Configurable Highly Available Distributed Services

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 providing highly available services in distributed systems. In particular, we examine the situation where a service may be used by a large continuously changing set of clients. The requirements for providing services in this environment are analysed and an architecture and partial implementation for a replicated server group meeting a range of client requirements is presented. The architecture facilitates the dynamic configuration management of the replicated server group, while maintaining the service. Dynamic configuration management is required in order to replace failed replicas, upgrade the server implementation, or change the availability characteristics of the service. The paper reports on initial implementation results.

1 Introduction

A *service* provided by a computing system is characterised as *fault-tolerant* [5] when it continues to be provided according to its specifications despite failures of system components (software or hardware) that participate in the service provision. With the ever increasing introduction of computing systems in many aspects of today's life, fault-tolerance of critical services becomes of great importance.

There are two parameters related to the fault-tolerant behaviour of a service:

Reliability: Defined as the *eventual correctness* of the service - improved *"mean time to failure"* and guaranteed recovery (in case of failure) in finite time.

Availability: Defined as the *instantaneous availability* of the service - the probability that the service is provided correctly at a specific moment in time.

Techniques for consistent distributed recovery from failures are employed to achieve reliability. Redundancy is used to improve availability: *replicated servers* are employed for the provision of the service.

In the literature [20, 14, 3, 11, 13], the interest of researchers focuses on the problem of maintaining replica consistency, when replication is employed to achieve high availability. An additional problem raised is that of *dynamic system reconfiguration* - i.e. actions to be taken so that the availability requirements for a service are satisfied despite failures and/or system configuration changes. Dynamic configuration management is required in order to replace failed replicas, upgrade the server implementation, or change the (probabilistic) availability characteristics of the service.

Cristian [6] describes the availability characteristics of a service in terms of an *availability policy* for that service. Two dimensions are distinguished, namely *replication policy* (number of replicas required) and *synchronisation policy* (how close the states of replicas are synchronised). Here, we focus only on *replication policy* as a reconfiguration concern. We assume *close synchronisation* policy [5] (also known as *active replication*) throughout the paper and adopt the *State Machine Approach* [20] to maintain replica consistency.

The paper concentrates on services which may be used by a *large, continuously changing* set of clients. This is characteristic of services provided in the open systems environment. It is not practical in this environment to consider clients as special members of the group of replicated servers which provide the service, as is the case with systems such as *ISIS* [3]. We consider clients as entities external to the group of servers, 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 with replicated servers. Consequently, we do not assume that client communication stubs are aware of replication. In the work presented,

no special invocation protocol is required for clients as is the case with systems such as *Consul* [14] and *Delta-4* [17]. Ideally, clients use the same protocol to access ordinary servers as they do to access replicated servers. We will show that this objective is feasible when the service provides *at most one* reply guarantees to its clients, but it is not feasible when *exactly one* reply guarantees are required.

The transparency objective means that we cannot cope with the problem of *inter-client consistency* by using an approach such as *Lazy Replication* [11]. Since clients are unaware of replication, it is impractical to record and distribute context information concerning any service a client might use. Consequently, we require clients to adopt synchronous (blocking, or RPC-like) communication with servers. While a client is waiting blocked for the reply to its last service request, it cannot transmit messages to other potential clients of the same service.

In the next section, our system assumptions are stated and a precise definition is developed for the clients' view of a highly available service. Section 3 determines a set of requirements for server replica synchronisation which meet this client view definition, and section 4 proposes a software architecture to cope with replica state synchronisation in a generic way that covers both client requests and dynamic reconfiguration operations. The architecture facilitates dynamic configuration management of the replicated server group while maintaining service provision. Section 5 outlines the implementation approach we have taken and presents some initial implementation results.

2 System model

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*. The system is augmented with a *failure detector* [4] to circumvent the impossibility results for distributed consensus in this system model [9].

The paper assumes the client-server organisation depicted in figure 1. The basic elements are described in the following.

Service: Is a *typed interface* accepting client requests; it is realised as a *reference* (e.g. a multi-destination address). The interface consists of *two parts*, one for the clients and one for the availability manager (see below). The actual replication policy of the service is hidden from all system entities but the manager.

