Preserving and Using Context Information in Interprocess Communication

Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting[1]
University of Arizona

When processes in a network communicate, the messages they exchange define a partial ordering of externally visible events. While the significance of this partial order in distributed computing is well understood, it has not been made an explicit part of the communication substrate upon which distributed programs are implemented. This paper describes a new interprocess communication mechanism, called *Psync*, that explicitly encodes this partial ordering with each message. The paper shows how Psync can be efficiently implemented on an unreliable communications network, and it demonstrates how conversations serve as an elegant foundation for ordering messages exchanged in a distributed computation and for recovering from processor failures.

1 Introduction

An interprocess communication (IPC) mechanism provides an abstraction through which processes that do not necessarily share an address space can exchange messages. While there exists considerable experience with IPC mechanisms for one-to-one communication—examples of such mechanisms include datagrams, virtual circuits, remote procedure calls [5], and channels [11]—much less is understood about IPC mechanisms for many-to-many communication. Work in this area includes low-level broadcast protocols [7] and high-level programming toolkits [3,4].

This paper introduces a new IPC protocol, called *Psync* (for "pseudosynchronous"), that supports the exchange of messages among a well-defined set of processes. Psync explicitly preserves the partial ordering of messages exchanged among a collection of processes in the presence of communication and processor failures. Because of the fundamental nature of this partial order, Psync has several desirable characteristics: it can be implemented on an unreliable network with performance comparable to conventional one-to-one protocols like UDP [19] and TCP [28], it supports elegant implementations of a wide range of existing communication protocols, it allows applications to directly access information not made available by other IPC mechanisms, and it facilitates recovery from processor failure.

[1]Authors' address: Department of Computer Science, University of Arizona, Tucson, AZ 85721. This work supported in part by National Science Foundation Grants DCR-8402090 and CCR-8701516, and Air Force Office of Scientific Research Grant AFOSR-84-0072.

Psync is a low-level protocol designed to support a variety of high-level protocols and distributed applications. This design has two implications [23]. First, Psync makes very few assumptions about the underlying network. For example, it does not assume expensive mechanisms such as reliable broadcast are available. Second, Psync defers to higher-levels any functionality that not all applications need. In other words, Psync only maintains the partial order among messages; a collection of "library routines" enforce various ordering discipline using Psync.

The paper is organized as follows. The next two sections describe Psync: Section 2 gives a formal definition of the abstraction upon which Psync is based and Section 3 describes an algorithm for implementing Psync in a distributed system. By analogy with virtual circuits, we first define a FIFO queue and we then describe the sliding window protocol by which a queue can be implemented on two processors connected by an unreliable network. Section 4 then demonstrates several applications of Psync and Section 5 shows how Psync can be extended to support the reintegration of failed processes into an ongoing conversation. Finally, Section 6 reports on the performance of Psync, Section 7 comments on related work, and Section 8 offers some conclusions.

2 Abstraction

Psync is based on a *conversation* abstraction that provides a shared message space through which a collection of processes exchange messages. The general form of this message space is defined by a directed acyclic graph that preserves the partial order of the exchanged messages. For the purpose of this section, we view a conversation as an abstract data type that is implemented in shared memory; Section 3 gives an algorithm for implementing a conversation in an unreliable network.

A conversation behaves much like any connection-oriented IPC abstraction: A well-defined set of processes—called *participants*—explicitly open a conversation, exchange messages through it, and close the conversation. Only processes that have been identified as participants may exchange message through the conversation, and this set is fixed for the duration of the conversation. Processes begin a conversation with the operations:

```
conv = active_open(participant_set)
conv = passive_open(pid)
```

The first operation actively begins a conversation with the specified set of participants. The operation creates an empty conversation—i.e., one that contains no messages—and the invoking process is not blocked. The second operation passively begins a conversation. The argument identifies

the invoking process. This process is blocked until some active process starts a conversation that contains the invoking process in its participant set. Pending conversations for a process—those for which the process has not invoked `passive_open`—are queued on the `passive_open` operation. The `conv` returned by the two operations can be thought of as an external handle for that process' view of the conversation. A process closes its view of a conversation with a

```
close(conv)
```

operation.

Once a process possesses a `conv` handle, it can send and receive messages using the operations:

```
node = send(msg, conv)
node, msg = receive(conv)
```

where `msg` is an actual message—an untyped block of data—and `node` is a handle or capability for that message. Each participant is able to receive all the messages sent by the other participants in the conversation but it does not receive the messages it has sent. 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*. The semantics of `send` and `receive` are defined in terms of this graph.

Formally, let P denote the set of participants in a conversation and let M denote the set of messages they exchange. Each element of M encapsulates both the actual message and the sender's identity. Define $\prec$ (read "precedes") to be a transitive relation on M, such that $m \prec m'$ if and only if message m' is sent in the context of message m; i.e., the process that sent m' had either sent m or already received m. Let $G_\prec$ denote the directed acyclic graph representation of $\prec$. A context graph, denoted $G = (M, E)$, is taken to be the transitive reduction of $G_\prec$ [2]. That is, G contains all the vertices and none of the *redundant* edges of $G_\prec$, where edge (m, m') is redundant if $G_\prec$ also contains a path from m to m' of length greater than one.

Figure 1 gives $G_\prec$ and G for a conversation in which m_1 was the initial message of the conversation; m_2 and m_3 were sent by processes that had received m_1, but independent of each other; m_4 was sent by a process that had received m_1 and m_3, but not m_2; and m_5 was sent in the context of all the other messages. We refer to the nodes to which a given message is attached in G as the message's immediate *predecessors*. For example, m_2 and m_4 are the immediate predecessors of m_5.

Also, two messages that are not in the context of the other are said to have been sent at the *same logical time*. For example,m_2 and m_3 were sent at the same logical time.

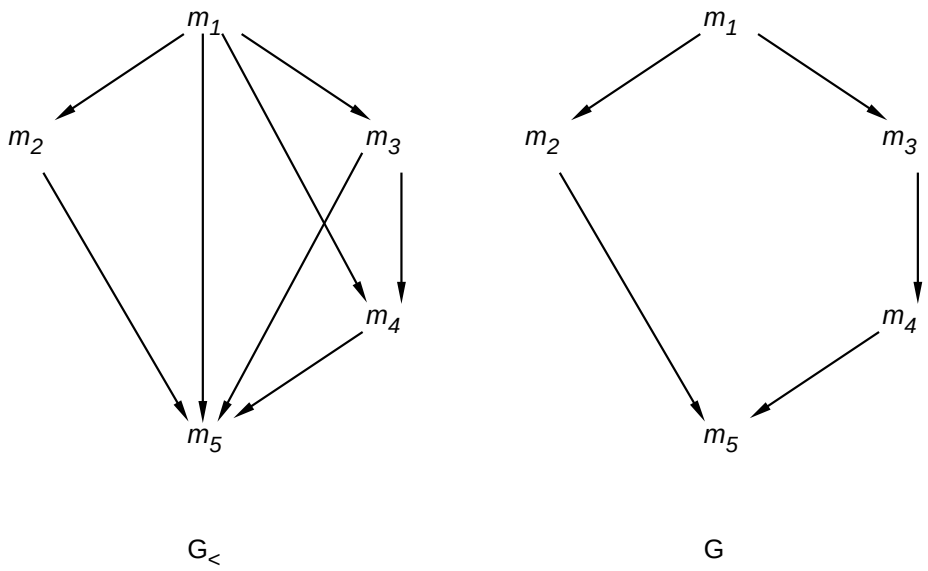

Figure 1: Example Context Graph

Each participant in a conversation has a *view* of the context graph that corresponds to those messages it has sent or received. Let $M_p \subseteq M$ denote the subset of messages sent or received by participant $p \in P$. Process p's view, denoted V_p, is a restriction of G to the vertices in M_p and the edges in E incident upon those vertices. A process with a view equal to G has received all the messages sent by other participants. Messages outside the participant's view—i.e., those in the set $(M - M_p)$—are said to be *outstanding*. For example, at the time a participant sent m_4, its view consisted of m_1 and m_3; m_2 was outstanding.

When process p invokes the `receive` operation, the "earliest" outstanding message is returned. Formally, `receive` returns an outstanding message m in G such that there is no other outstanding message m' for which $m' \prec m$. Also, V_p is extended to include m. The `receive` operation blocks if there are no outstanding messages. The abstraction has the important property that for any pair of messages m and m' received by a process, m is received before message m' if $m \prec m'$. Thus, when a process receives a given message, it is guaranteed to have already received all messages that precede it in the context graph. For example, $(m_1, m_2, m_3, m_4, m_5)$, $(m_1, m_3, m_2, m_4, m_5)$, and

$(m_1, m_3, m_4, m_2, m_5)$ are all valid total orderings for returning the messages in Figure 1, where different participants might see a different ordering.

