The protocol has some distinct advantages over others proposed in literature. One advantage is the way it handles simultaneous failures and recoveries. In other protocols that have been proposed [**?**, **?**, **?**, **?**], the protocol is restarted if a subsequent failure or recovery is detected while the agreement algorithm is being executed. However, this is not in the case in our protocol—failures or recoveries that are detected while the protocol is in progress are taken into account incrementally by updating SuspectUpList or SuspectDownList appropriately.

A second advantage is the way our protocol relaxes the requirement that removal of a failed process from the membership list be totally ordered with respect to all other events. In particular, a process waits to update its membership list only until it has determined the last message sent by the failed process; it need not wait for other processes to update their membership lists. In contrast, other protocols force a process to wait until all alive processes have confirmed the failure.

A final advantage is that removal of failed processes from the membership list need not be done at the same time at all the processes. This results from the fact that sf-groups are created dynamically at each process, and these groups need not be the same at all processes. Thus, a process that does not have to merge two sf-groups will be able to remove the members of the first group before another process that does have to merge the two groups. This improves the efficiency of the application and simplifies the design of the protocol.

Let $q$ be NACKed by $s$, where $s, q \in Q$. From [C], we see that $\langle$Nack, q is down$\rangle$ is not followed by $\langle$s is down$\rangle$ message. Since $\langle$Nack, q is down$\rangle$ is sent by $s$ and [B] implies that $r$ sent $\langle$Ack, s is down$\rangle$ in response to $\langle$s is down$\rangle$, $r$ must have received the $\langle$s is down$\rangle$ message before receiving $\langle$Nack, q is down$\rangle$. Since $r$ received the $\langle$s is down$\rangle$ message and later received $\langle$Nack, q is down$\rangle$ from $s$, $s$ must have been freed before $r$ received $\langle$Nack, q is down$\rangle$. This implies that $s$ has been freed before $q$. So, we have the following:

> **[D] :** If a participant $q_i$ is NACKed by another process $q_j$, then $q_j$ is freed before $q_i$ for all $q_i, q_j \in Q$.

From [A], every participant in $Q$ must be NACKed by some other participant in $Q$, and from [D] we see that every participant in $Q$ has a participant in $Q$ that was freed before it. Since $Q$ is a finite set and "freed before" is a total relation, this is impossible for any finite non-empty $Q$. Thus, the set Q must be empty.

Therefore, $Q = L_r - L_p = \phi$. Thus $L_r = L_p$. So, $q \in L_r \Leftrightarrow q \in L_p$. □

**Theorem 6** *The removal of failed processes from the membership list follows the total order of sf-groups and all processes in one sf-group are removed simultaneously.*

**Proof:** Trivial. □

In summary, we have shown the following. First, since a process starts accepting messages from a recovering process as soon as the recovering process is added to the membership list, by Theorem **??** we conclude that every alive process starts accepting messages from a rejoining process at the same time in the view. Theorem **??** confirms that the decision reached by the membership protocol about a failed (or suspected failed) participant is consistent at all alive processes. Thus, external consistency is ensured by Theorems **??** and **??**. Similarly, the internal consistency is ensured by Theorems **??** and **??**. Finally, Theorem **??** guarantees that the order of the removal of failed processes is correct.

# 5 Concluding Remarks

This paper presents a membership protocol based on the partial ordering of messages provided by the Psync multicast mechanism. The protocol—along with a complete system that includes ordering, recovery, and monitor protocols—is implemented on a network of Sun3 workstations running the *x*-kernel [**?**]. To demonstrate the system, we have also implemented a replicated directory object [**?**]. We are currently performing experiments to measure the overhead of our algorithm under different situations.

In the following theorem we prove that if one process decides that a process $p$ has failed, every alive process eventually decides the same.

**Theorem 5** *Let $L_p$ and $L_r$ be the set of alive participants at the end of membership check period containing $\langle$q is down$\rangle$ in views of participants p and r respectively. If $r \in L_p$ and $p \in L_r$ then $q \in L_p \Leftrightarrow q \in L_r$.*

**Proof:** Using the above observation, we logically combine sufficient previous membership check periods so that the periods in the two processes start at the same time.

Let $Q = L_r - L_p = \{q_1, ..., q_k\}$. That is, $Q$ is the set of processes that are in membership list of process $r$ but not $p$ at the end of the membership check period.

