

Implementing high availability CORBA applications with Java

M. C. Little and S. K. Shrivastava,

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

Appeared in the Proceedings of the IEEE Workshop on Internet Applications (WIAPP '99)

Abstract

The integration of Java and CORBA has opened the way for a wide variety of Internet applications. However, such applications will frequently come across communication and node failures which can affect both their performance and consistency. Therefore, there is a need for techniques which will allow applications to tolerate such failures and continue to provide expected services. A common technique is through the use of atomic transactions, which ensure that only consistent state changes take place despite concurrent access and failures. However, they may be insufficient to guarantee forward progress. This can be accomplished by replicating resources, so that the failure of a subset of replicas can be masked from users. This paper describes a toolkit that integrates transactions and replication by making use of standard CORBA services .

1. