Understanding the Role of Atomic Transactions and Group Communications in Implementing Persistent Replicated Objects

M.C. Little and S.K. Shrivastava

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

To appear in the Proceedings of the 8th International Workshop on Persistent Object Systems: Design Implementation and Use

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 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 explores the role of these models in building fault-tolerant distributed applications. The paper develops a general model of distributed persistent objects and investigates how such objects can be replicated for availability using a system that supports only: (i) transactions; and (ii) process groups. A comparative evaluation shows how the two models can be used together to provide a more flexible approach to supporting high availability applications.

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, 2]. The process group with ordered group communications (process groups for short) has also emerged as a model for building available distributed applications [e.g., 3, 4, 5, 6, 7]. 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 [8], a claim hotly denied by the supporters of the process group approach [9]. There have been attempts to 'unify' the two models, the main thrust of the work being to enrich group communication with some transactional flavour [10]. This paper also explores the role of the two models in building fault-tolerant distributed applications. But 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. We 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. Our investigation shows how the features from the two systems can be used together to provide a more flexible approach to supporting high availability distributed applications. The system models that we assume here are essentially the same as those used in [8]. 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.

The main problem to be overcome in such an investigation is that there does not appear to be any easy way of constructing exact functionally equivalent models of the two systems referred to above. Transaction systems routinely make use of stable (non-volatile) storage, so node crashes do not destroy all of the system state; whereas process groups do not normally maintain stable states. Transaction systems routinely make use of backward recovery (transaction aborts) for restoring application state to cope with lower level as well as application level failures that cannot be masked; however, backward recovery is not an integral feature of process groups. Process groups automatically maintain mutually consistent membership views, a feature that is absent from transaction systems. Therefore the approach we have taken is quite pragmatic. We do not consider object replication just in isolation, but also take into account the overall design issues of building systems that require availability. We do this by developing a model of distributed object system without replication and then investigate how replication can be incorporated. We do not claim that our model is ‘universal’, but that it is representative of a sufficiently wide class of real systems to make our approach worthwhile. Thus we are able to show how passive replication scheme of broadly similar functionality can be implemented in either system, and then compare the two schemes.

Our approach to exploring ways of using transactions and group communications together is also pragmatic: we actually investigate how transaction systems can make use of process groups, rather than the other way round as hinted in [10]. 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, industry backed distributed object architectures (e.g., CORBA) have firmly 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. If in addition we assume the availability of a group communication service, then 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. 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. Although the recommendations that we come up may well (upon hindsight) appear quite straightforward, we believe that the reader will find our analysis illuminating.

2. Models

2.1. Failure 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 message transmission times cannot be estimated accurately, and the underlying network may well get partitioned (e.g., due to a crash of a gateway node and/or network congestion) preventing functioning processes from communicating with each other; in such an environment, timeouts and network level ‘ping’ mechanisms cannot act as an accurate indication of node failures (they can only be used for *suspecting* failures). We will call such a communication environment *partitionable*. In a synchronous communication environment, functioning nodes are capable of communicating with each other, and judiciously chosen timeouts together with network level ‘ping’ mechanisms can act as an accurate indication of node failures. We will call such a communication environment *non-partitionable*. In this paper, unless stated explicitly, we will make the more general assumption of partitionable communication environment; however, we assume that a partition in a network is eventually repaired.

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.

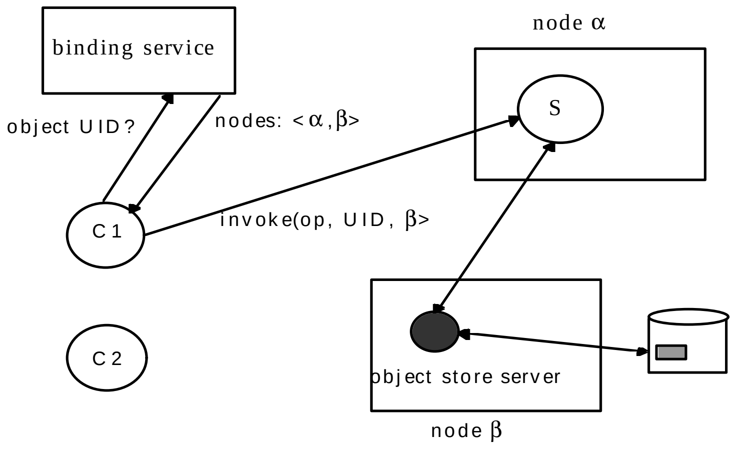

Fig. 1: Object binding

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 <α, β>. 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 failures*: 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 model with replication

In the object model discussed in the previous subsection, a persistent object can become temporarily *unavailable* due to failures such as a crash of the object server, or network partition preventing communications between clients and the server. The *service availability* of an object can be increased by replicating it on distinct nodes such that the object remains accessible even if partitions have not (yet) healed and nodes have not (yet) recovered. We will assume that the capability of guarding against total failures is not to be compromised and the system should be able to bootstrap itself.

