

Client–Access Protocols for Replicated Services

Christos Karamanolis

Jeff Magee

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

Abstract

The paper addresses the problem of client–service interaction in the case of replicated service provision. Existing systems that follow the State Machine approach concentrate on the synchronisation of the server replicas and do not consider the problem of client interaction with the server group. Client interaction is analysed and a number of access protocols are proposed to meet a range of client requirements. 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, dynamic set of clients.

1. Introduction

With the ever increasing introduction of computing systems in many aspects of today’s life, availability of critical computing services becomes of great importance. The State Machine approach [20] 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 inter-server coordination.

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

Most systems [15, 16, 11] 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 for-

warding its requests to the other servers (this is also called an “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 on the service state despite service reconfiguration taking place concurrently to the processing of the request. Thus, some of the replication concerns move to the application algorithm of the client.

In order to address these problems, systems like ISIS [1], Horus (CLTSVR layer) [21], and Transis [14] follow the “closed” group approach: clients are members (or at least special members) of the server group. In that case, the application algorithms 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 dynamic environment. Although, this approach caters for a straight-forward solution to the problem of clients accessing dynamically reconfigurable replicated services, its performance implications are not clear in the literature.

The 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 analysed in section 2. Section 3 discusses client–access methods for different system models and replication protocols. A replication protocol that conforms to the “closed” model is outlined, with emphasis on the properties guaranteed to the clients. This protocol forms the basis for comparison with two instances of the “open” model: The first demonstrates a novel access protocol implemented by a replication-related communication stub in the client. The second is a novel replication protocol that hides replication from the clients, at the price of higher response times. The protocols are evaluated and compared us-

ing experimental performance results obtained by a first implementation in the system Regis [13] (section 4). The results show that the “closed” protocol is expensive, in terms of response times and throughput, compared to either of the two “open” protocols. Section 5 summarises the results of the paper and presents the conclusions.

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 main non-deterministic event that must be synchronised among them is the delivery of client requests. The State Machine approach puts two requirements concerning *internal service state consistency* [20]:

Agreement: All non-faulty replica servers deliver the same set of client requests.

Order: All non-faulty replica servers deliver the requests in the same relative order.

As far as the overall *system state consistency* is concerned, two more requirements must be satisfied:

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

- requests invoked by a single client must be delivered in the order of invocation;
- if request r of client c could have caused the invocation of request r' of client c' , then r must be delivered before r' .

Uniformity: If a replica server produces the output related to a client request r (e.g. reply to the client), then all correct replicas eventually deliver r .

The Causality requirement is inherited from the case of non-replicated service provision. In most cases, the clients adopt a synchronous style of communication waiting blocked (interacting neither among them or with the service) to deliver back a reply to their last request (e.g. RPC). In that way, inter-client consistency is trivially guaranteed. When clients adopt an asynchronous style of interaction with the service and inter-client consistency is of importance, then request messages must be time-stamped by means of logical or physical clocks and these times must be respected by the delivery order on the server side [12, 20].

The Uniformity requirement, which is not explicitly stated in the State Machine approach, is of importance in

systems where the membership of the replica server group changes dynamically [10]. It states that, if output is produced by the service as a result of processing request r , then the results of r persist on the state of the service. For example, consider the scenario according to 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 *agreement* requirement does not apply, since s has failed). 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 this paper, we are concerned only with transmission of replies back to clients, a special case of service output.

3. Client–Access Protocols

For the discussion of the protocols, we assume a *message-passing asynchronous system*. The communication network exhibits omission failures; messages can be delivered out of order and can be arbitrarily duplicated, but there are no spurious messages generated. The network supports an unreliable multicast capability, realised as a convenient addressing mode. The system is augmented with an unreliable failure detector [3] to circumvent the impossibility results for distributed consensus in this asynchronous environment.

We assume that clients use a service by means of a synchronous (request-reply) communication primitive, like Remote Procedure Call (RPC). Typically these primitives are built on top of an unreliable datagram communication service. Defining the exact properties of RPC 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* [2] delivery guarantees, the operation of the RPC end-points can be summarised 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 a specific 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 re-transmitted; if a request before the last of c is received, then it is discarded.

As discussed in the introduction, a main requirement for every replication protocol, not met by all existing systems, is to make replication transparent to the application layer.

