

Integrating Group Communication with Transactions for Implementing Persistent Replicated Objects

M.C. Little and S.K. Shrivastava

Department of Computing Science
University of Newcastle, Newcastle upon Tyne, NE1 7RU, England

Abstract

A widely used computational model for constructing fault-tolerant distributed applications employs atomic transactions for controlling operations on persistent objects. There has been considerable work on data replication techniques for increasing the availability of persistent data that is manipulated under the control of transactions. Process groups with ordered group communications (process groups for short) has also emerged as a model for building available distributed applications. High service availability can be achieved by replicating the service state on multiple processes managed by a group communication infrastructure. These two models are often seen as rivals. This paper examines whether a distributed transaction system can profit from process groups for supporting replication of objects. A general model of distributed objects is used to investigate how objects can be replicated for availability using a system that supports transactions (but no process groups) and a system that supports process groups (but no transactions). A comparative evaluation reveals how a distributed transaction system can exploit group communications for obtaining a flexible approach to supporting replication of objects.

Key words: distributed systems, fault-tolerance, persistent objects, group communication, atomic transactions, replication

1. Introduction

A widely used computational model for constructing fault-tolerant distributed applications employs atomic transactions (atomic actions) for controlling operations on persistent (long-lived) objects. There has been considerable amount of work done on data replication techniques for increasing the availability of persistent data that is manipulated under the control of transactions [1]. The process group with ordered group communications (process groups for short) has also emerged as a model for building available distributed applications [e.g., 2-8]. High service availability can be achieved by replicating the service state on multiple server processes that are managed by an underlying group communication infrastructure.

These two models are often seen as rivals; for example a recent paper suggested that transactional systems are a better alternative to process groups [9], a claim hotly denied by the supporters of the process group approach [10]. There have been attempts to 'unify' the two models, the main thrust of the work being to enrich group communication with some transactional flavour [11]. This paper also explores the role of the two models in building fault-tolerant distributed applications. We do it in very focussed manner, dealing with just the replication of persistent objects, exploring the overall system design issues that have not been addressed by previous studies.

Our approach to exploring ways of using transactions and group communications together is quite pragmatic: we actually investigate how transaction systems can make use of process groups, rather than the other way round as hinted in [11]. We envisage that future distributed applications will increasingly rely on the so called 'middleware services' for support for naming, concurrency control, event management, persistence etc. Further, CORBA, the industry backed distributed object architecture has adopted transactions as the application structuring paradigm for manipulating long-lived objects, and transaction services are already available on CORBA platforms alongside other services mentioned before. In addition, given the availability of CORBA group communication services (such as [12, 13]),

we have the possibility of supporting object replication in more than one way. For applications that are programmed using transactions, it is then worth investigating if any benefits can be gained by exploiting group communication for replicating transactional objects. We show that this is indeed the case. Our ideas on how transaction systems can make use of group communication for supporting high availability distributed applications are therefore directly applicable to CORBA and similar architectures. An earlier version of this paper has appeared in [14].

We begin by presenting a model of distributed object system without replication. We do not claim that our model is ‘universal’, but that it is representative of a sufficiently wide class of real systems. We then investigate how objects can be replicated for availability in two types of systems: (i) a system that supports transactions but no process groups; object replication approaches used here will be termed transaction based (or *TR-based* for short); and (ii) a system that supports process groups but no transactions; object replication approaches used here will be termed group communication based (or *GC-based* for short). We then evaluate the two approaches. This evaluation enables us to understand how a distributed transaction system can exploit group communications for obtaining a flexible approach to supporting replication of objects. We do not assume any special hardware support for replication (e.g., mirrored disks, tightly coupled primary-backup processor clusters etc.), but consider just a distributed collection of processors with secondary storage.

2. Models

2.1. Assumptions

It will be assumed that the hardware components of the system are computers (nodes), connected by a communication subsystem. A node is assumed to work either as specified or simply to stop working (crash). After a crash, a node is repaired within a finite amount of time and made active again. A node may have both stable (crash-proof) and non-stable (volatile) storage or just non-stable storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs; any data stored on stable storage remains unaffected by a crash.

We model the communication environment as either *asynchronous* or *synchronous*. In an asynchronous environment connected and correctly functioning processes are capable of communicating in *bounded but unknown* time (i.e., the time interval between the sending of a message and its reception by the receiver cannot be estimated accurately) whereas in a synchronous environment connected and correctly functioning processes are capable of communicating with each other in *bounded and known time*. A network will be said to be *partitionable*, if correctly functioning processes are not able to communicate (e.g., due to physical breakdown such as a crash of a gateway node and/or network congestion). In an asynchronous environment, even without partitions, timeouts and network level ‘ping’ mechanisms cannot act as an accurate indication of process failures (they can only be used for *suspecting* failures); whereas in a synchronous environment, assuming no partitions, judiciously chosen timeouts together with network level ‘ping’ mechanisms can act as an accurate indication of process failures. In this paper we will make the assumption of asynchronous, partitionable communication environment; however, we assume that a partition in a network is eventually repaired.

