Client-Access Protocols for Replicated Services

Christos T. Karamanolis and Jeffrey N. Magee

Abstract—This paper addresses the problem of replicated service provision in distributed systems. Existing systems that follow the State Machine approach concentrate on the synchronization of the server replicas and do not consider the problem of client interaction with the server group. The paper analyzes client interaction and identifies a number of access protocols to meet a range of client requirements and system models. The paper demonstrates that protocols for the "open" group model—clients external to the group of servers—satisfy the requirements of the State Machine approach, even when replication is transparent to the clients. Experimental performance results indicate that the "open" model is clearly desirable when the service is used by a large, dynamically changing set of clients. The situation which pertains to Internet service provision.

Index Terms—Replication, availability, client-server, client access.

1 Introduction

Our increased reliance on computing systems for many aspects of day to day life suggests that we should consider increasing the availability of the services provided by these systems. The State Machine approach [27] is a general method for implementing highly available services by means of replication; that is, replicas of the servers providing the service are distributed on different processors in a distributed system. The approach sets the requirements for both client-server interaction as well as interserver coordination.

Existing research in the area has focused on the problem of interserver coordination. A range of low-level tools such as clock synchronization mechanisms, group communication protocols, and membership services are provided to the application programmer to implement replica server synchronization. However, the problem of client interaction with the server group is not explicitly addressed.

Most systems [22], [23], [18] assume that clients communicate with one of the replica servers, and that the latter acts as the representative of the client in the group by forwarding its requests to the other servers (this is also known as the "open" group model). It is, therefore, claimed that the problem of client interaction with the group is reduced to typical one-to-one communication. The implementation of the actual Client-Access protocol is left to the application programmer. These systems ignore the special problems of client-service interaction in the case of dynamic reconfiguration of the replicated server group. The programmer has to make sure, for example, that the results of a request persist in the service state despite service reconfiguration taking place concurrently to the processing of the request. Thus, some of the replication concerns affect the client and the server application program.

• *C.T. Karamanolis and J.N. Magee are with the Department of Computing, Imperial College of Science, Technology and Medicine, 180 Queen's Gate, London SW7 2BZ, UK. E-mail: {ctk, jnm}@doc.ic.ac.uk.*

Manuscript received 20 Nov. 1997; revised 27 Apr. 1997.
Recommended for acceptance by C. Ghezzi and M. Pezzè.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 106595.

In order to address these problems, systems like ISIS [14], Horus (CLTSVR layer) [28], and Transis [21] follow the "closed" group approach: clients are members (or at least special members) of the server group. The application programs of the clients and the servers employ group communication primitives which provide clear delivery semantics (atomicity, order) for requests and replies, even in the case of a dynamically changing environment. Although, this approach provides a straightforward solution to the problem of clients accessing dynamically reconfigurable replicated services, its performance implication is not clear from existing publications.

This paper addresses the problem of client-service interaction, in the case of replicated service provision. The fundamental requirements for state consistency between clients and servers are analyzed in Section 2. Section 3 outlines the assumed system model and the design principles followed in order to keep replication transparent to the application layer. A Replication protocol that conforms to the "closed" model is outlined in Section 4, with emphasis on the properties guaranteed to the clients. This protocol forms the basis for comparison with two instances of the "open" model (Section 5). The first demonstrates a novel access protocol implemented by a special communication stub in the client. Replication is kept transparent to the application program of the client. The second is a novel Replication protocol that completely hides replication from the clients (clients do not accommodate special communication stubs), at the price of higher response times. The correctness of the protocols is formally argued against the requirements of the State Machine approach. The section is concluded by stressing the orthogonal nature of the access method used by the clients and the Replication protocol used in the server group. The protocols are evaluated and compared using experimental performance results obtained by a first implementation in the Regis system (Section 6). The results show that only the "open" model offers viable solutions to the problem of replicated service provision in large, open distributed systems, such as the Internet. Section 7 discusses related work and summarizes the results of the paper.

2 Providing Highly Available Services

The State Machine approach is based on the assumption that a large class of service applications can be considered deterministic: the state transitions and the output of a server (state machine) are completely determined by the sequence of requests it processes, independent of time or any other activity in the system. As a result, when server replicas are introduced to improve availability, the nondeterministic event that must be synchronized among them is the delivery of client requests. By "delivery," we refer to message delivery to the application. The State Machine approach places two requirements on client requests,[1] with respect to preserving *internal service state* consistency [27]:

Delivery Agreement. If nonfaulty replica *p* delivers request *r*, then replica *q* eventually delivers *r* or *q* is faulty.

Delivery Order (Uniform). If nonfaulty replica *p* delivers requests *r* and *r′* and delivers *r* first, then replica *q* (whether correct or not) does not deliver request *r′* unless it has already delivered *r*.

These two requirements imply that all nonfaulty replicas deliver the same set of client requests and that they deliver them in the same relative order. Moreover, a replica does not violate the order of delivery even if it is faulty.

The typical requirements for validity and integrity of request delivery, which apply to the *nonreplicated* case, are restated as follows.

Delivery Validity. If nonfaulty client *c* transmits request *r*, then some replica *p* (whether correct or not) eventually delivers *r*, or there is no correct replica.

Delivery Integrity (Uniform). If replica *p* (whether correct or not) delivers a request *r*, then *p* delivers *r* only once and only if some client *c* has previously transmitted *r*.

In order to achieve overall *system state consistency*, we have identified two additional requirements. In the following definitions, we consider service output as equivalent to the transmission of replies back to clients. The problems which need to be addressed in the case of general service output are in principle the same as the problems discussed here for replies.

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

- If client *c* invokes request *r* and after that it invokes request *r′*, then replica *p* does not deliver *r′* unless it has already delivered *r*, or *p* is faulty.
- If request *r* of client *c* causally precedes [19] request *r′* of client *c′*, then replica *p* does not deliver *r′* unless it has already delivered *r*, or *p* is faulty.

Uniform Output

Agreement. If replica *p* (whether nonfaulty or faulty) delivers a request *r* and the sender of *r* is correct and receives a reply from *p*, then replica *q* eventually delivers *r*, or *q* is faulty.

Order. If replica *p* (whether nonfaulty or faulty) delivers requests *r* and *r′*, delivers *r* first and the sender of *r′* is correct and receives a reply from *p*, then replica *q* (whether nonfaulty or faulty) does not deliver request *r′* unless it has already delivered *r*.

The *Causality* requirements capture the semantics inherent in nonreplicated service provision. In most cases, clients adopt a synchronous style of interaction with the service, awaiting blocked (interacting neither among themselves nor with the service) for a reply to their last request. In this case, the the causality requirements are trivially guaranteed. When clients adopt an asynchronous style of interaction with the service and causal consistency among clients is of importance, then request messages must be timestamped by means of logical or physical clocks and these times must be respected by the delivery order on the server side [19], [27]. Throughout this paper, we are only concerned with synchronous interaction primitives (such as Remote Procedure Call-RPC) and therefore we do not make any explicit arguments about Causality.

The *Uniform Output* requirement, which is not explicitly stated in the literature of the State Machine approach, is of importance in systems where the membership of the replica server group changes dynamically [16]. *Uniform Output Agreement* requires that, if output is produced by the service as a result of processing request *r*, then the results of *r* persist on the service state. For example, consider the scenario in which request *r* of client *c* is received and delivered by replica server *s* of service *S*; the server processes *r* and produces a reply *r′* which is sent back to *c*; after that, *s* crashes and because of a combination of communication failures no other server of *S* has the chance to receive and deliver *r* (the *Delivery Agreement* requirement does not apply, since *s* is faulty). As a result, the state of service *S* does not reflect the results of request *r* and is inconsistent with the state of client *c* (although the surviving servers have mutually consistent states).

In the general case, where the results of the requests are noncommutative, Uniform Output Agreement is not enough to guarantee that the state of the correct (surviving) replicas is consistent with the clients' state. If output has been produced for certain requests by a replica that fails, then the surviving replicas must deliver the requests at hand in the same relative order as the replica that originally produced the output did. The latter requirement is stated explicitly by the *Uniform Output Order* property. Since the original delivery order respected the causal dependencies of the requests, this is also implied by the uniformity property for the surviving replicas.

The Uniform Output requirement is, in the general case, less strict than a combination of the uniform counterparts of the Delivery Agreement and Order properties. This is due to the fact that uniformity is required, in our case, only when output is produced to system entities external to the server group; not all client-service interaction primitives require the transmission of output. Intuitively, in the first case, request delivery must be delayed until uniformity is guaranteed, while in the second case, output is delayed or recorded (only when produced) to guarantee uniformity.

1. All the requirements presented here refer to client requests invoked to a specific replicated service. By the term "replica" we refer to a replica server of that service..

3 CLIENT-ACCESS PROTOCOLS

3.1 System Model

In the following, we assume a *message-passing asynchronous system*. The communication network provides an *unreliable datagram service*, which exhibits benign failures; messages may be delivered out of order and may be arbitrarily duplicated, but there are no spurious messages generated. The network supports an unreliable multicast primitive realized as a multidestination addressing scheme. Processors and processes fail in a graceful way, by crashing.

According to the State Machine approach, any Replication and corresponding Client-Access Protocol must satisfy the Uniform Output requirements in the face of dynamic system reconfiguration. Reconfiguration may occur for two reasons: 1) system changes such as processor failures, process crashes and network partitioning, and 2) explicit management operations such as removal and addition of servers providing a service and creation / deletion of clients using that service. The latter operations are the subject of a Configuration Management system service [9], an area out of the scope of this paper. Instead, we concentrate on the former dimension of server group reconfiguration, namely process failures and system partitioning. For reasons that will be discussed later in the paper, we adopt the primary partition model [26]. Thus, no distinction is made between a failed entity and one isolated from the main partition.

We assume that clients access services using a synchronous (request-reply) communication primitive, such as a Remote Procedure Call (RPC), or a Remote Method Invocation primitive in the case of object structured systems. These primitives are typically built on top of an unreliable datagram communication service. Defining the exact properties of remote invocation is essential for the design of the replication related protocol modules in the clients and the servers. In most cases, where RPC provides *exactly-once* delivery guarantees [5], the operation of the RPC end-points can be summarized to the following.

- *Client end-point.* Buffers the last request until a reply is received. Retransmits the last request, if no reply is received within a timeout period. Detects duplicate / old replies and discards them.
- *Server end-point.* Buffers the last reply to every service client *c*, until the next request of *c* is received. Detects duplicate / old requests: if the last request of client *c* is received again, then the reply is retransmitted; if a request older than the last request of *c* is received, then it is discarded.

For the definitions above, it is assumed that servers do not fail. In practice, existing RPC implementations use timeouts or the output of an (unreliable) failure detector [17] to speculate about server failures. In case of suspected server failure, a rebinding protocol is employed at the client end-point. Similarly, the server end-point uses a mechanism based on some type of weakly synchronized time to garbage-collect "old" buffered requests of clients that do not use the service anymore.

3.2 Architectural Requirements

As discussed in the introduction, an important requirement for every Replication and Client-Access protocol, not met by existing systems, is to make replication transparent to the application layer. The application programmer should not need to change the program of the client and / or server to cater for a specific replication method. In this paper, we propose Replication and Client-Access protocols that live in the communication substrate of clients and servers. The novel aspect of all these protocols is that they keep replication transparent to the application algorithm. The structure of a replica server providing a highly available service and the structure of a client using that service are as depicted in Fig. 1.

- *Server.* The communication substrate is augmented with a replication specific layer, which accommodates two protocols. The *Client-Access Protocol* receives and handles client requests and replies. It synchronises replica output according to an output policy for the service—single output, in the case of benign failures. The *Replication Protocol* synchronizes request delivery amongst replica servers, in accordance with the Agreement, Order, and Causality requirements of the State Machine approach. To avoid the introduction of too many layers and interfaces in the design, the two protocols are described as an integral module of the server communication substrate.
- *Client.* The communication substrate is (in the general case) augmented with a special communication stub, which implements the client's part of the Client-Access protocol.

