by two network nodes p and q is *correct* if any datagram message m accepted at p for a target process at q is delivered to that process by q within **d** time units. A datagram *omission failure* occurs if m is never delivered at q, while a *performance failure* occurs if m is delivered after **d** time units [CASD85]. There can be many causes for such failures. One is that the physical packets used to transmit messages were (repeatedly, if low level re-transmissions are implemented) corrupted and discarded. Another is that a buffer overflow occurred at the target node. Yet another cause is a slow processing of datagram packets at the source or target communication subsystems, due to randomly occurring excessive network load conditions.

We assume that the datagram service can only suffer omission or performance failures.

In addition to the datagram service, we assume the communication network also provides a *diffusion* service. A node p *diffuses* a message to another node q by sending diffusion packets in parallel on all physically independent routes between p and q. In point-to-point networks, diffusion can be implemented as discussed in [CASD85], while in networks based on redundant broadcast channels (e.g. rings, busses) diffusion can be implemented as reported in [Cri90]. We assume that there is some arbitrary, but fixed bound F on the number of communication components (nodes and links) that can be faulty during a diffusion, and that the network possesses enough *redundant* independent physical links between any two pairs of nodes (p, q), so that a node q always receives at least one copy of each packet diffused by another node p despite up to F faulty communication components. We also assume that the rate at which diffusions are initiated is bounded, that this rate is smaller than the rate at which nodes can receive and process diffusion packets, and that the communication subsystems are properly dimensioned to provide enough buffer capacity as well as bounded processing delays for diffusion packets under worst case network load conditions (typically, diffusion packets have higher priority than any other type of packets). Under the above assumptions, any message m diffused by a correct node p is always received and processed at another correct node q within a known *network delay* N (which is a function of F, network topology, and the worst case transmission and processing delay for diffusion packets [CASD85, Cri90]). We call a communication network diffusion-synchronous (or simply *synchronous*) if it ensures that any diffusion message entrusted to a correct node is received and processed by all correct nodes within N time units. The characteristic property of a synchronous network is that *communication partitions* which can prevent correct nodes from diffusing information to each other within N time units, do not occur. Thus, in a synchronous network, the only reason why two nodes cannot communicate by using the network message diffusion service is the failure of at least one of them.

A synchronous communication network enables processor clocks to be synchronized. We assume that processor clocks are reliable, monotonic (successive readings yield strictly increasing values), run within a linear envelope of real-time and show at any real-time values within a known constant maximum deviation ε, which is a function of the network delay N, and the re-synchronization period [CAS86]. Synchronized clocks allow us to reason in terms of global system clock time, instead of real time. Throughout this paper "time" means "clock time". For instance, phrases such as "message m takes at most **d** time units between p and q" and "processes p, q broadcast message m at time T" mean "the delay experienced by m between p and q as measured on any processor clock is at most **d**" and "p broadcasts m when its local clock displays time T and q broadcasts m when its clock displays T", respectively.

A synchronous communication network also enables the implementation of a synchronous *atomic broadcast* communication service [CASD85, Cri90]. Synchronous atomic broadcast protocols ensure, for some time constant **D** that depends on N and ε, the following properties. If a node attempts to broadcast a message m at time T, then at time $T + \mathbf{D}$ either all correct nodes deliver m or none of them delivers m (atomicity). All messages delivered are delivered in the same order at each correct node (order). If the sending node of m is correct, then all correct nodes deliver m at $T + \mathbf{D}$ (termination). Since in this paper atomic broadcast is the only primitive used for broadcasting information, when we say "broadcast", we mean "atomic broadcast".

Although the description of our solutions to the membership problem relies on the existence of lower level clock synchronization and atomic broadcast network services, it should be clear these lower level protocols are just used as "structuring tools" to shorten our presentation. The only essential assumption made is that of a synchronous communication network. Protocols for implementing these lower level services in synchronous communication networks in the presence of an arbitrary, but fixed number F of faulty components that can suffer crash, omission and performance failures have been described in previous publications: [CAS86] describes a diffusion based clock synchronization protocol, while [CASD85] and [Cri90] describe diffusion and clock synchronization based atomic broadcast protocols for point-to-point and broadcast networks.

The reliable communication achieved by using diffusions to broadcast information comes with a price. While datagram messages are cheap, the diffusions used for atomic broadcast can be expensive. For example, in a point-to-point network, each atomic broadcast costs approximately $n \times d - n + 1$ datagram messages, where n is the number of correct nodes and d is the average node degree, i.e. number of neighbors of a node [CASD85]. Similarly, the protocols of [Cri90] require $F \times n + 1$ datagram messages to atomically broadcast information despite up to F processor performance failures (when only processor crash failures can occur, the number of messages in the absence of failures is $F + 1$).

2.2 Task Scheduling Rules

The datagram, clock synchronization and atomic broadcast services provided by the communication network are accessible to *processes* running under operating system supervision. Such processes can be structured into several parallel *tasks* (or threads). To explicitly start a task A with input parameters X at time T, the operating systems running on network processors provide a "*schedule* (A, X) *at* T" primitive. A correct operating system interprets an invocation of this primitive as follows. If the invocation is made at a time $T\prime$ after the *task start deadline* T, A is not started. Otherwise, if $T\prime \leq T$, A is started with input parameters X at a time in the interval $[T,\ T + \eta)$, where η, the maximum task *scheduling delay* is a positive constant. Several invocations of *schedule* (A, X) at T with identical parameters X made at times smaller than T result in just one start of A with parameters X. Task execution can be *suspended*, for instance, when a task waits for messages. Let B be a task suspended because it waits for a message m. If m is delivered by the communication subsystem at time T, a correct operating system interprets T as a *start deadline* for B, i.e. the system starts B in the interval $[T,\ T + \eta)$. We assume that tasks scheduled to be started at different times are started *in the order* of their start deadlines. For example if tasks A and B of a process are startable in the intervals $[T,\ T + \eta)$ and $[T\prime,\ T\prime + \eta)$, respectively where $T < T\prime$, then A is started before B. If two tasks A and B have a common start deadline T, we assume they are started in some arbitrary, but fixed, order, known to all operating system instances on all processors. For example, if tasks are uniquely identified by alphanumeric identifiers, such an order can be the lexicographic order on character strings. This is the total order on task identifiers assumed in this paper.

