The Design and Implementation of Lansis/E

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

by
Ohad Rodeh

supervised by
Prof. Danny Dolev

Institute of Computer Science
The Hebrew University of Jerusalem
Jerusalem, Israel

May 25, 1997

Acknowledgments

I would like to thank my advisor Prof. Danny Dolev for creating the opportunity for this work, sending me to Cornell for the summer of 1996, and for supporting and believing in it all along.

This work would never have reached its state of completeness without the guidance and diligence of my father. Through long weekends of intermittent work in our Avocado plantation and proofreading of the paper he made this work readable, not only to myself, but to others.

Mark Hayden, the creator of the Ensemble system, helped with coding advice and support for changes I introduced to Ensemble. His efforts and openness to new ideas made Lansis/E work with all other parts of Ensemble.

Fruitful conversations with friends in the lab provided insight and new ideas. There are always new ideas to explore!

Finally, I'd like to thank my family: my grandparents Shimshon and Tova, my parents Bruria and Micky, and my brother and sister Alona and Yoav.

Abstract

Transis is a group communication system developed in the Hebrew University. It is a monolithic system written in the C programming language. Lansis is a subsystem of Transis containing its basic algorithms and data structures.

Ensemble is a novel group communication system developed in Cornell. It has a modern design stressing flexibility and modularity. The code is layered, written in a verifiable, machine independent, functional language — OCAML. The Ensemble infrastructure supports tracing, debugging and semi-automatic optimization.

This thesis presents the unification of Lansis and Ensemble in the form of the Lansis/E system.

We redesigned Lansis structure and implemented it in the form of a set of Ensemble protocols. Thus, we achieved simple, robust, flexible, extendable, portable code. Lansis was enhanced to support a stronger application programming model — Strong Virtual Synchrony. The user can now intermix Lansis and Ensemble protocols, further, new protocols may be added to the system using a framework we defined. Such protocols would benefit from full interoperability with existing Lansis/E and Ensemble protocols

Contents

1 Introduction

This work presents the unification of two group communication systems: Lansis ([ADKM92], [MADK94]), developed at the Hebrew university, and Ensemble [Hay96], developed at Cornell university.

A Group Communication system (GCS) introduces the *process group* abstraction. A process group coherently binds together many processes into one entity. Within the context of a group a process may reliably broadcast messages using known paradigms such as fifo, causal, or total (atomic). Processes may dynamically join and leave a group. Groups may dynamically partition into many components due to network failures/partitions. The GCS is responsible for hiding these "inconveniences" from the programmer, overcoming the asynchronous nature of the network and keeping the group abstraction consistent.

The Transis ([ADKM92], [MADK94]) system is a GCS developed in the Hebrew university. It is written in the C programming language and consists of several disjoint modules. Transis' core functions are encapsulated in the Lansis subsystem which is responsible for basic membership and message reliability services. Lansis assumes the existence of one group whose members all reside on the same LAN. Transis contains mechanisms which extend Lansis to efficiently handle multiple groups whose members are not in the same LAN.

The Horus [VRCGS92] system is a GCS developed in Cornell. Similar to Transis it is written in C. Its core functionality has been reimplemented in the Ensemble system. Ensemble, as Lansis, implements the basic membership and message reliability algorithms. Ensemble is written in OCAML [Ler97] — a modern version of ML [MTH90]. This makes a semi-automatic proof of the system closer to reality. The design methodology behind Ensemble draws from experience gained through many years of coding realistic GCS's (Isis [BVR94], Horus [VRCGS92]) stressing modularity and flexibility. Thus, Ensemble is divided into many *layers* each implementing a simple protocol; where layers communicate by message passing. By stacking together these layers, much like one uses lego blocks, the user may customize the system to suit his needs. For example, one may use only membership services and fifo ordered messages, and "pay" (via latency, memory consumption, and CPU usage) for these services alone.

In order to achieve greater flexibility and modularity facilitating easy extension of our system we decided to reimplement and extend the Lansis algorithms in the Ensemble framework. We implemented the Lansis protocols, and extended the Ensemble infrastructure to support new protocols. Now the user can intermix Lansis and Ensemble protocols.

The rest of this thesis is organized as follows: Chapter 2 briefly describes related work, Chapter 3 describes the system and programming model. Chapter 4 describes the first stage in the redesign of the Lansis subsystem. In that stage the Lansis monolithic system is split into several disjoint layers with well defined interfaces and it is enhanced to support a stronger programming model (*Strong Virtual Synchrony*). The new system we call Lansis/E. Chapter 5 describes Ensemble. Chapter 6 describes the second stage in the redesign of Lansis and the implementation in Ensemble. We introduced message passing as a communication paradigm between Lansis/E layers. We also translated the Lansis/E symmetric membership and virtual synchrony algorithms into asymmetric algorithms. Chapter 7 describes modifications made to Ensemble, Chapter 8 describes the interoperability of Lansis/E protocols with Ensemble protocols including new protocol stacks now possible. Chapter 9 concludes.

2 Related Work

Group communication systems (GCSs) are powerful tools for the development of fault-tolerant distributed applications. GCSs provide the application builder with reliable multicast services; they support message ordering paradigms such as fifo, causal, and total (atomic) order. GCSs also provide group membership services that guarantee strong semantics which is useful for the construction of fault tolerant applications.

The basic abstraction GCSs introduce is that of a *process group*. A process group coherently binds together many processes into one entity. Processes may dynamically join and leave a process group; a group may dynamically partition into many components due to network failures/partitions. The GCS is responsible for hiding these inconveniences from the programmer thereby facilitating simple distributed programming. GCSs are designed to achieve high throughput and low latency while tolerating extreme network conditions (process crashes or network partitions). Ongoing work is conducted on the scalability issues. As networks grow support is needed for geographically disperse process groups. Support is also needed for large process groups comprised of more than two dozen processes.

Today there are quite a few full scale applications written over the Isis commercial GCS [BJ87]. This includes air traffic control, process management, network simulation, file management, replicated databases, and more.

In the Totem system [AMMS$^+$95] a token is passed through the members of a process group. Only the token holder may broadcast messages; other members are restricted to receiving messages and must buffer outgoing messages until they receive the token. This scheme facilitates efficient usage of the Ethernet broadcast media while preventing collisions. Furthermore, allowing only a single member to broadcast at each time instance makes it easy to support total order efficiently, to detect stability, and to prevent fragmentation.

The RMP [CM84] system uses a closely related idea. It uses a *postordering token ring*. Processes may broadcast at any time. Received messages are buffered until the leader's *ordering message* is received. The current holder of the token (the leader) broadcasts an *ordering message* every time interval, specifying the order in which buffered messages are to be delivered to the application. All members order messages according to the leader's instructions.

The Phoenix project [MFSW95] is geared towards large scale systems. Phoenix identifies the basic predicate between sites to be *reachability*. Site

A can reach Site B if and only if B receives A's messages. Is this relation transitive? If A reaches B and B reaches C is it true that A reaches C? In large systems this may not be the case. Examine three Phoenix sites Ch, Sw, and It, in Chile, Switzerland, and Italy respectively. It is quite conceivable that Sw can reach It and It can reach Ch but Sw cannot reach Ch. This can happen because the low level routing mechanism fails to provide a path $Sw \rightarrow Ch$. The system overcomes this dynamically via rerouting Sw's messages to Ch through It.

By making reachability a transitive relation the network can be cleanly split into disjoint components within which all sites can communicate. This mechanism coupled with the reachability relation forms the basis of the Phoenix membership and group services.

Newtop [MES] developed at the University of Newcastle supports inter-group semantics and causal/total ordering via an extension of Lamport's time stamping technique [Lam78].

The Relacs system [BDGB94] also supports the process group abstraction, though at the semantic level virtual synchrony [BJ87] is compromised in order to facilitate group communication over large scale systems.

Amoeba [KTHB89] supports the process group abstraction at the operating system level. This provides better performance for applications that make heavy use of group communication. Most applications do not rely on low level reliable communication or group abstractions. Furthermore, if specific messages must be sent reliably, the application implements reliable communication for these messages. The performance of these applications degrades in Amoeba since they are encumbered by the heavy weight algorithms in the kernel that implement reliable communication and process groups.

3 Model

3.1 The GCS Programming Model

In order to provide the application with reliable group oriented communication, the GCS acts as a transport layer. This means that it separates the application from the network through a software module (see Figure 1). This software module uses the network send/deliver (denoted by N_send/N_deliver) capabilities while providing the application with an interface of strong semantics. The new interface follows the Strong Virtual Synchrony model [FVR95] and is denoted by SVS_send/SVS_deliver. This separation allows the GCS to buffer messages, delay their delivery until correct ordering is achieved, retransmit messages lost by the network, and more. It allows hiding from the application the asynchronous lossy nature of the network.

3.2 The System Model

Consider a universe which consists of a finite group $\mathcal{P}$ of n processes. Processes communicate with each other by passing messages through a network of channels. There is no global clock and messages may be arbitrarily delayed. Thus, our universe is asynchronous.

We assume that messages may get lost although the network does not corrupt messages.

We will be interested in the *broadcast* primitive. We say that a process p broadcasts a message m if it sends m to all the processes in $\mathcal{P}$.

Processes may *get partitioned* from each other. A partition occurs when $\mathcal{P}$ is split into a set $\{P_1, \ldots, P_k\}$ of disjoint subgroups. Each process in P_i can communicate only with processes in P_i. The subgroups P_i are sometimes called *components*. We shall consider *dynamic partitions*, when components may dynamically merge and split.

In the sequel p, q, s, t, u, and v denote processes and $m, m_1, m_2, \ldots$ denote messages they send to each other. An *event* occurs when a process sends a message or receives a message. For simplicity we assume that processes receive their own messages (eventually).

Each message has an origin and a sequence number. All the messages that a process sends are numbered. A message *id* is its origin to which its sequence number is concatenated. We denote:

$m_{p,seq} \stackrel{\text{def}}{=}$ the message m that process p has sent whose sequence number

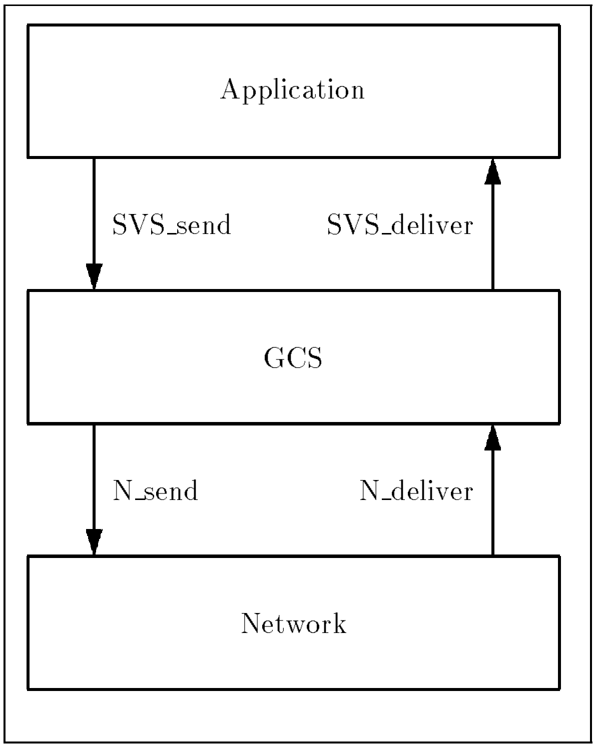

Figure 1: The GCS Programming Model. The GCS uses the network N_send/N_deliver interface while providing the application an SVS_send/SVS_deliver interface.

is *seq*.

$m_p \stackrel{\text{def}}{=}$ abbreviation, some message m from process p.

$\text{N\_send}_p(m) \stackrel{\text{def}}{=}$ p broadcasts the message m through the network.

$\text{N\_deliver}_p(m) \stackrel{\text{def}}{=}$ the network delivers message m to process p.

In general, since the system is asynchronous, the relative order between events occuring at different processes may not be uniquely determined. To overcome this difficulty, we first define an order among the events occuring at a single process p:

$e_1 < e_2 \stackrel{\text{def}}{=} e_1$ and e_2 are events occuring at the same process, and e_1 occurs before e_2.

This allows us to define a *fifo order* between messages:

$m_1 < m_2 \stackrel{\text{def}}{=} \exists p \mid \text{N\_send}_p(m_1) < \text{N\_send}_p(m_2)$

In asynchronous environments messages may not arrive in fifo order. Examine Figure 2. In the figure and henceforth time flows from top to bottom. In the scenario process p executes $\text{N\_send}_p(m_1)$ followed by $\text{N\_send}_p(m_2)$, while process q executes $\text{N\_deliver}_q(m_2)$ followed by $\text{N\_deliver}_q(m_1)$.

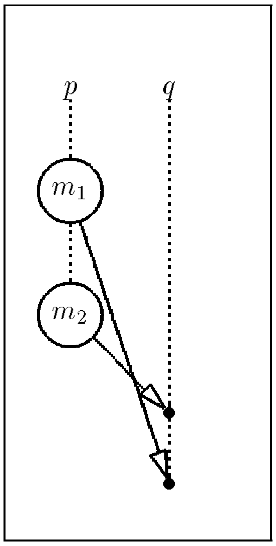

Figure 2: Process p broadcasts message m_1 and m_2. Process q receives m_2 then m_1. Fifo order is not preserved. The arrows stand for message sending.

When more than two parties communicate, "inconvenient" scenarios may arise, even though fifo order is preserved. Since we are interested in providing the application with a simple programming model, we would like to eliminate such scenarios. An example is given in Figure 3. In this example process p broadcasts a question m_1. Process q receives m_1 replying with m_2. Process s receives m_2 before m_1, thus it receives the answer before the question.

In order to guard against such scenarios we define the notion of *causal order* (denoted by $\leftarrow$) among messages such that if m_2 is the response to message m_1 then $m_1 \leftarrow m_2$. More generally, we wish to capture the *happened-before* relation defined by Lamport [Lam78], and have $m_1 \leftarrow m_2$ if m_1 causes m_2. To this end we first define:

m_2 follows $m_1 \stackrel{\text{def}}{=}$

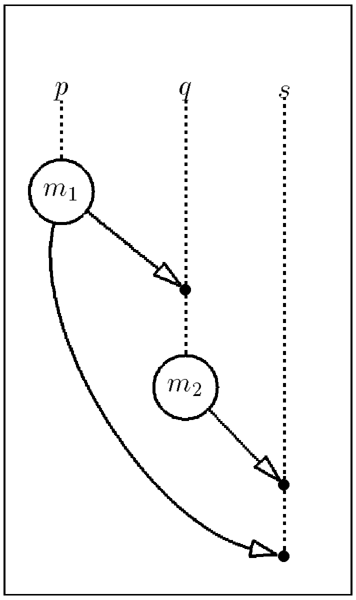

Figure 3: Process p broadcasts m_1. Process q receives m_1 and broadcast m_2. Process s receives m_2 then m_1. Causal order is not preserved.

$$\exists p \mid \text{N\_deliver}_p(m_1) < \text{N\_send}_p(m_2) \ \bigvee \ \text{N\_send}_p(m_1) < \text{N\_send}_p(m_2)$$

We then take causal order to be the transitive closure of the follows relation, and denote it by $\leftarrow$.

Causal order is a partial order between messages. Thus, it is possible for two messages m_1 and m_2 to be incomparable; neither $m_1 \leftarrow m_2$ nor $m_2 \leftarrow m_1$. In this case we shall say the m_1 and m_2 are concurrent and denote this by $m_1 \parallel m_2$.

If $|\mathcal{P}| = 2$ then fifo order is equivalent to causal order. If the number of processes exceeds 2 then the causal order is stronger. For example, in Figure 3 despite the fact that fifo order is preserved causal order does not; $m_1 \leftarrow m_2$ but process s receives m_2 and then m_1. We view the causal order as a natural extension of the fifo order to the case where more than two parties communicate (for an extensive discussion see [BVR94]). Thus, our GCS supports it.

3.3 Membership

Following the Relacs and Phoenix projects we define *reachability* between processes. We say that process q is *reachable* from process p if and only if p's messages eventually reach q (we denote this by $p \rightsquigarrow q$). Reachability changes with time and network conditions. Typically, a *failure detector* [CT91] is used by each process to decide which processes are reachable at a given time instance.

A failure detector usually works via timeouts and "I'm alive" messages. Denote the failure detector at process p by FD_p. FD_p broadcasts each timeout an "ImAlive" message. A process q from which FD_p does not receive ImAlive messages for more than a predetermined duration of time is determined to be temporarily unreachable. FD_p cannot tell if q has crashed, is slow, or cannot be reached because of a routing or link failure. Furthermore, the impossibility of creating a perfect failure detector has been shown [CHTCB96]. If FD_p receives ImAlive messages from process q, previously determined to be unreachable, then q is determined to be reachable again.

