Proceedings of the 28th Annual Hawaii International Conference on System Sciences — 1995

RELACS: A Communications Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems*

Özalp Babaoğlu Renzo Davoli Luigi-Alberto Giachini Mary Gray Baker

Department of Mathematics
University of Bologna
40127 Bologna (Italy)

Department of Computer Science
Stanford University
Stanford, California 94305 (USA)

Abstract

Distributed systems that span large geographic distances or manage large numbers of objects are already common place. In such systems, programming applications with even modest reliability requirements to run correctly and efficiently is a difficult task due to asynchrony and the possibility of complex failure scenarios. In this paper, we describe the architecture of the RELACS *communication subsystem that constitutes the microkernel of a layered approach to reliable computing in large-scale distributed systems.* RELACS *is designed to be highly portable and implements a very small number of abstractions and primitives that should be sufficient for building a variety of interesting higher-level paradigms.*

1 Introduction

Traditionally, global networks such as the Internet have been thought of exactly as that — networks. With recent gains in bandwidth and connectivity, these networks increasingly resemble the communication infrastructures of large-scale distributed systems. As such, it is tempting to deploy distributed reliable applications on them that permit higher levels of cooperation between geographically-distant sites than the traditional electronic mail exchanges or file transfers.

The principal impediment to exploiting the potential of large-scale distributed systems is the possibility of failures. In a system that spans large geographic distances, failures may result in complex scenarios with respect to communication patterns and network partitions. Furthermore, transient failures and unpredictable communication and computation delays make reasoning based on time and timeouts impractical. Developing and reasoning about applications to be deployed in wide-area distributed systems would be an extremely difficult task if all of the above complexities had to be confronted directly.

Over the last decade, process groups and group-based communication have emerged as appropriate technologies for reliable computing in traditional distributed systems [5]. Process groups were initially introduced by the V Kernel as a convenient structuring and naming mechanism [11]. Subsequently, the paradigm has been extended by the inclusion of multicast communication primitives with powerful consistency guarantees even in the presence of failures [7, 17, 1, 16]. Experience with these systems has confirmed the adequacy of process groups in greatly simplifying the construction of reliable distributed applications [6].

In this paper we examine the problem of designing communication infrastructures that enable reliable computing in distributed systems with dimensions considerably larger than previously considered. We believe that the process group approach remains a valid paradigm even in such large-scale distributed systems. To investigate this claim, we have implemented RELACS, a system explicitly designed to support group-based communication over wide-area networks. The system is based on off-the-shelf technologies for both communication (Internet UDP service) and computation (UNIX boxes). We describe the architecture of RELACS, the design issues we faced, and why we believe the system should scale efficiently to very large dimensions.

RELACS can be considered the microkernel of a layered architecture for the full suite of group mechanisms [4].[1] It implements a very small set of prim-

*This work has been supported in part by the Commission of European Communities under ESPRIT Programme Basic Research Project 6360 (BROADCAST), the National Research Council of Italy (CNR) and the Ministry of University, Research and Technology.

[1] RELACS corresponds to the *core* layer of the architecture described in [4].

itives that allow user applications to join, leave and multicast messages within groups. The consistency guarantees provided by RELACS are based on the notion of *view synchrony* [22, 21].[2] Informally, view synchrony cleanly transforms failures into group membership changes and provides global guarantees about the messages that have been delivered by a group as a function of changes to the group's composition. Higher-level services and abstractions, such as total-order and causal-ordered message delivery, uniformity, and atomic transactions, can be easily built on top of RELACS [20]. Being able to reason even with just view synchrony should greatly simplify application development. For example, in [3] Babaoğlu et al. describe how an interface very similar to RELACS can be used to manage replicated files in a large-scale system with one-copy serializability semantics.

A number of other systems have goals similar to those of RELACS. Historically, the Isis system [7] has been one of the most influential sources for ideas in applying group-based technology to reliable distributed computing. Our microkernel approach for structuring group mechanisms is shared by the more recent incarnation of Isis as Horus [19]. These systems, however, are still oriented towards local-area network environments and do not deal adequately with large scale. Transis [1] and Newtop [14] are perhaps the systems that are closest to RELACS with respect to large scale. Both systems, like RELACS, are able to deal with network partitions and mergers. Starting from a system model that is quite similar to RELACS, Newtop guarantees total-order message delivery within a highly-flexible group architecture. The Transis system model is composed of *broadcast domains* representing local-area networks that are in turn interconnected through point-to-point links. Our system model, on the other hand, is motivated by applications that result in uniformly distributed groups spanning large geographic distances. Thus the architecture of RELACS does not distinguish local-area segments, but rather, treats the system uniformly as a network of point-to-point links. Furthermore, RELACS and Transis differ with respect to the multicast primitives that they implement. Whereas RELACS provides only view synchrony and leaves ordering guarantees to higher layers, Transis offers the full suite of ordering semantics.