Assume that $q \in Q$. Since $q$ is in the membership list of $r$, the message $\langle$q is down$\rangle$ is immediately followed by $\langle$Nack, q is down$\rangle$ from some participant (say $s$) in the view of participant $r$. Since $q$ is not in the membership list of $p$, message $\langle$q is down$\rangle$ is not followed by any $\langle$Nack, q is down$\rangle$ message in $p$'s view. Since $p$ receives every message from all participants in $L_p$, and has not received the $\langle$Nack, q is down$\rangle$ message from $s$, then $s$ must belong to $Q$. Thus, we have the following:

> **[A] :** For every $q \in Q$, the $\langle$q is down$\rangle$ message is followed by a $\langle$Nack, q is down$\rangle$ message from some participant $q_i \in Q$ in the view of $r$ but not in the view of $p$.

Next, since $r \in L_p$, $p$ has received every message from $r$. Since $p$ has not received any of the NACKs, $r$ must have sent $\langle$Ack, q is down$\rangle$ for every $\langle$q is down$\rangle$ message where $q \in Q$. Thus, we have the following:

> **[B] :** $r$ has sent $\langle$Ack, q is down$\rangle$ for every $\langle$q is down$\rangle$ message for all $q \in Q$.

Now, since $p$ has not received any of these NACKs and has received all of the $\langle$q is down$\rangle$ messages, none of these NACKs are followed by a $\langle$q is down$\rangle$ message. So, we conclude the following:

> **[C] :** For any $q \in Q$, the message $\langle$Nack, q is down$\rangle$ is not followed by any $\langle$p is down$\rangle$ message where $p \in Q$.

In the following discussion, let $s$ be some arbitrary process in $ML$. We say that "$q$ is NACKed by $s$" if process $r$ has received the message $\langle$Nack, q is down$\rangle$ from $s$, and this is the first NACK received immediately following the $\langle$q is down$\rangle$ message. Also, we say "q is freed before s" if $q$ is NACKed before $s$; i.e., the first NACK message immediately following $\langle$q is down$\rangle$ is received before the first NACK message immediately following $\langle$s is down$\rangle$.

**Proof:** Stability decisions are correct if a message is decided stable when it is followed by messages from every alive process. The stability check of a message *m* can go wrong if some process that has a message *m'* following *m* is not considered for the stability check and *m'* has not been received. If a process *p* fails, then it is removed from the membership list at its RDT. Thus, the failed participant is removed from consideration for stability checks only after its RDT. Since all the messages from a failed participant have been received by its RDT (Theorem **??**), all of its messages have been received before it is removed from consideration for stability checks. Thus, removal of a failed participant does not introduce any inconsistency in the stability checks. When a participant *p* recovers, it starts sending messages only after its RUT. Since the recovering participant is incorporated into the membership list at its RUT, it is considered for stability checks for all the messages at or following RUT. Thus, stability checks remain consistent during the recovery of a process.

Completeness decisions are correct if a wave is decided to be complete only when it contains all of its messages. As noted in Section 2, this is typically done by testing the stability of a message in the wave. In the presence of failures, however, a failed process is not considered to determine stability, and a wave may contain messages from processes that later fail. This does not cause a problem because a failed process is not considered when determining the completeness of a wave only after it has been removed from the membership list; i.e., after its RDT. But by Theorem **??**, all messages from failed processes have been received by its RDT. Therefore, a wave that is determined to be complete has all messages from failed processes and all the alive processes (since all the alive processes are considered to determine stability), and as a consequence, removing a failed participant does not cause any inconsistency in wave completeness checks. A recovering participant *p* starts sending messages after its RUT, so no waves between its RDT and RUT inclusive contain a message from *p*. Since *p* is added to the membership list at its RUT, it is considered for completeness checks for all succeeding waves. Thus, completeness decisions are consistent during the process of participant recovery. □

Before considering the next theorem, we observe that we can always consider the logical time at which a given membership check period started as the same in all alive processes. This follows from the property that consecutive membership check periods (or equivalently, sf-groups) can be combined into a single membership check period without changing the semantics. Thus, given a specific check period, we can successively merge earlier check periods as required in each process to obtain a single membership check period that starts at the same logical time everywhere. This iteration is guaranteed to stop since the initial membership check period must start at the same time in every process. Note that this merging is done only for the purposes of the proof, and is not actually reflected in the protocol itself.

### 4.4 Correctness Arguments

We now argue that internal consistency and external consistency are maintained by the membership protocol.