Since different processes use different timeouts and since network conditions change over time, reachability may be hard to compute and even harder to agree upon. As we see in Figure 4 reachability is neither symmetric ($s \rightsquigarrow q$ and $q \not\rightsquigarrow s$), nor transitive ($p \rightsquigarrow s$ and $s \rightsquigarrow q$ but $p \not\rightsquigarrow q$). We assume that the network satisfies:

- *Eventual symmetry*: if $p \rightsquigarrow q$ always holds then eventually $q \rightsquigarrow p$
- *Eventual transitivity*: if $p \rightsquigarrow q$ and $q \rightsquigarrow s$ always hold then eventually $p \rightsquigarrow s$

Thus, we extend reachability to its transitive closure, denoted $\overset{*}{\rightsquigarrow}$. We say that p and q are *E-reachable* if and only if $p \overset{*}{\rightsquigarrow} q$ and $q \overset{*}{\rightsquigarrow} p$. E-reachability is an equivalence relation on $\mathcal{P}$. As such it partitions $\mathcal{P}$ into disjoint components $P_1, \ldots, P_k$. In Figure 4 processes p, q, and s form one component.

If one had a global view of the network situation one could use this global knowledge to compute the "real" partitioning of the network. Since each process only has partial knowledge, and due to message delays and instability of the reachability relation, each process has its own *view* of the E-reachability relation. The view of a process is thus an approximation of the "real" network situation.

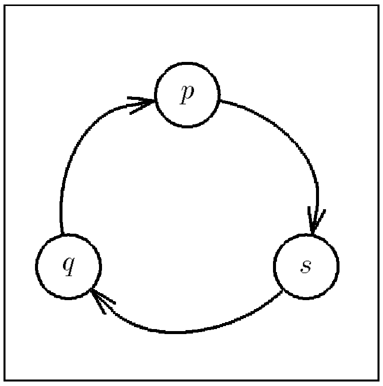

Figure 4: Processes p, q and s are pairwise E-reachable.

If no link or crash failures occur for a sufficiently long period of time (stability period) then the real network component can be computed by every process.

The *membership problem* at process p is that of identifying the network component P_i such that $p \in P_i$. P_i is also termed process p's *view*, denoted by V_p. The Lansis and Ensemble membership algorithms satisfy:

1. *True agreement:* If the network is stable for a sufficiently long period of time, then eventually for all processes p in P_i, $V_p = P_i$.

2. *Termination:* Termination within finite time.

Note that if the network is unstable then p's membership algorithm will provide only an approximation of P_i. Thus, accuracy is required only at periods of stability.

Theoretically, it is possible to design a membership algorithm that always decides $V_p = \{p\}$. This *trivial solution* does not satisfy true agreement but always terminates within finite time. The weakness of this algorithm is that it does not reflect the real network situation by not providing good approximations of network components. To exclude such algorithms we formally require true agreement.

We can drop the termination requirement and get a better approximation of the component. However, in most GCSs processes are blocked during network instability periods. Thus, if the membership algorithm does not guarantee finite termination in all scenarios then applications may be blocked for unbounded periods of time during network instability. We believe this to be an undesirable property, thus we require termination from our membership algorithms.

Viewing the period of time in which the application is blocked as overhead, there is a tradeoff between accuracy and overhead. If accuracy is essential we cannot insist on termination. Otherwise, algorithms that always terminate but provide approximations of lesser quality may be used.

4 Lansis/E — an Enhanced Layered Lansis

Embedding Lansis in Ensemble we had to fit Lansis into the Ensemble layered structure. This included restructuring Lansis into layers, distribution of shared data structures, and translation of symmetric algorithms to asymmetric algorithms.

We also enhanced the virtual synchrony specification supported by Lansis from *extended virtual synchrony* [MAMSA94] (EVS) to *strong virtual synchrony* [FVR95] (SVS). This provides the application with a stronger programming model.

In this section we describe the restructuring of Lansis into the five Lansis/E layers. Each layer performs a distinct algorithm and provides a specific interface with guaranteed semantics (such as fifo order, causal order, etc.). This allows better debugging since each module may be checked individually against its specification. It also allows better modularity since the system may function without all the layers in place. New layers may be inserted so long as they respect existing inter module interfaces and semantics.

This section also describes the basic concepts of group communication, message ordering and virtual synchrony which we use throughout this work.

4.1 Lansis

The Lansis package contains the core algorithms and functionality of Transis in a LAN setting. The Transis session service, supporting lightweight groups [BSS91], is built on top of Lansis. Above the Transis session service higher level services and applications are constructed.

Lansis is a multicast communication layer implemented as a library connected through a socket (or a memory buffer) interface to the user application. Lansis processes create a group in which process failures and message loss are transparently handled. Three message ordering paradigms are supported: causal, agreed, and safe order. The current view of the processes is maintained and view changes are delivered in a consistent manner with application messages. In this Lansis follows the *extended virtual synchrony* [MAMSA94] model (EVS). Lansis interacts with the network through an N_send/N_deliver interface providing the client with an EVS_send/EVS_deliver interface.

In order to save copy operations all messages are kept in a global buffer to which Lansis modules refer through pointers. All Lansis modules can access this buffer. This design allows copying a message at reception while performing all further operations without move instructions. At the end the message is copied to the user memory space. For Lansis modules a "message" is actually a pointer. Thus, "copying" and "moving" messages are, in fact, copying and moving pointers. Garbage collection is performed in the background in order to avoid buffer overflow.

In Lansis the application can send three types of messages: Causal, Agree, and Safe. Thus, the EVS_send primitive accepts two parameters — the message payload and the message type. Causal messages are reliably sent, and delivered in causal order. Agree messages are Causal messages that are also totally ordered. Safe messages are Agree messages that are delivered only when all members of the current view acknowledged them.

The global buffer maintains the message content along with its type so that all modules can access this information. From here on we denote Causal type messages by cm, Agree type messages by am, and Safe type messages by sm.

Lansis totally orders process names using lexicographic order. Thus processes in $\mathcal{P} = \{p_0, p_1, \ldots\}$ have ranks — 0,1,2, ...

4.2 Lansis/E Design

In Figure 5 the general structure of Lansis/E is depicted. This is a refinement of Figure 1 presenting the inner structure of our system. Lansis/E provides an SVS_send/SVS_deliver interface, supporting strong semantics. It uses a failure detector mechanism, denoted in the figure as FD, whose description is outside the scope of this work. In the following we shall describe the Lansis/E algorithms as we abstracted and changed them from the Lansis algorithms:

1. Causal Delivery
2. Safe Delivery
3. Agreed Delivery
4. Membership
5. Strong Virtual Synchrony.

Each Lansis/E layer uses the interface it gets from the layer below it and provides a new interface to the layer above it. Message ordering guarantees are provided using a simple buffering strategy. A message is buffered in

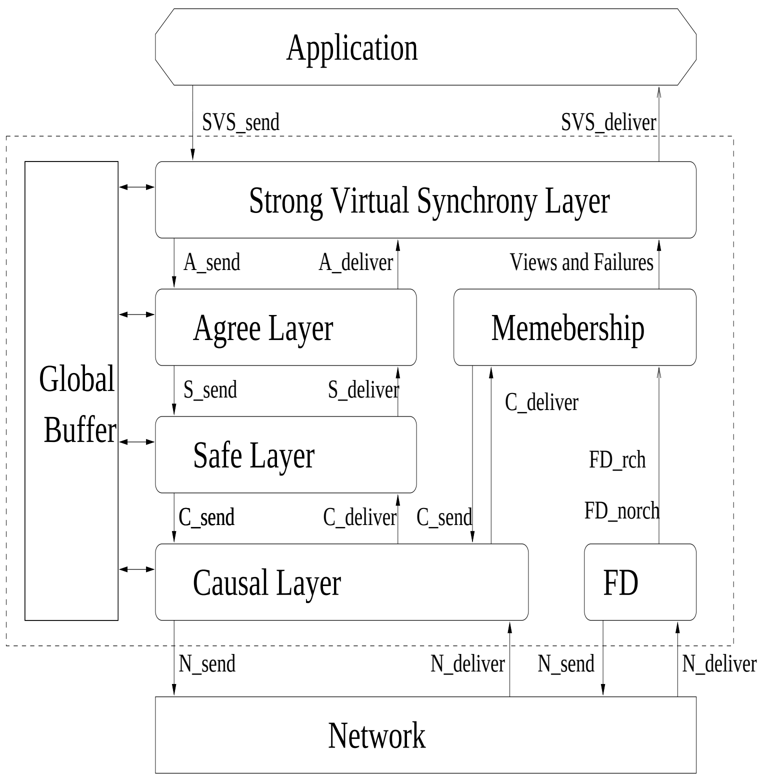

Figure 5: Lansis/E layered design

layer x until delivery conditions for it are met. Thus, each layer guarantees a certain delivery condition. The Causal layer delays messages until they can be delivered in causal order. The Safe layer delays Safe messages until all processes acknowledged receipt. The Agree layer delays Agree messages until a total order on them is determined. Safe messages are processed by all layers – the Causal layer guaranties causality, the Safe layer guaranties acknowledgements from all processes in the view, and the Agree layer guaranties a total order on them.

The Agree and Safe layers do not perform any processing upon message sending; they simply forward the message to the layer below. Thus an application message is sent through the Causal layer and delivered through the Causal/Safe/Agree layers.

We have redesigned the safe delivery and agree delivery algorithms. In Lansis there was one algorithm that provided both functionalities (See appendix 10.1 for the original Lansis algorithm). We also redesigned the Lansis membership and virtual synchrony algorithms. The original algorithm performs both tasks concurrently using a symmetric algorithm. This has been split into two algorithms which are now asymmetric, and can easily be extended to have a symmetric version.

Describing ordering algorithms we handle only message omissions. Network partitioning and process crashes are addressed when we discuss membership and virtual synchrony (Sections 4.6 and 4.7 respectively). Discussing network partitions and process failures, we say that *failed* members are group members that have partitioned from the view while *live* members are those members that stay live and connected.

In order for all algorithms described below to work properly, all processes must send messages on a regular basis (say, at least once every second). If a process does not adhere to this requirement Lansis remedies this by sending null messages on its behalf. Thus we assume that all processes are sending messages at all times.

This mechanism along with the failure detector employed by Lansis/E are the only time driven mechanisms in the system. All other modules are completely asynchronous and message driven.

4.3 Causal Layer

While a process p executes the causal ordering algorithm it interacts with the network through the $\text{N\_send}_p(m)$/ $\text{N\_deliver}_p(m)$ interface providing an interface $\text{C\_send}_p(m)$/ $\text{C\_deliver}_p(m)$. This algorithm guarantees causally ordered message delivery.

The purpose of this algorithm is threefold:

1. *Atomic messaging:* if p sends a message m then either all processes get it or no one does.
2. *Causal delivery:* if $m_1 \leftarrow m_2$ then m_2 is delivered after m_1.
3. *Self delivery:* if p sends m then p receives m.

To simplify the causal ordering algorithm we redefine the causal order ($\leftarrow$). First we redefine *follows*:

$$m_2 \text{ follows } m_1 \stackrel{\text{def}}{=} \exists q \,|\, (\text{C\_send}_q(m_1) < \text{C\_send}_q(m_2) \ \bigvee \ \text{C\_deliver}_q(m_1) < \text{C\_send}_q(m_2))$$

We now take causal order to be the transitive closure of the new follows relation. The new definition restricts causality between messages. By the old definition if a process q executes N_deliver$_q(m_1)$ and then N_send$_q(m_2)$ then $m_1 \leftarrow m_2$. By the new definition $m_1 \leftarrow m_2$ only if q executes C_deliver$_q(m_1)$ and then C_send$_q(m_2)$. This allows the causal algorithm more flexibility since it may, in some cases, delay message delivery. This ability is used to delay delivery of out of order messages until missing messages are recovered and correct order is achieved.

The global buffer, in which messages are stored, is molded into a directed acyclic graph structure termed dag_p at each process p. Onto each message m an id list of messages that it causally follows is attached. This list is called m's *acknowledgment list*. It is stored in the global buffer along with the message m such that all algorithms can access it. The nodes of dag_p represent messages and the edges are explicit acknowledgments.

In the dag the *leaves* and the *roots* play a special role. The leafs are those messages that are the "oldest" in the dag. That is, they acknowledge no message. The roots are those messages that are the "newest" in the dag. That is, no message causally follows them.

The correctness of the algorithm hinges on the following observation: If the sender p attaches to each sent message m an acknowledgment list containing all dag_p roots then, in effect, it has acknowledged all messages in dag_p. The receiver q, upon receiving m, will deliver all messages in m's acknowledgment list before delivering m. This is done in recursive fashion, thus q must deliver all of dag_p before delivering m. Since dag_p contains exactly all causal predecessors of m then q is ensured of causal delivery.

In order to overcome message omission, for each message m that q receives from the network, q checks its acknowledgment list. If m causally follows missing messages then *negative acknowledgments* are sent in order to recover them. When all the predecessors of m have been delivered then m is delivered. Prior to that m is kept in a *pending queue*. The pending queue holds all messages missing some causal predecessor; dag_q contains all messages for which all predecessors have been received.

Define C_send$_p(m)$ to be:

1. Insert m into dag_p.

2. Deliver m (implementing self delivery).

3. Attach the list of all dag_p root id's onto m.

4. N_send$_p(m)$.

Define C_deliver$_p(m)$ to be:

1. Check m's acknowledgment list. Deliver all preceding messages before delivering m.
2. If any messages are detected missing then N_send$_p$ *negative acknowledgments*. Place m_q in a pending buffer (local to the Causal algorithm).
3. Check if any messages in the pending buffer became deliverable. If so — insert them into dag_p.

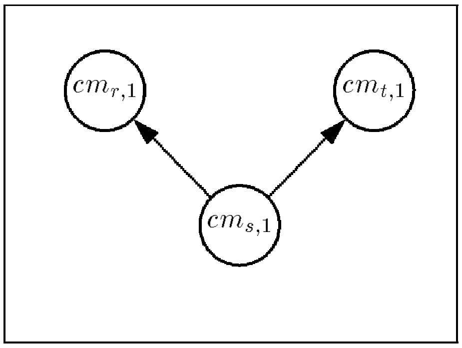

Figure 6: $\mathcal{P} = \{rst\}$. dag_s — $cm_{s,1}$ causally follows $cm_{r,1}$ and $cm_{t,1}$. Message $cm_{s,1}$'s acknowledgment list includes $cm_{r,1}$ and $cm_{t,1}$. The arrows stand for acknowledgments. The nodes stand for messages.

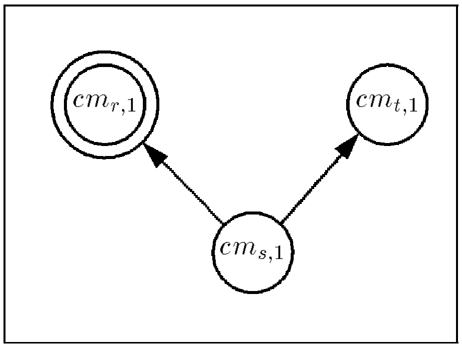

Figure 7: $\mathcal{P} = \{rst\}$. dag_t — $cm_{s,1}$ causally follows $cm_{r,1}$ and $cm_{t,1}$ but $cm_{r,1}$ is missing. The double node stands for a missing message.

In Figure 6 the universe $\mathcal{P}$ includes processes r, s, and t; $cm_{r,1}, cm_{s,1}$, and $cm_{t,1}$ are messages they send to each other. In the figure, and from here on, the nodes stand for messages and arrows stand for causal dependencies. Whenever a message m_2 points at message m_1 then m_2 causally follows message m_1. Processes are linearly ordered from left to right $r < s < t$. Process s receives messages $cm_{r,1}$ and $cm_{t,1}$ and sends $cm_{s,1}$ causally following both. Process t on the other hand (Figure 7) receives $cm_{s,1}$ and $cm_{t,1}$. It detects via the acknowledgment list attached to $cm_{s,1}$ that message $cm_{r,1}$ is missing. Process t asks for a retransmission of $cm_{r,1}$ by executing N_send$_p$($nack(r, 1)$). Any process that has received $cm_{r,1}$ and $nack(r, 1)$ will retransmit $cm_{r,1}$ to t. Message $cm_{s,1}$ is kept in t's pending queue until the receipt of $cm_{r,1}$.

A message m_q in dag_p in not erased from dag_p immediately after delivery. A copy is kept for retransmission requests. Since p's buffers are not unlimited it must eventually free m_q. Once all processes in $\mathcal{P}$ acknowledge m_q, p can erase m_q. Processes backup each other. Any process can retransmit on request any message. If r sends m, s receives it, and t asks for it, s can retransmit m in r's place. Lansis uses a probabilistic answering mechanism to restrict answers to retransmission requests. Thus, when a member receives a request it answers only with some probability.

If a process fails its messages may never enter the dag of other processes. For example, assume that process p executes C_send$_p(m_1)$ followed by C_send$_p(m_2)$ and then fails. Also assume that no other process receives m_1. Process q receives m_2 and requests m_1. Since no live and connected process can retransmit m_1 then message m_2 will always be pending at q and will never enter dag_q. This issue is addressed in Section 4.7.

4.4 Safe Layer

We define a message m to be *safe* at process p if and only if

> There are acknowledgments in dag_p for m from all $q \in \mathcal{P}$.

