Policies for using Replica Groups and their effectiveness over the Internet

G. Morgan and P. D. Ezilchelvan
Department of Computing Science
University of Newcastle
United Kingdom
44 191 222 7972
{Graham.Morgan, Paul.Ezhilchelvan}@ncl.ac.uk

ABSTRACT

Replication is known to offer high availability in the presence of failures. This paper considers the case of a client making invocations on a group of replicated servers. It identifies attributes that typically characterise group invocation and replica management, and the options generally available for each attribute. A combination of options on these attributes constitutes a policy. The paper proposes an implementation framework which, by its group-oriented nature, simplifies the task of supporting these policies. It then considers a client (in UCL, London) making invocations on a replica group (in Newcastle, UK) over the Internet. It evaluates the response latencies for four policies that seem appropriate for this set-up. The evaluation takes into account the timing of server crashes with respect to client invocations; both real and virtual failures are considered, the latter being not uncommon in the Internet environment. The experiments are carried out using a CORBA compliant system called NewTop.

Keywords

Server crashes, group invocation, replica management, total order, policy attributes, causal precedence, latency, CORBA.

1. INTRODUCTION

Replication of entities (e.g., objects, processes) is the most commonly used approach for maintaining high availability of data despite failures. Managing replicas in a networked environment, particularly in the asynchronous environment where communication delays cannot be bounded with certainty, is a difficult task. The group paradigm (primarily concerned with application-level fault-tolerance as opposed to IP-multicast) has proven to be a useful abstraction that simplifies this task [2]. Informally, a group is a collection of distributed entities in which a member entity communicates with other members by multicasting to the full membership of the group. A replicated group refers to a group in which each member manages a copy of the same data. Building applications based on groups in general and on replicated groups in particular is considerably simplified if the members of a group can multicast reliably and have a mutually consistent view of the order in which events (such as invocations, membership changes) have taken place. By reliable multicast we mean that either all the functioning members deliver a given multicast or none of them does. An additional property required for replicated groups is total order: all the functioning members deliver a set of multicasts in the same order that preserves causal precedence. Total ordering is needed to ensure that the replica states remain mutually consistent, and that the state changes are consistent with causal precedence. Design and development of middleware systems that provide group services such as the membership service, reliable multicasts with specific ordering properties, has been an active area of research [1-6]. As distributed applications are being increasingly designed and implemented using CORBA middleware services, recent research efforts have been aimed at enriching CORBA with an object group service [7-14, 21].

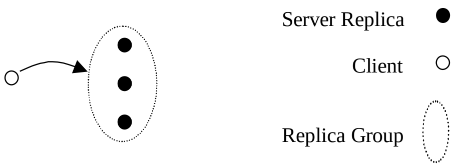


Figure 1. A client invocation of a server group

Figure 1 depicts the most common mode of invocation on a replicated group: a single client issues a request to a group of three server replicas, and waits for a response. If the client and server replicas are all connected by high-speed, low latency network, then an efficient way of invoking the replicas would be for the client to multicast to all the replicas. On the other hand, existing literature [21] indicates that if the client is separated from servers by a high latency communication path (e.g., WAN, Internet), then this method would be unattractive. So, an alternative method that would help a client avoid unicasting to *each* replica would be desirable. Such a method could be for the client to send its request to only one replica (using a single unicast), which then forwards the request to all other replicas on behalf of the client. Depending mainly on the way in which a client request is disseminated into the server group, the server replicas may wish to process the request in ways that minimise the response latency. Replicated processing is typically done in two different ways: in *passive replication*, where a single server, the *primary*, performs processing, the other servers act as backups providing tolerance to primary crash; in *active replication*, all servers process a given request in parallel, a server crash during processing does not significantly increase the response latency. When the client directs its request to only one server, making that server act as the primary (i.e., opting for passive replication) appears to offer small response times (latencies) if the primary does not crash until it completes processing the request [21].

We identify in this paper four different aspects or *attributes* in the invocation and management of a replicated object group, and options commonly available for each attribute. A combination of choices for each attribute will constitute a *policy*. We observe some policies (i.e., some choice combinations) to be practically not sensible, and some others obviously inefficient in certain environments. The experiments we carried out in [21] provide some useful insight into the effectiveness of these policies in terms of service response latencies. These experiments were carried out in a failure-free environment, and lead us to conclude that some of the policies that did well for a client that was in the same LAN as servers include active replication and could therefore be expected to do equally well masking any server crash that might occur during processing; however, the policy that did extremely well for a long-distance Internet client employed passive replication and dissemination of client request into the server group through a single server. These choices embody single points of failures, and when failures do occur the latency will undoubtedly increase. An objective of this paper is to evaluate, through experiments, how various policies perform for an Internet client in a failure-prone environment. The results and analysis presented here would enable an application developer to choose a policy that is most appropriate to the expected failure probability of the application environment.

We achieve the stated objective in a systematic and comprehensive manner. We first provide a group-oriented implementation framework that keeps track of causal precedence that might exist between distinct clients' invocations on the server group. A support for this tracking considerably simplifies the task of building applications based on replicated groups, and groups in general. Regarding failures, we assume them to be crash, i.e., a replica fails by stopping to function. We evaluate the impact of failures by considering the timing of their occurrences. Consider that a server replica has already crashed when a client unicasts its request. Upon receiving no acknowledgement, the client will detect the server crash and unicast its response to another server replica. Thus the failure detection delay is not high. On the other hand, if the first server crashes after acknowledging the request and while processing the request, the client can detect the failure only after the expiry of the timeout it has set to receive the reply. This increases the failure detection delay and the overall response latency. We consider both real and virtual failures, the latter are said to occur when the client incorrectly perceives a server replica to have crashed only because the server's response got unduly delayed due to transient partitions or network congestions that are not uncommon over WAN and Internet. We note here that many papers in the literature which report on the performance of fault-tolerant group services rarely consider failures, very rarely virtual failures; in this regard, the results in our paper represent an advancement. The experiments were conducted using a long-distance client in Univ. College London while the server replica group consisted of three servers on the same LAN in Newcastle, United Kingdom. The replica management and group invocation policies were supported by a CORBA compliant object group service called NewTop, which meets all requirements identified in our implementation framework.

The paper is organised as follows. The next section lists the attributes of group invocation and replica management, and the options generally available for each attribute. Section 3 presents and motivates the implementation framework that supports various policies and helps simplify the building of group-based applications. In the context of this framework, the impact of the timing of failures (with respect to group invocations) and virtual failures on the response latencies are discussed. Section 4 provides an overview of the NewTop system, with emphasis on aspects of its CORBA-compliant implementation that has bearings on its performance. Section 5 presents and analyses the performance figures. Conclusions are in section 6.

