A Replication-Transparent Remote Invocation Protocol[1]

Luís Rodrigues
ler@inesc.pt
IST-INESC[3]

Ellen Siegel[2]
siegel@parc.xerox.com
ERCIM-INESC

Paulo Veríssimo
paulov@inesc.pt
IST-INESC

October 3, 1994

[1]Selected sections of this report will be published in the Proceedings of the 13th Symposium On Reliable Distributed Systems, Oct. 25-27, 1994, Dana Point, California, © 1994 IEEE

[2]Currently at Xerox PARC.

[3]Instituto de Engenharia de Sistemas e Computadores, R. Alves Redol, 9 - 6º - 1000 Lisboa - Portugal. This work was partially supported by the CEC, through Esprit Project BR 6360 (Broadcast) and by the ERCIM fellowship programme.

Abstract

Although many algorithms and implementations of replicated services have been developed, most have embedded aspects of the replication management in the invocation protocol. This makes it extremely difficult to modify the replication protocol without changing the protocol used by the clients, and causes an undesirable violation of both transparency and modularity. Our protocol supports the fault-tolerant remote invocation of replicated services, providing not only the usual location transparency but also transparency of replication semantics. It is designed as a collection of modular services which can be configured according to the needs of the application.

Our approach is independent of the details of the replica control protocol used to maintain the consistency of server replicas. We use a lightweight remote invocation protocol in order to minimize the impact on the client of issues such as scale and replication consistency maintenance. Distributed retry detection can be optionally invoked on a per-message basis with three different levels of reliability; even in this case, our communication extensions allow the client to remain decoupled from the stronger communication requirements of the inter-replica protocols at the server. Furthermore, unlike most previous systems we provide explicit support for weakly consistent replication protocols.

Contents

Chapter 1

Introduction

The goal of providing effective programming support for distributed applications has been a research issue for more than two decades. Although often limited in earlier systems, support for fault-tolerance and replication has been subject to a growing interest in the last few years. Several approaches have been proposed, including many in the area of remote invocation [7,9,8]. However, most existing systems provide support for at most a limited set of pre-defined replication strategies. Approaches like problem-oriented shared memory [6,16,11] which allow different replication strategies to be applied to different objects are a key factor for efficiency in distributed platforms.

The choice of a replication algorithm for a particular replicated service can be based on many constraints, most of which depend on some combination of the service semantics and environment. The inevitable evolution of system components or service requirements is thus likely to affect the choice of replication algorithm, and makes the availability of a wide range of protocol options particularly advantageous.

Unfortunately, in most systems a replicated service can be accessed remotely only via protocols tailored to its specific replication protocol, causing an undesirable violation of both transparency and modularity; many even implicitly consider clients to be members of the replicas' communication group. This makes it extremely difficult to re-implement the service using a different replication technique without changing the protocol used by the clients. In addition, the synchronization costs associated with such tightly coupled systems make it virtually impossible to adequately address issues inherent to large scale. Thus, embedding knowledge of the protocol in all service clients is in most cases an unnecessary and undesirable violation of transparency, modularity, and flexibility.

In order to maintain the transparency of an object replication scheme, we propose a fault-tolerant Generic Remote Invocation Protocol (GRIP) which is independent of the service replication protocol and which places no constraints on the replica consistency model. Clients use a lightweight remote access protocol which allows them to remain independent of the details of the replication protocol and of the resulting inter-replica synchronization. The implementation of the replication management protocol itself remains almost completely unaffected: service replicas communicate among themselves using their private replication protocol, and only limited modifications are required in order to take advantage of the GRIP functionality. Distributed retry detection can be optionally invoked on a per-message basis with three different levels of reliability; even in this case, our communication extensions allow the client to remain decoupled from the stronger communication requirements of the inter-replica protocols at the server. Additionally, our design provides explicit support for the implementation of weakly consistent replication schemes. GRIP is thus designed as a collection of modular services which can be configured according to the needs of the application.

The paper is organized as follows: the design goals and the global GRIP system model are described in Chapter 2. Chapter 3 describes the underlying communication service and our extensions for efficiency and to support weak replication. Chapters 4 and 5 describe the interface and protocols of the main service modules, and Chapter 6 illustrates how GRIP can be configured and used to support different replication protocols and discusses some of the implications of your approach. Chapter 7 relates our model to existing work, and concluding remarks appear in Chapter 8.

Chapter 2

The GRIP Model

The following sections introduce the GRIP model. We first present our basic system model and assumptions, then provide a brief design overview.

2.1 System Model and Assumptions

We base our protocols on a model of distributed programming with active entities (*processes*) which communicate exclusively via message passing. Processes execute on (possibly heterogeneous) nodes which are in turn connected by a network. Our model of interaction is one of *client-server* (see Figure 2.1), where some processes (*servers*) provide services to other processes (*clients*). A client interacts with a server using a *remote invocation* protocol, implemented by a pair of client and server communication *stub*s. The client stub hides distribution and possibly replication from the client application by acting as a local representative of the remote service, marshaling the request in a message and forwarding it to the server stub(s). When a reply is expected, the client stub waits for a reply from the server stub(s), unmarshals it, and returns the results to the client.

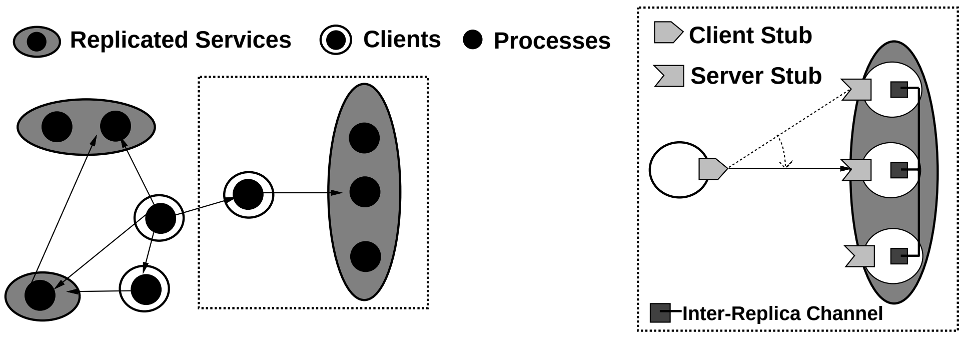

Figure 2.1: GRIP Communication Model

As this implies, servers can be replicated. In order to maintain an internally consistent state, replicas communicate over a logically independent channel according to the semantics of their replica control protocol. This communication is orthogonal to that presented in our protocol descriptions. However, in order to preserve global consistency guarantees, the sharing of a common communication substrate is assumed.

Processes, nodes, and network are all subject to failures, but we assume a communication system which can guarantee reliable delivery and duplicate detection over point-to-point links in the absence of failures. We assume processes and nodes are fail-silent. We do not consider arbitrary failures.

We also assume the existence of a group communication substrate capable of providing a number of services including virtual synchrony and causal reliable communication [3][1]. Such services may involve extra bookkeeping overhead and possible delivery delays. However, in cases where the functionality is required it is simpler and more efficient to have them implemented by the communication system rather than re-implemented by each application. We thus include them in our communication layer, but invoke them only on demand. Chapter 3 describes the communication service in more detail.

2.2 Design Overview

Our goal was to design a protocol to be run between a client and an arbitrary server replica with support for the following set of properties.

- **fault-tolerance:** the client should be able to access the server as long as at least one replica is accessible.
- **at most once semantics:** even in the presence of faults, an operation should be performed at most once on the server. This property is optional on a per-invocation basis.
- **independence from the replication mechanism:** the protocol guarantees do not depend on the internal replica consistency protocol.
- **low overhead:** for a particular combination of replication and correctness semantics, the protocol should exhibit little or no overhead as compared to dedicated solutions; in particular, the client is decoupled from inter-replica synchronization.

In order to achieve these goals, our design splits the protocol functionality into several modular services, represented as logical layers, some of which are optional (see Figure 2.2). The lowest layer is the communication service. We extend the basic communication model with two optional services which we call respectively *transparent messages* and *unpropagated messages*; these services provide support for efficient large-scale operation and weakly consistent replication schemes. We postpone the discussion of these services to Chapter 3.

Clients contact replicated services by using a lightweight Remote Access Protocol (RAP) to contact an individual replica. The corresponding RAP server entity relays the request to its local replica, where it is handled according to the semantics of the chosen

[1]Although group communication is not strictly necessary, its use simplifies both the implementation and the presentation of our protocols.