The safe delivery algorithm uses the C_send$_p$/C_deliver$_p$ interface and offers an S_send$_p$/S_deliver$_p$ interface. No processing is performed in the S_send$_p$ function, thus, it simply forwards the message to the layer below. Messages are delayed in delivery until proper conditions are met, thus the functionality of this layer is encapsulated in the S_deliver$_p$ function.

The layer is event driven by C_deliver$_p$ events, thus it handles only the part of dag_p delivered to it by the Causal layer. The Causal layer delivers messages in correct causal order. Therefore, if a message m is delivered to the Safe layer then all its predecessors have been delivered previously. Furthermore, by the specification of C_deliver$_p$ all messages in dag_p are eventually delivered.

Let m_q be the oldest message from process q in dag_p. If m_q is a leaf of dag_p and it is of type Safe then we define an *S-cut* for it. An *S-cut* for a message m_q in dag_p is a set of messages $\mathcal{S}_{m_q}$, such that:

1. All messages in $\mathcal{S}_{m_q} \setminus \{m_q\}$ causally follow m_q.
2. There is exactly one message from each $t \in \mathcal{P}$ in $\mathcal{S}_{m_q}$.
3. Each message $m_t \in \mathcal{S}_{m_q}$ is the oldest message from t in dag_p

Notice that the number of messages in $\mathcal{S}_{m_q}$ is equal to the number of processes in $\mathcal{P}$. Also notice that m is a member of $\mathcal{S}_{m_q}$.

Define S_deliver$_p$:

- A Safe message sm is delivered when an S-cut containing it is found in dag_p.
- A Causal / Agree message m is delivered as soon as all its predecessors have been delivered.

Note that once a Causal / Agree message m causally follows some Safe message sm, it is delayed until sm is delivered.

In Figure 8 process s has received all messages except $cm_{t,1}$. Thus, it cannot execute S_deliver$_s(sm_{s,1})$. When $cm_{t,1}$ arrives a complete S-cut is formed $\mathcal{S}_{sm_{s,1}} = \{cm_{r,1}, sm_{s,1}, cm_{t,1}\}$ and $sm_{s,1}$ is delivered. In this figure and from here on, dotted messages form a cut in the dag.

When a process fails and is therefore unable to provide acknowledgments for some messages, the calculation of S-cuts cannot complete. For example, if t fails and does not send $cm_{t,1}$ process s would be blocked since the S-cut containing $sm_{s,1}$ cannot be formed. This issue is addressed in Section 4.7.

If no process fails then this layer is non-blocking. Thus, any message received by the Safe layer is eventually delivered.

Safe messages are also totally ordered such that any two concurrent Safe messages will be delivered at all processes in the same order. This is performed by the Agree layer.

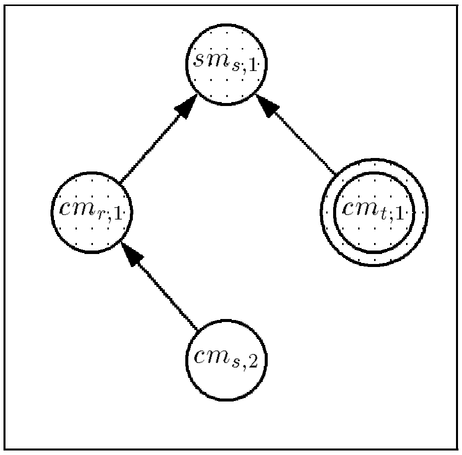

Figure 8: dag_s — s sent $sm_{s,1}$ and received $cm_{r,1}$. It sent $cm_{s,2}$ causally following $cm_{r,1}$. It is waiting for $cm_{t,1}$. Dotted nodes form a cut in the dag.

4.5 Agree Layer

This algorithm totally orders messages of Agree type. The algorithm interacts with the safe algorithm using the S_deliver$_p$/S_send$_p$ interface providing an interface A_deliver$_p$/A_send$_p$. It is event driven by S_deliver$_p$ events. The A_send$_p$ function simply forwards messages to the layer below. The functionality of this layer is encapsulated in the A_deliver$_p$ function.

We assume that no process fails, thus the Safe layer does not block. This implies that all messages in dag_p are eventually delivered to the Agree algorithm.

We say that a message m_1 is *globally agreed* if and only if all processes $q \in \mathcal{P}$ receiving m_1 and some other message (of arbitrary type) m_2 satisfy:

$$\begin{array}{ll} \forall q \in \mathcal{P} & \text{A\_deliver}_q(m_1) < \text{A\_deliver}_q(m_2) \\ or \quad \forall q \in \mathcal{P} & \text{A\_deliver}_q(m_1) > \text{A\_deliver}_q(m_2) \end{array}$$

The purpose of the Agreed algorithm is that of achieving globally agreed ordering for all Agree messages. The Agree algorithm waits until dag_p forms an *A-cut*. An A-cut is comprised of two sets of messages ($\mathcal{L}$,$\mathcal{A}$) where $\mathcal{L}$ is the set of leaves of dag_p and $\mathcal{A}$ is the set of oldest messages in dag_p that are not leafs. $\mathcal{A}$ contains the oldest message in dag_p from each process, provided it is not a leaf. An A-cut must satisfy:

1. All messages in $\mathcal{L}$ are of type Agree.

2. $|\mathcal{A} \cup \mathcal{L}|=|\mathcal{P}|$

3. Each message $m \in \mathcal{A}$ causally follows some message in $\mathcal{L}$.

Since all messages in $\mathcal{A}$ must causally follow some message in $\mathcal{L}$ we say the $\mathcal{A}$ is an *set of acknowledgments* for $\mathcal{L}$. An A-cut appears only when specific conditions hold; only then can A_deliver$_p$ be executed. A_deliver$_p$ is defined as follows:

- Let ($\mathcal{L}$,$\mathcal{A}$) be an A-cut. Deliver all messages in $\mathcal{L}$ according to process ranks (deliver a message from process p_0 before a message from process p_1, etc.).
- Causal messages are delivered as soon as all their predecessors have been delivered.

Note that once a message m causally follows some Agree message am it is delayed until am is delivered.

In Figure 9 dag_r is shown. Processes r, s, t, and v send Agree messages: $\{am_{s,1}, am_{t,1}\}$ and Causal messages $\{cm_{r,1}, cm_{s,2}, cm_{v,1}\}$. Process r has received all messages except $cm_{v,1}$. Thus, it cannot deliver any of the messages in dag_r. When $cm_{v,1}$ arrives an A-cut is formed in dag_r. The set of leaves $\mathcal{L}$ is $\{am_{s,1}, am_{t,1}\}$. Both messages in $\mathcal{L}$ are of type Agree. The set of acknowledgments $\mathcal{A}$ includes $\{cm_{r,1}, cm_{v,1}\}$ therefore $|\mathcal{A} \cup \mathcal{L}|=|\mathcal{P}|$. Consequently, A_deliver$_p$ is executed and messages $am_{s,1}$ and $am_{t,1}$ are delivered. Messages $cm_{r,1}$ and $cm_{v,1}$ become deliverable since no Agree message is blocking them.

When a process fails the algorithm blocks since A-cuts can no longer be found in the dag. Consider Figure 9 once again. If v fails before sending $cm_{v,1}$ then process r would be blocked. The A-cut containing $am_{s,1}$ and $am_{t,1}$ cannot be formed. We address this issue in Section 4.7.

If no process fails then this layer is non-blocking. Thus, any message received by the Agree layer is eventually delivered.

4.6 The Membership Algorithm

Many membership algorithms are known. In Lansis a symmetric algorithm is used. In Ensemble an asymmetric (leader-based) algorithm is used. In Totem [AMMS+95] a token based algorithm is used. Any membership algorithm must provide view and failure notifications to the system. In Lansis/E

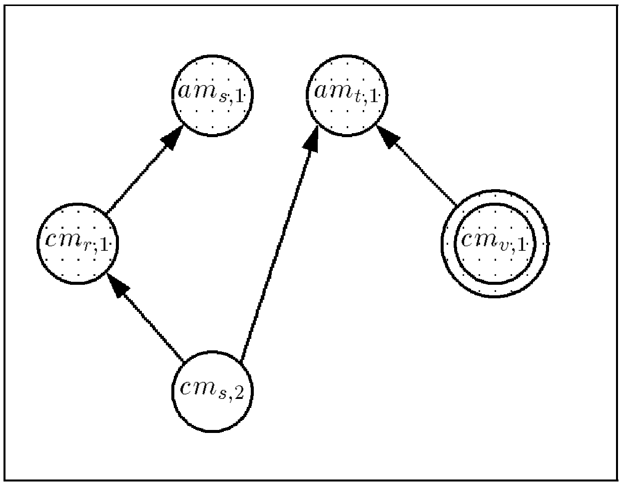

Figure 9: dag_r — Process r receives $am_{s,1}$ and $am_{t,1}$ sends $cm_{r,1}$ and receives $cm_{s,2}$ causally following $cm_{r,1}$. It is waiting for message $cm_{v,1}$ in order to complete an A-cut.

we chose to use the Ensemble algorithm. For the sake of completeness we decided to include here a description of Lansis' original algorithm.

This section provides an overview of the membership algorithm used in Lansis [DMS94] as well as a general discussion of membership algorithm semantics and interface. The Lansis algorithm is symmetric, that is, all processes play an equivalent role in it.

As described in the system model (see Section 3), the membership algorithm has to compute the E-reachability relation and its equivalence classes. In asynchronous environments this can only be approximated and this is what the algorithm does.

The membership algorithm at process p (Mem_p) uses a *failure detector* (FD) mechanism which notifies of any changes in the reachability relation (see Figure 5). When a process q becomes reachable a notification FD_rch(q) is delivered by the failure detector to Mem_p. When process q becomes unreachable a notification FD_norch(q) is delivered to Mem_p. In order to compute E-reachability, Mem_p broadcasts its approximation of the current network component. This is done recursively until a good enough approximation is achieved.

Mem_p is responsible for maintaining p's view V_p of the network component it sees. Mem_p is activated When network instability is detected. When the above computation terminates p is delivered a view notification with the new view.

4.6.1 The Mem_p State Machine

The membership algorithm can be described as a state machine (see Figure 10). Initially p has a view V_p of its group membership. Due to detected failures it recomputes the view via an iterative procedure in which a candidate view CM_p is computed. CM_p is initialize with a first approximation of the membership. This approximation is obtained by creating a list of all reachable members, observed during a preset period of time. Henceforth CM_p is modified as appropriate. In every iteration each process p broadcasts CM_p and collects CM_q from every process q in CM_p. If CM_p is equal to CM_q for every such q then an agreement has been reached. At this point p decides that the new view is $V_p = CM_p$.

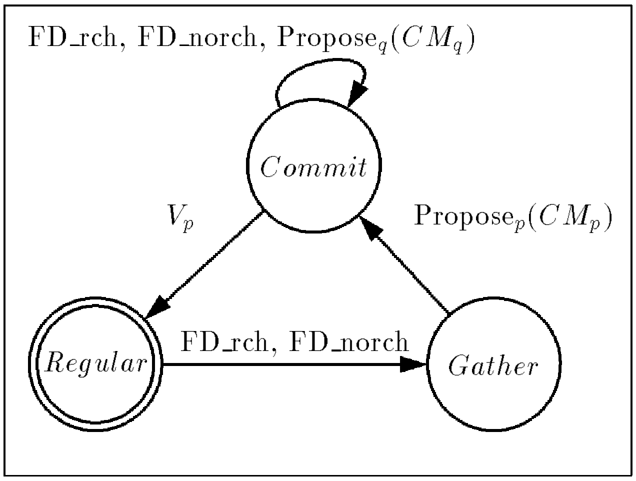

Figure 10: Membership states

We now describe the algorithm in a formal fashion. Let Q be a set of processes and $\text{Propose}_p(Q)$ be a procedure defined by:

$$\text{Propose}_p(Q) \stackrel{\text{def}}{=} CM_p := Q \text{ and } \text{C\_send}_p(Q)$$

Mem_p maintains the approximation of the component in a local variable CM_p. The state machine Mem_p has three states (see Figure 10): *Regular*, *Gather*, and *Commit* defined as follows:

Regular: Mem_p is inactive in this state. Upon receiving an FD_norch/FD_rch, switch to *Gather*.

Gather: Mem_p waits a preset period of time and then sets CM_p to the list of all currently reachable processes. It then proposes CM_p:

1. $\text{Propose}_p(CM_p)$
2. Switch to *Commit*.

Note: The size of CM_p may be different than the previous view. In particular it can be larger unlike the *Commit* state which can only reduce the size of CM_p.

Commit: Mem_p recursively performs:

1. If FD_rch(q) then ignore
2. If FD_norch(q) then $\text{Propose}_p[CM_p \setminus \{q\}]$
3. If $\text{C\_deliver}_p(CM_q)$ and $CM_q = CM_p$ for all $q \in CM_p$ then decide on a new view $V_p = CM_p$. Return to *Regular*.
4. If $\text{C\_deliver}_p(CM_q)$ from q and $p \in CM_q$ and $q \in CM_p$ then $\text{Propose}_p[CM_p \cap CM_q]$
   else $\text{Propose}_p[CM_p \setminus \{q\}]$.

Note: At this stage processes iteratively propose the new component CM_p. When all processes $q \in CM_p$ propose CM_p a decision is reached. At this stage CM_p can only "shrink" since addition of new processes into CM_p is not possible. In the worst case a singleton view $V_p = \{p\}$ will be decided by p since no other process agrees to its proposals.

4.6.2 Scenarios

In the following we say that process p *decides* Q if p decides to make Q its view V_p. We denote this by $D(Q)$. Let $P(Q)$ is a shorthand for $\text{Propose}_p(Q)$.

Example 1 In the *Commit* phase, the view can only decrease:

iteration/id	p	q	t	s
1	$P(\{pqst\})$	$P(\{pqst\})$	$P(\{qst\})$	$P(\{pst\})$
2	$P(\{ps\})$	$P(\{qt\})$	$P(\{st\})$	$P(\{st\})$
3	$P(\{p\})$	$P(\{q\})$	$D(\{st\})$	$D(\{st\})$
4	$D(\{p\})$	$D(\{q\})$		

Example 2 Agreement is not accurate, since when p decides that its group membership is V_p it does know that $\forall q \in V_p \mid \text{Propose}_q(V_p)$. But this does not imply that $\forall q \in V_p \mid V_q = V_p$.

In the following example p decides $V_p = \{pqt\}$. Processes q and t, after proposing $\{pqt\}$, detach from p. They decide that p is faulty, propose $\{qt\}$, and decide $V_q = V_t = \{qt\}$.

iteration/id	p	q	t
1	$P(\{pqt\})$	$P(\{pqt\})$	$P(\{pqt\})$
2	$D(\{pqt\})$		
3		$P(\{qt\})$	$P(\{qt\})$
4		$D(\{qt\})$	$D(\{qt\})$

Example 3 In the following scenario p decides $V_p = \{pqt\}$. Processes q and t have not yet received p's proposal and they have not decided yet. Thus, when p decides V_p it still does not know if $q, t \in V_p$ have decided. Process p only knows that q and t have proposed.

iteration/id	p	q	t
1	$P(\{pqt\})$	$P(\{pqt\})$	$P(\{pqt\})$
2	$D(\{pqt\})$		

4.6.3 Failure Notifications

The membership algorithm provides failure notifications as well as view notifications to the SVS layer. A failure notification is delivered if a process is decided to be unreachable. This may occur either due to a local FD_norch event, created by the failure detector, or due to the need to agree on the view.

In the example below processes p and q are initially in view $V_1 = \{pq\}$. They detach from each other deciding on views $\{p\}$ and $\{q\}$ respectively. Process p receives a failure notification for q before deciding on the new view $\{p\}$. Process q receives a failure notification for p before deciding on the new view $\{q\}$.

In the example $F(Q)$ denotes that the processes in the set Q have failed.

iteration/id	p	q
1	$D(\{pq\})$	$D(\{pq\})$
2	$F(\{q\})$	$F(\{p\})$
3	$D(\{p\})$	$D(\{q\})$

4.6.4 Constraining View and Failure Notifications

Membership is a non-trivial concept which calls for a simple programming model. Therefore we impose restrictions on view and failure notifications. The Lansis algorithm does not satisfy the constraints, though the Ensemble algorithm does. Constraining the algorithm gives it simpler more convenient semantics at the price of, in certain scenarios, the addition of view notifications (up to a multiple of 2). We assume that the network is stable most of the time thus we do not incur a performance penalty for using a less efficient membership algorithm.

Assume that process p is passing from V_1 to V_2. We require:

1. Process p receives a failure notification (exactly) for all processes in $V_1 \setminus V_1 \cap V_2$ between view notification V_1 and view notification V_2.

Note that if p receives a failure notification for q then q will not be included in V_2. Further, we constrain the membership algorithm such that:

2. If a member q fails during the negotiation of V_2 then a failure notification is delivered to p, and q is excluded from V_2 (although it may later rejoin).

We further constrain view changes; assume p passes from V_1 to V_2:

3. V_2 may either be:
   (a) A *split*: $V_1 \supseteq V_2$
   (b) A *merge*: $V_1 \subseteq V_2$

The Lansis algorithm does not adhere to the second nor the third criteria. Below we give specific scenarios. Throughout the examples we assume that the universe $\mathcal{P}$ contains processes $\{pqs\}$.