We assume that an operating system never violates the *ordering rules* on task start events given above. However, due to excessive load conditions, a system might experience delays longer than η between the occurrence of task start deadlines and task starts. Like the **d** constant, the η constant is determined empirically, by studying statistics on task scheduling delays under various load conditions, so as to ensure that an event which occurs at time T results in the start of the task waiting for it by time $T + \eta$ with high probability. An operating system that starts a task startable at time T at a time beyond $T + \eta$ suffers a *performance failure*. Such operating system performance failures cause task (and hence process) performance failures. A process *crash* occurs either because the underlying operating system crashes or because the execution of a process command results in an unanticipated exception that causes the underlying system to *abort* the process. An operating system crashes when the handler of an exception detected during its execution terminates with an *abort* command. The invocation of this command triggers the execution of a predefined sequence of system *restart* commands. Similarly, a process *abort* (whether explicitly invoked from a process task or by the underlying operating system) prevents the process from reacting to all events that occur process restart. After a process p crashes there is minimum restart delay R: if the crash occurs at time T, p does not resume correct operation before time $T + R$.

We assume that processes can only suffer crash and performance failures.

Since in a typical system the constants **d** and η are at least two orders of magnitude greater than the time needed to consume and process a message by a short non-interruptible task, to simplify our presentation we ignore message processing times: we assume that once a task is started, it completes in zero time units. That is, on a correct processor, a task startable at time T starts *and completes* in the interval $[T,\ T + \eta)$. (The very short non-interruptible tasks to be described later are consistent with this assumption.) Note also that the η uncertainty on task (and hence, process) scheduling delays introduces a new bound $\delta = \eta + \mathbf{d} + \eta$ on process-to-process datagram message delays. The first term represents the upper bound on the random delay between the moment a process sends a message and the moment the message is accepted by the source node, the second is the upper bound on the node to node delay, and the third is the upper bound on the delay between the target node and the target process. In a similar manner, we denote by $\Delta = \eta + \mathbf{D} + \eta$ the bound on process-to-process atomic broadcast delays.

3 The processor-group membership service

To enable any client process on any correct processor of a synchronous distributed system to use the processor membership service, this service is implemented by a group of server processes replicated on all processors of the system. As far as the processor-group service is concerned, processors are represented by the membership servers that run on them. The failure of such a server is interpreted as the failure of its underlying processor. Because of this one-to-one correspondence, we equate membership servers and processors in what follows. We denote P by the set of correct membership servers, and adopt the convention that we refer to "correct servers" simply as "servers". We use the qualifying adjective "failed" only when talking about failed servers.

The service provided by membership server $p \in P$ is as follows. When a local client process c declares an interest in knowing the processor membership, p gives c the membership of the current processor group and then notifies c of any subsequent membership changes in a timely manner. Such notifications are sent to c until either c declares that it is no longer interested in receiving them or the failure of c is reported to p by the underlying operating system. The management of the set of local clients interested in membership and the mechanics of their notification when membership changes occur are straightforward and will not be explicitly mentioned in what follows. Instead, we focus only on the difficult problem of maintaining consistent knowledge of the processor-group membership at each correct processor.

New processor-groups are created in response to either processor starts or failures. A processor start occurs either when a processor restarts after a failure or a shut-down or when a new processor is added to a system. To simplify our presentation, we equate a voluntary processor shut-down, for example for maintenance reasons, with a processor failure. At any point in real-time, we require a processor to be joined to at most one group. There exist times at which a processor might not be joined to any group, for example, between its start and the moment the processor joins its first group. To unambiguously designate the different processor-groups that might exist in time, we uniquely identify each group by a unique *processor-group identifier* g (some reasons for this requirement are discussed in section 7.3). Let G denote the set of all possible processor-group identifiers. We denote by *joined*: P $\longrightarrow$ { *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. A processor not joined to a group cannot take part in any coordinated group activity. Let

group: P $\longrightarrow$ G

be the (partial) mapping that records the group to which a processor is joined, when it is joined, i.e. if *joined*(p) then *group*(p) yields the identifier of the group joined by p. Let furthermore

members: P $\longrightarrow$ *Set-of*-P

be 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 with time. However, to keep our presentation simple we decided not to mention the time domain explicitly.

We require a processor-group membership service to satisfy the following safety requirements:

(Ss): *Stability of local views.* After a processor joins a group, it stays joined to that group until a failure is detected or a processor start occurs.

(Sh): *Agreement on history.* Let p and q be processors correct throughout a certain time interval. If during that interval, p and q are joined to a common group g_1 and the next groups joined by p and q after leaving g_1 are g_2 and g_2', respectively, then $g_2 = g_2'$.

(Sa): *Agreement on group membership.* If two correct processors p and q are joined to the same group, that is *joined*(p) & *joined*(q) and *group*(p) = *group*(q), then the two processors have the same view of the membership of that group: *members*(p) = *members*(q).

Because of this property, we let *members*(g) denote the view common to all members of a group *g*. When requirements (Sa) or (Sh) are violated, different group servers that depend on a processor membership service can perform inconsistent operations on the replicated data they manage, leading to inconsistent replicas. The easiest way to achieve (Sa) would be to set the views of all processors to the empty set. To avoid such trivial solutions, we require that a processor p that has joined a group be a member of that group:

(Sr): *Reflexivity.* If *joined*(p) then p $\in$ *members*(p).

While (Sr) rules out the trivial solution of setting all processor views to the empty set, it does not rule out the equally trivial solution of setting them all to the total set of all processor. To make the views of group members be as close as possible to reality, we require that a processor-group membership protocol satisfy some timeliness requirements. First, we require that there be an upper bound J (for join delay), on the delay that can elapse between the start of a processor j and the moment j joins a processor-group, even when other failures or joins occur concurrently with j's join:

(Tj): *Bounded join delays.* There exists a time constant J such that, if a processor j starts at time T and j stays correct until time $T + J$, then by $T + J$ the processor j joins a group that is also joined by each other processor that was correct throughout $[T,\ T + J]$.

Note that all processors that were correct between T and the join completion time are required to join *the same* group as the newcomer. In this way, any two correct processors are forced to agree on the identifier of the group resulting from the join.

Second, we require that there be a bound D (for failure detection delay) on the time needed to detect processor failures:

(Td): *Bounded failure detection delays.* There exists a time constant D such that, if a processor f joined to a group g fails at time T, then each member of g that stays correct throughout $[T,\ T + D]$ joins by $T + D$ a group g$\prime$ such that f$\notin$*members*(g$\prime$).

We require this property to hold despite any number of other failure and join events that could occur concurrently with the failure of f. The (Td) property lets surviving processors automatically distribute among themselves the workload of any departing processors within a *bounded* time. For systems that have to meet hard real-time deadlines — even when component failures and joins occur clustered in time — this requirement is essential.