**Theorem 1** *All messages from process p have been received by a process q before q decides the failure of p.*

**Proof:** The set of messages from process $p$ is the union of all messages from $p$ received at all alive processes. At the RDT of $p$, the message $\langle$p is down$\rangle$ is immediately followed by message $\langle$Ack, p is down$\rangle$ from every alive process. Thus, all messages sent by $p$ that have been received by any alive participants prior to sending $\langle$Ack, p is down$\rangle$ have been received by $q$. Since a process ignores any further incoming messages from $p$ as soon as it sends $\langle$Ack, p is down$\rangle$ $q$ receives no other messages from $p$. Thus, $q$ has received all messages from $p$ at its RDT; i.e., when it decides that $p$ is down. □

**Theorem 2** *An alive process receives all the messages in the conversation.*

**Proof:** The membership check period ends at the RDT or RUT of any participant. From Theorem **??**, we know that $p$ has received all messages from failed participants. Since the membership check period terminates at a wave that contains messages from all alive processes (ACK or NACK), all messages from all alive processes have also been received. Thus, a processes receives all messages in the conversation in the membership check period. Since Psync guarantees the delivery of all the messages in the normal state (i.e., when a process is not in a membership check state), an alive process receives all the messages in the conversation. □

**Theorem 3** *A recovering process is incorporated in the conversation at the same logical time at all alive processes.*

**Proof:** A recovering process $p$ is incorporated into the membership list when the $\langle$p is up$\rangle$ message is immediately followed by a $\langle$Ack, p is up$\rangle$ message from every alive process, i.e., when the wave containing the message $\langle$p is up$\rangle$ is complete. Since this wave is the same at all processes, $p$ is added to the membership list at the same time by all processes. □

**Theorem 4** *Stability and completeness decisions taken in the application during the execution of membership protocol are valid.*

| Message | Actions of the Membership Protocol |
|---|---|
| ⟨p is down⟩ | If a message from $p$ at the same logical time as ⟨p is down⟩ has been received, then send ⟨Nack, p is down⟩; otherwise send ⟨Ack, p is down⟩, stop accepting messages from $p$, insert ⟨p is down⟩ in SuspectDownList, and adjust count. |
| ⟨Nack, p is down⟩ | Remove ⟨p is down⟩ message from SuspectDownList and start accepting messages from $p$. |
| ⟨Ack, p is down⟩ | If message ⟨p is down⟩ is stable then decrement count. |
| ⟨p is up⟩ | Send ⟨Ack, p is up⟩, insert ⟨p is down⟩ in the SuspectUpList, and increment count. |
| ⟨Ack, p is up⟩ | If ⟨p is up⟩ is stable, then incorporate $p$ into the membership list, remove ⟨p is up⟩ from SuspectUpList, and adjust count. |

Figure 4: Membership Protocol

all ⟨Ack, p is down⟩ messages before receiving ⟨q is down⟩. As a result, the right-most process uses two membership check periods. The implication of this situation is that $p$ is removed from the membership list earlier at the right process than in at the left process, thus allowing its execution to proceed without unnecessary delay.