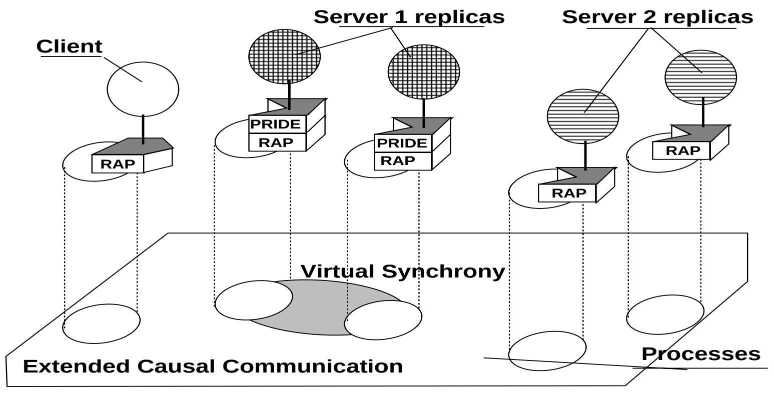

Figure 2.2: Interface Layer

replication protocol; replies are relayed back to the client in a similar manner. The RAP layer thus contains the core functionality required to support the generic remote invocation protocol; the details are described in more detail in Chapter 4. . Complete and up-to-date information about the membership of the replicated service is not required, thus avoiding expensive synchronization among servers and potential clients. Because it is designed to be light-weight, distributed at-most-once semantics are not explicitly guaranteed by RAP. If desired, they can be implemented for a particular service with the help of the optional Protocol for Repeated Invocation DEtection (PRIDE) layer, as described in Chapter 5.

Although our protocols assume the availability of (logical) reliable multicast and causal communication among service replicas, our approach explicitly does not require their use by service clients. This allows servers and clients to operate with low synchronization costs. If required by the application semantics, stronger synchronization can be established among the relevant layers of the service replicas. This duality is reflected in the structure of the GRIP design.

Chapter 3

Communication Service

In this chapter we describe the communication system underlying the GRIP protocol. It is composed of a base system including causal and group communication facilities extended by additional services. This extended subsystem introduces two novel qualities of service that improve the overall efficiency of the system: *transparent causal messages* and *unpropagated messages*.

3.1 On causal communication

In a distributed system, consisting of a collection of processes that communicate by exchanging messages, the order in which messages are delivered to processes is of major relevance to the application design. It can be argued that there are cases where the programmer can use application-specific mechanisms to preserve causal relations. In order to increase the modularity of the application, in GRIP we favor the alternative approach of defining an interface to a causal communication substrate. Since causal communication platforms are now widely available (a number of systems providing such services have been developed in recent years, such as ISIS [3], PSYNC/CONSUL [12], xAMp [15], Transis [2] and Totem [1]), this clear separation allows the programmer to use off-the-shelf software components and increases the portability and speed of prototyping of the resulting systems. Additionally, there is evidence that when causal delivery properties are necessary, ad-hoc solutions to the problem are usually complex and hard to prove correct [4]. Finally, since the optimal solution for the problem of reliable communication is highly dependent on the communications technology (speed, delays, error rate, etc.), having these services as a detachable module greatly simplifies software maintainability.

Despite its advantages, the use of causal communication has been somewhat limited by the overhead incurred by existing implementations. We can cite some disadvantages of existing causal communication services [5]: (i) little user control over message piggybacking policies; (ii) mandatory use of reliable communication to avoid blocking of message delivery. Thus, it is of major relevance to provide in such infrastructures a better match between the service provided and the application needs. The two new qualities of service that we will describe in this section, namely *transparent causal messages* and *unpropagated messages*, are intended to increase user control over automatic mechanisms that track and preserve causal relations.

3.2 Basic Service Properties

This section describes the properties we assume are provided by our basic communication service. We note that the basic communication service itself is a building block rather than a part of our design: we provide the associated description for completeness. We begin by assuming that processes are connected by a multicast communication network; whether multicast is physically or only logically supported is irrelevant at this level of abstraction. We assume the existence of two types of multicast channels: reliable FIFO channels and cheaper unreliable channels. We also assume that, on top of these services the communication layer preserves global causal delivery order (based on the *logical precedence* relation originally defined by Lamport [10]):

Logical precedence: *A message m_1 is said to* precede *or to be potentially causally related to a message m_2, represented as $m_1 \rightarrow m_2$, only if: (i) m_1 and m_2 were sent by the same process and m_2 was sent after m_1, or (ii) m_1 had been delivered to the emitter of m_2 before m_2 was sent, or (iii) m_3 exists such that $m_1 \rightarrow m_3$ and $m_3 \rightarrow m_2$.*

Additionally, we assume that group communication and membership services are optionally available. The membership service is responsible for giving each process in a group information, also called *views*, about the operational processes in the system. We assume that views are *linearly* ordered, i.e., that in case of network partitions, only a majority partition remains active and continues to receive views. Group communication follows the virtually-synchronous model as defined in [19]:

vs-**multicast:** *Consider a group g, a view $v_i(g)$, and a message M multicast to a subset $\mathcal{D}$ of members of group g. The multicast M is* vs-multicast *in view $v_i(g)$ iff: if $\exists_p \in \mathcal{D} \subset v_i(g)$ which has delivered M in view $v_i(g)$ and has installed view $v_{i+1}(g)$, then all processes $q \in \mathcal{D} \subset v_i(g)$ which have installed $v_{i+1}(g)$ have delivered M before installing $v_{i+1}(g)$.*

Using a sequencing mechanism, a total order protocol can be easily implemented on top of *vs*-multicast, yielding a *vs*-atomic service:

vs-**atomic:** *Let M_1 and M_2 be two* vs-multicasts *addressed to $\mathcal{D}$ and delivered in view $v_i(g)$ (with $\mathcal{D} \subset v_i(g)$). We say that M_1 and M_2 are* vs-atomic *if $\exists_{p \in \mathcal{D} \subset v_{i+1}(g)}$: $deliv_p(M_1) < deliv_p(M_2) \Rightarrow \forall_{q \in \mathcal{D} \subset v_{i+1}(g)} : deliv_q(M_1) < deliv_q(M_2)$.*

Finally, we assume the existence of a stronger quality of service that does not delivers a message until there is a guarantee that the message will be delivered to all correct processes. This service is called *uniform* [18] multicast and can be defined as follows:

vs-**uniform:** *Consider a group g, a view $v_i(g)$, and a message M multicast to a subset $\mathcal{D}$ of members of group g. The multicast M is* vs-multicast *in view $v_i(g)$ iff: if $\exists_p \in \mathcal{D} \subset v_i(g)$ which has delivered M in view $v_i(g)$, then all processes $q \in \mathcal{D} \subset v_i(g)$ which have installed $v_{i+1}(g)$ have delivered M before installing $v_{i+1}(g)$.*

Note that this definition is similar to that of *vs-multicast* except that it enforces the delivery of M to all processes that install a new view as long as some process delivers M, even if that process does not install the new view.

3.3 Communication Extensions

To avoid the costs inherent in the use of causal communication in large-scale systems, we use a novel scheme which permits batching of *unpropagated messages* and which also distinguishes two categories of messages: (normal) *opaque* causal messages and *transparent* causal messages.

3.3.1 Unpropagated messages

Many systems take advantage of particular application semantics to improve performance by selectively weakening the consistency constraints on replicated data. This generally results in non-identical replicas, which although it may not violate the replication semantics nevertheless means that two consecutive invocations to different replicas can produce possibly inconsistent results[1]. In order to provide a client view consistent with causality without sacrificing the performance benefits of weak replication, we extend our basic communication service to include the management of *unpropagated messages* (UMs).

An unpropagated message is a message that is logically sent but not physically propagated into the network. When the transmission of a UM is requested a dependency is created at the communication system level as for any normal (opaque) message, but rather than being transmitted the corresponding update is cached at the application level for future transmission. However, whenever an opaque message is prevented from being delivered because of a dependency on one or more UMs, the communication system automatically detects this and requests the originator of the UM to flush its cache to the network in order to guarantee progress.

Our protocols use piggybacking of messages as an optimization to reduce the number of messages physically exchanged. The unpropagated messages technique gives the application control of the piggybacking policy without the responsibility for the bookkeeping of the associated precedence information. In particular, techniques for reducing the amount of information required to enforce causality are the exclusive responsibility of the communication layer. We will show that our approach is extremely useful for avoiding violations of causality in the implementation of weakly consistent replicated services.

3.3.2 Transparent messages

One of the criticisms made of causal multicast communication systems [5] concerns the mandatory reliability requirement. Once a message introduces a causal dependency,

[1]There is usually an implicit assumption that any particular client always accesses the same replica, and thus sees an internally consistent view of the state.

that message must be reliably delivered; otherwise, successive messages will be prevented from being delivered. In some cases the delivery of a causal message is delayed until there is a guarantee that the message will be successfully delivered at all destinations; the sender may even be prevented from sending new messages until this guarantee is obtained.

To avoid this problem, we propose a scheme that distinguishes two types of messages: (normal) *opaque* causal messages and *transparent* causal messages. Transparent causal messages are messages that are delivered in causal order with respect to (normal) opaque messages but that do not themselves introduce causal dependencies. Thus:

- no message is ever delayed by a transparent message;
- no reliability constraints are imposed on the transmission of transparent messages.