Criteria 2: The initial view of all processes is $V_1 = \{pqs\}$. Process s temporarily detaches from p and q; p, q, and s activate their membership mechanisms. During the *Gather* phase s reconnects to p and q; it is included in the first approximation $CM_p = CM_q = CM_s = \{pqs\}$. No failures occur henceforth; p, q, and s decide $V_2 = \{pqs\}$.

iteration/id	p	q	s
initial view	$\{pqs\}$	$\{pqs\}$	$\{s\}$
failures	$F(\{s\}$	$F(\{s\}$	$F(\{pq\}$
Gather proposal	$P(\{pqs\}$	$P(\{pqs\})$	$P(\{pqs\})$
final view	$\{pqs\}$	$\{pqs\}$	$\{pqs\}$

Criteria 3: Processes p and q are initially in the same component. Process s is initially in a different component. Process s reconnects to p and q while p detaches. Process p forms a view by itself while processes q and s both propose $\{qs\}$ and form the view $\{qs\}$.

iteration/id	p	q	s
initial view	$\{pq\}$	$\{pq\}$	$\{s\}$
proposition	$P(\{p\})$	$P(\{qs\})$	$P(\{qs\})$
final view	$\{p\}$	$\{qs\}$	$\{qs\}$

From here on we assume that views may be either merges or splits and that failure notifications adhere to the above criteria.

4.7 The Virtual Synchrony Model and Algorithm

The virtual synchrony model gives the application a simple programming model despite the asynchronous nature of the network. For a simple example consider a set of processors $\mathcal{P}$ that may suffer from crash failures. Assume that processors s, p, and q are in $\mathcal{P}$ and that s broadcasts a message m to p and q, and then fails. Processors p and q may observe the failure of s at different times. Thus, p may get the message m and then a failure notification about s while q will first observe the failure of s and then receive the message m. Guaranteeing applications that failures are perceived "atomically" could simplify application writing considerably. This is the rational behind the virtual synchrony model.

The virtual synchrony model has many variants. Here we reference three specifications of increasing strength — *Virtual Synchrony* (VS) [BVR94], *Extended Virtual Synchrony* (EVS) [MAMSA94], and *Strong Virtual Synchrony* (SVS) [FVR95]. The algorithms implementing these specifications are executed only in the event of a view change. The stronger the specification, the stronger the guarantees made to the application. Stronger guarantees come at the expense of a longer delay at view changes.

Among the alternative virtual synchrony specifications Lansis supports EVS. The virtual synchrony functionality and the membership functionality are coupled in Lansis into a single algorithm. Our approach is to provide the two functions separately. In Lansis/E we developed an algorithm that achieves SVS and uses the Ensemble membership algorithm. SVS was preferred over EVS in order to provide the application builder with an easier programming model than before.

Since Lansis/E supports SVS it incurs a longer delay than Lansis in delivering new views. We assume that the network and machines are stable most of the time, thus view changes are rare. Since the virtual synchrony algorithm is activated only at instability periods, we do not suffer performance penalties for implementing SVS during regular execution.

The virtual synchrony model defines the relationship between view notifications and regular message delivery. In the sequel we shall say that an event e occurs at process p in view V_1 if and only if e happens after p decides on a view V_1 and before it decides on the consecutive view V_2.

We define (for general reference see Figure 5):

$$\text{SVS\_send}_p(V,m) \stackrel{\text{def}}{=} \text{SVS\_send}_p(m) \text{ in view } V.$$

$$\text{SVS\_deliver}_p(V,m) \stackrel{\text{def}}{=} \text{SVS\_deliver}_p(m) \text{ in view } V.$$

We shall say that a process p receives, at the application level, a message m in a view V if its Strong Virtual Synchrony layer preforms $\text{SVS\_deliver}_p(V,m)$. Assume that p and q are processes and that V_1 and V_2 are views. The following definitions provide execution criteria:

- An execution satisfies the *self delivery* criterion if and only if whenever $\text{SVS\_send}_p(V,m)$ is executed then $\text{SVS\_deliver}_p(V,m)$ follows.

- An execution satisfies the *agreement* criterion if and only if whenever p and q go together through consecutive view changes V_1 and V_2, they receive the same set of messages in V_1.

- An execution satisfies the *strong agreement* criterion if and only if whenever p goes through consecutive view changes from V_1 to V_2 then all the processes $q \in V_1 \cap V_2$ receive in V_1 the same set of messages that p receives in V_1

An algorithm that constrains all possible executions such that every execution adheres to the agreement and self delivery criteria satisfies the *virtual synchrony* (VS) specification. An algorithm that constrains executions such that they adhere to strong agreement as well satisfies the *strong virtual synchrony* (SVS) specification.

4.7.1 Scenarios

In this section we describe possible executions of three processes — p, q, and t. The processes send messages and decide on views according to failure situations. We consider four different executions and discuss whether VS and SVS hold.

Example 1: Self delivery is not preserved. In the following example processes p and q begin in view $\{pq\}$. Process p sends m_1 in view $\{pq\}$ and q sends m_2 in view $\{pq\}$. Processes p and q first execute a view change and then received m_1 and m_2 respectively. This is a violation of self delivery.

iteration/id	p	q
1	$\{pq\}$	$\{pq\}$
2	$\{p\}$	$\{q\}$
3	m_1	m_2
4	$\{pq\}$	$\{pq\}$

From here on we assume that all messages are self delivered; we also omit message sources. Notice that since the sender receives its own messages, it is always amongst the recipients. If a message m has only one recipient p then p must also be m's source.

Example 2: VS does not hold because agreement does not hold. In the example below processes p, q, and t begin in view $\{pqt\}$. Then t splits to form the singleton group $\{t\}$. Processes p and q form the view $\{pq\}$. Process p receives m_1 in $\{pqt\}$; q receives m_2 in $\{pqt\}$; t receives m_3 in $\{pqt\}$. Agreement does not hold since p and q go together from $\{pqt\}$ to $\{pq\}$, but do not receive the same set of messages in $\{pqt\}$. Thus VS does not hold.

iteration/id	p	q	t
1	$\{pqt\}$	$\{pqt\}$	$\{pqt\}$
2	m_1	m_2	m_3
3	$\{pq\}$	$\{pq\}$	$\{t\}$

Example 3: VS holds but SVS does not because agreement is preserved while strong agreement is not. In the following example q and t pass from $\{qt\}$ to $\{pqt\}$ together ($2nd$ iteration) receiving m_2 in $\{qt\}$. Process p receives m_1 and passes from $\{pq\}$ to $\{pqt\}$. We check all messages sent and show that agreement holds for each of them:

- Message m_2: q and t receive m_2 in view $\{qt\}$. They pass together from $\{qt\}$ to $\{pqt\}$, thus agreement holds.
- Message m_1: p receives m_1 in $\{pq\}$. Since no other process goes from $\{pq\}$ to $\{pqt\}$ then agreement holds.

Since agreement and self delivery hold then VS holds. Although VS holds SVS does not. Since $q \in \{pq\} \cap \{pqt\}$ then according to SVS q should have received all messages that p received in $\{pq\}$. But q does not receive m_1 in $\{pq\}$.

iteration/id	p	q	t
1	$\{pq\}$	$\{pq\}$	$\{t\}$
2		$\{qt\}$	$\{qt\}$
3	m_1	m_2	m_2
4	$\{pqt\}$	$\{pqt\}$	$\{pqt\}$

Example 4: SVS holds because strong agreement holds. Processes p and q pass from $\{pqt\}$ to $\{pq\}$ receiving m_1 and m_2 in $\{pqt\}$. Process t passes from $\{pqt\}$ to $\{t\}$ receiving m_3. For p and q we observe that $\{pq\} = \{pqt\} \cap \{pq\}$ and both p and q receive $\{m_1, m_2\}$ in $\{pqt\}$. For t we observe that $\{t\} = \{t\} \cap \{pqt\}$. Thus, t has to agree with itself on the set of messages sent in $\{pqt\}$ but that is obvious, thus strong agreement holds. Since strong agreement and self delivery hold then SVS holds.

iteration/id	p	q	t
1	$\{pqt\}$	$\{pqt\}$	$\{pqt\}$
2	m_1	m_2	m_3
3	m_2	m_1	
4	$\{pq\}$	$\{pq\}$	$\{t\}$

4.7.2 The Strong Virtual Synchrony Algorithm

As shown in Figure 5 the membership algorithm provides failure and view notifications to the SVS algorithm. Whenever the membership algorithm detects instability it activates the SVS algorithm so as to *stabilize* (or *synchronize*) the previous view — making sure that all messages sent in the previous view are stable, namely, that every such message has been acknowledged by all processes.

Consider a process p that is passing from view V_1 to V_2. To achieve VS each process p in V_1 blocks the application at p, and sends a new type of message termed Block (denoted bm) by executing C_send$_q$. Process p then negotiates the new view V_2 using the membership algorithm. It will install V_2 only after receiving Block messages from all live processes in V_1.

Any other process q passing with p from V_1 to V_2 observes the same set of process failures in view V_1 before the installation of V_2. Thus, q receives the same set of Block messages as p. To ensure that p and q deliver the same set of messages M_{V_1} in V_1 they deliver (in V_1) all messages causally preceding the set of Block messages.

In Figure 11 dag_s is depicted. The universe $\mathcal{P}$ includes processes r, s, t, and v. All the processes in $\mathcal{P}$ are initially in view $V_1 = \{rstv\}$. Process r sends $cm_{r,2}$ and partitions away. Processes s, t, and v block. Let M_{V_1} be the set of messages causally preceding the block messages; $M_{V_1} = \{cm_{r,1}, cm_{s,1}, cm_{t,1}\}$. M_{V_1} does not include $cm_{r,2}$ since it does not precede any of the block messages. Thus, s receives $\{cm_{r,1}, cm_{s,1}, cm_{t,1}\}$ in V_1. It has finished stabilizing the previous view and is ready to install the next view V_2.

Assume that t passes with s to V_2 and it has received all messages in dag_s except $cm_{r,2}$. Still, it delivers M_{V_1} as above since it delivers all messages preceding $\{bm_s, bm_t, bm_v\}$ in V_1. Thus we observe that VS is preserved.

View V_2 may vary according to the specific failure scenario encountered. The membership algorithm used in Lansis/E defines that if a process has failed during V_1 then it will not appear in V_2. Thus, in this example $r \notin V_2$.

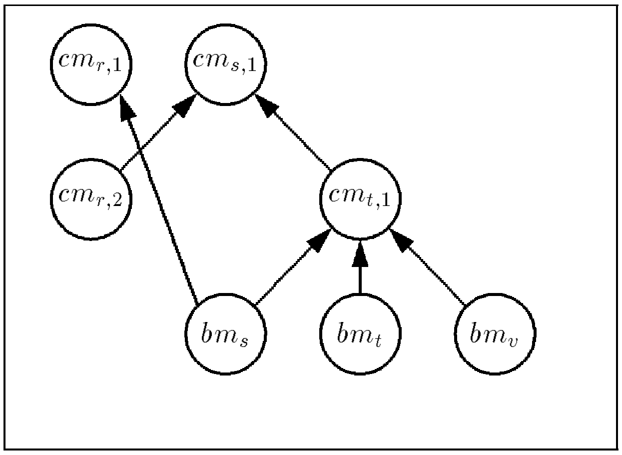

Figure 11: $\mathcal{P}= \{rstv\}$, dag_s is shown. All the processes are initially in view $V_1 = \{rstv\}$. Process r partitions away. Process s delivers $M_{V_1} = \{cm_{r,1}, cm_{s,1}, cm_{t,1}\}$ in V_1.

However, it is possible that one of s, t, and v will fail during the negotiation of V_2 in which case it will not appear in V_2. A member always appears in its own view. In this example V_2 at process s may be any subset of $\{stv\}$ that includes s.

To achieve SVS a process installs V_2 only after receiving acknowledgments for M_{V_1} from all live processes in V_1. This is performed via a round of acknowledgments for the set of block messages. If a process fails to acknowledge the exact set M_{V_1} then it is declared faulty, and will be included in V_2. In Figure 12 we continue the previous example.

Process s received block messages from s, t, and v. It now sends an acknowledgment message km_s that causally follows all three block messages. It waits for a round of messages km_s, km_t, and km_v from s, t, and v respectively.

Acknowledgment messages are received from s and t but not from v. These messages both follow bm_s, bm_t, bm_v, thus they causally follow all messages in M_{V_1}.

After waiting a preset time period s's failure detector decides that v is not reachable. Subsequentally Mem$_s$ gives a failure notification and s stops waiting for an acknowledgment from v and installs the new view V_2 after delivering M_{V_1} as before.

SVS is preserved in this protocol; assume p passes from V_1 to V_2 and

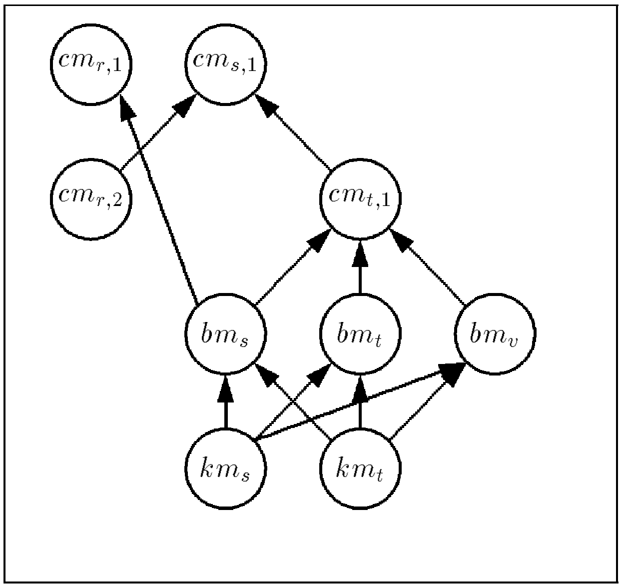

Figure 12: $\mathcal{P}$= $\{rstv\}$, dag_s is shown. All the processes are initially in view V_1. = $\{rstv\}$. Process r partitions away. Process s receives Block messages from s, v, and t, and another round of acknowledgments from s and v. It then delivers $M_{V_1} = \{cm_{r,1}, cm_{s,1}, cm_{t,1}\}$ in $V_1 = \{sv\}$.

$q \in V_1 \cap V_2$. Since q is in both V_1 and V_2 then p has not received a failure notification for it. Process p received an Acknowledge from q, thus, q receives (exactly) all messages in M_{V_1} which it delivers in V_1.

4.7.3 Causal Order

We now recall a problem encountered during causal ordering (see Section 4.3). For ease of reference we repeat it here:

> When a process fails its messages may never enter the dag of other processes. For example, assume process p executes C_send$_p(m_1)$ followed by C_send$_p(m_2)$ and then fails. Also assume that no other process receives m_1. Process q receives m_2 and requests m_1. Since no process can retransmit m_1 then message m_2 will always be pending at q and will never enter dag_q.

We erase the pending message buffer prior to an install operation of a new view since at this point in time message recovery is almost impossible.

Furthermore, assume, in the above example, that process q switches from view V_1 to V_2. Messages that remain in q's pending buffer (such as m_2) do not belong to M_{V_1} thus no process delivers them in V_1. It is therefore safe to erase the pending queue.

4.7.4 Message Ordering in the Face of Failures

Assume that a process p is switching from view V_1 to V_2. Before installing V_2 process p must deliver all messages in M_{V_1}. This is not a trivial task if there are Agree or Safe messages in M_{V_1} since these message types require special conditions for delivery. Further, it is apparent that Safe messages cannot get acknowledgments from all members of the view, if one of them fails. We first handle the algorithmic mechanisms and then deal with the change in message type semantics and guarantees.

Delivering a Safe message m at process $p \in V_1$ may be problematic if some process $q \in V_1$ partitions before acknowledging m. Consider the scenario depicted in Figure 13. The universe $\mathcal{P}$ contains processes $\{rst\}$. Processes r, s, and t are initially in view $V_1 = \{rst\}$. Process r sends Safe message $sm_{r,1}$, s sends a causally following message $cm_{s,1}$ and t partitions from r and s before acknowledging $sm_{r,1}$.

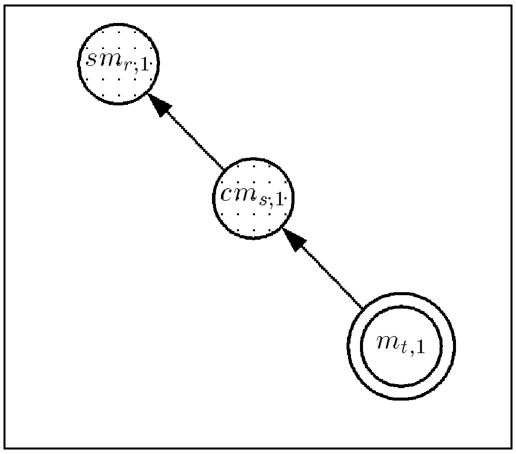

Figure 13: $\mathcal{P} = \{rst\}$. dag_s includes $sm_{r,1}, cm_{s,1}$ but not $m_{t,1}$. Process s cannot complete an S-cut for $sm_{r,1}$.

