A Replication Protocol to Support Dynamically Configurable Groups of Servers

Christos T. Karamanolis Jeff N. Magee

Department of Computing,
Imperial College of Science, Technology and Medicine,
London SW7 2BZ, U.K.

Abstract

The paper addresses the problem of dynamic configuration management of highly available services. In particular, we are concerned with services that are provided by a group of actively replicated servers and are used by a large, rapidly changing set of clients. In this system model, we propose a Replication Management Protocol to facilitate the dynamic configuration management of the server group, while maintaining service state consistency. We argue about the correctness of the protocol and report on initial implementation results.

1 Introduction

With the ever increasing introduction of computing systems in many aspects of today's life, availability of critical computing services becomes of great importance. The *availability* of a service is defined as the probability that the service is provided correctly at a specific moment in time. Traditionally, availability is improved by employing groups of redundant (replica) *servers* to provide a *service* [5]. In this paper, we address the problem of availability in systems where a service is provided by a relatively small and stable group of servers, and is used by a large, fairly dynamic set of "occasional" clients. This model is typical of large, open distributed systems. Examples of applications that comply to our model are name services for large-scale distributed systems, switching-board services in telecommunication systems, and file-system services provided over wide-area networks.

It is not practical, in this environment, to consider clients as special members of the group of replicated servers, as is it is suggested in *ISIS* [2]. Rather, we consider clients as entities external to the group, in the sense that clients cannot participate in replica state synchronisation in any way. We wish to permit transparent and dynamic replacement of non-replicated servers by groups of servers. Consequently, we do not assume client communication stubs that are aware of replication, in contrast to systems like *Consul* [14] and *Delta-4* [16]. Neither we assume client stubs that install a communication channel with one of the replica servers, by monitoring the group membership, as it is the alternative model of *ISIS* [2].

The requirements for providing services in this environment have initially been analysed in [10], where a *replication protocol* is proposed to meet a range of client requirements, in line with the *State Machine* approach [17]. That paper also introduces the problem of the dynamic configuration management of the server group. *Dynamic configuration management* is necessary in order to replace failed server replicas, upgrade the server implementation by replacing existing servers, or change the (probabilistic) availability characteristics of a service by adding/removing servers to/from the group. The management is carried out by an *Availability Manager* [6], which invokes a set of configuration operations (in the form of special requests) to the server group. The way that a manager interprets a policy into a set of configuration operations is not of our concern.

In this paper, a *replication management protocol* is presented to coordinate the effects of manager requests to the replicated server group. The protocol facilitates the dynamic configuration management of the group, while it guarantees replica state consistency and causes minimum (if any) interruption to the service provision.

Existing research in the area [2, 11, 14, 13], does not explicitly consider the problem of the configuration management of the replica group. Even when dynamic reconfiguration is addressed [2, 14, 13], it is restricted to failures, joins, and mergers at the group communication level; no replica state consistency is taken under consideration. Further, Cristian and Mishra describe in [6] the internal consistency constraints of an availability manager, but they do not address general replica synchronisation problems in the dynamically reconfigured group.

2 Highly Available Services

We assume an *asynchronous* system in the sense that processors do not own synchronised clocks, there are no bounds on the time required for a process to execute a single step, and there is no known upper bound on message transmission delay over the communication network.

Processes are *deterministic*, and exhibit *crash* failure semantics. The communication network is *unreliable* (omission failures) and can deliver messages out of order; it supports a multicast primitive. The system is augmented with a *failure detector* [3] to circumvent the impossibility results for distributed consensus in this system model.

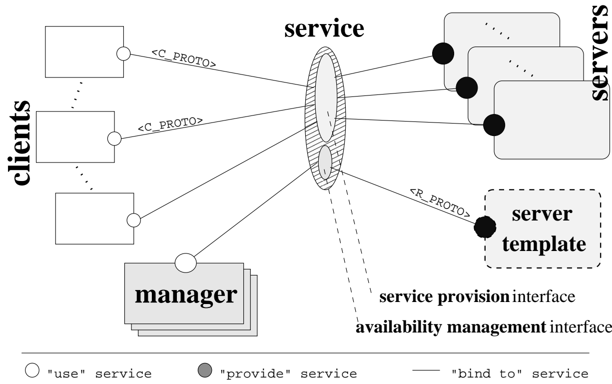

Figure 1: Extended Client - Service model.