The delivery order for transparent messages with regard to opaque messages is summarized in the following table (where opaque messages are represented in upper-case, transparent messages in lower-case, and right-arrows, $\rightarrow$, represent the logical precedence relation as defined in Section 3.2).

relation	delivery order	relation	delivery order
$M_1 \rightarrow M_2$	M_2 after M_1	$m_1 \rightarrow M_2$	undefined
$M_1 \rightarrow m_2$	m_2 after M_1	$m_1 \rightarrow m_2$	undefined

The implementation of transparent messages is fairly simple and any causal communication protocol can be adapted to provide this service at *almost no cost*. We present a possible implementation in Section 3.5.

3.4 Communication subsystem interface

Table 3.1 presents the interface of our communication extensions for managing unpropagated and transparent messages. We use the `type` field in our interface tables to differentiate between *upcalls* (represented in the table with *up*), and default procedure calls (represented with *down*). We use this notation in interface tables throughout the document.

- `c_transp` takes a destination list and a message as arguments, and sends the message using an inexpensive unreliable mode. A transparent message depends on all previous opaque or UM messages, but no message ever depends on a transparent message.

  Note: If there are UMs pending from the initiating process to any of the destination processes of a transparent message, a `c_sync` call will inevitably be triggered (since the transparent message depends on the pending UMs). For efficiency reasons, this call is logically activated *before* the transparent message is sent, and the actual transmission of the transparent message is delayed until the synchronization data is sent.

Name	Type	Parameters
Communication		
c_transp	down	PidList dest, Mesg m
c_vsmulticast	down	Group g, PidList dest, Mesg m
c_vsatomic	down	Group g, PidList dest, Mesg m
c_vsuniform	down	Group g, PidList dest, Mesg m
c_um	down	PidList dest, Mesg m
c_receive	up	Pid src, Mesg m, Serv qos, PidList sync, PidList miss
c_sync	up	PidList need
Membership		
g_join	down	Group g
g_leave	down	Group g
g_view	up	Group g, PidList view

Table 3.1: Communication Service Interface

- `c_vsmulticast/c_vsatomic/c_vsuniform` take a destination list and a message as arguments, and send the message according to the semantics of the communication service guarantees. The message is assumed to logically contain all previous UMs "sent" to any of the destination processes from the initiating process.

- `c_um` logically sends a unpropagated message. Since no real data is actually sent to the network only the destination addresses need to be specified in order for the communication layer to enforce causality.

- `c_receive` is an upcall invoked by the communication layer during the processing of an incoming message. If all preceding messages have been successfully delivered it invokes the `receive` routine provided by the next higher layer with the message and the identifier of the message source. If it had to contact any sites to request data corresponding to UMs in the message's causal history, it provides the list in `sync`; if any of these sites have failed, the corresponding process ids are provided in `miss`.

- `c_sync` is an upcall invoked by the communication system in order to request that an application process flush its cache of unpropagated messages. The list of processes which need the data (`need`) is provided, and a opaque message (logically containing the results of all relevant unsent messages) should be sent by the application to these destinations. This call is provided only as an optimization; failure to initiate a message as requested does not compromise correctness, since the next opaque message sent will implicitly include all previous unpropagated messages.

- `c_join` allows a process to become a member of a group;

- `c_leave` removes a process from a group;

- `c_view` informs a group member of changes in the membership.

Many protocols exist to enforce causal delivery. Some well-known examples of systems that provide such guarantees for both point-to-point and multicast communication are enumerated in Section 3.1; any of these can be extended to take into account the management of unpropagated and transparent messages. In the following section, we illustrate our scheme by sketching one possible implementation of such an extension.

3.5 An implementation

In this section, we choose a simple scheme for enforcing causal delivery [14] in order to illustrate how transparent and unpropagated messages can be easily implemented as extension to an existing algorithm. We note that several optimizations are possible, but we present a simplified case for reasons of clarity.

For the description of the algorithm, we make use of the multicast channels provided by the underlying network. We denote by `net_mul_unreliable` the unreliable multicast channels and by `net_mul_fifo` the reliable FIFO channels.

In the original protocol, each process maintains state information which describes its view of the global system state: a vector representing messages delivered, and a matrix representing messages sent. We extend the original protocol by adding extra state information for unpropagated messages. This information is represented by three variables, where the combination of $VSENT$ and $RSENT$ correspond to the single $SENT$ matrix in the original protocol, and n is equal to the number of cooperating processes:

DELIV:	*array [1..n] of integer;*
VSENT:	*array [1..n, 1..n] of integer;*
RSENT:	*array [1..n, 1..n] of integer;*

At initialization all elements are set to zero. The notation $DELIV_i[j]$, $VSENT_i[j,k]$, and $RSENT_i[j,k]$ indicates the variables local to process i, with j and k as indices of the arrays. $VSENT$ and $RSENT$ are matrices where for each process i a row j represents process i's view of messages sent (but not necessarily delivered) by j to each other process k; $RSENT$ corresponds to normal (opaque) messages, and $VSENT$ corresponds to the total number of messages logically sent, both opaque and unpropagated (UMs). $DELIV$ is an array where each element j corresponds to the number of messages from process j which have been delivered at process i. We distinguish between *reception* of a message, when the communication system receives it, and *delivery* of a message, when the local state is updated and it is delivered to the application.

The system state is updated on each process when a opaque message is sent, when an unpropagated message is virtually sent, when a (opaque) message is received, and when a received message is delivered to the application.

Sending an Opaque Message: When a message M is sent from process i to a set of processes $\mathcal{J}$ (see Figure 3.1), each (opaque) message sent is timestamped with the current values of $RSENT_i$ and $VSENT_i$; after the message has been sent, $\forall_{j\in\mathcal{J}}$ both $RSENT_i[i,j]$ and $VSENT_i[i,j]$ are set to the old value of $VSENT_i[i,j]$ incremented by one. Thus, after a send, the opaque and unpropagated message counters are always resynchronized.

```
//Sending (from process $i$ to processes $\mathcal{J}$)
opaque:
  net_mul_fifo ($M$, $VSENT_i$, $RSENT_i$) to $\mathcal{J}$
  $\forall_{j \in \mathcal{J}}$:
    set $VSENT_i[i,j], RSENT_i[i,j] := VSENT_i[i,j] + 1$
unpropagated (UM):
  $\forall_{j \in \mathcal{J}}$:
    set $VSENT_i[i,j] := VSENT_i[i,j] + 1$
transparent:
  optimization:
    if $\exists_{j \in \mathcal{J}}$ $VSENT_i[i,j] > RSENT_i[i,j]$ then
      call c_sync ($\mathcal{J}$) and wait for synchronization message to be sent
  net_mul_unreliable ($M$, $VSENT_i$, $RSENT_i$) to $\mathcal{J}$
```

Figure 3.1: Message Sending Protocol

Sending an Unpropagated Message: When a unpropagated message (UM) is sent from process i to a set of processes $\mathcal{J}$ (see Figure 3.1), no data is actually transmitted, and $\forall_{j \in \mathcal{J}}$ only the the $VSENT_i[i,j]$ matrix fields are incremented; the values in the $RSENT_i$ matrix are left unchanged, since they correspond to the values at the time the last opaque message was sent.

Sending a Transparent Message: When a message M is sent from process i to a set of processes $\mathcal{J}$ (see Figure 3.1), each transparent message sent is timestamped with the current values of $RSENT_i$ and $VSENT_i$. These vectors are left unchanged. If there are local UMs pending to $\mathcal{J}$, for efficiency reasons the message is promoted to opaque so that the UMs can be piggybacked and only a single message is required. In this case the message is sent using the opaque send procedure described above.

Since transparent messages do not generate dependencies, they do not consume space in the causal history. This observation also applies to this implementation if compression techniques are applied to the timestamps. For instance in [3] it is shown that a message only needs to be timestamped with the fields of the vector matrix that have changed since the last multicast. Consider for instance the case of a pure client in our GRIP protocol. Since the client exlusively uses transparent messages to contact the replicated service, its row of the matrix never changes and can thus be omitted in all message exchanges.

Receiving a Message: When a message M (along with the corresponding $VSENT_M$ and $RSENT_M$ matrices) is received from process i at proccess j (see Figure 3.2), the communication system must verify that all messages that causally precede the received message are delivered before it can deliver M. In the original protocol, it is sufficient to wait until each element of $DELIV_j$ is greater than or equal to the corresponding element of $VSENT_M[i]$, the row corresponding to the sending process.