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. To tolerate K replica failures, in a non-partitionable network, it is necessary to maintain at least K+1 replicas of an object, whereas in a partitionable network, a minimum of 2K+1 replicas are necessary to maintain availability in the partition with access to the majority of the replicas (the object becomes unavailable in all of the other partitions) [2].

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 ∈ Sv_A (say α) and loading the state from any of the nodes ∈ St_A. If the server crashes, then a new one is created at some other node ∈ Sv_A.

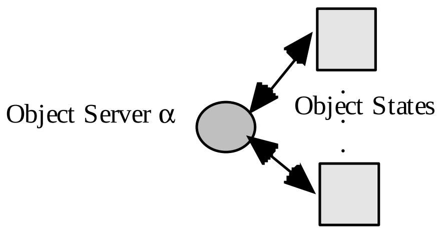

Fig. 2: Object state replication

(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 ∈ St_A. Each server is free to load the state of the object from any of the nodes ∈ St_A.

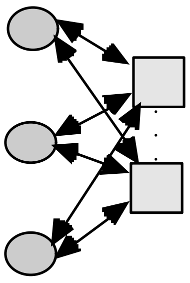

Fig. 3: Server and state replication

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 [11]. 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.

We want to 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 if this server fails), 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. Our investigation can certainly be extended to include active replication and we discuss this briefly towards the end of this paper.

3. Replication using transactions

As stated before, the replication scheme to be considered here requires only a single copy of the object to be activated. 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 ∈ 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. The objects providing the binding service themselves can be replicated in order to be able to provide a highly available binding service; 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 $\alpha' \in Sv_A$. These aspects are discussed further below with the help of fig.4.

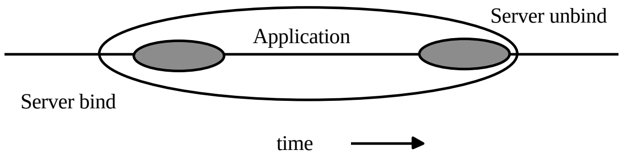

Fig. 4: Binding and unbinding performed using nested transactions.

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 apart 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]. 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 Sv for an object provide redundancy for server creation and the set of nodes in St 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 $\in$ Sv_A and that node has access to at least one node $\in$ St_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 [12].

4. Replication using group communication

4.1. Introduction to process groups

A *group* is defined as a collection of distributed processes in which a member process can communicate with other members by multicasting to the full membership of the group. We require the property that a given multicast be *atomic*: either all the functioning members are delivered the message or none. An additional property of interest is guranteeing total order: all the functioning members are delivered messages in identical order. Clearly, these properties are ideal for replicated data management: each process manages a copy of data, and given atomic delivery and order, it is easy to ensure that copies of data do not diverge. However, achieving these properties in the presence of failures is not simple.

For example, a multicast made by a process can be interrupted due to the crash of that process; this can result in some connected destinations not receiving the message. Process crashes should ideally be handled by a fault tolerant protocol in the following manner: when a process does crash, all functioning processes 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 processes are prone to failures, it is impossible to guarantee that all non-faulty processes will reach agreement in finite time [13]. This impossibility stems from the inability of a process to distinguish slow processes 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.

A process group therefore needs the services of a *membership service* that executes an agreement protocol to ensure that functioning processes within any given group will have identical views about the 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.

A word on the treatment of partitions. Despite efforts to minimise incorrect suspicions by processes, it is possible for a subgroup of mutually unsuspecting processes to wrongly agree (though rare it may be in practice) on a functioning and connected process as a crashed one, leading to a ‘virtual’ partition. There is thus always a possibility for a group of processes to partition themselves (either due to virtual or real network partitioning) 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. We will assume the existence of such a primary partition membership service. In common with most process group systems that have been designed, we will assume that when a member of a group fails by crashing, it looses all its state. If a crashed process joins a group after recovery, it does so as a new member.

4.2. Primary-backup passive 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 a recent review paper [14].

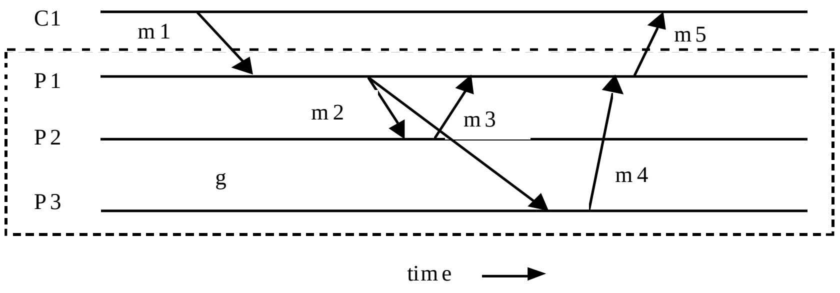

fig. 5: Primary-backup passive replication

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_{sv} and g_{st}). However, since it is the server group that performs the computation, a simplification is possible and the need for g_{st} can be dispensed. Since a process group manages its membership view information, the binding service is no longer required to maintain this information for Sv. 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, (as we are not creating g_{st}) the binding service maintains the list St 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 Sv_A, and St_A. The client constructs a list of 2K+1 nodes in Sv_A, with one listed as primary and requests one of them (say α) to create a group; α creates the group (g_{sv}), and registers the name of the primary with the binding service. The primary of g_{sv} acts as a client to obtain the persistent state of the object from any node $\in St_A$. It then multicasts the state to other members of g_{sv}. The client sends requests to the primary of g_{sv} (these are normal object invocations). If a 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 an object, it explicitly sends a 'disconnect' invocation to the corresponding g_{sv}. 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, and provides tolerance against total failures). There are two steps involved here: (i) the group updates the object stores named by St_A, and then makes a list of nodes where the updates have not succeeded; and (ii) the group 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.