When process p applies the `send` operation to message m, m is added to M and the edge (m_l, m) is added to E for each node m_l that is a leaf of V_p. Also, p's view is extended to include m, even though a participant never receives a message it sent. Note that the data structures that represent a conversation include a single "shared" context graph and a "private" view for each participant. It is therefore accurate to think of the context graph as the principle data structure and each view as a window on G. That is, while the leaves of V_p are used to determine how a message sent by p is inserted into the conversation, the message is attached to G, and as a consequence, available for the other participants to receive.

Notice that the `send` and `receive` operations modify the context graph in such a way that G remains the transitive reduction of $G_{\prec}$. Two conditions must be satisfied for this to be true. First, it must be the case that a path from m_1 to m_2 in G implies $m_1 \prec m_2$. To see that this is the case, observe that a given process always receives message m before m' if $m \prec m'$. As a consequence, the participant's view is always a connected subgraph of G, and it is therefore not possible to send a message in the context of one message that is not also in the context of all messages that precede it in the graph. Second, it must be the case that the existence of a path of length greater than one from node m_1 to node m_2 implies that there cannot exist an edge from m_1 to m_2. To see that this is the case, observe that each new message is attached to the leaves of the sending participant's view. Because these nodes are leaves, there cannot also be a path through another node to the new node.

Also note that a message sent by a given process is, by definition, in the context of the previous message sent by that same process. Therefore, it is not possible to have an edge in G leading from a given message to two or more messages sent by the same participant. Likewise, a given message cannot have two or more immediate predecessors sent by the same participant. These two observations imply that the outdegree and indegree of any node in G is bounded by the number of participating processes.

The context graph contains information about which processes have received what messages. In particular, receipt of a message implies that the sender has seen all its predecessor messages. Thus, if some message m is followed in the context graph by a message from all the participants except for m's sender, then m is necessarily in each participant's view. Formally, message m_p sent by process p is said to be *stable* if for each participant $q \neq p$, there exists vertex m_q in G sent by q,

such that $m_p \prec m_q$. Intuitively, each m_q serves as an acknowledgement of m_p from some process q. Because for a message to be stable implies that all processes other than the sender have received it, it follows that all future messages sent to the conversation must be in the context of the stable message; i.e., they cannot precede or be at the same logical time as the stable message.

For example, suppose the context graph depicted in Figure 2 is associated with a conversation that has 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. Messages a_1, b_1, and c_1 are the only stable messages. Also, participant a has sent two unstable messages: a_2 and a_3.

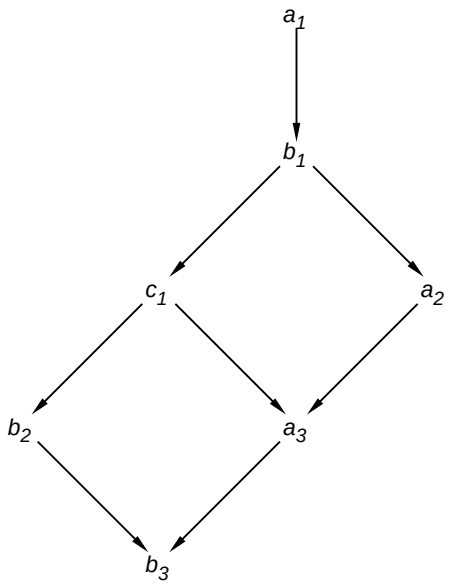

Figure 2: Another Example Context Graph

Because the context graph provides such useful information, the conversation abstraction supports the following operations for traversing G and querying the state of nodes in G:

`node = root(conv)`: root vertex of V_p.

`node_set = leaves(conv)`: set of leaf vertices of V_p.

`process_id = sender(node)`: process that sent `node`.

`participant_set = participants(conv)`: set of participating processes.

`node_set = next(node)`: set of vertices to which there is an edge from `node` in V_p.

`node_set = prev(node)`: set of vertices from which there is an edge to `node` in V_p.

`outstanding(conv)`: true if $V_p \neq G$.

`precedes(node`$_1$`, node`$_2$`, conv)`: true if V_p contains a path from `node`$_1$ to `node`$_2$.

`stable(node, conv)`: true if `node` is stable.

3 Protocol

This section describes the Psync protocol (algorithm) that implements conversations in a distributed system. While more than one implementation strategy is possible—for example, sending and receiving messages could be implemented as atomic transactions on replicated copies of the context graph—Psync replicates G throughout a network in a way that preserves the important properties of the conversation abstraction without incurring the high cost of atomic updates. Psync is designed this way because it is intended to be a low-level IPC protocol upon which a wide range of other mechanisms can be built.

To simplify the discussion, we describe the protocol in three stages. First, we present a basic protocol that accommodates varying communication delays; this description assumes an infinite amount of memory at each processor. Second, we augment the basic protocol to account for network and host failures; this discussion also assumes infinite memory. Finally, we remove the infinite memory assumption by considering garbage collection and flow control. Throughout the discussion, Psync uses internal identifiers to denote the three basic objects: it assigns a network-wide unique *cid* to each conversation, it assigns a conversation-wide unique *mid* to each message, and it uses a network-dependent *pid* to identify each participant. For simplicity, we assume each *pid* can be divided into a host part and a local part; i.e., it is possible to determine the host on which a process resides given its *pid*.

3.1 Basic Protocol

We begin by describing the implementation of a conversation on a set of hosts connected by an asynchronous message passing facility with varying communication delays between hosts. For the purpose of this discussion, assume no network or host failures.

3.1.1 Distributed Images

Psync maintains a copy of a conversation's context graph G at each of a set of hosts on which a participant in P resides. The copy of G on host h is called an *image* and is denoted I_h. Psync at each host also maintains the view for each local participant. For simplicity, assume there is a one-to-one relationship between hosts and participants; i.e., there is a single image and a single view at each host. While context graph G still exists in the abstract, in practice only the individual images are implemented, and Psync only guarantees that the union of the I_h for all hosts h is equal to G given no host failures; it does not attempt to keep all the images equivalent. Messages in the abstract context graph G but not in some image I_h correspond to messages sent by some participant that are still in transit to host h. Messages in I_h but not in the local participant's view correspond to messages that have arrived at host h but have not yet been received by the participant.

3.1.2 Opening and Closing Conversations

The `active_open` operation creates an empty local image, but no messages are exchanged and the invoking process is not blocked. The information necessary to establish the conversation at those hosts on which a process invoked the `passive_open` operation is piggybacked on the first message sent by the process that actively opened the conversation. The arrival of this message at a given host initializes the local image, which in turn causes the local participant's invocation of `passive_open` to complete. The format of a conversation's first message, called an `SC` (start conversation) message, is given in Figure 3; $pid_1 \ldots pid_n$ identifies the participating processes (pid_1 is the message sender) and the message's *mid* is the same as the conversation's *cid*.

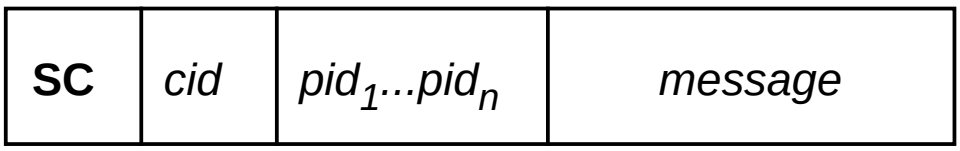

Figure 3: Start Conversation Message

Psync exchanges no messages when a process closes a conversation. Therefore, it is possible for a process to close its view of a conversation before the other processes are finished sending messages, implying that new messages may arrive later for that conversation. From the perspective

of the remaining processes, the process that closed the conversation too early will appear to have failed (see Section 3.2). We expect applications for which such early closings are not acceptable to implement a "termination agreement" protocol on top of Psync.

3.1.3 Sending and Receiving Messages

When process p on host h invokes the `send` operation, the new message is attached to image I_h according to the definitions given in Section 2 and a copy of the message—along with information specifying the edges that connect the message to the context graph—is propagated to each remote host. This message can be delivered using either a point-to-point delivery mechanism or a broadcast mechanism. When process p on host h executes the `receive` operation, an outstanding message from I_h is returned. The `receive` operation blocks until I_h contains an outstanding message.

The format of each message sent to an existing conversation, called an `AN` (add node) message, is given in Figure 4; each *pred_mid* is the unique identifier for one of the message's immediate predecessors in the context graph and pid_{sender} identifies the sending participant. Note that the number of predecessor messages identified in the message is bounded by the number of participants in the conversation, corresponding to the bound on the indegree of each vertex. Each `AN` message that arrives at a host h for which the predecessor message set is present in I_h is immediately inserted in I_h. If one or more of the predecessor messages have not yet arrived, then the message is placed in a holding queue until all the predecessors are present. Such messages are not considered attached to I_h, and therefore cannot be returned by the `receive` operation. When all preceding messages have arrived, the earlier message is removed from the holding queue and incorporated into I_h. Observe that each message in G is contained in at least its sender's image, even though the multiple images of G are not equivalent while messages are in transit.