Process s cannot deliver $sm_{r,1}$ and $cm_{s,1}$ since it will never receive a message from t causally following $sm_{r,1}$. Thus, it will never complete an

S-cut for $sm_{r,1}$. After a timeout, s's failure detector decides that t has failed. Subsequentally Mem_s gives a failure notification for t and initiates the synchronization of the previous view V_1. After synchronization s knows that no new messages will arrive in V_1. Thus, the Safe delivery algorithm changes its criterion to "best effort" and delivers $sm_{r,1}$ via the S-cut $\{sm_{r,1}, cm_{s,1}\}$.

The best effort mode means changing the S-cut criterion to include the maximal amount of acknowledges possible. This means that for each Safe message sm the appropriate set $\mathcal{S}_{sm}$ includes all the acknowledgements in the dag for sm. In Figure 13 since dag_s includes $\{sm_{r,1}, cm_{s,1}\}$ then $\mathcal{S}_{sm_{r,1}} = \{sm_{r,1}, cm_{s,1}\}$. When the algorithm switches to the best effort mode it iteratively traverses the dag delivering all messages that are still delayed using the relaxed S-cut criterion.

We now turn to the Agree algorithm. In Figure 14 we depict a situation in which $\mathcal{P} = \{rstv\}$ and processes r, s, t, and v are initially in view $V_1 = \{rstv\}$. Processes r, s, t, and v send Agree messages $am_{r,1}, am_{s,1}$, and $am_{t,1}$ and a Causal message $cm_{v,1}$. Then r, s, t, and v split and form views $\{r\}, \{s\}, \{t\}$, and $\{v\}$ respectively. They synchronize the view V_1 and intend to install the new view(s) V_2.

In the figure dag_s is depicted. Process s cannot A_deliver_s $M_{V_1} = \{am_{r,1}, am_{s,1}, am_{t,1}\}$ since an A-cut containing these messages cannot be found in dag_s. A message from v ($cm_{v,1}$) is missing. Since V_1 is already synchronized s knows that new messages will not arrive in view V_1. Thus, the agreed order algorithm changes mode to best effort, that is trying to find the largest possible A-cuts in (what remains of) dag_s. In this example the best A-cut ($\mathcal{L}$,$\mathcal{A}$) is $\mathcal{L} = \{am_{r,1}, am_{s,1}, am_{t,1}\}, \mathcal{A} = \phi$. Thus, it will deliver $am_{r,1}$ then $am_{s,1}$ then $am_{t,1}$. We shall return to the best effort mode later. We now continue the example from another perspective.

In Figure 15 we depict dag_t. Process t has a different picture of the dag, receiving all messages in dag_t except $am_{r,1}$. Still, t's best A-cut ($\mathcal{L}$, $\mathcal{A}$) is defined by $\mathcal{L} = \{am_{s,1}, am_{t,1}\}, \mathcal{A} = \{cm_{v,1}\}$. Therefore, t will deliver $am_{s,1}$ then $am_{t,1}$. Although s and t are not in the same view V_2 they deliver in the same order Agree messages from the previous view V_1. A protocol guaranteeing this adheres to *inter-partition ordering*. Formally, the inter-partition ordering criterion is: delivery of Agree messages in uniform order throughout $\mathcal{P}$ regardless of partitions.

The best effort mode in the Agree algorithm case means changing the A-cut criterion to include for each $\mathcal{L}$ the maximal set $\mathcal{A}$ in the dag. As in the Safe algorithm when the algorithm switches to a best effort mode, it has a bounded unchanging dag in which it iteratively finds A-cuts and delivers

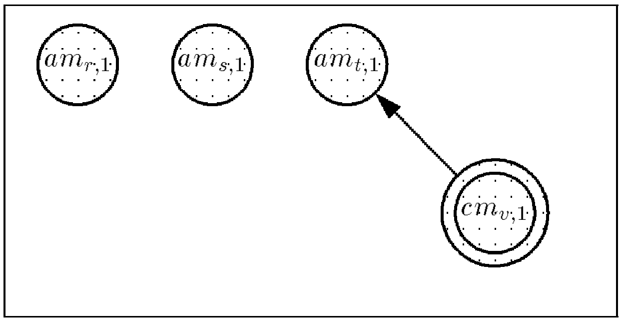

Figure 14: $\mathcal{P} = \{rstv\}$. $V_1 = \{rstv\}$; dag_s includes $am_{r,1}, am_{s,1}$, and $am_{t,1}$. Process s cannot complete an A-cut since $cm_{v,1}$ is missing. Process v is detached and this message cannot be received.

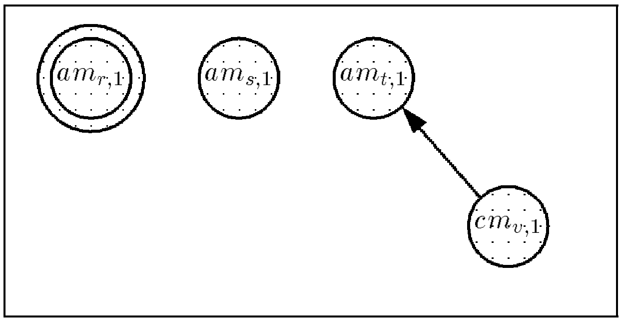

Figure 15: $\mathcal{P} = \{rstv\}$. dag_t includes $am_{s,1}, am_{t,1}, cm_{v,1}$. Process t cannot complete an A-cut in its dag since a message from r is missing.

them.

In Figure 16 dag_t is depicted. The universe includes processes $\{rst\}$ and for all processes in $\mathcal{P}$, $V_1 = \{rst\}$. A view change occurs and t is left with dag_t after it synchronizes V_1. It now has two Agree messages which are undeliverable according to the regular criteria. For $am_{s,1}$ it traverses the dag but finds no acknowledgment from r. Thus, the A-cut for $am_{s,1}$ includes $\mathcal{L} = \{am_{s,1}\}, \mathcal{A} = \{cm_{t,1}\}$. For $am_{s,3}$ there are no acknowledgments from either r or t. Thus its A-cut is $\mathcal{L} = \{am_{s,3}\}, \mathcal{A} = \phi$.

4.7.5 Complete View of the Dag

This subsection describes the interface that the SVS layer has with the other layers.

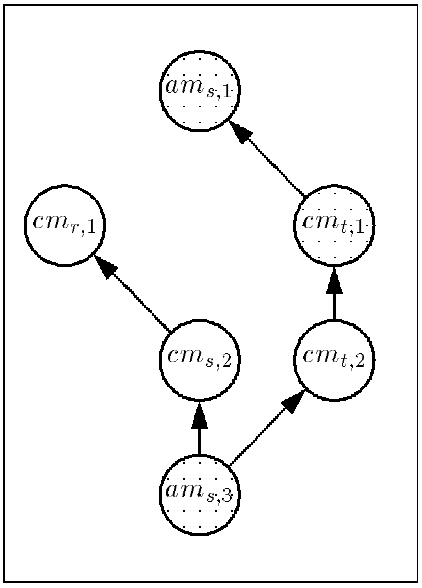

Figure 16: $\mathcal{P} = \{rst\}$. The figure depicts dag_t. The first A-cut delivered includes $\mathcal{L} = \{am_{s,1}\}$, $\mathcal{A} = \{cm_{t,1}\}$. The second includes $\mathcal{L} = \{am_{s,3}\}, \mathcal{A} = \phi$. Message $cm_{r,1}$ is immediately deliverable and t would not end up with it after the synchronization of V_1. It is drawn to show the existence of process r.

The SVS layer needs a view of the complete dag. Receiving messages from the Agree layer in the form of A_deliver$_p$ events is not enough. Examine Figure 17. The universe consists of processes r, s, and t; all processes start with $V_1 = \{rst\}$; the figure shows dag_s. Process r detaches form s and t; synchronization of V_1 ensues. At s, since $am_{s,1}$ is an Agree message the Agree layer blocks the dag until a complete cut for $am_{s,1}$ is received. Note that $cm_{r,1}$ is immediately deliverable, it is drawn to show the existence of r. The SVS layer must bypass the Agree layer in order to receive the Block messages bm_s and bm_t. This is done via a direct access to the dag (see Figure 5).

The same problem occurs for the SVS layer with the Safe layer and with the Acknowledge messages.

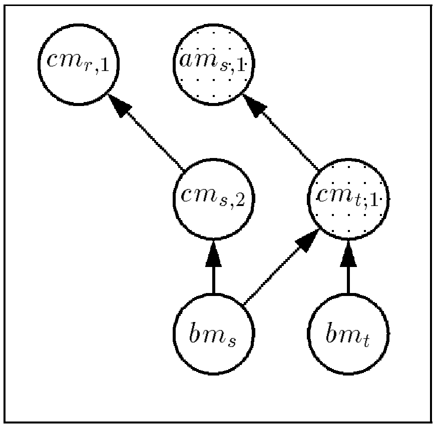

Figure 17: $\mathcal{P} = \{rst\}$. The figure depicts dag_s after the first phase of view synchronization. Message $cm_{r,1}$ is drawn to show $cm_{r,1}$. The Agree layer delays all messages in dag_r (except $cm_{r,1}$).

4.7.6 EVS Message Ordering

The EVS paradigm defines an application notification called *transitional view*. This notification is delivered when the Agree and Safe algorithms switch to best effort mode. The transitional notification indicates that from this point on there are holes in the total order and safe messages are delivered without acknowledgments from the full membership of the previous view.

The transitional notification alerts the application that the semantics of Agree and Safe have changed. This may be important for certain applications. For example, if a Safe message is a database transaction ([Gra78],[Ske82]) then the application should be notified that, possibly, not all destinations received it. Furthermore, since the environment is lossy and asynchronous it is not always possible to ensure message safety, uniform total ordering, and reliability.

5 Ensemble

This section describes the Ensemble system [Hay96]. The system is written in OCAML [Ler97], a novel dialect of ML, developed at the French institute for computer science research at INRIA. OCAML is a high level functional language that is machine independent. Ensemble is written as a set of protocols stacked one on top of the other in a fashion similar to the ISO model.

In Ensemble each layer implements a simple protocol using its local state (set of local variables) communicating with the layers above/below it using the Ensemble message passing infrastructure. Logically, an Ensemble stack acts as a transport layer hiding the inconvenient nature of the network (asynchrony, link failures, node crashes, etc.) from the application.

The user can pick and choose layers, thus configuring a particular transport layer. The user can choose reliable/fifo-ordered/totally-ordered communication, using ATM/UDP/Deering UDP, employing the membership service, etc.. The regular stack, which will be discussed further below, supports virtual synchrony and reliable fifo.

The following subsection is based on the Ensemble manual [Hay96] and is given here for the sake of completeness.

5.1 Infrastructure

5.1.1 Event Protocol: Intra-Stack Communication

Ensemble embodies two forms of communication. The first is communication between protocol stacks in a process group using messages sent via some communication transport (UDP, ATM, etc.). The second is inter-stack communication between protocol layers using Ensemble events. See Figure 18.

When the application sends a message m its content is stored in a common message buffer and an event em corresponding to the message is created. The em event contains an opaque pointer to the message data. It is propagated down the stack; each layer attaches a header to em. The lowest layer in the stack bundles, marshals, and sends em + content of m.

The receiving stack unmarshals the message, stores the message content in the common message buffer, and creates an event corresponding the message. The event contains the message headers and an opaque pointer to the message content. It is propagated up the stack; each layer strips off its header (only). Finally, the application receives the event and reads the message content. Ensemble does not read/use the message content with the exception of (possible) encryption/decryption/hash operations.

The above type of event is the most common — directly related to messages. However, events are also used for notifying layers of group membership changes, telling other layers about suspected failed members, synchronizing protocol layers for view changes, passing acknowledgment and stability information, alarm requests and timeouts, etc.. In order for a set of protocol layers to harmoniously implement a higher level protocol, they have to agree on what these various events "mean" and follow what is called here the Ensemble *event protocol*.

The layering in Ensemble is advantageous because it allows complex protocols to be decomposed into a set of smaller, more understandable protocols. However, layering also introduces the event protocol which complicates the system through the addition of intra-stack communication.

Ensemble, unlike Lansis, uses the message passing paradigm for inter layer communication. This allows layers to be physically separated, i.e., they may reside on different machines. The Maestro system [BFH97] exploits this to separate the membership layers from the rest of stack. In order to avoid unnecessary coping of message content when crossing a layer boundary, when layers are situated on the same machine, content is stored in a shared buffer. This is performed by the Ensemble infrastructure, transparent to the layers. This has the effect of using the general message passing semantics while writing layer code, relying on the infrastructure to optimize the layer code when possible.

5.1.2 Callbacks

Layers are implemented as a set of callbacks that handle events passed to them from other layers. Logically, a layer is initialized with one callback for passing events to the layer above it, and another callback for passing events to the layer below it. After initialization, a layer returns two callbacks to the system: one for handling events from the layer below it, and another for handling events from the layer above it. We shall refer to these callbacks as — UpRc and DnRc. Each layer uses two other callbacks given by the layers above/below it — UpSd and DnSd.

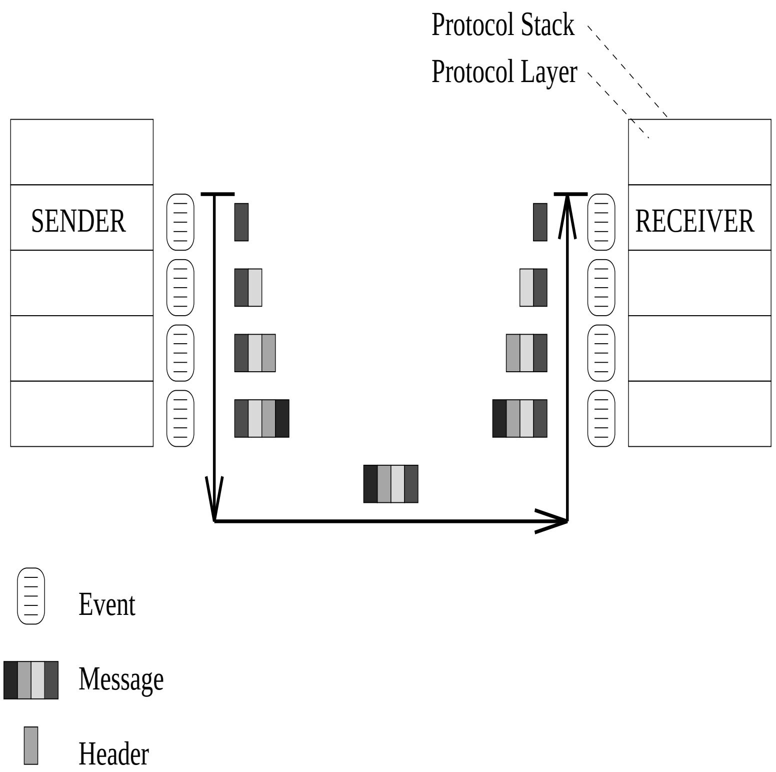

Figure 18: Events are used for intra-stack communication: layers can only communicate with other layers by modifying events; layers never read or modify other layer's message headers. Messages are used for intra-stack communication: only messages are sent between group members; events are never sent between members.

5.1.3 Ordering Properties

The system infrastructure that handles scheduling of protocol layers and the passing of events between protocol layers provides the following guarantees:

Fifo ordering: Events passed between two layers are delivered in order. For instance, if layer A is stacked above layer B then all events layer A passes to layer B are guaranteed to be delivered in fifo order to layer B. In addition, events that layer B passes up to layer A are guaranteed to be delivered in fifo order to layer A. Note that these ordering properties allow the scheduler some flexibility because they

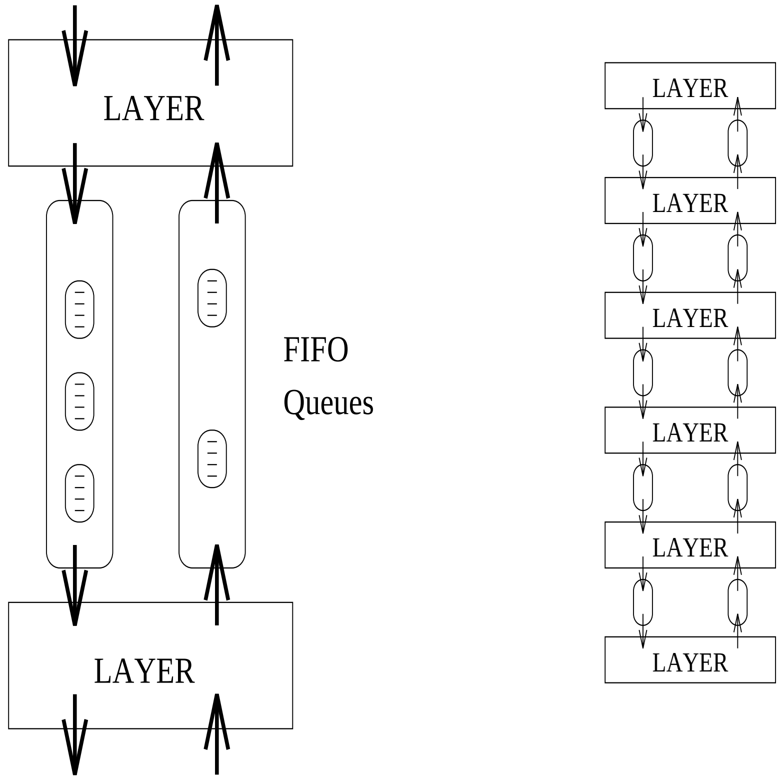

Figure 19: Layers are executed as I/O automata, with pairs of FIFO event queues connecting adjacent layers.

only specify the ordering of events in a single channel between a pair of layers.