Note. The requirements above imply the following *essential property*: after a processor p starts and joins a group g that has at least another member q, both p and q will see the "same" sequence of membership changes for as long as both remain correct. More precisely, if p and q stay correct from joining g until some time T and p sees a certain sequence of membership changes by time $T\text{-}m\bar{a}x(J,\ D)$, then q sees the same membership changes in the same order by time T. This property, together with the total order on the delivery of group atomic broadcast messages (that can be used by correct processors to convey system-specific state updates to the other correct processors) enable all events that affect the state of a system to be seen by all correct processors in the same order. This total event ordering can substantially simplify the programming of replicated fault-tolerant services [Lam84].

4 The periodic broadcast membership protocol

We begin our presentation of membership protocols by introducing a simple "periodic broadcast" protocol. In sketching it, we first make the simplifying assumption that η, the bound on task scheduling delays, is 0. We relax this assumption later when we present the protocol in more detail.

4.1 Sketch of the periodic broadcast membership protocol

Join handling. A membership server j that starts at time S invites the other servers to form a new group by broadcasting a "new-group" message timestamped S. This message is received by an arbitrary correct server p $\in$ P by time $V = S + \Delta$. In response to a "new-group" message, each server p $\in$ P broadcasts a "present" message that contains its identifier and indicates its willingness to join a new group. We call the set of servers that broadcast "present" messages for time V *the processor membership as of view time* V and denote by $MEMBERS(V)$ this set. The atomicity and termination properties of atomic broadcast ensure that any two servers p, q $\in$ P and are correct during $[V, V + \Delta]$ receive at time $C = V + \Delta$ the same set of "present" messages, and hence, can compute at C local membership views *members*(p) and *members*(q) equal to $MEMBERS(V)$. Since all such servers agree on V and $MEMBERS(V)$, we use V as (unique) identifier for the new group with membership $MEMBERS(V)$. After a server p computes (at C) the membership of the group V, pp leaves the group (if any) to which it was previously joined, and joins V.

Failure handling. While a server that starts can alert its peers about this event, a failed server f cannot tell the surviving members of its group that they should form a new group without it. In such a case, the passage of time is used to trigger after f's failure the formation of a new group from which f is excluded. This can be implemented by letting each member p of a group V check that, π time units after the view time V, its view of the membership is still consistent with reality. For simplicity we require $\pi > \Delta$, to let p learn the membership as of view time V before it checks it for "currentness". (It is possible to design membership protocols for $\pi \leq \Delta$, but their analysis is somewhat more complex.) One simple way of checking that the membership at time $V + \pi$ is still the same as that at time V is to let all members of the group V broadcast at *membership check time* $O = V + \pi$ "present" messages. If at *membership confirmation time* $O + \Delta$ a member p of V detects that some f$\in$*members*(p) is not in $MEMBERS(V + \pi)$, then — by the atomicity and termination properties of atomic broadcast — all surviving members q of V detect at $O + \Delta$ that f$\in MEMBERS(V + \pi)$. When a surviving member q detects such a failure, it joins a new group $V + \pi$ with $MEMBERS(V + \pi)$. Whether membership changes are detected or not, each surviving processor q schedules a new membership check time $O\prime = O + \pi$ to check again the currentness of its view of the membership. This method effectively imposes a discrete sampling of the continuously changing processor membership reality into a sequence of snapshots taken at V, $V + \pi$, $V + 2\pi$, etc.

Agreement on membership check times. Let g be a processor-group with next membership check time $O = g + k\pi$, for some integer k, and consider that a join initiated by a processor j results in a new group V, $O - \pi < V < O$. What membership check time should exist for the new group V that will be created? The check time O scheduled in the old group g or a new check time $V + \pi$? We choose the new check time. In this way, the join of j leads to the creation of an *unscheduled* check time $V + \pi$ that *cancels* the previously scheduled check time O that g members know. Our choice is motivated by the following two reasons.

```
task Membership;
var $group$: Time; $members$: set-of-P initially {};
    $joined$: Boolean initially false;
broadcast("new-group", $myclock + \Delta$);
cycle
when receive ("new-group", $V$)
do if myclock $> V$ then abort fi;
    cancel(Broadcast);
    broadcast("present", $V$, $M$)
    schedule(Broadcast, $V + \pi$) at $V + \pi - \eta$
od;
endcycle;

task Broadcast ($V$: Time);
if  myclock $> V$ then abort fi;
broadcast ("present", $V$, myid);
schedule(Broadcast, $V + \pi$) $bf\ at$ $V + \pi - \eta$;
```

First, members of g do not have to communicate to j their next scheduled check time O. Second, they do not have to worry whether this information will get to j fast enough for j to be able to broadcast "present" at time O.

4.2 Detailed description of the periodic broadcast membership protocol

Each processor membership server is structured into two concurrent tasks as shown in Figure 1. We refer to line m in figure n as (n.m.).

The Membership task is started on each processor after the local clock is first synchronized. It controls processor joins and processes "present" messages received from the communication subsystem. It also has to respond to membership inquiries from clients and deal with membership change notifications, as discussed in Sect. 3, but for simplicity the code dealing with membership inquiries and change notifications is omitted from Fig. 1.

The Broadcast task broadcasts "present" messages for scheduled membership check times. The *when* construct 1.6 – 1.15 expresses nondeterministic choice: if both alternatives are enabled one is chosen at random. After each alternative execution, a new cycle 1.5–1.16 starts. We assume that between a start and a subsequent suspension a task executes in mutual exclusion. Thus, the execution of command sequences such as 1.7–1.11, 1.13–1.15, and 1.18–1.20 is atomic with respect to synchronization. There is no requirement,however, that the execution of task commands be atomic with respect to failures i.e., a failure can interrupt or delay their execution at any time.

Programs are made independent from the processors they run on by the use of a standard function myid. When invoked from a program, this function returns the identifier of the processor on which the program runs (e.g. 1.9). To simplify our description, we assume an "intelligent" communication subsystem that batches all "present" messages broadcast for identical view times V into one message (1.12) containing V and the set $M = MEMBERS(V)$ of processors who have broadcast "present" at V, instead of delivering all such messages one by one. (It is straightforward to modify the message deliver tasks given in [CASD85] and [Cri90] to achieve this.) The tests (1.7, 1.18) detect task performance failures. The invocation of **abort** commands (1.7, 1.18) transforms such performance failures into membership server crash failures. The **cancel**(A) command (1.18) cancels all previously scheduled starts of a task A.