2. GROUP INVOCATION AND MANAGEMENT POLICIES

2.1 Invocation Policy Attributes

A client's invocation of an object group is characterised by two attributes: *request dissemination* (D) and *reply collection* (C). These attributes refer to the way in which a client sends its request to, and collects the replies from the replicated servers, respectively. A client can disseminate its request to the server group, by sending it directly to only one of the server replicas called the *request manager* (D1), or directly to all server replicas of the group (D2). In dealing with the replies generated by the server replicas, a client can exercise one of the following options: wait for no reply (C0), wait for one reply (C1), wait for all replicas' replies (C2), and wait for replies from a majority (C3). Any combination of the options for D and C can be supported for a crash fault model, as shown in figure 2 which assumes D1 for dissemination and a total order protocol within the server group.

i. *Receiving client request* - A request is sent to the request manager of the group (figure 2(i)).

ii. *Distributing client request* - The request manager multicasts the request within the server group (figure 2(ii)). This is achieved by the request manager acting as a client and issuing the incoming invocation as a new invocation (of the same type, e.g., wait for first, wait for all).

iii. *Receiving server replies* - Each member of the server group multicasts replies within the group (figure 2(iii)). (Here we assume that all server replicas process the request, as in active replication; different types of commonly used replicated processing are to be discussed shortly).

iv. *Returning server replies to client* - Server replies - one, majority or all - are gathered by the request manager and returned to the client (figure 2(iv)), depending on whether the client has chosen C1, C2, or C3 respectively. No reply is sent if C0 has been chosen.

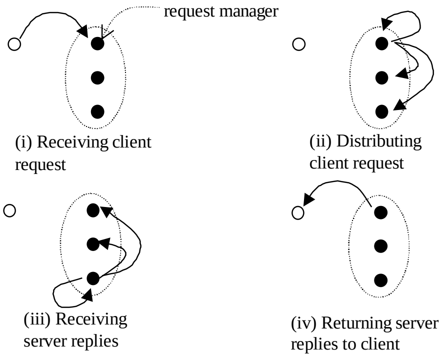

request manager
(i) Receiving client request
(ii) Distributing client request
(iii) Receiving server replies
(iv) Returning server replies to client

Figure 2. Client invocation through request manager

2.2 Replica Management Policy Attributes

Two key issues in replica management are: *replication type* (R) and protocol used for *total ordering* of concurrent requests from multiple clients (O). In *passive replication* (R1), only one replica, called the *primary*, processes the client request; for every received request, it multicasts to other replicas (i) the request itself (if necessary) before processing it, and (ii) the state changes effected and any response produced due to processing of the request. If ever the primary crashes, one of the surviving replicas becomes the new primary and continues with the processing of client requests. A group of (*f*+1) replicas can thus provide services despite at most *f* replica crashes. Passive replication cannot therefore meaningfully support result collection options of C2 and C3. In *active replication* (R2) all replicas process the request in parallel, and all options on C can be supported.

There are basically two ways of achieving total order on requests. In the *asymmetric* version (O1), one of the members of the replica group assumes the responsibility for the ordering of requests directed at the group. Such a member is commonly termed the *sequencer.* A member that wishes to multicast a message *m* will only unicast *m* to the sequencer which in turn multicasts *m* with relevant ordering information appended. In the *symmetric* version (O2), all members use the same deterministic algorithm for ordering: this requires that for a multicast to be ordered, every member other than the multicast initiator must multicast either an application message or a protocol specific message. (Figure 3 provides an example). The principles of symmetric ordering were used in the seminal paper [20] for solving the mutual exclusion problem in the absence of synchronised global time. It has been shown that symmetric ordering tends to be more attractive in situations where all the members are lively, and multicasting regularly, so the need for making protocol specific multicasts just for ordering is eliminated, whereas asymmetric protocols are better in other situations [15].

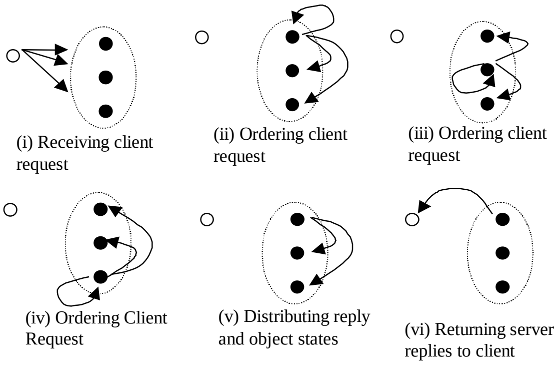

(i) Receiving client request
(ii) Ordering client request
(iii) Ordering client request
(iv) Ordering Client Request
(v) Distributing reply and object states
(vi) Returning server replies to client

Figure 3. Passive Replication with Total Ordering

2.3 Policies and their Cost-Effectiveness

A combination of choices made for attributes D, C, R and O will constitute a *policy* for invocation and management of a server group. Figure 3 depicts the exchange of messages between the client and server replicas, and also between server replicas when the policy is {D2, C1, R1, O2} where D2 stands for disseminate to all, C1 for collect (any) one reply, R1 passive replication, and O2 symmetric ordering. The client multicasts its request to all replicas (figure 3(i)). For this request to be symmetrically ordered in the server group, each replica must multicast an ordering message within the group. This full message exchange (which actually takes place concurrently) is shown in figures 3(ii) – 3(iv). Then, the primary (top server replica) alone processes the ordered request and sends the reply and state updates to other replicas (figure 3(v)); it then sends the reply to the client.

We already noted that it does not make sense to combine passive replication with C2 or C3. Thus, ignoring the exceptional cases of combining R1 with C2 or C3, there are 24 different policies for getting a service from a replica group. Of course some of these policies may be inefficient in terms of latency and message cost. For example, using asymmetric ordering (i.e., using O1 instead of O2) in figure 3 would have been more efficient as it would have avoided the need for full message exchange and therefore the multicasts by non-primary replicas (figures 3(iii) and 3(iv)). Thus, in the absence of failures, it brings performance benefits to combine R1 with O1, with the same replica acting as both the primary and the sequencer. When primary fails, R2 can respond faster than R1 if C2 is not the option. This is because C2 requires that the client receive replies from all servers that were in the group when the invocation was made; an intervening server crash needs to be detected and the client be informed of the reduction in membership when replies are being sent back. This need to detect and announce a server crash undermines the failure-masking potentials of R2. Because of its requirement on full message exchange for ordering, O2 gets slowed down by the crash of *any* member; whereas, O1 slows down only if the sequencer crashes.