The next sections describe the assumptions made by RELACS about the underlying communication layer, the semantics of view synchrony in a large-scale system, and the architecture of RELACS in light of these assumptions and considerations. We conclude by describing the current state of the system and outlining directions for future work.

[2] In [21], the abstraction is called *virtual* synchrony. We are reluctant to use this term since it is loaded with other semantics, including causal- and total-order delivery that are associated with the Isis system.

2 The System Model

The system characteristics and services that RELACS builds upon are those typical of distributed systems. Abstractly, the system can be modeled as a collection of processes executing at potentially remote sites. Processes communicate through a message exchange service provided by the network. Informally, the consequences of large scale on the system are the following. The network is not fully connected and is typically quite sparse. Both processes and communication links may fail by crashing. Furthermore, the network may allow delivery of duplicate messages and it provides no message sequencing guarantees. Given that the computing and communication resources may be shared by large numbers of processes and messages, the load on the system will be highly variable and unpredictable. Thus, it is not possible to place bounds on communication delays or relative speeds of processes. As such, the system is adequately modeled as an asynchronous distributed system.

Asynchronous systems place fundamental limits on what can be achieved by distributed computations in the presence of failures [15, 5]. In particular, the inability of some process p to communicate with another process q cannot be attributed to its real cause — q may have crashed, q may be slow, communication to q may have been disconnected or it may be slow. From the point of view of p, all of these scenarios result in process q being *unreachable*.

What further distinguishes communication in the presence of failures in large-scale asynchronous distributed systems are the resulting properties of reachability. Formally, we can define reachability as follows: given two processes p and q, let $\rightsquigarrow$ be a binary relation such that $p \rightsquigarrow q$ if and only if q is *reachable* from p in the sense that if p were to send a message to q, q would eventually receive it. Note that as defined, reachability is a non-stable predicate on the evolving global state of the system. As such, in an asynchronous system, the reachability relation can never be known accurately but can only be approximated. The system service that is typically used for deriving approximations of reachability is called a *failure suspector* [10, 9]. Informally, failure suspectors generate suspicions of failures by relying on timeouts to detect missing responses to either application-generated messages or forced messages from periodic "pings". The resulting information can only be classified as suspicions since timeouts in an

asynchronous system can never be set perfectly. Furthermore, information that is obtained through communication can only reflect some past state of the system due to message delays. By deriving it directly from suspicions (processes that are suspected are declared unreachable while all others are reachable), we obtain approximations for reachability.

In an asynchronous system, no matter what mechanism is used, conclusions regarding reachability derived by individual processes can never be totally accurate and may be mutually inconsistent. Furthermore, in a large-scale system, communication delays could be comparable to inter-failure times. This may result in significant periods during which symmetry and transitivity of the reachability relation are not satisfied due to inconsistencies either among the failure suspectors or the network routing tables. Despite these possibilities, we assume that the communication layer we are building upon satisfies the following properties for the reachability relation (the symbols $\Box$ and $\Diamond$ denote the temporal operators "always" and "eventually", respectively):

- *Eventual symmetry.* If process q is reachable from process p and there are no new failures for a sufficiently long time, then process p will be eventually reachable from process q:

$$\Box(p \rightsquigarrow q) \Rightarrow \Diamond(q \rightsquigarrow p).$$

- *Eventual transitivity.* If process q is reachable from process p, process r is reachable from process q and there are no new failures for a sufficiently long time, then process r will be eventually reachable from process p:

$$\Box(p \rightsquigarrow q) \wedge \Box(q \rightsquigarrow r) \Rightarrow \Diamond(p \rightsquigarrow r).$$

Achieving these properties requires two conditions. First, the failure-free communication structure must be fully connected. Second, the failure suspectors must be constructed such that if the interval between failures is sufficiently long, then a process that has not crashed and remains connected should eventually not be suspected.

Note that the above properties do not exclude the possibility of *network partitions* [18]. It may be that the set of processes is partitioned into several disjoint islands that are mutually unreachable. In addition to these so-called *clean* partitions, periods during which symmetry or transitivity are not satisfied may lead to more complex scenarios with partitions that have non-empty intersections. In RELACS we are able to cope even with these cases as discussed in the next section.