The removal of the null η assumption adopted for simplicity in section 4.1 introduces two kinds of complications. A minor one is in the computation of task start times and delays. For example, we have taken into account the existence of the η uncertainty on task starts by setting $V + \pi - \eta$ as start deadline for Broadcast (1.10, 1.20) if we want this task to broadcast a "present" message *by* time $V + \pi$. Similarly, if we want each member of a group V to broadcast its first "present" message for check time $V + \pi$ after it learns the membership of V, the initial requirement of $\Delta < \pi$ becomes $\Delta + \eta < \pi$. A second complication is the need to make sure that all joined replicated servers execute identical sequences of actions despite random task scheduling delays. For instance, it would be unacceptable if a "new-group" message creating an unscheduled view time $V\prime$ is received at a moment when the Broadcast task is startable for some previously scheduled check time V, such that $V - V\prime < \eta$. With a nondeterministic choice of which task to start among several startable tasks, it would be then possible for some joined membership servers to broadcast "present" messages for V and for some others not to broadcast such messages. This would result in incorrect failure detections by time $V + \Delta$, when each server p $\in$ P will assign (1.14) the value $MEMBERS(V)$ to $members(p)$. In particular, a server q who cancelled V because of its earlier processing of the ("new-group", $V\prime$) message could detect its own failure when it would process at $V + \Delta$ the "present" messages send by other servers for time V! The task scheduling rules presented in Sect. 2.2 are thus essential for achieving *agreement* on view and membership check times among servers in the presence of random task scheduling delays. According to these rules, the reception of a "new-group" message for an unscheduled view time $V\prime$ (even within η time units from a check time V) can only have the following consequences. If $V > V\prime$, the start of the Membership task before the Broadcast task at all servers results in the consistent cancellation of V. If $V \leq V\prime$ then none of the previously joined servers cancels V, i.e. they all broadcast "present" by time V.

4.3 Analysis

It is not our intention to provide lengthy formal proofs of correctness for our protocols. This would make this (already long) paper even longer. Instead, we describe informally why these protocols work properly.

Time is used to uniquely identify successive processor groups. Whenever a membership change is detected for some view or check time V (1.14), the atomicity and termination properties of atomic broadcast imply that the values assigned to the local variables *group*(p) and *members*(p) are V and $MEMBERS(V)$, respectively. In this way, between the creation of two successive *groups* V and $V\prime$, any server p joined to V has its local variables *group*(p) equal to V and $MEMBERS(V)$, respectively. Thus, our implementation satisfies requirements (Ss) and (Sa). Since broadcasts are also delivered to their source processor, and the test (1.13) ensures that processor p will not incorrectly join a group of which it is not a member, requirement (Sr) is also satisfied. Requirement (Sh) follows from the termination and atomicity properties of the atomic broadcast service used by a joining processor: these properties force all correct processors to agree on a new view time V as well as on the sequence $V + \pi$, $V + 2\pi, \ldots$ of membership check times that might serve as group identifiers until the occurrence of a new processor start event.

Our implementation also satisfies the timeliness requirements (Tj) and (Td). For instance, consider a processor j that starts a join procedure (1.4) at time S, and an arbitrary processor p, so that both p and j stay correct past time S. If no other joins are started by other processors after time $S - \Delta$, j and p will receive a "new-group" message at times in the interval $[S + \mathbf{D}, S + \Delta]$. Since by hypothesis j and p are correct, they will both broadcast "present" for the view time $S + \Delta$. These messages will be received (1.12) by both p and j in the interval $[S + 2\mathbf{D}, S + 2\Delta]$. Processor j as well as processor p will then both become joined to the group $S + \Delta$ by time $S + 2\Delta$. Consider now the other case when at least another processor started a join procedure in the time interval (S - Δ, S), and let q be a processor that started the join procedure at the smallest time in the above interval. It is then possible that both j and p receive the "new-group"' message broadcast by q for a view time $V < S + \Delta$. In such a case both p and j (and q as well) will broadcast "present" for view time V, and will join the group V at times smaller than $S + 2\Delta$. This yields $J = 2\Delta$ as the worst case join delay.

If a joined processor f fails immediately after broadcasting "present" (this can happen as early as time V - 2η for a view time V created by a join), it will take in the worst case (i.e. no concurrent joins cause other view times smaller than $V + \pi)2\eta + \pi$ time units until the occurrence of the next scheduled membership check time $O = V + \pi$. This will be missed by f, i.e. f $\notin MEMBERS(O)$, so by time $O + \Delta$, when each surviving server p detects that $MEMBERS(O) \neq MEMBERS(V)$, the failure of f is detected (1.4). Thus, the worst case failure detection time is $D = \pi + \Delta + 2\eta$.

We assume $D < R + J$, that is π - $\Delta + 2\eta <$ R. In this way a surviving member p removes a failed processor f from members (p) before f can be reinserted.

5 The attendance list membership protocol

The main drawback of the periodic broadcast protocol is that it requires n atomic broadcasts every π time units, even when no failures or joins occur, where n is the cardinality of the "current" membership. To reduce message overhead in the absence of joins and failures, we now derive a second protocol that uses only n datagram messages every π time units to check for group stability. As in the sketch of the previous protocol, we first assume $\eta = 0$ and deal with the complications caused by a positive η only in the detailed protocol description.

5.1 Sketch of the attendance list membership protocol

Join handling. Joins are handled exactly as in the "periodic broadcast" protocol.

Failure handling. Consider that after starting at time S, a processor j causes the creation of a new processor-group $V = S + \Delta$ with membership $M = MEMBERS(V)$. By time $C = V + \Delta$ all correct members of V know the new membership. To check that this membership is still "current" π time units after the view time V, the members of V agree that one of them (e.g. the one with highest processor identifier) will issue at membership check time $O = V + \pi$ an "attendance list" to be circulated (by using datagram messages) to all group members. We assume $C < O$, i.e. $\Delta < \pi$, to allow the list originator the time to learn the membership of the group V before issuing the list. All group members must relay the list before it returns to the initiator. One way of passing the attendance list among members is to arrange them in a virtual ring [LeL78, Wal82] and let each member m relay the list to its known successor n = $next$(m, M) on this ring, where

$next(\mathrm{m}, M) \equiv$ **if** $\mathrm{m} = \max(M)$ **then** $\min(M)$ **else** $\min \{\mathrm{p} \in M \mid \mathrm{m} \prec \mathrm{p}\}$

If all members of the group V are correct and no datagram service failures occur, each member must receive the list in the time interval $[O - \varepsilon, O + \gamma]$ where $\gamma \equiv \mathrm{n}\,\delta + \varepsilon$ and $\mathrm{n} = |MEMBERS(V)|$. If lists are stamped by their origination time O, late lists received after $O + \gamma$ because of performance failures can be detected and discarded. Since we assume reliable clocks, there is no need to check for "early" lists, which arrive before $O - \varepsilon$.