The system structure is organised according to an extended Client-Service model, as depicted in figure 1. The basic elements are described in the following:

Service: Is a *typed interface* accepting client and manager requests. It is composed of two sub-interfaces: i) a **service provision** interface typed according to the request/reply messages from/to clients; ii) an **availability management** interface related to the availability manager requests. The service *reference* is realised as a multi-destination (e.g. multicast) address. A *server template* is bound to the service interface. The template is the program module of a server and is resident on any system node the service application may span.

Client: Binds to a service and *uses* it by sending request messages to the service reference and by receiving back replies. The client-service binding is type-safe with regard to the *service provision interface* and the client-service communication protocol; the latter is replication-independent and implements either reliable or unreliable communication. In any case, clients adopt a synchronous style of communication (RPC-like), in the sense that they wait blocked for a reply to their request. In this way, inter-client causal consistency is respected, without having to explicitly record and diffuse service-specific context information (which would be against our transparency assumption), as it is the case with *Lazy Replication* [11].

Server: Binds to a service and *provides* it by delivering and processing client requests sent to the service reference. It is realised as a deterministic process. While processing a request, it potentially produces output including the reply to the client. The server-service binding is type-safe with regard to the *service interface*. The binding is parameterised with the replication protocol (R_PROTO) used for the synchronisation of the server replicas.

Manager: Binds to and uses a service as a special kind of client. It can be considered as part of the *configuration management service* of the system. It carries out the dynamic configuration management of services by invoking special manager requests: `retrieve_membership`, `add_server`, `remove_server`. The availability manager is highly available itself according to an ad-hoc policy [5], e.g. active replication, with a server on every node of the system.

As far as the service application consistency is concerned, replicated service behaviour must be indistinguishable from that of a service provided by a single server. This intuition is formalised as the *one-copy* correctness criteria for replicated services. The other aspect of correctness has to do with the processing of concurrent client requests. In the general case, all requests must be processed in an overall serialisable way. Taken together, the correctness criteria can be expressed as the traditional notion of *one-copy serialisability* (related to message delivery rather than transaction execution, in our case).

3 Protocol Layers

Since servers are considered to be deterministic, there are two main events that must be synchronised among replicas, so that one-copy serialisability is guaranteed: *1)* the *delivery* of client requests from the communication substrate to the application layer, and *2)* the *output* (including the reply to the client) produced by the application layer as result of processing a request. Without loss of generality, we consider the output as equivalent to the reply to the client, for the rest of this paper.

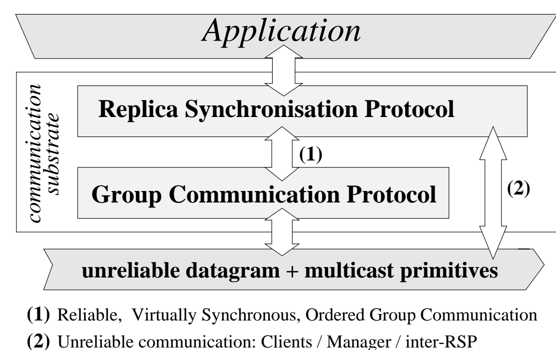

Figure 2: Structure of a replica server.

In order to address these problems, we follow the approach of [10], where a two-layer design is proposed. Specifically, the service application is built on top of a *Replica Synchronisation Protocol (RSP)* layer. The *RSP* layer co-

ordinates request delivery and output among replica servers by exploiting the properties of a *Group Communication Protocol (GCP)* layer.

3.1 The Group Communication Protocol layer

The functionality of *GCP* resembles the properties of typical group communication protocols [15, 1, 18, 8] (no membership changes considered yet):

GCP – Reliability. *Validity:* if a replica broadcasts a message m, then it eventually delivers m. *Agreement:* if a replica delivers a message m, then all replicas eventually deliver m. *Integrity:* for any message m, every replica that delivers m, delivers it at most once, and only if some replica has previously broadcast m.