The application programmers should not change the program of the client and/or server to cater for a specific replication method. We follow this approach, here, describing replication protocols that live in the communication substrate of clients and servers. In the general case, the structure of a replica server providing a highly available service and the structure of a client using that service are as depicted in figure 1.

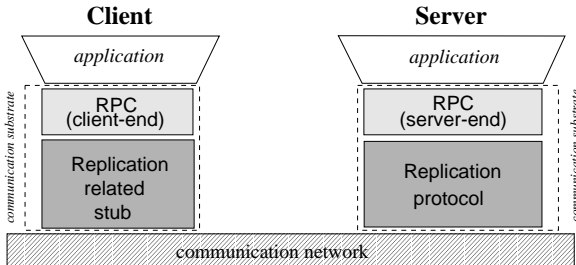


Figure 1. The structure of a replica server and a client—the general case.

- **Server:** The communication substrate is augmented with a *Replication Protocol* layer. The protocol receives and handles client requests. It synchronises request delivery amongst replica servers, in accordance with the Agreement, Order and Causality requirements of the State Machine approach. Replica output is synchronised according to an output policy for the service—single output, in the case of benign failures. The Replication protocol does not add any properties, like reliability, to the client–service communication. The same communication primitives used in the non replicated case (e.g. RPC server end-point) are also used on top of the Replication protocol.
- **Client:** The communication substrate is (in the general case) augmented with a replication related stub, which implements the client–access protocol to the replicated service. Again, the non replicated communication primitives (e.g. RPC client end-point) are re-used on top of the replication stub.

A main concern of every replication and corresponding client-access protocol is to satisfy the Uniformity requirement in the case of dynamic system reconfiguration. Reconfiguration is due to two reasons: *i*) system changes like processor failures, process crashes, system partitioning; *ii*) explicit management operation like removal and addition of servers that provide a service, instantiation or removal of clients using a service. For reasons of clarity, we adopt the primary partition model for the discussion that follows. No distinction is made between a failed and a partitioned (from the main partition) system entity.

3.1. The “closed” group model

As discussed earlier in the paper, systems like ISIS and Transis favour a model where clients form a group together with the servers that provide the replicated service. Servers maintain a consistent membership view of the service clients and vice versa. In that way, the delivery properties of group communication protocols are exploited to satisfy the requirements of the State Machine approach. Client requests and replies from the servers are multicast to the whole group—see figure 2.

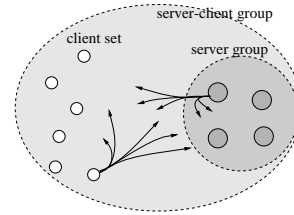


Figure 2. The “closed” group model.

Protocol 1: Clients members of the server group

We summarise here the basic elements of protocols for the “closed” model [1, 21]. We describe a protocol which is implemented in the communication substrate of clients and servers. Replication is transparent to the application—client and server algorithms are still implemented using the non replicated RPC primitives (existing systems, like ISIS, fail to meet the transparency requirement). The discussion focuses on the guarantees provided to the clients, especially in the case of group reconfiguration (Uniformity). The main objective is to compare the performance of this class of protocols with the protocols of section 3.2.

The structure of clients and servers is depicted in figure 3. *GCP* stands for Group Communication Protocol. A replication related protocol layer is placed on top of that, filtering out messages (requests or replies) which are not of interest to a specific entity (client or server).

Binding: The client replication stub (filter) initiates a join procedure to the group. The *join* primitive of *GCP* is called parameterised with the group reference.

Request delivery: A request r transmitted through the client’s RPC end-point, is broadcast through *GCP* and it is delivered to every member of the group, whether client or server. However, r is let through the Replication Filter of only server members and it is delivered to the application through the RPC end-point. The Reliability and Order properties of *GCP* directly imply the Agreement and Order requirements for inter-server synchronisation. Note that the causal order of typical group communication protocols would imply the Causality requirement, even if the communication primitives used by clients were not synchronous.