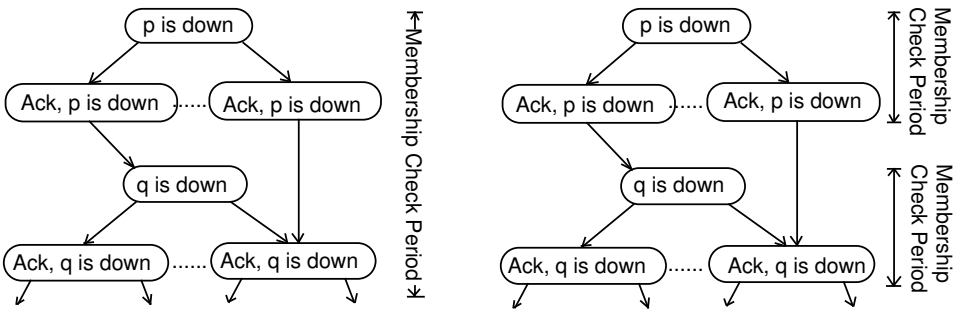


Figure 5: Two Different Processes' View

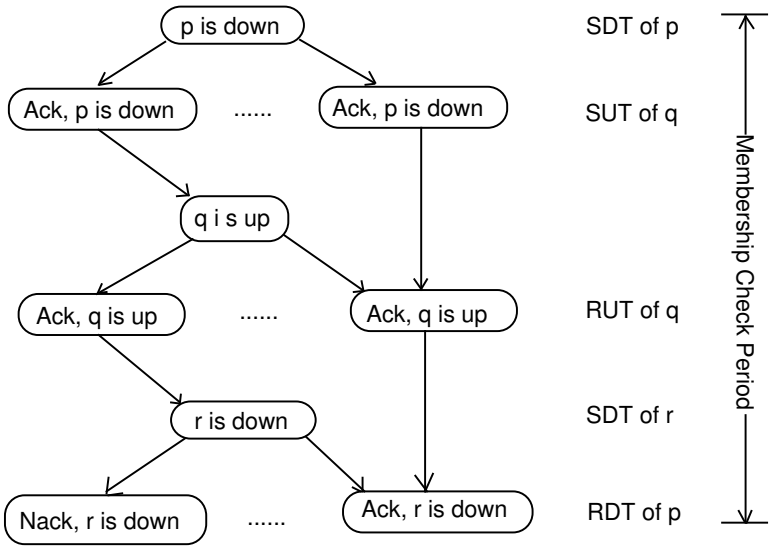


Figure 3: View Representing a Membership Check Period

Figure 3 illustrates one possible scenario, in which processes $p$ and $r$ are checked for a possible failure and process $q$ rejoins the membership list. The membership check period starts with the arrival of the ⟨p is down⟩ message. At the end of this period—that is, the RDT of $p$—process $p$ is removed from the membership list. However, $r$ remains since ⟨$r$ is down⟩ is immediately followed by ⟨Nack, $r$ is down⟩.

Figure 4 gives a more formal description of the protocol based on actions taken by a given process upon receipt of each type of message. The phrase "count is adjusted" means that the variable count is assigned the number of messages in SuspectUpList plus the number of unstable messages in SuspectDownList. In addition, if count goes to zero as the result of processing a message, then SuspectDownList is reinitialized to empty and the corresponding processes are removed from the membership list.

The key observation about the algorithm is that different processes can form different sf-groups, or equivalently, different processes can have different membership check periods. The reason is that the membership protocol includes a process $p$ for consideration as soon as it receives ⟨p is up⟩ or ⟨p is down⟩ protocol initiation message. To see the effect of this, consider the scenario outlined in Figure 5. Assume the process on the left receives ⟨q is down⟩ before the final ⟨Ack, p is down⟩ message.[2] This is possible because there is no path from the final ⟨Ack, p is down⟩ to the ⟨q is down⟩ message. Thus, both $p$ and $q$ are considered in one membership check period. In contrast, assume the process on the right receives

[2]Keep in mind that any pair of messages in the context graph for which there is no path leading from one to the other could have been received by the process in either order, and in fact, in different orders at different processes.

**Suspected Up Time (SUT):** The SUT of a process $p$ is the logical time containing the $\langle$p is up$\rangle$ message.

**Realized Up Time (RUT)** The RUT of a process $ML$ is the logical time when $p$ is masked back into the membership list.

Furthermore, define a *membership check state* as any state where either SuspectDownList or SuspectUpList is non-empty. In a similar manner, let a *membership check period* be the time interval over which the system is in a membership check state. A membership check period always starts at the SDT or SUT of some process and ends when the two suspect lists are empty. In other words, a membership check period starts when count becomes non-zero and continues until count becomes zero. The end of a membership check period is always signified by the RDT of some process.

In the membership protocol, a message is considered stable if it is followed by messages from all the processes that are not in SuspectDownList. Note that this definition of stability is applicable only in the membership protocol; all other protocols, including those that run concurrently with the membership protocol, use the standard definition of stability given in Section 2.

### 4.3 Membership Protocol

The main idea of the membership protocol is to establish agreement among all the alive processes about the membership list at the end of the corresponding membership check period. Thus, if there are $n$ processes of which $k$ are suspected to have failed, agreement on the failure of the $k$ processes is reached if none of the other $n - k$ processes have a message from any of the suspected processes at the same logical time as the first protocol initiation message sent by the monitor protocol. To maintain external consistency, the membership protocol also synchronizes the RUT of rejoining processes and the SDT of failed processes among all the alive participants.

Informally, the membership protocol may be described as follows. Upon receiving a $\langle$p is down$\rangle$ message, the message is added to SuspectDownList. Upon receiving a $\langle$p is up$\rangle$ message, the message is added to SuspectUpList. The message associated with a suspected down process is subsequently removed from SuspectDownList if there is any process that has evidence to contradict the hypothesis that it has failed; that is, if a $\langle$Nack, p is down$\rangle$ message is received immediately following the $\langle$p is down$\rangle$ message. A suspected up process is removed from SuspectUpList and added to the membership list as soon as the appropriate $\langle$p is up$\rangle$ message becomes stable. The membership check period ends when all messages in SuspectDownList become stable and SuspectUpList becomes empty. At this point, all of the suspect down processes are removed from the membership list, and SuspectDownList is reinitialized to empty.

The other type of membership list modification is the addition of recovering processes. In this case, it is sufficient to add a process to the membership list at every process sometime before the recovered process sends its first message. Once this incorporation is complete, the process participates normally in system activity, including execution of future membership protocol agreement algorithms. Since the set of alive processes must be same at all the processes while executing—for example, a failure agreement algorithm—the recovering process must be incorporated at the same logical time with respect to all other membership protocol events at all processes.