GCP – Order. If any two replicas p and q in the group deliver m and m', then p delivers m before m' if and only if q delivers m before m'.

A part of the interface provided by *GCP* to the *RSP* layer is shown in table 1.

- `G_bcast(`m`)`: broadcast m to the group, in a *reliable*, *ordered* way.
- `G_deliver(`m`)`: deliver the next message from the group, according to the delivery properties of the protocol.

Table 1: *GCP* interface.

3.2 The Replica Synchronisation Protocol layer

The interface of the *RSP* layer to the application is described in table 2. The interaction between the clients (including the manager) and the *RSP* layer of servers is implemented using the basic communication datagram primitives of table 3.

- `RPC_deliver(`m`)`: deliver the next client request to be processed by the application layer.
- `RPC_reply(`m`)`: send reply to the lastly delivered client request.
- `R_install_state(`m`)`: replication-depended hook—install application state by unmarshalling m; defined by the application programmer according to the application semantics.
- `R_retrieve_state(`m`)`: replication-depended hook—marshal application state into m; defined by the application programmer.

Table 2: *RSP* interface to the application layer.

Messages in our model consist of three fields — `m.id`: the unique (in the system) message id realised as a pair `<sender_id, seq_no>`; `m.tag`: indicating the message type; `m.data` the actual data field of the message.

Each replica's *RSP* layer executes the same algorithm. Thus, the *RSP* layers of different replicas have local pro-

- `send(`m`)`: transmit message m to an explicitly specified recipient p.
- `receive(`m`)`: receive message m sent through the communication network from $m.id.sender$.
- `multicast(`m`)`: an alias of the primitive `send(`m`)` for multicast target references; multicast message m to an explicitly specified recipient group g.

Table 3: Communication primitives of the unreliable datagram.

tocol variables with the same name. When naming ambiguity is possible, we subscribe a variable with the name (membership identifier) of the local replica. Specifically, a replica p maintains the well known reference `gs` of the service group it belongs to. It also maintains a local view of the membership set, in the form of an ordered list of replica identifiers: $\texttt{view}_p = \langle id_1, id_2, ..., id_n \rangle$, where $p = id_j,\ 1 \leq j \leq n$.

Clients transmit messages to the service reference, which are interpreted by the communication network into unreliable multicasts to the *RSP* layers of *all* replicas. Every replica that receives request m (figure 3) buffers it locally, in a data structure `req_buffer`. A unique replica is, then, decided to take the onus for synchronising the delivery of m in the group. The decision is made by each replica individually by applying a "deterministic" *onus* function:

Definition 3.1 *A function* ONUS$(m.id,$ `view`$)$ *is an acceptable onus function when: for any two replicas* p *and* q *of a service group with* $view_p = view_q$ *and* $m_p.id = m_q.id$,

$$\text{ONUS}_p(m_p.id, \texttt{view}_p) = \text{ONUS}_q(m_q.id, \texttt{view}_q) = r,$$

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

Example: ONUS$(m.id,$ `view`$) \equiv$ return $p : p = id_i$ where $id_i \in \texttt{view} = \langle id_1, id_2, ..., id_n \rangle$ and $i = m.id.sender$ *mod* length(`view`)

```
/* RSP – Replica p */
upon receive(m) do
    if m.tag = "client request" then
        store m in req_buffer;
        if ONUS(m.id, view) = p then
            m_g := {tag := "client request", data := m.id};
            G_bcast(m);
```

Figure 3: Receiving a client request at *RSP*.