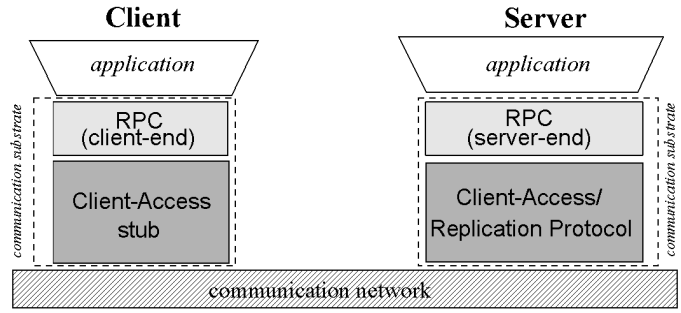


Fig.1. The structure of a replica server and a client—the general case.

The design of the proposed replication modules (Client-Access stub and Client-Access / Replication Protocol) assumes a well-defined functionality for the adjacent layers. The specifications of these layers, namely RPC and datagram communication service, have been outlined in the previous section.

The new replication modules satisfy the requirements of the State Machine approach, but they do not provide any additional properties, such as reliability or atomicity, for client-service communication on top of the properties provided by the datagram service in the nonreplicated case. Thus, typical RPC end-points, designed for the nonreplicated case, can be reused in combination with the replication modules. The interface of the replication module includes two basic primitives:

- `send()`. Invoked by the RPC end-point for the transmission of requests in the clients and replies in the servers.

- `receive()`. Invoked by the datagram service for the delivery of requests in the servers and replies in the clients.

The replication module invokes the `deliver()` primitive of the RPC end-point to deliver requests (servers) or replies (clients) to the application layer. It invokes the `transmit()` primitive of the datagram service to transmit request (clients) or reply (servers) messages through the communication network. The transmission is a unicast, unless if a multicast reference is explicitly specified, in which case it is an unreliable multicast.

3.3 Group Communication

In the following sections, a modular approach is followed for the design of the replication modules. The design of the proposed Replication protocols is greatly simplified if we assume the existence of a Group Communication Protocol. The Group Communication Protocol is used for the diffusion of synchronization information among replicated servers (or, in some cases, among servers and clients).

Group Communication Protocols (GCP) are characterised by a "many-to-many" model of communication within a group of system entities, server replicas in our case. They guarantee that messages multicast to the whole group, by members of the group, are delivered in some mutually consistent way by all members of the group including the sender. We assume, here, the existence of a protocol that combines typical properties of systems from the literature. A protocol such as Horus [28], Newtop [12], or RELACS [2] could be used to implement the functionality of the required GCP module. The required properties of GCP are outlined below. They apply to messages that are multicast within a specific group *g*, although we do not refer to the group explicitly.

- **GCP-Reliability**
  - *Validity*. If member *p* multicasts message *m*, then *p* eventually delivers *m* or *p* is faulty.
  - *Agreement*. If correct member *p* delivers message *m*, then member *q* eventually delivers *m* or *q* is faulty.
  - *Integrity* (*Uniform*). For any message *m*, every member (whether correct or faulty) that delivers *m*, delivers it at most once and only if some member has previously multicast *m*.
- **GCP-Total Order** (**Uniform**). If correct member *p* delivers messages *m* and *m′* and delivers *m* first, then member *q* (whether correct or not) does not deliver *m′* unless it has already delivered *m*.

It is known from practical experience [28], [14] that algorithms for Uniform Total Order perform better, in terms of delivery latency, compared with algorithms for other uniform properties, such as Uniform Agreement. This is due to the fact that all known algorithms for Uniform Agreement are blocking, i.e., messages may be delayed (blocked) before delivery on the recipient in order to guarantee uniformity (messages must be stable in the group before delivery). On the other hand, there are algorithms, such as token passing algorithms, for Uniform Total Order, which are nonblocking [1]. In fact, uniformity is provided as a "side-effect" of the Total Order algorithm in these cases.

Group membership changes are also recorded and agreed upon in GCP. Membership information is recorded in the form of 'views,' which are vectors of process identities. For GCP to provide useful membership information, the contents of the views must reflect the actual condition of the system as far as member failures, voluntary removals, or joins are concerned.

The membership protocol of GCP uses a *failure detector* system service [7] to retrieve information about the status of group members. The failure detector suspects and reports failed system entities (processes), without necessarily making complete and accurate suspicions. Babaoglu et al. [3] have shown how to implement a membership protocol for asynchronous distributed systems using an eventually perfect ($\diamond P$) failure detector. In theory, it is impossible to implement an eventually perfect failure detector in a completely asynchronous system. In practice, however, eventually perfect failure detectors can be implemented, by making timeliness assumptions that are reasonable for actual systems.

In the following specifications of GCP membership, we have adopted the primary partition approach [26], as the most general model for any type of application semantics. According to this model, there is a *total order* of membership views installed in the correct members of the group. It has been shown [6], that membership agreement protocols may not terminate in asynchronous systems, when a primary partition has to be formed. In the assumed system model, the solution of this problem is delegated to a Configuration Management system service, where the decision about which partition should be augmented with new members in order to become operational is done in a heuristic way [14].

- **GCP-Membership**
  - *View accuracy*. If member *q* is correct, then eventually the current view of correct member *p* will always include *q*. If *q* joins the group, then eventually the current view of correct member *p* will always include *q*.
  - *View completeness*. If member *q* has failed or has voluntarily left the group, then eventually the current view of every correct member *p* will always exclude *q*.
  - *View integrity*. Every view installed by member *p* (whether correct or not) includes *p* itself.
  - *View agreement*. If correct member *p* installs view *v*, then for every member *q* in *v* either *q* eventually installs *v*, or *q* is not correct and *p* eventually installs an immediate successor to *v* that excludes *q*.

In the case of dynamic groups, it is important to consider the context messages are multicast and delivered in. We say that a message *m* is multicast (delivered) *in view v* by member *p*, when *v* is the last view installed in *p* before it multicast (delivered) *m*. In addition to the Reliability properties above, GCP is required to satisfy the following message delivery properties in order to provide a virtually synchronous communication behavior among group members [4], [13]. In particular, correct group members must deliver messages in the same view, which is not necessarily the same as the view the messages have been multicast in.

- **GCP-Virtual Synchrony**. If correct members p and q both install successive views v_i and v_{i+1}, then p delivers message m in view v_i if and only if q delivers m in v_i.

Message *stability* is another important concept of Group Communication Protocols. It refers to 'global knowledge' concerning delivery of messages in the group.

- **GCP-Stability**. A message m is stable in the group, if m has been delivered by every correct member of the group and its delivery has been explicitly acknowledged by all correct members.

The definition implies that message stability is determined by means of delivery acknowledgment down-calls from the "application" layer to the Group Communication Protocol. A similar approach is also followed in Horus [28].

The interface that GCP provides to its "application" layer (the Replication Protocols, in our case) is depicted in Table 1. GCP requires a typical `deliver()` primitive from the layer above for the delivery of messages, and an optional `install-view()` when membership information must be passed upwards. `install-view()` is called as soon as a new view is installed in GCP (before any messages are delivered in the new view).

TABLE 1
THE INTERFACE OF GCP TO THE LAYER ABOVE

Primitive	Description
`G-mcast(`m`)`	Multicast message m to the group
`G-ack(`id`)`	Acknowledge delivery of message with identity id
`G-stable(`id`)`	Check for stability of delivered message with identity id
`G-join(`g`)`	Add caller to the group with reference g
`G-leave()`	Remove caller from the group it currently joins

4 THE "CLOSED" GROUP MODEL

As mentioned earlier, ISIS and Transis favor a model where clients form a group together with the servers that provide the replicated service. Servers maintain a consistent membership view of the client set and vice versa. In this way, the delivery properties of group communication protocols are directly exploited to satisfy the requirements of the State Machine approach. Client requests and replies from the servers are multicast to the entire group in a reliable way (see Fig. 2).

4.1 Protocol 1: Clients Members of the Server Group

The protocol presented in this section exhibits the basic principles of the "closed" group model [28]. In this respect, the protocol is not original and it is introduced mainly as a reference point in assessing the performance of the protocols of Section 5. The original aspect of the presented protocol is that it is implemented in the communication substrate of clients and servers. Replication is transparent to the application. Client and server application programs are implemented using the nonreplicated RPC primitives. To the best knowledge of the authors, neither ISIS nor Horus currently satisfy this transparency requirement.

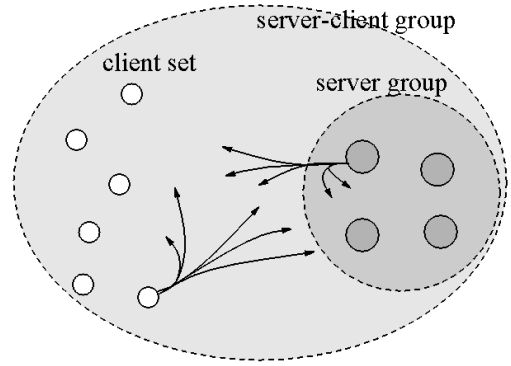


Fig. 2. The "closed" group model.

The structure of the communication substrate of clients and servers is depicted in Fig. 3. The replication modules consist, in this case, of two layers. A Group Communication Protocol (GCP) layer provides reliable, ordered multicast in the client-server group. A Client or Server Replication Filter is placed on top of GCP, on clients and servers respectively, processing request and reply messages in order to satisfy the State Machine requirements. Requests and replies are all multicast to the whole group (clients and servers) through GCP. The replication filters let only messages that are of interest to the local entity pass to the application layer through RPC. For example, if client c is waiting for a reply to its last request, the Client-Access module of c filters out any other message that is delivered from the group through GCP. In the closed group model, the Replication protocol is basically implemented by the functionality of the GCP module. The design of the Client-Access and Server Replication Filters is kept simple.

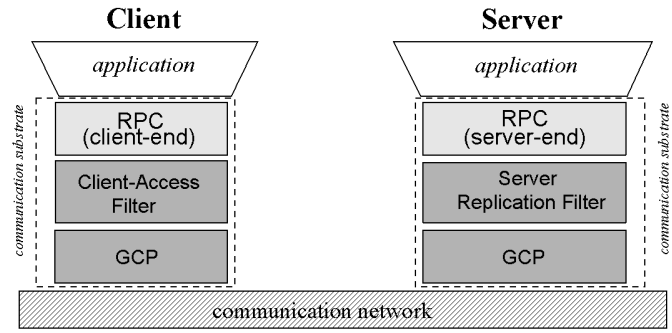


Fig. 3. "Closed" group model—structure of clients and replica servers.

4.1.1 Protocol Description

Binding. The bind primitive of RPC initiates a join procedure in the Client-Access Filter. The `G-join()` primitive of GCP is invoked with the group reference as a parameter.

Request delivery. The application program of a client invokes a request to the service by calling the corresponding primitive of the RPC client end-point parameterised with the request message. In turn, RPC calls the `send()` primitive of the Client-Access Filter (see Fig. 4). Every submitted request message is multicast by `send()` to the whole group through the GCP layer (using `G-mcast()`).

On delivery of a request message from GCP, the Replication Filter of a server passes the request directly to the application by calling RPC's `deliver()` (see Fig. 5). Client–

```
Client−Access Filter  (client c_i :)
Initially:
    last-req = -1 // id of last invoked request
    request-back = 0 // flag: last request delivered back?
procedure Client::send(m): // from RPC (invoke request)
1   last-req := m.id;
    request-back := 0;
    G-mcast(m); // through GCP
end; // procedure send(m)

procedure Client::deliver(m): // from GCP
1   if m.type = request and m.sender = c_i
            and m.id = last-req then
        request-back := 1;
    else if m.type = reply and m.to = c_i then
        if m.reply-to = last-req and request-back then
5           deliver(m); // to RPC
    // else discard m
end; // procedure deliver(m)
```

Fig. 4. Protocol 1: client-access filter (pseudocode).

```
Server Replication Filter  (server s_i)
Initially:
    reply-flag = 0 // transmit reply to last delivered request?
    view = ∅ // current membership view
procedure Server::send(m): // from RPC (send reply)
1   if reply-flag then
        G-mcast(m); // multicast reply m
end; // procedure send(m)

procedure Server::deliver(m): // from GCP
1   if m.type = request then
        if DECISION(view, m.id) = s_i then
            reply-flag := 1;
        else reply-flag := 0;
5       deliver(m); // to RPC
end; // procedure deliver(m)
```

Fig. 5. Protocol 1: server replication filter (pseudocode).