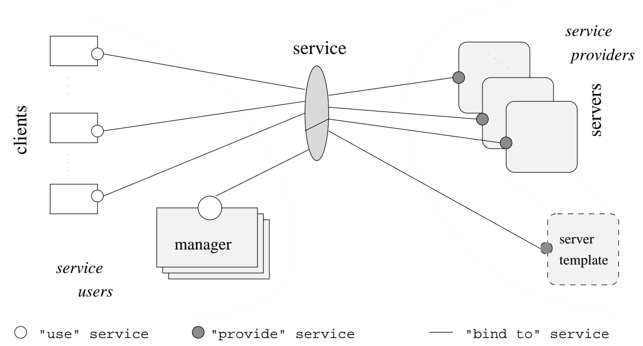

Figure 1: Extended client - server model.

Client: Binds to a service and *uses* it by sending messages to the service reference and receiving back replies. A request sent to the service reference is automatically forwarded by the network to all the servers of the service.

Server: Binds to a service and *provides* it by responding to client requests received through the service interface. While processing a request it potentially produces output including the reply to the client. Servers do not have a consistent view of the (evolving) client set. At creation time, a server *joins* the group of the service it binds to (by retrieving the current state from the other servers), in a way transparent to the application layer.

Manager: Binds to and *uses* a service as a special kind of client. It *creates* a service interface by specifying its type and binding a server template to it. The template is the program module of a server and is resident on any host the service application spans. The manager enforces the service replication policy by invoking special management requests to it, namely: *"add"* server (create a new server on a specific host, from the template), *"remove"* server, and *"retrieve"* membership information. The manager is highly available itself according to an ad hoc availability policy [5].

The introduction of the *service* entity has the ad-

vantage of separating the concerns of dynamic changes of the client set from those relating to reconfiguration actions required to satisfy the availability policy for a group of servers. We can therefore define the properties required by a client from a service without reference to the organisation of servers. To define this client's view, we assume that the configuration management of the server group guarantees its continuous availability. That is that failed or removed servers are replaced "fast" enough such that a minimal operational set is always available. The client's view is then defined by the following properties:

C1 *(weak reply)*: If a correct *client* C invokes a request to a *service* S, then C receives *at most* one reply.

C2 *(strong reply)*: If a correct *client* C invokes a request to a *service* S, then C eventually receives a reply, given that C remains correct.

C3 *(validity 1)*: A correct *client* C receives a reply from a *service* S, only if C has previously invoked a request to S.

C4 *(validity 2)*: If a *client* C receives a reply from a *service* S, then the reply causally depends on *all* previous requests of C to S (i.e. the state of S has not "missed" any of the previous requests of C).

Property *C2* reflects the intuitive definition of *highly available service provision*, which implies that the client application is guaranteed a reply to any request invoked to the service. *C2* combined with *C1* articulates the *exactly one* reply guarantees to the service clients. The *exactly one* reply guarantees are not just a matter of replica synchronisation, but are also related to the problem of reliable client-service communication (in the sense that no request or reply messages are ever lost). We will show that only *at most one* reply guarantees (property *C1* - weak reply) can be provided by the replica synchronisation protocol. Provision of *exactly one* guarantees requires special communication stubs on the clients' site to cope with client-service communication. The latter is against our initial objective for transparency of replication to the service clients, and we will cope with it only under certain circumstances.

How do *reconfiguration concerns* fit in this model? It has already been mentioned that the configuration manager should maintain a minimal operational set of servers, which is *eventually* consistent with the (probabilistic) availability requirements for the service. In addition, any configuration operation on a service must not violate properties *C1* - *C4* above. Specifically, no inconsistencies of service state concerning processed client requests should be caused; no unprocessed requests to be lost or redundant requests to be processed; no omission of replies to clients or redundant reply delivery by clients. Moreover, dynamic configuration management must be facilitated in a way that is transparent to, and independent from the application layer.

3 Replica synchronisation

In this section we present an interpretation of the above properties into a set of requirements for replica synchronisation. In the first two parts we consider the two *validity* properties, but only *weak reply* (at most one reply) client requirements. We analyse the requirements for coordinated *request delivery* and *output commit* on the replicas, under both static and dynamic server group configuration. At the end of the section, we discuss the the reasons that *strong reply* (and consequently *exactly one* reply) cannot be guaranteed by the replica synchronisation protocol without the participation of clients.