We will consider the case of *strong consistency* which requires that the states of all replicas that are regarded as *available* be mutually consistent (so the persistent states of all available replicas are required to be identical). Object replicas must therefore be managed through appropriate replica-consistency protocols to ensure strong consistency.

2.2 Distributed non-replicated Objects

An object is an instance of some class. The class defines the set of *instance variables* each object will contain and the *operations* or *methods* that determine the externally visible behaviour of the object. The operations of an object have access to the instance variables and can thus modify the internal state of that object. An operation invocation upon a remote object is performed via a remote procedure call (RPC).

We assume that objects are *persistent*, by which we simply mean that they are long-lived entities. Objects are assigned unique identifiers (UIDs) for naming them. We further assume that some application specific naming service can map an application level object name (a string) to the

corresponding UID. We do not discuss such a naming scheme further here, and assume that an application can always obtain the UIDs of objects it is manipulating.

A persistent object not in use is assumed to be held in a *passive* state with its state residing in an object store (a stable object repository) and *activated* on demand (i.e., when an invocation is made) by loading its state from the object store to the volatile store, and associating an *object server* process for receiving RPC invocations. To be able to access a persistent object, an application program must be able to obtain information about the object's location. We assume that the application program can request this information from a *binding service* by presenting it with the UID of the object. Once the application program (client) has obtained the location of the object it can direct its invocations to that node. It will be the responsibility of that node to activate the object (if the object was in a passive state). Thus, a number of system services are required to support distributed computations structured out of objects. We enumerate them below:

- *RPC service*: provide an object invocation facility;
- *Object Storage service*: provide a stable storage repository for object states; a state can be retrieved by presenting the service with the UID of the object.
- *Binding service*: provides a mapping from a given UID to the necessary information required for binding to the object (this information is: host name of the object server, host name of the object store, see below). The binding service is assumed to run at a well known address.

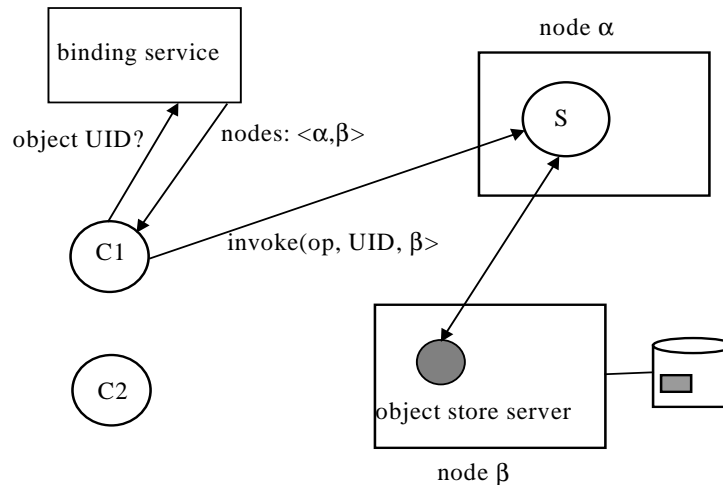


Figure 1: Object binding.

We assume that for each persistent object there is a node (say α) which, if functioning, is capable of running a server for that object (in effect, this would require that the node has access to the executable code for the object's methods). If α receives an invocation from the client, and the object is passive at α , then the object needs to be activated before the invocation can be performed; this requires allocating a server and if necessary loading the executable code for the object and then fetching the object state from some object store. We will assume that there is a node (say β) whose object store contains the state of the object; we do not require that α be the same as β . Thus, we assume that the binding service maps a given UID to location related information, namely the pair $\langle \alpha, \beta \rangle$. Fig. 1 illustrates the steps involved for the case of an object that is passive: client C1 presents the UID of an object to the binding service; sends the invocation (to perform operation 'op') to node α . This node allocates a server S and gets the state from the object store at β , before S can perform the operation. During the termination of an application, the new states of objects are migrated to their object stores.

We assume that objects can be shared between clients, and an object server is responsible for enforcing concurrency control on its objects (e.g., shared read but exclusive write). In the above example, if some client C2 invokes the same object, then that invocation will be sent to α who will direct the invocation to the activated copy being managed by S.

Remark 1: In the object model presented above, for a given set of objects, the binding service requirement is simple as it needs to provide read only access to location information. We assume that a copy of the information is also held on stable store (see below).

Remark 2: The model goes a long way towards supporting the ability to recover from *total crashes*: even if all of the nodes crash, the system can eventually bootstrap itself with information held on stable stores remaining consistent, provided partitions eventually heal and crashed nodes eventually recover.