No Concurrency : The system infrastructure that hands events to layers through the callbacks never invokes a layer's callbacks concurrently. It guarantees that at most one invocation of any callback is executing at a time and that the current callback returns before another callback is made to the protocol layer. See Figure 19 for a diagram of a layer automata. Note that although a single layer may not be executed concurrently, different layers *may* be run concurrently by the scheduler.

The execution of a protocol stack can be visualized as a set of protocol layers executing with a pair of event queues between each pair: one queue

Layer	Functionality
Gmp	Membership algorithm
Sync	Block during membership change
Top_appl	Interface to the application
Sequencer	Total ordering
Stable	Stability detection
Mnak	Reliable fifo
Bottom	Interface to the network

Table 1: The basic Ensemble stack

for events going up and another for events going down. Each protocol layers is then an automata, repeatedly scheduled to take pending events from one of the adjacent incoming queues. Each event is then executed, and zero or more events are deposited into two adjacent outgoing queues (see Figure 19).

5.2 The Basic Ensemble Stack

In Table 1 the basic Ensemble stack is depicted. Several layers have been excluded since their description is not relevant to this work. Below, we describe each layer and its significance. We shall describe the layers, as in Section 4 on Lansis/E, from the bottom up.

The bottom layer interfaces with the network providing an abstract N_send$_p$/N_deliver$_p$ interface. Its description is not relevant to this work.

The Mnak and Stable layers provide reliable fifo while the Sequencer layer provides total order. The Top_appl layer interfaces with the application allowing it to send and receive messages. Logically, the application is above the stack, but application messages bypass the Gmp and Sync layers. Thus, the application interface was positioned as close as possible to the network to allow for quicker access. In this way application messages only traverse the layers below the Top_appl later until they reach the network.

The Sync layer blocks the stack prior to a view change.

The Gmp is actually a substack of seven layers. It is responsible for maintaining the current view. We are not interested in the inner workings of the Gmp thus we do not describe it here. It suffices to know that it provides view and failure notifications to the rest of the stack. Ensemble supports the virtual synchrony model (see Section 4.7).

Each group component has a *leader*. The leader's main responsibility is the membership protocol. When a member suspects another of being faulty it sends this as a *suspicion* message to the leader. The leader *fails* suspected members $p_1, \ldots, p_k$ via a reliable broadcast of a $\mathit{fail}(p_1, \ldots, p_k)$ message. Such a message is passed in the stack in the form of an UpFail event. New views are reliably broadcasted by the leader and are passed in the stack in the form of a UpView event. All live members receive the same set of failure messages before they get a new view.

View changes are implemented in a clever way. When a process p passes from view V_1 to V_2 a new stack is initialized for V_2 This facilitates using different protocols in V_2 and helps in optimization [HVR96]. Since the network is asynchronous, the old stack is kept alive until all processes have passed from V_1 to V_2. It is subsequentally garbage collected. We say that a process *installs* a new view when it performs the above. We say that a leader installs a new view when it reliably broadcasts a new view V_2 and locally installs V_2 (as above).

5.3 Mnak

The Mnak (Multicast NAK) layer implements a reliable fifo-ordered broadcast protocol. Broadcast messages from each sender are delivered in fifo-order at their destinations. Messages from live members are delivered reliably and messages from failed members are retransmitted by live members.

The Mnak layer keeps a record of all broadcasted messages such that it will be able to retransmit any message on demand. It uses the *StblVct* vector created by the Stable layer for garbage collection. This vector, received from the Stable layer via the *DnStable* event (see below), defines the set of stable messages. The Mnak layer discards these messages upon receipt of the DnStable event. The *NumCast* vector (also included in the DnStable events) gives an indication of missing messages. The Mnak layer proceeds to try to recover these messages.

An UpFail event specifies a list of members that have failed. When such an event arrives, the Mnak layer stops receiving messages from the failed members. It still accepts retransmissions of failed members messages.

5.4 Stable

A message is defined as *stable* once all members of the group acknowledge it. The Stable layer tracks the stability of broadcast messages. Each member maintains a vector indicating the newest message (reliably) received from each member in the view. This is termed $acks_p$ for process p. Every preset

id's	p	q
p	3	1
q	1	2
NumCast	3	2
StblVct	1	1

Table 2: $\mathcal{P} = \{p, q\}$. Process p acknowledged messages $m_{p,1}, m_{p,2}, m_{p,3}$, and $m_{q,1}$ while q acknowledges $m_{p,1}, m_{q,1}$, and $m_{q,2}$.

time period p *gossips* its $acks_p$ vector. *Gossip* means to unreliably broadcast a message to anyone who can hear. Process p maintains an acknowledgment matrix containing the newest $acks_q$ vector for each $q \in V_q$. Thus, the acknowledgment matrix (of size $|\mathcal{P}|^2$) contains $acks_p$ in row p.

In the matrix each entry (s, t) corresponds to the number of member s's messages member t has acknowledged (the diagonal entries, (s, s) contain the number of broadcast messages sent by s). The minimum of column s (disregarding entries for failed members) is the number of broadcast messages from s that are stable. The vector of these minimums is called the stability vector (*StblVct*). The maximum of column s (disregarding entries of failed members) is the number of broadcast messages member s has sent that are held by at least one live member. The vector of the maximums is called the *NumCast* vector. Examine Table 2. The universe consists of processes p and q. Process p acknowledges messages $m_{p,1}, m_{p,2}, m_{p,3}$, and $m_{q,1}$ while q acknowledges $m_{p,1}, m_{q,1}$, and $m_{q,2}$. We assume that both p and q have received these acknowledgment vectors. Thus, the NumCast vector includes (3,2) and the Stability vector includes (1,1) at both p and q.

Occasionally, the protocol layer recomputes the stability and NumCast vectors and delivers them down in a *DnStable* event. The Bottom layer then bounces this event up as an *UpStable* event. The NumCast vector gives an indication to lost messages. The Stability vector is used for garbage collection. The DnStable event consists of both vectors.

Failing members may cause problems. Consider the example in Table 3. The universe $\mathcal{P}$ contains three processes $\{pqt\}$. Processes q and t reliably send messages $m_{q,1}$ and $m_{t,1}$ which q and t receive. Process p sends $m_{p,1}$ which no other process receives. It proceeds to send a gossip message declaring that it has reliably received $m_{p,1}$ and subsequentally fails. The acknowledgment matrix at processes q and t is depicted in Table 3.

In the q and t rows we observe that both processes have reliably received

id's	p	q	t
p	1	0	0
q	0	1	1
t	0	1	1
NumCast	1	1	1
StblVct	0	0	0

Table 3: Acknowledgment matrixes of process q and t.

id's	p	q	t
p	1	0	0
q	0	1	1
t	0	1	1
NumCast	0	1	1
StblVct	0	1	1

Table 4: Process q and t's acknowledgment matrix without p.

messages $\{m_{q,1}, m_{t,1}\}$. At p's row though, we observe that there is another message that p and q have not received — $m_{p,1}$. Now q and t proceed to try to recover that message. When p fails they will never succeed. Furthermore, p did not acknowledge the receipt of $m_{q,1}$ and $m_{t,1}$, thus those messages cannot become stable.

To solve this problem, when p fails it is not taken into account when computing the NumCast and StblVct vectors, thus we obtain the matrix in Table 4.

5.5 Sequencer

The Sequencer layer provides total ordering. The protocol used is simple. Members buffer all messages received from below in a local buffer. The leader periodically broadcasts an *ordering message*. This message determines the delivery order of messages currently in the buffer. All members proceed to deliver to the upper layer all messages in the buffer according to the leader's instructions.

In addition to the Sequencer layer, Ensemble provides two alternative layers called Totem and Total. As their names imply they implement the total ordering algorithms of the Totem [AMMS+95] and Total [AMMSA93] token-ring systems. The layers do not implement the membership and message reliability mechanisms of the systems.

The Sequencer, Total, and Totem layers totally order all messages. They do not allow different types of messages (Causal, Safe, Agree) as in Lansis. Furthermore, these algorithms do not support the inter-partition ordering property (see Section 4.7) that Lansis' agreed order provides. On the positive side, these algorithms typically have lower latencies than Lansis' agreed ordering.

5.6 Sync

This layer implements a protocol for blocking a group during view changes (see Section 4.7). The leader initiates the protocol by broadcasting Block. Other members block their stack by sending down a DnBlock event. This event bounces up from Bottom as UpBlock and bounces down from the top layer as DnBlockOk. The Sync layer catches this event and concurs to the leader with a BlockOk message. The leader, receiving BlockOk from all non-failed members, sends up an UpBlockOk event.

The up/down motion through the stack of the DnBlock event causes the stack to flush. After the Sync layer receives DnBlockOk it knows that no additional events are passing through the stack and all layers have been blocked.

Upon receiving an UpBlockOk from the Sync layer the leader knows that all surviving group members have stopped broadcasting in the previous view. It does not know yet whether all messages are stable in the previous view.

We shall say that the group is "blocked" once the leader has applied the Sync mechanism to stop broadcasts (and received an UpBlockOk).

5.7 Membership Algorithm — the Gmp Suite

Compared to other membership algorithms, this algorithm creates more view notifications. Keeping the protocol simple, it was not optimized to avoid such unnecessary notifications.

This algorithm, as we shall see later, satisfies the constraints of Section 4.6.4. We repeat them here:

Assume process p is passing from view V_1 to view V_2:

1. p receives a failure notification (exactly) for all processes in $V_1 \setminus V_1 \cap V_2$ between view notification V_1 and view notification V_2.

2. If a member q fails during the negotiation of V_2 then a failure notification is delivered to p, and q is excluded from V_2 (although it may later rejoin).

3. V_2 may either be:

   (a) A split: $V_1 \supseteq V_2$

   (b) A merge: $V_1 \subseteq V_2$

Members are ordered via some numbering scheme such that they have ranks $0, \ldots, n-1$. A leader is elected in each component; by default it is ranked 0. The leader is responsible for maintaining the current view by the reliable broadcast of membership events — failure and view notifications. It uses a synchronization mechanism, explained in the next section, to the effect of synchronizing (see Section 4.7) the previous view before *installing* (reliably broadcasting) the new view. The algorithm distinguishes between two cases — merge and split.

5.7.1 Merge

In order to facilitate merging of group views, each view leader gossips from time to time an "I'm alive" message. When a leader detects another view of the group it tries to merge with it. Denote the local leader as l and the remote leader as r. Process l initiates the following protocol:

1. l synchronizes its view.

2. l sends a merge request to r.

3. r synchronizes its view.

4. r installs a new view (with the mergers in it) and sends the view to l.

5. l installs the view in its group.

View merges are constrained so that:

- Views cannot be both merging and accepting mergers at the same time. This eliminates the potential for cycles in the "merge-graph."

- Only non-intersecting views can merge. If a view P tries to merge with a view Q, and $P \cap Q \neq \phi$ then Q's leader fails $P \cap Q$. This causes a new view, $Q' = Q \setminus P$, to be installed by all $p \in Q'$.

5.7.2 Split

When a member detaches from the view other members *suspect* it. Each member keeps a list of suspected members; these are members for which it has received a local failure detection via its local failure detector (as in Section 4.6). A member reliably broadcasts its suspicions to the view and members collect all suspicions. We say that p suspects q if q is in p's suspect list.

When view leader l suspects some members $p_{i_0}, \ldots, p_{i_k}$ it reliably broadcasts a $fail(p_{i_0}, \ldots, p_{i_k})$ message. A failure message, received by member p, is propagated in the stack in the form of an UpFail event. Process l then synchronizes the view and subsequentally installs a new view without $p_{i_0}, \ldots, p_{i_k}$. Non-leaders cannot fail other processes.

If the leader partitions from the view then a new leader is elected by the following protocol. If a process p_k in a view V_1 suspects that all lower ranked members are faulty then it elects itself as leader and broadcasts $fail(p_0, \ldots, p_{k-1})$. It subsequentally synchronizes V_1 and installs a new view V_2 without $p_0, \ldots, p_{k-1}$.

All other processes that suspect members $p_0, \ldots, p_{k-1}$ accept p_k's failure message and continue with p_k to the new view V_2 with p_k as the leader. A member who disagrees, since it does not suspect members $p_0, \ldots, p_{k-1}$, suspects p_k.

The leader of a singleton group is its only member.

5.8 Strong Virtual Synchrony

In Ensemble Strong Virtual Synchrony is implemented via a relatively simple leader-based protocol.

Before a membership change from V_1 to V_2 all messages in V_1 must become stable. The leader performs the following protocol before every membership change.

The leader activates the Sync protocol. After an UpBlockOk is delivered no new messages will be sent by live members in V_1. Furthermore, all live members stop receiving messages from failed members, thus the set, M_{V_1}, of messages that needs to be delivered in V_1 is bounded.

The leader waits until live members agree on M_{V_1} via sending negative acknowledgments and recovering lost messages. When the UpStable events (these are DnStable events bounced by the Bottom layer) it receives satisfy NumCast = StblVct the leader knows that this process is complete.

Any $p \in V_1$ receiving V_2 knows that all messages sent in V_1 are stable. Thus, any $t \in V_1 \cap V_2$ has received the same set of messages in V_1. This is equivalent to SVS. Thus, synchronizing a view before a view change ensures SVS.

We now discuss the problems raised in Section 4.7 and their solutions in Ensemble:

Causal Order: In Lansis there are out of order messages left in the Causal layer before the installation of a new view. These messages are discarded after the synchronization phase since there is no hope to deliver them in correct order.

Since the basic order type in Ensemble is fifo then the equivalent problem is out of order messages left in the Mnak layer before the installation of a new view. The Mnak simply discards such messages since, as in Lansis, there is no hope to deliver them in correct order.

Message Ordering in the Face of Failures: In Lansis delivery of Agree and Safe is blocked when processes fail. Only when the synchronization phase is over and a new view is installed can these messages be delivered.

In Ensemble such message types are not supported. Further, the total ordering layers provide weaker guarantees for totally ordered messages than Lansis agreed order.

Complete View of the Dag: In Lansis the SVS layer must have a complete view of the dag in order to bypass the Safe and Agree layers. Ensemble does not employ a dag structure but a similar problem occurs with the Sequencer,Total, and Totem layers. These layers may delay the Sync layer' messages, when a view change occurs. For example, the Sequencer layer may be waiting for an ordering message from a process who has failed. It will unblock when a new view is installed. Meanwhile it delays all messages on the way up the stack. In order to bypass these layers, the Sync layer' messages are mark "no_total", thus, they bypass the Sequencer, Totem, and Total layers.

EVS Message Ordering: Ensemble supports neither transitional notifications nor EVS message ordering.

5.9 Ensemble Approach to Performance

In Ensemble layers may communicate with each other even if they are far apart in the stack. Thus, the only consideration as to where to place a layer in the stack has to do with the *amount* of communication going on. Since Ensemble layers are designed not to communicate much among each other, the usual type of communication is between the application and the network.

Usually, the application sends a message and each layer adds a header to it and passes it down. The Bottom layer sends it on the network. At the destination/s each layer strips off its own piece of the header and passes the message up until it reaches the application.

Ensemble tries to shorten this path as much as possible. Thus, all layers which see little use are placed above the application. The Gmp and Sync layers, which are active (almost) only when view changes occur, are stashed there (see Table 1). The most active layers, the Mnak and Stable, are placed closest to the network.

New ways of optimizing stack have been suggested ([VR96], [HVR96]) and are being explored.

6 Lansis/E — Implementation in Ensemble

6.1 Overview

In order to implement Lansis/E in the Ensemble framework four major changes had to be made:

- *Linearization:* The interfaces between Lansis/E algorithms (recall Figure 5) are not linear, they had to be put into a strait line.
- *Introduction of the message passing paradigm:* Lansis/E modules all use the same dag. This is not feasible in Ensemble which uses the message passing paradigm for inter-layer communication.
- *New leader-based strong virtual synchrony algorithm:* The Gmp uses a leader-based synchronization protocol. The leader's Gmp activates its Sync layer. When the view is blocked the leader's Gmp performs stability detection. Thus, any sub-stack using the Gmp must provide a leader-based blocking protocol. We modified the original symmetric Lansis/E algorithm (Section 4.7) and created a leader-based protocol implemented in the Tsync layer.
- *Support for Message Types:* Ensemble does not allow the application to send different types of messages; Lansis/E supports sending three types of messages. This was a mostly technical modification and we do not go into the details here.

We implemented the Causal, Agree, and Safe algorithms as Ensemble layers. These were placed below the Appl_top layer in order to allow for good performance. In Table 5 the Lansis/E stack is depicted.