At membership confirmation time $E = O + \gamma$ each correct group member checks whether it has accepted a recent attendance list with origin O greater or equal than $E - \gamma$. (If $\gamma + \varepsilon \geq \pi$ several attendance lists can coexist in real-time.) If all group members have accepted such lists, then the V membership is still current at time E. If at least one member p of V detects at time E that the last accepted attendance list had an origin O smaller than $E - \gamma$, p concludes that either a processor or a datagram service failure has occurred. To distinguish between these two cases, p initiates a join procedure by broadcasting a "new-group" message. If a processor f$\in$*members*(V) has failed, f will be excluded from the new group $E + \Delta$ that this join will create. Once the new group is created, new rounds of attendance list circulation will check the stability of its membership, until a subsequent join or failure, and so on. Note that, while in the periodic broadcast protocol, each check time costs n atomic broadcasts, in the attendance list protocol each check time costs only n datagram messages. This can result in a significant saving in steady state message traffic in the absence of failures and joins. For example in a point-to-point network, the reduction is from $n(n \times d - n + 1)$ datagram messages every π time units to n datagram messages every π time units.

The *minimum* number of messages necessary for checking stability of a group of size n is n. Indeed, if for a group of size n there were a protocol with only $n - 1$ messages, one of the group members, say x, would not have to send a message during a group stability check. If this hypothetical protocol assumes (optimistically) that no message from x means x is correct, then that protocol would violate the (Td) requirement, because it would not detect a failure of x. If the hypothetical protocol would (pessimistically) assume that x is down when it does not receive a message from it, and would create a new group of which x would be excluded (even though x remains correct), then it would violate the (Ss) requirement that no group changes should occur unless failures or joins occur. Thus, such a protocol cannot exist, and n is the minimum number of messages needed to check membership stability in a group with n members.

Agreement on membership check times. As in the previous protocol, unscheduled view times caused by processor joins cancel all membership check and membership confirmation events scheduled before old processors learn of such joins.

5.2 Detailed description of the attendance list membership protocol

Each membership server is structured into 3 tasks: Membership, Membership-Check, and Membership-Confirmation (Fig. 2). The Membership task controls the initial join of a processor and receives all arriving messages. The Membership-Check task periodically generates attendance lists to check membership stability. The Membership-Confirmation task examines in each joined processor the timely passage of these lists.

A new state variable L (2.3) is used to record the origin time of the last accepted attendance list (2.19). In a manner similar to that discussed previously, a positive constant η influences the computation of start and termination task deadlines (2.11, 2.29, 2.30). It also changes the previously mentioned requirement $\Delta < \pi$ (imposed to let a list initiator learn the membership of a new group before circulating a list in that group) to a final requirement of $\Delta + \eta < \pi$.

```
task Membership:
var members: Set-of-P initially { }
    group: Time: L: Time initially $-\infty$;
    joined: Boolean initially false;
broadcast ("new group", myclock + $\Delta$);
cycle when receive ("new-group", $V$
do if myclock $>$ $V$ then abort fi;
    cancel (Membership-Check, Membership-Confirmation);
    broadcast ("present", $V$, myid);
    schedule(Membership-Check, $V + \pi$) at $V + \pi - \eta$
 od;
when receive ("present", $V$, $M$)
do if $\neg$ joined & (myid $\in M$) then joined $\longleftarrow$ true fi;
    if members $\neq M$ then members $\longleftarrow M$; group $\longleftarrow V$ fi
od;
when receive ("list", $O$)
do if myclock $\leq O + \gamma$
    then $L \longleftarrow O$;
         if myid $\neq$ max (members)
         then send ("list", $O$) to next (myid, members)
         fi;
    fi;
od;
endcycle

task Membership-Check ($O$: Time);
if myid = max (members) then
send ("list", $O$) to next (myid, members) fi;
schedule(Membership-Confirmation, $O + \gamma + \eta$) at $O + \gamma$;
schedule(Membership-Check, $O + \pi$) at $O + \pi - \eta$;

task Membership-Confirmation ($E$: Time);
if myclock $>$ $E$ then abort fi;
if $L + \gamma + \eta < E$ then broadcast ("new group", $E + \Delta$) fi;
```

5.3 Analysis

The upper bound on join delays J for the attendance list protocol is the same as for the periodic broadcast protocol: $J = 2\Delta$. However, the saving in steady state message overhead causes an increase in the processor failure detection time. The worst case that can happen is that a processor f fails immediately after it has broadcast a "present" message for some view time created by a join (2.10) and no other joins or failures occur for a sufficiently long time. If V is the last view time for which f has broadcast "present" before failing, then f's failure could have actually occurred as early as V - 2η. Assuming no joins or other failures occur until the next scheduled membership check time $O = V + \pi$ (2.11), f will be unable to relay the attendance list timestamped O in time (2.27–2.28, 2.2.23). If by membership confirmation time $E = O + \gamma + \eta$ (2.29, 2.31) there still exists a correct member p of V, p will detect that it missed this list (2.33) and will initiate a join procedure that will lead to the creation of a new processor-group (from which f is excluded) by time $E + J$. Thus between the moment a processor fails and the moment the failure is detected by all surviving processors there can be a worst case delay of $D = \pi + \gamma + 3\eta + J$. (The detection of f's failure will be faster if f fails after relaying an attendance list or other concurrent joins or failures cause a view time to occur before $E + \Delta$.) As for the first algorithm, we assume that the restart delay R for a failed processor and the join delay J are bigger than the failure detection time: $D <$ R + J. This imposes the constraint: $\pi + \gamma + 3\eta <$ R.

6 The neighbor surveillance protocol

The attendance list protocol checks membership stability by requiring all members to relay an "attendance list" message. An alternative approach for checking group stability is to check for the stability of the "next" neighborhood relation defined in section 5.1 for all pairs of group members (n, *next* (n, M)). We denote "*previous*" the inverse of the "*next*" relation, i.e. *previous* (*next*(n, M), M) = n. We refer to processor *next* (n, M) as the *successor* of n in M, and to *previous* (n, M) as the *predecessor* of n in M.

The "neighbor surveillance" protocol handles joins in the same manner as the previous two protocols. What is different is failure handling. We briefly sketch below how failures are detected by assuming first a null η.