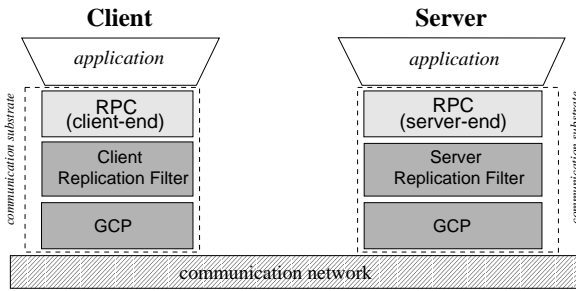


Figure 3. “Closed” group model—structure of clients and replica servers.

Service output: As a result of processing r , every server attempts to transmit a reply back to the sender of r , by calling the reply primitive of the RPC end-point. 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 reply is discarded by every member in the group (Replication Filter), except the client that transmitted r . A straight-forward optimisation is for the server to uni-cast the reply to the client. In any case, the reply is passed to the client application, which is blocked on the RPC end-point.

Group reconfiguration: Typically, *GCP* is required to exhibit a Virtually Synchronous behaviour [1, 19]. The membership protocol of *GCP* agrees on membership changes which are delivered as membership *views* to the layer above, the Replication module in this case. Virtual Synchrony guarantees that all correct group members deliver the same total order of views and also deliver the same set of messages between two consequent views. Thus, servers reach consistent decisions about responsibility for transmitting replies.

The potential server failures introduce the following problem related to the Uniformity requirement: a server s may deliver a request r from client c , in some view v_i ; then, it takes responsibility for r and transmits a reply r' back to c . Immediately after that, it fails causing a view v_{i+1} to be installed in the surviving group members. Even if c delivers the reply r' back, Virtual Synchrony does not guarantee that all correct (surviving) servers in the group deliver r in v_i . It does guarantee, however, that if c delivers back its own request r in v_i ¹, then all correct servers in the group also deliver r in v_i . Thus, if a new view is installed on client c and there is a request r of c which has been multicast through *GCP* but not delivered back yet, then the replication stub of c re-multicasts r to the group. In other words, Uniformity is satisfied by buffering requests on the client stub

¹Typically, Group Communication Protocols deliver messages back to the sender, if it is a member of the group.

for potential retransmission in a new view. Requests can be garbage collected as soon as they become stable in the group (delivered to the replication module of every group member).

3.2. The “open” group model

According to this model, the clients are external to the group of servers. That is, servers do not maintain a consistent view of the client set. The “open” model is suitable to environments where a service is used by a large and fairly dynamic set of short-lived clients, and is provided by a relatively small and stable group of long-lived servers (see figure 4). Two sub-cases of this model are distinguished, according to the requirements of the application classes:

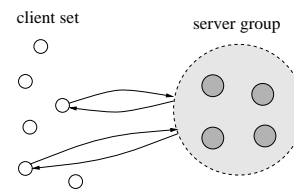


Figure 4. The “open” group model.

1. Clients are aware of replication; that is, they accommodate replication related communication stubs. This is the case in small and homogeneous environments, where clients can be linked to replication related stubs for (potentially) improved performance.
2. Replication must be completely transparent to the client [9]. This is typically the case in open distributed systems, where clients cannot necessarily be re-programmed or re-linked to cope with replication. A similar requirement stems 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.

Protocol 2: Clients aware of replication

Systems that adopt this model [15, 16, 11] focus on internal server synchronisation and consider the client-access protocol as an application level concern. In the following paragraphs, we outline the basic principles of these protocols. We propose a generic client-access protocol which is implemented by a stub in the client’s communication substrate. The stub co-operates with the Replication protocol on the server site to meet the State Machine requirements. Thus, replication is transparent to the application, even on the client side. Figure 5 illustrates the structure of clients and servers, for this protocol.

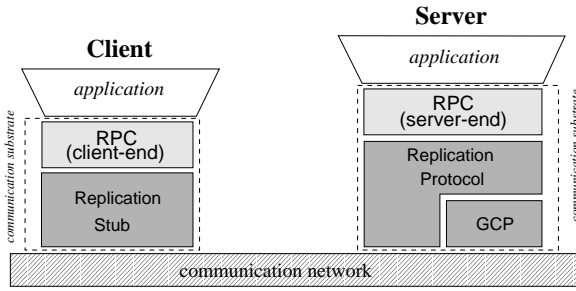


Figure 5. Protocol 2—structure of clients and replica servers.