Some inconsistency can creep in if an application is unable to terminate cleanly due to crashes and not all the servers have managed to update stable stores (transactions are used precisely to avoid this situation).

Remark 3: An invocation from a client could give rise to nested invocations, as the methods of an object can contain calls on other objects. Thus, an activated object could end up activating other objects. For the sake of simplicity, we will not consider nesting in our subsequent discussions.

Remark 4: Let us hint at an interesting extension to the model that complicates matters! Assume that there are several nodes that are capable of running an object server for a given object, and the binding service has a list of such nodes. To guarantee consistent object sharing, we must ensure that if an object is active, then it is active at only one object server. A possible activation policy could be that for a passive object, the binding service returns the list, enabling a client to select the most appropriate node. However, if the object is active, then the service returns the identity of the node where the object has been activated. Clearly, object binding has become complicated, requiring interaction between object servers and the binding service; this and related issues will need to be addressed when we consider replication of objects.

2.3. Object Replication

In the scheme shown in fig. 1, an object can potentially become unavailable to the client if any of the communication paths get disrupted due to partitions or any of the nodes hosting α , β , or the binding service crash. We will assume that we do not have direct control over communication resources, so we will investigate exploitation of redundancy in the form of multiple machines capable of running servers for an object and multiple machines capable of storing the state of an object.

The binding service needs to maintain sufficient information about replicas to enable clients to be bound to available replicas. We assume that for every persistent object, the binding service maintains two sets of node related data: (i) Sv_A : for an object A , this set contains the names of nodes each capable of running a server for A ; and, (ii) St_A : this set contains the names of nodes whose object stores contain states of A . The way this information is maintained by the binding service will depend on which replication approach is in use (TR-based or GC-based). We now consider two ways of activating an object (assume that A is currently passive, i.e., not in use).

(i) *Single copy activated:* This represents the case where only the state of an object is replicated (see fig. 2). Activating A will consist of creating a server at the node which is a member of Sv_A (say α) and loading the state from any of the nodes in St_A . If the server crashes, then a new one is created at some other node in Sv_A .

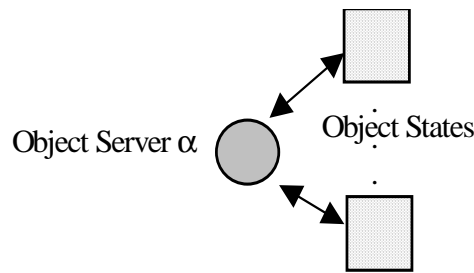


Figure 2: Object state replication.

(ii) *Multiple copies activated:* Activating A consist of creating servers at one or more nodes listed in Sv_A (let Sv_A' , where $Sv_A' \subseteq Sv_A$, be the set of such nodes), and loading the state of the object from the node in St_A .

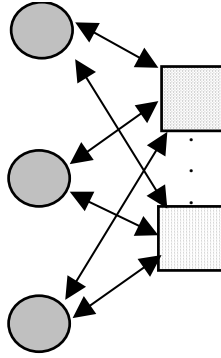


Figure 3: Server and state replication.

A process group, such as the group of servers, Sv_A , must be managed as a single entity. There are two well-known group management policies: (i) *Active replication*: In active replication, all the functioning members of the group perform processing [15]. Of course, it is necessary that all the functioning members receive client invocations in the same order and that the computation performed by each replica be deterministic. (ii) *Primary-backup replication*: Here, only one replica, the *primary* (coordinator), carries out processing. The primary regularly checkpoints its state to the remaining replicas (cohorts). If the failure of the primary is detected, then the cohorts elect one of them as the new primary to continue processing. In both the cases, the initial membership of the group, (e.g., $|Sv_A|$) is $2K+1$, $K \geq 1$. If partitions cause the group to be broken into sub-groups, then only the *majority* sub-group with at least $K+1$ members (if any) remains functioning while other sub-groups delete themselves. Clients with access to the majority group can continue to make forward progress.

The binding service must be replicated $2K+1$ times, and only the clients with access to the majority of name service replicas ($K+1$) can make forward progress.

2.4. Atomic transactions

An atomic transaction guarantees that, despite failures, either all of the work conducted within its scope will be performed or it will all be undone. Atomic transactions have the well known *ACID* properties of *Atomicity*, *Consistency*, *Isolation* and *Durability*.

Atomicity property ensures that a computation will either be terminated normally (*committed*), producing the intended results (that is, intended state changes to the objects involved) or *aborted* producing no results (no state changes to the objects). This atomicity property may be obtained by the appropriate use of backward error recovery, which can be invoked whenever a failure occurs that cannot be masked. Typical failures causing a computation to be aborted include node crashes and communication failures such as the continued loss of messages.