3.1 Normal operation

Since we assume deterministic servers, the main event that must be synchronised among replicas is the **delivery** of requests from the communication substrate to the application layer of the replica. Delivery must be synchronised according to the following properties:

- *D-Termination*: If a correct replica *receives* a client request m, it will eventually *deliver* m (given that the replica remains correct for long enough).
- *D-Reliability*: If a correct replica *delivers* a client request m, then all correct replicas will eventually *deliver* m (given that the first replica will remain correct for long enough), i.e. all replicas deliver the same set of requests.
- *D-Order*: All replicas must deliver the requests in an order that does not allow the replica states to diverge. The most general (strongest) constraint is: all replicas deliver all client requests in the same *total order* [20].

Another activity that requires synchronisation is the **output** (including reply to the client) during the processing of a request. We adopt the *single output* approach - all correct replicas must agree on exactly

one replica to send out messages during the state transition triggered by the delivery of a specific request. The latter is expressed as the combination of two properties:

- *O-Validity*: If a request is delivered by one or more correct replicas, then *at least one* replica will produce the output during the related state transition.
- *O-Agreement*: If a request is delivered by one or more correct replicas, then *at most one* replica will be selected to do the actual output.

The notation `resp(m)` will stand for the replica assigned the responsibility of the output due to request m. Responsibility for m is decided in a distributed manner (individually by each replica) according to some deterministic rule (e.g. client vicinity). Output will be considered as equivalent to *reply* to the client, throughout the rest of the paper.

3.2 Group reconfiguration

Manager requests are handled in the same way as any other client request. Further, some manager requests (namely, *"remove" server* and *"create" server*) affect the membership set of server replicas. Membership changes are also due to server failures (crashes). As it has already been mentioned (section 2), we assume the existence of a system *failure detector* [4, 18], which suspects and announces crashed replicas. The proposed architecture should accommodate a *membership service* [15, 18, 8], which will consistently interpret all the above configuration changes and detected failures into *views* [2] of membership, in the replicas.

The decision about responsibility is a function on the current membership set, as it is perceived by a replica. Therefore, it is important for all significant events on replicas (i.e. delivery of client *requests* and *view* updates) to appear as if each of them occurred at the same logical time on all replicas. For this reason, we adopt the *virtually synchronous* model of systems like *ISIS* [3] and *Transis* [13]. Informally, all correct replicas must follow the same total order of view updates, and also deliver the same set of client requests between any two successive view updates.

The potential failures of server replicas introduce an *output commit* problem. It may be the case, that a client request m is received (and delivered) by just a fraction of the server group, due to communication network failures. In addition, all these servers may crash due to a combination of failures, before the remaining servers are forced to receive and deliver m according to the delivery properties of sec.3.1. If `resp(m)` was among the failed servers and had sent a reply back to the client before it crashed, then the service and the client would be in inconsistent states.

In order to avoid such scenarios, our virtually synchronous model must be augmented with a safety property concerning the output produced by the responsible replica:

Safe output: `resp(m)` is enabled to send a reply back to the originator of request m only when m becomes *stable*, i.e. it has already been received by all correct replicas in the current membership set.

The reader may have already noticed that, although the safe output property guarantees *output agreement*, it does not guarantee *output validity*. In other words, only *at most one* reply guarantees are provided, but not *exactly one*. There may be reconfiguration scenarios where `resp(m)` does not send a reply, although the other replicas consider it has. It is guaranteed, however, that if a reply to a request m is sent out, the service state will reflect the delivery of m despite any reconfiguration changes.

The only way to guarantee *exactly one* reply from the server group would be to have `resp(m)` transmit the reply to both the client and the rest of the replicas in a reliable and atomic way. This would require the client to participate in the group communication protocol of replicas, which is against our system model assumptions (see sec.2). For this reason, we propose an architecture that provides just *at most one* guarantees. The *exactly one* reply problem, considered equivalent to reliable client-service communication, is solved in an extension of the architecture that requires the participation of the clients. Specifically, clients must retransmit lost requests and detect duplicate replies. The latter is incompatible with our other initial objective about replication transparency to clients. We, therefore, intend to provide this extension as an option to the application programmer, when *exactly one* reply guarantees are indeed required by the application semantics, and the clients of the service can be programmed to accommodate the required communication stubs.