3 View Synchrony

The basic abstractions of view synchrony are *process groups*, *views* and *messages*. A *group* is a named set of processes that can be treated as a single unit from the outside. Processes may join a group by naming it or may leave the group they are currently in. Once a member of the group, a process may communicate with the other group members through multicasts of messages. View synchrony guarantees that the delivery of these messages are totally ordered with respect to changes in the group's membership.

At the level of RELACS, our design does not allow groups to overlap in membership. As discussed in [4], this restriction is not limiting in that overlapping group structures may be permitted by higher levels of the architecture that do not require view synchrony. Given that a process is a member of at most one group, we will omit the group name from our notation for simplicity. Furthermore, issues related to the interaction between a group and processes external to it are beyond the scope of this paper [4].

3.1 Views and View Changes

At any given time, each process in the group has its own perception of which other known group members are reachable. For each process p, this perceived reachability set is called its *view* of the group, and denoted V_p. Views are assigned unique names such that they can be distinguished even if their composition is identical.[3] View synchrony tracks relevant system events and transforms them into *view changes* that are delivered to processes for installation. View changes are triggered by process crashes and recoveries, communication failures and repairs, network partitions and mergers, or explicit requests to join or leave the group.

At each process, the set of installed views forms a sequence, with the successor view relation defined as follows:

Definition 3.1 *View V^j is called the* successor *of view V^i, denoted $V^i \prec V^j$, if and only if there exists some process p at which V^j is installed immediately after V^i. Let $\overset{*}{\prec}$ denote the transitive closure of this successor view relation.*

The last view in this sequence at process p is called p's *current view*. Two processes p and q may share the same current view, which we denote as $V_{p,q}$.

[3] For notational simplicity, we use the view name also when referring to the set of its members.

One of the problems in asynchronous systems is that the reachability sets perceived by individual processes may be mutually inconsistent and inaccurate with respect to the actual system state. View synchrony renders the reachability information encoded within views nevertheless useful. In particular, it guarantees that views installed at individual processes (i) have some relation to the actual state of the system with respect to failures, and (ii) are mutually consistent. We can formalize these ideas through the following definition:

Definition 3.2 View installation.

1. *If no new failures occur for a sufficiently long time, the current view V_p of each process p that has not crashed is such that*

   (a) $\Box(p \rightsquigarrow q) \Rightarrow \Diamond(q \in V_p)$

   (b) $\Box(p \not\rightsquigarrow r) \Rightarrow \Diamond(r \notin V_p)$

   (c) $\Box(s \in V_p) \Rightarrow \Diamond(V_p = V_s)$.

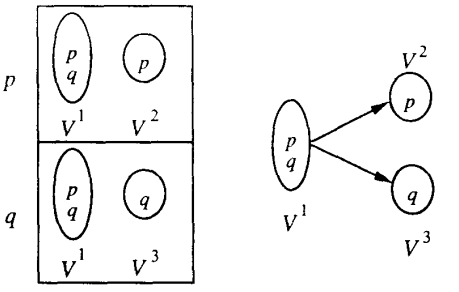
2. *For any pair of installed views V^1 and V^2*

$$\neg((V^1 \overset{*}{\prec} V^2) \wedge (V^2 \overset{*}{\prec} V^1)).$$

In other words, view synchrony guarantees that installed views are closed under reachability and maximally shared. Furthermore, views are installed so as to preserve the partial order structure defined by the $\overset{*}{\prec}$ relation on the global set of views. Note that the maximal sharing property of views would be sufficient to guarantee that partitions result in non-overlapping concurrent views. The property, however, can only be guaranteed only during sufficiently long periods without failures. Thus, there may be transient periods where the property does not hold, resulting in overlapping views. In RELACS, view installations are not performed as atomic actions since the resulting cost in a large-scale system would be prohibitive. Thus, failures that occur during the view agreement phase may cause a process to install a view that is different from the one it initially agreed upon. The consequences of this possibility are discussed further below.

Another problem in large-scale distributed systems is the possibility of network partitions. In terms of view synchrony, network partitions result in *concurrent* views, which are views unrelated through $\prec$. As a simple example, consider the scenario depicted in Figure 1 where two processes p and q initially belong to and share the view $V^1_{p,q}$. Due to a partition, they become mutually unreachable and install two different successor views V^2_p and V^3_q.