In summary, the order in which membership list modification events are handled is as follows:

1. All processes in the same sf-group are removed simultaneously. The order of removal of processes in different sf-groups follow the relative order of the sf-groups.
2. A recovering process is incorporated in the membership list at the same logical time at all the processes.

### 4.2 Protocol Preliminaries

For simplicity, we first define the terms and data structures used by the protocol. We say a message $m_2$ *immediately follows* $m_1$ if there is a direct edge from $m_1$ to $m_2$ in the context graph. We say $m_2$ *follows* $m_1$ if there is a path from $m_1$ to $m_2$ in the context graph. A process $p$ is *suspected down* if it is in the membership list and a $\langle$p is down$\rangle$ message has been received. Similarly, a process $p$ is *suspected up* if it is not in the membership list and $\langle$p is up$\rangle$ has been received.

The membership protocol maintains two lists: SuspectDownList and SuspectUpList. SuspectDownList contains the list of $\langle$p is down$\rangle$ messages that have been received and SuspectUpList contains the list of $\langle$p is up$\rangle$ messages that have been received. As described below, messages are removed from these two lists once the process can reach a conclusion about the status of each $p$. The protocol also maintains an integer variable count that contains the total number of messages in SuspectUpList plus the number of unstable messages in SuspectDownList. Initially the value of count is zero and the two lists are empty.

We define the following logical times related to process failures and recoveries, where by logical time we are referring to a wave in the context graph.

**Suspected Down Time (SDT):** The SDT of a failed process $p$ is the logical time containing the $\langle$p is down$\rangle$ message.

**Actual Down Time (ADT):** The ADT of a failed process $p$ is the earliest logical time such that there are no messages from $p$ at or after ADT.

**Realized Down Time (RDT):** The RDT of a failed process $p$ is the logical time when $p$ is masked out of the membership list.

agreement for processes that fail in this way must be done simultaneously. On the other hand, if the last message sent by $q$ is in the context of the last message sent by $p$, then $q$ might be able to participate in the failure agreement of $p$.

Now, expand this scenario to include a third failing process $r$. Suppose that the last message sent by $r$ is at the same logical time as the last message sent by $q$, but follows the last message sent by $p$. By the argument made above, this implies that the failure agreement of $q$ and $r$ must also be done simultaneously, leading to the conclusion that all three processes must be treated as a group. In general, then, the failure agreement of a set of processes must be done simultaneously whenever the last message sent by any process in the set is at the same logical time as the last message sent by at least one other process in the set.

We formalize this notion by defining a *simultaneous failure group* (*sf-group*) $S$ as follows:

1. $S$ consists of a set of processes that have failed,
2. For every proper subset $S' \subset S$, there is a process $p$ in $S - S'$, such that the last message sent by $p$ is at the same logical time as the last message sent by at least one process in $S'$, and
3. $S$ is maximal; i.e., adding any other process in $S$ violates the second point.

From the point of view of the membership protocol at a given process, all processes in an sf-group are treated as a unit: one failure agreement algorithm is used for the entire group and they are eventually removed from the membership list simultaneously. Thus, as execution of a process proceeds, there are a series of sf-groups totally ordered with respect to one another. Specifically, an sf-group $S_2$ is said to *follow* another sf-group $S_1$ if the last message sent by any process in $S_2$ follows the last message sent by all processes in $S_1$. In this situation, the failure agreement for $S_2$ is typically performed after the failure agreement for $S_1$.

Notice, however, that it is also correct to perform the agreement for $S_1$ and $S_2$ simultaneously; i.e., as if they were a single sf-group. This type of merging may be necessary in certain situations, such as if one or more processes in $S_2$ fail before they can participate in the failure agreement associated with $S_1$, or if an alive process receives the protocol initiation message for a process in $S_2$ before receiving all messages associated with the protocol for $S_1$. Interestingly, since messages are received in partial order, this latter situation can result in different sf-groups being formed at different functioning processes. As discussed more fully in Section 4.3, our protocol exploits this fact to allow some processes to remove processes from the membership list earlier than others, while still preserving the semantic correctness of the application.

has sent an $\langle$Ack, p is down$\rangle$ message. Since a process stops accepting messages from $p$ before sending $\langle$Ack, p is down$\rangle$ message, all messages from failed process have been received at the time of its removal. Thus, a process receives all the messages in the conversation. Stability decisions are correct because the failed process is removed at the same time and a recovering process is incorporated at the same time at all the processes. This means that every process determines the stability of a message with respect to the same set of processes. Thus, both internal and external consistency are satisfied.

