Construction and management of highly available services in open distributed systems

To cite this article: Christos Karamanolis and Jeff Magee 1998 Distrib. Syst. Engng. 5 29

View the article online for updates and enhancements.

Related content
- High-performance asynchronous atomic broadcast
  Flaviu Cristian, Shivakant Mishra and Guillermo Alvarez
- Supporting customized failure models for distributed software
  Matti A Hiltunen, Vijaykumar Immanuel and Richard D Schlichting
- Consul: a communication substrate for fault-tolerant distributed programs
  S Mishra, L L Peterson and R D Schlichting

Recent citations
- Client-access protocols for replicated services
  C.T. Karamanolis and J.N. Magee
Construction and management of highly available services in open distributed systems

Christos Karamanolis† and Jeff Magee‡

Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK

Received 7 March 1997

Abstract. This paper addresses the problem of constructing and managing highly available services for large, open distributed systems. A novel replication protocol is presented that satisfies two fundamental requirements of this environment. First, it hides replication from the service clients and secondly, it facilitates the dynamic reconfiguration of the server group. The protocol has been implemented and tested in the Regis distributed environment. The experimental results indicate that the cost of replication transparency is acceptable.

1. Introduction

The increased reliance that we place on computing systems for many aspects of day-to-day life suggests that improving the availability of the services provided by these systems is an important research issue. The availability of a service is defined to be the probability that the service is provided correctly at a specific moment in time. We adopt the State Machine approach of Lamport [1] and Schneider [2] as a general method for implementing highly available services by means of software replication. Replicas of the servers that provide a service are located on different physically distributed machines. The approach supports a client–server interaction model and dictates the requirements both for client–service communication and for inter-server coordination.

This paper is specifically concerned with the provision of highly available services in large-scale open distributed systems. In this type of system, a service is typically provided by a relatively small and stable group of long-lived servers and is used by a large, dynamic changing set of short-lived clients. Examples of applications that comply with this system model are distributed system name services, switchboard services in telecommunication systems and file-system services provided over wide-area networks.

Replication in the above environment raises a number of important issues and requirements.

- Scalability. The replication protocol should scale well even if the service is used by large dynamically changing sets of clients.
- Transparency. Clients cannot necessarily be reprogrammed to accommodate replication-aware communication stubs. Replication should thus be transparent to the service clients. Moreover, the replication protocol should be compatible with typical non-replicated client–server interaction primitives.
- Management of availability. Dynamic replacement of non-replicated servers by groups of replica servers is required in order to transparently improve service availability. Another requirement is the long-term maintenance and management of the server group, by adding, replacing or removing servers, in a way that guarantees state consistency and causes minimal service interruption.

Systems that provide reliable multicast communication, such as ISIS [3], Horus [4] and Transis [5], can be used directly to implement a ‘closed’ group approach to replicating servers. In this approach, clients are either full members or ‘special’ members of the server group. This is a straightforward solution to the problem of providing the required atomicity and ordering properties for service request and service reply delivery. It also permits group reconfiguration. However, previous work [30] has indicated that the ‘closed’ group model does not scale well for large, dynamic client sets. It can result in severe performance degradation for message delivery times and group reconfiguration operations.

An alternative approach, known as the ‘open’ group model, has been adopted by Consul [6], Delta-4 [7], and Lazy Replication [8]. In this approach, clients are external to the server group. A client communicates with one of the replica servers, which acts as the representative of the client by forwarding its requests to the other servers in the group. All these systems require that clients have replication-aware...
communication stubs for accessing the replicated service. Our objective, as mentioned above, is to provide replication in a way that is transparent to clients.

Existing systems which support the provision of replicated services tend to be weak in their support for configuration management of the replica group. While [3, 5, 6] addressed the problems of dynamic configuration, this is restricted to failures, partitions, joins and mergers at the group communication level. Replica state consistency in the case of group reconfiguration is left as an application level concern. Cristian and Mishra [9] described the internal consistency constraints of a policy-driven Availability Manager, however, they did not address replica resynchronization in the dynamically reconfigured group.

In the following section, the problem of providing highly available services by means of replication is concisely defined, and the fundamental requirement of the State Machine approach for state consistency between clients and service is analyzed. In section 3, a replication protocol is proposed to satisfy the requirements identified above for large, open distributed systems. In particular the protocol:

- provides server replication transparently to clients. The protocol is designed to cope with large, dynamically changing client sets;
- supports on-line management of the server group without interruption to service provision.

The protocol is described in two parts. The first part outlines replica server synchronization without regard to group changes. The second part addresses the problem of group reconfiguration due to replica failures/removals and additions. The combination of higher-level communication primitives, such as Remote Procedure Call (RPC) and Remote Method Invocation, with the proposed replication protocol is discussed in section 4. The replication protocol is evaluated in comparison with experimental performance results in section 5. Section 6 summarizes the results of the paper and presents conclusions and directions for future work.

2. Providing highly available services

The State Machine approach to replication assumes that state transitions and the output of a server are completely determined by the sequence of requests it processes, independently of time or any other activity in the system. When replication is introduced to improve availability, the main non-deterministic event that must be synchronized among the replicated servers is the delivery of client requests. The State Machine approach has two requirements for internal state consistency [2]:

**Agreement.** If non-faulty replica \( p \) delivers request \( r \), then replica \( q \) eventually delivers \( r \) or \( q \) is faulty.

**Order.** If non-faulty replica \( p \) delivers request \( r \) and after that \( p \) delivers request \( r' \), then replica \( q \) does not deliver \( r' \) unless it has already delivered \( r \), or \( q \) is faulty.

† All the requirements presented here refer to client requests invoked to a specific replicated service. By the term ‘replica’, we refer to a replica server of that service.

Informally, the latter two requirements state that all non-faulty replicas deliver the same set of client requests and they deliver them in the same relative order. As far as the overall system state consistency is concerned, two more requirements must be satisfied.

**Causality.** The delivery order of client requests must respect their potential causal relations.

- If client \( c \) invokes request \( r \) and after that it invokes request \( r' \), then replica \( p \) does not deliver \( r' \) unless it has already delivered \( r \), or \( p \) is faulty.
- If request \( r \) of client \( c \) causally precedes request \( r' \) of client \( c' \), then replica \( p \) does not deliver \( r' \) unless it has already delivered \( r \), or \( p \) is faulty.

**Uniformity.** If replica \( p \) (whether non-faulty or faulty) produces the output related to request \( r \) (e.g. reply to client), then replica \( q \) eventually delivers \( r \) or \( q \) is faulty.

The causality requirement is needed so that replicated service behaviour is compatible with non-replicated service provision. In most cases, clients adopt a synchronous style of communication waiting blocked (interacting neither with the service nor amongst themselves) to deliver back a reply to their last request. For synchronous communication, inter-client consistency is trivially guaranteed. When clients adopt an asynchronous style of interaction with the service and inter-client consistency is important, then request messages must be time-stamped and these times must be respected by the delivery order in the servers [1]. We assume synchronous client–service communication, such as RPC, throughout this paper.

The uniformity requirement is of importance in systems where the membership of the replica server group changes dynamically [10]. It states that, if output is produced by the service as a result of processing request \( r \), then the results of \( r \) persist on the state of the service. For example, consider the scenario in which request \( r \) of client \( c \) is received and delivered by replica \( s \) of service \( S \). The server processes \( r \) and produces a reply \( r' \) which is sent back to \( c \). Then \( s \) crashes and due to a combination of communication failures no other server of \( S \) has the chance to receive and deliver \( r \). The state of service \( S \) does not reflect the results of request \( r \) and is inconsistent with the state of the client \( c \) (although the surviving servers have mutually consistent states). In this scenario, the agreement requirement is satisfied since \( s \) has failed but the uniformity requirement is not satisfied.