In the case of the system splitting into more than one partition, we follow the *primary partition* (*pp*-) model [18, 19]; the replicas in *at most one* of these partitions continue providing the service. Any replica not in the main partition "commits suicide". It can only join a main partition as a new member. In this way, the service is provided by a unique totally ordered sequence of primary replica partitions. The main advantage of the *pp*-model is that it does not depend

on application semantics, as it is the case with *partial virtual synchrony* [8].

The benefits of the *pp*-model come at the price of potential blocking of the system when no primary (e.g. majority) partition can be formed. We delegate this problem to the *configuration manager* (which also partitions). There, the decision about which (if any) *service partition* should be augmented with new servers, so that it becomes operational, is done in a heuristic way (possibly requiring the operator's intervention).

4 Architecture design

This section outlines the design of an architecture that conforms to the requirements defined in the previous section. We will show how a replica synchronisation protocol copes with client requests and reconfiguration events in a generic way by exploiting the properties of (traditional) "closed" group communication protocols. In the first two subsections we present an architecture that provides *at most one* reply guarantees. Only at the end of the section, the architecture is augmented with another layer to support reliable client-service communication and *exactly one* reply guarantees, when possible.

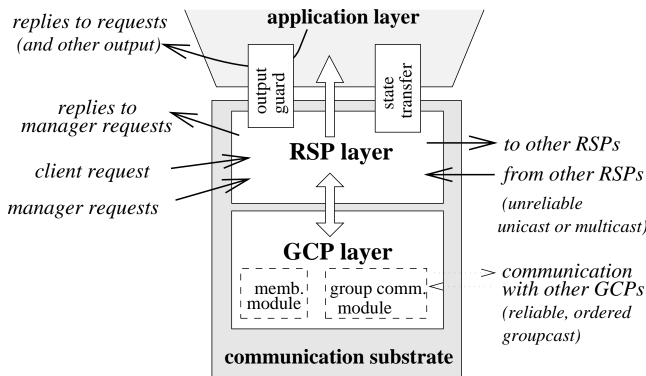

Figure 2: Structure of a replica server.

4.1 Replica synchronisation

The architecture consists of two layers. The service application, on each replica, is implemented on top of a *Replica Synchronisation Protocol (RSP)* layer, which in turn employs a *Group Communication Protocol (GCP)* - see figure 2.

RSP receives and handles *client* and *manager requests*. It implements the replica synchronisation protocol (specified in sec.3) by transmitting special inter-group messages through the *GCP* layer.

GCP guarantees reliable and ordered delivery of inter-group messages. It provides a consistent membership view to the *RSP* layer. These properties are satisfied by many of the groupcast protocols described in the literature [16, 2, 21].

It will become apparent that this modular approach facilitates not only the presentation of the proposed architecture, but also the reasoning for its correctness and completeness.

The RSP layer. We will describe the functionality of *RSP* based on the different events that can occur in this layer. These events are either arrivals of external request messages or deliveries of messages from *GCP*.

The external messages can be of three types: *i)* a client request; *ii)* a manager request; *iii)* another replica's *RSP* layer message. Figure 3 outlines the way external requests are handled.

```
/* Replica Ri - RSP layer */
on  receive(mreq)  do :
  if already_received(mreq) then
     handle_duplicate(mreq);
  resp = responsible(mreq.sender, current_view);
  case mreq.type :
  1. "client request" or "manager request" :
     if  resp and mreq.type=="view retrieval" then
        send to mreq.sender (current_view);
     else  if  resp  then
              mid = {mreq.id, mreq.type};
              GCP.broadcast(mid);
           endif;
           Buffer.put(mreq);
           if  not resp  then  timer.set(mreq);
  2. "RSP retransmission request" :
     if  resp  then
       send to mreq.sender (Buffer.retrieve(mreq));
        /* unreliable transmission */
  3. "RSP join request" :
     if  resp  then
        m' = APPL.current_state();
        send to mreq.sender (m'); /* unreliable */
        mid = {m'.id, m.id, "state transfer"};
        GCP.broadcast(mid); /* reliable,ordered */
  4. "RSP reply to join request" :
     if  resp  then    /* I'm the one joining */
        Buffer.put(m); /* ... wait for mid */
end;
```

Figure 3: RSP events: external requests.