Finally, notice that the events associated with the failure of a process, i.e., the halt in accepting messages and its removal from the membership list, are only partially ordered with respect to other messages in the system. Compared with other protocols in which these events are totally ordered with respect to other messages, this approach enhances the concurrency and efficiency of the application.

# 4 Handling Multiple Failures and Recoveries

This section extends the membership protocol to handle concurrent failures and recoveries. In presence of such concurrent events, the protocol becomes much more complex. Perhaps the predominant reason for this is the inherent lack of knowledge about the set of processes that participate in the membership agreement process itself. That is, processes may fail or recover at any time and, in particular, they may fail or recover while the membership protocol is in progress. Another source of complexity stems from the requirement that a consistent order of removal or incorporation of processes in the membership list be maintained. This order must be the same at all the processes to ensure correctness of the application, however, it is not at all clear what this order should be, or even the correct interpretation of "the same."

In this section, we first address this latter question by deriving an order in which these list modification events must be performed. We show that the semantics of *remove*—removing a failed process from the membership list $ML$—and *join*—incorporating a recovering process in the membership list $ML$—put a restriction on the order in which the modifications of the membership list take place. We then describe the actual membership protocol.

## 4.1 Ordering List Modification Events

Suppose that two processes $p$ and $q$ fail at approximately the same time. If the last message sent by $p$ is at the same logical time as the last message sent by $q$ (that is, neither is in the context of the other in the context graph), then $p$ and $q$ can obviously not participate in each other's failure agreement protocol. Since establishing agreement about the failure of a process requires concurrence of all functioning processes,

the process or network being "slow" rather than an actual failure. The membership protocol uses this heuristic to establish the failure of a process.

The goal of the protocol is to establish an agreement among the $n-1$ alive process about the failure or recovery of the $n^{th}$ process. As outlined above, the basic strategy is to agree on the failure of the process if and only if none of the $n-1$ participants have received a message from the suspect process at the same logical time as the protocol initiation message. In case of recovery, the process is incorporated in the membership list once all the remaining $n-1$ participants have acknowledged its recovery.

The actual details of the protocol are illustrated in Figure 2. Upon suspecting the failure of a process $p$, the monitor protocol submits a $\langle$p is down$\rangle$ message to the conversation. $p$ is subsequently considered to have failed if the $\langle$p is down$\rangle$ message is later followed in the context graph by a $\langle$Ack, p is down$\rangle$ message from every other participant.

| **Message** | **Actions of the Membership Protocol** |
|---|---|
| $\langle$p is down$\rangle$ | If a message from $p$ at the same logical time as $\langle$p is down$\rangle$ has been received, then send $\langle$Nack, p is down$\rangle$; otherwise send $\langle$Ack, p is down$\rangle$ and stop accepting messages from $p$. |
| $\langle$Nack, p is down$\rangle$ | Start accepting messages from $p$ and terminate protocol. |
| $\langle$Ack, p is down$\rangle$ | If message $\langle$p is down$\rangle$ is stable then remove $p$ and terminate protocol. |
| $\langle$p is up$\rangle$ | Send $\langle$Ack, p is up$\rangle$ |
| $\langle$Ack, p is up$\rangle$ | If $\langle$p is up$\rangle$ is stable, then add $p$ to the membership list and terminate protocol. |

Figure 2: Membership Protocol Assuming Only a Single Failure

Internal and external consistency are easily demonstrated for this algorithm. Every process stops accepting messages from the failed process at the same logical time in the conversation—on receipt of the $\langle$p is down$\rangle$ message. Similarly, a process is incorporated at the same logical time—the wave containing the $\langle$p is up$\rangle$ message—at all the processes. As a result, every process starts accepting messages from the recovered partipant at the same time—as soon as it is incorporated. Since Psync guarantees the delivery of messages, every participant receives the same set of messages, and as result, every participant reaches the same conclusion about the failure of a process. The failed process is removed when every process

a change of state of a process, initiates the membership protocol by submitting a distinguished message to the conversation. A failure is typically suspected when no message has been received from a process in some interval of time, while recovery is based on the asynchronous notification generated when the recovering process executes the Psync restart primitive. However, the monitor protocol is independent of the membership protocol and may employ any strategy to detect process status changes without affecting the membership protocol.

### 3.1 Correctness Criteria

To ensure the correctness of the application implemented over Psync in the presence of failures, the membership protocol must guarantee two properties: that all functioning processes receive the same set of messages in partial order, and that the decisions taken by the application based on these messages are correct even in the presence of failures. We call these aspects *external consistency* and *internal consistency*, respectively. These are defined more formally as follows:

**External Consistency:** Every process sees the same set of messages in the conversation. This has two aspects. First, all functioning processes reach the same decision about a failed (or suspected failed) process. Second, every functioning process starts accepting messages from a recovering participant at the same time.