Access Filters discard any request messages delivered from GCP; they do not pass them to the application.

Reply transmission. Every server program that processes a request *r* attempts to transmit a reply back to the sender of *r*, by calling the `reply()` primitive of the RPC server end-point, which in turn calls the `send()` of the Replication Filter. Following a *single output policy*, the Replication Filter of only one server in the group allows the reply message to be actually transmitted back through *GCP*. The decision is made when the corresponding request is delivered to the application (the reasons will become apparent in the correctness arguments) and it is based on the current membership view (a common view is provided to all group members by *GCP*) and it can be made in a distributed manner. The decision as to which server replies requires only that the choice procedure makes the same decision at each server. For example, an ordering on server identities can be used to choose the one with the smallest id. For efficiency, it might be sensible to choose the nearest replica to the client if this decision can be made unambiguously at each server. In the pseudocode description, the decision scheme is represented by the DECISION() function on the view vector and the unique message id (Fig. 5).

Replication Filters of servers discard all reply messages delivered from GCP. A Client-Access Filter discards all reply messages, except the one that refers to the latest request of the local client. This reply message is passed to the application through RPC's `deliver()`.

A straightforward optimization would be for the server to unicast the reply to the client directly through the communication network.

Group reconfiguration. The Virtual Synchrony property of GCP guaranties that whenever the Replication Filters of two correct servers deliver a message *m*, they both "see" the same group membership view. Therefore, the two servers make mutually consistent decisions about who is to send the reply for *m*, even in the presence of group reconfiguration.

However, the potential server failures introduce another problem related to the requirement for Uniform Output. Consider server *s* that delivers a request *r* from client *c*, in some view v_i; *s* decides that it takes responsibility to reply to *r* and then it transmits a reply *r′* back to *c*. Immediately after that, *s* fails causing a view v_{i+1} to be installed in the surviving group members. Even if *c* delivers the reply *r′* back (*r′* delivered from GCP to all correct members), Virtual Synchrony does *not* guarantee that the correct servers deliver *r* in some view (v_i or later).

To address this problem, the Client-Access Filter of client *c* does not deliver to RPC a reply to request *r*, unless request *r* itself has been already delivered back from GCP.

4.1.2 Correctness Arguments

We argue that the proposed Client-Access and Replication Filters, in combination with the assumed GCP and RPC modules, satisfy the requirements of the State Machine approach. It is reminded that by "delivery" we mean, here, delivery of requests to the application layer. As explained in Section 2, we do not have to argue about Causality, which is trivially satisfied since RPC has been used.

THEOREM 4.1. *The protocol of Figs. 4 and 5 satisfies the Delivery Validity property.*

PROOF. If correct client *c* invokes request *r* to the service, then the Client-Access (CA) Filter of *c* multicasts *r* to the group through GCP (Client::send():3).[2] Let v_i be the view that *c* multicasts *r* in. GCP Validity guarantees (even in the presence of communication failures and / or group reconfiguration) that *r* is delivered back from GCP to CA of correct client *c*, in some view v_{i+k} ($k \geq 0$). From GCP Agreement and Virtual Synchrony, *r* is also delivered from GCP in all correct members, in v_{i+k}. Thus, *r* is eventually delivered from GCP to the Server Replication (SR) Filter of some replica (or no correct replica exists). A request is never discarded or blocked in the SR Filter of a replica (whether correct or not). Thus, *r* is delivered to RPC of some replica

2. The number after ":" denotes the line number in the pseudocode description of the procedure.

(Server::deliver():5). There, if r has already been delivered, we are done; if this is the first time r is delivered, then it is passed to the application and Validity holds. □

THEOREM 4.2. *The protocol of Figs. 4 and 5 and satisfies the Uniform Delivery Integrity property.*

PROOF. From the algorithm, replica p delivers request r only if it has previously executed Server::deliver(r). That is, r has been delivered from GCP to SR. From Uniform GCP Integrity, some member of the group has previously multicast r. Only clients multicast requests to the group (Client::send():3). Therefore, some client has previously multicast r. The RPC end-point guarantees that no request is delivered to the application more than once. Thus, Uniform Delivery Integrity holds. □

THEOREM 4.3. *The protocol of Figs. 4 and 5 and satisfies the Delivery Agreement property.*

PROOF. Let p and q be two correct replica servers. We must show that if p delivers request r then q also delivers r. Since, p delivers r, then r has been delivered from GCP to the SR Filter in p (Server::deliver()). If no group reconfiguration occurs (in the presence, though, of potential communication failures), GCP Agreement guarantees that r is eventually delivered to the SR Filter of correct replica q. Requests are immediately passed by SR to RPC (Server::deliver():5) and we are done.

In the presence of group reconfiguration (including replica failures), GCP guarantees Virtual Synchrony. That is, correct replicas p and q either both deliver r in the same view, or none of them delivers r. In both cases we are done. □

THEOREM 4.4. *The protocol of Figs. 4 and 5 satisfies the Uniform Delivery Order property.*

PROOF. Let correct replica p, which delivers request r_i of client c_i and request r_j of c_j and delivers r_i before r_j. Requests are not blocked in the SR filter and they are delivered to the application through RPC as soon as they are delivered from GCP (Server::deliver()). Thus, GCP delivers r_i before r_j to SR, in replica p. (1)

Let replica q (not necessarily correct), which delivers r_j (to the application through RPC). Since r_j is delivered from the SR filter to RPC (Server::deliver():5), r_j has been already delivered from GCP to SR (Server::deliver()), in q. By Uniform GCP Total Order, (1) implies (even in the presence of communication and/or process failures) that r_i has been already delivered, in q, from GCP to SR. Delivered requests are never blocked in SR. Thus, r_i has been already delivered from SR to the application through RPC, in q, and we are done. □

THEOREM 4.5. *The protocol of Figs. 4 and 5 satisfies the Uniform Output Agreement property.*

PROOF. Assume that replica p, which delivers request r of client c in view v_i, decides to take responsibility for the reply to r and actually transmits (through the SR Filter) a reply r' to r. This means that, on delivery of m, p has decided to take responsibility for the transmission of the reply to r (Server::deliver():2-3). If p is correct, then it is straightforward to show that GCP Validity and Agreement guarantee Output Agreement among the replica servers, even in the presence of communication failures. Similarly, in the case where new replicas join the group, replicas voluntarily leave the group, or replicas other than p fail, GCP Virtual Synchrony guarantees that Output Agreement is achieved among replicas that share the same view.

Assume, now, that p is not correct. That is, p is not included in v_{i+1}. If c is faulty, then Uniform Output is not required. If c is correct, then let v_{i+k} ($k \geq 0$) be the view in which the CA Filter of c delivers r' to RPC for the first time. For r' to be delivered to RPC, the protocol guarantees that *request-back* = 1 (Client::deliver():4). This flag is set only when the corresponding request (referenced in r'.*reply–to*) has been already delivered back from GCP to CA of its sender (Client::deliver():1-2). Thus, r has been delivered back to c in some view v_{i+m} ($m \leq k$). From GCP Agreement, r is also delivered in every correct group member, including the correct servers, in v_{i+m}.

The latter argument implies that if correct client c delivers reply r' for request r, then all correct replica servers deliver r in the same or an earlier view as the one in which r' is delivered to c. Thus, Uniform Output Agreement is satisfied. □

Uniform Output Agreement, in combination with Delivery Validity, guarantee that if client c invokes request r and remains correct for long enough and there is at least one correct replica, then c eventually delivers back a reply to r and the results of r persist on the service state despite potential server failures.

THEOREM 4.6. *The protocol of Figs. 4 and 5 satisfies the Uniform Output Order property.*

PROOF. Let replica p, which delivers requests r_i of client c_i and r_j of c_j, r_i first, in views v_i and v_j respectively ($i \leq j$). Assume that replica p decides, in v_j, to transmit the reply r'_j back to r_j and actually c_j delivers r'_j. Again, we focus on the case where p is faulty; that is, p in not included in v_{j+1}. Since request r_i is delivered to the application layer of p before r_j, it is implied that r_i is delivered before r_j from GCP to SR in p. Uniform GCP Total Order guarantees that GCP does not deliver r_j to a correct group member, unless it has already delivered r_i. Thus, r_j is not delivered from SR to RPC in a correct replica, unless r_i has already been delivered. □

The latter two arguments imply that if correct clients deliver replies from a faulty replica, then the surviving replicas deliver all the corresponding requests and deliver them in the same relative order as the faulty replica originally did. However, this does not mean that clients always receive replies transmitted by faulty replicas. Reliable client–service communication is implemented by the RPC communication end-points.

It has been shown that the proposed protocol (Protocol 1) is robust against benign communication and process failures, by exploiting a fault tolerant Group Communication Protocol that provides Virtual Synchrony.

5 The "Open" Group Model

In this model, the clients are external to the group of servers. Servers do not maintain a consistent view of the client set. The "open" model is suitable for environments where a service is used by a large and dynamically changing set of short-lived clients, and where the service is provided by a relatively small and stable group of long-lived servers (see Fig. 6). Two subcases of this model can be distinguished with respect to the requirements of applications.

1) Clients are aware of replication; that is, they can accommodate special communication stubs. This is the case in smallomogeneous environments, where clients can be linked to replication specific stubs in order to communicate with groups of servers.
2) Replication must be completely transparent to the client [17]. This is typically the case in open distributed systems, where clients cannot necessarily be reprogrammed or relinked to cope with replication. A similar requirement occurs in environments where we wish to permit dynamic (on-line) replacement of non-replicated servers by groups of servers, as part of the system configuration management.

A Client-Access protocol and a corresponding Replication protocol are discussed for each of these cases.

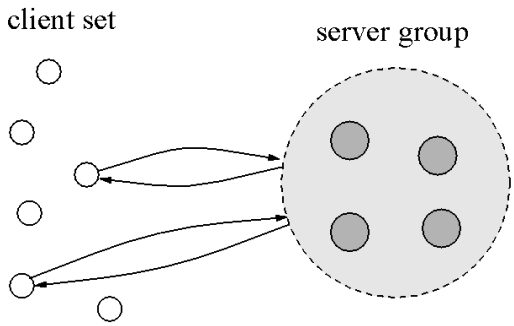


Fig. 6. The "open" group model.

5.1 Protocol 2: Clients Aware of Replication

Systems that adopt this model [22], [23], [18] focus on internal server synchronization and consider the Client–Access protocol as an application level concern. Even when a generic Replication layer is introduced in the servers, on top of the Group Communication Protocol [22], the client application programmer must still deal explicitly with replication concerns. In the following, a novel Replication protocol is proposed, which is in line with this model (clients aware of replication), but keeps replication transparent to the application algorithm of both clients and servers.

The Client-Access protocol is implemented by a stub in the client's communication substrate. The stub cooperates with the Replication protocol on the server to meet the State Machine requirements. Request messages invoked by a client

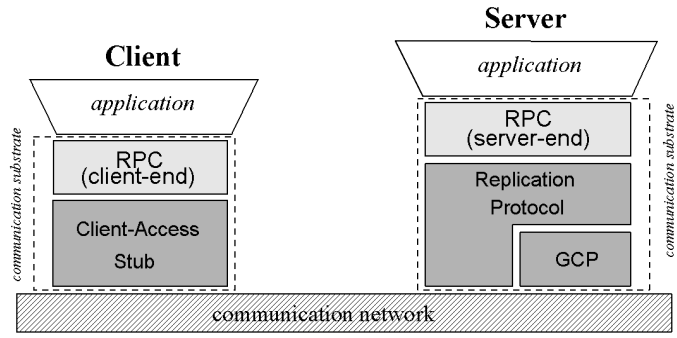


Fig. 7. Protocol 2: structure of clients and replica servers.