The single replica p for which $\text{ONUS}_p(m.id, \texttt{view}_p) = p$ generates a synchronisation message m_g with $m_g.tag =$ *"client request"* and $m_g.data = m.id$. m_g is broadcast in a reliable and ordered way to the group using the

`G_bcast()` primitive of *GCP* (figure 3). The delivery of m_g from *GCP* to *RSP* (using `G_deliver()`) indicates the logical time at which the corresponding client request m must be delivered from *RSP* to the application (figure 4). If some replicas have missed (or not yet received) m, they detect that, and require its retransmission from the rest of the group (replicas that have received and buffered it). This inter-*RSP* communication does not have to be reliable; the replica that misses m keeps re-multicasting the requirement until m is received. The replica which would take the onus for m may also miss it. For this reason, a replica with a buffered m expires, if no related m_g is delivered from *GCP* within a timeout period, and re-multicasts m to the group.

```
/* RSP – Replica p */
upon G_deliver(m_g) do
  if m_g.tag = "client request" then
    if m : m.id = m_g.data ∉ req_buffer then
      do
        m_rr := {tag := "retrns.req.", data := m_g.data};
        multicast(m_rr) to gs;
      while timeout and until receive(m);
      store m in req_buffer;
    RPC_deliver(m);
    if ONUS(m.id, view) = p then
      enable replica's output for m;
```

Figure 4: Delivering synchronisation messages from *GCP*.

The reliability and order primitives of the delivery of m_g from *GCP* are used to guarantee the reliable and ordered delivery of the corresponding client request m to the application layer. Therefore, the main correctness criteria for the delivery of requests is:

RSP-1. If at least one replica *receives* a client request, then *exactly one* replica is decided to synchronise the delivery of the request in the group.

The *RSP* layer also coordinates the output of the server group. At the logical time of the delivery of an m_g from *GCP*, and when the client request m is delivered to the application, the `ONUS()` function is employed again. This time, it is used to decide (according to $m.id$) on the replica p which will take the onus of sending the reply to the client, when `RPC_reply()` is called by the application. In that case, only p will actually transmit the reply. Every replica (including p) buffers the reply, in a data structure `out_buffer`, for potential future retransmissions.

When reliable client–service communication is required, the reliable-RPC communication stub of the client (the same as in the non-replicated case) retransmits a request, if no reply is received in some timeout period; it also detects duplicate replies. *RSP* accommodates functionality that implements the reliable RPC channel, on the service side. It maintains a data structure `req_record` to record the id of the last request of any potential client, so that duplicate requests are detected. The receipt of a duplicate is considered as retransmission from the client, and the buffered reply (if any) to the original request is retransmitted by the replica decided to take the onus at this logical time. For a more detailed description of the delivery and output coordination algorithms, the interested reader is referred to [10].

From the way that output is coordinated among replicas using *GCP*'s primitives, we conclude that another correctness criteria for *RSP* is the following:

RSP-2. If some replicas *deliver* a client request, then *exactly one* replica is decided to commit the output of the request.

The intuition behind the introduction of the two correctness criteria *RSP-1* and *RSP-2* is that, in the case of static group membership, they trivially satisfy one-copy serialisability. It is easy to prove that *RSP-1* and *RSP-2* are satisfied by the presented replication protocol, using the properties of the ONUS function.

4 Group Reconfiguration

The dynamic reconfiguration of the server group is due to two reasons. On one hand, system changes like processor failures, process crashes, or system partitioning are all realised as *replica crashes* in our model. On the other hand, the manager operations `remove_server` and `add_server` also cause reconfiguration of the group. Group configuration changes should not violate the service application consistency, as it is expressed by the two correctness criteria above. The main difficulty to cope with this requirement stems from the inherent asynchronous nature of the events that cause the reconfiguration (being either system changes, or manager requests). We require that all these events are interpreted in some pseudo-synchronous way in respect to the other group activity. In particular, all reconfiguration events must be reflected on membership *view* installations, in *RSP*, according to a *Virtually Synchronous* environment provided by *GCP*.

Membership changes in *GCP* are interpreted into special "view" messages delivered to the layer above, as part of the message up-stream. On delivery of a view message, *RSP* installs the corresponding membership view. Virtual Synchrony is defined in terms of the following delivery rules for *GCP*:

GCP – Virtual Synchrony: 1) All correct replicas deliver the same *total order* of views. 2) Any two replicas that deliver two successive views v_i and v_{i+1}, *deliver* the same set of messages in v_i. 3) A message is *delivered* only within

the view in which it is *broadcast*[1].