After the creation of a new processor-group V, each member of V agrees to check whether its predecessor is still correctly functioning at *neighbor check time* $O = V + \pi$. To let the members of V learn who their neighbors are before checking whether these are still alive, we require $\Delta < \pi$. At time O, each member sends a "neighbor" message — timestamped O — to its successor. In the absence of failures, each member receives such a message before time $O + \gamma\prime$ where $\gamma\prime = \delta + \varepsilon$. A "neighbor" message that arrives after $O + \gamma\prime$ is not accepted. At *neighbor confirmation time* $E = O + \gamma\prime$, each member checks whether it has accepted a "neighbor" message with timestamp O greater or equal than $E - \gamma\prime$. If all members receive such messages, then membership stability is confirmed and a new neighbor check time is scheduled for time $O + \pi$. If a member p of V detects that the last "neighbor" message it has accepted had an origin $O < E - \gamma\prime$, then a processor or a datagram service failure has occurred. To distinguish between these two possibilities, p will — as in the attendance list protocol — initiate at time E a join procedure leading to the creation of a new processor-group $E + \Delta$ and to new periodic neighborhood stability checks. Because of its great similarity with the attendance list protocol, we do not give a detailed description of the neighbor surveillance protocol.

6.1 Analysis

The protocol has the same join detection delay constant J as the attendance list protocol. The main advantage of this protocol over the attendance list protocol is an improved worst case *single* processor failure detection delay, which we derive below. Assume that a processor f departs at time T and no other failure or join occurs before time $T\prime = T + \pi + \gamma\prime + 3\eta$. The failure of f will be detected by f's successor by time $T\prime$; this will initiate a join that will cause all other surviving group members to detect f's failure by time $T\prime + J$. Thus, the worst case delay for detecting a *single failure* is $D_1 = \pi + \gamma\prime + 3\eta + J$. The worst case scenario for multiple failures is the following. A processor n_1 departs at time $T - 2\eta$ immediately after broadcasting "present" for a view time T caused by a join, the processor n_2 succeeding n_1 departs at some time in the interval $(T + \pi - \eta, T + \pi + \gamma\prime + \eta)$ before detecting the failure of its predecessor n_1 but after sending its "neighbor" message for check time $T + \pi$ to its successor n_3, n_3 departs before detecting the failure of n_2, but after sending its "neighbor" message for check time $T + 2\pi$ to its own successor n_4, and so on. Such a chain of close failures of members $n_1, n_2, \ldots, n_k$ of a group, where $n_{i+1} = next(n_i, M)$, leads to a worst case *multiple failure* detection delay $D_k = \mathrm{k}\pi + \gamma\prime + 3\eta + J$, $\mathrm{k} \geq 1$. This yields an upper bound on failure detection delay for any possible processor membership of $D\prime = (\mathrm{n} - 1)\pi + \gamma\prime + 3\eta + \mathrm{J}$. For large values of π and n, this can be worse than the D upper bound derived for the "attendance list" protocol.

7 Why is a processor-group membership service useful?

This section illustrates how one can implement a server-group membership service on top of a processor-group membership service without adding any message overhead when no joins or failures occur. We also sketch how the processor-group membership service helps solve two other problems: a service availability problem, and the problem of limiting communication among replicated servers to group members only.

7.1 Server-group membership

The objective of a server-group membership (SGM) service is to let each member z of a replicated server team Z know the membership of the group of correctly functioning Z members. There can be several server teams Z, Z′, ... The SGM service gives any joining server z the identity of all other correct Z members, and subsequently notifies z of any group membership changes. We briefly sketch below how an SGM service can be implemented as an extension of a processor-group membership service. We assume that the operating system of any processor p∈P is capable of notifying the SGM server on p of the failure of any server running on p.

The global state of the SGM service is recorded in the "joined" "group" and "members" variables maintained by the underlying processor-group membership service and a new state variable (or table) "S" of type Server × Team × Processor. The existence of an entry (z,Z,p) in S means "server z of team Z runs on processor p".

When a SGM server on processor p learns that a local server z wants to join the group of correct Z members, it atomically broadcasts a global SGM state update "z joins Z on p" to all SGM servers (including itself). When this message is received by an arbitrary joined SGM server q, q inserts a (z,Z,p) entry in its local S table and notifies all members of Z that run on q of their group membership change. The SGM server on p also sends to z the current Z membership. (A more complex protocol is needed if at join time, the local SGM server would have to give z not only the membership of Z, but the entire global state of the Z group. We leave it to the interested reader to derive this protocol, by analogy with the SGM server join protocol given below.)

The failure of a server z on processor p can either be explicitly notified by z or implicitly notified by the operating system on p when it detects z's failure (in the latter case the SGM server needs to consult S to find the team Z to which z belonged). When the SGM server on p learns about the failure of z, it atomically broadcasts a global SGM state update "z departs Z on p" to all SGM servers. When this message is received by an arbitrary joined SGM server q, q removes (z,Z,p) from its local S variable and notifies all members of Z running on q of the failure of z.

When a joined SGM server on a processor q detects the failure of a processor p, it interprets this as the failure of all servers running on p. The local table S on q contains enough information to identify all servers that were running on p. For each such server z, the SGM server consults S to determine the group Z to which z belonged, deletes (z,Z,p) from S, and notifies all local Z members of the failure of z.

Finally, let us sketch how the SGM server on a newly started processor j joins an existing group of correctly functioning SGM servers. For simplicity, we assume a null η and leave it to the reader to analyze joins when η is positive. If the server starts the join at time T by sending a "new-group" message, it will create an unscheduled view time $V = T + \Delta$. When this message is processed by an arbitrary other joined SGM server q at time V, q adds to the "present" message that it broadcasts not only its identity (as discussed previously) but also the value S(V) of its local S variable at time V The server j monitors all global SGM state updates initiated at times between T and V by joined SGM members; let U be the sequence of these updates. If by time $T + 2\Delta$, j does not receive any "present" message, it concludes that it is alone, joins a processor-group of size 1 with only itself as a member and initializes S to the empty relation $\{\}$. Suppose by time $T + 2\Delta$ j receives at least one "present" message containing a value S(V). This value is outdated if the sequence U of updates to S broadcast between T and V is not empty, so to compute the "current" state of S at time $V + \Delta$, j has to apply all updates in U to S(V) before assigning the resulting S($V + \Delta$) value to its local S variable. This concludes the join of j.

Reliance on processor-group membership to detect processor failures and on operating system services to detect server failures leads to an efficient server-group membership service implementation: no periodic message exchange overhead for checking server-group stability is required other than the lower level overhead needed for solving the processor-group membership problem. Thus, an increase in the number of server-groups or the number of servers does not cause the amount of periodic overhead to grow when the number of processors remains constant. The solution given satisfies all the safety and timeliness properties described for the processor-group membership service in section 3. In particular, if there is a total order **O** on the server names of each team Z and any SGM server interprets the simultaneous failure of any set of Z servers as a failure in that order, then SGM servers can totally order all server join and server failure events in the system. Thus, at the server-group level it is possible to have a total ordering over server-group state updates similar to that mentioned for the processor-group membership abstraction.