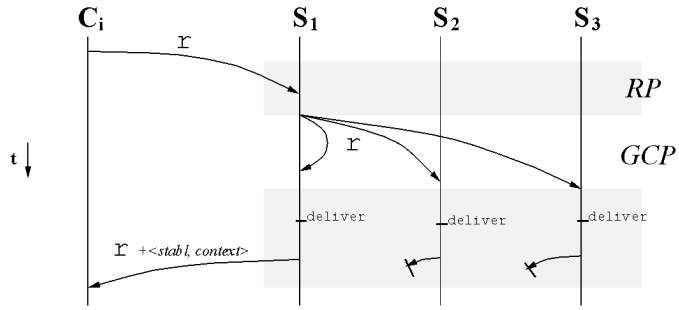


Fig. 8. Protocol 2: message diagram (no group reconfiguration).

are transmitted to a "representative" server of the client in the group. This server diffuses the request to the group through a Group Communication Protocol (GCP). The Replication protocol incorporates a nontrivial algorithm to satisfy Uniformity in the case of group reconfiguration. Fig. 7 illustrates the structure of clients and servers for this protocol. The message diagram of Fig. 8 introduces the basic principles of the protocol functionality (the 'deliver' event of the diagram refers to request delivery from the Replication Protocol layer to the RPC end-point).

5.1.1 Protocol Description

Binding. The Client-Access stub resolves the multidestination reference of a replicated service (for example, by contacting a name service) and binds to a single replica server according to a *binding policy* (taking into account, for example, client vicinity to servers or load balancing). This server acts as the representative of the client in the group. The server maintains a list with the references of the clients that are bound to it, denoted `ClientRef`. The contents of this list do not form a consistent membership view of the client set. `ClientRef` is periodically multicast in the group (through GCP), so that servers in the group have a global "hint" of client allocations to replica servers. A vector of reference lists is maintained in the Replication Protocol layer of each server:

- $\texttt{ClientRef}_s[s_i]$. List of client references; denotes the hint that server s has about the clients allocated to server s_i ($\texttt{ClientRef}_s[s]$ is s's own list).

The server replication module can use a heuristic method (based, for example, on weakly synchronized clocks) to remove from the list clients that have not interacted with the server for a long time.

```
Client—Access Stub (client c_i):
Initially:
    Epoch: initialised during binding (not presented here)
procedure Client::send(m): // from RPC (invoke request)
1   buffer m in transmittedRequests;
    m.epoch := Epoch;
    transmit(m); // through the comm. network
end; // procedure send(m)

procedure Client::receive(m): // from comm. network
1   if m.type = reply then
        for each m' ∈ transmittedRequests:
                m'.id ≤ m.LastStable do
            garbage-collect m';
        if m'' ∈ transmittedRequests:
                m''.id = m.ReplyTo then
5           m''.lastReq[] := m.lastReq[];
        deliver(m); // to RPC
    else if m.type = retransmission request then
        Epoch := m.epoch;
        for each m' ∈ transmittedRequests do
10          ReplayList := ReplayList +
                ⟨m'.id, m'.lastReq[]⟩;
        transmit(ReplayList);
        for each m' ∈ transmittedRequests do
            m'.epoch := Epoch;
            m'.type := retransmitted request ;
15          transmit(m'); // re-transmit
    else if m.type = new Epoch then
        Epoch := m.epoch;
        transmit( Epoch OK );
end; // procedure deliver(m)
```

Fig. 9. Protocol 2: client-access stub (pseudocode).

Request delivery. The client transmits a request *r* to the single server replica *s* it is bound to (Client::send():3 in Fig. 9). The request message *r* is received by the Replication Protocol (RP) layer of *s*, which then multicasts *r* to the group using the `G-mcast()` primitive of GCP (Server::receive():3 in Fig. 10). The RP layer of every server in the group (including *s*) delivers *r* to the application (through the RPC end-point) as soon as *r* is delivered from GCP (Server::deliver():2). The following lemma can be easily shown.

LEMMA 5.1. *If $r_{c,s}$ the last request of client c delivered by server s, then s has also delivered every request of c invoked before $r_{c,s}$.*

Reply transmission. Following a single output policy, the *RP* layer of only the representative server *s* actually transmits the replies to the requests of *c* (see Server::send()).

The Client-Access protocol does not guarantee reliable client-server communication. Requests or replies can be lost in the communication network. The replication layers of clients and servers do not handle message retransmission and duplicate detection, since the context of messages is transparent to them. Reliable client-server interaction is implemented by the RPC end-points in the same way as in the nonreplicated case.

In order to cope with group reconfiguration (see correctness arguments), the *Client Stub* buffers a request to the service for potential future retransmission (Client::send():1), even after a reply is received back for the request. A request *r* is garbage-collected in the Client Stub, when the client learns that *r* has become stable in the group; that is, it has been received and delivered by all correct replica servers in the group. Reply messages to client *c* are piggy-backed with information indicating the most recent request of *c* that has become stable in the group (Server::send():2-3).[3] Note that, if request *r* of client *c* is stable, then any older request of *c* is also stable in the server group. The following lemma can be easily derived from Lemma 5.1.

```
Server Replication Protocol (server s_k):
Initially:
    ClientRef[] initialised during join procedure
    lastReq[i] = -1 for all i ∈ ClientRef[]
    Epoch = 0 // epoch number
    reconfiguration-phase = 0 // flag
procedure Server::receive(m): // from comm. network
1   if m.epoch = Epoch then
        if not reconfiguration-phase then
            G-mcast(m); // to the group
        else // Reconfiguration taking place
5           if m.type = retransmitted request then
                if m.lastReq[] ≤ lastReq[] then
                    G-mcast(m);
                else buffer m in retransmittedRequests;
            else block m in receivedRequests;
end; // procedure receive(m)

procedure Server::send(m): // from RPC (send reply m)
1   if MYCLIENT(m.to) then
        m.lastReq[] := lastReq[];
        m.lastStable := max { m'.id:
            m'.sender = m.to and G-stable(m'.id) };
        transmit(m); // through the communication network
end; // procedure send(m)

procedure Server::deliver(m): // from GCP
1   lastReq[m.sender] := m.id;
    deliver(m); // to RPC
    if reconfiguration-phase then
        while ∃m' ∈ retransmittedRequests:
                m'.lastReq[] ≤ lastReq[] then
5           remove m' from retransmittedRequests;
            G-mcast(m');
        if ∄m' ∈ retransmittedRequests and
            no more expected retransmissions from C_k then
                Terminate-Reconfiguration-Phase();
    else if m.type = ClientRef then
10      ClientRef[m.sender] := m.ClientRef[];
end; // procedure deliver(m)

procedure Server::install-view(v): // from GCP
1   if view - v ≠ ∅ then // replicas have been removed
        Epoch := Epoch + 1;
        for all c ∈ ClientRef[s_k] do
            do transmit(Epoch);
5           until reply received or
                c removed from ClientRef[s_k];
        reconfiguration-phase := 1;
        Reconfiguration-Phase(v);
    view := v;
end; // procedure install-view(v)
```

Fig. 10. Protocol 2: server replication protocol (pseudocode).

3. The synchronous style of client-service interaction imposed by RPC defines a total ordering of the requests invoked by a single client. We assume that the relative position of requests in this order can be derived from the message identities.

LEMMA 5.2. *At any moment in time,*[4] *the requests of client c that are buffered in the Client Stub of c are a superset of the requests of c that are unstable in the server group.*

The Replication Protocol of server s maintains a vector $\texttt{lastReq}_s[]$.

- $\texttt{lastReq}_s[c]$: the identity of the last request of client c *delivered* by server s (application), for every client that accesses the service (updated in Server::deliver():1).

The vector $\texttt{lastReq}_s[]$ is a record of the causal dependencies among client requests. Its value, on delivery of request r, indicates the requests (of every service client) that have been delivered before r, by server s. The value of $\texttt{lastReq}_s[]$, on delivery of request r, is piggy-backed to reply r' which is transmitted back to the sender of r (Server::send():2). On receipt of the reply, the Client Stub buffers the vector together with request r (Client::receive():5). As discussed in the following paragraphs, the contents of the buffered vector are used for the reconstruction of the service state in the presence of replica failures.

Group reconfiguration. In case of group reconfiguration, the Virtual Synchronous behavior of GCP guarantees that correct replica servers deliver the same set of client requests in each view. However, if a faulty replica delivers request r of client c in some view v_i, then there are no guarantees that correct replicas also deliver r in v_i. Moreover, they may not deliver r at all, even if c receives back a reply from the faulty replica. The latter is clearly a Uniform Output concern.

Client reallocation. As soon as a new view is installed in the server group indicating the failure of one or more servers (Server::install-view():1), the Replication Protocol enters a *Reconfiguration Phase*. Let server s, which fails causing view v_i to be installed. Correct server $s_k \in v_i$ traverses $\texttt{ClientRef}_{s_k}[s]$ and decides about which of the clients that where originally assigned to s are to be reassigned to s_k according to the applied binding policy (this decision can be made in a deterministic distributed way in the new view). Let C_k be the set of these clients ($C_k \subseteq \texttt{ClientRef}_{s_k}[s]$). A special message is transmitted by s_k to each client $c \in C_k$ indicating the last request of c that s_k (and therefore any other member of v_i) has delivered in v_{i-1} (Server::Reconfiguration-Phase():3-5). That is, the message carries the value of $\texttt{lastReq}_{s_k}[c]$. With this message, correct server s_k requires from c the retransmission of any buffered requests "newer" than $\texttt{lastReq}_{s_k}[c]$. From the synchronous style of interaction of RPC and the Virtually Synchronous behavior of GCP, it is easy to derive the following lemma.

LEMMA 5.3. *When view v_{i+1} is installed, all correct replica servers agree on the last value of* $\texttt{lastReq}[]$ *in v_i.*

On receipt of such a retransmission request, client $c \in C_k$ rebinds to server s_k and transmits a "replay" message containing a list of $\langle \texttt{ReqID}, \texttt{lastReq}[] \rangle$ pairs, one for each of the requests to be retransmitted to the group (Client::receive():7-11). Following this message, the Client-Access stub of c retransmits all the required requests, each piggy-backed with its associated buffered $\texttt{lastReq}[]$ vector (Client::receive():12-15) indicating its causal context.

Server s_k waits for the "replay" message with the $\langle \texttt{ReqID}, \texttt{lastReq}[] \rangle$ list from every client of C_k. If no such message is received within a timeout period from some $c \in C_k$, then the retransmission message of s_k is sent again to the client. If a client does not respond after a number of attempts, it is removed from the client list of s_k. In this case, if the client is still alive and uses the service, it has to explicitly rebind to the service following an application specific procedure; its state has to be initialized as far as this service is concerned.

In order to avoid clients communicating with the service after such unsuccessful reallocations, we introduce the use of *epoch* numbers in the server group. Every time a new view is installed indicating the failure of one or more servers, the epoch of the group is increased and the service clients are informed about it as soon as the Reconfiguration Phase starts. Client stubs attach the latest epoch number to the requests they transmit (or retransmit) to the service.

Service state reconstruction. When the Replication Protocol of server s_k receives the expected "replay" messages, it updates the local client list $\texttt{ClientRef}_{s_k}$ with the clients of C_k that responded. s_k multicasts the updated $\texttt{ClientRef}_{s_k}$ to the group through GCP (Server::Reconfiguration-Phase():11-12). In this way, the RP of all correct replicas update their local hints for the total client set. Moreover, they use the updated $\texttt{ClientRef}$ lists to remove "old" clients from the local $\texttt{lastReq}[]$ vector.

The server waits, then, to receive all the requests referenced in the corresponding linebreak $\langle \texttt{ReqID}\}, \texttt{lastReq}[] \rangle$ list, from each client. If some of these messages are delayed, their retransmission is requested repeatedly, in a similar fashion as with the "replay" messages (the client is discarded from $\texttt{ClientRef}_{s_k}$ after a number of unsuccessful attempts).

When a correct server receives a *retransmitted* client request r, it checks the piggy-backed $\texttt{lastReq}[]$ vector. If there are causal predecessors of r (whether from the same or other clients) that have not been delivered yet, then r is buffered locally. Otherwise, the Replication Protocol layer multicasts r to the group through GCP. On delivery of r from GCP to the Replication Protocol, all correct servers deliver r to the application through RPC; their local $\texttt{lastReq}[]$ is updated accordingly (Server::deliver():1-2). Then, the Replication Protocol traverses the buffered retransmitted requests (Server::deliver():4-6) looking for a request r' which can be delivered in the current context (the current value of $\texttt{lastReq}[]$ indicates that all the causal predecessors of r' have been delivered). r' is multicast to the group through GCP. The buffer is traversed until no deliverable request is buffered there. The Reconfiguration Phase is not terminated while there are more expected retransmitted requests.