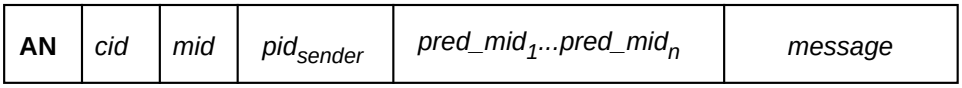

Figure 4: Add Node Message

Finally, note that the Psync operations that allow a process to inspect the context graph are defined relative to the participant's view and the local image, not in terms of the abstract graph G. For example, the `stable` operation reports on the stability of a message in a given participant's

view, where stability in V_p implies stability in G.

3.2 Failures

Implementing conversations in a distributed environment is in practice complicated by three factors: the underlying network fails to deliver messages, hosts fail, and host failures are indistinguishable from network partitions and hosts that are slow to respond. This section extends the basic protocol to account for these factors. For the purpose of this section, we assume that when a host does in fact fail, it remains failed for the duration of the conversation; techniques for recovering and reintegrating failed processes into an on-going conversation are described in Section 5.

3.2.1 Transient Network Failures

Consider the possibility of transient network failures. Such failures imply that for a given message sent from one host to another, zero or more copies of the message are delivered to the destination host. For the purpose of this discussion, assume hosts do not fail.

Recall that Psync places any message received out-of-order in a holding queue until all messages upon which it depends arrive. Let m be a message sent by a participant on host h in the context of m', and let h' be a host that receives m but has not yet received m'; i.e., m is placed in the holding queue on h'. Psync associates a timer with each message in the holding queue. When the timer for message m expires, a request to retransmit m' is sent to h. That host is guaranteed to have m' in its image because a local participant just sent a message in the context of m'. This is true even if the participant that originally sent m' does not reside on h.

The retransmission request, called an `RR` message, is schematically depicted in Figure 5. Because it is possible that the predecessors' predecessors are also missing, the retransmission request identifies the subgraph of G that needs to be retransmitted, not just the message(s) known to be missing. The set of $leaf\_mid$'s identify the current leaves of the local image and mid identifies the message whose predecessors are missing. The leaf set and the last received message effectively define the boundary of the missing portion of G. When a host receives an `RR` message, it responds by resending all messages between the leaf set and the out-of-order message, exclusive. If a mid is not given, the host responds with all messages sent in the context of the leaf set. An empty leaf set implies that the root node(s) should be retransmitted.

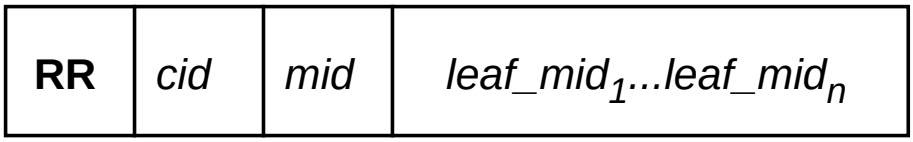

Figure 5: Retransmission Request Message

3.2.2 Last ACK Problem

Although Psync automatically recovers from missing messages upon which some other message depends, it is possible for the last message sent—i.e., a message upon which no messages depend—to be lost. We characterize this as an instance of a general "last ACK problem" faced by many protocols. To help applications accommodate this possibility, Psync is augmented to allow its blocking operations—`passive_open` and `receive`—to include a timeout argument. The return code then indicates whether the operation was successful or the timeout expired. Processes use a timeout larger than the maximum communication delay to and from all participating hosts.

In addition, Psync provides a

```
resend(node)
```

operation. Applying this operation to a `node` causes an exact duplicate of the corresponding message to be sent to all hosts maintaining an image of G. The resent version of the message is identical to the original copy of the message—i.e., it is an `SC` or an `AN` message with the same *mid*—except for that it is flaged as having been resent. Should a host that receives a resent message already have a copy of the message, it (1) discards the duplicate copy, and (2) resends all the messages in its image that are immediate successors of the duplicate message. Finally, should a participant apply `resend` to a stable message, Psync does nothing; i.e., it does not resend the message as instructed. This is because resending a stable message is unnecessary: by definition, a stable message has been delivered to all participants and a reply has been received from all participants.

The `resend` operation is used by a process that has reason to believe a message it sent earlier was never delivered; i.e., if it sent a message and timed-out while waiting for a reply message. A generalization of waiting for a reply message is to wait for a message to become stable. One can therefore implement a "synchronous send" routine that does not return until the sent message has stabilized. If the message does not stabilize because it was not delivered to all applications, then the

routine would resend it several times. Figure 6 defines a `send_stable` routine as a library protocol implemented on top of Psync. Note that if `send_stable` returns `FAILURE`, the application is likely to conclude that one or more hosts have failed (see Section 3.2.4). Also note that `send_stable` works correctly if one of the reply messages, as opposed to the sender's message, was lost. This is because if a host receives a duplicate copy of the resent message, it responds with all the messages that immediately depend on the resent message; i.e., Psync automatically resends the reply message.

```
send_stable(message, conv)
{
     send_node = send(message, conv);
     for (try=0; try<LIMIT; try++)
     {
          node, msg = receive(conv, timeout);
          while (outstanding(conv))
               node, msg = receive(conv, timeout);
          if (stable(send_node))
               return(SUCCESS);
          else
               resend(node);
     }
     return(FAILURE);
}
```

Figure 6: Library Routine for Sending a Stable Message

3.2.3 Host Failures

Now consider the effect host failures have on the maintenance of the context graph. For the purpose of this discussion, assume hosts fail silently without undergoing incorrect state transitions or generating spurious messages; it is not necessary that such failures be accurately detectable.

Psync guarantees two things about the context graph in the presence of host failures:

- All running processes are able to continue exchanging messages.
- A message contained in any running host's image will eventually be incorporated into every running host's image if host failures are infrequent.

The first condition is easy to guarantee because each process depends only on the local state of the conversation. Thus, a participant can successfully invoke `send` because being able to send

a message depends only on the leaves of the participant's view. Also, a participant's ability to successfully `receive` messages sent by another running process depends only on the host's ability to incorporate new messages into the local image. The host, in turn, can always incorporate messages received from another running host into its image because the only prerequisite for doing so is that all the predecessor messages be present. Should some of the predecessor messages not be present, the receiving host can retreive them from the sending host. The sending host is guaranteed to have all the preceding messages because it just sent a message that depends on them.

The key to satisfying the second condition is to correctly deal with a host failing after it has sent a message. Psync addresses this problem with the following extension to the retransmission request strategy defined in Section 3.2.1: When a host does not receive a response to an `RR` message that it sent to a particular host, it broadcasts the `RR` message to all the hosts. Should the broadcast `RR` message fail to yield the missing message, the message that triggered the retransmission request is discarded. Given this extension to the protocol, consider how the second condition is satisfied for two different quantifications of "infrequent".

First, assume a single host failure. Without loss of generality, suppose host h fails immediately after sending message m in the context of message m'. There are three cases to consider.

- *Case 1:* No other host receives m. Message m does not appear in any running host's image.
- *Case 2:* All hosts receive m.
  - *Subcase a:* No host has m' in its image; thus, the broadcast `RR` fails. Neither messages m' nor m appear in any host's image. Note that m' must have been sent from host h, otherwise at least one running host (the sending host) would have a copy of it.
  - *Subcase b:* All hosts have m' in their image. Message m can be successfully incorporated in each host's image.
  - *Subcase c:* Some hosts have m' in their image. Broadcasting the `RR` message will retrieve m' and both m and m' will be incorporated into each host's image.
- *Case 3:* Some hosts receive m. A host that receives m incorporates it into its image as in case 2. A host that does not receive m will at some future time receive message m'' in the context of m, causing the host to retreive m from the host that sent m''.

Thus, the same set of messages are incorporated into all images when a single host fails.

Second, suppose there are multiple host failures. Psync continues to incorporate messages into all images unless there are "too many" failures, where "too many" is precisely quantified as follows. A message m is defined to be *n-stable* if n-1 processes other than the sender of m have sent a message in the context of m. For a message to be n-stable implies that a copy of m is contained in

at least n images, assuming a one-to-one correspondence between images and processes. Thus, a copy of m can be retrieved from some image in the presense of up to n-1 host failures. A message that is stable is contained in all images.

Note that the preceding discussion does not assume perfect knowledge of when a particular host has failed; i.e., it can be implemented using a simple timeout and retry strategy. In the worst case, a given host might decide that another host is down when it is not, but this does not affect the correctness of the protocol. For example, suppose a host that receives m incorrectly decides that h is down. Sending the broadcast `RR` message is wasteful but not incorrect. As another example, suppose a host that receives m decides to ignore m' and all the messages that depend on it (case 2b), but some host that has a copy of m' is still running. A new message will eventually arrive that directly or indirectly depends on m' and the recovery procedure outlined in Section 3.2.1 will be exercised.

