Maintaining Information about Persistent Replicated Objects in a Distributed System

M.C. Little†, D.L. McCue‡ and S.K. Shrivastava†

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

‡Xerox Corporation
Webster, New York 14580

Appeared in the Proceedings of the Thirteenth International Conference on Distributed Computing Systems, Pittsburgh, May 1993, pp 491 - 498

Abstract

The paper presents a general model for persistent replicated object management and identifies what meta information about objects needs to be maintained by a naming and binding service to ensure that objects named by application programs are bound to only those object replicas which are in a mutually consistent state. These ideas are developed within the framework of a distributed system in which application programs are composed of atomic actions (atomic transactions) manipulating persistent (long-lived) objects.

Key words

persistent objects, atomic actions, distributed systems, replication, naming and binding.

1. Introduction

We consider a distributed system in which application programs are composed out of atomic actions (atomic transactions) manipulating persistent (long-lived) objects. Atomic actions ensure that only consistent state changes to objects take place despite failures such as node crashes. Objects not in use normally remain in a passive state: their states are stored in object stores (filing systems for objects). Passive objects are activated on demand: an active object will have its state loaded from the object store to a server for that object. In such a system, a naming and binding service is normally required for maintaining information about objects (e.g., host name of the object store). The service is responsible for ensuring that any objects named in an application program get bound to the corresponding servers. When objects are replicated for increased availability, the naming and binding service is then required to ensure that objects named by application programs are bound to only those object replicas which are in a mutually consistent state. There is much more to this aspect of replica management than meets the eye. We develop a model of replicated object management and describe various ways of maintaining information about replicated objects in a consistent manner. Although the topic of naming and binding in persistent object systems has received much attention (e.g., [7][8][17]), the technical issues of binding to replicated objects in a distributed system have not been addressed adequately. Our ideas derive from an actual implementation of replica management protocols designed and implemented for the Arjuna distributed system [11][12][18].

2. Background

2.1. Failure assumptions

It is assumed that the hardware components of the system are workstations (nodes), connected by a communication subsystem (for example, a local area network). A node is assumed to possess the *fail-silent* property: it either works as specified or simply stops working (crashes). A node may have both stable object store 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 assume further that processes on functioning nodes are capable of communicating with each other.

2.2. Objects and atomic actions

An object is an instance of some *class*, where 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. 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. In a distributed system, some of the objects accessed by an application may be physically remote from the application (i.e., on a different machine). An operation invocation upon a remote object is typically performed via a *remote procedure call* (RPC).

A computational model that has been widely advocated for constructing robust distributed systems is based upon the concept of using atomic actions (atomic

transactions) [5] controlling operations on persistent objects. All operation invocations may be controlled by the use of atomic actions which have the well-known properties of (i) serialisability, (ii) failure atomicity, and (iii) permanence of effect. The object and atomic action model provides a natural framework for designing fault-tolerant systems with persistent (long-lived) objects [1][4][13][18][19]. Persistent objects not in use are normally resident in a *passive* state on object stores. A passive object is made *active* by loading its state and methods from the object store to volatile store and associating a process (a *server*) for receiving RPCs for method executions. Atomic actions are employed to control the state changes to these objects, and the properties of atomic actions given above ensure that only consistent state changes take place, despite shared access and any failures.

A number of system services are required to support distributed computations structured out of objects and actions. We enumerate them below:

Atomic Action service: provide atomic action support to application programs.

RPC service: provide an object invocation facility through an RPC mechanism;

Object Storage service: provide a stable storage repository for objects; these object are assigned unique identifiers (UIDs) for naming them;

Naming and binding service: provides a mapping from user-given names of objects to UIDs, and from UIDs to location information such as the identity of the host containing the object store where the state of the object resides.

To be able to access a persistent object, an application program must be able to obtain information about the object's location. The application program can request this information from the naming and binding service by presenting it the name of the object (a string). The naming service can map this string to the UID of the object and then map the UID to the location information. 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). Below we discuss this aspect further for the case of replicated objects.