6.1.1 Message-Based Lansis/E

The algorithms described in the Lansis/E Section 4 assume knowledge of the complete dag by all modules. In Ensemble each layer has its own relatively small buffer and its interaction with other layers is minimal. In order for all layers to know the dag each layer would have to rebuild it separately. This would be inefficient.

We decided to let the Causal layer maintain the dag. To every message delivered m the Causal layer adds an acknowledgment vector. This vector defines the list of messages which m causally follows. Each layer

Ensemble	Lansis/E	
Gmp		Membership algorithm
	Tsync	Block during view change
Top_appl		Interface to the application
	Agree	Agreed Delivery Ordering
	Safe	Safe Delivery
	Causal	Causal Delivery Ordering
Bottom		Interface to the network

Table 5: The regular Lansis/E stack

above Causal can compare any two messages m_1 and m_2 and decide whether $m_1 \leftarrow m_2, m_2 \leftarrow m_1$, or if they are concurrent.

The Safe layer (see Section 4.4) needs to know only the set of oldest messages in the dag. Thus, it buffers that part of the dag which it needs. Using the additional information encapsulated by the acknowledgment vectors it can decide when Safe messages have been all-ack'ed.

Similarly to the Safe layer, the Agree layer (see Section 4.5) needs to know only the set of oldest messages in the dag. Thus, it buffers that part of the dag which it needs. Using the acknowledgment vectors it find A-cuts in the dag.

6.2 Causal Layer

This Lansis/E layer implements the causal delivery ordering of Section 4.3. This layer is responsible for:

- Causal delivery
- Message completion
- Garbage collection

The causal delivery layer "owns" the dag. Therefore, the interface to the upper layers had to be enriched. To this end the Causal layer adds to each message m an acknowledgment vector $acks_m$ before delivery to the layer above. This vector includes at each entry p the largest message sequence number from process p that m causally follows. For example, the vector $[5, 7, 13]$ acknowledges messages $m_{p,5}, m_{q,7}, m_{s,13}$ in view $\{pqs\}$. Thus, message $m_1 = m_{p,x}$ can be compared to message $m_2 = m_{q,y}$ via the ack vectors:

- If $acks_{m_1}[q] \geq y$ then $m_2 \leftarrow m_1$.
- If $acks_{m_2}[p] \geq x$ then $m_1 \leftarrow m_2$.
- Otherwise $m_1 \parallel m_2$.

Thus, the layers above Causal rely on correct causally ordered delivery of messages from the Causal layer. Furthermore, they can use the information attached to each message in the form of the $acks_m$ vector. Thus, any two messages m_1 and m_2 may be compared without accessing the complete dag (as done in Lansis).

6.3 Safe Layer

This Lansis/E layer implements the safe delivery ordering of Section 4.4.

Recalling the original algorithm, two message types are handled — Causal and Safe (Agree messages are equivalent to Causal messages with respect to this layer). Causal messages are immediately delivered to the layer above. For each Safe message sm an S-cut is created, thus delaying all messages causally following sm. We explain below the creation of such an S-cut.

For each Safe message sm, an S-cut $\mathcal{S}_{sm}$ is initialized; $\mathcal{S}_{sm} = \{sm\}$. Another data structure used is a queue in which messages are delayed until delivery. This queue is initialized empty.

Keeping the description simple, we describe the handling of a single Safe message. When multiple concurrent Safe messages are received the algorithm is executed in parallel for all of them.

We now describe what the layers does upon receipt of a message m:

- m is Causal:
    1. If m directly follows sm then $\mathcal{S}_{sm_i} := \mathcal{S}_{sm_i} \cup \{m\}$.
    2. If m indirectly follows sm it is enqueued.
    3. Otherwise it is sent upwards.
- m is Safe(we assume it is not concurrent to sm): It is treated as a Causal message (see above).

When an S-cut is completed for sm, sm is delivered. All Causal messages in the S-cut are delivered. The algorithm is recursively applied to all messages remaining in the S-cut and the queue.

6.4 Agree Layer

This Lansis/E layer implements the agreed delivery ordering of Section 4.5.

Recalling the original algorithm, two message types are handled — Causal and Agree (Safe messages are equivalent to Agree messages with respect to this layer). Causal messages are immediately delivered to the layer above. We explain below the creation of an A-cut ($\mathcal{L}$,$\mathcal{A}$).

The layer is initialized with $\mathcal{L}, \mathcal{A} = \phi$. Another data structure used is a queue in which messages are delayed until delivery. This queue is initialized empty.

We now describe what the layers does upon receipt of a message m:

- m is Causal: If m immediately follows some message in $\mathcal{L}$ it is added to $\mathcal{A}$. If it indirectly follows some message in $\mathcal{L}$ then it is enqueued. Otherwise it is sent upwards.

- m is Agree: If m is concurrent to the messages in $\mathcal{L}$ then it is added to $\mathcal{L}$. Otherwise, it is enqueued.

Once $|\mathcal{A} \cup \mathcal{L}| = |V|$ the algorithm deduces that an A-cut ($\mathcal{L}$, $\mathcal{A}$) is complete. It proceeds to order the messages in $\mathcal{L}$ according to process rank order and to deliver them. subsequentally, the messages in $\mathcal{A}$ are delivered as well. The algorithm is then recursively applied to the queue.

6.5 Strong Virtual Synchrony

This section describes how the Lansis/E strong virtual synchrony algorithm (see Section 4.7) is implemented.

6.5.1 SVS Algorithm — Tsync Layer

The Tsync layer implements the Strong Virtual Synchrony algorithm of Section 4.7.2. The original algorithm first broadcasts a Block message. When Block messages have been received from all live processes in the view it broadcasts an Acknowledge. When Acknowledge messages are received from all live processes in the view it installs the new view.

We modified the algorithm to be leader-based. In the revised algorithm the Tsync layer is activated by the Gmp via a DnBlock event. The group leader uses this event to request its Tsync layer to synchronize the view. The Tsync layer catches the BlockReq event and broadcasts a BlockReq message. Each member's Tsync layer receives this message and blocks its stack. It then replies with a Block message to the leader. The leader, upon receiving Block messages from all non-failed members, sends up an UpBlockOk. This event informs the Gmp that the previous view is synchronized. The Gmp then installs the new view V_2. The installation is performed via the reliable broadcast of the message $view(V_2)$.

An example scenario is given in Figure 20. The universe includes processes r, s, t, and v where s is the leader. The br message stand for the leader's BlockReq message. The bm messages stand for Block messages. V_2 represents the $view(V_2)$ message. The leader, s, broadcasts br_s the members concur via messages bm_r, bm_t, bm_v. Message bm_s is not sent since s counts itself as blocked. Process s then installs the new view V_2.

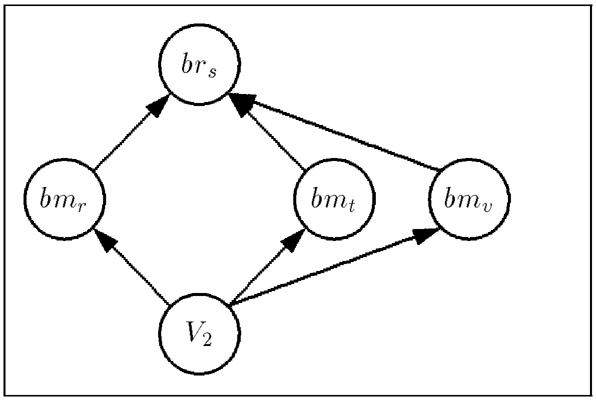

Figure 20: $\mathcal{P} = \{rstv\}$ where s is the leader. br denotes a BlockReq message; bm denotes a Block message; V_2 denotes the new view V_2. The leader, s, broadcasts a BlockReq (br), the members concur via Block messages (bm_r, bm_t, bm_v). Process s then installs the new view V_2.

The $view(V_2)$ message causally follows all the Block messages. Thus, receiving V_2 implicitly implies the receipt of all Block messages. We now view these messages as Block messages (recall Section 4.7). The arrival of $view(V_2)$ implies the receipt of all Block messages, therefore, the receiver has completed the blocking phase and can install the new view according to the old protocol. Thus, this protocol implements VS and the first stage in the SVS protocol.

To implement the second phase of the SVS protocol we added another round of acknowledgments. When the leader receives Block messages from all members it broadcasts a BlockReq2 message. Members acknowledge via an Ack message. These messages are equivalent to Acknowledge messages.

Upon receipt of all Ack messages, the leader installs V_2.

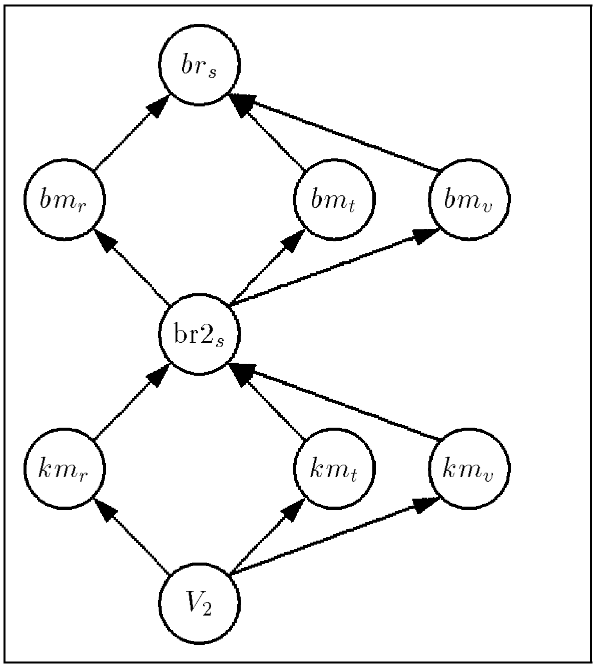

Figure 21: $\mathcal{P} = \{rstv\}$ where the leader is s. $br2$ denotes BlockReq2 and km denotes Ack. Process s broadcasts $br2_s$ the members concur via Ack messages (km_r, km_t, km_v). Process s then installs the new view V_2.

In Figure 21 we continue the scenario. A BlockReq2 message is denoted by $br2$; Ack messages are denoted by km. The leader, s, broadcasts $br2_s$. The members respond by messages km_r, km_t, and km_v. Process s then installs the new view V_2.

This protocol provides the same functionality as the symmetric protocol. When a member receives V_2, it has implicitly received all Acknowledge messages. Thus the acknowledgment round has been implemented.

6.5.2 Causal Order

When a view change occurs, out of order messages remain in the pending queue at the Causal layer (see Section 4.7.3). These messages should be erased before installing the new view. When the Causal layer receives an UpView it erases its out of order buffer and delivers the UpView. Thus, such messages are removed.

6.5.3 Message Ordering in the Face of Failures

When a view change from V_1 to V_2 occurs messages may remain blocked in the Safe and Agree layers because delivery conditions for them have not been met (see Section 4.7.4). These messages must be delivered in V_1, adhering to self-delivery, although delivery is performed with reduced semantics. Safe messages are delivered without acknowledgments from all processes in V_1; Agree messages are delivered despite holes in the total order.

When the Safe layer receives an UpView, it buffers the event and switches to best effort mode freeing all its blocked messages. It then delivers the UpView event to the upper layers.

The Agree layer first receives all messages previously blocked in the Safe layer; then the UpView event is received. The UpView is buffered and Agree switches to best-effort mode, freeing all blocked messages. When no more messages are blocked the UpView is delivered to the upper layers.

When the view change is received by the Top_appl no message remains blocked in the Safe and Agree layers.

6.5.4 Complete View of the Dag

In Lansis, the SVS layer must have a complete view of the dag. In Lansis/E we use the Ensemble solution of using the "no_total" label. All the Tsync messages are marked no_total. The Causal layer upon causal delivery of such a message m performs:

- Sends up the real message marked no_total. The message will not be delayed by any layer until it reaches the Tsync layer.
- Sends up a regular message, with null contents, to keep the dag complete. Thus, we avoid creating holes in the dag by recklessly bypassing the Safe and Agree layers. Null messages are discarded by default by the Appl_top layer.
- Using the existing Ensemble label no_total also allows interoperability with other total ordering layers (Sequencer, Total, and Totem). The Tsync' messages will bypass these layers as well.

6.5.5 EVS Message Ordering

When a view change occurs Agree and Safe messages are delivered with reduced semantics to the application. An application notification, the *tran-*

sitional view (see Section 4.7.6) should be delivered to the application at the point where Safe and Agree messages are delivered in a best effort mode.

In our implementation, when the Safe layer receives an UpView event it sends up a notification to the application declaring that all messages received from now on are delivered with reduced semantics. The transitional notification is delivered to the application at the point where Safe and Agree delivery is performed in best effort mode. Thus, we have implemented the transitional view.

7 Modifications to Ensemble

In order to use the Gmp we had to slightly change it and the Sync layer. In the regular Ensemble stack (see Section 5) when a view change occurs the Sync layer blocks the stack and the Gmp checks for stability before installing the new view. The Lansis/E stack is not compatible with this mechanism since it does not create DnStable (and UpStable) events, thus, Gmp cannot check stability of a Lansis/E stack.

We chose to modify the Gmp and Sync layers. We created a unified interface between the *synchronization* layers, Sync/Tsync, and the Gmp. Each of the synchronization layers accepts a DnBlockReq event, synchronizes the view, and delivers an UpBlockOk to the Gmp. The Sync layer uses the UpStable events and Block messages for synchronization. Tsync uses Block and Ack messages to achieve synchronization.

The Gmp requires reliable fifo and unreliable broadcast capabilities, which both Lansis/E and the regular stack provide. Each stack requires failure and view notifications, which the Gmp provides.

This simple modification allows the Gmp to operate with Lansis/E and the regular Ensemble stack.

The new Gmp does not check for stability but installs the new view immediately upon receiving an UpBlockOk from Sync. Since an UpBlockOk is delivered by the synchronization layer after stability has been established then it is safe to install the new view.

The new Sync performs synchronization via the following protocol:

1. Catch the DnBlockReq message.

2. Block the view.

3. When the UpStable events show StabVct = NumCast, UpSd(UpBlockOk).

These modification were later embedded in the current version of Ensemble due to other research efforts [BFH97].

8 Interoperability of Lansis/E and Ensemble

Embedding Lansis in Ensemble we intended to achieve interoperability between Lansis protocols and Ensemble protocols. Thus, the user could intermix Lansis/E and Ensemble layers to achieve the protocol combinations most suitable to its application.

The first step was the redesign and implementation of Lansis in the form of Lansis/E. This section describes the new possibilities for the configuration of stacks.

8.1 Old Stacks

The regular Ensemble stack is shown in Table 6; while the Lansis/E stack is shown in Table 7. We strove to reused as many layers Ensemble layers as possible in Lansis/E. We only created four new layers and used native Ensemble membership application and network interfaces. Only minor changes were made to these layers and modules.

Layer Name
Gmp
Sync
Top_appl
Sequencer
Stable
Mnak
Bottom

Table 6: Ensemble stack

8.2 New Stacks — Beyond Lansis/E and Ensemble

The user can now configure stacks that provide a great variety of reliability guarantees. In the following we describe two new stacks that are now conceivable.

For example, the stack in Table 8 causally orders all messages. It also totally orders Agree messages though it may break causal order between Agree messages and other messages. For example, if $am \leftarrow m$, then there is no guarantee on the relative order of delivery for m and am. Thus, messages are separated into two sets — the Agree messages and the rest.

Layer Name
Gmp
Tsync
Top_appl
Agree
Safe
Causal
Bottom

Table 7: Lansis/E stack

Ensemble	Lansis/E
Gmp	
	Tsync
Top_appl	
Sequencer	
	Causal
Bottom	

Table 8: Causal delivery with a non matching total ordering algorithm.

Agree messages are totally ordered and causally ordered relative to each other. The other messages are causally ordered. There are no guarantees as to relative ordering between the two sets.

It is possible to add the Safe layer above Causal adding support for the Safe message type. It is also possible to replace the Sequencer with any other total ordering protocol, such as totem (see Table 9).

Since Safe messages are in particular Agree messages they will be totally ordered by the Totem layer. As in the previous stack Safe messages will be causally + totally ordered amongst each other; other messages will be causally ordered relative to each other. There are no guarantees as to relative ordering between the two sets.

8.3 Stack Methodology

Here we define a framework into which all new stacks should fit. Writing new protocols and stacks in the way we shall describe will ensure their interoperability with all other Ensemble protocols.

We separate any stack $\mathcal{S}$ into three disjoint *suites*. These are, from top

Ensemble	Lansis/E
Gmp	
	Tsync
Top_appl	
Totem	
	Safe
	Causal
Bottom	

Table 9: Lansis/E stack with a different total ordering algorithm

to bottom:

Membership suite: Maintain the current membership. In Ensemble and Lansis/E this is performed by the Gmp.

Synchronization suite: Stabilize the group before a membership change. In Ensemble the Sync performs this; in Lansis/E it is performed by the Tsync layer.

Regular suite: These layers guarantee message reliability and ordering. In Ensemble this is performed by the Mnak/Stable/Sequencer layers; in Lansis/E by the Causal/Agree/Safe algorithms. Any regular suite must implement (at least) fifo reliable and unreliable broadcast.