Not admitting failures, we conducted in our earlier work [21] experiments for both Internet and LAN clients. For an Internet client {D1, C1, R1, O1} did extremely well in terms of response latency (nearly as good as an unreplicated server), when the same server played the role of the request manager (in D1), the primary (in R1), and the sequencer (in O1). For LAN clients, both {D1, C1, R1, O1} and {D2, C1, R2, O1} did equally well. Over LAN, O2 did equally well as O1 when replicas are kept lively, e.g., when a large number of clients are simultaneously invoking the server group. From these results, we make the following conclusions:

i. The policy {D2, C1, R2, O1} that performed well for a LAN client in a failure-free environment, can be expected to do equally well even in the presence of failures because of the failure masking potentials of R2. However, the same cannot be claimed for an Internet client since the request manager that was also the primary and the sequencer in {D1, C1, R1, O1}, constitutes a single point of failure; if it fails during processing or ordering of the received request, the surviving replicas need to detect this failure and elect a new request manager before they can respond; this increases the overall response latency. So, the most responsive fault-tolerant policy (if one exists at all) for an Internet client is yet unknown.

ii. It is the client that should decide on the D attribute of a policy, judging by its proximity to the server group; and,

iii. It is the servers who should decide on the O attribute of a policy as they alone can know how lively they are at any given time.

Note that the liveliness of servers may change with time; so the servers should switch from O1 to O2, and vice versa, provided this switching can be done at no extra cost. This is possible if the underlying object group service supports overlapping groups, which we argue in subsection 3.2 to be an essential requirement to simplify the building of group based applications. We believe that the decision on R should be a subject of negotiation between the client and servers. If a client insists on C2 or C3, the decision has to be on R2. On the other hand, if a new client requests on R1 while servers are already doing R2 for existing clients, the new request for R1 may have to be disregarded as it is simple and efficient for servers to process all requests by one form of replication. To illustrate this point, consider a server group (s_1, s_2, s_3} having to process requests r_1, r_2, .. r_p, .. r_n, n >p, in that order. Say, processing is by R2 except r_p needs to be processed by R1 with s_1 acting as the primary. After processing r_p and before processing r_{p+1}, s_1 must halt processing, checkpoint its state, and multicast the checkpoint. The other replicas, after processing r_{p-1}, must update their states using s_1's checkpoint before continuing with r_{p+1}.

3. A GROUP-ORIENTED IMPLEMENTATION FRAMEWORK

3.1 Assumptions

It is left to the server group to nominate the request manager (if D1 is opted for), the primary (if R1 is opted for), and the sequencer (for O1). We make two assumptions for reasons of better performance: within the server group, the same member is designated to perform the roles of request manager, primary, and sequencer when required; also, the same member acts as the request manager for different clients that opt for D1.

Note that, in theory, different members can take up the role of the request manager, the primary, and the sequencer. But when the same member plays all these roles, message cost is reduced. Similarly, different members can act as the request manager for different clients. For example, if server replicas are geographically apart, the replica that is closer to a client can act as the request manager for that client. In this paper, we regard the server replicas to be on the same LAN; therefore, assigning different managers for different clients does not appear to bring any obvious advantages.

3.2 Group Invocations as Group Communication

Figure 4(i) depicts invocation of a server group gx by clients A and B. B makes an invocation m1 of type D1 and C0. It then communicates (m2) with A. After processing m2, A invokes gx by sending m3 to the request manager. Now suppose that A's message m3 reaches the request manager before B's m1. If messages are processed in the received order, causal relation [20] is violated, as m1 causally precedes m3 and must therefore be processed by gx before m3. It is now left to the application developer to ensure that the processing of requests respects the causal precedence. The developer is relieved of this burden when invocations are made within groups as shown in figure 4(ii) and if the group management service permits an entity to be a member of more than one group and satisfies the following message delivery requirements.

i. *multi-group causal precedence*: say s is a member of groups gx and gy in which m_x and m_y are multicast respectively. If m_x causally precedes m_y, then s delivers m_x before m_y.

ii. *multi-group identical order*: say s and s' are members of groups gx and gy in which m_x and m_y are multicast respecively. s delivers m_x before m_y if and only if s' delivers m_x before m_y.

Figure 4(ii) depicts the case when invocations are made as group communications. B issues m1 in gy that is made up of B and the request manager of the server group gx. Note that gx and gy overlap as the request manager is a member of both the groups. B then sends m2 in gz which consists of clients A and B. After processing of m2, A issues m3 in gw which overlaps with gx due to the common membership of the request manager. Property (a) ensures that the request manager of gx delivers m1 first and then m3.

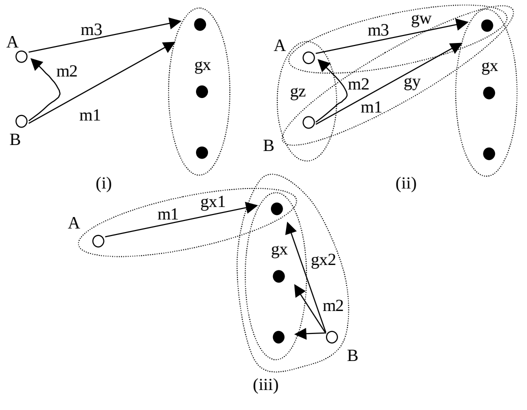


Figure 4. (i) Direct Invocations. (ii) Invocations are group communication. (iii) Simultaneous support for different ordering protocols.

The Isis system was first to introduce overlapping groups [2]; the AQuA system [12] also uses overlapping groups in a variety of ways for replica management. The group communication protocols used in NewTop have been designed to cope with overlapping groups in an efficient manner [5]; in particular, they allow a multi-group member to simultaneously execute symmetric protocol in one group and the asymmetric protocol in the same or in another group. This flexibility is exploited to support the simultaneous executions of both the ordering protocols by server replicas. In figure 4(iii), clients A and B multicast their invocations m1 and m2 in groups gx1 and gx2, repectively. Suppose that A has preferred the asymmetric total ordering and B the symmetric ordering. The request manager delivers m1 using NewTop in gx1. (It should not deliver m1 simply upon reception; otherwise causal relation may not be preserved in cases shown in fig 4(i).) Having delivered m1, the request manager, acting as the sequencer, would initiate the asymmetric ordering of m1 in gx. At the same time, all replicas will be executing the symmetric protocol for ordering m2 multicast in gx2. By property (b) all replicas are guaranteed to order m1 and m2 identically. To be able to simultaneously execute both protocols in the same group, say gx, two logical groups, g_{xa} and g_{xs}, are formed in a logical sense out of the same physical gx, to execute asymmetric protocol in g_{xa} and symmetric protocol in g_{xs}. By property (b) all members of gx, present in both g_{xa} and g_{xs}, are guaranteed to identically order all messages delivered in gx.