**Internal Consistency:** Decisions made by the application based on the process' view of the context graph are correct. This has two aspects. First, stability and completeness decisions are made correctly; i.e., a message is considered stable only if it is followed by a message from all other functioning processes, and a wave is considered complete only when it has all of its messages. Second, processes receive all messages in the conversation.

We prove the correctness of the membership protocol by demonstrating that it guarantees both internal and external consistency.

### 3.2 Single Failures

Consider the case where at most one process fails at a time. Assume that $ML$ initially contains $n$ processes. The membership protocol is based on the effect that the failure has on the context graph. In particular, since a process obviously sends no messages once it has failed, it can be guaranteed there is no message from the failed participant at the same logical time as the membership protocol's initiation message sent by the monitor protocol. If, on the other hand, there *is* a message from the suspect process at the same logical time as the initiation message, then it can be viewed as evidence that the process has in fact not failed. In this case, it is likely that the original suspicion of process failure was caused by

these other processors. This propagated message contains the ids of all the messages upon which the new message depends; i.e., it identifies the nodes to which the new message is to be attached to the context graph. Psync recovers from network failures by recognizing when a new message is to be attached to a message that is not present in the local copy of the graph. In this case, Psync asks the processor that sent the new message to retransmit the missing message. That processor is guaranteed to have a copy of the missing message because it just sent a message in the context of it.

Psync provides only minimal support for recovering from a process failure, where by process failure we mean that the processor on which the process is running has crashed without undergoing any erroneous state transitions. Specifically, Psync supports three operations that affect the local definition of $ML$. Note that these operations are purely local; they do not affect the definition of $ML$ at other processors. First, the local process can instruct Psync to ignore a particular process; i.e., to stop accepting messages from that process. This action does not affect the definition of $ML$, but rather makes it look as though messages sent by the ignored process have been dropped by the network while being delivered to the local processor. Second, the local process can tell Psync to maskout a certain process. This operation actually removes the process from the local definition of $ML$; it also has the same affect as ignoring a message. Third, the local process can tell Psync to maskin a certain process. This has the effect of returning the process to the local definition of $ML$ and accepting future messages sent by that process.

In addition, a process can instruct Psync to restart a conversation. This operation is used to initiate recovery actions following a failure. Restarting a conversation has two effects: to inform other processes that the invoking process has restarted, and to initiate reconstruction of the local copy of the context graph. Psync accomplishes this by broadcasting a special restart message. When this message is received at a processor, the local instance of Psync performs two actions. First, it transmits its current set of leaf nodes to the participant that generated the restart message; this allows reconstruction of the lost context graph image using the standard lost message protocol. Second, it generates a local notification of the restart event; this is implemented as an out-of-band control message that is delivered to the local process.

## 3 Basic Protocol

As outlined in the Introduction, the purpose of the membership protocol is to reach agreement about the failure or recovery of a process. The protocol is initiated when one of these two events is suspected. In our scheme, the task of detecting these events is assigned to another protocol called the *monitor protocol*. Specifically, this protocol monitors the messages exchanged in the system and on suspecting

three participants, denoted $a$, $b$ and $c$, where $a_1, a_2, \ldots$ denotes the sequence of messages sent by process $a$, and so on.

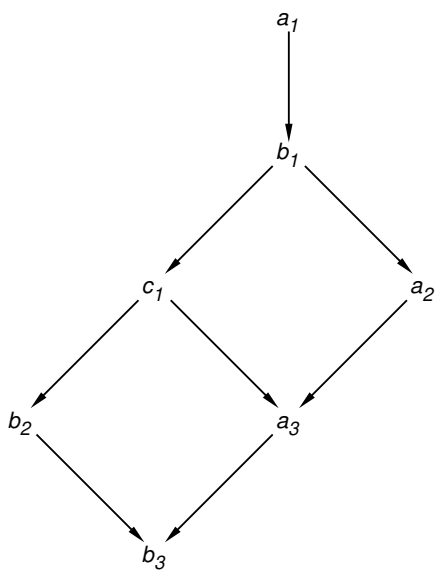


Figure 1: Example Context Graph

Several important terms can be defined relative to the context graph. First, if there is a path in the context graph from one message to another, then the former message is said to *precede* the latter message; for example, $a_1$ precedes $b_3$. Second, a pair of messages that are not in the context of each other are said to have been sent at the *same logical time*; for example, $c_1$ and $a_2$. Third, a message sent by a process is said to be *stable* if it precedes a message sent by all the other participants; for example, messages $a_1$, $b_1$, and $c_1$ are the only stable messages in Figure 1. Finally, a *wave* is a set of messages sent at the same logical time, that is, the context relation does not hold between any pair of messages in a wave. A wave is known to be *complete*—i.e., a process inspecting the context can be certain that no future messages will arrive that belong to the wave—as soon as a single message in the wave becomes stable. This is because for a message to be stable implies that all processes other than the sender have received it, which in turn implies that all future messages sent to the conversation must be in the context of the stable message; they cannot precede or be at the same logical time as the stable message.