The first clause, above, refers to the *Primary Partition Group Membership* approach [2] followed in *GCP*. This approach is considered to be more generic, since no application-depended partition merging protocols have to be supported. In the case of dynamically changing groups, two more functions of the *GCP* interface are of interest to our design—see table 4.

- `G_leave()`: require removal from the group.
- `G_join(`g_{ref}`)`: join the specified group.

Table 4: Extended *GCP* interface.

5 Replication Management Protocol

In this section, we propose a *replication management protocol* for *RSP*, which handles configuration management requests by exploiting the virtual synchrony properties of *GCP*. We present the three modules of the protocol, one for each manager operation, and we argue that the two correctness criteria *RSP-1* and *RSP-2* are still satisfied.

For clarity, we do not consider replica crashes in the description of the protocol. However, crashes introduce problems equivalent to those presented in the case of the `remove_server` request. The only difference is that replica crashes are detected and agreed upon in *GCP*. They are reflected on view messages delivered from *GCP*, as is the case with any manager originated configuration change.

Manager requests are received and handled in a way similar to that of client requests. We assume a reliable communication channel established between the manager and the replicas (as it is described in the previous section). However, for clarity, we do not explicitly present reply buffering, duplicate control, and retransmissions, in the following discussion. Unless otherwise stated, the term "replica" is used, in the next paragraphs, to refer to the *RSP* layer of a server.

Retrieve membership view. In the simple case of the manager requiring the current membership view, a single replica is decided, using ONUS(), to send back a reply with the current view as perceived by this replica. Although this view may not be the most up-to-date among all replicas, it is still a consistent view. The manager can retrieve another view with a future request to the group (potentially answered by another replica). No request has to be delivered to the application and, therefore, no broadcast through *GCP* is required to synchronise replica activity. If the group membership is changing concurrently to the request arrival, then more than one replicas may decide to send a reply. Only the first reply will be accepted by the manager. The others will be discarded as duplicates.

[1] See, for example, the *Strong Virtual Synchronous* model of [9].

Remove Server. If the received manager request is a `remove_server` message, then the replica p that has to be removed calls `G_leave()` (figure 5). The latter triggers a group membership agreement in the *GCP* layer. As a result, *GCP* delivers a view message to all the remaining members, which excludes p (possibly among others) from the group (figure 7). This message is used to install a new local view on any surviving replica q. p itself waits blocked for `G_leave()` to terminate. Therefore, no client requests are delivered, in the meanwhile, to the application layer of p. On termination, the *RSP* layer of p sends a reply to the manager, and "commits suicide" on behalf of the replica server.

```
/* RSP – Replica p */
upon receive(m) do
    if m.tag = "manager req–remove server" then
        if m.data = p then /* I am to be removed! */
            wait for G_leave();
            m_rep := {tag := "removal OK"};
            send(m_rep) to m.id.sender; /* manager */
            EXIT();
```

Figure 5: Manager `remove_server` request.

Add Server. The `add_server` manager request is multicast to all the system nodes where the server template resides. Using the dynamic component instantiation primitives of our system model [12], the request triggers the instantiation of a new replica server on the specified node. As part of the instantiation procedure (figure 6) of the *RSP* layer, the `G_join()` method of *GCP* is called parameterised with the service reference (as specified by the binding of the template). The latter triggers a membership agreement protocol in *GCP*.

As a result, a view message specifying the new member is delivered to all replicas including the newcomer (figure 7). On delivery of this view, a single replica q among the old members of the group is decided to take the onus of sending to the newcomer a message with the current (i.e. in the "context" of the view message) service state. This message, denoted m_{state}, has a data field with two parts:
1) The *application state* of replica server q retrieved in a way intrusive to the application layer, by calling `R_retrieve_state()`.
2) The necessary *RSP* layer state: the contents of `out_buffer`, and the set of ids of the last *delivered* request of every client recorded in `req_record`.
`req_buffer`$_q$ does not have to be sent; any buffered messages are detected as "lost" by the newcomer on delivery of the related synchronisation message from *GCP*, and they are required from the group. m_{state} is transmitted to the