3.3 Client/Server Groups and the Effects of Failures

A ‘down-side’ to requiring that client invocations be done as group communication is that a client should first form a group with the request manager (in case of D1) or with all servers (in case of D2). This obviously incurs an overhead in the form of messages exchanged to form the group which need not be done if invocations were sent directly as shown in figure 4(i). We, however, believe this overhead to be small if the client is to negotiate with the server group on certain policy attributes, as the information related to group formation can be piggybacked onto these negotiation messages.

We call the group that contains a client and one or all servers the *client/server* group. It is said to be *one-inclusive*, or simply *one-clusive*, if it includes only one server (the request manager) as in fig 5(i); it is said to be *all-inclusive*, or simply *inclusive*, if it includes all server replicas (see fig 5(ii)).

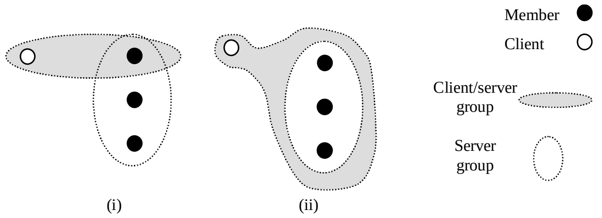


Figure 5. Client/server groups. (i) one-clusive and (ii) inclusive groups.

In fig 5(i), a failure of the request manager will cause (a) the surviving servers to deliver a view-change message indicating the change in the membership of the server group, and (b) the binding between the client and the request manager to be broken and the one-clusive group be disbanded. The client has to form another one-clusive group with the new request manager elected within the (new) server group. Consider this scenario further. Assume that the request manager fails as the servers are multicasting their replies (during the stage depicted in figure 2(iii)). The server group will be reformed with the request manager removed, and no reply will be sent to the client. Client retries can be handled by the new request manager without causing re-execution, provided retries contain the same call number as the original call and servers retain the data of the last reply message (enabling the request manager to resend the reply). These are ‘standard’ techniques used in any RPC implementation.

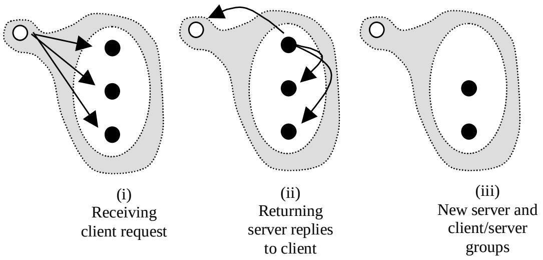

(i)
Receiving client request
(ii)
Returning server replies to client
(iii)
New server and client/server groups

Figure 6. Passive Replication and Failure Handling in Inclusive groups.

In the inclusive group, server failures do not cause the client to form any new group; the client only delivers a view-change message indicating the change in the membership of the client-server group. Let us revisit Figure 3 which depicts the case of the client’s policy: {D2, C1, R1, O2}. With the client now making its invocation within the client/server group, only figures 3(i) and 3(vi) change which are shown in figures 6(i) and 6(ii) respectively. The client’s request (figure 6(i)) and the primary’s reply (figure 6(ii)) are multicast to the full membership of the client/server group. Suppose that the primary crashes before multicasting the reply. The new client/server group and the server group formed are shown in fig 6(iii). When the group communication service supports *virtual synchrony* [2], all members that go on to form the new client/server group are guaranteed to have delivered an identical set of messages in the old client/server group view. Thus, the new primary will know whether or not the client has delivered the reply sent by the old primary prior to crash. So, the fact that the old primary crashed before multicasting the reply, is detected and handled; further, the primary crash increases the response time by the time it takes for the surviving members of the client/server group to effect the membership change in the *virtually synchronous* manner.

Recall that the request manager may constitute a single point of failure in a one-clusive group, and the timing of its failure with respect to the client’s dissemination of its request is significant in determining the response latency. Suppose that the request manager crashes after the formation of the one-clusive group and before the client multicasts its request in the group. We call this scenario *pre-send*. When the client attempts to multicast its request, the underlying communication service (if it is TCP/IP connection as is the case with many ORBs) will inform the client that the multicast is unsuccessful; thus, the failure detection latency is small. On the other hand, consider the *post-send* failure scenario: the request manager crashes after receiving the request from the client and at some time before returning the reply to the client (i.e. before stage (iv) in figure 2). The client can suspect the request manager's crash only after the expiry of the timeout it had set for receiving the reply; this may mean a large failure-detection latency. Thus, a post-send failure may result in a large response latency which could include the time taken to detect a failure and to rebind, and the failure-free response latency if the request manager had been acting as the primary. The notion of pre- and post-send failures also exist in the inclusive groups: crash of any server before and after that server received the client multicast, respectively. A pre-send or post-send failure in an inclusive group leads to membership changes in both the client/server and server groups (i.e. change from fig 6(ii) to 6(iii)) before the request is sent or while the request is being processed.

3.3.1 Virtual Failures

Three clients A, B and C form one-clusive groups with the server group gx (see fig 7(i)). Suppose that A is a long-distance client while B and C are in the same LAN as the server replicas, and that the reply from the request manager to A gets delayed due to network congestion over the Internet. A's timeout will expire leaving A to conclude that the request manager has crashed. That is, from A's point of view the request manager appears to have crashed, we call this apparent failure a *virtual failure*. Having decided that the manager has crashed, A will then try to form a new inclusive group, say it succeeds in forming it with the third server as the next request manager. Now, as seen in fig 7(ii), different clients do not use the same request manager, which as indicated in 3.1, degrades the performance. For perfomance reasons, the third server is programmed to reject the client's attempt to form the new client/server group with itself, if it sees the original request manager not crashed; thus A is forced to re-form the old client/server group which it disbanded by mistake.

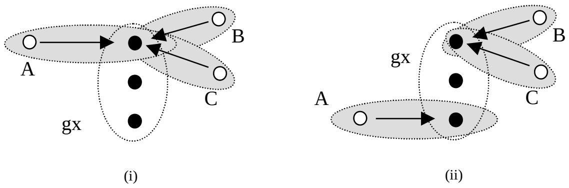


Figure 7: Effect of Virtual Failures in One-clusive groups.

In inclusive groups, the effect of virtual failures depends on how failure suspicions are acted upon, which can be done in one of two ways: in *stable suspicion* model, a member trusts and acts on another member's suspicion, even though it has not itself suspected a failure. Systems such as Transis[1, 4], Isis[2], adopt this model. In *refutation* model, a member keeps the received suspicion in abeyance until it independently (with its own timeout) assesses that it also observes the reported suspicion; if it finds the received suspicion to be untrue, it *refutes* the reported suspicion. A single refutal is enough to suppress a suspicion groupwide, and thus false suspicions are in effect discarded and prevented from causing unnecessary membership changes. Since NewTop permits refutation, a long distance client's incorrect suspicion (such as A's in the example above) will be refuted by another server that does not share the same suspicion. So, virtual failures do not lead to costly membership changes.