While reconfiguration is taking place, the Replication Protocol blocks any normal (not retransmissions) requests received from the clients. These requests are buffered

4. We refer here to an abstract notion of real time as understood by some hypothetical omniscient observer of the asynchronous system.

(Server::receive()-:9) and they are multicast to the group in the usual way as soon as Reconfiguration is completed.

In the case of server addition in the group, the Replication Protocol of the new server multicasts a special message to the group trying to construct its own "hint" of the membership of the client set. As soon as it receives a reply from every server in the current view, it applies the actual binding policy to decide on clients that must be rebound to itself. It constructs the local `ClientRef` list and multicast it to the rest of the group. The clients that have to be rebound are contacted following a repetitive method as with request retransmission.

5.1.2 Correctness Arguments

LEMMA 5.4. *If the procedure Server::Reconfiguration-Phase() is called in some correct replica server s, then Server::Terminate-Reconfiguration-Phase() is eventually called.*

In other words, the Reconfiguration Phase lasts for finite time. Informally, this Lemma is based on the fact that each replica server waits for "replay" messages from finite clients. Also, it waits for a finite number of request retransmissions from each of these clients. Heuristic timeout values are used to remove from the client list clients that do not respond timely. Thus, no replica server waits infinitely for the response of an allocated client. There is no other point in the reconfiguration algorithm at which a correct replica may be blocked. Thus, the reconfiguration algorithm eventually terminates.

THEOREM 5.1. *The protocol of Figs. 9, 10, and 11 satisfies the Delivery Validity property.*

PROOF. Let client *c*, which invokes request *r* to the service. From the properties of the RPC client end-point (retransmits the request while a reply is not received back) and the nonzero probability for a message to be transmitted correctly through the communication network, we are guaranteed that eventually some "representative" server *s receives r* (even if rebinding has to take place), or there is no correct server in the group.

In normal operation, *r* is directly multicast by the Replication Protocol (RP) of *s* to the group, through GCP (Server::receive():3). By GCP Validity, *r* is delivered back to the RP of *s*, which passes it directly to RPC. If this is the first time *r* is delivered to the RPC of *s*, *r* is passed to the application, otherwise it has already been delivered to the application. In any case, we are done.

There are two cases for *r* to be discarded or blocked in the Replication Protocol layer of *s*. 1) *r* carries an old epoch number and is discarded (Server::receive():1). *c* eventually receives the correct (up-to-date) epoch number by some representative server that *c* is (re)allocated to. Eventually, *c*'s RPC will initiate a retransmission, during which *r* will be attached the correct epoch number. 2) *r* is received while reconfiguration is taking place and is blocked in *receivedRequests* (Server::receive():9). From Lemma Lemma 5.4, the Reconfiguration Phase terminates in finite time and blocked requests are eventually processed as in normal operation (Server::Terminate-Reconfiguration-Phase():1-2).

It is clear, from the above, that Validity is guaranteed even in the presence of communication and/or replica failures. □

Replication Protocol of server s_k — Reconfiguration algorithm:

```
procedure Server::Reconfiguration-Phase(v):
1   for all s ∈ view - v do // removed servers
        C_k := C_k ∪ {c : c ∈ ClientRef[s] and c rebinds to s_k};
    for all c ∈ C_k do
        do transmit( retransmit req. after lastReq[c], Epoch );
5       until ReplayList_c received or c removed from C_k;
    for all c ∈ C_k do
        wait for all requests referenced in ReplayList_c;
        // see deliver() procedure about how retransmitted
        // requests are handled during Reconfiguration Phase
10      until requests received or c removed from C_k;
    ClientRef[s_k] := ClientRef[s_k] ∪ C_k;
    G-mcast(ClientRef[s_k]);
end; // procedure Reconfiguration-Phase(v)

procedure Server::Terminate-Reconfiguration-Phase():
1   for all m ∈ receivedRequests do
        G-mcast(m);
    reconfiguration-phase := 0;
end; // procedure Terminate-Reconfiguration-Phase()

procedure Server::timeout(reconfiguration):
1   for all m ∈ retransmittedRequests do
        remove m;
    Terminate-Reconfiguration-Phase();
end; // procedure timeout(reconfiguration)
```

Fig. 11. Protocol 2: reconfiguration algorithm.

THEOREM 5.2. *The protocol of Figs. 9, 10, and 11 satisfies the Delivery Integrity property.*

Informally, the Uniform Integrity of GCP and the properties of the server RPC end-point can be used to show that if replica *s* (whether correct or not) delivers request *r*, then it does so only once and only if some client *c* has previously invoked *r*.

THEOREM 5.3. *The protocol of Figs. 9 and 10 satisfies the Delivery Agreement property.*

PROOF. Let *p* and *q* be two correct replica servers. We must show that if *p* delivers request *r* then *q* also delivers *r*. Since, *p* delivers *r*, then *r* has been delivered from GCP to the Replication Protocol (RP) in *p* (Server::deliver()). If no group reconfiguration occurs (but in the presence of potential communication failures), GCP Agreement guarantees that *r* is eventually delivered from GCP to RP in correct replica *q*. Requests are directly passed to the RPC (Server::deliver():2), and we are done.

In the presence of group reconfiguration (including replica failures), GCP guarantees Virtual Synchrony. That is, *r* is delivered from GCP to RP in the same view in both replicas *p* and *q*, or it is not delivered at all. Thus, both replicas deliver *r* from RP to RPC in the same view, or they do not deliver it at all. □

THEOREM 5.4. *The protocol of Figs. 9, 10, and 11 satisfies the Delivery Order property.*

Employing arguments similar to those in the previous theorem and using the GCP Uniform Total Order property, we can easily show that the proposed protocol guarantees Uniform Delivery Order, even in the presence of communication and/or process failures.

THEOREM 5.5. *The protocol of Figs. 9, 10, and 11 and satisfies the Uniform Output Agreement property.*

PROOF. Let client c, which invokes request r to the group and delivers back reply r' for r. Let p be the replica server, which is the "representative" of c in the group and has transmitted the reply back to c. Let v_i be the view in which p delivers request r (from GCP to the Replication Protocol and from there to the application—assume this is the first time r is delivered to the RPC of p).

If p is correct, then it is straightforward to show that GCP Validity and Agreement guarantee Output Agreement among replica servers, even in the presence of communication failures. Similarly, in the case where new replicas join the group, replicas voluntarily leave the group, or replicas other than p fail, GCP Virtual Synchrony guarantees that Output Agreement is achieved among replicas that share view v_i with p.

Assume, now, that p is not correct. That is, p is not included in v_{i+1}. There are two cases to consider.

1) Despite p's failure, all correct replicas deliver r in v_i. Then, we are done.
2) No correct replica delivers r in v_i. In this case, r, the message originally multicast by p, is either delivered in some view v_{i+k} ($k \geq 1$) or it is not delivered at all.

Because of p's failure, $v_i - v_{i+1} \neq \emptyset$. Thus, the installation of v_{i+1} initiates the Reconfiguration Phase procedure (Server::install-view():7) in the Replication Protocol (RP) layer of every correct replica. Let q be the replica which decides to become the new representative of client c. That is, $c \in C_q$. The RP layer of q transmits a "retransmission request" to c together with $\texttt{lastReq}_q[c]$ and the new **Epoch** value (Server::Reconfiguration-Phase():4). According to Lemma 5.3, the transmitted value of $\texttt{lastReq}[c]$ is the same irrespectively of which is the new representative of c. By the nonzero probability for correct message transmission over the communication network, it is guaranteed that this message is eventually received by the Client-Access (CA) stub of c (possibly after a number of retransmissions) and the corresponding "replay" list is eventually received by the RP layer of q. Also, when reply r' was transmitted, request r was not stable in the group (by assumption). By Lemma 5.2, we are guaranteed that r is buffered in the CA stub of c. $\langle r.\texttt{id}, r.\texttt{lastReq}[]\rangle$ is then included in the "replay" vector transmitted by c to q; c also retransmits r, which is eventually received by q.

According to the state reconstruction algorithm (Server::deliver():4-6), received request r is eventually multicast to the group through GCP, by q. GCP Validity and Agreement guarantee that r is eventually delivered to RP and then (directly) to the application (Server::deliver():9) in all correct replicas, despite communication failures and/or group reconfiguration. If replica q fails during the Reconfiguration Phase, the reconfiguration procedure is initiated in some other replica. Eventually, if there is a correct replica, r is delivered by all correct replicas. □

The reader may have noticed, that r may be delivered more than once (twice) from GCP to RP, in the correct replicas. In the case that p fails causing the installation of view v_{i+1}, GCP may deliver message r, originally multicast by p, in some future view v_{i+m} ($m \geq 1$). However, request r is also multicast through GCP by the new representative of c, q in our case, during the Reconfiguration Phase and it is delivered in view v_{i+n} ($n \geq 1$). RP does not detect duplicate requests, but RPC does so. Thus, the second time r is delivered to RPC, it is discarded there.

THEOREM 5.6. *The protocol of Figs. 9, 10, and 11 satisfies the Uniform Output Order property.*

PROOF. Assume that replica p delivers requests r_i from c_i and r_j from c_j, r_i first, in views v_i and v_j, respectively, ($i \leq j$). Also assume that c_j receives a reply, say r_j', from p for r_j. If p is correct, then it is easy to show that GCP Uniform Total Order guarantees Output Order. We focus on the case where p is faulty and at least r_j is not delivered by the correct replicas.

1) $i < j$.
2) $i = j$ and r_i is delivered by the correct replicas.

In both cases, it is easy to show that Uniform Output Order is guaranteed by means of the Uniform total Order and the Virtual Synchronous behavior of GCP (in the lines of the relevant proof in the previous section).

3) $i = j$ and r_i is not delivered by any correct replica. There are two subcases to consider:

3.1) c_i is correct and retransmits r_i to q (may be a new or the old representative of c_i). c_j also retransmits r_j to q', its new representative in the group (c_j is correct by assumption, otherwise uniformity is not required). The retransmitted r_j carries its associated vector $\texttt{lastReq}$ []. By assumption, $r_j.\texttt{lastReq}[c_i] \geq r_i.\texttt{id}$. (1)

When the RP layer of some server s delivers r_j to RPC, then $r_j.\texttt{lastReq}\ [] \leq \texttt{lastReq}_s[]$,[5] which implies $r_j.\texttt{lastReq}\ [c_i] \leq \texttt{lastReq}_s[c_i]$. (2)

From (1) and (2), we have that on delivery of r_j from RP to RPC in server s the following holds: $r_i.\texttt{id} \leq \texttt{lastReq}_s[c_i]$. However, $\texttt{lastReq}_s[c_i]$ increases by steps of 1 and becomes equal to $r_i.\texttt{id}$ only when r_i is delivered to RPC. Thus, when r_j is delivered to RPC, r_i has already been delivered to RPC.

During the Reconfiguration Phase, it is possible for a correct replica to deliver from GCP either r_i or r_j, originally multicast in v_i (the original requests, not the retransmissions). These messages are processed in the normal way and as a result r_i and r_j may be delivered to RPC more than once. RPC detects and handles duplicate requests.

3.2) c_i is not correct, does not transmit requests and does not respond to retransmission requests of

5. $\leq$: comparison operator among vectors.

its representative, if so required. Thus, r_i may be not received eventually by any correct replica. On the other hand, c_j retransmits r_j to q', its new representative in the group. The retransmitted r_j carries its associated vector `lastReq`[]. By assumption, r_j.`lastReq`$[c_i] \geq r_i$.`id`. Thus, r_j cannot be removed from *retransmittedRequests* and multicast to the group, unless r_i is already delivered. Eventually, a timeout occurs, Reconfiguration is terminated (Server::timeout(reconfiguration)) and all undeliverable buffered retransmissions, including r_j, are discarded. The RPC of c_j timeouts without any response from the service and forces c_j to explicitly rebind to the service according to application semantics. Thus, Uniformity is not violated. □

It has been shown that the proposed protocol (Protocol 2) is robust against benign communication and process failures, by exploiting a fault tolerant Group Communication Protocol that provides Virtual Synchrony.