3.2.4 Application-Level Support

From the application's perspective, host failure involves two issues: determining when a host has failed and deciding what to do about a failure. In the case of the first issue, Psync provides no explicit mechanism for detecting host failures. Instead, each participant determines on its own that some other process has failed. For example, a given participant might decide that a host has failed because a routine like `send_stable` returns `FAILURE`. In the case of the second issue, Psync allows processes on any subset of running hosts to continue exchanging messages when one or more other hosts have failed. Whether a given participant choses to stop executing or continue executing when it detects a host failure depends on the application.

For applications that chose to continue when processes fail, each participant must be able to remove the failed process from its definition of the participant set. This is necessary so that messages will eventually stablize relative to the currently running set of participants. In other words, if the failed participant is not removed from the working definition of P, then messages will never stabilize because a message from the failed participant will never arrive. Psync provides a

```
mask_out(participant)
```

operation for this purpose. A process invokes this operation to remove a participant from its working definition of P. Once a given participant has masked out some other participant p, Psync ignores (discards) all messages m_p received from p unless it has in its holding queue a message m_q from some participant $q \neq p$, such that m_q is in the context of m_p. An inverse operation,

`mask_in(participant)`

is provided to return a participant back into the local definition of P. Note that both operations "mask" the participant set; they do not permanently delete existing participants or add new participants.

Note that in practice it is impossible to determine with absolute certainty that a particular host has failed; it may be slow to respond or it may be isolated by a network partition. This is a critical observation because it is possible for a process that is thought to have failed to start sending messages again. As a consequence, it is necessary for the running processes to be able to agree as to when a particular process has failed. While Psync does not provide a direct mechanism for doing this, algorithms for agreement about failure have been developed [10] and they can be implemented on top of Psync, analogous to the `send_stable` routine. A more thorough description of a *delete protocol* that specifies the actions to be taken by functioning participants when another participant fails is presented elsewhere [18].

3.3 Memory Management

The previous discussion implies that the entire history of a conversation is maintained throughout the lifetime of the conversation. While preserving some or all the history is necessary if failed processes are allowed to rejoin as described in Section 5, in many cases maintaining the entire context graph is unnecessary. This section outlines how to garbage collect portions of the context graph and how to implement flow control.

It is useful to think of each node in an image as having two parts: an entry in the data structure that implements the graph—this entry is a few dozen bytes long and contains the message sender, the message's id, pointers to other nodes, and so on—and a buffer that holds the message itself. In the case of the actual message, the buffer is reclaimed as soon as the corresponding node becomes stable. This is because a host cannot be asked to retransmit a message that is stable; such messages are already contained in all images of G. In the case of the graph node, no simple rule exists. This is because an application process may inquire about any node in the graph; e.g., it may apply the `msg_sender` operation to an arbitrary node. While reclaiming graph nodes is not as critical as reclaiming message buffers, some mechanism is necessary if conversations are to support arbitrarily many messages. One solution is to provide a `free_node` operation that explicitly causes a particular node, along with all of its predecessors to be reclaimed. An application would invoke this operation whenever it finishes with a particular portion of the context graph. Another solution is let the

application set some threshold parameter Θ, such that the application is only permitted to invoke operations on the last Θ messages sent to the conversation. This latter approach is practical because applications can reasonablely chose a value for Θ that is proportional to the number of participants in the conversation.

In addition, Psync has three flow-control limitations. First, because it is possible for an application to send many messages without any of them becoming stable, Psync limits the amount of buffer space allocated to each conversation; the `send` operation blocks and newly arriving messages are discarded if this limit is exceeded. Second, only a fixed number of pending conversations[2] are allowed to queue for any single process, where only one message associated each such conversation is stored; all additional messages belonging to a pending conversation are discarded. Third, only a fixed number of out-of-order messages are saved in each conversation's holding queue; additional messages are discarded. Note that in all three cases, newly arriving messages that exceed buffer limits are simply discarded since discarding a message is indistinguishable from a transient network failure. As a consequence, the mechanisms described in Section 3.2 are later used to recover the messages.

3.4 Remarks

Psync has been designed to include only that functionality essential to maintaining context information; all other functionality has been pushed onto higher-level protocols. For example, rather than support a `send_stable` operation, we have built a library version of the operation on top of Psync. As another example, rather than support a conversation-wide operation for removing a failed processes, Psync provides only `mask_out` and `mask_in` operations that modify the local definition of P; one can build conversation-wide `remove_process` and `add_process` routines on top of Psync. Other useful library routines include a `quorum_stable` routine that determines if a majority of processes have received and responded to a particular message, and `initialize` and `terminate` routines that employ a three-way handshake protocol to begin and end a conversation. The important point is that the list of useful library routines is both large and diverse. Because different applications use different combinations of these routines, we chose to implement them on top of Psync rather than embed them in Psync. In other words, our design draws an explicit line between the *mechanism* that preserves ordering information and higher-level protocols that enforce

[2] A pending conversation is one that has been actively opened but for which the local process has not invoked a corresponding `passive_open`.

a particular ordering *policy*.

4 Ordering Messages

The context graph explicitly records the partial ordering of messages exchanged in a distributed computation. Participants enforce a particular ordering discipline on the context graph based on the requirements of the application. This section gives several examples of how the context graph supports elegant implementations of a variety of ordering policies. These policies can be thought of as "filters" placed on top of Psync.

4.1 Conventional Protocols

Psync supports efficient implementations of well-known communication protocols due to the fundamental nature of the context relation. For example, the unreliable datagram corresponds to a degenerate context graph that contains a single vertex, a reliable datagram causes an acknowledgement message to be sent in the context of a datagram, and an RPC mechanism sends a result message in the context of an request message and subsequent request messages in the context of previous reply messages. This section makes three observations about implementing conventional protocols on top Psync.

First, while one could argue that it would be more efficient to implement a virtual circuit protocol or an RPC protocol directly on the underlying network rather than on top of Psync, it is nontheless interesting to observe that the context graph provides a useful mental tool for thinking about such protocols. Consider, for example, a virtual circuit protocol. The context graph that models a virtual circuit grows in a "nearly linear" manner, where the breadth of the graph intuitively corresponds to the number of unstable messages sent by the local participant. A process stops sending data when the number of unstable messages it has sent exceeds the size of the circuit's sliding window. Thus, a linear context graph would result if a *stop-and-wait* protocol is employed [27]. Moreover, as long as a both sides have data to send, the act of sending a message in the context of received messages effectively acknowledges those messages, thereby providing a natural implementation of the *piggyback* optimization.

Second, Psync offers an alternative IPC paradigm to applications that currently use whatever existing IPC mechanism provides the "best fit", even if that mechanism does not provide exactly the semantics that the application needs. Consider, for example, a distributed program that ex-

hibits an interactive communication pattern in which a client process sends a request message, a server process replies, the client responds to the server's reply, and so on. Such a pattern is commonly called *conversational continuity*, and can be viewed as a generalization of the message transaction paradigm. The mail protocol SMTP is an example of an application that exhibits conversational continuity [21]. Psync is an ideal communication substrate for the conversational continuity paradigm because it maintains the desired connectivity from message to message without duplicating the efforts of the application. In contrast, virtual circuits—the IPC mechanism conventionally used for such applications, including SMTP—send an acknowledgement message for each application message, even though the application level response sent in the context of the request message is sufficient acknowledgement. While virtual circuits protocols are usually optimized to piggyback acknowledgements, such optimizations are a heuristic because the virtual circuit protocol has no knowledge of when or if the application will send its next message. Also, opening and closing a virtual circuit causes overhead messages to be exchanged even though the application is able to determine, based on its own state and the last message sent from the other process, that no more messages will be exchanged and the conversation can be safely closed.

Third, because Psync supports a many-to-many communication paradigm, its behavior is subtly different from conventional one-to-one protocols that have be augmented to support one-to-many (multicast) communication. Consider, for example, a simple message transaction in which a client sends a request message to a collection of servers, and one or more of the servers receive the request and sends a reply message [8]. Because Psync distributes all messages to all participants, the servers will receive each others reply messages. In contrast, only the client receives the reply messages in the case of a multicast. The former mechanism is desirable if a server is able to avoid doing unnecessary work because it can detect that another server has already responded.

4.2 Ordered Broadcast

As an example of how Psync provides an elegant base for implementing various ordering disciplines in a many-to-many communication paradigm, consider the following implementation of ordered broadcast. Such a broadcast ensures messages sent in a many-to-many communication are received by all participating processes in the same order. Ordered broadcasts are commonly used by a set of processes that are applying operations to a set of replicated data objects, where operations are encapsulated in messages. Because each process receives the messages (processes the operations) in the same order, they are able to maintain consistent copies of the object.