In this paper, we are concerned only with the transmission of replies back to clients, a special case of service output. The uniformity problems to be addressed in the case of general service output are in principle the same as the problems discussed here for the replies.

3. Replica synchronization protocol

This section describes a Replica Synchronization Protocol (RSP), that satisfies the state consistency requirements described in the previous section and, in addition, provides replication transparency to clients [11]. An overview of the protocol is presented followed by a more detailed
description of each of its components. The protocol is designed for asynchronous, message passing distributed systems, which exhibit benign failures. The benign failures being processors (and processes) which fail by crashing, and the communication network which fails by losing messages (omission errors). However, we assume a non-zero probability of message transfer by the network. Communication failures may result in system partitioning. The communication network is assumed to provide unreliable multicast such as Internet Protocol (IP) Multicast.

3.1. Overview

The RSP is implemented in the communication substrate of a replica server. The protocol receives and handles client requests; it synchronizes request delivery and output among replicas. As figure 1 depicts, client requests are transmitted to the server group using the communication network’s unreliable multicast. On receipt of a request \( r \) by the RSP layer, the server group makes a distributed decision as to which server replica will take the onus for synchronizing the delivery of \( r \) in the group. This replica generates a special synchronization message, which refers to \( r \). The synchronization message is communicated to the group by a group communication protocol (GCP) layer, which is also resident in the replica communication substrate. Figure 2 illustrates the structure of the communication substrate of a replica server.

GCP guarantees reliable, totally ordered, virtually synchronous delivery of synchronization messages in the group. It provides the RSP layer with a consistent membership view of the replica group. The delivery of a synchronization message indicates the logical time at which the client request it refers to is delivered to the application layer. Reliable delivery of synchronization messages permits replicas to detect lost client requests and to retrieve them from the rest of the group. The ordered delivery of synchronization messages guarantees that all replicas deliver client requests in the same relative order. The delivery properties of synchronization messages are used by RSP to satisfy the State Machine requirements.

Contemporary networking technologies are characterized by a low failure probability, even over large geographical distances. Consequently, we assume that a high percentage of the requests multicast by clients are received by all the replicas of a service. It is thus more efficient to reliably multicast synchronization messages between replicas rather than the request message itself. Synchronization messages are small, typically a fraction of the request size, and therefore delivery times through GCP are lower than the times for the requests. A request message has to be retransmitted in the group only when it has been lost by some replicas. Section 5 discusses the performance and scaling benefits of using synchronization messages.

The number and the identity of the replicas that transmit the results of a client request are specified by the output policy of the replicated service. A single output policy, only one replica transmitting the reply to the request, is sufficient in the presence of benign processor/process failures.

The RSP layer also synchronizes replica activity while group reconfiguration is taking place. Reconfiguration may be due to either server failures/partitions or management operations involving the explicit addition and removal of replica servers. The protocol guarantees uniformity in the case of replica failures, by exploiting the virtually synchronous model of communication provided by GCP. Server group management is executed by a policy-driven Availability Manager, which invokes management requests on the group. RSP accommodates a management module,
which handles addition and removal of replicas and controls the application specific state transfer algorithm.

The RSP layer interacts with the application layer through a typical server communication endpoint. This endpoint together with its counterpart on the client side implements a non-replicated client–service interaction protocol, such as RPC or Remote Method Invocation. We assume RPC in the rest of the paper. RSP implements the basic transport layer interface and provides a send primitive to the layer above for the transmission of replies back to clients. Client requests are delivered to the layer above by means of a deliver primitive. In addition, the RSP interface includes two replication specific hooks for state installation and transfer. Concrete implementations of the hooks depend on application semantics. The state installation and transfer primitives form the only elements that expose replication to the application algorithm. Apart from this, the server application algorithm is not affected by the introduction of replication. Table 1 outlines the RSP interface to the application layer above.

Section 3.2 outlines the properties of the GCP required by the design of the RSP. The rest of the section is divided in two parts. The first part describes the functionality of the RSP layer for synchronizing request delivery and reply transmission among replicas, in the presence of communication failures but without considering group reconfiguration. The second part addresses the problems introduced when replicas may fail and describes the protocol modules that handle server additions and removals.

For the discussion of the protocols, we assume that all the messages in the system (whether client requests/replies, or synchronization messages) consist of three fields <id, type, data>. The message id is a pair <sender, seqno>. The communication network’s unreliable datagram service provides two primitives for the transmission of messages:

• send(m,r) unicasts message m to the recipient with reference r in the system.

• multicast(m,g) multi-casts message m to the group with reference g.

Both primitives provide best-effort delivery semantics. The network dispatcher delivers messages by means of a standard deliver() upcall to the layers above.

3.2. Group communication protocol

The GCP required by the RSP is similar to the group-cast and reliable multicast protocols described in the literature [4, 5, 12–14, 31]. A protocol such as Horus [4], Transis [5] or RELACS [31] could be used to implement the functionality required of GCP. In the following, we use the term multicast to refer to a multicast initiated by RSP and implemented by GCP. Similarly, deliver refers to message delivery by GCP to RSP. The properties required of GCP specified below apply to messages multicast to all the members (replicas, in our case) of a group g by a member of group g, although we do not refer to the group explicitly.

Validity. If member p multicasts message m, then p eventually delivers m or p is faulty.

Agreement. If correct member p delivers message m, then member q eventually delivers m or q is faulty.

Integrity. For any message m, every member (whether correct or faulty) that delivers m, delivers it at most once and only if some member has previously multicast m.

Informally, GCP reliability states that all group members deliver the same set of messages and that they do not deliver any spurious messages.

• CP—total order. If correct members p and q both deliver messages m and m’, then p delivers m before m’ if and only if q delivers m before m’.

Group membership changes are recorded and agreed upon in GCP. Membership information is recorded in the form of membership ‘views’ (vectors of process identities). A message m is multicast (delivered) in view v by member p, when v is the last view installed in p before it multicast (delivered) m. For GCP to provide useful membership information, the contents of installed views must reflect the actual condition of the system as far as failures and/or partitions are concerned.

We assume the existence of a Failure Detector service [17] in the system, with a module on every processor. The failure detector is extended as shown in [18] to handle system partitions. It suspects and reports failed or unreachable members, without necessarily making complete and accurate suspicions. The membership service we have implemented for GCP is based on failure detector suspicions initiating membership agreement protocols. Babaoglu et al. [18] showed how to implement a membership service for partitionable asynchronous distributed systems using an eventually perfect (O(1)) failure detector. It is a theoretical impossibility to implement an eventually perfect failure detector in a completely asynchronous system. In practice however, by making timelines assumptions which are reasonable for actual systems, eventually perfect failure detectors can be implemented. The following specifications for GCP are based on [18] and imply a partitionable membership model.

• GCP—membership service.

View accuracy. If member q remains reachable from some correct member p, then eventually the current view of p will always include q (the latter also applies to members that join the group).

View completeness. If all members in some set P remain partitioned from the rest of the group or have voluntarily left the group, then eventually the current view of every correct member not in P will always exclude all members in P.

View integrity. Every view installed by a member p (whether correct or not) includes p itself.
View agreement. If a correct member \( p \) installs view \( v_i \), then for every member \( q \) included in \( v_i \), either: (i) \( q \) also installs \( v_i \), or (ii) \( p \) eventually installs an immediate successor to \( v_i \) that excludes \( q \).