It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent (class specific) state changes to the object. Transactions then ensure that only consistent state changes to objects take place despite concurrent access and any failures. This consistency property goes hand in hand with the isolation property that ensures freedom from interference: each transaction accesses shared objects without interfering with other transactions. In other words, the effect of concurrently executing transactions can be shown to be equivalent to some serial order of execution. Some form of concurrency control policy, such as that enforced by two-phase locking [1], is required to ensure isolation and consistency properties of transactions.

It is reasonable to assume that once a transaction terminates normally, the results produced are not destroyed by subsequent node crashes. This is ensured by the durability property, which requires that any *committed* state changes (i.e., new states of objects modified in the transaction) are recorded on stable (crash-proof) storage. A (two phase) commit protocol is required during the termination of a transaction to ensure that either all the objects updated within the transaction have their new states recorded on stable storage, or, if the transaction aborts, no updates get recorded. Atomic transactions can also be nested; the effects of a nested transaction are provisional upon the commit/abort of the outermost (*top-level*) atomic transaction.

2.5. Process groups

A *group* is defined as a collection of distributed entities (objects, processes) in which a member entity can communicate with other members by multicasting to the full membership of the group. A desirable property is that a given multicast be *atomic*: either all or none of the functioning members are delivered the message. An additional property of interest is guaranteeing *total order*: all the functioning members are delivered messages in identical order. As an example, these properties are ideal for replicated data management: each member manages a copy of data, and given atomic delivery and total order, it can be ensured that copies of data do not diverge. However, as we discuss below, achieving these properties in the presence of failures is not simple.

Suppose a multicast is interrupted due to the crash of the member making the multicast; this can result in some members not receiving the message. Member crashes should ideally be handled by a fault tolerant protocol in the following manner: when a member does crash, all functioning members must promptly observe that crash event and agree on the order of that event relative to other events in the system. In an asynchronous environment this is impossible to achieve: when members are prone to failures, it is impossible to guarantee that all functioning members will reach agreement in finite time [16]. This impossibility stems from the inability of a process to distinguish slow members from crashed ones. Asynchronous protocols can circumvent this impossibility result by permitting processes to *suspect* process crashes and to reach agreement only among those processes which they do not suspect to have crashed [17].

A group therefore needs the services of a *membership service* that executes an agreement protocol to ensure that functioning members within any given group will have identical views about the group membership. The membership service also ensures that the sequence of views installed by any two functioning member processes of a group that do not suspect each other are identical. Network partitions can lead to the group of processes to partition themselves into several subgroups of mutually unsuspecting processes. In a *primary partition membership service*, members of one subgroup (primary subgroup) continue to function while members of the other subgroups are deemed faulty. A normal way of deciding on a primary is to select the subgroup with the majority of the members of the original group. A *partitionable membership service* on the other hand permits the subgroups of processes to coexist and later merge (see [18] for discussion on partitionable membership service). We will assume a primary partition membership service.

The failure model of process groups differs from that used in transaction model in one important way: when a member of a group fails by crashing, it loses all its state (i.e., members have no stable states).

3. TR-based and GC-based Replication

For comparative evaluation, we will now investigate how replication schemes of broadly similar functionality can be implemented in TR (a system that supports transactions but no process groups) and GC (a system that supports process groups but no transactions) and then compare the two. We will use single copy activation scheme with state replication for the TR-based approach (fig. 2). Since only a single server is activated (a backup is created only if the failure of the original server is suspected), there is no need to rely on group communication for managing the server. We will use primary-backup replication scheme (both server and state replication, fig. 3) for the GC-based approach. Here the primary server does the processing and a backup takes over if the primary fails, so the scheme resembles the single copy activation scheme to be used in TR.

3.1. TR-based Replication

As stated before, the replication scheme to be considered here requires only a single copy of the object to be activated. As regards to state replication, replicated data management techniques that go hand in hand with transaction commit processing [1] provide an integrated solution to object state management. The binding service plays a central role here, by maintaining up-to-date Sv and St related 'group view' information. We assume that it is possible to update Sv and St related information for objects maintained by the naming and binding service. For example, it should be possible to exclude the name of a node currently in St_A if the node is found not to contain the latest (committed) state of the object (say the node has crashed). We assume that the binding service itself is built out of one or more persistent objects, so the above state changes are (naturally) performed under the control of transactions.

For now we will assume that the service is available, and discuss later the issues concerning its replication.

Here is an overview, for the simple case of a client accessing some object A: assume object A is passive and this is recorded in the binding service. The client's binding request to A then returns the sets Sv_A and St_A enabling a client to select the most appropriate server node. The client directs its invocation to some such node in Sv_A (say α); α allocates a server and loads the state from any node in St_A . (This has been discussed already for the case of non-replicated objects, see fig. 1). The identity of the activated server (α) must be registered with the binding service, so that subsequent binding requests to A return α in place of Sv_A . At commit time, an attempt is made to copy the updated state of the object at α to the object stores of all the nodes in St_A . To ensure that St_A contains the names of only those nodes with mutually consistent states of A, the names of all those nodes for which the copy operation failed must be removed from St_A . A transaction using A will abort if α crashes (or is suspected to have crashed) during execution. Restarting the transaction could cause A to be activated at some node α' in Sv_A . These aspects are discussed further below with the help of fig.4.