Whenever such a message, say m_{req}, arrives at a replica R_i (in the next paragraphs, when we mention

a replica, we will mean its RSP layer) the *responsibility rule* is applied locally to decide whether R_i is responsible for m_{req} or not, according to its current membership view. The role of the responsible replica depends on the type of m_{req}. If m_{req} is a *manager request* inquiring the *current membership set*, the responsible replica will reply with its current view. Any other replica will discard the message.

In the case of m_{req} being a *client request* or a *manager "remove server" request*, every replica will buffer m_{req} locally. In addition, the responsible replica will create and broadcast through GCP a message m_{id} containing m_{req}'s *id* (which is unique in the system).

This inter-group m_{id} message is delivered in a reliable and ordered way to all replicas RSP layers, and it is the means used to achieve replica synchronisation. If it is related to a client m_{req}, it is used to coordinate the delivery of the request to the application layer. If it is related to a manager request, it is used to coordinate the effects of the request on the membership view in the RSP layers.

In figure 3 we have not considered manager "create server" requests arriving at RSP. These requests are forwarded (through the service interface) to the *server template* on the appropriate host, from which a new replica is dynamically created. As part of its instantiation procedure, the RSP layer of the new replica transmits (it could be just an unreliable multicast) a *"join"* request to the RSP layers of the existing group members. When such a *"join"* request of a new replica R_j is received, the unique responsible replica R_i creates and sends an m'_{req} with the current application state back to R_j. The latter is accompanied by an m_{id} containing the ids of both m_{req} and m'_{req}. Receiving a reply to a join request is of interest only to the replica that previously invoked the related join request (it is the responsibility function's result in this case). The installation of the service state in the application layer of the new member is done according to the delivery of the related m_{id}, as we will see in the next paragraphs.

The replica selected as responsible for a request does not only enable the output (for this request) from its application layer, but also coordinates the reliable and ordered delivery of the request in the group.

The delivery of m_{id} messages from GCP to RSP is used for replica synchronisation. A delivered m_{id} is handled according to the type of the m_{req} it is related to (see figure 4):

1. A message m_{id} related to a *client request*.

In that case, RSP delivers the related buffered m_{req} to the application layer. If m_{req} is not present in R_i, its retransmission is requested from the responsible

```
/* Replica Ri - RSP layer */
on deliver(mid) from GCP do :
  case  mid.type :
  1. "related to client request" :
     if  mreq = Buffer.retrieve(m.rel_id) OK  then
        APPL.deliver(mreq);
        ack(mid);   /* ack mid's delivery */
        if responsible(mreq)  then
           APPL.enable_output();
        else  APPL.disable_output();
     else   /* related mreq not received */
        send to mid.sender(mid.rel_id,"retr.req.");
  2. "view" :
     ack(mid);
     case subtype(m) :
     2.1 "rel. to manager's `remove server'" :
         if responsible(mid.rel_id) then
            commit suicide!;
         else  current_view.install(mid.data);
     2.2 "rel. to `state transfer'" :
         if responsible(mid.orig_id)  then
            /* I'm the joining replica */
            if (mst = Buffer.retrieve(mid.rel_id)
                       NOT OK) then
               send to mid.sender
                      (mid.rel_id, "retrans.req.");
            else  /* state message here */
               APPL.init_state(mst);
               current_view.install(mid.data);
         else  /* not responsible */
            current_view.install(mid.data);
     2.3 "due to replica failure" :
         current_view.install(mid.data);
     endcase;
     for all mreq in Buffer & mreq not delivered do:
         if responsible(mreq) in new view  then
            mid = {mreq.id, mreq.type};
            GCP.broadcast(mid);
  endcase;
end;
on timer.expires(buffered mreq) do:
    multicast mreq to all RSPs in current_view;
    /* unreliable multicast */
    timer.set(mreq);
end;
periodically do:
    Buffer.garbage_collect() all stable(mreq);
end;
```

Figure 4: RSP events: messages delivered by GCP.

replica (message class 2 in fig.3). One-to-one (not necessarily reliable) communication can be used. The retransmission request is re-sent, if it timeouts without m_{req} having been received. Equivalently, the retransmission request could be an (unreliable) multicast to the group and, then, *RSP* is waiting for the first reply. In any case, R_i waits blocked to receive m_{req}. *RSP* explicitly acknowledges the delivery of m_{id}, only after m_{req} has been received locally. In that way, stability of an m_{id} in *GCP* implies also stability of the related m_{req} in *RSP*.