Binding: The client replication stub resolves the multi-destination reference of a replicated service (e.g. by contacting a name service) and binds to a single replica server. This server acts as the representative of the client in the group.

Request delivery: The client transmits a request r to the single server replica s it is bound to. The request r is received by the Replication Protocol (RP) layer of s , which then broadcasts r to the group through GCP (see figure 6).

Every server in the group (including s) delivers r to the application as soon as it is delivered from GCP to RP . In this way, the reliability and order properties of GCP are exploited to satisfy the *Agreement* and *Order* requirements of the State Machine approach, respectively. Reliable client-server communication is implemented by the RPC protocol.

Service output: Following a single output policy, the RP layer of only the representative server s actually transmits the replies to the requests of c .

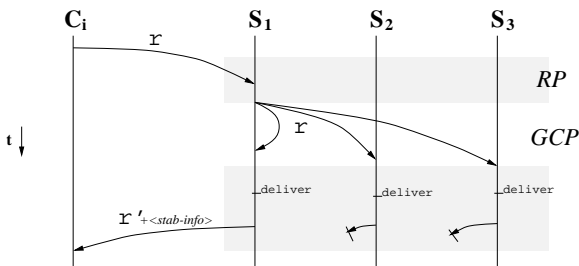


Figure 6. Protocol 2—message diagram (no group reconfiguration).

Group reconfiguration: The design of the Replication Protocol requires a Virtually Synchronous behaviour from GCP , in the case of group reconfiguration. Thus, servers can reach mutually consistent decisions about the membership of the server group and agree on the set of client requests delivered within a view.

Even then, server failures introduce the same problems of violation of the *Uniformity* requirement, as the ones

presented in section 2. For example, server s receives request r from client c , broadcasts it through GCP , delivers it back, passes it to the application, and finally sends a reply back to the client; s then fails, and because of communication failures, no other server in the group delivers r from GCP (note, that since s fails, the non uniform reliability property of GCP does not guarantee that the rest of the group delivers r). As a result, the states of the client and the service are inconsistent.

In order to achieve Uniformity, the client-access protocol stub buffers all the requests to the service, even after replies are received back for them. A request r is removed, when the client learns that r has become stable in the group. For this reason, replies must be piggy-backed with information indicating the most recent request of the client that has become stable in the group (note, that if a request r of a client is stable, then any previous request of the same client is also stable in the group).

In case of failure/removal of server s , the clients of s detect this event (e.g. after a number of unsuccessful invocations) and re-bind to another server s' —this procedure is initiated by the RPC protocol. Assume that r is the last request of client c to the service, the invocation of which resulted in detecting the removal of s (note that c has not received a reply to r).

Because of the failure of s , the rest of the group may have lost some of the broadcasts of s (or may have not delivered them by the time the view indicating the removal of s is installed). That is, surviving servers may not be aware of the last κ unstable requests of c to the service. Note, that due to the reliability and order properties of GCP and because of the synchronous style of communication of the clients, there can be no “gaps” in the sequence of the missing requests.

For this reason, as soon as c binds to s' , the client-access stub of c re-transmits all the λ locally buffered requests² to the service. The RP layer of s' handles all these requests as if they were received for the first time: it broadcasts them to the group, and they are delivered up to the application. Any duplicate deliveries (the first $\lambda - \kappa$) are discarded by the server RPC end-point. (Even in the case that the original requests broadcast by s are eventually delivered in some future view, the same case of duplicate delivery applies). Any requests (the last κ) that are indeed delivered for the first time, are processed by the application and replies are produced, which are then transmitted back to c by s' . All (duplicate) replies are discarded at the RPC end-point of c , and only the reply to the last request r is actually passed to the application.

An advantage of the replication related stubs on the client side is that they can accommodate the algorithmic exten-

²In the general case, the set of messages considered unstable in the stub of c is a superset of the set of c 's requests being actually unstable in the server group, i.e. $\lambda \geq \kappa$.

sions required to allow clients with asynchronous communication primitives to interact with a replicated service. In particular, the time-stamping mechanism describe in section 2 (see the discussion about overall system consistency) is extended to cope with multiple servers. An example of such a mechanism can be found in Lazy Replication [11], where logical time *vectors* are used to record the causal dependencies (as far as the service state is concerned) of client requests and inter-client messages.

