

Configurable highly available distributed services

Christos Karamanolis

Jeff Magee

*Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, UK
Email: {ctk, jnm}@doc.ic.ac.uk*

1 Introduction

A *service* provided by a computing system is characterised as *fault-tolerant* [4] 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 main parameters related to the fault-tolerant behaviour of a service: *reliability* - defined as the *eventual correctness* of the service, and *availability* - the *probability of a service being correct 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 [14, 11, 9, 2], 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. Dynamic configuration management is required in order to replace failed replicas, upgrade the server implementation, or change the availability characteristics of the service.

According to Cristian [5], the availability characteristics of a service are described in terms of an *availability policy* for the 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 problem. For simplicity, we assume *close* [4] *synchronisation policy* throughout the paper.

In this context, the requirements for providing highly available services are analysed, and an architecture and partial implementation for a replicated server group is presented. The architecture facilitates the dynamic configuration management of the replicated server group, while maintaining replica consistency.

2 System model

We assume an *asynchronous* system augmented with an *Unreliable Failure Detector* [3] to circumvent the impossibility results for distributed consensus in this system model. Processes are deterministic and exhibit *crash* failure semantics, while only *omission* failures occur on message transmission.

A system organised according to the *client - server* paradigm is assumed. In our model, however, the single server "behind" a service is substituted by a group of replicated servers.

The main characteristic of our model is that a service may be used by a *large continuously changing* set of clients. Clients may spread over more than one program (in open systems). For these reasons, clients are considered entities external to the group of servers, in the sense that they cannot participate in the replica state synchronisation, in any way.

More specifically, the basic elements of the model are (see fig.1):

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. The actual replication policy of the service is hidden from all system entities but the manager.

Client: *Binds to* a service and *uses* it by sending request messages to the service reference and receiving back replies. Clients adopt a synchronous (blocking) style of communication, so that no inter-client consistency problems have to be considered. A request sent to the service reference is automatically forwarded by the network to all servers of the service. The way that the replication policy of the service is enforced is transparent to the clients.

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 some 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 service state), 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 while specifying its type, and binds a server template (used for the creation of servers) to it. It enforces the service replication policy by invoking special management requests to it, namely: “*add*” server (create a new server from the template), “*remove*” server, and “*retrieve*” membership information. The manager is highly available itself according to an ad-hoc availability policy [4].

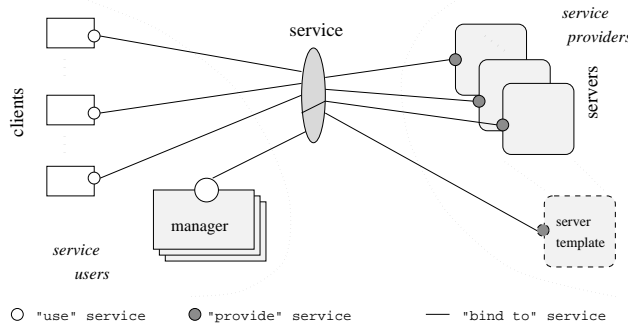


Figure 1: Extended client - server model.

The introduction of the *service* entity exhibits the following advantage: the dynamic changes of the client set can be ignored by the configuration management operations related to the enforcement of the replication policy of the server group.

We can, therefore, focus on the client’s view of a highly available service, without considering dynamic changes on the client set.

- ◊ If a correct client C invokes a request to a service S , C should receive *exactly one* reply in *finite time* (eventually), given that C remains correct. C receives a reply from S *only if* it has previously sent a request to S . The reply should be consistent with *any* replies to previous requests of C to S (i.e. the state of S has not “missed” any of the previous requests of C).

How do *reconfiguration concerns* fit in this model? Any configuration operations on a service must not violate the above view of the service clients concerning service availability. 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.

The client view of high availability is reflected on a set of requirements for any architecture that implements the server replica synchronisation:

Normal operation

The main event that must be synchronised among replica servers is the **delivery** of requests from the communication substrate to the application layer. Delivery must be synchronised according to the following properties:

- *RD-Termination*: If a correct replica receives a client request m , it will eventually deliver m (given that the replica remains correct for long enough).
 - *RD-Reliability*: *i) Agreement*: 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. *ii) Integrity*: For any request m , every correct replica delivers m at most once, and only if m was previously received by some correct replica, i.e. no spurious messages are delivered by any replica.
 - *RD-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.
- 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 agree on *exactly one* replica to send out messages during the state transition triggered by the delivery of a specific request. This approach can be expressed in terms of two properties:
- *O-Validity*: If a request is delivered by all the correct replicas, then *at least one* replica will actually produce the output during the related state transition.
 - *O-Agreement*: If a request is delivered by all the 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). For simplicity reasons, output will be considered as equivalent to *reply* to the client, throughout the rest of the paper.