2.3 Object replication

The persistent state of an object typically resides on a single object store of a node; if that node is down, then the object becomes *unavailable*. The *availability* of an object can be increased by replicating it on several nodes and thus storing its state in more than one object store. Such object replicas must then be managed through appropriate replica-consistency protocols to ensure that the object copies remain mutually consistent. We will consider the case of *strong consistency* which requires that all replicas that are regarded as *available* be mutually consistent (so the persistent states of all available replicas are required to be identical). We discuss below three aspects of replica consistency management; the first and the third are concerned mainly with the management of *meta* information about object replicas, whereas the second is concerned mainly with the management of replicas themselves:

(1) *Object binding:* It is necessary to ensure that, when an application program presents the name of an object, which is currently passive, to the naming and binding service, the service returns a list containing information about only those replicas of the object that are (a) mutually consistent, and also (b) contain the latest state of the object. From this information, one, more, or all replicas, depending upon the replication policy in use (see below), can be activated. If the object has been activated already, then given the name of the object, the naming and binding service must permit binding to all of the servers that are managing replicas of the activated object. If we assume a dynamic system permitting changes to the degree of replication for an object (e.g., a new replica for an object can be added to the system), then it is important to ensure that such changes are reflected in the naming and binding service without causing inconsistencies to current users of the object.

(2) *Object activation and access*: A passive object must be activated according to a given replication policy. We identify three basic object replication policies [3]:

(i) *Active replication:* In active replication, more than one copy of a passive object is activated on distinct nodes and all activated copies perform processing [5]. So, in the absence of network partitions preventing communication, an object remains available to applications so long as at least one replica is functioning.

(ii) *Coordinator-cohort passive replication:* Here, as before, several copies of an object are activated; however only one replica, the *coordinator*, carries out processing [6]. The coordinator regularly checkpoints its state to the remaining replicas, the *cohorts*. If the failure of the coordinator is detected, then the cohorts elect one of them as the new coordinator to continue processing.

(iii) *Single copy passive replication*: In contrast to the previous two schemes, only a single copy is activated; the activated copy regularly checkpoints its state to the object stores where states are stored [2]. This

checkpointing normally occurs as a part of the commit processing of the application, so if the activated copy fails, then the application must abort the affected atomic action (restarting the action will result in a new copy being activated).

Activated copies of replicas (cases (i) and (ii)) must be treated as a single group by the application in a manner which preserves mutual consistency. Suppose the replication policy is active replication. Consider the following scenario (see figure 1), where group G_A (replicas A_1, A_2) is invoking a service operation on group G_B (a single object B) and B fails during delivery of the reply to G_A. Suppose that the reply message is received by A_1 but not by A_2, in which case the subsequent action taken by A_1 and A_2 can diverge. The problem is caused by the fact that the failure of B has been 'seen' by A_2 and not A_1. To avoid these problems, communication between replica groups can require reliable distribution and ordering guarantees not associated with non-replicated systems: reliability ensures that all correctly functioning members of a group receive messages intended for that group and ordering ensures that these messages are received in an identical order at each functioning member [16].

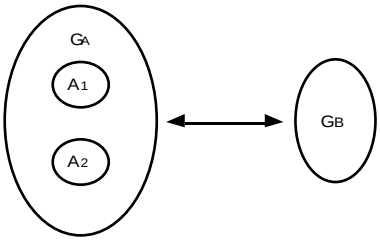

Figure 1: Operation Invocation for Replicated Objects

(3) *Commit processing and object passivation*: Once an application has finished using an object, it is necessary to ensure that the new states of mutually consistent object replicas get recorded to their object stores; this takes place during the commit time of the application's atomic action. At the same time, it is also necessary to ensure that the information about object replicas maintained by the naming and binding service remains accurate. Consider an application that modifies some object, say A, and active replication is in use; suppose at the start of the application two replicas for A (A_1 and A_2) are available, but that the crash of a node makes one of them (say A_2) unavailable, so only A_1 gets modified; then at commit time, information maintained about A within the naming and binding service should be modified to 'exclude' A_2 from the list of available replicas of A (else subsequent applications may end up using mutually inconsistent copies of A). An active copy of an object which is no longer in use will be said to be in a *quiescent* state; a quiescent object can passivate itself by destroying the server, and if necessary, letting the naming and binding system 'know' of this state change.