7.2 Service availability despite processor failures

We define this problem as follows: choose k processors from a group of n processors (n $\geq$ k) to run a totally ordered set of k services S = (s_1, s_2, ..., s_k) for as long as there are at least k correctly functioning processors in the network. For simplicity we assume that any service can be hosted by any processor and each processor has enough capacity to run only a single service. Often the services are further partitioned into a subset of v vital services (v $\leq$ k) and a subset of k $-$ v non-vital services. Availability of all vital services is necessary for the system to be able to do any work. If some non-vital services become unavailable, the system goes into a "gracefully" degraded mode. The objective is to ensure that all vital services are available for as long as there are at least v working processors and that as many of the other services are provided as working processors in excess of v. Non-vital service s_i, i $>$ v, is considered more important than non-vital service s_{i+j}, j $>$ 0. Examples of common vital services are network reconfiguration coordinator, central lock manager, name manager, and primary host for an important database. Examples of less vital services are network administrator, back-up host for a database service, and communication gateway to some other network. To avoid confusion, it is necessary that, at any instant in real-time, there be *at most* one processor providing a given service. Moreover, should a processor that provides a certain service fail, another processor should start that service within a short bounded time interval. If k = 1, this problem becomes a "continuous leader" election problem [GM82].

A processor-group membership service allows the above problem to be solved quite simply (we assume in what follows that each time a processor-group change occurs at most one processor joins or departs; we leave it to the reader to deal with the case when arbitrary subsets of processors can join or depart simultaneously). If processor p joins a group of size j < k whose older members already provide a set S′ of services, it starts service min(S – S′), otherwise, if j $\geq$ k, p does nothing. If the failure of a processor p is detected when a new group g is created, then, if p was not running any service nothing must be done. Otherwise, if p was running a service s_i, and the size j of g is greater than k, the oldest member of g without a service must start s_i. If v < j $\leq$ k and i < j, the processor that runs s_j aborts s_j and starts s_i. The above rules ensure that in any processor membership of size j $\geq$ v all vital services as well as the most important j – v non-vital services are available, and that processor failures cause a minimum number of service migrations. Such a solution is superior to a static assignment of services to processors (that would render a service unavailable as long as the processor statically assigned to it is down) or a static primary/back-up scheme where the crash of just two processors to which some vital system services are assigned (such as primary and back-up lock manager) can bring the entire system down despite the fact that all other network processors function correctly.

7.3 Limiting group communication to members only

Consider a group g of processors that must ensure the availability of a certain service s such that s can be correctly provided only when there exists at most one server for s (i.e. the co-existence of two servers for s would lead to a service failure). Suppose that at time T a member p of g has to start s locally. If a performance failure prevents p from starting s in time, by time $T + D$, the surviving members of g will form a new group g′ from which p will be excluded. For availability reasons, s will have to be started in the new group g′ by another processor q. For availability reasons, s will have to be started in the new group g′ by another processor q. If p is not equipped with timeliness checks that transform its performance failures into crash failures (such as those illustrated at (1.7, 1.18, 2.8, 2.32)), p might after time $T + D$ broadcast a message that it has (finally) started s. The members of g′ must ignore this message to avoid confusion.

To solve this (and other related problems), it is necessary to clearly define the *scope* of group communication. Communication other than join communication, as illustrated in section 7.1, must be restricted to established group members only. If all groups that can exist in time are uniquely identified by distinct group identifiers, these identifiers can be used to filter group messages. In our example, if all messages are stamped by the group identifier to which their sender is joined, it is easy for all members of g′ to discard messages stamped g coming from a member of a past group that is no longer a member of g′. The systematic stamping of group messages with group identifiers together with rules for dealing with message transmissions that overlap group transitions can prevent members of a group from sending messages to, or receiving confusing messages from, outside the group.

8 Optimizations and extensions

This section discusses three optimizations that can be applied to all protocols given, as well as a possible hierarchical extension of the attendance list protocol that can handle an increased number of processors.

A straightforward optimization that will lead to better failure and join delays consists of merging the membership layer presented in this paper and the atomic broadcast algorithms discussed in [CASD85] and [Cri90] into a single layer. The elimination of communication between a lower broadcast layer and an upper membership layer will save several η's from the join and failure processing delays given previously. A second optimization relies on the observation that, at each view time, all members broadcast "present" messages for a same view time V within a short real-time interval of length ε. The message diffusions corresponding to these broadcasts can be "merged" so as to reduce the total number of exchanged messages if the following *diffusion optimization rule* is followed: a processor p should send a message containing some information I to a neighbor n only if p is not sure that n knows I. One way of enforcing this rule is to record on each "present" message diffused for view time V that carries information I the identity P(I) of all processors that know I. A processor will then relay a "present" message for view time V that carries information I only to those neighbors that are not in P(I). A third optimization prevents excessive message traffic that could be caused by a processor that crashes and restarts frequently. To prevent such traffic, it is possible to associate with each processor in the network the set of its last join times and to check at each join whether the processor is not a "frequent joiner".

When the number n of processor-group members becomes significant, it becomes undesirable for all processors to send messages at the same clock times, especially in a system with a small maximum deviation between clocks ε. In such a case, one can spread the message traffic by using a technique called "staggering". The main idea behind staggering is as follows. Let k be the rank of a member p of a processor-group g, where k is defined as the relative position of p on the "next" virtual circle associated with g as described in section 5.1, starting with the member with the greatest processor identifier. If a membership protocol requires p to send a message at some time T, the "staggered" protocol will require p to send the message at $T - ks$, where s is a staggering time constant, chosen greater than ε to ensure a proper dispersion of message traffic. Since staggering can lead to an increase in the membership check period π, it can worsen the bounds on failure detection delays. Often the resulting increase is more than compensated by the elimination of membership protocol related traffic spikes.