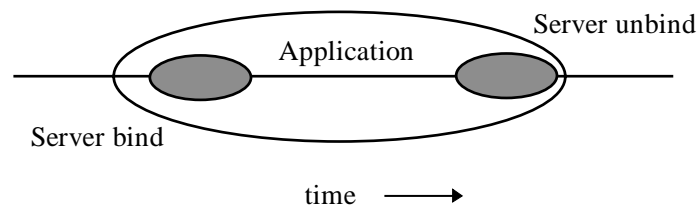


Figure 4: Binding and unbinding performed using nested transactions.

Binding: Prior to contacting the binding service the client (say C_1) begins an application level transaction. To guarantee consistency in the presence of concurrent bind requests for the same object, binding has to be an atomic operation. This is achieved by C_1 performing the binding related operations mentioned earlier from within a transaction (this is the nested transaction 'server bind' in fig. 4). So, if the object is passive, and C_1 manages to get a server at α , then the binding service update to record that the object is active at α is performed as a part of this nested transaction; at the same time, a 'use count' for this object maintained by the service is set to one (indicating one user). A subsequent bind request (say from C_2) to this activated object will return the address of node α . C_2 , as a part of its bind transaction, will try to connect to the server at α , and if this succeeds, the 'use count' is incremented and the bind transaction is terminated. If C_2 is unable to connect, then it suspects a crash of the server and is free to select some other node from Sv_A to activate the object, and the entry for α is replaced by the identity of the new node (with a 'use count' of one). Note that if C_2 's suspicion is incorrect and the server at α is functioning (say there is a temporary partition) then there is an inconsistency (the object has been activated at more than one place) that must be resolved. This is done at commit time as we will describe shortly. If the application level transaction aborts after binding, any binding related updates to the binding service are undone.

Unbind: Unbinding is also performed as a transaction (see fig. 4) executed during commit processing of the application level transaction. The client contacts the binding service to check for any binding inconsistency: if this is detected, then the client aborts. Otherwise, the client decreases the 'use count'. If the use count reaches zero, then the server can be told to passivate the object and the binding service can treat the object as passive again. Commit processing also involves updating all the states held at nodes $\in St_A$. The names of any nodes where these updates fail are removed from the set St_A kept at the binding service. This ensures that the set always maintains the names of node with the latest state of the object.

The binding service itself must be available at all times. This requirement can be met by replicating the service on $2K+1$ distinct nodes. To eliminate the requirement of the service itself requiring a binding system for replicated objects, three simplifying restrictions can be made: (i) the nodes storing Sv and St related data objects also run servers for them; (ii) every client is expected to 'know' the locations of these nodes (that is the binding service addresses are well known); and (iii) client updates to binding data are allowed only if a majority of the replicas can be updated (else the client action aborts). This also means that a client has to contact a majority of the binding servers to be sure to obtain the most recent binding information. This is the well known quorum consensus replication approach [1, 19]. Forward progress is not possible if the majority assumption cannot be met, and application level processing must block till enough nodes recover and/or partitions heal.

In summary: the binding service is used transactionally to activate, bind, unbind and passivate objects in a consistent manner; the set of nodes in S_v for an object provide redundancy for server creation and the set of nodes in S_t provide redundancy in storage. So long as a client has access to the binding service, an object A remains available provided the client has access to at least one node in S_{v_A} and that node has access to at least one node in S_{t_A} .

A final observation: the scheme illustrated by fig. 4 has a shortcoming in that the binding information concerning an object remains locked for the entire duration of the application level transaction (because the service access is performed as nested transactions). If this is considered a concurrency control bottleneck, then an alternative concurrency control scheme is possible whereby the bind and unbind transactions are run as ‘nested top level’ transactions; these details are discussed elsewhere [20, 21].

3.2. GC-based Replication

We will first describe the basics of the primary-backup passive replication scheme using a process group and then discuss how such groups can be used in our object model. Since process groups explicitly deal with message passing, we will describe and illustrate the scheme in terms of messages; the treatment given here is based on [22].

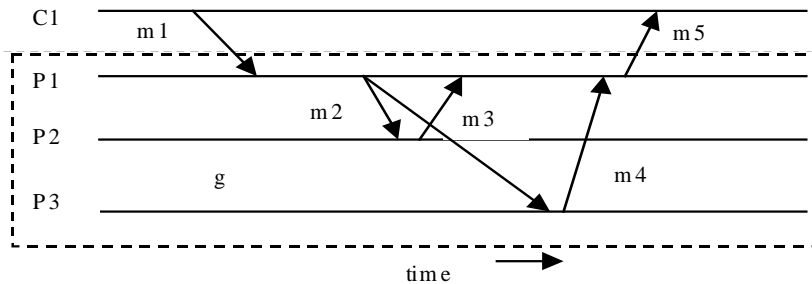


Figure 5: Primary-backup passive replication.