This paper is primarily concerned with the first and the third aspects of replica management. Many replica management protocols have been presented in the literature dealing with the second aspect of object replication (e.g., [1][2][5][6][9][11][14]), but little attention has been paid as to how the information about replicas is maintained in a consistent manner. Yet, as the discussion above has indicated, this issue is at least as important as the treatment of replica groups.

3. System model

3.1. Client-server bindings

With the above discussion in mind, we present a general model for persistent replicated objects. We assume that for each persistent object there is at least one 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 binary of the code for the object's methods which may well be stored in the object store of α). It is the responsibility of a node, such as α, to bind an incoming object invocation from a client application to the right server. This binding can *break* if the server (client) node crashes; we assume that a broken binding stays that way till the application level action terminates. So in our model, bindings to servers are created only during the start of a client atomic action (as invocations are made); if some bound server subsequently crashes then the corresponding binding is broken and not repaired (even if the server node is functioning again); all the surviving bindings are broken at the termination time of the action.

Subject to the above binding criterion, a client is bound as follows. If α receives an invocation from the client, and the server is already present, then the invocation is sent to that server; otherwise the object is passive at α and needs to be activated before invocation can be performed; this requires creating a server and loading the methods and the state from some object store. We will also assume that there is at least one node (say β) whose object store contains the state of the object; we do not require that α be the same as β. So, in order to obtain the state of an object for activation, α may have to contact the naming and binding service to obtain the name of the node (β) storing the object state. We therefore assume that for every persistent object, the naming and binding service maintains two sets of node related data:

(i) for an object A, the set St_A contains the names of nodes whose object stores contain states of A; and

(ii) the set Sv_A contains the names of nodes capable of running a server for *A*.

An object can become unavailable if all the nodes $\in Sv_A$ are down (have crashed) and/or all the nodes $\in St_A$ are down. 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 a node currently $\in St_A$ if it is found not to contain the latest (committed) state of the object (say the node has crashed); similarly, it should be possible for this node to be included back in St_A once it does contain the latest committed state for the object. We assume that the naming and binding service itself is built out of one or more persistent objects, so the above state transitions are (naturally) performed under the control of atomic actions. The objects providing the naming and binding service themselves can be replicated in order to be able to provide highly available service; in this paper we will assume that the service is always available.

3.2. Object replica management

We now consider several possible ways of managing object replicas of *A*, given the cardinality of the sets St_A and Sv_A (assume that *A* is currently passive, i.e., not in use).

(1)$|Sv_A| = |St_A| = 1$: This represents the case of a non-replicated object (see figure 2). Activating *A* will consist of creating a server at the node $\in Sv_A$ (say α) and loading the state from the node $\in St_A$ (say β; a common case is $\alpha=\beta$). At commit time, the state of the object is copied back to β. If either α or β is down, or crashes during the execution of an atomic action, then the action must abort.

Object Server α Object State β

Figure 2: $|Sv_A| = |St_A| = 1$

(2)$|Sv_A| = 1 \& |St_A| > 1$: This represents the case where only the state of an object is replicated (see figure 3). Activating *A* will consist of creating a server at the node $\in Sv_A$ (say α) and loading the state from any node $\in St_A$. At commit time, an attempt is made to copy the 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. An atomic action using *A* must abort if α is down (or crashes during execution), or all the nodes $\in St_A$ are down (or crash during execution). Note that this scheme represents the single copy passive replication technique.