If membership of g_{sv} changes, functioning members of minority partitions simply suicide, leaving stable states unaffected. If clients crash or get separated from groups due to partitions, or if failures prevent the formation of a majority group (*group failure*) then forward progress is possible only in an application specific manner. Unfortunately, there are further pitfalls; the problem being that we have no standard way of ensuring that steps (i) and (ii) discussed earlier (updating stable states and set St_A at the binding service) are completed atomically in the presence of total/group failures. If step (i) completes but not (ii), then the binding service can contain the names of replicas with old states. The problem does not go away in a scheme that uses a group g_{st} for managing stable states.

The only way out is to enrich the system with either (i) transactional facility; or (ii) enhance process groups with direct support for maintaining views on stable store, such that groups can be reconstructed even after group failures (recent research has shown how process groups can be enhanced in this manner [15,16]). In the spirit of this investigation, assume the second option. Then we essentially require some application specific way of dealing with client failures or consequences of partitions that separate clients from server groups or server groups from the binding service.

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.

5. Using transactions and group communication

5.1. 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 failures (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 service 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 failures (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 service 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. They can conveniently support service availability.

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 failures. As indicated earlier, to do this would require building a subsystem with either transactional facilities or to enhance 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 failures, 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 failures.

5.2. 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 an application structuring mechanism 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’ [17]: 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 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. Although a scheme has been designed that does not make use of process groups [18], however, it is static and not resilient to total failures. We briefly outline an active replication scheme that made use of group communication and was implemented for the Arjuna transaction system [19]. The details of binding essentially remain as before.

It is possible to exploit transaction concurrency control to do away with message ordering, but membership service and atomic multicasts are still requirted. Transaction concurrency control must ensure that all of the members of a server group are either read locked or write locked, so that 'exclusive write/shared read' policy extends to object groups. Once a server group gets write locked, then all invocations from the client are naturally serviced by the members of the server group in the order invocations are made, where identical state changes will occur at the member replicas. On the other hand, if a group is read locked, then the members of the group could receive invocations in different order from different clients, but this does not cause any problems of state divergence since no state changes are taking place. Thus, a transaction system strictly requires multicasts with just the atomic message delivery property (all functioning and mutually connected members of a group are delivered a message) but without the ordering property. Such an approach has performance advantage, since it is usually possible to implement faster protocols for atomic unordered delivery compared to atomic and ordered delivery. Locking using atomic unordered multicasts to a group works as follows: a client must lock a majority of the members, else it aborts and retries.

6. 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 [20]. 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 and BT Research.

References

[1] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
[2] M.C. Little, "Object Replication in a Distributed System", PhD Thesis, University of Newcastle upon Tyne, September 1991.
[3] K.Birman and T. Joseph, "Exploiting virtual synchrony in distributed systems", Proc. of 11th ACM Symposium on Operating System Principles, Austin, November 1987, pp. 123-138.
[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] D. Dolev and D. Malki, “The Transis approach to high availability cluster communication”, CACM, 39 (4), April 1996, pp. 64-70.
[6] F. Cristian, “Synchronous and asynchronous group communication”, CACM, 39 (4), April 1996, pp. 88-97.
[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] 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.
[9] Rebuttals from Cornell, Operating Systems Review, 28 (1), January 1994.
[10] A. Schiper and M. Raynal, “From group communication to transactions in distributed systems”, CACM, 39(4), April 1996.
[11] F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.
[12] 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.
[13] M.J. Fiscer. N. Lynch and M.S. Patterson, “Impossibility of distributed consensus with one faulty process”, JACM, 32 (2), 1985, pp. 374-383.
[14] R. Guerraoui and A. Schiper, “Software-based replication for fault tolerance”, IEEE Computer, April 1997, pp. 68-74.
[15] I. Keider and D. Dolev, “Efficient message ordering in dynamic networks”, ACM Symp. on Principles of Distributed Computing, PODC, May 1996.
[16] D. Black, P. Ezhilchelvan and S. K. Shrivastava, “Enhancing replica management services to tolerate group failures”, Technical Report, University of Newcastle upon Tyne.
[17] 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.
[18] E. Cooper, "Replicated distributed programs", Proc. of 10th ACM Symposium on Operating System Principles, Washington, Dec. 1985, pp. 63-78.
[19] M.C. Little and S.K. Shrivastava, "Replicated K-resilient objects in Arjuna", in Proc. of 1st IEEE Workshop on the Management of Replicated Data, Houston, November 1990, pp. 53-58.
[20] 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.