5.2 Protocol 3: Replication Transparent to the Clients

In earlier work [15], [17], the authors have proposed a Replication protocol for the case where clients are not aware of replication. The protocol is outlined in the following paragraphs (with some improvements) and its correctness is formally proven. We identify the cost of satisfying Uniformity without a replication specific stub on the client. The structure of clients and servers is depicted in Fig. 12.

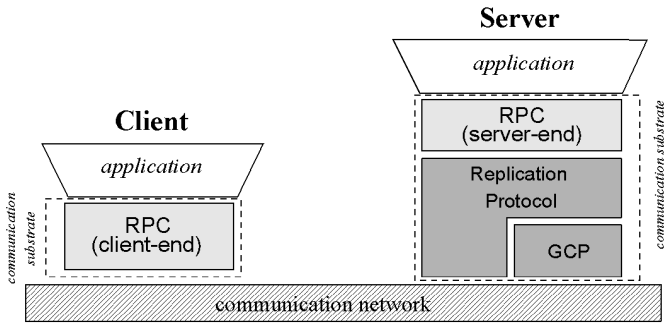


Fig. 12. Protocol 3: structure of clients and replica servers.

According to this protocol, client requests are multicast to the whole server group. A server is decided, on a per message basis, to diffuse a synchronization message through GCP for the coordinated delivery of the request. The protocol follows a conservative output mechanism, in the sense that replies are not transmitted unless requests are stable in the group. Fig. 13 illustrates the protocol operation, in the presence of communication failures but without changes to the group membership (the 'deliver' event in the diagram refers to request delivery from the Replication Protocol layer to the RPC end-point).

5.2.1 Protocol Description

Binding. The client's RPC end-point binds to the service reference. The type of the reference is transparent to the end-point—it can be either a unicast or a multicast reference.

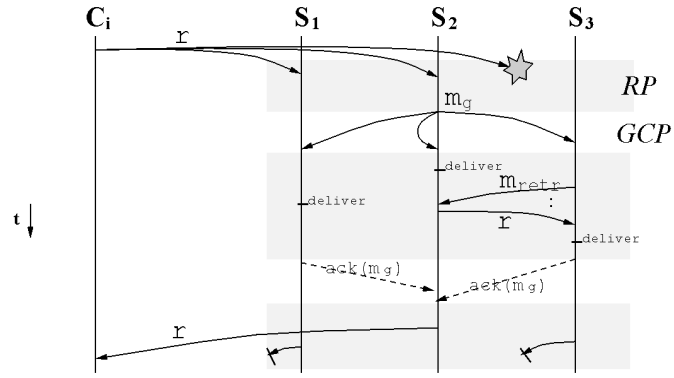


Fig. 13. Protocol 3: message diagram (no group reconfiguration).

Request delivery. A request r is sent to the service reference, which is, in this case, a multicast network address. Request r is received by the Replication Protocol (RP) layer of replica servers that share the service reference. According to a deterministic function on the group membership set and the request id, the RP layer of a single server s in the group decides to take the responsibility to synchronize the delivery of r in the group (Server::receive():3 in Fig. 14). In particular, s generates a special synchronization message m_g, which references the unique identity of r; m_g is multicast to the group through the GCP layer (Server::receive():4-5).

The delivery of m_g from GCP to RP in a server (including s itself) indicates the logical time at which r must be delivered to the application, through the RPC end-point. If the RP layer of some server s' has already received r on delivery of m_g, then r is immediately passed to the RPC end-point (Server::deliver():11). Otherwise, RP detects the existence of request r, which is either lost or delayed in the communication network. In that case, the RP of s' requests r from the group (Server::deliver():1-8). At least one replica, server s, has received r, since a relevant synchronization message has been multicast to the group. If no response is received after a number of attempts, the request is assumed lost by all correct replicas (s may have failed in the meanwhile) and m_g is discarded. Reliable client-access communication is implemented by the RPC end-points. Thus, in the latter case, r may be eventually retransmitted by the RPC end-point of the originator client.

Due to the unreliable nature of the communication network, a server may "miss" a request for which it would be responsible to send a synchronization message in the group. Therefore, if a server's RP receives a request (for which it is not responsible) and does not receive a corresponding synchronization message for a timeout period, then it remulticasts (unreliable, network multicast) the request to the group (Server::timeout():2-3). Duplicate requests are not detected by RP and they are handled as normal requests—their delivery is synchronized in the group and they are passed to the RPC end-point, where duplicates are handled as described in Section 3.1. A performance optimisation could be achieved, if RP detected duplicate requests. In that case, duplicates would be passed directly to RPC without any synchronization in the group.

```
Server Replication Protocol (server s_i):
Initially:
    reply-flag = 0 // s_i responsible for reply to last request
procedure Server::receive(m): // from comm. network
1   if m.type = client request then
        buffer m in receivedRequests;
        if DECISION(view, m.id) = s_i then
            m_g := { type := client request, ref := m.id };
5           G-mcast(m_g);
    else if m.type = retransmission request then
        if ∃m_r ∈ receivedRequests : m_r.id = m.ref then
            transmit(serv-ref, m_r); // unreliable multicast
end; // procedure receive(m)

procedure Server::send(m): // from RPC (send reply m)
1   if reply-flag then
        if m': last delivered request and
                m_g.id with m' in receivedRequests then
            wait for G-stable(m_g.id);
        transmit(m); // reply to client
end; // procedure send(m)

procedure Server::deliver(m_g): // from GCP
1   if ∄m ∈ receivedRequests : m.id = m_g.ref then
        m_rr := {type := retransm. request, ref := m_g.ref };
        do transmit(serv-ref, m_rr); // unreliable multicast
        until m : m.id = m_g.ref received or timeout;
5       if timeout then
            discard m_g;
            return;
        else buffer m in receivedRequests;
    store m_g.id with m in receivedRequests;
10  reply-flag := (DECISION(view, m.id) = s_i);
    deliver(m); // to RPC
    G-ack(m.id); // ack. delivery of synch. message for stability
end; // procedure deliver(m)

procedure Server::install-view(v): // from GCP
1   view := v;
    for all m ∈ receivedRequests without associated m_g.id do
        if DECISION(view, m.id) then
            m_g := { type := client request, ref := m.id };
5           G-mcast(m_g);
end; // procedure install-view(v)

procedure Server::timeout(): // called periodically
1   for all m ∈ receivedRequests do
        if m not delivered and buffered longer than T_buf then
            transmit(serv-ref, m); // unreliable multicast
        else if m has associated m_g.id and G-stable(m_g.id) then
5           remove m from receivedRequests;
end; // procedure timeout()
```

Fig. 14. Protocol 3: server replication protocol (pseudocode).

Since there are cases when client requests have to be retransmitted to the group, received request messages are buffered in the RP layer. A request r is garbage-collected, when it is known that r has been *received* by every server (RP) in the group. This information is determined according to the stability of the corresponding synchronization message in GCP. Message m_g is stable in the group, if its delivery (from GCP to RP) has been explicitly acknowledged by every member (see Section 3.3). The RP layer of a server acknowledges the delivery of m_g only after the corresponding r is received and delivered locally (Server::deliver():12). RP periodically traverses the buffer of received / delivered client requests and garbage-collects the stable requests (Server::timeout():4-5).

Reply transmission. The processing of a client request, in the application layer of a server, results in the transmission of a reply back to the client. RP filters the replies produced by the application through the RPC end-point, and only one replica actually transmits the reply for a specific request. This replica is again decided in a distributed manner in the group and it is not necessarily the same as the replica which transmitted the synchronization messages for the corresponding request. The decision is made on the delivery of the request (Server::deliver():10).

In order to cope with group reconfiguration (see correctness arguments), a conservative policy is adopted for the transmission of replies to the clients. In particular, the Replication Protocol implements the following output policy:

Safe output. Replica server s, which decides to transmit the reply r' for request r, does not transmit r' until r has been received by every correct replica in the group.

Safe Output is implemented using information about stability of synchronization messages in GCP. That is, the reply to request r can be transmitted to the client as soon as the corresponding synchronization message m_g has become stable in GCP (Server::send():2-4).

The potential blocking of the reply transmission is the price to be paid for the lack of a replication related stub on the client. The performance implications of the Safe Output policy are discussed in Section 6.

Group reconfiguration. In case of group reconfiguration, the Virtual Synchronous behavior of GCP guarantees that correct replica servers deliver the same set of client requests in each view. It is important to notice, that, in the presence of group reconfiguration, more than one synchronization messages may be delivered from GCP for the same client request. For example, let s be a server, which receives request r and multicasts a corresponding m_g to the group, in view v_i. Another server s' receives r in another view v_{i+k} ($k \geq 1$), while m_g has not been delivered from GCP yet (this is possible, since requests arrive asynchronously to internal group activity). Server s' takes responsibility for the synchronization of r in the new view and multicasts another synchronization message m'_g in v_{i+k}. The delivery of messages m_g and m'_g from GCP may result in multiple deliveries of r to RPC. However, duplicate requests are handled in RPC and they are not a concern of the design of the Replication Protocol.

On installation of a new view, each replica's RP layer traverses the unstable buffered requests and reevaluates responsibility for each of them, in the new membership set (Server::install-view():2-5). If responsibility is decided for a buffered request r, a synchronization message is multicast through GCP for r, following the usual synchronization procedure. This part of the algorithm is not required for the correctness of the protocol; it is a performance optimisation for earlier delivery of buffered requests.

The interested reader is referred to [17] for a more detailed description of the protocol, augmented with a replication management mechanism.

5.2.2 Correctness Arguments

THEOREM 5.7. *The protocol of Fig. 14 satisfies Delivery Validity and Uniform Delivery Integrity.*

Speaking informally, the properties of the RPC end-points (retransmission of requests, if no reply is received and retransmission of replies on receipt of duplicate requests) together with GCP Validity and Uniform Integrity can be easily used to show these two properties for the proposed design.

Also, GCP Agreement and Uniform Total Order are directly exploited to guarantee Agreement and Uniform Order for request delivery. The interested reader is referred to [14] for detailed proofs.

THEOREM 5.8. *The protocol of Fig. 14 satisfies Delivery Agreement and Uniform Delivery Order.*

We focus, here, on the correctness of the protocol regarding Uniform Output, especially in the presence of process failures. It will become clear from the following arguments, that the Safe Output policy caters for straightforward solutions to the problem of Uniform Output.

THEOREM 5.9. *The protocol of Fig. 14 satisfies Uniform Output Agreement.*

PROOF. Let replica p, which transmits a reply r' to request r (Server::send():4), in view v_i. This can only happen, if synchronization message m_r, which corresponds to r (the delivery of m_r from GCP triggers the delivery of r from RP to RPC), is stable in GCP (Server::send():2). That is, all correct replicas in v_i have acknowledged the delivery of m_r from GCP. RP acknowledges the delivery of a synchronization message only after the referenced request has been delivered to RPC (Server::deliver():12). Thus, stability of m_r implies that r has been delivered by all correct replicas in v_i. Therefore, a correct replica q delivers r even if p fails after the transmission of r' ($p \notin v_{i+1}$). □

THEOREM 5.10. *The protocol of Fig. 14 satisfies Uniform Output Order.*

PROOF. Let replica p, which delivers requests r_a and r_b, r_a first, and transmits a reply r_b' to r_b, in view v_i. The order of delivery of the requests in p implies the order of delivery of the corresponding synchronization messages. In particular, if m_a and m_b are the synchronization messages for r_a and r_b , respectively, then m_a is delivered before m_b from GCP to RP, in replica p.

Since p transmits reply r_b', it is implied that m_b is stable in the group (Server::send():3). Thus, m_b has been delivered from GCP to RP, in all correct replicas. (1)

From the Uniform Total Order property of GCP, we are guaranteed that no replica RP delivers m_b from GCP unless it has already delivered m_a, even if p fails and a new view v_{i+1} is installed. (2)

From (1) and (2), when p transmits r_b', all correct replicas have delivered both m_a and m_b, m_a first, and have acknowledged both m_a and m_b. The latter means that the corresponding requests r_a and r_b have been both delivered from RP to RPC, r_a first, in all correct replicas. □

In conclusion, Protocol 3 satisfies all the requirements of the State Machine approach and it is robust against benign communication and process failures.

5.3 Client-Access vs. Replication Protocol

In the last two sections, Replication and Client-Access protocols have been presented together as an integral module in the server communication substrate. This served to keep the description simple by avoiding details of communication substrate sublayering. However, the two protocols have different roles in the replication layer of a server, as discussed in Section 3.2.