4. OVERVIEW OF THE NEWTOP OBJECT GROUP SERVICE

The NewTop object group service, or NewTop service for short, itself has been composed of a group communication subsystem that handles membership and reliable multicasts and an invocation subsystem. The architecture of the NewTop service is depicted in figure 8. The function of the invocation layer is to support various invocation and management policies idenitifed in section 2. The figure shows how a request-reply interaction between a client and a server group is handled (only a single server is shown). The client application makes its request to the NewTop service; internal to the service, the request is handled by the invocation layer which then uses the group communication service to send NewTop specific message to servers; the message then travels up and down the protocol stack on the server side. The invocation layer employs the chosen policy to implement request-reply interactions.

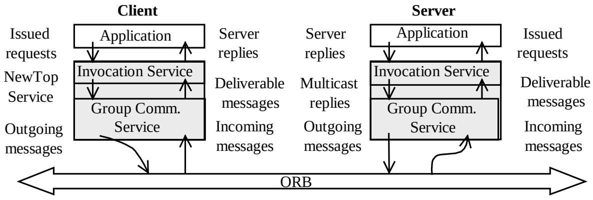


Figure 8. System architecture

The underlying group communication service has been designed to be suitable for a wide variety of group based applications; objects can simultaneously belong to many groups, group size could be large, and objects could be geographically widely separated. The service can provide causality preserving total order delivery to members of a group, ensuring that total order delivery is preserved even for multi-group objects. Both symmetric and asymmetric total order protocols are supported, permitting a member to use say symmetric version in one group and asymmetric version in another group simultaneously [5].

The failure assumptions made by the NewTop service are as follows. Processes/objects fail only by crashing, i.e., by stopping to function. The communication environment is modelled as asynchronous, where message transmission times cannot be accurately estimated, and the underlying network may well get partitioned, preventing functioning members from communicating with each other. The actual protocols used in the NewTop service will not be described here, as these details are not directly relevant to this paper; the interested reader is referred to [5].

The group communication system provides clients (via the invocation layer interface) with *create*, *delete* and *leave* group operations and causal and total order multicasts. In addition, it maintains the membership information (group view) and ensures that this information is kept mutually consistent at each member. This is achieved with the help of a failure suspector that initiates membership agreement as soon as a member is suspected to have failed. A member can obtain the current membership information by invoking 'groupDetails' operation. View updates are atomic with respect to message deliveries, as in *virtually synchronous communication* [2]. Message delivery is atomic with two types of ordering guarantees (causal and causality preserving total order) and in the case of total order, two types of ordering techniques, symmetric and asymmetric, are supported.

In a group communication system a member is often required to stay lively within a group to avoid being suspected by other members. This usually takes the form of a member periodically sending "I am alive" or "NULL" messages during periods it has no application level messages to send. In NewTop, after a member has neglected to send a message for a period of time, the NewTop time-silence mechanism will send a "I am alive" message. For further details, see [13, 21].

4.1 Related Work

NewTop implements group communication services as CORBA services 'from scratch'. In addition to being CORBA compliant, the advantage here is that the services are directly available to application builders so can be used for a variety of purposes. This approach was first developed in the Object Group Service (OGS) [7,8], and has been taken in the NewTop service. The NewTop service offers a more comprehensive set of group management facilities than OGS. In particular, OGS does not support overlapping groups.

The service approach we have taken for building NewTop needs to be contrasted with approaches taken elsewhere, as it has bearings on the relative system performance. There are two other ways of incorporating object groups in CORBA (see [7,8] for details). The *integration* approach takes an existing group communication system and replaces the transport service of the ORB with the group service [9]. Although this is a very efficient way of incorporating group functionality in an ORB, this approach is not CORBA compliant, lacking in interoperability.

In the other approach, called the *interceptor* approach, messages issued by an ORB are intercepted and mapped on to calls of a group communication system. Well known examples of this approach are the Eternal [10,11] and AQuA [12] systems; Eternal uses the Totem group communication system [6], whereas AQuA uses the Ensemble group communication system [3]. The need to intercept calls makes these systems platform dependent. Both Eternal and AQuA make use of group communication for supporting object replication only (and not for other uses of group communication, such as collaborative applications). They do so by using the inclusive approach, and have been engineered for use in high speed LAN environments, rather than over the Internet. Consequently, these systems can efficiently support certain policies, but cannot be flexible enough to support any given policy that is deemed efficient in a given setup, say, policies appropriate for an Internet client.

The NewTop service, being fully CORBA compliant, has to rely only on the standard ORB message passing mechanisms. Since at present ORBs only provide one to one communication, multicasting has to be implemented as multiple unicasts – a thread is created to handle each synchronous unicast. Using multiple threads of execution obtains parallelism and prevents client blocking. Such a measure to prevent blocking will not be required had the ORB supported asynchronous invocation. A multicast is more time consuming in NewTop than in Totem [6] or Transis [1,4] which assume a broadcast network. NewTop can however be adapted to exploit forthcoming enhancements to ORBs. As part of the ongoing development of CORBA, the OMG have recently adopted interceptors, messaging, and fault-tolerance specifications. Availability of ORBs with interceptors will enable the use of NewTop as a multicast transport service as demonstrated by the Eternal system. Exploitation of the messaging service will enable more efficient implementation of multicasting than is possible now. In certain applications, our object group service will need to be used in conjunction with additional subsystems that provide specific functions; for example, in order to support passive replication, some form of state transfer facility would have to be implemented. We have shown elsewhere how a subsystem for replication of transactional objects (that itself uses the CORBA transaction service) can make use of the object group service [16].

Although not a CORBA service, the system described in [17] is worth mentioning. The paper describes a client access protocol for invoking object replicas, without the need for the client to use multicasts. We obtain the same functionality by making use of one-clusive groups.

5. PERFORMANCE EVALUATION

Using the NewTop service, we measure the response latencies for various policies for an Internet client. As the load on the Internet is not static, these measurements should not be treated as 'absolute' figures, but rather as an aid to compare the effectiveness of different policies; for a fair comparison, all experiments were conducted overnight during which load fluctuations over the Internet were small. To keep the experiment space finite, we fix the type of replication to be active (R2) and the reply-collection to wait for all (C2). Thus, four policies are possible with D and O as the parameters. To help analyse performance figures, we present a brief description of the message passing involved for different policies, giving particular emphasis on the use of ordered and unordered multicasts – the latter being delivered straight after reception. Recall that in asymmetric ordering, a member *M* that intends to multicast *m*, actually unicasts *m* to the sequencer which then appends the ordering information and multicasts *m*. The sequencer can order *m* as soon as it decides on the ordering information for *m*, and any member (including the sender *M*) as soon as it receives *m* from the sequencer. In the symmetric protocol, however, *M* multicasts *m* in the group; for any member (including *M*) to deliver *m*, every other member *M'* must perform a multicast in that group. That is, one *full message exchange* (FME for short) between members must happen for *m* to be delivered. It is particularly costly in a client/server group because of the presence of long-distance member(s) (the client for servers, and servers for the client).