A replica may "miss" an m_{req} for which it would have been responsible. For this reason, if a non-responsible replica has buffered an m_{req} but does not receive a related m_{id} within a timeout period, it retransmits (e.g. unreliably multicasts) m_{req} to the group. This m_{req} is handled as normal client request by other replicas, i.e. it is discarded in case it has already been received (handling of duplicate requests is described in sec.4.3).

After the delivery of m_{req} to the application layer of the *responsible replica*, output is enabled in the output guard, only when m_{req} becomes stable. Periodically, stable buffered requests are garbage collected on all replicas where they have been up-delivered to the application.

2. A *view* message indicating group membership change. We distinguish three kinds of view messages:

- Related to a manager *"remove server"* m_{req}. For a replica being the responsible, in this case, means that it is the one that must be removed. Therefore, on delivery of this *view* message, the *RSP* layer "commits suicide" on behalf of the replica. For any other *RSP*, such a message is interpreted into a new view installation (in a way synchronised with all other replicas).

- Related to a *"state transfer"* m'_{req}. If R_i is the replica that sent the original *join* request, m'_{req} is used to initialise the application state (or its retransmission is asked if missing, as usual). The m_{id} also indicates the context on which the delivered state depends on, so that *GCP* is initialised according to this context and participates (on behalf of the replica) in any group communication following this context.

- A *view* message originated by the *GCP* layer due to some replica(s) failure, that has been detected by the failure detector and agreed upon by the membership protocol (a module of *GCP*).

The delivery of any *view* message results to the installation of a new view in the *RSP* layer. After installing a new view, a replica re-evaluates the responsibility of buffered requests for which no m_{id} has been received yet. If it turns out to be responsible for a request for which it was not responsible in the previous view, it creates and broadcasts an m_{id} as above.

The GCP layer. The above architecture sketch quite clearly exhibits the reduction mentioned at the beginning of the section. The initial target was an *RSP* layer consistent with the requirements of section 3. We introduced, however, the *GCP* layer to broadcast the special, internal to the group, m_{id} messages. We made the two main *RSP* events, *i)* request delivery, and *ii)* membership view updates, equivalent to the delivery of two different classes of messages from *GCP* to *RSP*. In that way, the requirements for replica synchronisation are *reduced* to requirements for the *GCP* layer. The latter resembles traditional "closed" group communication protocols:

Reliability: m_{id} messages must be broadcast reliably within the current view of the group. All three properties defined in [10] must be satisfied, namely *Validity*, *Agreement*, and *Integrity*. *GCP Validity* guarantees replica delivery *Termination*, while *GCP Agreement* and *Integrity* assert replica delivery *Reliability*.

Order: We require that m_{id} messages are delivered in an *atomic* (total) *consistent with causality* order. This property asserts a totally ordered delivery of client requests from *RSP* to the application layer.

Stability: As it has already been mentioned, stability of m_{id} messages is specified according to the explicit acknowledgements of m_{id} delivery, from the *RSP* layers. Implicit acknowledgements of their receipt (or delivery) from the *GCP* layers would not provide any information about the receipt of the related m_{req}s in *RSP*. Therefore, we adopt the approach of *Horus* [21], where stability is decided by the layer that uses the group communication protocol.

Virtual Synchrony: The *vs*-model required for the *RSP* layer is directly based on the (non-uniform) virtually synchrony primitives provided by the membership module of the *GCP* layer. *GCP* must deliver *"view"* m_{id} messages in the same total order in all replicas; moreover, the same set of client request - related m_{id}s must be delivered between successive *"view"* messages.

Concerning the *inter-RSP communication* (retransmissions of client requests, join/state messages), no

special requirements are set. It can be unreliable multicast or unreliable one-to-one communication among the replicas. The reason is that request delivery requirements are satisfied by using the reliable, ordered m_{id} broadcasts and not the inter-*RSP* communication.