- The Client-Access protocol implements the actual access method of the clients to the service. It also handles reply transmission by applying the *Output Policy* [17] of the service. The Output Policy determines how many replies are transmitted for a request and which server(s) transmit those replies.
- The Replication protocol synchronizes request delivery among the replica servers by applying the *Synchronization Policy* of the service [8]. The Synchronization Policy determines how closely the states of the replica servers are synchronized. Active replication, and passive replication (primary-cohorts) are examples of synchronization policies.

The concerns of the Client-Access protocol are largely orthogonal to those of the Replication protocol. In general, the access method employed by the clients is independent of the synchronization policy applied within the server group. For example, the access method of Protocol 3 can be combined with any synchronization policy in the group. If a pure active replication policy is applied (as in Section 5.2), then the onus of generating and multicasting synchronization messages is evenly distributed among all the servers in the group. With a passive replication policy, a specific server in the group (the primary) takes always the onus for synchronization. In the latter case, techniques such as message batching and background diffusion are typically employed to update the state of the cohorts with some delay.

In a similar fashion, the Client-Access method of Protocol 2 is independent of the synchronization policy applied in the group, with the only difference being that the binding policy must be compatible with the synchronization policy. If passive replication is applied, then all the clients must bind to the primary server. If pure active replication is applied (as in Section 5.1), then a client may be bound to any server in the replica group.

In the case of Protocol 1, the Client-Access mechanism is closely integrated with an active Replication protocol. The latter is imposed by the properties of the Group Communication Protocol, which is directly used for the diffusion of requests and replies in the group. Although theoretically separate from the synchronization policy, the Client-Access protocols of the “closed” model imply active replication. Table 2 summarizes the possible combinations of Client–Access protocols and synchronization policies in the

cess protocols and synchronization policies in the replicated server group.

TABLE 2
CLIENT-ACCESS PROTOCOLS VS. SYNCHRONIZATION POLICIES

Synchronisation Policy	Client Access Protocol		
	Protocol 1	Protocol 2	Protocol 3
Active	Yes	Yes (with binding)	Yes
Passive	No	Yes (with binding)	Yes
Combinations	No	Yes (with binding)	Yes

6 PROTOCOL IMPLEMENTATION AND EVALUATION

6.1 Implementation in Regis

The protocols presented in this paper have been implemented and evaluated in the *Regis* distributed platform. Regis [20] is a programming environment aimed at supporting the development and execution of parallel and distributed programs. It embodies a constructive approach to the development of programs based on separating program structure from computation and communication.

The latest version of the system [24] incorporates a flexible communication subsystem, which facilitates the use of different protocols according to the needs of the application (style of interaction, QoS requirements) and the system model (transport layer). The system offers a range of built-in primitives, but also provides programmers with a framework in which to develop their own models of interaction. Communication protocols are implemented in the form of protocol stacks composed of light-weight micro-protocol modules. The latter characteristic of the system has facilitated experimentation with different implementations of the various protocol modules proposed in this paper as well as reusability of existing nonreplicated primitives (RPC end-points).

6.2 Performance Results

In the following paragraphs, we study the performance of the current implementation of Protocols 1–3, in Regis. The experiments have been contacted on a network of SUN SPARC IPX workstations interconnected by a loaded Ethernet consisting of multiple segments. The OS kernel has been augmented to support Deering's IP extensions for multicast [10], which are directly mapped on Ethernet's hardware multicast. IP multicast has been used for an efficient implementation of GCP [14] and, in the case of Protocol 3, for the transmission of client requests.

Fig. 15 presents the response latency for the two protocols of the "open" model.

Response latency. the time elapsed, in a client, between invoking a request to the service and receiving back a reply.

For these measurements, the number of clients is double that of the servers; requests and replies are of the same size. The times reported here are average times calculated for static server groups. That is, no group reconfiguration (including server failures) is taking place. No delay is introduced by the application layer of the servers. The latency time consists of two parts:

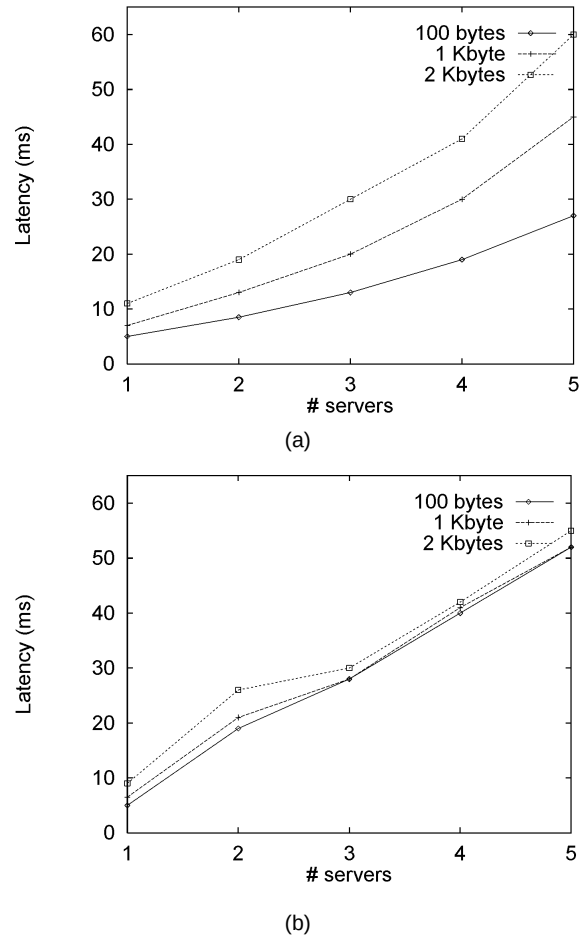


Fig. 15. Latency results for the "open" group model. (a) Protocol 2; (b) Protocol 3.

1) the time for client-service-client interaction: the total time to transmit the request from the client to the service and the reply from the service to the client (e.g., an average of 2.5 ms for requests / replies of 100 bytes);
2) the time for internal server synchronization: to deliver the request or the synchronization message through GCP, in the case of Protocols 2 and 3, respectively.

The results indicate that Protocol 2 provides, in general, better response times than Protocol 3 justifying earlier comments: no reply blocking is required in Protocol 2 to guarantee Uniformity. However, the difference becomes smaller for large messages (even reversed for large groups). The reason is that Protocol 3 uses small internal synchronization messages, which are independent from the request size. Thus, the size of the request / reply messages affects only part 1 of the latency time, which is, in general, a small fraction of the total latency. On the other hand, Protocol 2 multicasts the actual request messages through GCP. Thus, part 2 of the latency time is proportional to the size of the request message. Since the time required for internal group synchronization is the main part of the latency time, Protocol 2 does not scale well for large request messages and large

server groups. Moreover, the results for Protocol 2 have been obtained for an even distribution of clients to servers, creating a favorable environment for this protocol.

Fig. 16 depicts throughput results for messages of 100 bytes and 1 Kbyte.

Throughput. the total number of requests processed per second by the server group.

Due to the synchronous style of communication of the clients, the throughput is inversely proportional to the latency times, and Protocols 2 and 3 exhibit similar comparative performance as discussed above. Both protocols provide better throughput for larger sets of clients (more clients invoking concurrent requests) and smaller server groups. They scale well for large sets of clients. The best results are recorded for the trivial case of one-server group, where no internal synchronization is required. In the latter case, the throughput for messages of 1 Kbyte does not exhibit a peak similar to that for 100 byte messages. This is due to the available Ethernet bandwidth. Indeed, a throughput of 450 requests/sec implies 900 messages/sec (requests and replies), or 900 Kbytes/sec on the network, a value close to the 1,007 Kbytes/sec maximum Ethernet bandwidth.

Fig. 17 depicts the latency and throughput results measured for Protocol 1. The results are plotted against the number of servers in the group. The total size of the group, including clients and servers, is shown in parentheses. Note that the number of clients is always double that of servers, as it was also the case in Fig. 15. The performance of Protocol 1 is clearly poor, despite the fact we have used the optimised version where the reply is unicast to the client. The large size of the group is reflected on higher delivery times for requests. The results would be even worse, if we considered frequent membership changes in an environment of dynamic, short-lived clients. The bad performance of the "closed" group model has not been surprising, given that it results in the formation of large process groups. However, this is the first time—to the best knowledge of the authors—that such a comparative evaluation is performed, giving a precise estimation of the performance difference between the two models.

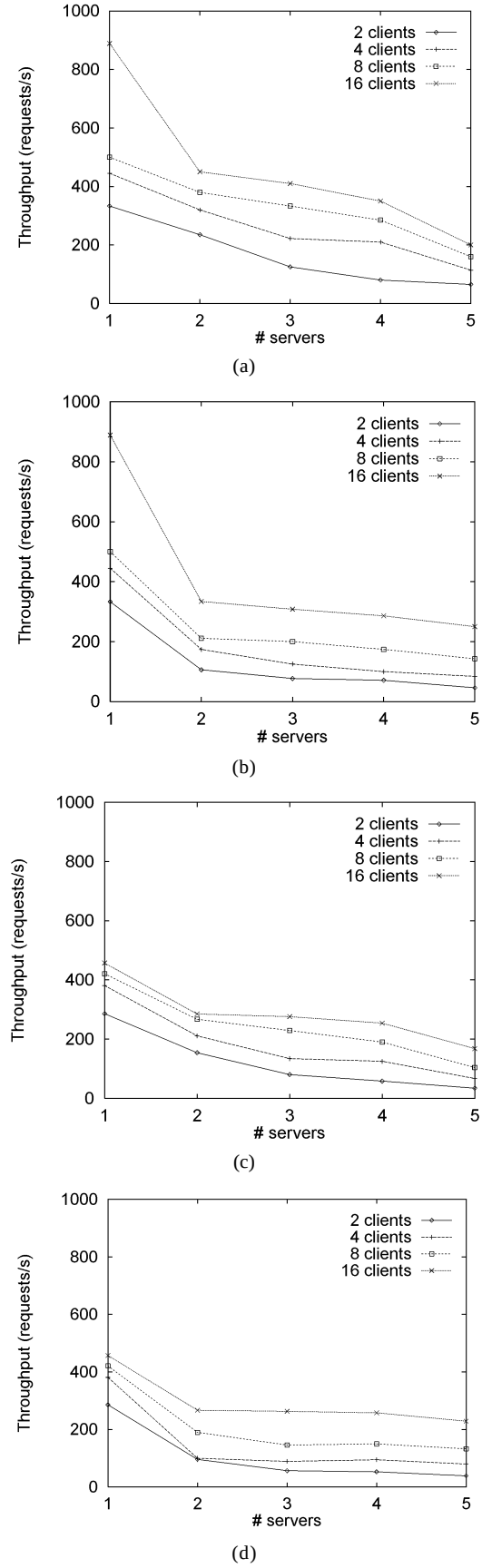


Fig. 16. Throughput results for the "open" group model. (a) Protocol 2: requests/replies: 100 bytes; (b) Protocol 3: requests/replies: 100 bytes; (c) Protocol 2: requests/replies: 1 Kbyte; (d) Protocol 3: requests/replies: 1 Kbyte.

7 Final Remarks

The paper addresses the problem of client-service interaction in the case of replicated service provision. This problem has not been addressed adequately in the literature, especially in the case of large, open distributed systems. However, it is fundamental for the provision of highly available distributed services.