For our extended case, it is possible that the values in $VSENT_M[i]$ have been incremented by the sending of UMs. In order to prevent blocking for such un-

```
//Receiving ( at process j from process i)
delay delivery of M until:
  $\forall_{k\neq i} DELIV_j[k] \geq VSENT_M[k,j] \wedge DELIV_j[i] = RSENT_M[i,j]$
optimization:
  $\forall k s.t. RSENT_M[k,j] < VSENT_M[k,j]$
    request initiation of c_sync at process k
```

Figure 3.2: Message receive Protocol

propagated messages, we keep in $RSENT_M$ the values corresponding to the last opaque message sent. As in the original protocol, the communication system must wait until any messages on which the current message is causally dependent have been delivered locally ($\forall_{k\neq i} DELIV_j[k] \geq VSENT_M[k,j]$); when $k = i$ only the last real message must have been received ($DELIV_j[i] = RSENT_M[i,j]$), since any subsequent UMs will automatically be included with the current message.

However, since when $k \neq i$ some of the counter values can represent UMs, in some cases the relevant messages have not necessarily been physically sent. To minimize the waiting time, the process therefore checks whether $RSENT_M[k,j] < VSENT_M[k,j]$ for any k. The communication system may then optionally choose to initiate the invocation of the `c_sync` upcall on process k (shown at the end of Figure 3.2) in order to explicitly request the propagation of the missing messages; or, it can simply wait for the next opaque message from process k. In either case, in the absence of failures it is necessary to wait for the delivery of a message logically containing these updates to be delivered before delivering M.

```
//Delivering ( at process j from process i)
set $DELIV_j[i], VSENT_j[i,j], RSENT_j[i,j] := VSENT_M[i,j] + 1$
$\forall_{k,l}$ s.t. $k \neq i \wedge l \neq j$
  $VSENT_j[k,l] := max(VSENT_j[k,l], VSENT_M[k,l])$
  $RSENT_j[k,l] := max(RSENT_j[k,l], RSENT_M[k,l])$
```

Figure 3.3: Message Delivery Protocol

Delivering a Message: We next consider the delivery at process j of an (opaque) message (M, $VSENT_M$, $RSENT_M$) sent from process i. Once the delivery criteria have been met, the communication system must update its local state (see Figure 3.3) and then deliver the message to the application.

First, updates corresponding specifically to the delivery of the message are performed. The ith element of the $DELIV_j$ array is set to the value of $VSENT_M[i,j]+1$, which corresponds to the total number of messages known to have been sent to the receiver by the sender, incremented by one for the current message. In the original protocol this update always corresponded to a simple increment, since only opaque messages were sent. With the introduction of UMs, it is possible for many unpropagated messages to be logically delivered in one physical message. Since delivery is constrained to respect causality, at the time of delivery we can be sure that all messages, opaque and unpropagated, known to have been sent by the sender prior to the current message must have already been received and delivered; thus, it is necessary to update the delivery record accordingly. $VSENT_j[i,j]$ and $RSENT_j[i,j]$ are also set to the same value, since sending a opaque message from process i implicitly includes all previous UMs sent by i. All other fields of the $VSENT$ and $RSENT$ matrices are set to the maximum of the local value and that in the message timestamp.

The information provided by the management of UMs can also allow optimizations in other layers of the system. For example, the RAP server tries to minimize synchronization by inspection of the `c_sync` indications. Since every `receive` upcall provides an indication of the nodes that had to flush their caches of unpropagated messages, when the RAP server detects that the delivery of a request required a cache flush it can redirect the client to the relevant replica via the `switch` parameter[2].

We also note that although we assume that the communication service can guarantee the absence of holes in the causal history for normal messages[3], we could make a weaker claim with respect to UMs. Since the application can introduce relaxed consistency semantics, we could also relay the blocking decision: when a strict interpretation of causality would require processing to block because of the inaccessibility of a site from which updates are required, we could refer final failure detection decisions to the application. However, this would be allowed *only* for the case of missing UM data; the absence of normal messages would still prevent delivery of any pending message(s).

3.6 Optional services: GUM

The un-propagated message mechanism offered by the communication system allows the application to select the caching and piggy-backing policies for updates generated during processing. In most cases, the application will take explicit control over these mechanisms, using semantic knowledge to optimize the size and number of the messages exchanged. For instance, a replication service can easily implement a lazy propagation scheme by marking an un-propagated message for each update and later sending a snapshot of the replica state containing all previous unpropagated updates (more examples will be given in section 6). However, a number of pre-defined message caching mechanisms can be offered as library functions, simplifying the application design. In this section we give a simple example of one such mechanism, called Grouped Unpropagated Messages (GUM).

GUM exports three primitives to manage un-propagated messages, as illustrated in Table 3.2: the upcall **gum_rec** is used to deliver a message to the user; **gum_isend** (immediate send), sends a opaque message to the network, automatically piggy-backing

[2]However, any decision made by RAP can always be overruled by an explicit switch request from the local replica.

[3]Known solutions, e.g. k-resiliency [3], can provide this modulo an associated cost even in the case of failures.

Name	Type	Parameters	Returns
gum_rec	up	Pid src, Mesg m	**none**
gum_isend	down	PidList dest, Mesg m	**none**
gum_gsend	down	PidList dest, Mesg m	**none**
gum_flush	down	PidList dest	**none**

Table 3.2: GUM Service Interface

all previous unpropagated messages with an intersecting destination set; **gum_gsend** (grouped send), registers the message as an unpropagated message and stores it to send later to the specified destination set; and **gum_flush** sends all buffered unpropagated messages in a opaque message to the specified destination set. GUM is also in charge of intercepting **c_sync** requests from the network and automatically sending all requested unpropagated messages.

A naive implementation of GUM is presented in figure 3.4. Our purpose is to illustrate the functionality and not to optimize the protocol. Messages exchanged among GUM entities logically contain a collection of user messages in the order they were generated. When sending a opaque message to a destination list *dest*, GUM piggybacks all unpropagated messages whose destination list intersects *dest*, adds a `gum` identifier, and sends the collection. The receiving entity detects the `gum` identifier, and if necessary splits the GUM messages into the complete set of user messages, delivering all messages addressed to the local entity in order of transmission.

3.7 Discussion

The communication extensions discussed in this chapter provide a mechanism for the controlled relaxation of causal and tightly synchronized communication. In particular, this approach acknowledges the difference between client and server and provides a much needed means of decoupling their reliability requirements. These new qualities of service thus provide the basis for most of the remainder of our system modules, and permit a great deal of flexibility and which is not often available in more typical tightly integrated systems.

```
Initialization:
  define type vmsg = (Msg,PidList);
  define function gumPiggyback ( PidList dest ) returns vmsgList, PidList
    vmsgList col := nil;
    PidList add := nil;
    ∀ elem in cache
      if ( (elem.dest ∩ dest) ≠ ∅ ) then do
        append (elem, col);
        add : = add ∪ elem.dest;
        elem.dest := elem.dest - dest;
        if elem.dest = ∅ then delete (elem, cache); od;
    return (col, add);
  declare vmsgList cache := nil;
Body:
  when gum_gsend ( PidList dest, Msg m ) invoked do
    append ( (dest, m), cache );
    c_send_um ( dest ); od
  when gum_isend ( PidList dest, Msg m ) invoked do
    (col, add) := gumPiggyback ( dest );
    append ((m,dest), col);
    c_send_opaque ( col, add ); od
  when gum_flush ( PidList dest ) or c_sync ( PidList dest ) invoked do
    (col, add) := gumPiggyback ( dest );
    c_send_opaque ( col, add ); od
  when c_rec ( Pid src, Msg m, ... ) invoked do
    if (gum_message) then
      call gum_rec for all grouped messages addressed to this process
    else
      call gum_rec directly with entire message
```

Figure 3.4: GUM Pseudo-Code

Chapter 4

Lightweight Remote Access

In our approach, clients of a (possibly) replicated service use a lightweight Remote Access Protocol (RAP) to contact an individual replica. The main characteristic of RAP is that it does not enforce the use of expensive communication primitives or tight synchronization between the client and the service replicas. In fact, complete and up-to-date information about the membership of the replicated service is not required and unreliable transparent messages are used. However, RAP guarantees that a client contacts a correct replica if at least one such replica is reacheable and provides the mechanisms to redirect the service contact when another more suitable replica exists. The services of RAP may be extended by the optional PRIDE layer, which is run only on the servers, as described in Chapter 5.

The RAP protocol distinguishes three types of messages, *requests*, *replies* and *end* markers. Requests are uniquely identified by the the client id plus a uniquifier; replies include the corresponding request identifier and a switch parameter whose purpose will be described in the following text. An end marker is used by a client to indicate the termination of an ongoing interaction with a server[1].

Name	Type	Parameters	Returns
r_invoke	down	Mesg request [, ReqId id]	Status s, Mesg reply
r_req	up	ReqId id, Mesg req, Bool undef, PidList contacts	**none**
r_reply	down	ReqId id, Mesg reply [, Pid switch]	**none**

Table 4.1: RAP Interface

The primitives offered by RAP are summarized in Table 4.1:

- **r_invoke** sends a request to the replicated service. The optional message identifier allows application clients to explicitly define the semantics of a retry. A status report is returned, along with the reply if the request was successful.