One typical implementation of an ordered broadcast is to assign a timestamp from a virtual clock to each message when the message is sent. The receivers then order the messages based on the timestamps. In contrast, Psync supports a partial ordering that can be used to give a total ordering if all participants do the same topological sort of the context graph. The topological sort must be incremental in the sense that each process waits for a portion of its view to stabilize before allowing the sort to proceed. This must be done to ensure that no future messages sent to the conversation will invalidate the total ordering. For simplicity, the following discussion distinguishes between the process that directly uses the context graph to implement the ordered broadcast (called the participant) and the application process that expects a total ordering of messages (called the application).

As schematically depicted in Figure 7, each participant's view is conceptually partitioned into *committed* and *uncommitted* subgraphs, denoted V_p^c and V_p^u, respectively. Dotted lines denote a path between two message nodes. Subgraph V_p^c corresponds to those messages that have been totally ordered and committed to the application. (Messages in V_p^c would also satisfy the definition of *queue stability* [23].) Subgraph V_p^u corresponds to the set of messages yet to be considered. Each iteration of the incremental topological sort moves through V_p in *waves*, where a wave is a maximal set of messages sent at the same logical time; i.e., the context relation does not hold between any pair of messages in a wave. As soon as the wave is known to be *complete*—i.e., the participant is certain that no future messages will arrive that belong to the wave—the messages in the wave are ordered according to some deterministic sorting algorithm and passed to the application. The messages in the wave are also moved from V_p^u to V_p^c. Note that defining a wave to be the roots of V_p^u results in a breadth first traversal of the context graph.[3]

The important remaining problem is determining when all possible roots of V_p^u are present. Recall that when a message is stable, all future messages must follow it in the context graph. Thus, a single stable message in a given wave implies that all possible members of the wave are contained in the participant's view. In other words, as soon as a single root of V_p^u becomes stable, all the roots of V_p^u can be sorted and committed to the application. In contrast, consider both a weaker and stronger condition for committing. On the one hand, it is not correct to commit a message as soon as it becomes stable. This is because the order in which messages become stable in two different views may differ due to varying communication delays, thereby resulting in potentially

[3]An alternative is to do a depth first traversal, in which case the entire disjoint branch of the context graph rooted at each node in the wave is committed in order.

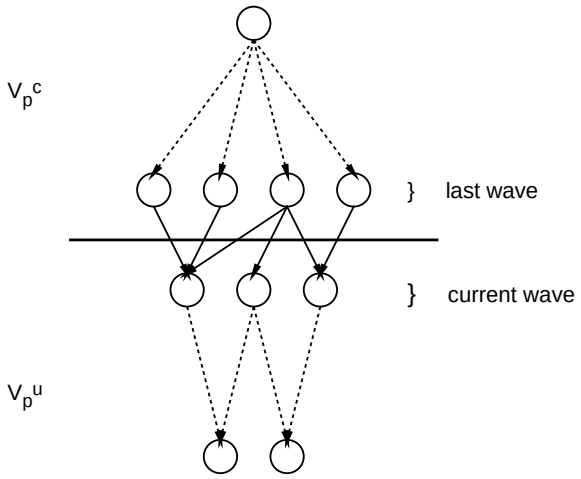

Figure 7: Committed and Uncommitted Partitions of V_p

different total orderings. On the other hand, it is not necessary to wait for all messages in the wave to become stable before committing the wave; a single stable message in the wave is sufficient.

Figure 8 gives the procedure `broadcast` that implements the algorithm just described. The procedure interfaces with the application process by a pair of message queues and the operations `snd_to_queue` and `rcv_from_queue`. A `wait_input` operation is used to allow the process to block waiting for input from multiple sources. To simplify the presentation, procedure `broadcast` does not include any error recovery code.

```
broadcast()
{
     conv, last_wave = initialize();
     while (TRUE)
     {
          snd_something = FALSE;
          rcv_something = FALSE;
          current_wave = ∅;
          for (each node ∈ last_wave)
               current_wave = current_wave ⋃ next(node);
          if (∃ node ∈ current_wave, s.t.  stable(node, conv))
          {
               last_wave = current_wave;
               sort(current_wave);
               for (each node ∈ current_wave)
```

```
                    snd_to_queue(msg_node);
               }
               wait_input();
               while (outstanding(conv))
               {
                    node,msg = receive(conv);
                    rcv_something = TRUE;
               }
               while (!empty(snd_queue))
               {
                    msg = rcv_from_queue();
                    send(msg, conv);
                    snd_something = TRUE;
               }
               if (!snd_something && rcv_something)
                    send(ACK, conv);
          }
     }
```

Figure 8: Ordered Broadcast Procedure

At the heart of the procedure are the two node sets `last_wave` and `current_wave`, corresponding to the leaves of V_p^c and the roots of V_p^u, respectively. When started, the procedure first calls an `initialize` routine similar to the one mentioned in Section 3.4. This routine also initializes `last_wave`. Next comes the algorithm's main loop. First, it adds all the known dependents of the nodes in `last_wave` to `current_wave`. Second, it checks to see if any of the nodes in `current_wave` are stable. If any are, `current_wave` is assigned to `last_wave`, the `sort` routine is called to order the messages in `current_wave`, and the sorted messages are sent to the application. Assume the `sort` routine weeds out any messages in `current_wave` that are not meant for the application—e.g., `ACK` messages—but it does not filter messages sent by the local process. The same `sort` routine must be applied by all participants; for example, it might sort the messages based on the sender's id. If none of the messages are stable, then the algorithm waits for new messages to arrive and checks the stability of `current_wave` the next time around the loop. Finally, any new input that has arrived is processed at the bottom of the main loop. Note that `rcv_from_queue` is invoked after receiving any outstanding messages from the conversation. This causes any new messages sent to acknowledge all the received messages. An explicit acknowledgement is sent only if a message is received but none are sent.

4.3 Replicated Objects

Although the total ordering of messages guaranteed by an ordered broadcast mechanism provides a foundation for synchronizing distributed computations, there are certain cases in which the same total ordering is not necessary at each host [13]. Suppose, for example, that a data object is replicated at n hosts, where a process running at each host manages the local copy. Furthermore, suppose that some of the operations that may be applied to the object are commutative with respect to other invocations of the same operation. In this scenario, the n processes can participate in a single conversation and implement operations on the data object by sending messages to the conversation. The partial ordering of messages (operations) preserved in the context graph is sufficient for ordering the commutative operations. The processes only have to synchronize with each other on the non-commutative operations, which they do by waiting for the corresponding message to become stable in the context graph. A detailed description of an algorithm that employs this idea is presented elsewhere [18].

To see how an ordering policy might take advantage of commutative operations, consider an object that supports operations α and and β, where multiple invocations of α can be executed in an arbitrary order with respect to each other. For example, α might insert an element into a set and β might perform some computation on the set and then clear the set. Because one is usually interested in applying the operations in an order that is consistent with the order in which the operations are invoked, the ordering policy is similar to the one given for the ordered broadcast in Section 4.2; that is, it moves through each participant's view in waves. The key difference is that we can gain additional concurrency by not waiting for the wave to be complete before executing some of the operations in the wave.

Consider the three graphs in Figure 9, where each message is denoted by the operation it represents and the previous operations that have been executed are omitted. Dotted lines denote a path between two message nodes. In (a), the current wave contains five operations that were invoked at the same logical time. Assuming the ordering policy gives preference to α operations over β operations, the local participant can execute all the α operations in any order before knowing that the wave is complete; i.e., before any message in the wave becomes stable. Once the wave is complete and all the α operations have been executed, the β operations can be sorted and executed serially. In general, it is possible for one or more other participants to not receive the β operations in the wave for some time, and for those participants to continue to invoke α operations, as depicted in (b). In this case, as long as those α operations do not depend on a β operation that has not yet

been executed, the local participant may continue to execute the α operations. Notice that newly arriving α operations continue to be executed even after a β operation is present in the local view. Finally, (c) shows the the condition that terminate the set of α operations: Each participant has a β operation is its view that has not been executed. Once this happens, all future operations sent to the conversation by that participant must be in the context of the β operation, and therefore must follow it in the total order of operations executed by that participant. The example illustrates two possible scenarios: a participant sends a β operation or it sends an α operation in the context of some other participant's β operation.