An interface connects the membership and synchronization suites:

Synchronization-Membership interface: A membership suite provides failure and view notifications to the synchronization and regular suites. The synchronization+regular suites provide reliable fifo and unreliable broadcast capabilities to the membership suite. The synchronization suite also allows the membership to synchronize a view before a view change via a DnBlock event.

Separating the stack into these three disjoint suites enables any membership suite to work with any regular suite. Furthermore, if a different membership algorithm will be implemented then it will be able to use existing protocol layers. If a new regular stack will be written, it will provide it's own synchronization suite and work with existing membership algorithms.

8.4 Membership Server

An added benefit from this separation of the stack into three suites is the possibility to separate the membership layers from the rest of the stack.

We can create a server, running on each machine, that computes the membership. Each Ensemble stack on that machine will connect to it, and receive membership changes and failures from it. Since the membership layers are used only rarely and contain many of the complicated protocols, we can significantly reduce the code size of a regular stack without compromising performance.

This was later implemented and enhanced as part of the Maestro [BFH97] system, built around Ensemble, providing a high level interface to the application.

9 Conclusions

We rewrote Lansis code in the form of a set of Ensemble protocols, creating a new system named Lansis/E.

Lansis/E is:

- Layered: allowing a rigorous proof of the system using an ML verifier, as we plan to do for Ensemble.
- Flexible: allowing other, non-Lansis, protocols to be intermixed with it in one stack.
- Extensible: allowing new protocol to be added while retaining full interoperability.
- Portable: written in a machine independent programming language making porting it to new platforms straitforward.

Additionally, we enhanced Lansis to support a strong application programming model — Strong Virtual Synchrony. Lansis/E also contains enhancements to the existing infrastructure of Ensemble for debugging though this infrastructure is outside the scope of this work.

During the implementation phase it became clear that the current version of Lansis/E is not suited for optimal operation in a layered stack since the Causal layer attaches long acknowledgment vectors to each message. Thus, each message event carries a significant amount of information. This is expensive as well as not scalable. A layer stack (after automatic optimization) can run fast provided that each layer uses its local state, does not create much intra-stack communication, and does not add long headers to events.

Lansis contains about 5000 lines of C code. The Lansis layers in Lansis/E contain about a 2000 lines of OCAML code. We believe that this proves Ensemble as a sound general framework for implementing communication protocols. Furthermore, Ensemble is specifically suited for group communication protocols.

9.1 Safe Order

The original motivation for the Safe message type was database updates (transactions). In order to increase database availability it can be replicated onto multiple servers. Since databases are constantly updated and queried consistency must be kept between replicas. In order to keep all the copies consistent two phase [Ske82] or three phase [Gra78] commit protocols are commonly used. Typically in such systems a server proposes an update, the other servers store the update in a stable storage (disk), verify that the update is consistent with their local state, and acknowledge. Once the initiator receives an all ack it broadcasts Commit — instructing all members to apply the update to the database.

It is possible to replicated a database consistently using a Lansis group. Each database server can be implemented as a Lansis application maintaining a local copy of the database. Each update is sent by the servers as a Lansis Safe message. When the update is self-delivered to the sender it knows that all members received it. It then follows to update its local copy of the database. Naively using Safe messages is not sufficient since safe delivery implies receipt by all Lansis stacks. It does not imply receipt by the applications. To solve this problem applications need to specifically acknowledge received updates. When a sender receives an all-ack from all (live and connected) applications it can perform an update. If an application has to specifically acknowledge updates then sending updates as Safe Lansis messages is of no use.

We believe that Safe ordering should not be supported at the Lansis/E level at all, but at the application level, allowing the application to add the semantics it requires making Lansis/E lighter and faster.

Two papers have been published describing how database replication can be efficiently accomplished using a *replication layer* over Lansis. One of these works [ADMSM94] benefits from the implementation of safe ordering at the Lansis level. The other [Kei94] does not require this.

9.2 Future Work

We intend to pursue the unification of Transis and Ensemble in this framework. We have gone ahead with this and implemented our security architecture for GCSs [HR] in Ensemble. Lansis/E groups can now be secured from outside tampering via authenticating all group communication using a security layer.

We believe that Lansis/E should implement only total, causal and fifo ordering. The simplest, strongest, and most intuitive ordering guarantee is total order. Thus, we are currently studying low latency, high throughput, adaptive total ordering algorithms that will work on existing LANs as well as novel networks — ATM, Fast Ethernet, etc..

We intend the next version of Lansis/E to employ an underling causal ordering algorithm that does not add ack vectors to messages. Thus, it will not incur a performance penalty as in the previous implementation. In order to implement total ordering we intend to use an algorithm that assumes fifo reliable delivery. Thus, it will not require the ack vectors used by the Agree algorithm.

10 Appendix

10.1 Safe and Agree Ordering in Lansis

This subsection describes the original Lansis algorithm for Safe and Agree ordering. The original algorithm orders both message types. This Safe-Agree ordering algorithm is situated above the reliable causal ordering algorithm (Section 4.3). It is a superposition of the Lansis/E Agree and Safe ordering algorithms with a couple of modifications.

Assuming that process p in view V has dag_p which contains only Safe and Causal messages, the algorithm executes as Lansis/E Safe ordering. Assuming only Agree and Causal messages, the algorithm executes like Lansis/E Agree. The situation is more complicated if both Agree and Safe messages are in dag_p. Below we list the major scenarios and the ways the algorithm handles them.

Generally speaking Safe is stronger than Agree. Thus, Agree messages concurrent to a Safe message will always be delivered first.

In the following examples the universe $\mathcal{P}$ consists of four processes r, s, t, and v. In the scenario depicted in Figure 22 Agree message $am_{t,1}$ is concurrent to Safe message $sm_{s,1}$. The algorithm considers $sm_{s,1}$ as an Agree message in order to form an A-cut for $am_{t,1}$. Thus, the A-cut of $am_{t,1}$ is $\mathcal{A} = \{cm_{r,1}, sm_{s,1}, am_{t,1}, cm_{v,1}\}, \mathcal{L} = \{sm_{s,1}, am_{t,1}\}$. Note that this A-cut is not sufficient to deliver $sm_{s,1}$ since it does not qualify as an S-cut for $sm_{s,1}$.

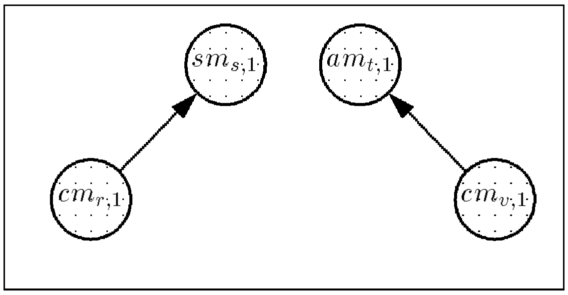

Figure 22: $\mathcal{P} = \{rstv\}$. dag_s — Agree message $am_{t,1}$ is concurrent with Safe message $sm_{s,1}$

In Figure 23 once process s receives $cm_{v,1}$, both $sm_{s,1}$ and $am_{t,1}$ become deliverable. Message $am_{t,1}$ is delivered followed by $sm_{s,1}$ in order to preserve the precedence of Agree over Safe.

When several concurrent Safe messages exist in dag_p process p waits until

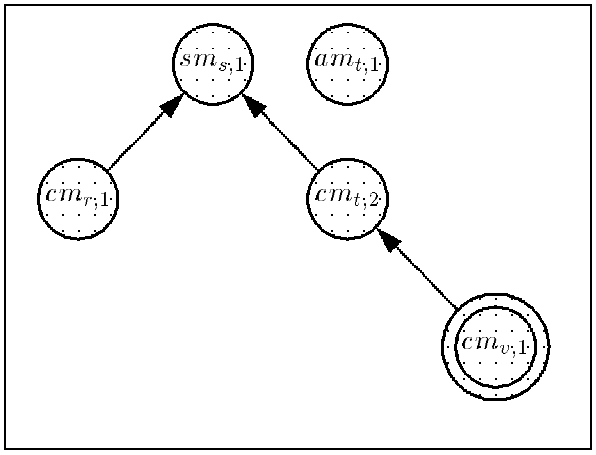

Figure 23: $\mathcal{P} = \{rstv\}$. dag_s — message $cm_{v,1}$ is missing. When it arrives both $sm_{s,1}$ and $am_{t,1}$ are deliverable.

S-cuts are formed for all messages before delivery in lexicographic order. This is done in order to guarantee uniform order of delivery throughout the view.

In Figure 24 an S-cut for $sm_{s,1}$ exists in dag_s. Still, s waits for the receipt of $cm_{r,2}$ and $cm_{s,2}$ in order to form an S-cut for $sm_{t,1}$. Then, s delivers $sm_{s,1}$ followed by $sm_{t,1}$.

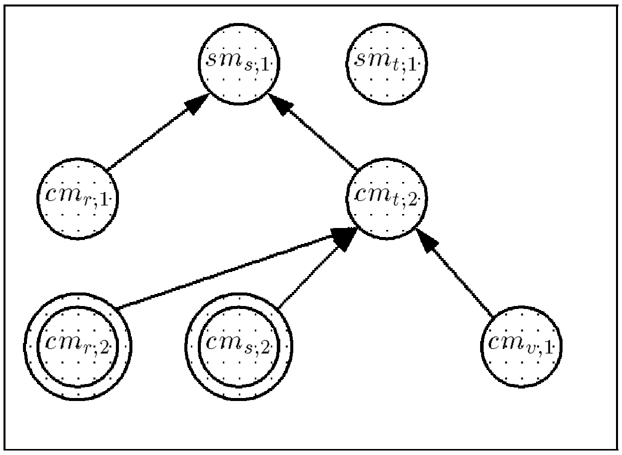

Figure 24: $\mathcal{P} = \{rstv\}$. dag_s — Safe messages $sm_{s,1}$ and $sm_{t,1}$ are concurrent. When messages $cm_{r,2}$ and $cm_{s,2}$ arrive both $sm_{s,1}$ and $sm_{r,1}$ become deliverable.

10.2 Notation Index

General Symbols	
$\mathcal{A}$	The Acknowledgment set of an A-cut.
$\mathcal{L}$	The Agree messages in an A-cut.
$\mathcal{S}_{sm}$	The set of messages in an S-cut.
$\mathcal{P}$	The universe of processes.
$m_{p,seq}$	the message m that process p sent, sequence number is seq.
m_p	Abbreviation, some message m from process p.
V_p	Process p's view.
$m_1 < m_2$	fifo relationship between messages.
$\leftarrow$	Causal relationship.
$\rightsquigarrow$	The reachability relation.
$\overset{*}{\rightsquigarrow}$	The transitive closure of reachability.
$\|\|$	Concurrency relationship
Message Types	
Causal	Causal message type.
Agree	Agree message type.
Safe	Safe message type.
Block	Block message type.
Acknowledge	Acknowledge message type.
Other	
Mem_p	Membership at process p.
$\text{Propose}_p(Q)$	Process p proposes Q.
M_V	The set of messages delivered in view V.

Receive/Send interfaces	
N_deliver$_p(m)$	p is delivered a messages m from the network.
N_send$_p(m)$	p sends message m on the network.
C_deliver$_p(m)$	causal delivery of message m at p.
C_send$_p(m)$	p causally sends message m.
A_deliver$_p(m)$	agreed delivery of message m at p.
A_send$_p(m)$	p agree sends message m.
S_deliver$_p(m)$	safe delivery of message m at p.
S_send$_p(m)$	p safe sends message m.
SVS_deliver$_p(m)$	virtual synchronous delivery of message m at p.
SVS_send$_p(m)$	p sends message m via the virtual synchronous module.
SVS_deliver$_p(V, m)$	p sends message m in view V.
SVS_send$_p(V, m)$	p is delivered message m in view V.
Ensemble	
UpRc	A layer receives a message from below.
DnRc	A layer receives a message from above.
UpSd	A layer sends a message up.
DnSd	A layer sends a message down.
Ensemble events	
DnStable	Stability messages created by the Stable layer.
UpStable	DnStable bounced by the Bottom layer.
DnBlockReq	Request for synchronization by the Gmp.
UpBlockOk	The Sync/Tsync replies, by this message, when view is stable.
DnBlock	In blocking protocol — flush the stack down.
UpBlock	In blocking protocol — bounce DnBlock upward.
DnBlockOk	In blocking protocol — bounce UpBlock downward.
Block	A message sent by a member to leader declaring blocked state.
UpFail	Failure event.
UpView	View event.
UpTransitional	Transitional notification
Ensemble labels	
no_total	Message label. this message should bypass total ordering.

References

[ADKM92] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A Communication Sub-System for High Availability. In *22nd FTCS conference*, pages 76–84, July 1992.

[ADMSM94] Y. Amir, D. Dolev, P. M. Melliar-Smith, and L. E. Moser. Robust and Efficient Replication using Group Communication. Technical Report CS94-20, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[AMMS+95] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. The Totem Single-Ring Ordering and Membership Protocol. In *13th ACM Transactions On Computer Systems*, 4, pages 311–342, November 1995.

[AMMSA93] Y. Amir, L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Fast Message Ordering and Membership Using a Logical Token-Passine Ring. In *13th International Conrerence on Distributed Computing Systems*, pages 551–560, May 1993.

[BDGB94] O. Babaoglu, R. Davoli, L.A. Giachini, and M. Baker. Relacs: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems. TR UBLC-94-15, University of Bologna, Jan 1994. homepage: www.cs.unibo.it/projects/relacs.html.

[BFH97] K. Birman, R. Friedman, and M. Hayden. The Maestro Group Manager: A Structuring Tool For Applications with Multiple Quality of Service Requirements. Technical Report TR97-1619, Cornell University, 1997. homepage: www.cs.cornell.edu/-Info/Projects/Ensemble/Maestro/Maestro.html.

[BJ87] K. Birman and T. Joseph. Exploiting Virtual Synchrony in Distributed Systems. In *11th ACM Symp. Operating Systems Principles*, pages 123–138, Nov 87. homepage: www.isis.com.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Trans. Comp. Syst.*, 9(3):272–314, 1991. TR 91-1192 of dept. comp. sci., Cornell Uni.

[BVR94] K. Birman and R. Van Renesse. *Reliable Distributed Computing with the Isis Toolkit.* IEEE Computer Society Press, 1994.

[CHTCB96] T.D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the Impossibility of Group Membership. In *10th ACM Symp. on Prin. of Database Systems (PODS)*, pages 322–330, May 1996.

[CM84] J. Chang and N.F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer systems*, 2(3):251–273, August 1984. homepage: www.atlantis.ivv.nasa.gov/-projects/RMP/RMP.html.

[CT91] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *10th ACM Symp. on Prin. of Database Systems (PODS)*, pages 325–340, 1991.

[DMS94] D. Dolev, D. Malki, and H. R. Strong. An Asynchronous Membership Protocol that Tolerates Partitions. Technical Report TR94-6, Institute of Computer Science, The Hebrew University of Jerusalem, 1994.

[FVR95] R. Friedman and R. Van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical Report TR95-1537, Cornell University, 1995.

[Gra78] J.N. Gray. Notes on Database Operating Systems. In *Operating Systems: An Advanced Course, Lecture Notes in Computer Science*, volume 60, pages 393–481. Springer-Verlag, Berlin, 1978.

[Hay96] M. Hayden. *The Ensemble Group Communication System*, 1996. homepage: www.cs.cornell.edu/Info/Projects/-Ensemble/index.html.

[HR] M. Hayden and O. Rodeh. Security Architecture for Group Communication. In preperation.

[HVR96] M. Hayden and R. Van Renesse. Optimizing Layered Communication Protocols. Technical Report TR96-1613, Cornell University, 1996.

[Kei94] I. Keidar. A Highly Available Paradigm for Consistent Object Replication. Master's thesis, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994. Also available as Technical Report CS95-5.

[KTHB89] M. F. Kaashoek, A. S. Tanenbaum, S. F. Hummel, and E. H. Bal. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(4):5–19, October 1989.

[Lam78] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 78.

[Ler97] Xavier Leroy. *The Objective Caml system release 1.05*, 1997. homepage: pauillac.inria.fr/ocaml.

[MADK94] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis Approach to High Availability Cluster Communication. TR CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, June 1994.

[MAMSA94] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended Virtual Synchrony. In *14th IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 56–65, June 1994.

[MES] R.A. Macedo, P.D. Ezhilchelvan, and S.K. Shrivastava. Newtop: A total order multicast protocol using causal blocks. TR NE1 7RU UL, Department of Computing Science, University of Newcastle.

[MFSW95] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Pheonix: A toolkit for building fault-tolerant, distributed applications in large scale. TR CH-1015, Ecole Polytechnic Federale de Lausanne, July 1995. homepage: lsewww.epfl.ch/projets/-phoenix/index.html.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Ske82] D. Skeen. A Quorum-Based Commit Protocol. In *6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 69–80, Feb. 1982.

[VR96] R. Van Renesse. Masking the Overhead of Protocol Layering. In *9th ACM SIGCOMM*, September 1996.

[VRCGS92] R. Van Renesse, R. Cooper, B. Glade, and P. Stephenson. A RISC Approach to Process Groups. In *Proceedings of the 5th ACM SIGOPS Workshop*, pages 21–23, September 1992.