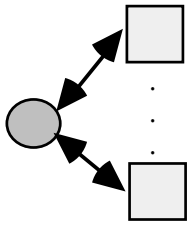

Figure 3: $|Sv_A| = 1 \& |St_A| > 1$

(3)$|Sv_A| > 1 \& |St_A| = 1$: This represents the case where servers can be replicated, but only a single object state is available (see figure 4). There is some flexibility available in choosing an activation policy for *A*; activating *A* can 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 $\in St_A$. If $|Sv_{A'}| = 1$, then no replication for *A* is available; if $|Sv_{A'}| = k$, where $k>1$, then active replication or coordinator-cohort replication for *A* is in operation, permitting up to k-1 server replica failures of *A* to be masked during the execution of an application.

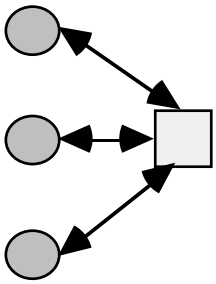

Figure 4: $|Sv_A| > 1 \& |St_A| = 1$

(4)$|Sv_A| > 1 \& |St_A| > 1$: This is the most general case (all the previous schemes being special cases) providing maximum flexibility during object activation (see figure 5). Activation takes place as in (3) above, except that each server is free to load the state of the object from any of the nodes $\in St_A$. At commit time, steps similar to those outlined in (2) are necessary.

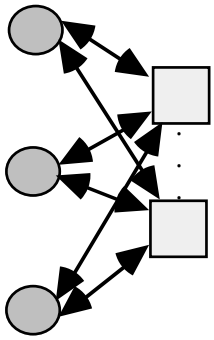

Figure 5: $|Sv_A| > 1$ & $|St_A| > 1$

We next consider the case of a client accessing an object (*A*) that has been activated already. If *A* was activated as described in (1) or (2), then the client must be bound to the server for *A* . If *A* was activated as described in (3) or (4), then again the client must be bound to all of the functioning servers $\in Sv_{A'}$.

4. Binding and activation

For simplicity, we shall assume that a client making use of the naming and binding services has already obtained the UID of the object it is accessing (say UID_A for *A*). We assume that the naming and binding service is composed from two distinct objects (although of course they could be implemented as one): an *Object Server* database for maintaining UID_A to Sv_A mappings for all the persistent objects and similarly an *Object State* database for maintaining UID_A to St_A mappings. From the description of the system model in Section 3, we can see that clients access the Object Server database, while servers make use of the Object State database.

4.1. The object server database

The Object Server database maintains information, in the form of a list per object, on the locations (host names) of servers. Each such list is concurrency controlled independently using locks. The operations provided by the database include:

- *GetServer(objectname):* for an object, *A*, this read operation returns the list of nodes $\in Sv_A$ given its name (UID_A).
- *Insert(objectname, hostname)* : This update operation is called to add the name of a node which can act as a server site for the named object.
- *Remove(objectname, hostname)*: A complementary operation to the previous one.

4.1.1. Selecting object servers

As indicated before, operations of the object server database (as well as on the object state database) are executed as atomic actions. We have identified three different ways in which these atomic actions are structured with respect to the application actions; these are discussed below. We will assume, for the sake of simplicity, that a client is accessing only a single object (*A*), further, a replication policy characterised by figure 5 will be assumed. Note that in the following diagrams shaded ellipses represent atomic actions operating on the object server database, while non-shaded ellipses are application level actions.

4.1.2. Standard atomic actions

In the first scheme we will assume that the *GetServer(UID_A)* operation of the object server database is executed as a nested atomic action of the client action (see figure 6). If this operation fails then the client action must abort. Assuming the operation does not fail, the client uses a fixed algorithm to select a subset $Sv_{A'}$ for binding; the binding will succeed for all the nodes $\in Sv_{A'}$ that are functioning. If necessary, these servers obtain the state of the object by contacting the Object State database (the operations exported by this database will be discussed in section 4). Note that a client must first execute the operation *GetServer(UID_A)* in order to be able to bind to the servers for *A*. If there are several clients accessing *A*, then this would result in several invocations of *GetServer(UID_A)*. This is possible in this scheme as *GetServer(UID_A)* is a read operation, permitting shared access from within client actions.