newcomer over the unreliable communication network.

```
/* RSP – Replica p */
To execute RSP_instantiate(g_ref) :
    wait for G_join(g_ref);
    gs := g_ref;
    G_deliver(m_v);
    view := m_v.data;
    if p not first member in view then
        do
            wait for receive(m_state);
            if TIMEOUT then
                m_rt := {tag := "retransmit state"};
                multicast(m_rt);
        while TIMEOUT and until received;
        R_install_state(m_state.data.appl_state);
        req_buffer := NULL;
        req_record := m_state.data.last_ids;
        out_buffer := m_state.data.out_buffer;
    if p first member in view then
        R_install_state(NULL);
        req_buffer := NULL;
        req_record := out_buffer := NULL;
    m_rep := {tag := "addition OK"};
    send(m_rep) to manager;
    enable receive();
```

Figure 6: *RSP* instantiation procedure.

When G_join() terminates in the new replica p, the first view message is delivered from *GCP* (it includes p itself) and is used to initialise $view_p$. p's *RSP* layer waits, then, to receive the corresponding m_{state}. Since, inter-*RSP* communication is unreliable, the new replica may expire waiting on receive() and the retransmission of the m_{state} is then required from the group (and this is repeated until the message is received). The contents of the message are used to initialise the application state of the new replica server. This is done, again, in a way intrusive to the application layer by calling R_install_state(). *RSP*'s out_buffer and req_record data structures are also initialised from m_{state}. If $view_p$ indicates that p is the first member of the group, no m_{state} is expected; application and *RSP* state are initialised with whatever is the default initial value. Only when state initialisation has been completed, the *RSP* layer of the new replica replies to the manager and starts receiving client (and manager) requests.

The way that view deliveries from *GCP* are used to coordinate the effects of manager requests on the local views of replicas is depicted in figure 7. In any case, after the installation of a new view, *RSP* (of an "old" replica) re-evaluates the onus for any non delivered requests in req_buffer.

Correctness arguments

Proposition 5.1 *The replication management protocol satisfies RSP-1 in the case of server removals.*

Proof: Initially, assume that *one* server s is removed, resulting in the delivery of a view $v_{i+1} = v_i - \{s\}$. A client request m may arrive "concurrently" to the removal of server s. Since, the arrival is asynchronous with respect to the delivery of messages from *GCP*, some of the surviving servers may receive m before the delivery of v_{i+1}, and others not. There are three cases to be considered:
Case 1: The leaving server s is decided (by those replicas that received m) to take the onus of m's delivery coordination in v_i. Since s is blocked on G_leave(), no corresponding m_g message is broadcast through *GCP*. Therefore, onus for m is re-evaluated when v_{i+1} is delivered (m is eventually received by all replicas, as explained in section 3).
Case 2: One of the (surviving) replicas that received m in v_i is decided to take the onus, and broadcasts an m_g in v_i. According to property (3) of Virtual Synchrony, all surviving replicas deliver m_g in v_i (before the delivery of v_{i+1}). Eventually all these replicas deliver m to the application in v_i. s does not do so, but we do not require uniform request delivery anyway.
Case 3: One of the (surviving) replicas that do not receive m in v_i is decided (by those that receive m in v_i) to take the onus. No m_g is broadcast in v_i, since the replica considered to have the onus ignores the existence of m. Onus for m will be re-evaluated, after the delivery of v_{i+1}.

In all cases, *exactly one* replica broadcasts a message through *GCP* to coordinate m's delivery. It can be easily proved by induction, that that *RSP-1* is not violated when *more than one* server removals are reflected in one view delivery from *GCP*. □

```
/* RSP – Replica p */
upon G_deliver(m_v) do
  if m_v.tag = "view" then
      for all q ∈ m_v.data − view do /* p: newcomer */
          if ONUS(q, m_v.data ∩ view) = p then
              R_retrieve_state(m_appl);
              last_ids := {id of last delivered req};
              m_state := {tag := "state transfer",
                          data := ⟨m_appl, last_ids⟩};
              store m_state in req_buffer;
              send(m_state) to q;
      view := m_v.data;
      for all m_req in req_buffer not delivered yet do
          if ONUS(m_req.id, view) = p then
              m_g := {tag := "client req", data := m_req.id};
              G_bcast(m_g);
```

Figure 7: View delivery from *GCP*.