We assume that clients know the name of the primary within the replica group (P_1 in the replica group g with three members, P_2 and P_3 are backups, fig. 5). The client sends its request message (m_1) to the primary. The primary carries out the work, updates its state and then multicasts a message (m_2) containing state changes performed. P_2 and P_3 update their states and send acknowledgements to the primary (messages m_3 , m_4). The primary then sends the response message (m_5).

If the primary is suspected to have failed, then the membership service of the group will reformulate the group with the primary removed. Any deterministic algorithm can be used by the members for deciding the next primary. The atomic delivery property of multicasts ensures that backups remain in ‘step’. For example, assume the primary crashes during the multicast of m_2 ; if P_2 is delivered the message, then P_3 will also be delivered m_2 . Continuing with this scenario, the client waiting for the response will eventually timeout. Assume it somehow finds out the name of the next primary and resends its request. If the primary has received m_2 , then it sends the response message (without redoing the work).

Returning to our object model: referring to fig. 3, we can see that two groups to manage a replicated object are suggested: a server group and a state group (let us denote them as G_s and G_t respectively). However, since it is the server group that performs the computation, a simplification is possible and the need for G_t can be dispensed. Since a process group manages its membership view information, the binding service is no longer required to maintain current membership information for activated groups. The binding service must however maintain the information about whether a group has been activated or not, and for an activated group, who the primary is. Further, the binding service maintains the list S_t that must only contain the names of nodes with the latest states of persistent objects. Here are the possible steps involved in activating and using an object (A).

The client contacts the binding service, and receives the sets S_{v_A} , and S_{t_A} . The client constructs a list of $2K+1$ nodes in S_{v_A} , with one listed as primary and requests one of them (say α) to create a group; α creates the group (G_s), and registers the name of the primary with the binding service. The primary of G_s acts as a client to obtain the persistent state of the object from any node $\in S_{t_A}$. It then multicasts the state to other members of G_s . The client sends requests to the primary of G_s (these are normal object invocations, and handled as discussed earlier with respect to fig. 5). If the group elects a

new primary, then the newly elected primary registers its identity with the binding service, deleting the name of the old one.

When a client finishes using A, it explicitly sends a 'disconnect' invocation to the primary of Gs. The group must now make the object state persistent by updating the object stores named by St_A (this corresponds to the commit operation of the transaction system). There are two steps involved here: (i) the primary updates the object stores named by St_A , and then makes a list of nodes where the updates have not succeeded; and (ii) the primary removes the names of these nodes from the St_A set maintained by the binding service.

When a group determines that it has no more users (this is possible to calculate provided connected clients survive and are able to communicate with the group), the primary of the group can unregister its name from the binding service, and the group can be destroyed.

In this scheme, forward progress is possible despite partitions and node failures provided a primary subgroup of Gs survives and remains connected with the client; further, this group must be able to access the binding service and at least one object store in St_A . If this is not possible then consistency cannot be guaranteed. For example, suppose that during step (ii) above all the group members crash (call it a *group failure*), as a result, the St_A list of the binding service is not updated as required. In that case the list cannot be guaranteed to contain the names of nodes with the latest states of persistent objects. This problem can be handled by maintaining, in a careful manner, group configuration information on stable store, such that the primary group configuration can be reconstructed after a group failure; this way the group can resume the interrupted operation (papers [23, 24] discuss how to enhance process groups with the capability of recovering from group failures). One problem still remains: as there is no support for backward recovery, if a client crashes, then there is no automatic way of restoring any states of objects.

The binding service itself can be made available quite simply: we assume that a process group with $2K+1$ members with well known addresses is created at start up time and remains in operation.

4. Integrated Replication Schemes

4.1. Comparative evaluation

We evaluate the two schemes discussed in the previous section by considering their effectiveness in meeting requirements of the ability to recover from total crashes (even if all the nodes crash, the system can eventually bootstrap itself with information held on stable stores remaining consistent, provided partitions eventually heal and crashed nodes eventually recover) and availability (objects remain accessible even if partitions have not yet healed and nodes have not yet recovered).

Replication using transactions: Transactions are ideally suited to meeting the requirement of ability to recover from total crashes (not surprising as they have been designed specifically for the very same purpose). Transactions ensure that shared information held on stable store is manipulated consistently despite failures. This is a powerful facility that can be utilised for building those facilities that are not directly supported by transactions. Transactions do not provide direct support for maintaining replica group related information, so we have to build a subsystem (transactional binding service in our case) that does exactly that. The mechanism for making the binding service available is rather heavyweight, requiring clients to read from a majority to be sure of obtaining the latest copy of the information.