- **r_req** is an upcall which forwards a request to the local replica along with a (possibly null) list of any previously contacted processes.

[1]The *end* marker is automatically generated by the client stub and its use is not mandatory for correctness; however, it can be used by the server to compress bookkeeping information.

- **r_reply** is invoked by a replica to forward a reply to the client. An optional `switch` parameter allows the replica to indicate an alternate replica as the new *rep_contact* for this client.

A pseudo-code description of the protocol can be found in Figure 4.1. We assume that a given client makes one request at a time (parallel threads are modeled by different clients). In normal operation RAP maintains a list of contacted replicas for each invocation which provides hints to the server about the retry status of the request. However, the application may disable this functionality and take responsibility for defining its own retry semantics by providing the optional `id` parameter[2]; if it chooses this option, GRIP represents the request state as `undef` and does not modify the `contacts` list.

The RAP client initializes its state by obtaining a hint on the service membership; this hint is provided as a list of replica address identifiers, and does not need to be fully up-to-date. The RAP client parses the list in order to choose a *rep_contact*[3] with which it will establish a connectionless point-to-point interaction; once chosen, it moves the rep_contact to the head of the list.

Each application invocation supplies the RAP client with a request message and an optional message id. If a value is specified for the `id` parameter, the `undef` flag is set to TRUE. Otherwise, an id is supplied by the RAP client and the `undef` flag is set to FALSE. The request is then forwarded to the selected replica. The RAP client waits for the reply during a predefined timeout period; since unreliable communication is used, when a request times out, the request is retransmitted before another replica is contacted. When a pre-defined number of retries is exceeded, the contact is considered inaccessible. The identifier of the inaccessible replica is added to a `contacts` list (initialized to `nil` at each invocation) unless the `undef` flag is set, in which case the `contacts` list remains empty. The state of a request can thus be determined from the values of the `undef` and `contacts` parameters (e.g., original requests can be identified by the combination of a FALSE `undef` flag and an empty `contacts` list).

The RAP client continues contacting replicas until either a reply is received or the list of service members is exhausted. If the list is exhausted and no reply has been received, either a new hint is obtained and the process recommences, or an error is returned to the client. With each reply the server includes a hint, the `switch` parameter, indicating a suggested contact replica for the client. The client may update its rep_contact based on this information[4]. This mechanism allows the replication service to control its own replica allocation and load balancing.

On the server side, RAP waits for requests and uses the `r_req` upcall to forward them, with their associated status information, to the local replica. The replica reply is then forwarded back to the RAP client via the `r_reply` routine. The reply includes an optional `switch` parameter which can be used as a hint to optimize service response: it

[2]The application specified `msgid` parameter provides support for scenarios such as replicated clients executing in lock-step, where only via an application specified message id can the set of requests be recognized as logically equivalent.

[3]Criteria for replica selection are orthogonal to the protocol, but can consider factors such as load balancing for performance optimizations.

[4]Although the update is optional and does not affect correctness, for simplicity the pseudo-code assumes the update always occurs.

```
Client
Initialization:
   PidList hint := getHint ();
   Pid r_contact := selectContact (hint);
   Int gl_uid := 0;
   Pid src := myself;
   Request req := nil;
   Bool undef := FALSE;
   PidList contacts := nil;
   Int retries :=0;
Body:
   for each r_invoke ( req_data, [cl_uid] )
      if cl_uid exists then do
         uid := cl_uid; undef := TRUE; od
      else do
         uid := gl_uid; gl_uid := gl_uid + 1; undef := FALSE; contacts := nil; od;
      until return do
         req:= [ src, uid, rq_data, undef, contacts ];
         retries := 0;
         until ( [src, uid, switch, reply_data] received or retries = MAX_RETRIES ) do
            c_transp ( r_contact, req );
            retries := retries + 1;
         od
         if [src, uid, switch, reply_data] received then do
            r_contact := switch; return reply_data to client; od;
         else do
            if (undef = FALSE) then append (r_contact, contacts);
            r_contact := nextContact (r_contact, hint);
            if r_contact = nil then get new hint or return with error; od;
      od //until return
Termination:
   c_transp ( r_contact, [src, END] );

Server
Body: // (for each request)
   when receive (src, uid, data, undef, contacts) do
      r_req ( uid, data, undef, contacts);
      switch := selectSwitch ( ); od; // default is myself
   when r_reply ( uid, reply, [s_switch]) do
      if switch specified then switch := s_switch;
      c_transp ( src, [uid, rpl_data, switch] ); od;
```

Figure 4.1: RAP Pseudo-Code

could be used, for example, as a part of a mechanism for load balancing across server replicas, or to optimize message traffic by co-locating requests from different clients that are accessing the same data.

The RAP protocol is thus extremely light-weight, but nevertheless offers support both for dynamic re-binding via the `switch` parameter and for client semantic control via the client-specified `msgid` parameter. It explicitly does not address the complex issue of distributed retry management. In the following section we present our Protocol for Repeated Invocation DEtection, which when layered on top of RAP optionally provides this functionality.

Chapter 5

Distributed Retry Detection

PRIDE is a protocol that can be run among a set of server replicas to guarantee at-most-once semantics for client requests. Note that since point-to-point links are reliable, for each individual replica local at-most-once properties are guaranteed *a priori*: PRIDE extends the at-most-once guarantees to the replica set as a whole. PRIDE's semantic guarantees are optional on a per-service and a per-invocation basis. Thus, there is negligible overhead for requests which do not require retry detection. The replication protocol is assumed to be responsible for any internal propagation of results according to its own semantics.

Since the protocol tolerates failures, three variants of at-most-once are available: (i) *reliable*, which guarantees that the request is executed by at most one correct replica (although it may have been previously executed by a failed replica); (ii) *propagated*, which causes knowledge of the request to be propagated to a designated set of replicas and guarantees that if at least one of these replicas receives the information and remains correct the request will not be re-executed by any other replica; (iii) and *uniform*, which guarantees that the request is executed by at most one replica even if that replica fails. Providing these different qualities of service allows the application to make performance/functionality tradeoffs based on its operation semantics. An advantage of PRIDE is that the selection of the appropriate detection semantics can be made (or upgraded) on an operation by operation basis, independently of the internal replication protocol, and at any point during the computation until the reply is sent to the client. This provides extra flexibility to the implementor of the replication scheme.

Types	p_status	`oneof` (original, executing, unknown, blocked)
	p_act	`oneof` (ignore, register, disamb)
	p_repmode	`oneof` (send, reg, reg_send)

	Name	Type	Parameters	Returns
Functions	**p_request**	up	ReqId msgid, Msg rq, p_status stat	p_act act
	p_reply	down	ReqId msgid, Msg reply, p_repmode action[, Pid switch]	none
	p_propagated	down	ReqIdList msgids, Msg msg, PidList dest, Bool delegate	none
	p_uniform	down	ReqIdList msgids, Msg msg, Bool delegate	none

Table 5.1: PRIDE Interface

5.1 PRIDE Interface

In the following paragraphs, we describe the interactions between PRIDE and its adjacent layers: RAP (below) and the replicated application (above). We first reiterate our assumption that the underlying communication service provides causally ordered communication and virtual synchrony, and recall that client requests use lightwieght *transparent* point-to-point communication to reduce unnecessary communication overhead. We note that the complexity inherent in such a distributed retry detection scheme is greatly reduced by the functionality of our communication service, as will become clear in the subsequent discussion.

The PRIDE interface is summarized in Table 5.1. PRIDE receives incoming requests from RAP (via the `r_req` upcall) and forwards them to the replication scheme using the `p_request` upcall. Original requests (for which the *contacts* list is defined and empty) are forwarded without any additional processing. This guarantees optimal behavior in the usual case.

Otherwise (the *contacts* list is either undefined or nonempty) PRIDE searches *locally* for a record of the request (PRIDE peer entities periodically exchange information about executed requests). If a record exists, then if a reply has already been generated PRIDE re-sends the reply to the client and exits without invoking the application layer. If no reply has been generated, or if no local record exists, PRIDE forwards the request to the application with a flag indicating the request status.

In the request indication, the status flag can assume one of the following values: (a) *original*, the request has not been delivered to any other active replica; (b) *executing*, the request is being executed by at least one active replica; (c) *unknown*, a search among the replicas needs to be performed to obtain more information; (d) *blocked*, a failed replica has propagated the knowledge of the request to at least one active replica (using *propagated* semantics) or the request was declared as *uniform* and no reply is promised by any active replica.