Psync maintains a copy of a conversation's context graph at each processor on which a participant in the conversation resides. A distinct copy of the membership list $ML$ is also maintained at each such processor. For simplicity, we assume only a single participating process is running on each of these processors. Each time a process sends a message, Psync propagates a copy of the message to each of

typically by a *membership protocol*, an agreement protocol that is executed by the cooperating processes when the failure or recovery of another process is suspected. One factor that adds to the complexity of such protocols is that the message ordering mechanisms provided by a multicast service and the membership service are not independent. For example, to order messages for a replicated object application consistently, the current set of running processes must be known [**?**].

The membership protocol is based on a the Psync multicast mechanism [**?**]. Like other multicast primitives, Psync is targeted at the problem of maintaining a consistent ordering of messages exchanged among a collection of processes in the presence of failures. Unlike many others, however, Psync maintains a *partial* ordering of messages rather than a total ordering. Our membership protocol takes advantage of this property to provide a more efficient implementation. In particular, it allows different processes to conclude that a failure has occurred at different times relative to the sequence of messages received, thereby reducing the amount of synchronization required. This increases the concurrency of the system, while still preserving the semantic correctness of the application. The advantages of our approach are especially pronounced if multiple processor failures occur concurrently.

This paper is organized as follows. Section 2 provides a brief overview of the Psync multicast mechanism. The basic membership protocol is then presented in Section 3; this protocol handles a single failure or recovery event at a time. Section 4 extends the protocol to handle multiple concurrent events. It also argues the correctness of the algorithm. Finally, Section 5 offers some conclusions.

## 2 Overview of Psync

Psync supports a *conversation* abstraction through which a collection of processes exchange messages. The key idea of the abstraction is that it explicitly preserves the partial order of the exchanged messages. This section gives a brief overview of Psync, introducing only those terms and concepts critical to understanding the membership protocol. A more complete description of Psync can be found in [**?**].

A conversation is explicitly opened by specifying a set of participating processes. This set is also called the *membership list*, $ML$. A message sent to the conversation is received by all the processes in $ML$. Fundamentally, each process sends a message in the *context* of those messages it has already sent or received. Informally, "in the context of" defines a relation among the messages exchanged through the conversation. This relation is represented in the form on a direct acyclic graph, called a *context graph*. Psync provides a set of operations for sending and receiving messages, as well as for inspecting the context graph. For example, Figure 1 depicts the context graph associated with a conversation that has

# A Membership Protocol Based on Partial Order[1]

Shivakant Mishra
Larry L. Peterson
Richard D. Schlichting

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

## Abstract

A membership service provides a consistent, system-wide view of which processes are currently functioning or failed in a distributed computation. This paper describes a *membership protocol* that implements this service. Our protocol is novel because it is based on a multicast facility that preserves only the partial order of messages exchanged among the communicating processes. Because it depends only on a partial ordering of messages rather than a total ordering, our protocol requires less synchronization overhead. The advantages of our approach are especially pronounced if multiple failures occur concurrently.

Keywords: distributed systems, membership service, dependable computing

## 1 Introduction

The construction of highly dependable applications is often simplified by the use of suitable system abstractions. For distributed systems, many of the most important abstractions—sometimes referred to as *highly dependable services*—are related in some way to providing consistent information to a collection of cooperating processes despite failures. One example is a dependable clock service, which can be viewed as a global (virtual) clock that will return consistent values to all processes despite failures [**?**, **?**, **?**]. Another example is a reliable multicast or atomic broadcast service, which typically guarantees reliable dissemination of messages and *order*, i.e., that all processes receive messages in the same sequence. In both cases, it is the consistency of the information that is the key; as has been amply demonstrated, such a property facilitates construction of dependable systems by providing all processes with identical information upon which to make decisions [**?**, **?**, **?**, **?**].

This paper focuses on a third service used to provide consistent information in a distributed system: a dependable membership service [**?**, **?**, **?**]. This service provides a consistent, system-wide view of which processes are functioning and which have failed at any given moment in time. Implementation is

[1]This work supported in part by National Science Foundation Grants DCR-8402090, CCR-8701516, and CCR-9003161.