More complex failure scenarios, such as the one depicted in Figure 2, may cause a given view to partition



Local evolution of views Global evolution of views

Figure 1: Partitioned processes install distinct views.

into multiple concurrent views that overlap. Initially, four processes p, q, r and s all belong to and share the same view V^1. Process q is partitioned from the rest, provoking a view change. The successor view of V^1 for q is V^2 which includes only q. The remaining processes agree on a new reachable set (p, r, s), and processes r and s install this as their new view V^3. However, before having a chance to install this view itself, p is partitioned from the others, resulting in p installing V^4, which includes only p, as the successor view of V^1. According to their views, r and s believe p to be reachable, but p's view indicates that r and s are not. The situation may seem bothersome, in that p appears to participate in two views simultaneously. But in fact, at any given time, p has a unique current view; it simply appears in some other view (V^3) that it does not share. Furthermore, Property 1(b) of view installation ensures that overlapping concurrent views cannot persist, since there will always be future views that exclude the overlapping elements. In the example, if r and s try to communicate with p, they will realize that p is in fact unreachable and will trigger a new view change to exclude it, leading to V^5.

Partition mergers result in view changes in which several processes with distinct current views all install a common successor view. In Figure 2, if processes p and q, which were partitioned with current views V^4 and V^2, once again become mutually reachable, they will both install view V^6 including themselves. RELACS does not specify any special action when view mergers occur. Appropriate handling of these situations is application-specific and is left to higher layers which may implement primitives such as state transfer [7].

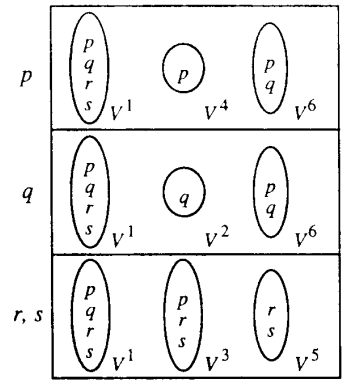


Figure 2: Failures during view agreement may result in overlapping concurrent views.

3.2 Message Delivery

In addition to managing views, RELACS implements a reliable multicast primitive for communication among group members. Informally, reliable multicast within a group ensures that the message is received by all or none of the group members. In defining the exact semantics of reliable multicast, we distinguish between a process *receiving* and *delivering* a message. Whereas the first primitive is provided by the underlying network transport services, delivery is implemented by RELACS by invoking an application-specified handler routine. A message m is said to be delivered by process p *during* view V_p if V_p is the current view at p when the handler is invoked with message m.

The real power of view synchrony is not in its individual components — view changes and reliable multicasts — but in their integration. Informally, view synchrony permits a process to reason globally about the set of messages other processes have delivered based on local information maintained as the sequence of installed views. In particular, view synchrony guarantees that for each path in the partial order of views, message deliveries are totally ordered with respect to view changes. Thus, during the phase when some view V is being terminated and the successor view V' is being established, processes must agree not only on the composition of V' but also on the set of messages that need to be delivered during V.

Ultimately, the semantics of view synchrony should allow an application process to reach useful conclusions regarding the set of messages delivered by other processes during a given view. As it turns out, formalizing the above ideas in a manner that can be implemented leads to decisions about what the agreed-upon set of messages should be, and to refinements of view membership semantics.

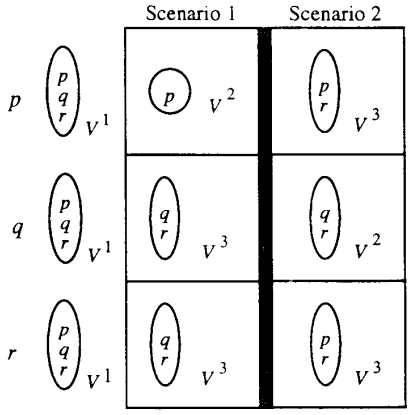


Figure 3: Scenarios that illustrate choices between possible message delivery semantics.

One decision concerning the message set agreement arises due to network partitions. As illustrated in Scenario 1 of Figure 3, suppose three processes p, q, and r initially belong to and share the common view V^1. Process p becomes partitioned from q and r, provoking a view change. Processes q and r construct a new view V^3. To do so, they agree both on the new view *and* the set of messages that must be delivered during V^1. Thus, once they have installed V^3, they may be sure of the messages delivered by each other. In deciding upon the set of messages, however, two semantics are possible. Processes q and r may either

agree on the set of messages that *have already been* delivered during V^1, or that *should be* delivered during V^1. The former semantics allows q and r to know for certain that they have delivered exactly the same set of messages. Achieving such a guarantee, however, requires additional communication (essentially, the second phase of a 2-phase commit protocol) during the agreement phase. The latter semantics guarantees only that q and r will deliver the same set of messages *provided that they survive the view change*. In RELACS we currently adopt this weaker semantics since incurring the additional cost for each view change may not be practical in a large-scale system.