View order. The order in which members install views is such that the ‘successor’ relation of views is a partial order.

The above specifications do not consider partition merging, since this is not required by the RSP, as discussed in section 3.4. Group communication support for partition merging is discussed in [32].

The partitionable membership model of GCP exhibits an important advantage. Since no membership agreement on a single primary partition is required, the design of GCP evades the impossibility result for primary-partition membership in asynchronous systems [19]. In other words, we are guaranteed that membership agreement protocols in GCP always terminate and eventually result in view installation.

Installed views are reflected on special view messages, which are delivered from GCP to the layer above in order to inform that layer about membership changes in the group. View messages are delivered as part of the normal message up-stream. In addition to the reliability properties above, GCP is required to satisfy the following view delivery requirements to provide virtually synchronous communication behaviour [13, 31].

- GCP—virtual synchrony
  1. If correct members \( p \) and \( q \) both deliver two successive views \( v_i \) and \( v_{i+1} \), then \( p \) delivers message \( m \) in view \( v_i \) if and only if \( q \) delivers \( m \) in \( v_i \).
  2. In some cases, GCP is required to satisfy a ‘stricter’ delivery property [16], as we will discuss in section 3.4.
  3. If member \( p \) multicasts message \( m \) in view \( v \) and at least one correct member delivers \( m \), then all correct members deliver \( m \) in \( v \).

GCP delivers messages (‘data’ or ‘views’) to the layer above by means of a \( \text{deliver()} \) upcall. Table 2 illustrates the interface of the GCP layer used by the RSP layer above it.

Message stability refers to the ‘global knowledge’ concerning delivery of messages in the group. More precisely, message stability in GCP is defined as follows:

- GCP—stability. A message \( m \) is considered to be stable in the group (partition), if \( m \) has been delivered by every member of the group (partition) and its delivery has been explicitly acknowledged by all members.

The definition means that message stability is determined by means of delivery acknowledgment down-calls from the application layer to the group communication protocol. A similar approach is also followed in Horus [4].

### 3.3. Replica synchronization

The RSP layer maintains the following local datastructures related to client requests:

- \( \text{requestBuffer} \): stores the received client requests for future delivery and for potential retransmission to other replicas.
- \( \text{requestRecord} \): records the identity of the last received and the last delivered request of every client that communicates with the replica; it is used to detect duplicate requests.

The RSP layer of each replica executes the same algorithm and uses variables with the same name. When naming ambiguity is possible during the description of RSP’s operation, the variable names are subscripted with the id of the local replica server.

In addition to the variables above, the RSP layer of replica \( r \) maintains the local view of the group membership, in the form of an ordered list of server identities:

\[
\text{view}_r = \langle \text{id}_1, \ldots, \text{id}_n \rangle
\]

where \( r = \text{id}_j \) with \( 1 \leq j \leq n \).

Clients transmit request messages to the service reference. In the case of a replicated service, this reference is a multicast address and the message is unreliable transmitted to all the replica servers of the service. Due to the unreliable nature of this multicast, some replicas may not receive a request sent by a client. On receipt of a request message \( m_r \) from client \( c \), the RSP layer of a replica \( s \) buffers \( m_r \) in \( \text{requestBuffer} \) and records \( m_r.\text{id} \) in \( \text{requestRecord} \) as the last received request of \( c \). RSP then decides whether \( s \) takes the responsibility for synchronizing the delivery of \( m_r \) in the group. This decision is made according to a distributed deterministic function, called the onus function. The onus function is applied to the group membership set and the unique message identity. The aim is to let at most one replica in the group take the responsibility for synchronizing a specific request.

#### 3.3.1. The synchronization onus function

A function \( \text{ONUS}(m, \text{id}, \text{view}) \) is an acceptable onus function, if for
any two replicas \( p \) and \( q \) in a service group with \( \text{view}_p = \text{view}_q \) and \( m_p, \text{id}=m_q, \text{id} \), we have:

\[
\text{ONUS}(m_p, \text{id}, \text{view}_p) = \text{ONUS}(m_q, \text{id}, \text{view}_q) = r,
\]

where \( r \in \text{view}_p \) (and \( r \in \text{view}_q \)).

**Example.** \( \text{ONUS}(m, \text{id}, \text{view}) \) returns \( p:p = \text{id}_i \)

where \( \text{id}_i \in \text{view} = \langle \text{id}_1, \ldots, \text{id}_n \rangle \) and \( i = m.\text{id} \cdot \text{sender mod length (view)} \).

Knowledge about the geographic vicinity of a client \( c \) to a replica \( s \) could also be used as a criterion to assign the responsibility for all the requests of \( c \) to replica \( s \). This might be used as a way of optimizing use of network resources. The onus function implements the Synchronization Policy [15] of the service. A fair distribution of the synchronization responsibility among replicas implements an Active Replication policy. A function that always allocates the onus to a single replica in the group implements a form of Primary-Backup Policy. In the latter case, synchronization messages could be batched by delaying the multicasting to cohort replicas. Combinations of the two policies are also possible according to the onus function employed. The introduction of the onus function in the design separates the concerns of the Replication protocol (state consistency) from those of the actual Synchronization policy used. In this sense, the discussion that follows applies to both Active Replication and Primary-Backup approaches (and hybrids). For the sake of brevity, we refer only to Active Replication in the following.

### 3.3.2. Synchronizing request delivery

The single replica \( s \), that takes the onus for a request \( m_r \), generates a synchronization message \( m_s \), which references \( m_r \); that is, \( m_s, \text{data}=m_r, \text{id} \). This synchronization message is reliably multicasted to the service group through the GCP layer. The delivery of \( m_s \) from GCP to RSP, in a replica (including \( s \) itself), indicates the logical time at which request \( m_r \) must be delivered to the application layer (through the RPC endpoint). Specifically, when \( m_s \) is delivered from GCP to RSP in replica \( s \) and the referenced request \( m_r \) is locally buffered in \( \text{requestBuffer}_s \), then \( m_r \) is delivered to the application layer. Thus, the Total Order of GCP implies the Order property for request delivery to the application.

Appendix A.1 gives pseudocode description of the RSP module that receives and handles client requests. Figure 3 depicts the messages exchanged for delivery and output (reply) synchronization, including the case of communication failures described in the following paragraphs.

### 3.3.3. Communication failures

A server \( s \) may deliver, from GCP, a synchronization message \( m_s \), which references a client request \( m_r \), that is not locally buffered in \( \text{requestBuffer}_s \). This may occur because the transmission of \( m_r \) to \( s \) is lost or delayed by the communication network.

In this situation, the RSP layer of \( s \) detects that \( m_r \) is not present locally and its retransmission is required from other replicas. Either the unicast or the multicast primitives of the unreliable datagram communication substrate can be used for this purpose. Using unicast, the request is made to a replica \( w \) which \( s \) believes to have the onus for \( m_r \). Since a synchronization message has been multicast through GCP for \( m_r \), there must be such a replica and it must already have received \( m_r \). If \( w \) (RSP layer) has taken the onus for request \( m_r \), it retransmits the locally buffered copy to \( s \).

Using multicast, the request for the retransmission of \( m_r \) is multicast by \( s \) to the whole group. Any replica (RSP layer) that receives a retransmission request for an \( m_r \) which it has buffered locally, retransmits \( m_r \) to \( s \), even if it does not have the onus for \( m_r \). Replica \( s \) waits for the first retransmission of \( m_r \) from any replica in the group.