Protocol 3: Replication transparent to the clients

The authors have proposed, in earlier work [9, 10], a Replication protocol for the case where clients are not aware of replication. We have shown that the State Machine requirements are met even in the case of dynamic reconfiguration of the server group. The basic principles of the protocol are discussed here. In particular, we stress the price for satisfying Uniformity without a replication related stub on the client site. The structure of clients and servers is depicted in Figure 7.

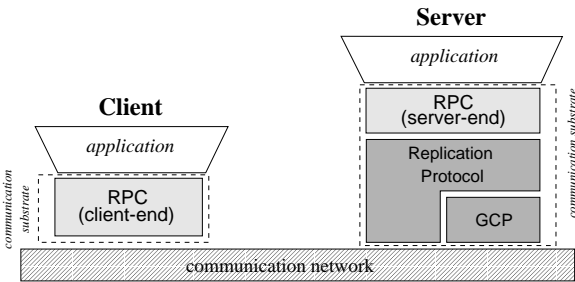


Figure 7. Protocol 3—structure of clients and replica servers.

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 uni- or a multi-cast reference.

Request delivery: Request r is sent to the service reference. In the case of replicated service provision, the service reference is a multicast address, and r is transmitted (unreliable multicast) to all replica servers joining the specific multicast address. r is received at the Replication Protocol (RP) layer of replica servers. According to a distributed deterministic function on the group membership, a single server s decides to take the responsibility to synchronise the delivery of r in the group. In particular, s generates a special synchronisation message m_g which references the unique id of r , and broadcasts m_g to the group through the GCP layer.

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; that is, all servers deliver r in the same order among other requests, satisfying the *Order* requirement of the State Machine approach. Furthermore, server s' may deliver a syn-

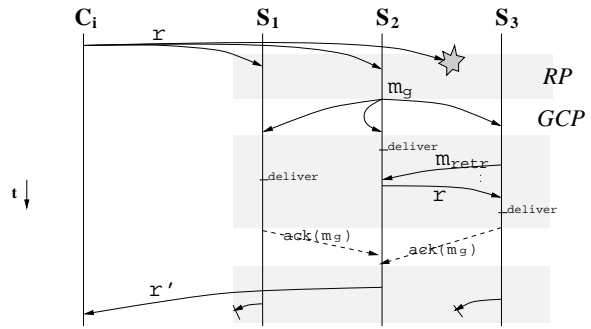


Figure 8. Protocol 3—message diagram (no group reconfiguration).

chronisation message m_g which references a request r that has not been received by s' (lost or just delayed). In that case, s' requests r from the group: at least s has already received r since it has broadcast the corresponding synchronisation message. In this way, the reliable broadcast of m_g is exploited to implement the *Agreement* requirement. Due to the unreliable nature of the communication network, a server may “miss” a request for which it would be responsible to multicast a synchronisation 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 synchronisation message for a timeout period, then it re-multicasts the request to the group. RP detects duplicate requests: a duplicate of the last delivered request of a client is passed directly (without synchronisation) to the RPC, which handles duplicates as described at the beginning of section 3 (reply retransmissions are filtered by RP in the usual way—see below); any duplicates of earlier requests are discarded by RP.

Since there are cases when client requests have to be retransmitted to the group, request messages are buffered in RP. 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 synchronisation message: m_g is stable in the group, if its delivery (from GCP to RP) has been explicitly acknowledged by every member; the RP layer of a server acknowledges the delivery of m_g only after the corresponding r is received locally (see figure 8).

Service output: The processing of a client request, at the application layer of a server, results in the transmission of a reply 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.

Group reconfiguration: Here, too, the design of RP requires a Virtually Synchronous behaviour from GCP, so that replica servers reach consistent decisions for message stability

and output onus, even in the presence of server group re-configuration. However, in this case, it is not enough for servers to deliver the same set of synchronisation messages within a specific view. A new server that joins the group in some view v may take responsibility for the synchronisation of a request r , for which a synchronisation message has already been multicast in a view earlier than v (but has not been delivered yet). To address this problem, a *Strict* variation of Virtual Synchrony [7] is required from *GCP*: a message is delivered in the view in which it has been multicast. Thus, replica servers agree on the set of requests for which synchronisation has been initiated (and completed with delivery) in a specific view (see [10] for a more detailed description of this problem).