Transactions can be used in a limited (but quite effective) manner for supporting availability. Their way of dealing with server failures (suspected or real) is to push the problem to application level transactions bound to the server; an affected transaction can abort and rebind to a new server, so that forward progress can be made. Transactions do not provide any direct support of agreeing over failure suspicions, so any consistency problem caused has to be pushed again to applications. In the scheme discussed here, inconsistent server bindings are checked at commit time and offending transactions aborted.

Replication using group communications: In contrast to transactions, process groups provide a very elegant way of managing replicas in the most general case of partitionable environments where functioning processes can be wrongly suspected to have failed. So long as the formation of a primary server subgroup is possible, server failures can be masked from clients.

Coordinated backward recovery is not an integral feature of process groups, so there is no direct way of dealing with client failures (real or suspected). Process groups cannot directly support the ability to recover from total crashes. As indicated earlier, to do this would require enhancing process groups with

support for maintaining views on stable store, such that groups can be reconstructed even after group failures.

In summary, transactions are good at dealing with total crashes, but extra effort is required for maintaining replica group related information and ensuring that inconsistent bindings do not occur due to servers wrongly suspected to have crashed. Process groups are good at managing replica server groups, but need additional support for dealing with total crashes and client failures.

4.2. Using Transaction systems with Process Groups

In the light of the analysis presented in the previous subsection, we discuss how the two system models can be used together. For the reasons given in the introduction, we will investigate how transaction systems can make use of process groups, and what practical benefits can be gained. We assume that transactions are used directly by application programmers and process groups are essentially hidden from application programmers.

Transactions provide a very effective solution to dealing with all the exceptions that cannot be masked: abort and retry. We have used this approach here for dealing with server failures and recovering from inconsistent bindings to servers. Since applications use transactions any way, why not exploit transactions for handling problems of replication? This is the application of the well known 'end to end argument in system design' [25]: see if application level mechanisms can be used for handling lower level problems. However, the 'end to end argument' does not preclude the use of specific lower level mechanisms for masking exceptions, if these mechanisms enhance efficiency. A transaction system can profitably make use of process groups for supporting replication in at least three ways:

(i) *Supporting binding service replication*: A process group with $2K+1$ members can be used to maintain the binding service. Clients now only need to contact the primary, and the need for reading from $K+1$ replicas and voting is eliminated.

(ii) *Fast switch over from a failed server*: A process group can provide a faster way of switching to a backup in the event of the failure of the primary. As described previously, in a transaction system, this would entail the transaction aborting and then rebinding. Commercial transaction systems in fact do make use of primary-backup process groups in a very specialised manner: a primary-secondary pair is used in a non-partitionable communication environment (so carefully chosen timeouts can be relied upon for failure detection). Process groups can provide this functionality in the general setting of $2K+1$ replicas in a partitionable communication environment.

(iii) *Supporting Active Replication*: Active replication is often the preferred choice for supporting high availability of services where masking of replica failures with minimum time penalty is considered highly desirable. Active replication requires that all the functioning members of a server group receive client invocations in the same order. A process group can provide this facility.

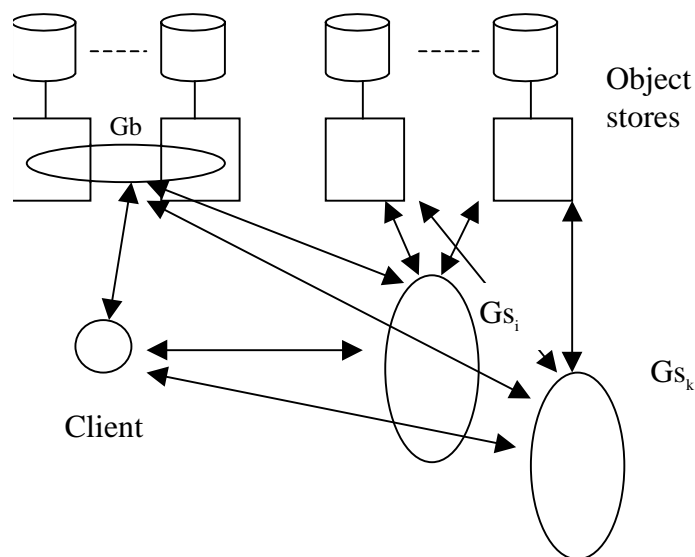


Figure 6: Use of process groups.

A general scheme for the exploitation of process groups is depicted in fig. 6; here Gb is the group for managing the transactional binding service and Gs (i..k) are the server groups that are created on

demand as discussed earlier. We have designed a toolkit for CORBA with the above architecture in mind [26]. The default implementation supports pure transaction approach discussed in section 3. Our toolkit has all the necessary hooks for exploiting the services of a process group, such as a CORBA group service [12, 13], enabling a client (more precisely, a client proxy) to invoke a group using group specific multicasts if required.