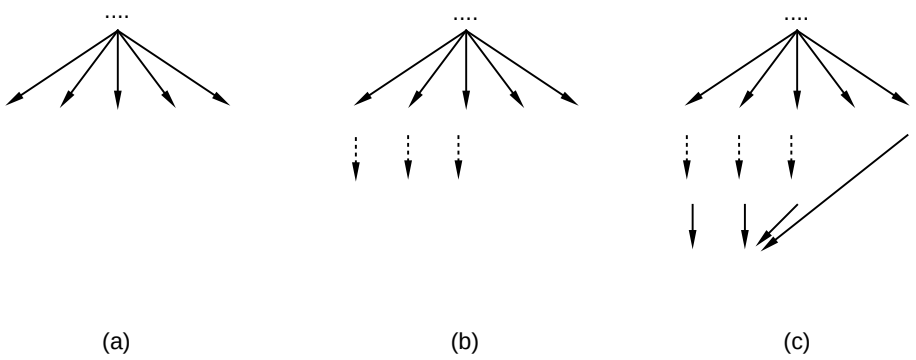

Figure 9: Example Operation Invocations

5 Reintegrating Failed Participants

When a processor fails, one or more participants may depart from an on-going conversation. Section 3 describes the Psync mechanisms that can be used to accommodate situations where the participants remain failed for the duration of the conversation. Although this situation is common, there are also cases where it is necessary for a participant to recover and rejoin a conversation. For example, the *two-phase commit protocol* used to maintain consistency between copies of a replicated database despite failures requires that the processes implementing the protocol recover to guarantee that changes to the database are applied to all copies [12].

Although the specifics of participant reintegration are highly application dependent, there are generally two tasks that must be accomplished before normal processing can continue. First, the functioning participants must be notified that the failed participant wants to be reintegrated into

the conversation. This notification facilitates the execution of an application-level *join protocol*, the inverse of the *delete protocol* mentioned in Section 3.2.4. The join protocol typically causes each functioning participant to return the recovering participant into its active participant set by invoking the `mask_in` operation. One example of such a protocol can be found in [18].

Second, an appropriate internal state of the participant must be restored. This state includes the application's local variables, the local image of the context graph, and the participant's view of the conversation. One common way to facilitate the restoration of the local variables is for the participant to *checkpoint* them onto non-volatile storage; the participant then reads the checkpoint upon host restart. Psync is responsible for restoring the local image. The application and Psync share responsibility for restoring the participant's view.

This section describes Psync support for participant reintegration. It first focuses on support provided for checkpoint-based techniques. It then discusses an alternative in which the internal state of the participant is recreated by rebuilding the local context graph image and reexecuting the process from its initial state.

5.1 Using Checkpoints

Recovery schemes that use checkpoints depend on the participant periodically writing its state to non-volatile storage. Following a failure, the participant reads this saved state to recover its local variables. It then executes the Psync `restart` operation to initiate recovery of the context graph. The form of this operation is as follows:

```
conv = restart(cid, pid, participant_set, leaf_mid_set)
```

Analogous to `active_open`, `restart` returns a handle for the conversation. The first argument is the system-wide unique identifier (*cid*) for the conversation, the second argument identifies the invoking participant, the third argument identifies the conversation's participant set, and the fourth argument gives the conversation-wide unique identifiers (*mids*) for the set of messages that are to form the leaves of the participant's view of the context graph upon recovery. Specifying the view is important because it defines the point at which the process starts receiving new messages. The `restart` operation is issued by a recovering participant in lieu of the standard operations for opening a new conversation. For example, it might be used in recovery code that is executed immediately upon processor restart instead of some standard prologue code charged with opening files and establishing conversations on the initial execution.

The values used as arguments to `restart` are typically included in the checkpoint so that they will be available following a failure. Psync provides operations that allow the application to retrieve the values into local variables. The participant set is retrieved by the `participants` operation described in Section 2. The *cid* and *mids* are retrieved using the following two operations

```
cid = get_cid(conv)
mid_set = get_mids(node_set, conv)
```

respectively. The `node_set` given as an argument to `get_mids` is the collection of nodes for which identifiers are desired; i.e., the set of messages the process wants to form the leaves of its view upon recovery. The related issues of which *mids* to include in the checkpoint and when checkpoints should be taken are addressed below.

The `restart` operation serves two purposes: to inform other participants that the invoking participant has restarted, and to initiate reconstruction of the local image of the context graph. Psync accomplishes this by sending a special `RS` (restart) message to all hosts on which a participant resides. The form of an `RS` message is shown in Figure 10. The *cid* field is the identifier for the conversation and the *pid* field identifies the invoking participant; both are given as an argument to `restart`.

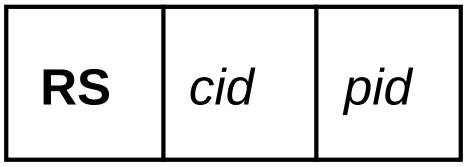

Figure 10: An `RS` Message

When an `RS` message is received at a host, the local instance of Psync performs two actions. First, it notifies the local participant of the restart event; this is implemented as an out-of-band control message that is delivered to the local participant. As outlined above, this notification usually results in the local participant returning the recovering participant to its active participant set using the `mask_in` operation. The notified process might also initiate the execution of an application-level join protocol.

Second, the local instance of Psync transmits the messages that make up the leaves of its context graph image to the participant that sent the restart message; these messages are sent as standard

`AN` messages. As these messages are received at the restarting host, the local instance of Psync reconstructs the lost context graph image according to the standard lost message protocol described in Section 3.2. That is, upon receipt of the first retransmitted messages m, Psync transmits an `RR` to the sender of m requesting the contents of the graph from the root to the node representing m. Should that request fail, the request is broadcast to all participants. Portions of the graph that are not in the context of m (e.g., siblings of m) are retrieved as required to fill in missing context of other messages as additional messages arrive from other hosts. Note that this procedure recovers the host's image of the context graph. Once the image has been recovered, the local participant's view is trivially reestablished according as specified by the set of *mid*s given as an argument to `restart`.

It is possible, given additional failures, that the entire graph will not be retrieved even when the request is broadcast. Define the *failure period* of a participant to be the time period beginning at the time of the failure and ending at the point when the participant's state and view have been reconstructed. If the failure period of $n-1$ other participants overlap with the failure period of a recovering participant p, it can be guaranteed only that the portion of the graph from the root to the lowest n-stable messages will be available upon recovery.[4] To see this, consider such an n-stable message m_s. Since m_s is in the context of messages sent by $n-1$ participants in addition to the participant that sent m_s, at least n context graph images will contain all messages from the root to m_s. Given that only $n-1$ participants have overlapping failure periods, one of the images containing that portion of the graph is assured to be available. It is worth emphasizing that the above is a worst-case scenario; it is possible that messages below m_s in the context graph will be retrieved, depending on exactly which participants fail when.

As described so far, the recovering host depends entirely on the retransmission of messages from other hosts to reconstruct its image. In fact, each host is able to reduce its dependency on the other hosts by saving a copy of the messages in its image to non-volatile storage. Thus, a restarting host first directly recovers a portion of its image from non-volatile storage, and then "falls back" on the above procedure to recover the rest of the image. An appealing aspect of this scheme is that the changes to the image on non-volatile storage can be performed asynchronously. There is no requirement that the volatile and non-volatile images be updated atomically or even that the changes to the non-volatile copy keep pace with changes to the copy in volatile memory. Upon recovery, those portions not available in non-volatile storage can be retrieved from other images

[4]This discussion assumes a one-to-one correspondence between images and participants.

as described above. Moreover, because a given host might be asked to retransmit an early part of its context to a recovering host, it cannot free stable messages as described in Section 3.3. In other words, the garbage collection mechanism must be modified to write messages to non-volatile storage rather than free them.

On a related topic, the copy of the context graph on non-volatile storage need not be allowed to grow infinitely large. Two things can be done to limit the size of a context graph. First, Psync can easily be extended to allow participants to explicitly free portions of the graph. Second, the participants in a conversation can reach agreement to close the current conversation and start a new conversation.

Finally, consider the issue of which message identifiers should be saved for later use in the `restart` operation and the related question of checkpoint frequency. To a large degree, the answers depend on how much reexecution, if any, the application can tolerate. This results from the fact that messages in the context graph between the nodes saved in the checkpoint and the leaves at the time of the failure will be rereceived—and presumably reprocessed—following recovery. One conservative strategy would be to take a checkpoint following each message transmission. At this point, there is only one leaf in the view, minimizing the number of message identifiers that must be saved on non-volatile storage. Also, since any additional state transitions made by the participant prior to a failure cannot have had any external effect, it is usually straightforward for an application to reexecute that portion of the computation. A less conservative strategy is discussed in the next section.

5.2 Using Participant Reexecution

As noted above, a typical recovery scenario involves having the participant start executing at the most recent checkpoint, with messages being received again and reprocessed if they arrived after the checkpoint but prior to the failure. It is possible to carry this notion of reexecution to its logical conclusion by reexecuting the failed participant from its initial state, thereby avoiding the need to checkpoint altogether. If the same sequence of messages is used as input, this technique will, under certain assumptions, reestablish the same state and conversation view as existed when the failure occurred. As detailed below, Psync provides an attractive and automatic alternative for achieving the same functionality. Not only does the context graph encapsulate the entire communication history of the recovering process, but its realization as a collection of replicated images allows recovery of messages despite multiple host failures. We note in passing that similar functionality

has been implemented elsewhere by *logging* messages onto non-volatile storage as they are received [24], logging them at a monitor site [22], or by retaining copies of messages in the volatile storage of sending processes [15,16,25].