Even with a Virtually Synchronous *GCP*, the above algorithm for delivery and output synchronisation does not address problems of client–service consistency in the case of server failures. A scenario similar to that described in section 2 may occur. In order to guarantee Uniformity, the *RP* layer implements the following safety property:

Safe output: Replica server s with the onus of transmitting the reply for request r , blocks the reply until r has been received by every other replica in the group.

Safe output is also implemented by exploiting the information for synchronisation message stability in *GCP*—the reply for request r can be transmitted as soon as the corresponding synchronisation message m_g has become stable in *GCP*. The case where s fails while the reply is blocked is considered equivalent to communication failures; that is, messages can anyway be lost on the network. The latter problem is solved by the *RPC* protocol.

The potential blocking of the reply transmission is the price to be paid for the luck of a replication related stub on the client. The performance implications of the safe output protocol are discussed in section 4.

4. Protocol implementation and evaluation

The protocols presented in this paper have been evaluated in the *Regis* distributed platform. The first part of this section discusses the basic principles used for the implementation in *Regis*. The second part presents comparative performance results for simple test application programs.

4.1. Implementation in *Regis*

Regis [13] 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 [17] 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.

The cornerstone of the system’s design is the concept of the *protocol stack*, which has been proved to simplify the development of communication protocols with negligible overhead [8]. Every communication protocol, in *Regis*, is realised as an aggregation of micro-protocols, each one implementing a sub-set of the overall functionality. Context independence and hence re-use is obtained by requiring each micro-protocol to conform to an abstract interface. Interaction between micro-protocols is exclusively based on *upcalls* [4]. End-points which provide synchronisation with user-level threads are placed at the top of the stack, while drivers which interact with the operating system (or the hardware) are placed at the bottom of the stack. Moreover, the communication end-points define the interaction style realised at the application level.

An established data path between two (or more) user-level components is supported by compatible protocol stacks at each participant. The stacks are instantiated as part of the binding procedure and are initialised with the references of remote end-points/protocols, when necessary (a reference includes information necessary for every micro-protocol in the stack). *Regis* supports dynamic stack construction at binding time. Further, it supports dynamic re-configuration of protocol stack instances; that is, micro-protocols can be introduced or removed at any time during the lifetime of a binding. Stack construction and re-configuration is implemented by means of protocol factories which employ demand-loading of micro-protocol code modules.

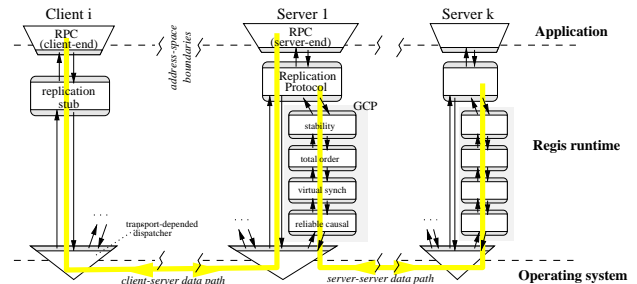


Figure 9. Protocol stack instances (clients and servers) for Protocol 2.

The replication, client–access, and group communication protocols presented in this paper have been all implemented in the form of collections of lightweight, re-usable micro-protocols. An snap-shot of the protocol stacks employed by clients and servers for the case of Protocol 2

is depicted in figure 9. In the client, the RPC end-point is placed on top of a micro-protocol layer implementing the replication stub. In the case of Protocol 3 (no replication stub assumed), the same RPC end-point would be placed directly on top of the transport layer dispatcher (this would be also the case for the client interacting with a non-replicated server). On the server side, the corresponding RPC end-point (again, the non-replicated primitive is re-used) is placed on the top of a stack that consists of a Replication micro-protocol and a collection of micro-protocols implementing *GCP*. The *RP* micro-protocol is the only part of the stack to be changed according to the replication model adopted. The *GCP* stack is re-used in the substrate of the clients, in the case of Protocol 1. The configurable nature of the stacks has facilitated experimentation with different micro-protocol implementations (e.g. for *total order*, *membership*, *reliability*), in different environments and platforms.