The return value of the `p_request` primitive is used to signal the application decision to PRIDE: `register`, `ignore`, or `disamb`. If at-most-once semantics are desired, retry detection is activated with `register`; this implicitly activates retry detection with the semantic guarantees set to *reliable*. If at-most-once semantics are desired but the request indication is `unknown`, the application can use `disamb` to instruct PRIDE to disambiguate the state by performing a search among the replicas for a record of this request. In this case the application implicitly discards the request; PRIDE is responsible for re-submitting it, if appropriate, as soon as the search ends.

Once a request has been registered with PRIDE, the semantic guarantees may optionally be upgraded to *propagated* via the `p_propagated` primitive, which propagates the relevant PRIDE record(s) to each of the processes in `dest` (usually by piggybacking the information on `msg`, which provides a vehicle for opaque application data). If the `delegate` flag is `TRUE`, each of the destination processes is responsible for independently producing a reply for the request. When the flag is set to `FALSE`, only the invoking node is responsible for (eventually) producing the reply. When this primitive returns, the associated request(s) is (are) guaranteed not to be re-executed as long as at least one of the processes in the *dest* list receives the propagated PRIDE record(s) and remains correct.

Alternatively, the semantic guarantees may be upgraded to *uniform* with the `p_uniform` primitive. This works in much the same manner as `p_propagated`, except that the propagation uses the `uniform` quality of service and is sent to the complete replica set. Use of this quality of service guarantees that as long as some process receives the message associated with the `p_uniform` primitive no other process will re-execute the request.

The reply can then be generated by the active replica(s) at any point during request execution. It is registered with the PRIDE layer, for possible future propagation, with the `p_reply` primitive (which also optionally forwards the reply to RAP to relay to the client).

5.2 PRIDE Protocols

In this section we describe the PRIDE retry detection protocols. In order to prevent the re-execution of requests, PRIDE keeps a record for each request for which retry detection has been activated[1] Since RAP guarantees that a given client does not forward a new request before receiving the reply for the previous one, at most one record must be kept for each client: receipt of a request more recent than the one recorded implicitly allows the old record to be deleted. The contents of the accounting record are as follows:

```
define type p_type = oneof ( reliable, propagated, uniform )
RECORD Request BEGIN
   Uniquifier   id;      // the uniquifier of the request
   p_type       tp;      // the type of guarantees provided
   Msg          rpl;     // the reply if generated
   Msg          reqst;   // the request if no reply yet
   Pid          srch;    // the initiator of a search
   PidList      exec;    // executing set
   PidList      prop;    // propagated set
   PidList      sent;    // sent set
END;
```

A record is created for a request on activation of retry-detection or search, and is updated when the reply is registered, when the the quality-of-service is upgraded, or when the search completes. Replicas also exchange local records among themselves to optimize the retry detection process. When sending an opaque message to another replica, PRIDE automatically piggybacks all relevant (i.e. unsent) request records with their corresponding data updates; it also appends the destination to the `sent` field of each record[2]. Upon message reception, PRIDE strips all records appended by its peer entity and updates its local state as necessary (see Figure 5.1).

Before describing in detail how these records are used, we summarize the functionality of PRIDE interface management primitives (see Figure 5.2).

[1]In the presence of loosely synchronized clocks and assumptions about the maximum lifetime of messages in the network it is possible to garbage collect last-ack records using a timeout scheme.

[2]This information is usually piggybacked on the normal inter-replica application message traffic, although in the absence of such traffic a dedicated message can be generated by PRIDE to disseminate the

```
define procedure pridePiggyback ( Msg m, PidList dest )
  $\forall$ record r: if ( dest - r.sent $\neq \emptyset$ ) then do
    r.sent := r.sent $\cup$ dest; piggyback r on msg; od;

define procedure prideStrip ( Msg m, PidList dest )
  $\forall$ record rnew $\in$ msg do rloc := localRec ( rnew.id );
    if (rloc = nil) then add rnew to local;
    else // update rloc fields
      if ((rnew.rpl $\neq$ nil) $\wedge$ (rloc.rpl = nil) then
        rloc.rpl := rnew.rpl;
      if (rnew.tp > rloc.tp) then do
          rloc.tp := rnew.tp; rloc.exec := rloc.exec $\cup$ rnew.exec;
          rloc.prop := rloc.prop $\cup$ rnew.prop;
          rloc.sent := rloc.sent $\cup$ rnew.sent;od;
```

Figure 5.1: Record Dissemination

```
// from RAP layer
when r_req (msgid, request, undef, contacts) invoked do
  pFilterRequest (msgid, request, undef, contacts);

// from local replica
when p_reply (msgid, reply, action[, switch]) invoked do
  if ((action = reg) $\vee$ (action = reg_send)) then do
    rec := localRec ( msgid );
    if ((rec = nil) $\vee$ me $\notin$ rec.exec then exit with error
    rec.rpl := reply; rec.sent := $\emptyset$; od
  if ((action = send) $\vee$ (action = reg_send)) then do
    r_reply (msgid, reply[, switch]); od // relay reply to client

when p_propagated (msgids, m, dest, delegate) invoked do
  $\forall_{mid \in msgids}$ :
    Request rec := localRec (mid); rec.tp := propagated;
    rec.sent := $\emptyset$; rec.prop := dest;
    if $\neg$ delegate then rec.exec := me; else rec.exec := dest;
  c_vsmulticast (mygroup, dest, m) ; // outstanding recs will be piggybacked

when p_uniform (msgids, m, delegate) invoked do
  $\forall_{mid \in msgids}$ :
    Request rq := localRec ( mid ); rq.tp := uniform;
    rec.sent := $\emptyset$; rec.prop := all_replicas;
    if $\neg$ delegate then rq.exec := me; else rq.exec := all_replicas;
  c_vsuniform ( mygroup, all_replicas, m ); // outstanding recs will be piggybacked
```

Figure 5.2: PRIDE Interface Management

- **r_req** is invoked from the RAP layer; it invokes pFilterRequest to filter the request before invoking `p_request` to pass to the application.

- **p_reply** performs some combination of reply registration and transmission, according to the the value of the `action` parameter. If registration is indicated, the routine retrieves the request record, updates the reply field, and reinitializes the `sent` list to ensure propagation of the new information. If transmission is indicated, it then invokes `r_reply`. Only a replica which is a member of the `exec` set may register a reply; furthermore, a reply may be registered only if the corresponding local request record already exists.

- **p_propagated** upgrades requests from the *reliable* to the *propagated* quality of service. It stores in the `prop` field the list of nodes to which the records are being propagated and, as above, reinitializes the `sent` list to ensure propagation of the new information; it also sets the `exec` field according to the `delegate` parameter to indicate the replica(s) responsible for producing a reply. Then, after piggybacking any relevant records (including the current request), the associated `msg` is *vs*-multicast to the replicas specified in `dest`.

- **p_uniform** upgrades requests from the *reliable* to the *uniform* quality of service. As above, initializes the `sent` and `exec` fields. Then, after piggybacking any relevant records (including the current request), the associated `msg` is disseminated using *vs*-uniform to the all replicas.

```
define procedure pFilterRequest (id, req, undef, contacts)
  if ((undef = FALSE) ∧ (contacts = ∅)) then // process
    if (p_request (id, req, original) = register)
      then do // create a local replica
        rec := newRequest (id); rec.reqst := req;
        rec.tp := reliable; rec.exec := me; od
  else // potential retry
    rec := localRec (id); // lookup record
    if (rec = nil) then // no record available
      if (p_request (id, req, unknown) = disamb) then do
        rec := newRequest (id); rec.reqst := req;
        rec.tp := reliable; rec.exec := ∅;
        prideSearch (id); od
    else // record available
      pProcessRecord ( rec, search_done = FALSE );
```

Figure 5.3: PRIDE Request Filter

We now describe the request filtering mechanism, illustrated in Figures 5.3 and 5.4. When RAP delivers a request to PRIDE (via `pProcessRequest`) it indicates the list of previously contacted replicas (when `undef = FALSE`). When the list is empty, PRIDE immediately forwards the request to the application. Otherwise, PRIDE looks for a

records.

local record. If no such record exists, the request is delivered marked as `unknown`. A request with `unknown` status can result in a request to trigger a global search among the replicas to concretely determine the request status; such a search is performed by the call to `prideSearch`.

```
define procedure pProcessRecord ( rec, search_done )
  if (rec.rpl $\neq$ nil) then
    r_reply (id, rec.rpl); // relay existing reply to client
  else if (rec.exec $\cap$ r_view $\neq \emptyset$ ) then
    p_request (id, rec.reqst, executing);
  else if (rec.prop $\cap$ r_view $\neq \emptyset$) then
    p_request (id, rec.reqst, blocked);
  else if rq.type = uniform then
    return blocked; // replica unreachable
  else if (rec.srch $\in$ r_view $\wedge$ rec.srch $\neq$ me) then
    p_request (id, rec.reqst, executing);
  else if (search_done = TRUE) then do// treat as original
    p_request ( id, rec.reqst, original );
    rec.tp := reliable; rec.exec := me; od
  else if (p_request (id, rec.reqst, unknown) = disamb) then
    prideSearch (id);
```

Figure 5.4: PRIDE Process Record

If a record does exist, it is examined in the procedure `pProcessRecord` in order to determine the appropriate action to take. If a reply is already available (the request has been served and recorded), it is resent to the client. If a reply is not available, the other fields are analyzed: if an active replica is already executing or has initiated a search to disambiguate the request's status, the request is delivered with an `executing` indication; if the request record has been *propagated* to an active replica or upgraded to *uniform* but no site responsible for execution is alive, it is delivered as `blocked`[3]. At this point, if `pProcessRecord` has been invoked at the end of a search procedure, the request is resubmitted to the application with a status of `original`. Otherwise, it is delivered as `unknown`.