A second issue arises from the fact that concurrent views may intersect in arbitrary ways. As an example consider the second scenario presented in Figure 3. As before, process p becomes partitioned from q and r, provoking a view change. Process p does not realize that the partition has occurred, and continues executing and delivering a set of messages N. Meanwhile, q and r initiate a view change to exclude p and agree to deliver a set of messages M while constructing V^2. Process q installs V^2 as the successor to V^1. Yet another partition between q and r causes r to abandon view V^2 and initiate a new view change. While doing so, p and r become mutually reachable while q remains partitioned. So when constructing V^3 together, p and r agree to deliver a set of messages that is the union of M (those known to r) and N (those known to p). Process q, on the other hand, delivers the set of messages M and installs view V^2 as the successor to V^1.

The guarantees provided by view synchrony in this situation are unsatisfactory for two reasons. First, process p is forced to deliver a set of messages (M) that are potentially from a process (e.g., q) extraneous to its successor view V^3. Second, process q can only know about the message set M delivered by r while r actually delivers more messages ($M \cup N$). In other words, a process is constrained to reason about another process in its view having delivered *at least* a set of messages as opposed to *exactly* the set of messages. In particular, views V^2 and V^3 have r as a common member but they include other disjoint members (p and q). The common member acts as a bridge between the two views and causes undesirable message deliveries with respect to the disjoint members.

To handle this problem we impose the following restrictions on view evolutions to define a membership semantics that is intermediate between weak- and strong-partial [21]:

Property 3.1 Quasi strong-partial membership service.

1. *(Partitioning Rule:) If V^1, V^2 and V^3 are three views such that*

$$(V^1 \prec V^2) \wedge (V^1 \prec V^3) \wedge (V^2 \neq V^3),$$

then $(V^2 \subset V^3) \vee (V^3 \subset V^2) \vee (V^2 \cap V^3 = \emptyset)$.

2. *(Merging Rule:) If V^1, V^2 and V^3 are three views such that*

$$(V^1 \prec V^3) \wedge (V^2 \prec V^3) \wedge (V^1 \neq V^2),$$

then $(V^1 \cap V^2 = \emptyset)$.

In other words, two concurrent views that result from the partitioning of a common view may overlap only if one is a proper subset of the other; two concurrent views that merge to form a single common view cannot intersect. RELACS guarantees these properties for view evolutions by not allowing unreachable processes to become reachable during view termination. The consequence of this restriction is a potential increase in the number of view changes. The benefit, on the other hand, is a stronger guarantee.

In light of these considerations, we can finally define view synchronous communication as implemented by RELACS.

Definition 3.3
View-synchronous communication. *For each multicast message m, if there exists some process p that delivers m during view V^i, then for all views V^j such that $V^i \prec V^j$ and $p \in V^j$, all processes $q \in V^i \cap V^j$ that have not crashed also deliver m. Furthermore, if a message is delivered, it is delivered in exactly one view that must include the sending process as a member.*

Note that there is a subtle but important difference between the above definition and the one given by Schiper and Ricciardi in [21]. The Schiper-Ricciardi definition requires all processes in $V^i \cap V^j$ to deliver the same set of messages *that were multicast in view* V^i. Our definition does not mention the view in which the message was multicast. All we require is that if some process p delivers message m in view V^i, then all processes that survive together with p into the same next view also deliver m. In our case, the message may have been multicast in view V^i or some *earlier* view since the notion of "multicasting in a view" as we have defined it, is with respect to the process local state (the last view installed before issuing the multicast). The alternative definition is made with respect to the global system state in which the multicast actually occurs. Unfortunately, asynchrony between applications and the support layer makes it impossible for a process to know in which (future) view its multicast request will be serviced.

Finally, note that view synchrony as implemented in RELACS guarantees nothing about the relative order of messages delivered during a given view. Applications that require ordering guarantees such as uniform [20], causal [8] or total [13] will have to rely on layers built on top of RELACS.

4 The Application Interface

Applications that require RELACS services invoke them through a small set of library functions. The proposed interface is an attempt to maximize flexibility while minimizing complexity. The following is an informal description of the RELACS application programming interface.

`v_init()` global system initialization and data structure allocation.

`v_join(g_name, <handlers>)` join the group identified through the string `g_name`. Attempts to join multiple groups or the same group more than once generate an error. If the group to be joined does not exist or has no members in the current partition, it will be created. For groups representing a unique global service, ensuring that a single instance is created despite partitions, requires that the application consult a global naming service in constructing the (globally unique) group name. Upon partition mergers, processes belonging to groups with matching names will be automatically joined. The call includes optional parameters for associating application-specified handlers with view change, message delivery and state transfer events. Each handler invocation is indivisible with respect to others in the sense that future events do not preempt an active handler. Thus, handler executions are serialized as defined by view synchrony semantics.