Group reconfiguration

Manager requests must be handled in the same way as any other client request. In addition, some manager requests (namely, “*remove*” *server*, and “*add*” *server*) affect the membership set of server replicas. Membership changes are also due to server failures (crashes). As it has already been mentioned, we assume the existence of a system *failure detector* [3, 12], which suspects crashed replicas. The proposed architecture should accommodate a *membership service* [7, 12], which will consistently interpret group changes into membership *views* [1], 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 instant on all replicas. For this reason, we adopt the *virtually synchronous* model (*vs-model*) of systems like *ISIS* [2] and *Transis* [11].

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. 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.

The provision of “*exactly one*” output guarantee, in combination with the *output commit* problem would require a strict *uniform vs-model* [13]. This model is expensive, in terms of request delivery delays due to uniformity constraints. By weakening the output guarantees to just “*at most one*” reply, we can accept the less strict (and inexpensive) simple *vs-model*, augmented with a safety requirement for the responsible replica:

Safe output: $resp(m)$ delivers m (and therefore produces output), only when m has already been delivered by all other replicas in the current membership set.

It is implied that $resp(m)$ may not deliver m , although the other replicas consider it has. Therefore, only “*at most one*” reply can be guaranteed. The provision of the additional “*exactly one*” property is considered as a problem equivalent to loss of the reply message. This problem is solved in a level implementing reliable client - service communication.

In the case of system splitting into more than one partition, we follow the *primary partition* model [12], i.e. the servers (replicas) in *at most one* of these partitions will continue providing the service. Any replica not in the main partition “commits suicide”. It can only join a main partition as a new member. This fact allows us to avoid expensive uniformity constraints on message delivery to guarantee re-joining consistency (which are required even if the replicas in non-main partitions are *blocked* until they re-join the main partition). We also prefer this model because it does not depend on application semantics as models that allow partitioned operation [7] do.

3 Architecture design

In this section we outline an architecture which conforms to the previously set requirements. The design exploits the properties of (traditional) “closed” group communication protocols.

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 fig.2).

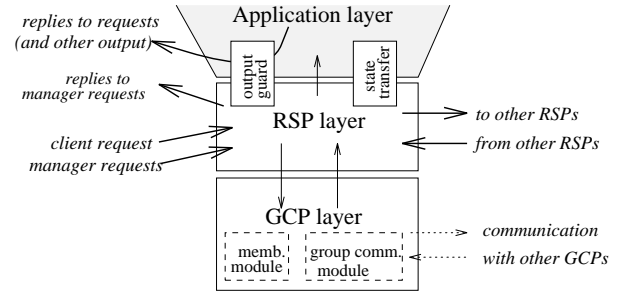


Figure 2: Structure of a replica server.

Whenever a request message m_{req} arrives at a replica, it is received by the *RSP* layer. This message can be *i)* a *client* request; *ii)* a *manager* request: requiring either the current group membership view or a replica removal; *iii)* another replica’s *RSP* layer message: requiring the current service (application) state and membership view.

According to its current view, a replica R_i applies the *responsibility rule* to decide whether it is responsible for the request or not. If R_i is decided not to be responsible, it just buffers m_{req} . Otherwise, it buffers m_{req} and broadcasts a message m_{id} containing m_{req} ’s *id* (which is unique in the system), through the *GCP* layer. This m_{id} is delivered reliably and ordered to all replicas *RSP* layers. 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 reflects of the request on the membership view in the *RSP* layers. The extended role of the responsible replica should be mentioned here: by broadcasting a special message through the *GCP* layer, it synchronises the effects of a request in the group.

If m_{req} is a manager request requiring the membership set, the responsible *RSP* replies to the manager with the current *view*. No m_{id} has to be broadcast and nothing is delivered to the application layer.

In the special case where m_{req} is another replica's R_j "join" message, R_i sends an m'_{req} with the current application state (see below) to R_j , and an m_{id} containing the id of m'_{req} .

How is the delivery of m_{id} messages from *GCP* to the *RSP* layer used for replica synchronisation, depends on the type of request to which an m_{id} is related to:

1. An m_{id} related to a *client request*. In that case, *RSP* delivers the related buffered m_{req} to the application layer and "notifies" (acknowledgement that is not necessarily reflected on a message transmission in *GCP*) m_{id} 's delivery from *GCP*. If m_{req} is not present in R_i , it is requested from other replicas (their *RSP* layers). Either one-to-one or multicast (not necessarily reliable) protocols can be used for this inter-*RSP* communication. The m_{req} delivered to the application is not removed from the local buffer, but is kept there for potential future retransmissions to other replicas. It is removed only when it becomes stable.