4.2. Performance results

In the following paragraphs, we study the performance of the current implementation of Protocols 1–3, in Regis. The experiments have been conducted 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 [5], which are directly mapped on Ethernet’s hardware multicast. IP multicast has been used for the *reliable causal broadcast* micro-protocol of *GCP* and, in the case of Protocol 3, for the transmission of client requests.

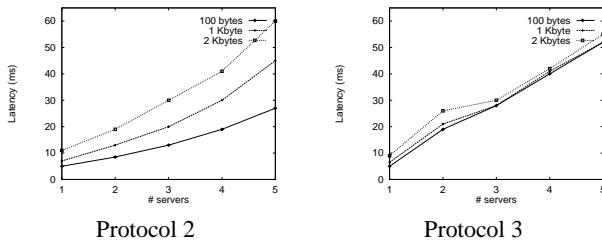


Figure 10. Latency results for the “open” group model.

Figure 10 presents the response latency for the two protocols of the “open” model. The *response latency* is defined as 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. No delay is introduced by the application layer of the servers. The latency time includes two parts: 1) the time for client–server–client interaction (approx. 2.5ms for

requests/replies of 100 bytes); 2) the time for internal server synchronisation.

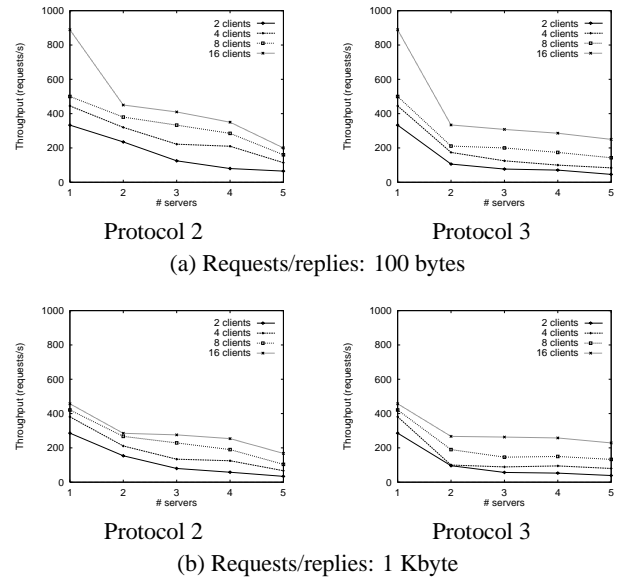


Figure 11. Throughput results for the “open” group model.

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 synchronisation messages, which are independent from the request size; on the other hand, Protocol 2 broadcasts the requests themselves among servers. The latter affects part 2 of the latency time (worse delivery times for larger messages, in *GCP*). In Protocol 3, the size of the request affects only part 1 of the latency time, which is a small percentage of the overall latency.

Figure 11 depicts throughput results for messages of 100 bytes and 1 Kbyte. The *throughput* is defined as 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 the one discussed above. Both protocols provide better throughput for larger sets of clients (more clients invoking concurrent requests) and smaller server groups. The best results are recorded for the trivial case of one-server group, where no internal synchronisation is required. Both protocols scale well for large sets of clients. The performance results presented for Protocol 2 have been obtained for an even distribution of clients to servers, creating a favourable environment for this protocol.

Figure 12 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 the clients, is shown in parentheses. Note that the number of clients is always double that of servers, as it was also the case for Protocols 2 and 3. The performance of Protocol 1 is clearly poor, despite the fact we have used the optimised version where the reply is uni-cast 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.

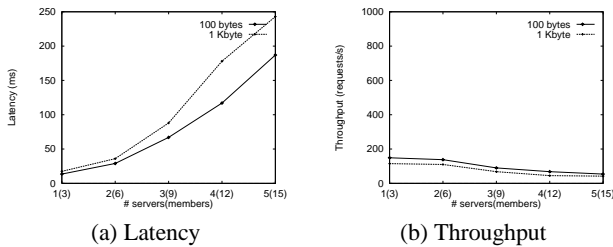


Figure 12. Performance results for the “closed” group model.

5. Conclusions

The paper has examined in detail three protocols for client access to a replicated group of servers. 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.

We have shown that the requirements of the State Machine approach *can* be satisfied by protocols that conform to the “open” group model, even in the face of dynamic reconfiguration of the server group. A typical server replication protocol is augmented with a client–access protocol to achieve client–service consistency in a way transparent to the application algorithm (Protocol 2). Uniformity is ensured by special client stubs buffering unstable requests.