`v_leave(type)` leave the group. Since a process can belong to at most one group, the group name is implicit. The parameter type selects if the leave is to be *immediate* or *delayed*. In the former case, the call returns immediately and the leave is treated as if the process crashed. In the latter case, the call returns only after a new view has been established in which the exiting process is marked as such. Thus, the exiting process has the same view synchrony guarantees as those remaining in the view.

`v_cast(msg)` multicast message `msg` to the current group with view-synchronous semantics.

`v_msend(dest_list, msg)` multisend message `msg` to the list of processes in `dest_list`. The call is equivalent to a sequence of point-to-point send operations, one for each destination, with best-effort delivery guarantees. Reception is performed through the same message delivery handler mechanism as multicasts. The destination process addresses can be extracted from the information contained in the view. This call is extraneous to the view synchrony semantics and is included to facilitate group interactions with non-member processes.

Note that RELACS itself does not include a threads (light-weight processes) package. If the underlying system supports such a mechanism, then application programs could be structured so as to associate each event handling within a separate thread. In this case, the handler terminates with the thread invocation. Consequently, event management enclosed within a thread cannot rely on view synchrony.

5 System Architecture

This section briefly describes the current implementation of the RELACS system. Further details can be found in [2]. Whereas the RELACS model treats all processes uniformly, for performance reasons, the implementation distinguishes between processes local to a given site and those that are remote. Architectural design choices are based on the following assumptions about the system model:

- Communication between processes local to a single site is reliable.
- Crashes of local processes can be reliably detected rather than simply suspected.
- Sites, like processes, may fail by crashing. A site crash is equivalent to the crash of all processes local to that site.
- While the number of processes that belong to groups may be large, the number of sites they span is typically small.

The first two assumptions simplify the management of reachability information within a given site since a local process is either reachable or has crashed, but can never be unreachable while operational.

Each site runs a single instance of the RELACS server. Processes requiring RELACS services link appropriate library functions and invoke them by sending requests to the local server. Figure 4 illustrates this organization. It is the servers that actually perform reachability management, multicasts, message deliveries, view changes, and communication with other remote sites. For example, a request to multicast a message is handled as follows: the process issues the request by invoking the function `v_cast()`. The RELACS server local to the site receives the request and delivers the message to all other local group members. The

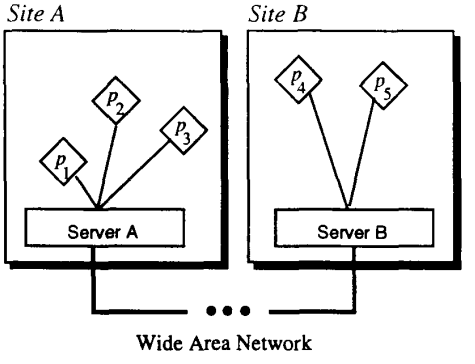


Figure 4: Processes do not communicate directly but request RELACS services by contacting the local server.

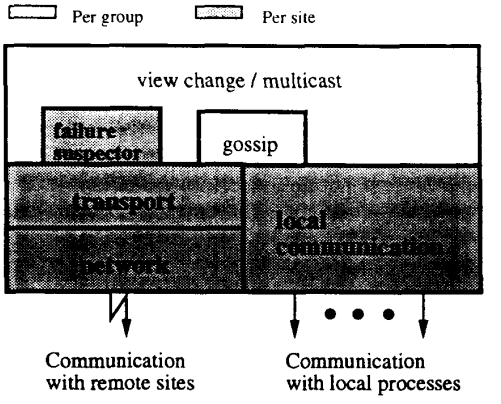


Figure 5: The architecture of the RELACS server.

message is also sent to all other RELACS servers on remote sites that have at least one member of the group. Each server delivers the message to its local processes. Delivering the message to a process triggers the process' message delivery handler function.

The RELACS server has a layered structure with the following logical components:

- *network*: Acts as the interface to the low-level (unreliable) network services for communicating with remote sites.
- *transport*: Implements reliable point-to-point communication on top of the underlying network services.
- *failure suspector*: Uses periodic "ping" messages and information passed up from the transport layer in order to construct reachability information about remote sites.
- *local communication*: Handles process/server communication internal to a single site.
- *view change/multicast*: Handles view changes, multicasts and delivery of messages.
- *gossip service*: Constructs reachability information regarding local processes and propagates it to remote sites.