In both cases, the retransmission of a client request in the group has best-effort semantics. Thus, \( s \) may have to repeat the retransmission request until \( m_r \) is received either from some other replica, or directly from the client (in the case of delayed transmission). Due to the network’s non-zero probability of delivering messages correctly, \( s \) will eventually receive \( m_r \). Our implementation of GCP uses multicasting for the retransmission request, since, as explained later, it improves the fault-tolerance of the protocol. Appendices A.1–A.3 list the pseudocode for request delivery in the RSP protocol.

While the RSP layer of a replica is waiting for the retransmission of a client request \( m_r \), client requests can be still be received. These requests are buffered and recorded locally and the corresponding synchronization messages are multicast through GCP if the replica takes the onus for their synchronization. However, no client requests are delivered to the application. The next request message to be delivered to the application must be \( m_r \).

Due to the unreliable communication network, a replica may miss a client request for which it would have taken the synchronization onus. Consequently, if a replica \( w \) receives and buffers a client request \( m_r \) for which it does not take the synchronization onus, it sets a timer on \( m_r \). If a timeout occurs in \( w \) without any corresponding synchronization message having been delivered from GCP, then \( w \) unreliablely remulticasts \( m_r \) to the group. This is periodically repeated until \( w \) delivers a synchronization message for \( m_r \). The non-zero probability for correct message transmission in

### Table 2. The interface of GCP to the layer above.

<table>
<thead>
<tr>
<th>G_mcast(m):</th>
<th>Multicast message ( m ) to the group.</th>
</tr>
</thead>
<tbody>
<tr>
<td>G_ack(id):</td>
<td>Acknowledge delivery of message with identity ( id ).</td>
</tr>
<tr>
<td>G_stable(id):</td>
<td>Check for stability of message with identity ( id ); ( m ) has already been delivered.</td>
</tr>
<tr>
<td>G_join(g):</td>
<td>Add caller to the group with reference ( g ).</td>
</tr>
<tr>
<td>G_leave():</td>
<td>Remove caller from the group it currently joins.</td>
</tr>
</tbody>
</table>
the communication network implies that the replica with the onus for \( m_r \) eventually receives it.

Client requests retransmitted within the replica group are received and handled in the same way as the original client request messages. RSP does not differentiate between original and retransmitted requests. Duplicate messages (received more than once) are detected and discarded in RSP using the requestRecord datastructure.

The reliable delivery of synchronization messages through GCP is used to achieve the Agreement requirement for request delivery to the application layer. We have discussed how synchronization messages are used for the detection and eventual retransmission of lost/delayed client requests. As a result, all replica servers deliver the same set of client requests, even in the presence of communication failures. Since request delivery is blocked while a replica is waiting for the retransmission of a missing request, GCP Total Order implies the Order requirement for client requests, even in the presence of communication failures. In conclusion, the proposed RSP is robust with respect to benign communication failures.

We have not yet considered the case of a client request being missed by all servers’ RSP layers in the group. This is equivalent to loss of the request message in the non-replicated case. Such failures are handled by the client–service protocol. Section 4 discusses the operation of a typical non-replicated, reliable client–service interaction protocol, on top of the proposed RSP layer.

### 3.3.4. Garbage collection in RSP

We have seen that a replica may have to retransmit a received client request to the replica group. Consequently, messages are stored in requestBuffer, even after they are delivered to the application layer. For the protocol to be practical, requests which do not need to be retransmitted in the future must be garbage-collected. A buffered request message \( m_r \) can only be safely removed from the requestBuffer of replica \( s \) when it is known that all replicas in the group have already received \( m_r \).

This ‘global knowledge’ is achieved using information about the stability of synchronization messages, in the GCP layer. The RSP layer of replica \( s \) acknowledges the delivery of synchronization message \( m_g \) only after the referenced request \( m_r \) is locally present in \( s \). Therefore, stability of \( m_g \) in the group implies that every replica has received (in fact, delivered) the corresponding client request \( m_r \).

After the delivery of synchronization message \( m_g \) is acknowledged in replica \( s \), \( m_g.id \) is buffered together with request \( m_r \) in requestBuffer. RSP periodically traverses requestBuffer and removes every request with a corresponding stable synchronization message. That is, if there is an \( m_g.id \) buffered together with request \( m_r \), then \( G_{stable}(mg.id) \) is called to check for stability of \( m_g \) in the group (see appendix A.4).

### 3.3.5. Synchronizing group output

While processing a client request, the application protocol in every replica server sends a reply back to the client, through the reply()
primitive of the server RPC endpoint. As a result, the `send()` primitive of the RSP layer is invoked, in every replica. However, the RSP layer of just one replica actually transmits the reply back to the client (see the message diagram of figure 3). The decision, as to which replica transmits the reply, uses a similar mechanism to that used for allocating the onus for delivery synchronization. The replica that transmits the reply back to the client may be the same replica that multicasts the synchronization message for the request, or it can be another replica in the group.

Due to the unreliable nature of the communication network, the reply transmitted to the client from the chosen replica may be lost. However, replies may be lost in the communication network, even in the non-replicated case. Section 4 discusses the way lost replies are handled by the client–service interaction protocol (e.g. RPC), when reliable communication is required. The RSP guarantees that at most one reply is transmitted from the server group back to the client.

### 3.4. Server group reconfiguration

The dynamic reconfiguration of the server group can occur for two reasons.

1. **System changes**, namely processor failures, process crashes or system partitioning.
2. **Management operations**, such as addition/removal of replica servers to/from the group.

We assume the existence of a policy-driven Availability Manager in the system [9]. The manager carries out the dynamic configuration management of a server group by means of management requests to the service, namely: `retrieve-membership`, `add-server`, `remove-server`. The Availability Manager must itself be replicated guided by an ad hoc policy, such as locating a manager replica at each server site.

The RSP must guarantee the requirements for state consistency in the group are satisfied despite dynamic group reconfiguration. The main difficulty here stems from the inherently asynchronous nature of the events that cause group reconfiguration, whether system changes or manager requests. To address this problem, we require that reconfiguration events are interpreted in some pseudosynchronous way with respect to the delivery and processing of client requests. In particular, the decision about the onus for synchronizing the delivery of requests (or the transmission of replies) is a function on the group membership set, as it is perceived by any individual replica (data structure `view` in RSP). Therefore, it is important that all significant events in RSP, namely request delivery and `view` updates, appear as if each of them occurred at the same logical time in all replicas.

Consequently GCP supports a virtually synchronous model of communication, as specified in section 3.2. Updated membership information is reflected in `view` messages, delivered from GCP to RSP as part of the normal up-stream of synchronization messages. The contents (vector of identities) of the data structure `view` in RSP is installed by the data field of these `view` messages.

In section 3.3 we noted that unreliable multicast rather than unicast is used to request the retransmission of lost client requests among replicas. The reason is the potential failure of the replica `p` which originally took the onus for synchronizing the delivery of a request `mr`. If, on delivery of the corresponding synchronization message `mg`, replica `q` finds out that `mr` is missing, it multicasts the retransmission request to the whole group, since `p` may have crashed in the meantime. In the worst case, no correct replica has received `mr`, and `q` time-outs without receiving `mr`. The corresponding `mg` is then discarded by all correct replicas in the group.