7.1 Related Work

Manetho [11] is a research system which has addressed the problem of interaction between a group of replicated servers and other entities in the system. Output delay is avoided during normal operation, by piggy-backing group output with information about the service history. This information is diffused in the system according to the causal dependencies of messages. In the event of primary's failure (a primary-cohorts replication model is followed), the

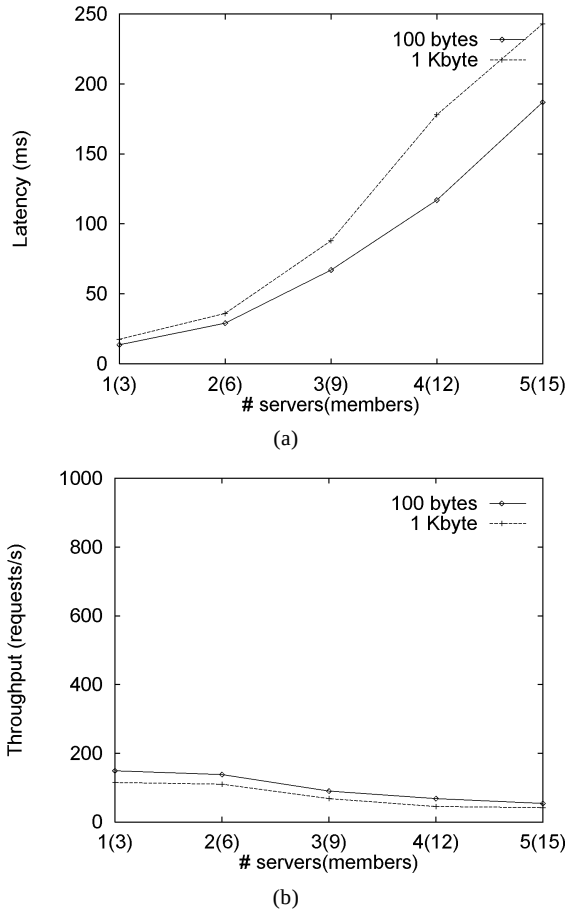


Fig. 17. Performance results for the "closed" group model. (a) latency; (b) throughput.

whole system is contacted by the surviving servers, to reconstruct any lost part of the service state in a way consistent with the rest of the system. This method also works for proactive service provision, or in the presence of other internally synchronized, nondeterministic events in the group. The obvious disadvantage of the method is that the effects of replication are exposed to the entire system. In our protocols, replication concerns are kept local: just in the server group in Protocol 3, and among servers and service clients in Protocol 2.

An earlier attempt to propose a Client-Access protocol that is independent from the actual Replication mechanism has been made in the GRIP protocol [25]. GRIP has focused on the specific case of the "open" model, where clients accommodate replication related stubs. However, the functionality of the Client-Access protocol is not clearly separated from that of the Replication protocol, especially in the case where "at-most-once" execution guarantees are required. Moreover, GRIP does not address explicitly the problems of system consistency in the case of reconfiguration of the server group; no Uniformity guarantees are provided.

7.2 Conclusions

The paper has introduced an extension of the State Machine approach. The requirements for uniform service output have been particularly stressed, since they have not been stated clearly in the existing literature.

The design methodology developed to address the problem of replicated service provision is based on the orthogonal nature of the Replication and the Client-Access protocol. The former protocol implements the binding and the output policy of the service, while the latter implements the synchronization policy applied in the server group. In this context, the paper introduces a novel architecture that keeps replication concerns local to the communication substrate. Replication is transparent to the application algorithm of both clients and servers.

This architecture has provided the framework for designing three protocols for two basic system models. The correctness of the protocols is formally argued against the requirements of the State Machine approach. Protocol 1 resembles the main characteristics of the "closed" group model, which has been proposed by some existing systems as a way to guarantee client-service state consistency in the face of system reconfiguration.

The paper focuses mainly on the "open" group model, where clients are external to the group of servers. A novel Replication and corresponding Client-Access protocol (Protocol 2) has been proposed for the case where clients can be linked to special Client-Access stubs. Uniformity is ensured by buffering requests on the client stub (together with some context information), in order to avoid expensive uniform operations in the server group.

However, it is not always possible for clients to accommodate replication related stubs. The paper outlines a novel Replication protocol (Protocol 3), introduced in earlier work of the authors, which hides replication completely from the clients. Uniformity concerns are met by blocking reply transmission until the request becomes stable in the server group.

In other words, Protocol 2 (client-aware) adopts an optimistic approach to the problem of Uniform service output, while Protocol 3 (client-transparent) adopts a more pessimistic approach. Indeed, the experimental results demonstrate that Protocol 2 performs better than Protocol 3, in the general case. The reason is that Protocol 3 delays replies during normal service provision to guarantee a property that may be violated in the (exceptional) case of server group reconfiguration. Another disadvantage of Protocol 3 is that client requests are multicast in the system, which may result in saturation of the network resources. On the other hand, Protocol 2 employs a sophisticated Reconfiguration procedure which requires substantial interaction with the service clients around the system and reconfiguration (rebinding) of the client set.

Surprisingly, though, the performance difference of the two "open" protocols is small and becomes even less significant for large messages and large server groups. This is because Protocol 2 must reliably multicast client requests between server replicas, while in the error free case Protocol 3 does so only for small synchronization messages. As it was expected, both protocols of the "open" model out-perform Protocol 1. The paper gives a practical estimation of the performance overhead of the "closed" model.

This paper has demonstrated that open group client-access protocols are clearly desirable in an environment

which supports large, dynamically changing client sets, and where clients interact with the service through synchronous communication primitives like RPC. Open group protocols would seem as the only hope of providing highly available services in the context of the Internet. The closed group approach, supported by systems such as ISIS, Horus, and Transis, is more appropriate for applications where it is important that servers maintain a consistent view of the client set. Such applications occur in trading systems for banks.

ACKNOWLEDGMENTS

The authors would like to acknowledge discussions with our colleagues in the Distributed Software Engineering group during the formulation of these ideas. We gratefully acknowledge the EPSRC (Grant Ref: GR/J87138) for their financial support.

REFERENCES

[1] Y. Amir, M. Moser, M. Melliar-Smith, D. Agarwal, and P. Ciarfella, "Fast Ordering and Membership Using a Logical Token-Passing ring," *Proc. 13th Int'l Conf. Distributed Computing Systems*, pp. 551–560. Pittsburghh, May 1993.

[2] O. Babaoglu, R. Davoli, L.A. Giachini, and M. Baker, "RELACS: A Communication Infrastructure for Constructing Reliable Applications in Large-Scale Distributed Systems," Technical Report UBLCS-94-15, Univ. of Bologna, Italy, Jan. 1995.

[3] O. Babaoglu, R. Davoli, and A. Montresor, "Failure Detectors, Group Membership and View-Synchronous Communication in Partitionable Asynchronous Systems," Technical Report UBLCS-95-18, Dept. of Computer Science, Univ. of Bologna, Bologna, Italy, Nov. 1995.

[4] K. Birman, "The Process Group Approach to Reliable Distributed Computing," *Comm. ACM*, vol. 36, no. 12, pp. 37–53, 1993.

[5] A. Birrell and B. Nelson, "Implementing Remote Procedure Calls," *ACM Trans. Computer Systems*," vol. 2, pp. 39–59, 1984.

[6] T.D. Chandra, V. Hadzilacos, and S. Toueg, "Impossibility of Group Membership in Asynchronous Systems," *Proc. 15th ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing*. ACM Press, Philadelphia, May 1996.

[7] T.D. Chandra and S. Toueg, "Unreliable Failure Detectors for Reliable Distributed Systems," *J. ACM*, vol. 43, no. 4, July 1996.

[8] F. Cristian, "Understanding Fault-Tolerant Distributed Systems," *Comm. ACM*, vol. 34, no. 2, Feb. 1991.

[9] F. Cristian and S. Mishra, "Automatic Service Availability Management in Asynchronous Systems," *Proc. Second Int'l Workshop Configurable Distributed Systems*, pp. 58–68, IEEE CS Press, Pittsburgh, Mar. 1994.

[10] S. Deering, "RFC 1112: Host Extensions for IP Multicasting," 1989.

[11] E.N. Elnozahy and W. Zwaenepoel, "Replicated Distributed Processes," *Proc. 22nd Int'l Symp. Fault-Tolerant Computing (FTCS-22)*, pp. 18–27, July 1992.

[12] P. Ezhilchelvan, R. Macedo, and S. Shrivastava, "Newtop: A Fault-Tolerant Group Communication Protocol," *Proc. 15th Int'l Conf. Distributed Computing Systems*, IEEE CS Press, Vancouver, BC, Canada, June 1995.

[13] R. Friedman and R. van Renesse, "Strong and Weak Virtual Synchrony in Horus," technical report, Dept of Computer Science, Cornell Univ., Ithaca, New York, Mar. 1995.

[14] C. Karamanolis, "Configurable Highly Available Distributed Services," PhD thesis, Dept. of Computing, Imperial College, Univ. of London, UK, July 1996.

[15] C. Karamanolis and J. Magee, "Configurable Highly Available Distributed Services," *Proc. 14th IEEE Symp. Reliable Distributed Systems*, pp. 118–127, Bad Neuenhar, Germany, IEEE CS Press, Sept. 1995.

[16] C. Karamanolis and J. Magee, "A Replication Protocol to Support Dynamically Configurable Groups of Servers," *Proc. Third Int'l Conf. Configurable Distributed Systems*, pp. 161–168, Annapolis Md., IEEE CS Press, May 1996.

[17] C. Karamanolis and J. Magee, "Providing Highly Available Services in Open Distributed Systems," *Distributed Systems Eng. J.*, vol. 5, no. 1, pp. 29–45, Mar. 1998.

[18] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat, "Providing High Availability Using Lazy Replication," *ACM Trans. Computer Systems*, vol. 10, no. 4, pp. 360–391, Nov. 1992.

[19] L. Lamport, "Time, Clocks, and The Ordering of Events in a Distributed System. *Comm. ACM*, vol. 21, no. 7, July 1978.

[20] J. Magee, N. Dulay, and J. Kramer, "A Constructive Development Environment for Parallel and Distributed Systems," *Proc. Int'l Workshop Configurable Distributed Systems*. Pittsburgh, Mar. 1994.

[21] D. Malki, Y. Amir, D. Dolev, and S. Kramer, "The Transis Approach to High Availability Cluster Communication," *Comm. ACM*, vol. 39, no. 4, Apr. 1996.

[22] S. Mishra, L.L. Peterson, and R.D. Schlichting, "Implementing Fault-Tolerant Replicated Objects Using Psync," *Proc. Eighth Symp. Reliable Distributed Systems*, pp. 42–52, IEEE, Seattle, Wash., Oct. 1989.

[23] D. Powell, ed., "Delta-4: A Generic Architecture for Dependable Distributed Computing," vol. 1 of *ESPRIT—Research Reports*. Project 818/2252, Springer-Verlag, 1991.

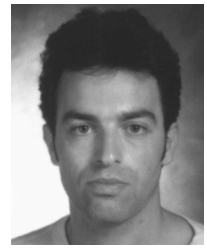
[24] N. Pryce and S. Crane, "A Uniform Approach to Configuration and Communication in Distributed Systems," *Proc. Third Int'l Conf. Configurable Distributed Systems (ICCDS'96)*, Annapolis Md., IEEE CS Press, May 1996.

[25] L. Rodrigues, E. Siegel, and P. Verissimo, "A Replication–Transparent Remote Invocation Protocol," *Proc. 13th IEEE Symp. Reliable Distributed Systems*. Dana-Point, Calif., Oct. 1994.

[26] A. Schiper and A. Sandoz, "Primary Partition "Virtually-Synchronous Communication Harder than Consenus," *Proc. Eighth Int'l Workshop Distributed Algorithms (WDAG-94)*, Lecture Notes in Computer Science 857, Terschelling, Springer-Verlag, Sept. 1994.

[27] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial," *ACM Computing Surveys*, vol. 22, no. 4, Dec. 1990.

[28] R. van Renesse, T.M. Hickey, and K.P. Birman, "Design and Performance of Horus: A Lightweight Group Communication System," technical report, Dept. of Computer Science, Cornell Univ., Ithaca, New York, 1994.



Christos T. Karamanolis received a Diploma in computer engineering from the University of Patras, Greece, and a PhD degree in distributed computing from Imperial College, London. Dr. Karamanolis is a research associate in the Department of Computing at Imperial College. His research interests include fault tolerance, replication methods, configuration and management of distributed systems.

Jeffrey N. Magee is a reader in the Department of Computing at Imperial College, London. His research interests focus on distributed and mobile systems, including techniques for describing and analyzing these systems from an architectural perspective. He is currently involved in the European Framework IV funded ARES project concerned with Software Architecture for product families and with United Kingdom funded projects investigating architecture tools, highly available distributed computing services, and location services for mobile computing systems. He is joint program chair of the Third International Software Architecture Workshop (ISAW-3) and is co-editor of the *IEE Proceedings on Software Engineering.*