Figure 5 shows the organization of the different layers. Adjacent layers may exchange information. Since a single server may handle several different groups that have processes running on the site, the upper layers of the server are instantiated once for each group. The lower layers are shared by all groups at the site and are instantiated only once.

The above system architecture reflects our emphasis on sites. In as much as possible, we strive to achieve costs that are proportional to the number of sites rather than the number of processes. In particular, reachability information derived from the failure suspectors is with respect to sites. This information is mapped to reachability of individual processes when needed by combining it with the data received from the gossip services.

Failure suspectors for large-scale systems with highly variable communication delays must be designed with care. The difficulty is in striking a balance between responsiveness of view changes to actual failures and overhead due to false suspicions. The RELACS failure suspector is highly adaptive in that the timeout periods are established individually for each of the communication channels based on the mean and variance of observed delay during a window of recent communication.

6 Status and Conclusions

We have implemented a prototype of RELACS on top of SunOS 4.3 (BSD UNIX) using the Internet UDP datagram service through the socket interface. We have also built a demonstration program to illustrate major system events using Tcl/Tk [23] for the graphical interface. We have found the tool useful not only for observing application behavior, but also for performance tuning and debugging of RELACS itself.

Figure 6 illustrates a snapshot of the demonstration

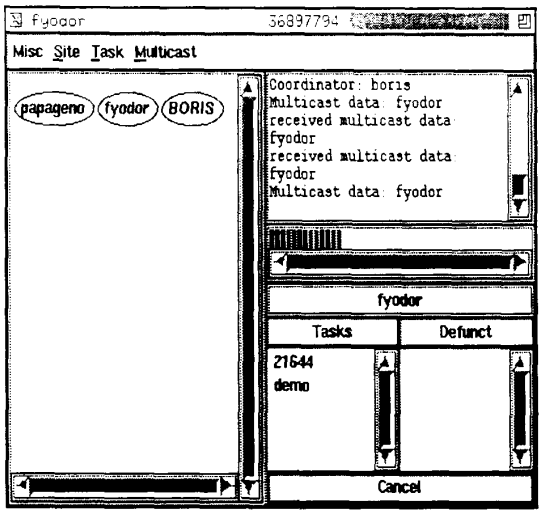


Figure 6: The graphical presentation of the RELACS demonstration program.

program. Three sites have processes in the demonstration group: *papageno*, *fyodor*, and *boris* with the name of the current coordinator for view agreements (see [2] for details) in capital letters. The upper right-hand window traces view changes and message multicasts/deliveries. Below this window, the same events are shown as a bar graph with black bars representing view changes and grey bars representing message deliveries. The lower right-hand window shows the processes active at a given server (here, *fyodor*) as well as those having executed a delayed leave (denoted as *Defunct*).

The current implementation has several limitations both in functionality and performance. We assume that communication satisfies the eventual symmetry and transitivity properties for reachability. Without these, sites may have conflicting opinions about reachability, and thus engage in a never-ending sequence of view agreement steps. In a wide-area network such as the Internet, there can be extended periods where transitivity is not satisfied. Ideally, the problem should be solved by modifying the current IP routing algorithms or inserting a layer on top of the UDP service that can re-route packets through alternative paths. In this manner, even if the underlying infrastructure does not guarantee transitivity, we may still achieve the property by forwarding messages between connected sites. Currently we make use of neither hardware broadcast capabilities of local-area segments nor the IP multicasting facility over wide-area links of the network [12].

We assume that interprocess communication within a single site is completely reliable. While this is usually a reasonable assumption, there may be rare situations of extreme local resource shortages where it does not hold. Similarly, on rare occasions (e.g., no more available inodes), our mechanism to detect local process crashes through file locks may prove unreliable [2].

In the current implementation there are no provisions for flow control. We assume that the RELACS server has unlimited buffering capability. Processes performing multicasts at a fast rate may overload the message buffers at a site with a slow process. One possible solution to this problem is to force the slow process to leave the group so that it may bring itself up to date through state transfer when rejoining the group without having to process the back logged messages.

The current prototype is in too early a stage to draw any conclusions regarding its performance or completeness. When the above shortcomings have been addressed and the performance enhanced, we will engage in an evaluation study of the architecture. The raw performance of multicast communication and view management has to be quantified. We also need to construct several realistic applications in order to evaluate the ease of use and completeness of the RELACS programming model. As distributed systems continue to grow in geographical scale, tools such as RELACS will become indispensable for building reliable applications to run on them.