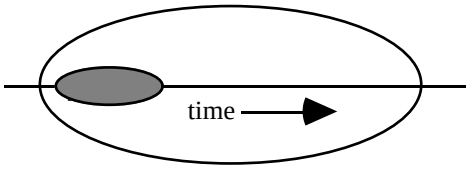

Figure 6: Using nested atomic actions

At the end of the action the client commits, and the read lock on the database entry is then released. In this scheme the set Sv_A represents the 'static' set of potential servers, and no attempt is made by a client to dynamically remove a node from this set if it detects a crash (since this would then require a write lock on the database entry, a situation which we are trying to avoid). If a node (δ) with a server crashes, then upon recovery, it executes the *Insert(UID_A, δ)* operation, before it is ready to act as a server node. Although $\delta \in Sv_A$, execution of this operation is necessary to check that *A* is quiescent (recall that the *Insert* operation requires a write lock, and will only succeed when there are no clients using *A*).

This ensures that bindings are managed correctly in the presence of server crashes and recovery. The *Insert* and *Remove* operations can be used by specific application programs for explicitly changing the membership of Sv_A (for varying the degree of server replication). A shortcoming of this scheme is that the set Sv_A need not represent the current view of functioning server nodes; so at binding time each and every client determines 'the hard way' that a server is unavailable. This deficiency can be rectified if clients are allowed to remove failed servers from Sv_A at binding time; such schemes will be discussed subsequently.

The above scheme does permit one optimisation: if clients are only performing read operations on an object then it is possible for concurrent clients to activate and bind to different (possibly disjoint sets of) servers for the object. In a simple scheme, a client binds to any convenient node $\in Sv_A$.

4.1.3. Enhancing the functionality of the object server database

We now consider two schemes which permit clients to update Sv_A. A client accesses the database by making use an independent top-level atomic action (this could be run separately, as shown in figure 7 or invoked from within the client action, as a nested top-level action, as shown in figure 8). Let us first describe the additional state information which the database is required to store for each persistent object. We assume that for each node $\in Sv_A$, a *use list*, of the form $<N_i, C_i>$, where C_i counts the number of clients of the server from node N_i , is kept. An object *A* will be either in a quiescent or passive state if for all nodes $\in Sv_A$, the corresponding *use list* is empty.

- *Increment(clientnode, hostname$_1$, ..., hostname$_n$)*: The counter value for *clientnode* within the *use list* of each *hostname* is incremented by one (if there is no entry for *clientnode*, then a new entry, with counter set to one is created).
- *Decrement(clientnode, hostname$_1$, ..., hostname$_n$)*: A complementary operation to the previous one. Both of these operations require a write lock.

In these two schemes, a crash of a client does not automatically undo changes made to the database. So, failure detection and cleanup protocols will be required. For example, the Object Server database could periodically check if its clients are functioning, and if necessary update *use list* if crashes are detected. These aspects of system design are beyond the scope of this paper.

(i) Independent actions

Within the first top-level action the client first contacts the database by invoking *GetServer(UID$_A$)*. We now assume that this operation also returns the *use lists* for the nodes $\in Sv_A$. If the use lists are all empty, the client is free to select any subset $Sv_{A'}$ for binding. The client then invokes *Increment (..)* and *Remove(..)* operations to respectively, update use lists and remove any failed servers from the set Sv_A. If the *use list* returned is non-empty, then the client tries to bind to only those servers with non-zero counters; if any failures are detected, then *Remove(..)* is called as before. After the client action has terminated, the *Decrement (..)* operation is called (the last top-level action in figure 7) to update the *use list*.