5. Concluding Remarks

We have carried out this investigation to understand how transactions and group communications (process groups) can be exploited to construct high availability distributed applications. In particular, we have investigated how a transaction system can benefit from an underlying group communication system. As we show, although transactions can be used for supporting replication of persistent objects without the need of process groups, if the underlying infrastructure does support process groups, then these can be exploited effectively for binding service replication, providing faster switch over to backups and for supporting active replication. A recent paper has investigated the use of group communication in a fully replicated database system [27]. Although the distributed system model used in that paper is different to what is assumed here (the unit of replication is the entire database, so no binding service is required), the study reinforces the observation made here that transaction systems can benefit from group communication services.

Acknowledgements

This work has been supported in part by grants from UK Engineering and Physical Sciences Research Council grant no GR/L73708, British Telecom and ESPRIT working group BROADCAST (project No. 22455).

References

- [1] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
- [2] R. Guerraoui, P. Felber, B. Garbinato and K. Mazouni, "System support for object groups", Proceedings of the ACM Conference on Object Oriented Programming Systems, Languages and Applications, OOPSLA 98.
- [3] K. Birman, "The process group approach to reliable computing", CACM, 36, 12, pp. 37-53, December 1993.
- [4] 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.
- [5] P. Narasimhan et al, "Replica consistency of CORBA objects in partitionable distributed systems", Distributed Systems Engineering Journal, No. 4, 1997, pp. 139-150.
- [6] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication", CACM, 39 (4), April 1996, pp. 64-70.
- [7] 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.
- [8] F. Cristian, "Synchronous and asynchronous group communication", CACM, 39 (4), April 1996, pp. 88-97.
- [9] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication", Proc. of 14th ACM Symp. on Operating Systems Principles, Operating Systems Review, 27 (5), pp. 44-57, December 1993.
- [10] Rebuttals from Cornell, Operating Systems Review, 28 (1), January 1994.
- [11] A. Schiper and M. Raynal, "From group communication to transactions in distributed systems", CACM, 39(4), April 1996.
- [12] 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.
- [13] G. Morgan, S. K. Shrivastava, P. D. Ezhilchelvan and M.C. Little, "Design and Implementation of a CORBA Fault-tolerant Object Group Service", Proceedings of 2nd IFIP International Working Conference on Distributed Applications and Interoperable Systems, DAIS'99, Helsinki, June 1999.
- [14] M.C. Little and S K Shrivastava, "Understanding the Role of Atomic Transactions and Group Communications in Implementing Persistent Replicated Objects", Advances in persistent Object Systems, ed. R. Morrison, M. Jordan and M. Atkinson, Morgan Kaufman, ISBN: 1 55860 585 1, 1999, pp. 17-28.
- [15] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.
- [16] M. Fischer, N. Lynch, and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", J. ACM, 32, April 1985, pp 374-382.
- [17] T.D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems", J. ACM, 43(2), 1996, pp. 225-267.
- [18] O. Babaoglu, R. Davoli and A. Montresor, "Group communication in partitionable systems: specification and algorithms", Chapter 3 of this book.
- [19] D. K. Gifford, "Weighted Voting for Replicated Data", 7th Symposium on Operating System Principles, December 1979.

- [20] M.C. Little, D. McCue and S.K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", ICDCS-13, Pittsburgh, pp. 491-498, May 1993.
- [21] M. C. Little and S. K. Shrivastava, "Object replication in Arjuna", Broadcast Project deliverable report, Vol.2, October 1994, Dept. of Computing Science, University of Newcastle upon Tyne, UK (<http://www.newcastle.research.ec.org/broadcast/trs/year2-deliv.html#a2.1>)
- [22] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance", IEEE Computer, April 1997, pp. 68-74.
- [23] I. Keider and D. Dolev, "Efficient message ordering in dynamic networks", ACM Symp. on Principles of Distributed Computing, PODC, May 1996.
- [24] P Ezhilchelvan and S K Shrivastava "Enhancing Replica Management Services to Tolerate Group Failures", The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing, ISORC'99, Saint-Malo, May 99, pp. 263-268. (also, Chapter 4 of this book).
- [25] J.H. Saltzer, D.P. Reed and D.D. Clark, "End to end argument in system design", ACM Trans. on Comp. Syst., 2(4), Nov. 1984, pp. 277-288.
- [26] M.C. Little and S K Shrivastava, "Implementing high availability CORBA applications with Java", IEEE Workshop on Internet Applications, WIAPP'99, San Jose, July 1999.
- [27] B. Kemme and G. Alonso, "A suite of replication protocols based on group communication primitives", Proc. 18th IEEE Intl. Conf. on Distributed Computing Systems, ICDCS'98, Amsterdam, May 1998, pp. 156-163.