4.2 Some implementation related aspects

The application layer entity at each server replica has a single thread of execution. *RSP* and *GCP* layers do not have their own thread of execution. *GCP*'s functionality is implemented within the system's (e.g. kernel's) thread of execution. *RSP*'s functionality is implemented partially by the system's thread of execution, and partially by the application's thread of execution (see fig.5).

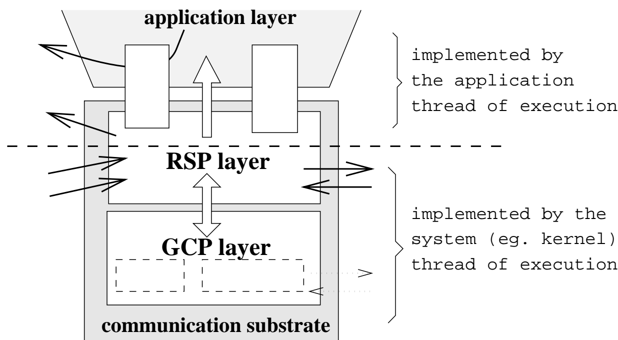

Figure 5: Layer implementation.

Requests arriving at the *RSP* layer are handled *eagerly*, i.e. the buffering and the potential m_{id} broadcast are triggered by the message arrival and are executed by the system thread. The retrieval of messages from the *GCP* layer is implemented *lazily*, i.e. a message delivered by *GCP* is retrieved by *RSP* when the application execution thread requires the next client request. Specifically, when the application requires the next request for processing, the *RSP* layer retrieves the oldest pending m_{id} among the messages that have been delivered by *GCP*. If it is related to a client request m_{req} that is already buffered locally, the request is delivered to the application layer. If not, m_{req} is requested from the *RSP* layers of other replicas (or from just the responsible), and the application thread is blocked until the request is received and delivered. The application thread is also blocked when there is no m_{id} delivered from the *GCP* layer, until such a message is delivered. If the delivered m_{id} is a *view* message, a new membership view is installed in the *RSP* layer. Retrieval of the next pending m_{id} is then attempted, until a message related to a client request is obtained, as above.

We have mentioned that when a replica receives a *join* request from a newly joining replica and it is decided to be responsible for this request, it sends an m'_{req} with the current state of the replica (considered as the state of the service). The state, though, is application dependent. Therefore, the state retrieval is intrusive to the application layer, in the sense that the programmer must define what the state is, and make it available to the *RSP* layer through some *"state transfer"* interface (implemented, for example, as a method of the application module, which can be invoked by the *RSP*).

Similarly, the *output policy* enforcement is based upon intrusion to the application layer. Application output is filtered through an *"output guard"* method of the *RSP* layer. The output guard of just *resp(m)* allows output due to m to be actually transmitted.

4.3 Extended architecture

The "exactly-one" reply guarantee to the clients requires (in general) retransmission of lost requests and handling of duplicates on both sides. The solution to this problem is considered by other researchers [1] as an application level concern. We follow a different approach by moving this burden into the communication substrate, when this is possible. Specifically, we propose an extension to our architecture which guarantees reliable client - service interaction. The only requirement is that the clients accommodate a special communication stub that is compatible with the replica synchronisation protocol.

This extended architecture level is implemented as a sub-module of the *RSP* layer. All external requests delivered by *RSP* are recorded, in order to detect duplicates among future requests. For example, the id of the last request of any occurring client is recorded (e.g. in a table hashed on the client id). Duplicates are not passed to the application, but are considered as retransmissions from the client that has not received a reply, in which case the old reply is retransmitted. Replies are buffered in the output guards of all replicas (not just the responsible), and are re-transmitted by the replica that is the responsible in the current membership set. For storage efficiency, "old" buffered replies are garbage collected according to some heuristic way (for example, making use of the existence of loosely synchronised clocks in the system - see [11] for a similar technique). This technique is introduced just for efficiency reasons and does not affect the correct-

ness of the presented architecture.

The requirement of recording delivered client requests does not contradict our initial assumption about servers ignoring the actual set of clients; no consistent client membership information has to be maintained in *RSP*. It is, however, the only part of the architecture that assumes the existence of a special communication stub on the client site to retransmit requests when they timeout without a reply received and to detect and discard duplicate replies. For this reason, and because re-programming the clients may be impossible (in open systems), we intend to provide the programmer with the option to use this extension only when the application semantics require *exactly one* reply guarantees and the clients can be programmed to accommodate the appropriate stubs.