P1 *One-clusive asymmetric* – Referring to figure 5(i), the client issues request to request manager which is also sequencer for both the server and client/server groups. Request manager delivers the request preserving causal precedence in the client/server group, and then multicasts request in server group. Each server immediately delivers the received request, and sends the reply as unordered multicasts within the server group. Servers ignore other servers' replies they deliver. The request manager bundles the replies and sends the bundle to the client (see figure 2(iii) - (iv)).

P2 *Inclusive asymmetric* – Referring to figure 5(ii), client issues request to request manager which is also sequencer for the client/server group. Sequencer appends ordering information onto the received request, and multicasts it to all members of the client/server group. Each server immediately delivers the received request and the client ignores its own request received from sequencer. Servers send their replies as unordered multicasts in the client/server group. Compared to the previous policy, the additional cost $\triangle C_{21}$ is due to multicasts having to be carried out in the client/server group: sequencer additionally carries out a long distance transmission when it multicasts the ordered request in the client/server group, and other servers makes one long-distance send when they multicast their replies in the client-server group.

P3 *One-clusive symmetric* – Client issues request to request manager (see figure 5(i)) which delivers the request as in policy P1 (*one-clusive asymmetric*). After delivering the request in the client/server group, request manager distributes the request in server group by executing symmetric protocol. After one FME, each server delivers the request; as in one-clusive asymmetric case, each server sends its reply as unordered multicasts within server group and the request manager unicasts the bundled message to client. The additional message cost $\triangle C_{31}$ over policy P1 is: servers other than the request manager contributing to FME for symmetrically ordering the request within server group. Since servers are all in the same LAN, $\triangle C_{31}$ is expected to be very small. Both P1 and P3 being *one-clusive*, they are equally vulnerable to a single server failure.

P4 *Inclusive symmetric* – Client multicasts request to all servers using symmetric protocol executed in the client/server group. After one FME (within the client/server group), each server delivers the request and sends its reply as unordered multicasts within client/server group. As in the other inclusive case, the client delivers all server replies while the servers ignore other servers' replies. The additional message cost $\triangle C_{43}$ over policy P3 is: client making two long-distance unicasts, each server making one long distance unicast (to client) for contributing to FME, and servers other than the request manager, making a long-distance unicast (to client) when multicasting the result. This case however has the most masking potential against server failures.

For each of these policies, we consider both pre-send and post-send failures and a failure can be real or virtual. A client never crashes, nor is ever suspected to have crashed by the servers.

A client issues a request to the server group and waits for their replies. Clients were configured to issue requests as frequently as possible; as soon as a reply is received, another request is issued. The server used in the experiment is a CORBA object that simply returns a pseudo random number when requested to do so by a client. Client numbers were increased gradually from one to ten. At each of these increments each participating client is timed for 100 requests, and the average is taken.

5.1 Implementation Details

Communications between clients and servers were enabled via the Internet. Pentium Linux machines were used as hosts for clients and servers. The server group was made up of three members which were on the same LAN in Newcastle (United Kingdom), and one server was designated to be request manager for one-clusive options. All clients were located in London. The ORB used was omniORB2 [19]. C++ was the implementation language.

To aid in ensuring a deterministic approach to request manager failure, each client request is associated with a sequence number Rx and the client's unique id Cy. In each experiment, appropriately marked requests would cause one of four server failure scenarios: pre- /post-send true/virtual failues; the following describes the activities involved in measuring latencies with request numbered Rx from client Cy being marked as:

i. *Pre-send true failure* – the server designated to be request manager fails after request Rx from client Cy has been processed by the server group (i.e., Cy receives reply for Rx). Thus, when Cy makes next request R_{x+1}, say at time T, it will encounter a (true) failure. The period between T and the time when Cy gets two replies for R_{x+1}, is noted as the latency.

ii. *Post-send true failure* – the request-manager designate fails after successfully receiving Rx from Cy, but before it sends any subsequent messages to other servers or clients. The time taken for Cy to obtain two replies for Rx is noted.

iii. *Pre-send virtual failure* – while receiving Rx from Cy the request-manager designate closes the network connection to Cy, causing Cy to catch a network exception and suspect a failure. The time taken for Cy to obtain three replies for Rx is noted.

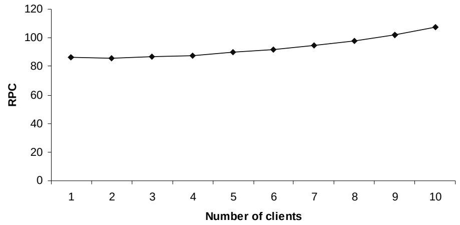
iv. *Post-send virtual failure* – after receiving Rx from Cy the request-manager designate stops sending messages only to Cy until it receives a message (from another server) refuting Cy's suspicion. In one-clusive group, the client will attempt to form a new client/server group with another server, which will deny client's invitation (for details, see subsection 3.3.1) by sending *refutal* messages to the client and to other servers.

5.2 Non-Replicated Service

To enable comparative analysis of the performance figures, the CORBA RPC time of a client in London communicating with a single server in Newcastle without the use of NewTop was obtained. The average latency was 78 ms.

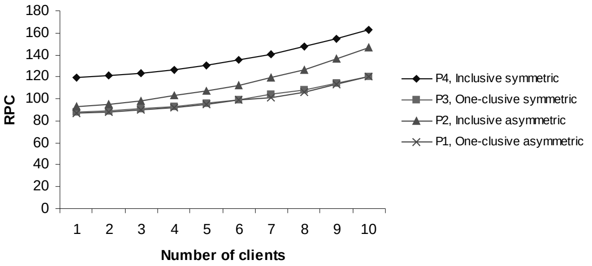
The experiment was repeated, with invocations being made via the NewTop service with client numbers 1 through 10. The latency figures obtained are shown in graph 1.

Graph 1 - Non Replicated Server - RPC (milliseconds)



The first observation to be made is that the RPC time of a single client making a call via the NewTop service is around 10 ms greater than the performance of a single client making an RPC without the NewTop service. This drop in performance may be explained by the way the NewTop service handles deliverable messages; once ordering and delivery guarantees have been satisfied the NewTop service delivers a message to the application level via a CORBA RPC. Although the NewTop service is making this call locally, the messages related to such a call must still pass through the ORB infrastructure, an expensive procedure due in part to the unnecessary mechanisms associated to network communications that the ORB applies (e.g., parameter marshaling). In the new generation of ORBs, application developers may by-pass such mechanisms when they are not required.