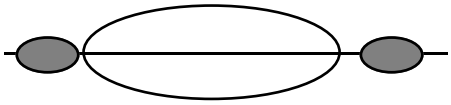

Figure 7: Using independent top-level actions.

In this scheme, the set Sv_A maintained by the database contains a relatively up-to-date list of functioning server nodes, thereby rectifying the shortcoming of the previous scheme.

(ii) Nested top-level actions

If the system provides the facility for invoking top-level atomic actions from within an action (such actions are called nested top-level actions), then the scheme illustrated by figure 7 can be replaced by the one shown in figure 8.

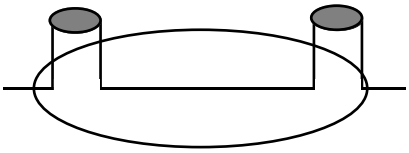

Figure 8: Using nested top-level actions.

4.2. The object state database

As stated earlier, the Object State database maintains information, in the form of a list per object, on the locations (host names) of object stores (UID_A to St_A mappings). As was the case with the server database, such lists can be concurrency controlled independently. The database exports several operations, including:

- *Getview(objectname)*: for an object, *A*, this read operation returns the list of nodes $\in St_A$ given its name (UID_A)

• *Exclude(<objectname$_1$, nodelist$_1$>, <objectname$_2$, nodelist$_2$>, ...)*, where <*objectname$_i$*, *nodelist$_i$*> is the *exclude list* for *objectname$_i$*: for every *objectname$_i$* the corresponding St set is modified to remove the entries for the hostnames listed in *nodelist$_i$* .

• *Include(objectname, hostname)* : This update operation is called to add the name of a node which can act as an object server site for the named object.

A crashed node with an object store must ensure, upon recovery that its objects do contain the latest committed states. For this purpose, it can run atomic actions to update its object states and then invoke the *Include (..)* operation for making the object states available again [11][12].

4.2.1. Accessing the object state database

The object state database can be accessed using schemes similar to the ones discussed for the server database.

(i) Standard atomic actions

This scheme is similar to the server database technique discussed with reference to figure 6. A freshly created server must obtain the state of the object from an object store. Such a server invokes *getview(UID_A)* operation (as a nested action of the client) to get the list of object server nodes $\in St_A$ and contacts any of those nodes to get the state. At commit time, the server copies the new state to all the functioning nodes $\in St_A$; if the copy operation fails for some nodes $\in St_A$, then the names of these nodes must be excluded from the set St_A. Since the *exclude* operation requires a write lock, the read lock currently held on the list must be *promoted* to a write lock; if the lock promotion succeeds, the *exclude* operation can be performed, else the client action must abort. An obvious read optimisation is possible: if the client has not changed the state of the object, then no copying to object stores is necessary.

The above scheme suffers from the disadvantage that if an object is being shared between several clients, several read locks would be held on to the list for the object, and a lock promotion request by a client would be refused. This can be rectified by introducing type-specific concurrency control for database entries. We introduce an *exclude-write lock* type, which can be shared with read locks. To *exclude*, a client first promotes its read lock to this new lock type.

(ii) Independent and nested top-level actions

The schemes for accessing the object server database using independent or nested top level actions can be adapted to work with the object state database as well; these schemes will require the kind of enhancements discussed in section 4.1.3. These details are not presented here to save space.

5. Concluding remarks

We have presented a general model for persistent replicated object management and identified what meta information about objects needs to be maintained (information about available servers and object stores maintained by the naming and binding service). We have then described how this information can be manipulated in a consistent manner. The ideas presented here can be developed further. A useful extension would be based on investigating possible ways of reducing dependence on the need for atomic action support for the naming and binding services. This is important as most such services do not provide such support. Although we do not see any simple way of completely doing away with atomic action support, one way would be to keep available *server related data* in a 'traditional (non-atomic)' name server, and retain the services of a modified object state server database with atomic action support. It would then become the responsibility of the Object State database to guarantee consistent binding of clients to servers.