If a non-responsible replica has buffered an m_{req} but does not receive any related m_{id} within a timeout period, it retransmits (e.g. unreliably multicasts) m_{req} to the group.

When the responsible replica delivers back its own m_{id} (done in the same order with any other replica), it does not up-deliver the related client request m_{req} to the application layer before it becomes stable (i.e. every other replica has already up-delivered it). This is due to the "safe output" constraint. Stability of m_{req} is decided according to the stability of m_{id} in the group communication substrate. As far as every replica's *RSP* notifies the delivery of an m_{id} only after it up-delivers the related m_{req} , stability of m_{id} implies also stability of m_{req} .

2. A *view* m_{id} message indicating group membership change. We distinguish three kinds of such messages:

- Related to a manager *remover server* m_{req} . For a replica the responsibility role, in this case, means that it is the one that must be removed. Therefore, on receipt of this m_{id} , the *RSP* layer "commits suicide" on behalf of the replica. For any other *RSP*, such an m_{id} 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 when 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*).

It must be mentioned, that "view" m_{id} messages are also used in the *GCP* layer to maintain a consistent view of the group membership in that 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 sec.2. 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 [8] 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, m_{id} stability in the *GCP* layer guarantees the "safe output" property.

Virtual Synchrony: The *non-uniform vs*-model required for the *RSP* layer is directly based on the 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.

Because of the “notification” scheme of m_{id} s mentioned above, messages broadcast in GCP are tagged with context information (e.g. vector timestamp) that reflects the messages that have been explicitly acknowledged by the RSP layer; not the messages that have been up-delivered by GCP to RSP . Informally, we can argue that this acknowledging technique does not cause deadlock, since stability is a prerequisite for m_{id} ’s notification only in the responsible replica.

Concerning the *inter-RSP communication* (e.g. retransmissions of client requests), no special requirements are set. It can be an unreliable multicast or any one-to-one communication among replicas. The reason is that request delivery requirements are satisfied by using the reliable, ordered m_{id} broadcasts, and not the inter- RSP communication.

Some implementation related aspects

A replica owns a single thread of execution, the one of the application level implementation. 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.3).

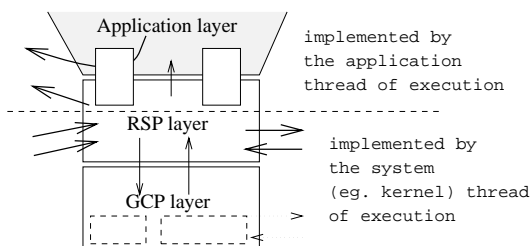


Figure 3: Layer implementation.

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 layer 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} transferring 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*” provided by the RSP layer. The output guard of just $resp(m)$ allows output due to m to be actually transmitted.

4 Discussion

Existing work

The architecture presented in the previous section is currently being implemented within the *Regis* test-bench. *Regis* [10] 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 of our architecture. The runtime system was extended to accommodate a group communication mechanism. It is based on hardware multicast (Deering’s IP extensions for multicast [6]) and 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*.

We have also completed a first implementation of the RSP layer. Table 1 presents some indicative performance results in our environment of Sun SPARC IPX workstations (network: loaded 10Mb ethernet;

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.

message size: 100bytes, both requests and replies). “Latency” stands for the average time a client waits to receive a reply to a request it invokes to the service. The presented times include 2.4msec (approximately) for client - group - client transmissions; the rest is the time required for the internal replica synchronisation. The first row is included in order to stress the performance overhead due to the *safe output* constraint.

The 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* [10], the configuration language used for structuring *Regis* programs. Client-service and service-server 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.

Conclusions & Future work

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

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] Kenneth Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.
- [3] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. Technical report, Computer Systems Research Institute, Univ. of Toronto, and Dept. of Computer Science, Cornell Univ., May 1994.
- [4] Flaviu Cristian. Understanding fault-tolerant distributed computing. *Communications of the ACM*, 34(2), February 1991.
- [5] 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.
- [6] S. Deering. RFC 1112: Host extensions for IP multicasting, 1989.
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] Andre Schiper and Alain Sandoz. Understanding the power of the virtually-synchronous model. In *5th European Workshop on Dependable Computing*, Lisbon, Portugal, February 1993.
- [14] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.