However, it is not always possible for clients to accommodate replication related stubs. The paper proposes a novel protocol for an environment where replication has to be completely hidden from the clients (Protocol 3). We show that all the requirements of the State Machine approach are met ensuring consistent states among clients and replicated service. To ensure Uniformity, the server group must delay the client reply until the request becomes stable in the group.

The experimental results demonstrate that both protocols of the “open” model out-perform Protocol 1. Moreover, 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.

The performance difference of the two “open” protocols becomes 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 synchronisation messages.

Manetho [6] is a research system which also addresses the problem of interaction between a group of replicated servers and other entities in the system. In that case, output delay is avoided during normal operation, by piggybacking group output with information about the service history. This information is diffused in the system according to the causal dependencies of messages. On the event of primary’s failure (a primary-backups replication model is followed), the 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 pro-active service provision, or in the presence of internally synchronised, non-deterministic 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 to the service clients in Protocol 2.

An earlier attempt to propose client–access protocols that are independent from the actual replication mechanism has been made in the GRIP protocol [18]. GRIP focused on the specific case of the “open” model, where clients accommodate replication related stubs; a protocol similar to 2 has been proposed. 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.

The 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. The closed group approach, supported by systems such as ISIS, Horus and Transis, is more appropriate for applications where the servers must maintain a consistent view of the client set (e.g. information dissemination).

Acknowledgements

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] K. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.
- [2] A. Birrell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2:39–59, 1984.
- [3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. Technical Report 93-1374, Dept. of Computing, Cornell University, Ithaca, New York, August 1993. A preliminary version appeared in the *Proc. of the 10th ACM Annual Symposium on Principles of Distributed Computing*, Aug. 1991, Montreal.
- [4] D. Clark. The structure of systems using upcalls. In *Proceedings of the Tenth ACM Symposium on Operating Systems Principles*, pages 171–180. ACM, December 1985.
- [5] S. Deering. RFC 1112: Host extensions for IP multicasting, 1989.
- [6] E. N. Elnozahy and W. Zwaenepoel. Replicated distributed processes. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 18–27. July 1992.
- [7] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical report, Dept of Computer Science, Cornell University, Ithaca, NY, USA, March 1995.
- [8] N. Hutchinson and L. Peterson. The x-Kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, January 1991.
- [9] C. Karamanolis and J. Magee. Configurable highly available distributed services. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 118–127. IEEE Computer Society Press, Bad Neuenhar, Germany, September 1995.
- [10] C. Karamanolis and J. Magee. A replication protocol to support dynamically configurable groups of servers. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, pages 161–168, Annapolis MD, May 1996. IEEE Computer Society Press.
- [11] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing high availability using lazy replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.
- [12] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.
- [13] J. Magee, N. Dulay, and J. Kramer. A constructive development environment for parallel and distributed systems. In *Proceedings of the International Workshop on Configurable Distributed Systems*. Pittsburg, March 1994.
- [14] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high availability cluster communication. Technical Report CS94-14, Institute of Computer Science, The Hebrew University of Jerusalem, Jerusalem, Israel, 1994.
- [15] S. Mishra, L. L. Peterson, and R. D. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Proceedings of the Eighth Symposium on Reliable Distributed Systems*, pages 42–52. IEEE, Seattle, Washington, October 1989.
- [16] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1 of *ESPRIT - Research Reports*. Springer-Verlag, 1991. Project 818/2252.
- [17] N. Pryce and S. Crane. A uniform approach to configuration and communication in distributed systems. In *Proceedings of the Third International Conference on Configurable Distributed Systems (ICCDs'96)*, Annapolis MD, May 1996. IEEE Computer Society Press.
- [18] L. Rodrigues, E. Siegel, and P. Verissimo. A replication-transparent remote invocation protocol. In *Proc. of the 13th IEEE Symposium on Reliable Distributed Systems*. Dana-Point, October 1994.
- [19] A. Schiper and A. Sandoz. Understanding the power of the virtually-synchronous model. In *5th European Workshop on Dependable Computing*, Lisbon, Portugal, February 1993.
- [20] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4), December 1990.
- [21] 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 University, Ithaca, New York, 1994.