The membership protocols presented do not scale well to large point-to-point networks, because of the number of messages broadcast when membership changes occur. The design of protocols that provide bounded membership change detection delays in the presence of concurrent failures and joins and that scale well for large point-to-point networks is a challenging problem. In an internet composed of several smaller networks, one natural solution to this problem is to use local attendance list protocols to determine the membership and the leader for each small network, and to let these leaders participate in a global leader attendance list protocol at the internet level. A leader can then report any change in the membership of its own small network by piggybacking it on the global attendance list. These changes can then in turn be propagated by all the other leaders into their own small networks. This 2-level hierarchy can, of course, be generalized to a k-level hierarchy. Such hierarchical schemes provide bounded crash and join processing delays and require less message traffic to process membership changes than a brute force 1-level attendance list protocol run on the entire internet. The cost is an increase in protocol complexity and in join and crash detection delays.

9 Conclusion

This paper has specified the processor-group membership problem and has provided three simple protocols for solving it in synchronous networks. The first protocol provides the fastest failure detection time, but can have a significant message overhead even when no failures or joins occur. The second and third protocols have a minimum message overhead in the absence of failures and joins. If failures are rare, the detection time provided by the third protocol is the best. However, if failures can cluster in time, the second protocol provides a better upper bound on failure detection times than the third.

The processor-group membership problem is a fundamental problem in distributed fault-tolerant system design in that it allows many other problems to be easily solved. We illustrated how a processor-group membership service helps to solve the server-group membership problem, a service availability problem and the problem of restricting group communication to group members only. A membership service is also useful in solving other problems, such as load balancing and enforcing service group replication strategies for various system services [Cri91]. Indeed, processor joins and failures are significant events for a load balancer or a service availability manager and a processor membership service that reliably notifies such programs of joins and failures considerably simplifies the design of such subsystems.

When this paper was first published [Cri88], we knew of no other publications that discussed the processor membership problem in synchronous networks. Since then, another synchronous membership protocol for Time Division Multiplexing Access (TDMA) based broadcast networks was published [KGR91]. Although Kopetz, Grünsteidl and Reisinger do not formally state the properties that their protocol possesses, if one interprets TDMA cycle numbers as measuring a global system time and uses cycle numbers to uniquely identify processor-groups, one realizes that the properties that their protocol satisfies are basically the same as the safety and timeliness properties discussed in this paper. The protocol of [KGR91] achieves agreement on local membership views by assuming what one might call "quasi-atomic" broadcast channels: for any message m sent by a processor on a channel, either (1) all correct processors receive m, or (2) no correct processor receives m, or (3) exactly one correct processor does not receive m. If channel failures could cause at least two correct processors to miss a message received by some correct processor, the agreement property on membership views can be violated. The protocols presented in this paper are network independent: they can be implemented in both point-to-point and broadcast synchronous environments. When implemented in a broadcast environment, our approach does not rely on the existence of quasi-atomic channels: the protocols described in [Cri90] achieve atomic broadcast even when faulty channels are allowed to deliver messages to arbitrary subsets of correct processors.

All other processor-group membership protocols that we know assume an asynchronous communication network in which partition failures can occur [BJ87, Car85, CM84, ASC85, KLW86], and [Wal82]. These acknowledgement-based, asynchronous protocols do not guarantee any bound on processor failure detection and processor join delays even when run in an environment where all delays are bounded. Progress in reaching agreement on new processor-group memberships and learning about new processor failures and joins can be infinitely delayed by appropriately chosen sequences of events such as joins, omission communication failures and processor crash failures. On the other hand. asynchronous protocols such as the ones described in [CM84] and [ASC85] can be designed to satisfy safety requirements such as (Sa) and (Sh) even when partitions occur. However, to ensure these properties in the presence of partitions, these protocols require that at any time there be at most one processor-group authorized to carry on work. This is achieved by requiring that a majority (or a quorum) of processors be in a group of processors before that group can do any work for system users. Thus, while asynchronous protocols are safe but not timely, synchronous protocols are always timely, but are only safe in the absence of partitions. For example, if a partition occurs while a correct processor diffuses a "present" for some membership check time T, some correctly functioning processors might not receive the message by $T + \Delta$ while others might receive it. This will lead to divergent views on membership, in violation of requirements (Sh) and (Sa). There exist known approaches (see for example [SSCA87]) for detecting and reconciling processor views computed by using synchronous protocols after partition occurrences, but such "optimistic" approaches cannot be used in applications in which one cannot compensate for the actions taken by certain processors when their local views were inconsistent with the views of other processors.

One is thus currently faced with the following dilemma. Either adopt a synchronous approach, which guarantees bounded response times in the presence of any sequence of failures and joins, can continue to work for as long as there exists at least one working processor in the system, but can violate its safety requirements when communication partitions occur, or adopt an asynchronous approach that never violates safety requirements such as (Sa) and (Sh), even when communication partitions occur, but requires the presence of a majority (or quorum) of correct processors before any work can be done, and cannot guarantee any bounded response times. Whether one can design protocols that are always safe, are timely in a synchronous environment and are untimely only when run in an asynchronous environment is an open problem.

Acknowledgements. Comments from Jean Raymond Arbrial, Houtan Aghili, Danny Dolev, Joe Halpern, Michael Raynal, Shri Sabnis, Frank Schmuck, Dale Skeen, Jim Stamos, Ray Strong, Moshe Vardi, Pierre Wolper, and the Referees helped clarify my understanding of the membership problem. This research was partially sponsored by IBM's System Integration Division as part of a FAA Sponsored project to build a new air traffic control system for the US.

References

[ASC85] A. Abbadi, D. Skeen, and F. Cristian. An efficient fault-tolerant protocol for replicated data management. In *SIGACT/SIGMOD*, 1985.

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

[Car85] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, Jun 1985.

[CAS86] F. Cristian, H. Aghili, and R. Strong. Approximate clock synchronization despite omission and performance failures and processor joins. In *Proceedings of the 16th International Symposium on Fault-Tolerant Computing*, pages 218–223, Wien, Austria, Jun 1986.

[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.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[Cri88] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proceedings of the Eighteenth International Conference on Fault-tolerant Computing*, pages 206–211, Tokyo, Jun 1988.

[Cri90] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real Time Systems*, 2:195–212, 1990.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[GM82] H. Garcia-Molina. Elections in a distributed computing system. *IEEE Transactions on Computers*, C-31(1):49–59, Jan 1982.

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

[KLW86] N. Kronenberg, H. Levy, and Strecker W. Vax clusters: a closely coupled distributed system. *ACM Tr. on Computer Systems*, 4(2):130–146, 1986.

[Lam84] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Transactions on Programming Languages and Systems*, 6(2):254–280, 1984.

[LeL78] G. LeLann. Algorithms for distributed data-sharing which use tickets. In *Proceedings of Third Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 259–272, Berkeley, CA, Aug 1978.

[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.

[Wal82] B. Walter. A robust and efficient protocol for checking the availability of remote sites. In *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 45–67, 1982.