Whenever a request is submitted as unknown, the application may choose to disambiguate this result. A `p_request` return value of `disamb` activates the PRIDE guarantees and implies the activation of the reliable quality of service. Disambiguate requires a search among all relevant replicas for records of the associated request. In some cases, such as that of a replicated client operating in lock-step (`undef` = TRUE), it is impossible to know of the existence of previously contacted replicas; in this case, it is possible for more than one replica to receive a request with a status of `unknown` and request a search to disambiguate the request status. Thus it is possible for multiple replicas to initiate concurrent searches for a given request.

The `prideSearch` process is illustrated in Figure 5.5. It first checks whether all the replicas in the `contacts` list are in the current replica view. If so, the search need

[3]Virtual synchrony, which is used to establish the *views* of reachable replicas, ensures that when a site is removed from the current view there are no messages pending from that site.

```
define procedure prideSearch (id)
   if ($\forall_{p \in contacts} : p \in$ r_view) then
      vs-atomic ( [SEARCH, id, me] contacts );
      else vs-atomic ( [SEARCH, id, me], all_replicas );
   wait for a SEARCH-ACK from each active destination replica;
   // strip remote records from SEARCH-ACK and merge with local record
   prideProcessSearch (id);

define procedure receiveSearch ( id, searcher)
   rec:=localRec(id);
   if ( (rec = nil) then do
      rec := newRequest ( id ); rec.exec := $\emptyset$;
      rec.tp := reliable; rec.srch := searcher; od
   else if ( rec.srch $\notin$ r_view) then
      rec.srch := searcher;
   c_vsmulticast ( mygroup, searcher, [SEARCH-ACK]);
   // piggyback local record on SEARCH-ACK

define procedure prideProcessSearch (id)
   rec := localRec ( id ); // lookup search result
   pProcessRecord ( rec, search_done = TRUE );
```

Figure 5.5: PRIDE Search Functions

only be addressed to those replicas, since if any processing has been intiated it must have been by one of them. If any of these replicas have been removed from the current replica view, or if the `contacts` list is undefined, then the search must be addressed to all replicas. The search itself is initiated with a virtually synchronous atomic request for information to all relevant replicas. The process then waits for the results and analyzes them with the `prideProcessSearch` routine. As discussed previously, this processing may result in the request being resubmitted to the application. The use of the atomic service is required to serialize concurrent search operations: the first search to be received for any given request wins the competition.

On receiving a search request, a replica processes it with the `receiveSearch` routine. This routine searches for a local record; in order to achieve deterministic results, a search implicitly creates a request record at each replica if one does not already exist. This record is initialized with the `exec` field empty, to indicate that its execution status is unknown (such a record is also created when a replica registers a submitted request, but with the `exec` field initialized to contain the registering replica). The `srch` field of the new record is used to store the identity of the searcher; if a local record does exist, if the `srch` field contains a replica which is no longer in the current replica view then the current searcher replaces the existing one. Finally, the updated local record then forwarded to the searching process using a point-to-point reliable message.

5.3 Membership changes

PRIDE is particularly robust with respect to membership changes because each replica always takes the conservative approach of starting a search when no local record exists for a given request, such as for example in the case of a potential retry. Thus, no special operation needs to be performed in order to integrate new replicas at the PRIDE level [4]: if a new replica receives a potential retry, it simply starts a search for the associated record. In particularly unreliable networks, where retries are likely to be frequent, it might prove worthwhile to initialize new replicas with recently updated records in order to avoid potential searches. This however is optional and is not required for correctness.

```
define procedure prideViewChange ( new_view )
  if ( (new_view - r_view) $\neq \emptyset$ ) then // new members
    // optionally, send recent records to new members
    r_view = new_view;
    $\forall_{rec}$ : rec.exec $\cap$ new_view = $\emptyset$ do
      prideSearch (rec.id); // optional eager recovery
```

Figure 5.6: PRIDE View Change

Similarly, if there is a group change due to the failure/disconnection of one or more replicas, PRIDE need take no special action since it can simply wait for RAP to generate a new retry for the request. As soon as the retry is received, the failure of any executing replica(s) will be detected in the `pFilterRequest` procedure and the appropriate corrective action will be executed (if possible, the request will be re-submitted to another replica). As an alternative to this lazy recovery approach, PRIDE can start an immediate search for all records of requests potentially depending on failed replicas[5] as soon as the failure is detected (see figure 5.6). This is functionally equivalent to receiving a retry for these requests: the `prideSearch` routine is invoked to check whether the request needs to be re-submitted. This eager approach provides a faster recovery from failures.

[4]Naturally, an application level protocol must be run to initiate the new replica, but this is transparent to PRIDE.

[5]If more than one record exists, several search procedures can be merged in a single operation.

Chapter 6

Examples

In this chapter we illustrate how our protocols can be used to implement different replication schemes. We consider two schemes, covering extremely different consistency semantics: a very strict state machine approach and a weak scheme which periodically propagates updates based on internal semantics.

The first example, illustrated by Figure 6.1, shows how a replicated state-machine can be implemented using GRIP/PRIDE support. All read requests are executed locally by any receiving replica without any additional synchronization. An update request that is an original is registered with PRIDE by returning `register`; it is then atomically multicast in a special RELAY message to the entire replica set using `p_propagate`, upgrading the retry detection to *propagated*. The `delegate` flag is set to TRUE, indicating that all designated receivers should execute the enclosed request. If the update status is unknown, then `disamb` is returned in order to trigger a global search for information: this implicitly results in a new `p_request` invocation if execution is an option. Otherwise, `ignore` is returned since all other cases are irrelevant to the state machine's consistency semantics. Finally, when the RELAY message is received, each receiving replica sets the `p_reply action` flag according to its identity and then executes the request: only the replica that received the original client request returns the reply.

```
when p_request ( id, rq, stat ) invoked and rq is a read do
    response := ignore; reply := execute ( rq );
    r_reply ( id, reply ) // no filtering required
when p_request (id, rq, stat) invoked and rq is a write do
    if stat = original then
        response := register;
        p_propagate (id, RELAY [id, rq], all_replicas, true );
    else if stat = unknown then response := disamb;
    else response := ignore;
when RELAY [id, rq] received from some replica R do
    reply:= execute (rq);
    if (me = R) then p_reply (id, reply, reg_send );
    else p_reply ( id, reply, reg )
```

Figure 6.1: State-machine implementation

The second example, illustrated in Figure 6.2, shows the implementation of a weak replication scheme. Read operations are assumed to be idempotent and are executed without further processing. In this scheme, updates are executed at the receiving replica, and may be legally re-executed at another replica as long as the results of the update have not been propagated to any active replica. Each replica can execute updates from concurrent clients in parallel; updates are only exchanged from time to time, according to the semantics of the replication policy. The re-execution of these requests after propagation is prevented by using the *propagated* quality of service of PRIDE.

```
when p_request ( id, rq, stat ) invoked and rq is read do
   response := ignore; reply := execute ( rq );
   r_reply ( id, reply ) // no filtering required
when p_request ( id, rq, stat ) invoked and rq is write do
   if stat = original then // safe to execute
      response := register; idlist := idlist + id;
      reply := execute (rq ); diffs := addDiffs();
      c_send_um (all_replicas); p_reply ( id, reply, reg_send );
   else if stat = unknown then response := disamb;
   else response := ignore;
when time to propagate to dest ∨ c_sync ( dest ) do
   p_propagate ( idlist, dest, diffs, false );
   diffs := 0; idlist := 0;
```

Figure 6.2: Weak propagation

As in the state machine example, if the request status is `original` it is safe to execute, and in the `unknown` case the receiver returns a `disamb` to request further information. If the request is known to be executing at an active replica or is blocked, the receiver chooses to discard the request. In this example, the propagation of updates is delayed arbitrarily according to internal semantics. Thus, during the processing of a request any relevant updates are registered with the extended communication service using `c_send_um`, and the id of the request is added to a local list of unpropagated updates. When request processing is complete, the reply is registered with PRIDE and sent to the application client, but is not necessarily propagated to any other replicas. Propagation is performed asynchronously by the periodic invocation of the p_propagate primitive, providing the list of unpropagated update ids as an argument and specifying the list of replicas to which the updates should be sent. Here, the collection of updates (`diffs`) is sent as p_propagate's `msg` parameter. The `diffs` and `idlist` variables are reinitialized to zero after each `p_propagate` request.