#### 3.4.1. Handling partitions

As discussed in section 3.2, GCP supports a partionable membership model, in order to achieve non-blocking membership protocols in asynchronous systems. However, we follow the Primary Partition approach [20] in the design of the RSP layer. In case of system partitioning, the RSP layers of different replicas deliver partitioned views of the group. The replicas in at most one of the partitions continue providing the service. This can be the majority partition (if any), or the partition containing the replica with the smallest id in the previous view. The RSP layer of a replica in a non-primary partition terminates the server. A replica can join the group of servers providing a service (join at the level of abstraction of the replication protocol) only as a newly joining server. In this way, the service is provided by a unique, totally ordered sequence of primary replica partitions.

We have used the Primary Partition approach since it does not require application participation. Partitionable models require application specific protocols for state merging, when partitions merge or crashed replicas recover. The adoption of the Primary Partition approach in RSP means that the impossibility results for termination in this model now apply to the Replication protocol. In particular, service provision may be blocked if there is no primary partition formed in the system. We delegate the solution of this problem to the Availability Manager (which also partitions). There, a heuristic decision can be made (possibly requiring the operator’s intervention) about which partition should be augmented with new servers to become operational.

#### 3.4.2. Safe output

The RSP as described in the previous section does not yet guarantee client–service consistency in the presence of server failures. In order to satisfy the uniformity requirement, the RSP layer supports the following safety property.

- **Safe output.** Replica `p` with the onus to transmit the reply to request `mr` sends the reply message only after the corresponding synchronization message `mg` becomes stable in GCP.

Stability of `mg` in the group implies that all correct replicas in the group have received request `mr`. Therefore, if a reply to request `mr` is transmitted back to the client, then all correct replicas in the group have delivered `mr`. This does not guarantee that a reply is always sent to the client. It is possible that the replica with the onus to send the
Reply for \( m_v \) crashes before the reply is actually transmitted. However, we are guaranteed that if a reply is transmitted, then the results of the corresponding request persist on the service state. Reliable reply transmission is implemented by the client–service protocol if required.

3.4.3. Handling manager requests. Manager requests are received and handled in a similar way to client requests. To simplify the description, in the following we do not consider the effects of lost manager requests and replies. We assume that a reliable manager–service communication channel is established in the same way as client–service communication (section 4). Unless otherwise stated, the term ‘replica’ is used to refer to the RSP layer of a replica server.

Membership set retrieval. In the simple case of the manager requiring the current membership set (request \( \text{retrieve-membership} \)), a single replica \( p \), decided by the onus function, replies with the current value of \( \text{view}_p \). Although this view may not be the most up-to-date (other replicas in the group may have already delivered view messages that \( p \) has not yet delivered), it is still a consistent view. The manager can retrieve another view with a future request. If group membership changes concurrently with request arrival, then more than one replica may take the onus to reply to the manager. Only the first reply will be accepted by the manager. The \( \text{retrieve-membership} \) management request is not delivered to the application and, therefore, no multicast through GCP is required to synchronize replica activity.

Server removal. A manager \( \text{remove-server} \) request (figure 4) is discarded by all but the replica \( p \) that is to be removed. The RSP layer of \( p \) calls the \( \text{G-leave}(\) primitive of GCP to initiate a membership agreement protocol for the removal of \( p \) from the group. \( \text{G-leave}(\) blocks \( p \) waiting for the membership protocol to terminate. When it resumes execution, the RSP layer of \( p \) sends a reply to the manager and terminates the replica server.

Server addition. The \( \text{add-server} \) request is multicast to all system nodes where the program module of the replica server resides (or can be dynamically loaded). Using the dynamic component instantiation primitives of the run-time support system (see section 5), the request triggers the instantiation of a new replica server on the specified node.

The instantiation procedure of the RSP layer invokes the \( \text{G-join}(\) primitive of GCP, in a new replica \( p \) (figure 5 and appendix B.2). The call is parametrized with the reference of the service group and initiates a membership agreement protocol in GCP to incorporate \( p \) in the group. As a result, a view message \( m_v \) is delivered from GCP to RSP, in all replicas including the newcomer \( p \). All replicas install a new view that includes \( p \) (and potentially other members). On delivery of \( m_v \), a single replica among the ‘old’ group members is chosen, in a distributed deterministic way, to send a message with the current (i.e. in the context of the view message) service state to the newcomer. The state message, denoted \( m_n \), contains two parts.

1. The application state retrieved from the application layer by calling the \( \text{retrieve-state}(\) procedure.
2. The RSP state: a set of the identities of the last delivered request of every client, recorded in \( \text{requestRecord} \); the set is denoted \( \text{last-ids} \).

The contents of \( \text{requestBuffer} \) do not have to be sent as part of the RSP state. If there are client requests that have already been received and buffered by existing replicas but not delivered to the application yet, then they will be detected by the RSP layer of \( p \) on delivery of the corresponding synchronization messages from GCP (following \( m_v \)). These client requests, that \( p \) has missed, are retransmitted to it on demand. Similarly, messages that must be synchronized in the new view by \( p \) and are buffered in some of the ‘old’ members are retransmitted to the group after a time-out period.

The state message \( m_v \) is transmitted using the unreliable multicast of the communication network. Every replica RSP layer that receives this message buffers it locally together with the related \( m_n, \text{id} \). However, the state message is only used by the newcomer.

After \( \text{G-join}(\) returns, the first message to be delivered from GCP to the RSP layer of \( p \), is the view message \( m_v \), which is used to initialize \( \text{view}_p \). RSP then blocks waiting to receive \( m_n \). In other words, \( m_n \) plays also the role of the synchronization message for \( m_v \). Since \( m_n \) is transmitted by unreliable multicast, \( p \) may time-out waiting for it. The retransmission of \( m_n \) is then requested from the group, in the same way as for missed client requests. If the replica that originally transmitted the state message \( m_v \) fails, then any other replica in the group that has buffered \( m_n \) retransmits it to \( p \). However, there may be scenarios where all the replicas that received and buffered \( m_n \) fail and no replica can retransmit \( m_n \) to \( p \). In that case, the instantiation procedure of \( p \) cannot be completed and \( p \) is terminated before it starts participating in the group’s activity. It is the manager’s responsibility to try a new \( \text{add-server} \) operation.

When replica \( p \) receives the state message \( m_n \), the state of \( p \) is initialized from the contents of the message. The application state is initialized by calling the programmer-defined \( \text{install-state}(\) procedure. RSP’s \( \text{requestRecord} \) is initialized from the \( \text{last-ids} \) part of \( m_n \). Thus, if \( p \) receives, in the future, a request that had already been delivered and processed before \( p \) joined the group, then this request will be detected as duplicate and will be discarded.

If \( m_n \) indicates that \( p \) is the first server in the group, no state message is expected in RSP. Application state is initialized with whatever is defined as the default initial value in \( \text{install-state}(\) procedure. \( \text{requestRecord} \) and \( \text{requestBuffer} \) are initialized with null contents. In any case, unless state initialization is completed, the RSP layer of the new replica does not start receiving and handling client (and
As soon as a new view is installed, the RSP layer of every replica in the group re-evaluates the onus for every non-delivered (to the application) client request in requestBuffer. Synchronization messages are multicast through GCP accordingly. Appendix B.1 gives a pseudocode description of the RSP module delivering view messages from GCP.

3.4.4. Why strict virtual synchrony? It may not be clear, from the preceding discussion, why we require a strict virtually synchronous behaviour from GCP. To clarify this point, consider the following scenario. Assume that a request $r$ is received by replica $p$ in view $v_i$ and that $p$ happens to take the onus for $r$’s synchronization in the membership set of $v_i$. The RSP layer of $p$ is thus multicasting a synchronization message $m_p$ through GCP in view $v_i$. Note, that the ‘weak’ version of virtual synchrony guarantees that all group members deliver $m_p$ in the same view, but this may be any view $v_{i+k}$ ($k \geq 0$).