Graph 2 - Replicated Server with no failures - RPC (milliseconds)

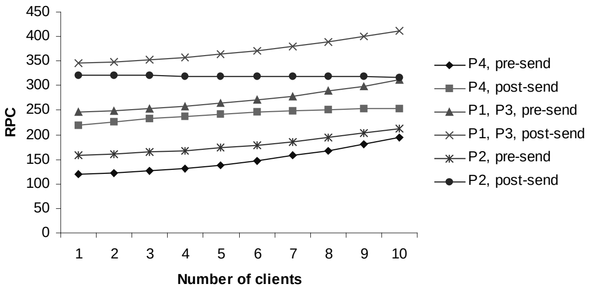


Graph 2 presents the latencies in the absence of failures. Consider first the policies (P1 and P3) with one-clusive option which do nearly as well as the non-replicated case (graph 1), since the (long distance) client needs to interact only with the request manager (which is also the sequence manager in P1). The difference between them also is marginal as $\triangle C_{31}$ (due to FME in server group over LAN) is small. (In later graphs, we do not distinguish these two policies, as the difference is too small for the scale we have to choose). The latency for policy P2 (inclusive, asymmetric) is more than that for P1 due to $\triangle C_{31}$. This difference increases with the number of clients, as the sequencer in P2 is dealing with a larger group that includes a long-distance client, and hence the load on the NewTop service increases correspondingly. The policy P4 of inclusive and symmetric is the most message costly policy. This is reflected on the high latency it yields; however this high latency, unlike in P2, does not get worse for small numbers of clients. This is because, as the number of clients increase, the servers (which get included in each client/server group formed) become livelier which is good for the symmetric protocol. As a final observation on graph 2, the difference between the best and the worst latencies is only about 40 ms.

5.3 True Failures

Graph 3 presents the measurements with the (real) failure of request-manager designate, occurring just before the client starts sending its request (pre-send) or after the request is received but not acted on (post-send). As the request manager constitutes a single point of failure in P1 and P3, its failure has the most devastating effect involving failure detection and formation of a new one-clusive group. The post-send, one-clusive case is slower than the pre-send, one-clusive case by 100 ms, which is the timeout used by the client to receive a reply from the request manager. In fact, compared to the results in the previous graph, the effectiveness of policies are simply reversed in this graph.

Graph 3 - True Failures - RPC (milliseconds)



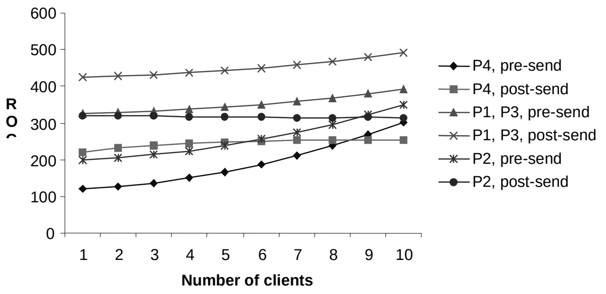
We will make two observations over the latencies for (P2, pre-send) and (P2, post-send). They are smaller than the latencies for (P1, pre-send) and (P1, post-send), respectively. This indicates that the formation of a new group is more time-consuming than effecting a membership change in an already formed group. Further, the latencies for (P2, post-send) approach those for (P2, pre-send) as the number of clients increases. This can be explained as follows. Consider a single client A. In P2, A sends its request only to the sequencer and the other two servers in the client/server group are unaware of this and therefore of the fact that the sequencer is doing nothing to order A's request. A then waits on timeout whose expiry alone can lead to failure detection. Now, introduce the second client B. If the functioning servers are expecting the sequencer to multicast its result message in the client/server group for B, then they will detect the failure of the sequencer and effect a membership change not just in the client/server for B but also in the client/server for A. Thus A may detect the failure early because of B's request being processed. So, the more clients, the larger is the likelihood of rapid failure detection with respect to a given client in the post-send scenario. In the limit, the failure detection becomes as rapid as in the pre-send case.

The convergence of post-send with pre-send is more pronounced in the case of P4, where rapid detection is helped not just by the presence of more clients but also by the fact that a client multicasts its request to *all* servers. This fact has two implications. First, by receiving a request (directly) from the client each server executes symmetric protocol, expecting every server to contribute to an FME. The prolonged absence of the crashed server will cause the functioning members to call for a change of membership in every client/server group. Thus, even if there is only one client, the failure is suspected well ahead of the expiry of client's timeout. This explains the latency for (P4, post-send) is better than the latency for even (P1, pre-send): rapid failure detection in P4 coupled with costly group formation in P1. The second implication explains the smaller latencies of (P4, pre-send) compared to (P2, pre-send), though inclusive client/server group is used in both cases. Say, there is only one client, A, which first sends its request at T - by a unicast to sequencer in P2 and by multicast in P4; let RTT_{cs} be the round trip time between A and any server, and *MC* be the time it takes to effect a group membership change after a failure is first suspected by a functioning member in the group. In the case of (P2, pre-send), A can suspect the server failure (through the absence of ack) no earlier than T + RTT_{cs}; it can know of the new sequencer no earlier than T+ RTT_{cs} + *MC*, and the new sequencer will first receive client request only by T+ RTT_{cs} + *MC* + $RTT_{cs}/2$. With (P4, pre-send) however, functioning servers receive A's multicast by T+ $RTT_{cs}/2$, effect the membership change by T+ *MC* + $RTT_{cs}/2$, and also order the request by that time as the membership change involves an FME which helps order the request. That is, processing begins in P4 early by at least RTTcs.

5.4 Virtual Failure

In case of virtual failures, policies (P1 and P3) involving one-clusive options suffer a double 'blow': the client is rebuffed when it attempts to form a new client/server group with another server and then it re-forms the client/server group with the old request manager. These time consuming activities increase the latencies for P1 and P3, compared to the previous case where re-formation does not take place. Apart from P1, P3 and (P2, pre-send), the measurements are quite similar to graph 2. This is because a failure suspicion, correct or not, does initiate the protocol for membership change; when suspicion is incorrect, it is refuted by an unsuspected member and this refutal terminates the protocol. A server can refute an incorrect suspicion quickly, if it has been in contact with the suspect recently, which happens when servers are kept active either due to a large number of clients accessing them or because the client's request is sent to all servers through symmetric protocol. (In NewTop, the suspicion initiator must indicate the last message it had received from the suspected; any member that has received from the allegedly crashed member a later message, can instantly send a refute).

Graph 4 - Virtual Failures - RPC



The under-performance of (P2, pre-send) can be explained as follows. Here, the client only attempts to unicast its request to the sequencer, but does not go ahead with the transmission as it wrongly suspects the sequencer to have failed. So, it engages itself in an unsuccessful attempt to change the membership and get a new sequencer. This waste of time is the cause for the under-performance. Whereas in (P2, post-send), the attempt to change the membership proceeds in parallel with the ordering and processing of the request, and hence the client's false suspicion has no impact on the latency.