The last example, presented in Figure 6.3, shows the implementation of a primary-backup replication scheme, where all updates must be committed at all active replicas before a reply is disseminated. Read requests are executed locally (at any replica) as usual. Updates are executed only at the primary replica; if a backup receives an update request, it forwards it to the primary[1]. The primary commits each update using the *p_uniform* quality of service of PRIDE, ensuring that the request is not re-executed and

[1]In some cases it may also be desirable to change the client contact at the same time, although if the client traffic is primarily read requests such a change may not be optimal.

that all backups record the update. It then forwards the reply to the client.

```
when p_request ( id, rq, retry ) invoked and rq is read operation do
  reply := execute (rq ); r_reply ( id, reply )
when p_request ( id, rq, retry ) invoked and rq is write operation but I'm not PRIMARY do
  send RELAY [id, rq] to PRIMARY; exit;
when p_request ( id, rq, retry ) invoked and rq is write operation and I'm PRIMARY or
when RELAY[id,rq] received do
  if retry = false ∨ p_search ( id ) = execute then
    reply[id] := execute (rq); p_reply ( id, reply[id], dosend := false );
    p_uniform ( id, update[id], all_replicas, delegate := false );
    // wait until propagation is assured
    p_reply ( id, reply[id], dosend := true, switch := me ); //commited
when update[id] received from PRIMARY do
  if I'm PRIMARY then do nothing
  else store update
when PRIMARY fails do
  select new PRIMARY;
```

Figure 6.3: Primary-backup implementation

Although we present only a few examples, it is of course possible to express others in a similar framework. We note also that optimistic applications which have no need for retry detection mechanisms can also invoke the lightweight RAP layer directly.

Chapter 7

Related Work

To our knowledge there are few examples in the literature of remote invocation protocols designed to interoperate with multiple replication strategies. Most existing approaches are designed with a specific replication scheme and thus are not concerned with some of the problems that our protocols tackle. Here we relate our work to some of the more relevant examples of replicated invocation protocols.

One of the first examples was presented by Cooper in [7]. The protocol allows invocation of a *troupe* of replicas, but is more restrictive than ours; it requires programs to be completely deterministic and it assumes that all troupe members receive and process all requests.

ISIS[3] offers a multicast communication system that supports virtual synchrony and preservers global order across groups. Other communications systems have also successfully applied this approach [13,15,2,1]. In this paper we pursue this line of research, but in order to avoid the synchronization costs required to enforce reliable multicast communication between clients and replicated services we only require virtually synchronous communication among replicas. Additionally, we add the concept of unpropagated messages to maintain causality in the presence of weak replication schemes.

Lazy replication [9] is similar to our scheme in that it relies on inexpensive point-to-point interactions between clients and the replicated service. Global consistency is achieved by collecting and exchanging *multipart* timestamps; we achieve the same effect implicitly by relying on the services of the underlying communication service. However, our approach is not restricted to a single replication model: the unpropagated messages mechanism allows us to achieve the same benefits as lazy replication with more transparency for the client and more control for the server. Furthermore, with our approach compression mechanisms can be automatically and transparently applied by the communication system [3,17].

More recently, [21] presented a replicated RPC running on top of Amoeba in which point-to-point RPC was used to contact a designated replicated service coordinator. This is primarily due to the closed nature of Amoeba groups, but the authors also stress the transparency advantages inherent in this approach. However, contrary to our generic approach, the contacted replica assumes a state-machine like replication scheme and always reliably broadcasts the request to the entire server group. Our approach allows the replication scheme to decide when the request should be propagated, and also allows dynamic selection of coordinators on a per-request basis, thus avoiding potential performance bottlenecks.

Recent work [20] also addresses the need for dynamic flexibility in distributed and replicated systems. Although not specifically stressed in this presentation, support for dynamic evolution and for large scale were strong motivating factors for the decoupling of clients from the implementation details of services.

Chapter 8

Conclusions

The GRIP protocol provides flexible support for the construction of replication-transparent remote invocation of replicated services. Unlike the functionality provided by most existing systems, it leaves the semantics of the replication protocol transparent to the remote invocation protocol and provides support for dynamic reconnection and client semantic control; moreover, it introduces explicit support for weakly consistent replication strategies and provides optional per-invocation distributed retry detection. GRIP addresses issues of dynamic evolution and large scale by keeping the client process insulated from any internal server evolution, and also ajusting readily to replica group membership changes. Our communication extensions for transparent causal messages introduce a much needed intermediate solution between causal and non-causal communication for weakly coupled communicating peers. Such transparency is of critical importance to the future development of fault-tolerant distributed applications, particularly for large-scale environments.

We are currently implementing the GRIP protocols in the context of the ROMANCE system [16] in order to enable further optimization and experimentation.

Acknowledgments

The authors are grateful to the anonymous reviewers of SRDS-13 for their comments on a reduced version of this paper.

Bibliography

[1] Y. Amir, L. Moser, P. Melliar Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, Pennsylvania, USA, May 1993.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A Communication Sub-System for High-Availability. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, pages 76–84. IEEE, 1992.

[3] Kenneth Birman, Andre Schiper, and Pat Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[4] Kenneth P. Birman. A Response to Cheriton and Skeen's Criticism of Causal and Totally Ordered Communication. Technical report, Cornell University, October 1993.

[5] D. Cheriton and D. Skeen. Understanding the Limitations of Causally and Totally Ordered Communication. In *Proceedings of the 14th Symposium on Operating Systems Principles*, Asheville, NC, USA, December 1993.

[6] David R. Cheriton. Problem oriented shared memory revisited. In *Proceedings of the 5th ACM SIGOPS European workshop*, Mont Saint-Michel, France, September 1992. ACM.

[7] Eric C. Cooper. Replicated procedure call. In *Proceedings of the 3rd ACM symposyum on Principles of Distributed Computing*, Berkeley, CA 94720, USA, August 1984. ACM.

[8] Elmootazbellah Elnozahy and Willy Zwaenepoel. Replicated distributed process in Manetho. In *Digest of Papers, The 22nd International Symposium on Fault-Tolerant Computing Systems*, page 18. IEEE, 1992.

[9] R. Ladin, B. Liskov, and L. Shrira. Lazy replication: Exploiting the semantics of distributed services. In *Proceedings of the Ninth Annual ACM Symposium of Principles of Distributed Computing*, pages 43–57, Quebec City – Canada, August 1990. ACM.

[10] Leslie Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *CACM*, 7(21), July 1978.

[11] Mesaac Makpangou, Yvon Gourhant, Jean-Pierre Narzul, and Marc Shapiro. Fragmented objects for distributed abstractions. In *Advances in Distributed Systems.* IEEE, 1992.

[12] S. Mishra, L.L. Peterson, and R.D. Schlichting. Protocol modularity in systems for managing replicated data. In *Proceedings of the Second Workshop on the Management of Replicated Data*, pages 78–81, Monterey, California, November 1992. IEEE.

[13] Larry L. Peterson, Nick C. Buchholdz, and Richard D. Schlichting. Preserving and Using Context Information in Interprocess Communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[14] Michel Raynal, Andre Schiper, and Sam Toueg. The causal ordering abstraction and a simple way to implement it. *Information processing letters*, 39(6):343–350, September 1991.

[15] L. Rodrigues and P. Veríssimo. xAMp: a Multi-primitive Group Communications Service. In *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Houston, Texas, October 1992.

[16] L. Rodrigues and P. Veríssimo. Replicated object management using group technology. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, pages 54–61, Lisboa, Portugal, September 1993. Also as INESC AR/28-93.

[17] L. Rodrigues and P. Verissimo. Causal separators and topological timestamping: an approach to support causal multicast in large-scale systems. Technical report, IST - INESC, Lisboa, Portugal, 1994.

[18] A. Schiper and A. Sandoz. Uniform reliable multicast in a virtually synchronous environement. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 561–568, Pittsburgh, Pennsylvania, USA, May 1993.

[19] Andre Schiper and Aleta Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing*, pages 534–543, Toulouse, France, June 1993. IEEE.

[20] Marc Shapiro. Binding should be flexible in a distributed system. In *Proceedings of the International Workshop on Object-Orientation in Operating Systems (IWOOOS)*, December 1993.

[21] Mark D. Wood. Replicated RPC using amoeba closed group communication. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 499–507, Pittsburgh, Pennsylvania, USA, May 1993. IEEE.