Let $q$ be a new replica server that joins the group concurrently to the arrival of $r$, causing view $v_{i+1}$ to be installed. Request arrivals are asynchronous with respect to the internal group activity. Therefore, replica $q$ may receive request $r$ in some view $v_{i+m}$ ($m > 0$), before it delivers the corresponding synchronization message $m_q$. Assume that according to the onus function, $q$ should take the onus of $r$’s synchronization in $v_{i+m}$. In this case, the RSP layer of $q$ does not have enough information to decide whether it should transmit a synchronization message or if such a message has already been transmitted in the group (but is not delivered yet). Multiple synchronization messages for the same request could lead to the delivery of the request in inconsistent order by different replicas.

Given a weak virtually synchronous behaviour by GCP, the RSP layer of $q$ cannot make a decision unless it knows what synchronization messages have been multicast to the group in earlier views but have not been delivered yet.
The latter information could only be transmitted to the newcomer \( q \) by means of a reliable multicast through GCP leading to the same problems as with the synchronization messages.

For these reasons, GCP is required to support a strict virtually synchronous model of communication (see section 3.2, GCP—virtual synchrony, property 2). According to this property, a message is delivered in the same view as the one it has been multicast in. To achieve this behaviour, the membership protocol of GCP has to block new multicasts in the group while a membership agreement protocol is taking place (until all messages originated from correct senders of the previous view become stable in the group). The characteristic results in a performance overhead for request delivery, in the presence of group reconfiguration [16]. However, it is not considered as a serious drawback in our model, where server groups are assumed to be small and fairly stable and reconfiguration is infrequent.

4. Reliable client–service communication

We assume that the RSP layer accepts and delivers messages to the application via a standard communication endpoint. This endpoint, together with a compatible endpoint at the client, implements a non-replicated, synchronous client–service (request-reply) protocol such as Remote Procedure Call (RPC). The RSP layer is designed so that replication is transparent to the client–service protocol, by supporting the behaviour and the interface of the underlying communication network. RPC protocols and in object-oriented systems Remote Method Invocation (RMI) can be built on either the unreliable datagram protocols provided by the network such as UDP or reliable connection oriented protocols such as TCP. Efficient RPC implementations generally use the unreliable datagram protocols and handle message loss and retransmission within the RPC protocol. RSP is most easily interposed in this sort of RPC implementation.

In the simple case of best-effort RPC, no changes are required to the RSP layer of section 3. If the client receives a reply, then it is guaranteed that the corresponding request has been delivered by the service; if no reply is received, the client can make no assumptions about the delivery of the request.

However, most existing systems incorporate RPC protocols that provide exactly once delivery guarantees [21] (also known as reliable client–service protocols). In this case, the operation of the client and server endpoints can be summarized to the following.

- **Client endpoint.** Time-outs and retransmits last request if no reply is received; detects and discards duplicate replies.
- **Server endpoint.** Buffers the reply to the last request of every client; detects duplicate requests; if a duplicate of the last request of a client is received, then the reply to that request is retransmitted (if available from the application).

To guarantee exactly one reply to the client, the RSP layer cannot always ignore duplicate client requests. If the RSP layer of a replica server \( p \) receives a duplicate request \( m_r \) from client \( c \) (detected using the contents of requestRecord\( _p \)), then:

- if \( m_r \) is earlier than the last received request of \( c \), then \( m_r \) is discarded;
- if \( m_r \) is the same as the last received request of \( c \), but \( m_r \) has not yet been delivered to the application, the duplicate \( m_r \) is discarded;
- if \( m_r \) is the same as the last received request of \( c \) and it has already been delivered to the application, then \( m_r \) is delivered again, without any synchronization procedure being employed (multicast of synchronization message through GCP).

In the latter case, the server endpoint ‘catches’ the duplicate request. If a reply \( m_r' \) to that request is already buffered there, then its retransmission is attempted calling send\( _m (m_r', c) \) to the layer below (RSP). The RSP layer of replica \( p \) actually transmits \( m_r' \) only if \( p \) takes the onus for transmission in the current group view. Since no synchronization takes place for the second delivery of \( m_r \), more than one replica may retransmit the reply. However, in this case the client endpoint can handle duplicate replies anyway.

5. Protocol implementation and evaluation

The RSP and the GCP have been implemented as part of the Regis distributed programming environment. The first part of this section describes the Regis framework for protocol development. The second part compares, in terms of performance, the RSP with a Replication Protocol that does not provide replication transparency to clients.

5.1. Implementation in Regis

Regis [22] is a programming environment aimed at supporting the development and execution of parallel and distributed programs. It embodies a constructive approach to the development of programs based on separating program structure from computation and communication. The latest version of the system [23] incorporates a flexible communication system, which facilitates the use of different protocols according to the needs of the application (style of communication, quality of service requirements) and the system model (transport layer). The system offers a range of built-in communication primitives, but also provides programmers with a framework in which to develop their own models of interaction.

The cornerstone of the system’s design is the concept of the protocol stack, which has been shown to simplify the development of communication protocols with negligible overhead [24]. Every protocol, in Regis, is realized as an aggregation of microprotocols, each one implementing a subset of the overall functionality. Context independence and hence reusability is obtained by requiring each microprotocol to conform to an abstract interface. The abstract microprotocol interface declares methods for passing data up and down the stack, as well as generic methods (parametrized with lists of name/value pairs) for control calls to the protocol below or exception calls to...
the layer above. Methods are also defined for connecting microprotocols during stack instantiation.

Data transfer between microprotocols is exclusively based on upcalls [25]. Each upcall sequence is assigned its own light-weight execution thread. Communication endpoints which provide synchronization with user-level threads are placed at the top of the stack, while drivers that interact with the operating system (or the hardware) are placed at the bottom of the stack. The communication endpoints also define the interaction style realized at the application level.

An established data path between two (or more) user-level components is supported by compatible protocol stacks at each participant. The stacks are instantiated as part of the binding procedure and are initialized with the references of remote endpoints/protocols. Regis supports a reference system which is independent from the specific transport layer. A reference contains information about the actual position of an endpoint in the system, as well as information about the protocol instance supporting the endpoint.

Regis supports dynamic stack construction at binding time; the required stack instance is inferred from the references that are passed around to establish the binding. Further, Regis supports dynamic reconfiguration of protocol stack instances: microprotocols can be introduced or removed at any time during the lifetime of a binding. Stack construction and reconfiguration is implemented by means of protocol factories, which employ demand-loading of microprotocol code modules when necessary.

The RSP and the GCP discussed in section 3 have been implemented in the form of collections of light-weight, reusable microprotocols. Figure 6 illustrates the protocol stack instances employed by clients and replica servers in our design. In the server, an RPC endpoint is placed on the top of the protocol stack providing the interface with the application program. This is the same endpoint that is used, directly on top of the transport layer dispatcher, in the non-replicated case.

In the replicated case, RPC is placed on top of a microprotocol module that implements RSP. Messages are delivered to the RSP module either directly from the transport layer dispatcher (client request messages, inter-RSP messages) or through that part of the stack that implements GCP (synchronization messages). GCP is implemented by a set of four subprotocols responsible for Reliable Causal Multicast, Total Order, Virtual Synchrony and message Stability.