Proposition 5.2 *The replication management protocol satisfies RSP-2 in the case of server removals.*

Proof: If one or more servers are removed from the group in view v_i, then according to Virtual Synchrony properties (1) and (2) of *GCP*, all replica *RSP*s are delivered v_{i+1} (specifying the removed servers) in the same logical time (i.e. in the same "position" in the delivery stream from *GCP*). The decision about output onus for a request is done at the logical time of the delivery of the related m_g from *GCP*. At this logical time, the local views of all replicas are equivalent, and according to the properties of the ONUS( ) function they conclude at the same single replica p to take the onus. If p is, in the meanwhile, removed, no reply is sent to the client; this is equivalent to lost reply. If the client implements reliable communication with the service, it will request the retransmission of the reply in the future; exactly one replica will be decided to take the onus for reply, in whatever is the membership view at that time. Therefore, it can be easily shown that *RSP-2* is satisfied in the case of one or more servers leaving the group. □

Proposition 5.3 *The replication management protocol satisfies RSP-1 in the case of server additions.*

Proof: A client request m may arrive "concurrently" to the join procedure of a new replica s. In particular, some of the existing servers may receive m in v_i (before delivering v_{i+1} that reflects s), whether others not. There are three cases to be considered:

Case 1: One of the replicas that receive m in v_i is decided to take the onus of broadcasting an m_g for m. According to Virtual Synchrony property (3), all replicas deliver m_g in v_i. Therefore, all "old" replicas eventually receive and deliver m in v_i. The id of m is recorded in $m_{state}.last_ids$ sent to s. No decision has to be made for m in v_{i+1}.

Case 2: The replica that is decided, in v_i, to take the onus does not receive m in v_i, and no m_g is broadcast in v_i. Responsibility for m will be re-evaluated in v_{i+1}.

Case 3: The new server s receives (in v_{i+1} that includes itself) a client request m that was received and delivered by the "old" replicas in v_i. s may also "believe" that it has the onus for m in v_{i+1}. Since m was already delivered on installation of v_{i+1}, it was included in the *last_ids* field of m_{state} to s, and, therefore, recorded on $\texttt{req_record}_s$. If a reply to m is buffered in $\texttt{out_buffer}_s$, then it is retransmitted to the client.

In any case, *exactly one* replica is decided to coordinate m's delivery in the group and m is delivered exactly once. This result can be easily generalised for the case of more than one concurrent server additions. □

Following arguments similar to those of **Proposition 5.2**, we can show that:

Proposition 5.4 *The replication management protocol satisfies RSP-2 in the case of server additions.*

Concurrent `remove_server` and `add_server` requests from the manager are reflected on installation of views with both *joining* and *leaving* group members. Therefore, the results of **Propositions 5.1–5.4** can be applied to conclude on:

Lemma 5.1 *The replication management protocol of figures 5–7, handles manager requests in a way that satisfies correctness criteria RSP-1 and RSP-2.*

6 Discussion and Conclusions

A first implementation of the architecture presented in sections 3 and 5 has been completed within *Regis*. *Regis* [12] 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 communication and computation. The emphasis is on constructing programs from multiple parallel computational components which cooperate to achieve the overall goal. The environment is designed to easily accommodate multiple communication mechanisms and primitives.

The *RSP* and *GCP* layers have been implemented in the form of configurable protocol stacks [18]. Two basic principles were followed for the design of the protocols: *i)* extensive layering, and *ii)* data flow between layers exclusively based on *upcalls* [4]. Deering's IP extensions for multicast [7] have been used for both the client access protocol (client - service communication) and the group communication protocol.