We have implemented major aspects of the services mentioned in this paper as part of the current version of the Arjuna distributed programming system [18][19]. Arjuna is an object-oriented programming system implemented in C++ that provides a set of tools for the construction of fault-tolerant distributed applications. Arjuna provides nested atomic actions for structuring application programs. Atomic actions control sequences of operations upon (local and remote) objects, which are instances of C++ classes. Operations upon remote objects are invoked through the use of remote procedure calls. The prototype implementation supports single copy passive replication and active replication [11][12]. The two databases have been implemented as a single Arjuna object, referred to as the *group view database*. Users (or the system) can choose to interact with this database using several of the techniques we have described - by default, standard atomic actions are used. The Arjuna stub generator tool [15], together with the supporting system libraries hide much of the interactions with the database.

Acknowledgements

The work reported here has been supported in part by grants from the UK Science and Engineering Research Council, MOD, and ESPRIT projects ISA (No. 2267) and BROADCAST (No. 6360).

References

1. M. Ahamad et al, "Fault Tolerant Computing in Object Based Distributed Operating Systems", Proceedings of the 6th IEEE Symposium on Reliable Distributed Systems, Williamsburgh, March 1987.
2. P.A. Alsberg and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources", Proceedings of the Second International Conference on Software Engineering, San Francisco, October 1976.
3. M.H. Olsen, E. Oskiewicz and J.P. Warne, "A model for interface groups", Proc. of 10th IEEE Symposium on Reliable Distributed Systems, SRDS-10, Pisa, October 1991.
4. R. Balter et al, "Architecture and Implementation of Guide, an Object-Oriented Distributed System", Computing Systems, 4(1) April 1991, pp. 31-67.
5. P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
6. 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.
7. M. Carey et al, "The Architecture of the EXODUS Extensible DBMS", Proc. of 1st International Workshop on Object-Oriented Database Systems, Pacific Grove, September 1986.
8. J.B. Moss, "Design of the Mneme Persistent Object Store", ACM Transactions on Information Systems, 8, April 1990, pp. 103-139.
9. D.K. Gifford, "Weighted Voting for Replicated Data", Proc. of 7th ACM Symposium on Operating System Principles, December 1979, pp. 150-162.
10. J.N. Gray, "Notes on Database Operating Systems", in Operating Systems: An Advanced Course, Lecture Notes in Computer Science, Vol. 60, Springer-Varlag, 1978, pp.393-481.
11. M.C. Little, "Object Replication in a Distributed System", PhD Thesis, Newcastle University Computing Science Department, September 1991.
12. 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.
13. B. Liskov and R. W. Scheifler, "Guardians and Actions: Linguistic Support for Robust, Distributed Programs", ACM Transactions on Programming Languages and Systems, Vol. 59(3), July 1983, pp. 381-404.
14. M.H. Oki, "Viewstamped Replication for Highly Available Distributed Systems", PhD Thesis, MIT Laboratory for Computer Science, August 1988.
15. G. Parrington, "Programming Distributed Applications Transparently in C++: Myth or Reality?", Proceedings of the OpenForum Technical Conference, Utrecht, November 1992, pp 205-219.
16. F.B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.
17. E. Shekita and M. Zwilling, "Cricket: A Mapped, Persistent Object Store", Proceedings of the 4th International Workshop on Persistent Object Systems Design, Implementation and Use, 1990.
18. S.K. Shrivastava, G. N. Dixon and G.D. Parrington, "An overview of the Arjuna distributed programming system", IEEE Software, January 1991, pp. 66-73.
19. S.K. Shrivastava and D.L. McCue, "Structuring fault-tolerant object systems for modularity in a distributed environment", IEEE Trans. on Parallel and Distributed Systems (to appear).