Context independence of microprotocol design has been exploited in testing alternative implementations of some of the microprotocols in the presented stacks. For example, various Total Order protocols, namely token-passing and history based, have been developed and tested in different environments and with different applications.

5.2. Protocol evaluation

The flexibility of the Regis communication system has facilitated the development of alternative Replication protocols, from the literature, for comparison with the RSP. With reference to figure 6, these different protocols are realized as alternative implementations of the RSP sub-protocol, in the server protocol stack. The GCP part of the stack is reused.

In particular, we have implemented a variation of the ‘open’ replication protocols (clients external to the server group) of Delta-4 and Lazy Replication, for non-commutative client requests that require total delivery order. As noted in section 1, these protocols do not provide replication transparency to clients. Thus, the client protocol stack accommodates a replication-dependent communication stub (just under the RPC endpoint). The client’s replication stub resolves the multidestination address of a replicated service and binds to a specific replica server. This server acts as the representative of the client in the group: it receives the client’s requests and multicasts them to the rest of the group in a reliable and ordered way; it also transmits back the replies to the client. In case of server group reconfiguration, the clients must repeat the binding procedure.

The main performance drawback of the RSP of section 3 is the reply delay introduced to satisfy the Safe Output property. The replica with the responsibility to transmit the reply must wait for stability of the corresponding synchronization message in the group. In this way, we are guaranteed that if a reply is sent back to the client, then the results of the corresponding request persist in the service state, despite any server failures (the uniformity requirement of the State Machine approach).

In the case where clients accommodate replication-dependent communication stubs, as is the case with the alternative implementation, the expensive Safe Output can be avoided. Instead, the client stub buffers requests to the service even after a reply has been received back. When the service group is reconfigured due to server failures, then all the service clients are queried for potential-state inconsistencies—requests for which replies have been received by the clients, but which are not reflected on the service state (see the failure scenario of section 2). If there are such requests, they are retransmitted to the service by the clients’ replication stubs. The interested reader is referred to [30] for a detailed study of client-access protocols for replicated services.

The objectives of this section are twofold.

(1) To compare the two protocol classes (transparent and non-transparent replication) and investigate the actual performance price for satisfying the requirement for replication transparency.

(2) To investigate the performance scaling of RSP.

The experiments have been conducted on a network of SUN SPARC IPX workstations interconnected by a lightly loaded Ethernet. The OS kernel has been augmented to support Deering’s IP extensions for multicast [27], which are directly mapped on Ethernet’s hardware multicast. IP multicast has been used for the transmission of client requests to the service, in the case of RSP, and by the Reliable Causal Multicast layer of GCP. The experimental results presented should not be taken as absolute performance indexes for the protocols. No special attempt has been made to optimize the protocol
Construction and management of highly available services in open distributed systems

Figure 6. Protocol stack instance for replicated service provision.

Figure 7. Response latency. (a) RSP, (b) non-transparent replication.

Figure 8. Throughput for request/reply messages of 100 bytes. (a) RSP, (b) non-transparent replication.

performance and the machines used for the experiments are not high performance. The aim is to study the comparative performance of the two classes of protocols and to investigate scaling.

Figure 7 presents the response latency for the two protocols. The response latency is defined as the time elapsed, in a client (application layer), between invoking a request to the service and receiving back a reply. In these experiments, no delay has been introduced at the application layer of the servers. The times, in the graphs, are average values measured for static groups. They are presented against the number of replica servers in the group and the message sizes for requests and replies are assumed to be of the same. The latency time consists of two parts:

1. the time for client–service–client interaction (e.g. approximately 2.5 ms for requests/replies of 100 bytes);
2. the time for internal replica synchronization.

The results indicate that the non-transparent Replication Protocol provides better response times as predicted earlier in this section—no reply blocking is required in
this protocol to guarantee uniformity. However, the performance advantage becomes smaller for large messages and it is even reversed for large groups. RSP would appear to perform better for large messages and large server groups. The reason is that RSP utilizes small internal synchronization messages, which are independent from the size of the request messages. The non-transparent Replication Protocol reliably multicasts the actual request message within the group. This affects not only part (1) but also part (2) of the latency time: worse delivery times for larger messages in GCP. In RSP, the size of the request and reply messages affects only part (1) of the latency time, which is just a fraction of the overall invocation response time.

Figures 8 and 9 illustrate throughput performance results for the two protocols, for messages of 100 bytes and 1 Kbyte respectively. The throughput is defined as the total number of requests processed by the server group/s. Due to the synchronous communication style of the clients, the throughput is inversely proportional to the latency times, and the two protocols exhibit similar comparative performance results. The throughput of RSP is comparable with that of the non-transparent protocol and scales well for large client sets. We expect throughput to improve even more in favour of RSP in the case of dynamically reconfigurable server groups, since RSP does not require client rebinding when the group is reconfigured. Moreover, the throughput of the non-transparent protocol has been evaluated with an even distribution of clients to servers—a favourable environment for this protocol.

The best throughput results, for both protocols, are recorded for the trivial case of one-server group, where no internal synchronization is required. The experiments indicate that the service throughput is proportional to the number of clients. The measured throughput values converge to a maximum value, which depends on the message size and is imposed by the available Ethernet bandwidth. Note, for example, that in the case of one-server group, the throughput for 1 Kbyte messages does not exhibit a peak similar to that for 100 byte messages. The explanation lies in the available Ethernet bandwidth: a throughput of approximately 450 requests/s (see figure 9, 1 server, 16 clients) implies 900 messages/s on the network (that is, the request and the following reply messages); this is equivalent to 900 Kbytes/s, a value close to the 1007 Kbytes/s real bandwidth of a 10 Mbps Ethernet segment.

6. Conclusion

This paper has presented a RSP which co-ordinates replica server activity. The protocol has been engineered for large, open distributed systems, where a service is provided by a fairly small and static group of servers and is used by a large set of ‘occasional’ clients. The protocol allows replication to be provided transparently to the clients of a service. The RSP is generic in that it can be used with any underlying GCP that supports the properties described in section 3.2. In addition, it requires little modification to servers other than the need to install and transfer state.

The disadvantage of the protocol is that the server group must delay the client reply until the request becomes stable in the group, in order to ensure uniformity. This performance penalty has been previously noted in the Manetho system [28] where output commit to entities external to the group requires an expensive uniform agreement. Manetho addresses the problem by buffering group output on the recipients, to be used for reconstructing the state of the service in some failure scenarios. This is equivalent to buffering requests on the clients for potential future retransmission, as is the case with the non-transparent replication protocol described in section 5. Such solutions are not possible when replication must be transparent to system entities external to the group of servers.

The experimental performance results obtained in the Regis system demonstrate that the performance overhead due to the transparency requirement is not prohibitive for the practical use of the protocol. Moreover, this overhead becomes smaller for large request/reply messages (>1 Kbyte) and it is even eliminated for groups of more than three servers. Thus, it is clearly justifiable, in terms of performance, to use a protocol like RSP in environments where the transparency requirement is important.

RSP also facilitates the dynamic configuration management of the server group. Server removals (due to either manager requests or system failures) and additions are handled in the background, while the rest of the group provides the service. During reconfiguration, group activity is not ‘frozen’, as suggested in [9]. The virtually synchronous model of the GCP is exploited to achieve replica consistency even in the presence of membership changes. The
service is managed by an external policy-driven Availability Manager, by means of a set of special management requests. This separation of concerns has clearly simplified the design of the RSP.