5 Discussion and Conclusions

The architecture presented in the previous section is currently being implemented within the *Regis* testbench. *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 latter characteristic of *Regis* has permitted an elegant implementation of the *GCP* layer. The runtime system is extended to accommodate a *group communication mechanism*, that is based on hardware multicast (Deering's IP extensions for multicast [7]). It provides *reliable* multicast with a range of *ordering* primitives, namely *FIFO*, *causal*, and *total consistent with causality* delivery. Unreliable *out-of-bound* delivery is also supported. The mechanism is augmented with a membership service which provides *virtual synchrony* using a system-wide *failure detector*. This substrate is a module independent from the implementation of the *Replica Synchronisation Protocol* and could be substituted by any other group communication mechanism which is consistent with our requirements. To verify this claim, we intend to substitute our current *GCP* layer with *Horus* [21].

We have also completed a first implementation of the *RSP* layer. Table 1 presents some indicative performance results of this implementation in our environment of Sun SPARC IPX workstations. A message size of 100 bytes is chosen for both requests and replies. We use the term *latency* for the *average* time between sending a request from a remote client to a group of servers and receiving a reply (no processing delay has been added at the service application level). The presented latency includes the time required to transmit the request to the service and the reply back to the client. This is a total of 2.5*msecs*. The rest is the time required for the internal *Replica Synchronisation Protocol*. The first row is included in order to stress the performance overhead introduced in the protocol due to the *safe output* constraint in the responsible replica.

latency (msecs)	*# servers*			
	1	*2*	*3*	*4*
No safe output	3.5	5	7	9
Safe output	3.8	11	20	35

Table 1: Performance results.

This architecture requires the existence of some additional components in the system configuration of *Regis*: a replicated *service reference repository*, and a highly available, replicated *availability manager* have been designed and are being implemented. Replicas of both these entities should be automatically created on every *node* of a *Regis* program.

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 and integrate new server replicas into the server group. 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.

This paper has described an architecture to support configurable highly available services. We have concentrated on the requirement of supporting large client sets. The next stage of the work described in this paper is to investigate how availability policy may be expressed in a general and portable manner. Our objectives are to allow policy to be changed dynamically by informing availability managers of the new policy. In addition, we plan to extend the architec-

ture to flexibly support different *synchronisation policies* (from completely active to passive replication) and *output policies* (single vs multiple).

Acknowledgements

We wish to thank the anonymous referees for their useful comments. This research was supported in part by a grant from the British Council.

References

[1] Ozalp Babaoglu and Andre Schiper. On group communication in large-scale distributed systems. In *Proc. of the 6th ACM SIGOPS European Workshop (Matching Operating Systems to Application Needs)*, pages 17–22. ACM, Wadern, Germany, September 1994.

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

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

[4] 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. A preliminary version appeared in the *Proc. of the 10th ACM Annual Symposium on Principles of Distributed Computing*, Aug. 1991, Montreal.

[5] Flaviu Cristian. Understanding fault-tolerant distributed computing. *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] Danny Dolev, Dalia Malki, and Ray Strong. An asynchronous membership protocol that tolerates partitions. Technical Report CS TR94-6, Inst. of Computer Science, The Hebrew Univ., and IBM Almaden Research Center, Jerusalem, Israel, and San Jose, USA, 1994.

[9] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[10] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Dept. of Computer Science, Univ. of Toronto, and Dept. of Computer Science, Cornell Univ., May 1994.

[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] Shivakant Mishra, Larry L. Peterson, and Richard D. Schlichting. A membership protocol based on partial order. In *Int. Working Conference on Dependable Computing for Critical Applications*, Tuscon, AZ, February 1991.

[16] 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.

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

[18] Aleta Ricciardi and Kenneth Birman. Using process groups to implement failure detection in asynchronous environments. In *Tenth Annual ACM Symposium on Principles of Distributed Computing*. ACM, August 1991.

[19] Aleta Ricciardi, Andre Schiper, and Kenneth Birman. Understanding partitions and the "no partition" assumption. In *Proc. of the 4th IEEE Workshop on Future Trends of Distributed Systems*, pages 354–360. IEEE, Lisboa, Portugal, September 1993.

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

[21] 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.