There are two conditions that must be satisfied to guarantee recreation of the appropriate state and view. One is that each participant in a conversation must be *deterministic*. For our purposes, this means that a process's state transitions and generated messages (i.e., its output) are determined solely by the sequence of messages it receives (i.e., its input). This assumption is satisfied by most applications.

The second condition is that sequence of messages received during reexecution of a participant must be exactly the same as the sequence received during its original execution. In other words, the same *total* ordering of messages must by presented to the application during the two executions. Since the context graph only directly preserves the appropriate *partial* ordering of messages, an application must impose an ordering filter on the conversation, e.g., the ordered broadcast filter described in Section 4.2. In general, any filter that preserves the total ordering at a given participant in subsequent executions is sufficient. The use of ordered broadcast is actually slightly stronger than necessary since it guarantees an identical total ordering at all participants.

The `restart` operation described above also serves as the mechanism to initiate message replay. This is achieved by specifying a null value for the `leaf_mid_set` argument to `restart`. When invoked in this manner, the local image of the context graph will be reconstructed exactly as described above, but the participant's view will be reinitialized to the empty graph. In other words, the participant will begin receiving messages again starting at the root of the graph.

Following completion of `restart`, the participant reestablishes its internal state and conversation view by simply executing normally. Messages sent by the application that are already in the context graph are suppressed at the sending host. This suppression is actually an optimization. If Psync assigns the same identifiers to messages during reexecution that it did during the initial execution, then the messages can be sent because they are automatically discarded as duplicates at the receiving host.

6 Performance

We have implemented Psync in the *x*-kernel: an operating system kernel designed to facilitate experimentation with network protocols [14]. The implementation corresponds to the protocol

described in Section 3; it does not currently support the reintegration of failed processes as described in Section 5. The implementation is both substantial and robust: it allows processes on the same host to share a conversation, it has supported conversations with tens of thousands of messages, and it has successfully recovered from significant rates of packet loss. By implementing Psync in the x-kernel, we have been able to evaluate it under conditions that match its intended role as a low-level IPC mechanism, and in particular, we have been able to make meaningful performance comparisons with other kernel-based protocols. This section reports on the performance of Psync and comments on several implementation details that affect its performance.

6.1 Experiments

The first set of experiments involve measuring the round trip delay for Psync, as well as three other IPC protocols: an unreliable datagram protocol (UDP), a remote procedure call protocol [1], and a virtual circuit protocol (TCP). Note that although Psync supports communication among more than two processes, experimenting with Psync in the one-to-one case is a good measure of the overhead it imposes on sending and receiving messages.

For the purpose of the experiments, the x-kernel was configured as follows: one-byte messages were exchanged between a pair of user processes, all four protocols were implemented on top of IP [20], and the tests were run on a pair of Sun 3/75s connected by a lightly loaded 10Mbs ethernet. The results are presented in Table 1. The numbers were derived by running each experiment for 10,000 round trips (20,000 total messages) and reporting the elapsed time every 1,000 round trips. Each of these measurements was then divided by 1,000 to produce an average round trip delay. Although we do not report the standard deviation of the various samples, they were observed to be small.

Protocol	Latency
UDP	2.9 ms
RPC	3.4 ms
Psync	4.0 ms
TCP	4.6 ms

Table 1: Comparing Psync with other Protocols

Psync's round trip delay of 4.0 msec is what one would expected: it falls between a trivial protocol (UDP) and a rather complex protocol (TCP). That Psync has lower latency than TCP is

encouraging. It means that Pysnc is a viable alternative protocol for one-to-one communication, especially considering that there is no overhead involved in starting a conversation. However, that Psync has a greater latency than RPC is disappointing. One (correctable) factor that we believe contributes to Psync's greater latency is that it incurs a moderate amount of overhead for allowing multiple processes on the same host to participate in a given conversation.

A second set of experiments measure Psync's performance with more than two participants. The experiments involve running an application program that passes a token among a set of processes that exeucte on different hosts. For comparative purposes, we implemented the same application program on top of TCP. In the TCP case, each process establishes a distinct virtual circuit to each of the other processes. Thus, each time an application process sends a message, it actually sends a copy of the message to all of the other participants using each of these circuits. To make the experiment fair, we configured Psync to use point-to-point message passing rather than take advantage of the ethernet's broadcast facility. That is, whenever an application process sends a Psync message, Psync in turn sends an IP datagram to each of the participating hosts.

The results are given in Table 2. The numbers were derived by allowing each application process to send and receive 20,000 messages, with each process reporting the elapsed time every 1,000 messages. Each of these measurements was then divided by 1,000 to produce the average delay per message. As in the first set of experiments, the variation in the elapsed times was observed to be small. Note that in the case of two participants, the token-passing application program is equivalent to the round trip program used in the first set of experiments. However, the times reported in Table 1 are twice those reported in Table 2. This is because the Table 1 times are based on 1,000 round trips (2,000 messages), while the Table 2 times are based on 1,000 messages.

Hosts	Psync	TCP
2	2.0 ms	2.3 ms
4	3.0 ms	3.6 ms
6	3.8 ms	4.8 ms
8	4.5 ms	7.0 ms

Table 2: Token Passing with Many Hosts

First, observe that Psync continues to perform well as more and more participants (hosts) are added to a conversation. Of particular importance is the fact that the incremental cost for each additional process is less for Psync than it is for TCP. This is the case even though Psync provides

a more powerful abstraction: it preserves the relationship among messages from all participants, whereas TCP provides no information about messages that arrive on different virtual circuits.

Second, observe that TCP's performance grows unexpectedly worse in the eight host case. This performance drop is a result of a measurable increase in the rate at which packets were lost. Specifically, because we were sending point-to-point messages, the load on the ethernet became substantial as additional hosts were added to the experiment. This heavy load, in turn, exposed a timing bug in the ethernet driver that caused packets to be dropped. Both Psync and TCP experienced negligible packet loss in the 2- and 4-host cases, 1 in 1000 messages were lost in the 6-host case, and 1 in 150 messages were lost in the 8-host case. TCP's performance suffers more from message loss than does Psync's because for every lost message, TCP has to wait for a timer to expire before it can request a retransmission, whereas Psync is able to request the retransmission as soon fas a message from another participant arrives that is in the context of the missing message.

6.2 Implementation Issues

The data structures and algorithms used to implement the context graph are tuned for the `send` and `receive` operations. Specifically, a hash table is used to map message identifiers (*mid*s) into the corresponding graph nodes, and a list of pointers to the leaf nodes of a view is maintained for each participant. This means that both `send` and `receive` can be implemented in linear time proportional to the number of participants in the conversation—i.e., the upper bound on the indegree/outdegree of each node—but independent of the size of the graph. Also, because Psync piggybacks the conversation establishment information on the first data message and no termination messages are exchanged, the cost to begin and end a conversation is negligible.

The current implementation can be configured to use either host-specific addresses or broadcast addresses. In the former case, a given Psync message is sent to each unique host on which a participant resides. In the latter case, a single Psync message is broadcast to all hosts. For the purpose of the experiments, host-specific addresses where used so as to facilitate a fair comparison with TCP. On a related note, while Psync's `active_open` operation is described as taking a set of participant ids as an argument, it could just as well take a single group id instead. To do so, however, the membership of the group must remain constant throughout the lifetime of the conversation and it must be possible to expand the group id into a set of individual process ids at each host. This is because Psync must be able to enumerate all the participants in the conversation in order to implement the `stable` operation.

Finally, because it is desirable to encapsulate what the application views as a logical message in a single Psync message, Psync uses an underlying blast protocol to send large messages. The interesting aspect of this blast mechanism is that it is encapsulated as a distinct protocol rather than embedded in Psync [1].

7 Related Work

Recent work on interprocess communication has explored several dimensions of the problem space, including support for group communication [8], the exchange of very large messages [9,29], alternative send/receive semantics [6], guaranteeing a consistent order on message delivery in a many-to-many communication [3,4], and techniques for logging messages so as to facilitate recovery from processor failure [15,16,22,24,25]. The work presented in this paper addresses the latter two issues.