Server addition and removal have a cost (latency time) proportional to the internal server synchronisation (i.e. group communication) cost. In the case of server crashes, the removal cost includes the failure detection time which increases the operation cost, but we do not consider this case here. Some indicative performance results are presented in the following table (measurements done in our network of Sun SPARC IPX workstations connected by a fairly loaded ethernet):

management operation cost (milliseconds)	*# servers*		
	2	*3*	*4*
Server addition	12	21	36
Server removal	8	17	32

In the case of server addition, the group size indicates the number of servers including the new one, whereas in the case of removal it indicates the number of servers including the leaving server. The times presented are average measured times and do *not* include the manager-service-manager communication latency, which is approximately 2.5 *ms*.

The structure and subsequent reconfiguration of a replicated service can be concisely expressed in *Darwin* [12], the configuration language used for structuring Regis programs. Client-service and server-service bindings require a

simple extension to the existing Darwin binding semantics. The dynamic component instantiation facilities supported by Darwin/Regis are used to create new server replicas from the server template of a service. The replicated server group can itself be managed as a single component when it is incorporated into a larger system although we have not yet addressed all the problems of interaction between server groups.

A direct performance improvement for the presented replication protocol can be achieved when clients are aware of replication and they establish a communication channel with *one* of the servers in the group (see [2] for a similar client-access protocol). The main disadvantage of this approach is that, in case of group reconfiguration (including failures), responsibility for clients must be re-distributed in the group, and clients must be informed about that.

In order to address this problem, we are considering the idea of providing clients with *weakly consistent* group membership views. A client can use its own perspective of the group membership as a "hint" for re-establishing the communication path with the service, when its original "representative" server is detected (by the client) as removed. It is only then that the client's membership information is updated. We are currently working on a concise definition of this client access protocol, and we intent to use it as the basis for the design of an alternative replication protocol. We would like to investigate how this model scales, in comparison with our current design, and what application classes it would be suitable for.

Acknowledgements

We wish to thank the anonymous referees for their useful comments. This work has been supported in part by the Engineering and Physical Sciences Research Council (Grant no. GR/J87138).

References

[1] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3), August 1991.

[2] Ken Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[3] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 93-1374, Dept. of Computing, Cornell University, Ithaca, New York, August 1993.

[4] David Clark. The structure of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.

[5] Flaviu Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2), February 1991.

[6] Flaviu Cristian and Shivakant Mishra. Automatic service availability management in asynchronous systems. In *Second International Workshop on Configurable Distributed Systems*, pages 58–68. IEEE Computer Society Press, Pittsburg, Pennsylvania, March 1994.

[7] S. Deering. RFC 1112: Host extensions for IP multicasting, 1989.

[8] Paul Ezhilchelvan, Raimundo Macedo, and Santosh Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems*. IEEE Computer Society Press, Vancouver, BC, Canada, June 1995.

[9] Roy Friedman and Robbert van Renesse. Strong and weak virtual synchrony in horus. Technical report, Dept of Computer Science, Cornell University, Ithaca, NY, USA, March 1995.

[10] Christos Karamanolis and Jeff Magee. Configurable highly available distributed services. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 118–127. IEEE Computer Society Press, Bad Neuenhar, Germany, September 1995.

[11] Rivka Ladin, Barbara Liskov, Liuba Shrira, and Sanjay Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[12] Jeff Magee, Naranker Dulay, and Jeff Kramer. A constructive development environment for parallel and distributed systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*. Pittsburg, March 1994.

[13] Dalia Malki, Yair Amir, Danny Dolev, and Shlomo Kramer. The Transis approach to high availability cluster communication. Technical Report CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.

[14] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 42–52. IEEE, Seatle, Washington, October 1989.

[15] Larry L. Peterson, Nick C. Buchholz, and Richard D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3), August 1989.

[16] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *ESPRIT - Research Reports*. Springer-Verlag, 1991. Project 818/2252.

[17] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.

[18] Robert van Renesse, Takako M. Hickey, and Kenneth P. Birman. Design and performance of Horus: A lightweight group communication system. Technical report, Dept. of Computer Science, Cornell University, Ithaca, New York, 1994.