The RSP of section 3 follows the Primary Partition model, allowing at most one partition of the server group to provide the service, at any time. We are currently investigating the problem of partition splits/mergers in the context of generic replication protocols. On the one hand, the properties required from the underlying GCP, and in particular from the Membership protocol, have to be specified. The protocol must be weak enough to be implementable in asynchronous systems, but also strong enough to give useful information about partitions and their dependencies (partial order), in the system. Existing research in the area [32] indicates that membership views must maintain some information about their context (views from which they have been formed) in order to facilitate state transfer protocols for different application classes. However, the behaviour of certain application classes during partitionable operation and consequent merging has to be defined and the corresponding state transfer protocols specified.

Currently, management of the replicated server group is explicitly performed by a designated Availability Manager component. The actual management procedure (enforcing a replication policy) is realized by the Manager’s computational behaviour. We are investigating the use of structural constraints [26, 29] as an alternative way of specifying the structure and evolution of large open distributed systems, where explicit management is not always feasible. Server replication is an architectural paradigm where this approach can be demonstrated: replica server components are required to configure themselves into the system, in such a way that they are consistent with the replication policy specified for the service they provide. That is, the number of required replica servers to provide the highly available service is a system constraint. Self-organization of system components can be implemented by means of local (in the components) configuration agents which are aware of the overall specification constraints, but which are still independent from the computational behaviour of the component.

Appendix A. Synchronizing request delivery in RSP—pseudocode description

Assume that the following protocol modules are executed in the RSP layer of replica p.

A.1. Receiving client requests

upon receive(m) do
  if m.tag = "client request" then
    if m.id in requestRecord then discard m;
    else record m.id in requestRecord;
    insert m in requestBuffer;
    set timer on m;
    if ONUS(m.id, view) = p then
      m := {tag:="client request",
      data:=m.id};
      G_mcast(m);
      if m.tag = "retransmission request" then
        if m' in requestBuffer with m'.id=m.data then
          multicast (m', g); /* unreliable */

A.2. Delivering synchronization messages from GCP

upon deliver(m_g) do
  if m.tag = "client request" then
    if m: m.id=m_g.data not in requestBuffer then
    m':={tag:="retransmission request",
      data:=m_g.data};
    do multicast(m', g); /*unreliable*/
    wait-for timeout;
    until receive(m);
    insert m in requestBuffer;
    record m.id in requestRecord;
    G_ack(m_g.id);
    deliver(m); /*to application through RPC*/
    save m_g.id with m in requestBuffer
    if ONUS(m.id, view) then
      enable output for m;

A.3. Retransmitting client requests

upon TIMEOUT for requestBuffer do
  for each m in requestBuffer do
    if not exists m_g.id with m in requestBuffer then/* m not
delivered to application yet */
    if m buffered for > time-limit then
      multicast (m, g); /*unreliable*/
      re-set timer for m in requestBuffer;

A.4. Garbage collection

upon TIMEOUT for requestBuffer
garbage-collection do
  for each m in requestBuffer do
    if exists m_g.id buffered with m in requestBuffer then/* m already delivered to the
    application */
    if G_stable (m_g.id) = true then
      /* is stable */
      remove m and m_g.id from requestBuffer;
      reset timer for garbage-collection;

Appendix B. Group reconfiguration in RSP—pseudocode description

Assume that the following protocol modules are executed in the RSP layer of replica p.
B.1. View delivery from GCP

upon G.deliver(mv) do
  if mv.tag = ‘view’ then
    if mv.data not primary partition
      then EXIT();
    for all q ∈ mv.data - view do
      if ONUS (q, mv.data) view = p then
        retrieve_state (mapply);
        last_ids := {m.id: m last
          delivered request for a
          client recorded in
          requestRecord};
        mstate := {tag:='state transfer',
          data:={mapply, last_ids};
        insert mstate in requestBuffer
        together with mv.id;
        multicast (mstate,g);
        view := mv.data;
        G.ack(mv.id);
      for each request m in requestBuffer not
        delivered yet do
        if ONUS (m.id, view) view = p then
          m_0 := {tag:='client request',
            data:=m.id};
          G.mcast (m_0);
  end;

B.2. RSP instantiation procedure

RSP.instantiate (g) :
  wait for G.join (g);
  G.deliver (mv);
  view := mv.data;
  if p not first member in view
    then do wait for receive(mstate);
     if TIMEOUT then
       m_0 := {tag:='retransmit state'};
       multicast (m_0, g);
     until received or > retries;
     if > retries then
       wait for G.leave();
       install_state (mstate.data.aappl);
       requestBuffer := null;
       requestRecord := mstate.data.last_ids;
     else /* first number in view */
       install_state (null);
       requestBuffer := requestRecord := null;
       G.ack (m_0);
     enable receive(); /* start receiving client
     requests */

References

[1] Lamport L 1978 Time, clocks and the ordering of events in
a distributed system Commun. ACM 21
[2] Schneider F 1990 Implementing fault-tolerant services
using the state machine approach: a tutorial ACM
Comput. Surv. 22
distributed computing Commun. ACM 36 37–53
flexible group communication system Commun. ACM 39
approach to high availability cluster communication
Commun. ACM 39
fault-tolerant replicated objects using Psync 8th Symp.
on Reliable Distributed Systems (Seattle, WA)
dependable distributed computing ESPRIT Research
Report 818/2252 vol I
Providing high availability using lazy replication ACM
Trans. Comput. Syst. 10 360–91
availability management in asynchronous systems 2nd
Int. Workshop on Configurable Distributed Systems
(Pittsburg, PA)
support dynamically configurable groups of servers 3rd
Int. Conf. on Configurable Distributed Systems
(Annapolis, MD)
Available Distributed Services 14th IEEE Symp. on
Reliable Distributed Systems (Bad Neuenhar)
and using context information in inter-process
communication ACM Trans. Comput. Syst. 7
Syst. 9
A fault-tolerant group communication protocol 15th Int.
Conf. on Distributed Computing Systems (Vanocoo, BC)
systems Commun. ACM 34
[16] Friedman R 1995 Strong and weak virtual synchrony in
horus Technical Report 95-1537 Department of
Computer Science, Cornell University
[17] Chandra T D and Toueg S 1996 Unreliable failure detectors
for reliable distributed systems J. ACM 43 225–67
group membership: specification and algorithms
Technical Report UBLCS-97-1 Department of
Computer Science, University of Bologna
impossibility of group membership 15th ACM Symp. on
Principles of Distributed Computing
[20] Ricciardi A and Birman K 1991 Using process groups to
implement failure detection in asynchronous
environments 10th Annual ACM Symp. on Principles of
Distributed Computing
development environment for parallel and distributed
systems 2nd Int. Workshop on Configurable Distributed
Systems (Pittsburg, PA)
[23] Pryce N and Crane S 1996 A uniform approach to
configuration and communication in distributed systems
3rd Int. Conf. on Configurable Distributed Systems
(Annapolis, MD)
architecture for implementing network protocols IEEE
Trans. Software Eng. 17 64–76
[25] Clark D 1985 The structure of systems using upcalls, 10th
ACM Symp. on Operating Systems Principles (Orcas
Island, WA)
Specifying distributed software architectures 5th Eur.
Software Eng. Conf. (Barcelona, Spain)
[27] Deering S 1989 RFC 1112: Host Extensions for IP
Multicasting (Xerox, RFC)

[31] Babaoglu O, Davoli R, Gianchini L. A and Baker M G 1995 Relacs: A communication infrastructure for constructing reliable applications in large-scale distributed systems 28th Hawaii Int. Conf. on System Sciences (Maui, Hawaii)