Psync is most closely related to the ISIS protocol suite—ABCAST (atomic broadcast), CBCAST (causal broadcast), and GBCAST (group broadcast) [3,4]. From Psync's perspective, ABCAST and CBCAST are specific message ordering disciplines that can be implemented on top of the context graph: ABCAST supports a total ordering of messages similar to the ordered broadcast mechanism described in Section 4.2, and CBCAST supports the same partial ordering as Psync. In fact, Psync can be viewed as an optimistic implementation of CBCAST. This is because Psync only transmits the messages from the context of a given message when the context messages are missing at a given image. In contrast, the original implementation of CBCAST sent a sufficient set of predecessor messages (rather than just message ids) along with each message. This technique was further optimized so that unnecessary messages would not be piggybacked on a given message whenever the sending host had direct knowledge that *it* (as opposed to some other host) had already sent those messages. CBCAST is currently being re-implemented to more closely adhere to the protocol described in Section 3.

A more important difference is that CBCAST does not explicitly preserve the context graph and make it available to the application. Thus, it would not be possible to implement ABCAST on top of CBCAST in the same way one can implement an ordered broadcast on top of Psync. Also, instead of being able to use a single protocol (i.e., Psync) to implement the replicated object application outlined in Section 4.3, the application would have to use a combination of ABCAST and CBCAST.

Also, because ISIS is designed to be used directly by application programs, it provides fun-

tionality not directly available in Psync. For example, the ISIS protocols provide elaborate failure detection and group management support, whereas Psync off-loads much of this functionality to library protocols. In fact, much of the complexity of GBCAST is concerned with inserting process failure and group join events into ABCAST and CBCAST message ordering in a consistent way. In other words, ISIS is designed to subsume a large amount of functionality in a single package, whereas Psync is explicitly designed to provide only the necessary support for maintaining the ordering among messages; library protocols take advantage of this ordering to implement various levels of service.

In addition to ordering messages, the context graph very naturally lends itself to preserving the history of messages exchanged in a distributed application. Similar to message logging systems, Psync records the message history across multiple machines; i.e, each host's image preserves a portion of the context graph. It is also the case that the cost of logging messages in Psync does not impact the performance of the application when there are no failures. This is because messages can be written to non-volatile storage asynchronously; the non-volatile copy of the context graph need not be kept identical to the volatile copy. Psync differs from message logging systems in that it integrates the logging of messages with the preservation of a meaningful ordering among messages. That is, whereas logging systems generally augment an existing many-to-many communication protocol, logging in Psync is an automatic by-product of maintaining the context graph.

Finally, note that many of the ideas underlying Psync are founded in the space-time view of distributed computing. For example, the context relation can be viewed as a variation of the happened before relation [17]. As another example, when a message is stable, it is as if it has been *fully acknowledged* [26], that is, an acknowledgement message from all other participants has been received.

8 Conclusions

One of the most difficult issues facing designers of distributed systems is the level at which the timing and message ordering problem should be addressed: within the communication system or by the application. The underlying thesis of this paper is that the *mechanism* that preserves timing information should be implemented within the communication system but the *policy* that dictates how the timing information is used to enforce various synchronization constraints belongs in the application. That is, one of the contributions this paper makes is to distinguish between policy

and mechanism. In particular, it shows how the conversation abstraction can be provided in the communication system at little cost and how it can be used to implement various application-dependent communication and synchronization paradigms.

Psync is a low-level IPC protocol that implements the conversation abstraction in a distributed environment. Psync can be built directly on an unreliable communications network at little cost. This is because messages are sent asynchronously, extra protocol messages are only exchanged in the case of failure, constant-time algorithms are used for manipulating the context graph, and the amount of timing information sent with each message is insignificant. Experiments substantiate this claim: Psync's performance falls between that of a simple datagram protocol (UDP) and a virtual circuit protocol (TCP).

Our experience using conversations suggests that Psync, taken together with a collection of library routines, offers a simple and elegant solution to the communication needs of a broad spectrum of distributed applications. We believe this is due to the fundamental nature of the partial ordering of messages in interprocess communication. The context graph not only provides a powerful mental tool for thinking about other protocols, but also provides a sound programming base for implementing them. For example, distributed application do not have to pay for a total ordering of messages when a partial ordering is sufficient. As demonstrated by the replicated object example, being able to inspect the context graph allows the application chose the partial order when it is sufficient, yet synchronize by waiting for a message to stabilize when a total order is necessary. This information is not made available by other single mechanism.

Finally, because of the way Psync automatically distributes the history of a conversation over multiple hosts, it lends itself to building applications that are able to recover from processor failures. The storage demands of such a preserving this history over long periods of time are significant, however. For example, to support applications that do not need to recover from processor failures the current implementation stores only those messages that have not yet become stable. In contrast, an implementation that supports participant reintegration must store the entire context graph. Although the messages in the graph can be off-loaded to non-volatile storage, this still involves a significant cost. Our belief that the mechanism should be separated from the policy argues that the implementation should allow the application to specify to what extent the context graph should be preserved rather than having the storage policy mandated by the implementation. We will maintain this philosophy as we extend the implementation of Psync to support the participant reintegration.

Acknowledgements

Greg Andrews, Norm Hutchinson, and the referees made valuable comments on earlier drafts of this paper, leading to significant improvements in the presentation. Vic Thomas, Shivakant Mishra, David Bakken, and Peter Druschel have contributed to Psync's implementation.

References

1. M. Abbott, N. Hutchinson, S. O'Malley, and L. Peterson. *RPC in the x-kernel: evaluating design alternatives.* University of Arizona Technical Report TR 89-4 (Mar. 1989).

2. A. Aho, M. Garey, and J. Ullman. The transitive reduction of a directed graph. *SIAM J. Computing* (1972), 131-137.

3. K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Trans. on Computer Systems 5*, 1 (Feb. 1987), 47-76.

4. K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the 11th Symposium on Operating System Principles* (Nov. 1987), 123-138.

5. A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Trans. on Computer Systems 2*, 1 (Feb. 1984), 39-59.

6. N. Carriero and D. Gelernter. The S/Net's Linda kernel. *ACM Trans. on Computer Systems 4*, 2 (May 1986), 110-129.

7. J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Trans. on Computer Systems 2*, 3 (Aug 1984), 251-273.

8. D. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Trans. on Computer Systems 3*, 2 (May 1985), 77-107.

9. D. Cheriton. VMTP: A transport protocol for the next generation of communications systems. In *Proceedings of SIGCOMM '86* (Aug. 1987), 406-415.

10. F. Cristian. Agreeing on who is present and who is absent in synchronous distributed systems. *Digest of Papers, Fault Tolerant Computing Systems, 18*, IEEE Computer Society Press (June 1988), 206-211.

11. D. Gifford and N. Glasser. Remote pipes and procedures for efficient distributed communication. *ACM Trans. on Computer Systems 6*, 3 (Aug. 1988), 258-283.

12. J. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science 60*, Springer-Verlag, Berlin (1987), 393-481.

13. M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers,* Vol. C-36, No. 4 (Apr. 1987), 443-448.

14. N. Hutchinson and L. Peterson. Design of the x-kernel. In *Proceedings of SIGCOMM '88* (Aug. 1988), 65-75.

15. D. Johnson and W. Zwaenopoel. Sender-based message logging. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing,* (June 1987), 14-19.

16. D. Johnson and W. Zwaenopoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Proceedings of the 7th PODC* (Aug. 1988), to appear.

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

18. S. Mishra, L. Peterson, and R. Schlichting. *Implementing fault-tolerant replicated objects using Psync.* University of Arizona Technical Report TR 89-3 (Mar. 1989).

19. J. Postel. User datagram protocol. *Request For Comments 768*, USC Information Sciences Institute, Marina del Rey, Calif. (Aug 1980).

20. J. Postel. Internet protocol. *Request For Comments 791*, USC Information Sciences Institute, Marina del Rey, Calif. (Sept. 1981).

21. J. Postel. Simple mail transfer protocol. *Request for Comments 821*, USC Information Sciences Institute, Marina del Rey, Calif. (Aug. 1982).

22. M. Powell and D. Presotto. Publishing: a reliable broadcast communication mechanism. In *Proceedings of the 9th Symposium on Operating System Principles* (Oct. 1983), 100-109.

23. J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Trans. on Computer Systems 2*, 4 (Nov. 1984), 277-288.

24. R. Strom and S.Yemini. Optimistic recovery in distributed systems. *ACM Trans. on Computer Systems 3*, 3 (Aug. 1985), 204-226.

25. R. Strom, D. Bacon, and S.Yemini. Volatile logging in n-fault-tolerant distributed systems. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing* (June 1988), to appear.

26. F. Schneider. Synchronization in distributed programs. *ACM Trans. on Programming Languages and Systems 4*, 2 (April 1982), 125-148.

27. A. Tanenbaum. Computer networks, second edition. Prentice Hall, Englewook Cliffs, New Jersey (1988).

28. USC Information Sciences Institute. Transmission control protocol. *Request For Comments 793*, Marina del Rey, Calif. (Sept. 1981).

29. W. Zwaenepoel. Protocols for large data transfers over local networks. In *Proceedings of the Ninth Data Communications Symposium* (Aug. 1985) 22-32.