Acknowledgements

We are grateful to Ian Jacobs for the extensive editorial help he provided during the preparation of this document. Niels Nes programmed the graphical interface, the demonstration program and contributed to the implementation of the server. The overall design of RELACS benefited from extensive discussions with our colleagues André Schiper, Uwe Wilhelm, Cristoph Malloth (Ecole Polytechnique Fédérale de Lausanne), Paulo Veríssimo and Luis Rodriguez (INESC, Lisbon).

References

[1] Y. Amir, D. Dolev, S. Kramer and D. Malki. Transis: A Communication Sub-System for High Availability. In *Proc. 22nd Annual International Symposium on Fault-Tolerant Computing*, pages 76–84, July 1992.

[2] Ö. Babaoğlu, M.G. Baker, R. Davoli, and L.A. Giachini. RELACS: A Communications Infrastruc-

ture for Constructing Reliable Applications in Large-Scale Distributed Systems. Technical Report UBLCS-94-15, Laboratory for Computer Science, University of Bologna, Italy, June 1994.

[3] Ö. Babaoğlu, A. Bartoli, and G. Dini. Replicated File Management in Large-Scale Distributed Systems. In *Proc. 8th Int. Workshop on Distributed Algorithms*, October 1994.

[4] Ö. Babaoğlu and A. Schiper. On Group Communication in Large-Scale Distributed Systems. In *Proc. ACM SIGOPS European Workshop*, Dagstuhl, Germany, September 1994.

[5] K. Birman, The Process Group Approach to Reliable Distributed Computing, *Communication of the ACM*, 9(12):36–53, December 1993.

[6] K. Birman and R. Cooper. The ISIS Project: Real Experience with a Fault-Tolerant Programming System. *ACM SIGOPS Operating Systems Review*, 25(2):103–107, April 1991.

[7] K. Birman, R. Cooper, T. Joseph, K. Marzullo, M. Makpangou, K. Kane, F. Schmuck and M. Wood. The ISIS System Manual, Version 2.1. Department of Computer Science, Cornell University, Ithaca, New York, September 1990.

[8] K. Birman, A. Schiper and P. Stephenson. Lightweight Causal and Atomic Multicast. *ACM Trans. Computer Systems*, 9(3):272–314, August 1991.

[9] T.D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *Proceedings 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158. ACM, August 1992.

[10] T.D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proceedings 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340. ACM, August 1991.

[11] D.R. Cheriton and W. Zwaenepoel. Distributed Process Groups in the V Kernel. *ACM Trans. Computer Systems*. 3(2):77–107, May 1985.

[12] S. Deering. Host Extensions for IP Multicasting. RFC1112, August 1989.

[13] D. Dolev, S. Kramer and D. Malki. Early Delivery Totally Ordered Multicast in Asynchronous Environments. In *Proc. 23nd Annual International Symposium on Fault-Tolerant Computing*, pages 544–553, June 1993.

[14] P.E. Ezhilchelvan, R.A. Macedo and S.K. Shrivastava. Newtop: A Fault-Tolerant Group Communication Protocol. Technical Report, Computer Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, United Kingdom, August 1994.

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

[16] M.F. Kaashoek and A.S. Tanenbaum. Group communication in the amoeba distributed operating system. In *Proceedings of the Eleventh International Conference on Distributed Computer Systems*, pages 222–230, Arlington, Texas, May 1991. IEEE Computer Society.

[17] L.L. Peterson, N.C. Bucholz, and R.D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, August 1989.

[18] A. Ricciardi, A. Schiper and K. Birman, Understanding Partitions and the "No Partition" Assumption. In *Proc. 4th IEEE Workshop on Future Trends of Distributed Systems*, Lisboa, September 1993.

[19] R. van Renesse, K. Birman, R. Cooper, B. Glade and P. Stephenson. The Horus System. In *Reliable Distributed Computing with the Isis Toolkit*, K.P. Birman, R. van Renesse (Ed.), IEEE Computer Society Press, Los Alamitos, CA, pages 133–147, 1993.

[20] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. 13th International Conference on Distributed Computing Systems*, pages 501–568, May 1993.

[21] A. Schiper and A. Ricciardi. Virtually-Synchronous Communication Based on a Weak Failure Suspector. In *Proc. 23rd International Symposium on Fault-Tolerant Computing*, Toulouse, pages 534–543, June 1993.

[22] K. P. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings Eleventh Symposium on Operating System Principles*, pages 123–138, November 1987.

[23] J.K. Ousterhout. *Tcl and the Tk Toolkit*, Addison-Wesley, 1994.