5.5 Observations

The measurements and the comparative analysis lead to the following observations. In the failure-free scenario, the one-clusive approach performs well, irrespective of the ordering protocols used. The performance becomes the poorest in the presence of real/virtual failures. The inclusive symmetric policy P4 is best suited to failure prone environments, though it under performs in the absence of failures by a margin of 40 ms which is about half the round trip time of 83 ms. A long-distance user of a replicated service has two extreme policy options: P1 and P3 if failures are deemed rare and the high cost incurred when failures do occur is acceptable; or, P4 if the user is particular about fault-tolerance and is willing to pay a small performance penalty when failures do not happen. Between these two extremes lies P2 except for the deteriorating performance with increasing number of clients in the case of pre-send virtual failures. A remedy for this can be for the client to switch its policy to P4 once it suspects a failure of the sequencer while it is unicasting its request. Note that in (P2, pre-send) the request does not get sent to the sequencer; therefore when the client re-sends its request with P4 option, it is not duplicating its request. If the servers are programmed to detect duplicate requests, client can switch from P2 to P4 even in post-send cases. Of course, a switch from P2 to P4 and back would require servers to switch between ordering protocols. The implementation framework presented in section 3.2 identifies the requirements on the object group service systems to support this flexibility, and the NewTop service meets these requirements.

6. CONCLUDING REMARKS

We have identified various policies for managing and making invoctions on a replica server group; the policy attributes and options on them, which constitute a policy are presented. We then presented an implementation framework to support these policies. By being group-oriented and supportive of overlapping groups, this framework relieves the application developer of having to track causal dependencies between client requests, and also provides a flexible support for servers to switch between order protocols. An overview of the NewTop service, which meets the requirements identified within this framework, is presented. Using this system, relative effectiveness of four different policies for an Internet client was analysed. Experiments allowed both real and virtual failures, the latter are incorrect suspicions caused typically by load fluctuations over the Internet. Certain policies did extremely well (nearly as well as unreplicated server) in the absence of failures but performed poorly when failures happened, and vice versa. A compromise between these two extremes is possible, if a policy shift is done whenever a client suspects a failure. For this policy shift to be possible, the group membership service must support servers that can switch between order protocols – a service which any system, like NewTop, that supports overlapping groups should be able to provide.

7. ACKNOWLEDGEMENTS

We would like to thank Professor S. K. Shrivastava for his help and advice during the preparation of this paper. G. Morgan was supported by EPSRC CASE PhD studentship with industrial sponsorship from HP Laboratories, Bristol.

8. REFERENCES

[1] Amir, Y., et al, "Transis: A Communication Sub-system for High Availability", Digest of Papers, FTCS-22, Boston, July 1992, pp. 76-84.

[2] K. Birman , "The process group approach to reliable computing", CACM , 36, 12, pp. 37-53, December 1993.

[3] M. Hayden, "The Ensemble system", PhD thesis, Dept. of Computer Science, Cornell University, 1998.

[4] D. Dolev and D. Malki, “The Transis approach to high availability cluster communication”, CACM, 39 (4), April 1996, pp. 64-70.

[5] P. Ezhilchelvan, R. Macedo and S. K. Shrivastava, “NewTop: a fault-tolerant group communication protocol”, 15th IEEE Intl. Conf. on Distributed Computing Systems, Vancouver, May 1995, pp. 296-306.

[6] L.E. Moser, P.M. Melliar-Smith et al, “Totem: a Fault-tolerant multicast group communication system”, CACM, 39 (4), April 1996, pp. 54-63.

[7] P. Felber, R. Guerraoui and A. Schiper, “The implementation of a CORBA object group service”, Theory and Practice of Object Systems, 4(2), 1998, pp. 93-105.

[8] P. Felber, “The CORBA Object Group Service: a Service Approach to Object Groups in CORBA”, PhD thesis, Ecole Polytechnique Federale de Lausanne, 1998.

[9] S. Maffeis, “Run-time support for object-oriented distributed programming”, PhD thesis, University of Zurich, February 1995.

[10] P. Narasimhan, L, E. Moser and P. M. Melliar-Smith, “Replica consistency of CORBA objects in partitionable distributed systems”, Distributed Systems Eng., 4, 1997, pp. 139-150.

[11] L.E. Moser, P.M. Melliar-Smith and P. Narasimhan, “A Fault tolerance framework for COBRA”, Proc. of 29th Symp. On Fault Tolerant Computing, FTCS-29, Madison, June 1999.

[12] M. Cukier et al., “AQuA: an adaptive architecture that provides dependable distributed objects”, Proc. of 17th IEEE Symp. on Reliable Distributred Computing (SRDS’98), West Lafayette, October 1998, pp. 245-253.

[13] G. Morgan, S.K. Shrivastava, P.D. Ezhilchelvan and M.C. Little, “Design and Implementation of a CORBA Fault-tolerant Object Group Service”, Distributed Applications and Interoperable Systems, Ed. Lea Kutvonen, Hartmut Konig, Martti Tienari, Kluwer Academic Publishers, 1999, ISBN 0-7923-8527-6, pp. 361-374.

[14] S. Misra, Lan Fei, and Guming Xing, “Design, Implementation and Performance Evaluation of a CORBA Group Communication Service”, Proc. of 29th Symp. On Fault Tolerant Computing, FTCS-29, Madison, June 1999.

[15] L. Rodriguez, H. Fonseca and P. Verissimo, “Totally ordered multicasts in large scale systems”, 16th IEEE Intl. Conf. on Distributed Computing Systems, Hong Kong, May 1996, pp. 503-510.

[16] M.C. Little and S K Shrivastava, “Implementing high availability CORBA applications with Java”, Proc. of IEEE Workshop on Internet Applications, WIAPP’99, San Jose, July 1999.

[17] C. T. Karamanolis and J.N. Magee, “Client access protocols for replicated services”, IEEE Transactions on Software Engineering, Vol. 25, No. 1, 1999, pp. 3-22.

[18] G. Morgan, “A middleware service for fault tolerant group communications”, Phd thesis, Dept. of Computing Science, University of Newcastle upon Tyne, September 1999.

[19] www.uk.research.att.com/omniORB/omniOB.html

[20] L. Lamport, "Time, Clocks, and Ordering of Events in a Distributed System", Communications of ACM, 21, 7, July 1978, pp. 558-565.

[21] G. Morgan and S.K. Shrivastava, “Implementing Flexible Object Group Invocation in Networked Systems”, International Conference on Dependable Systems and Networks, New York, June, 2000.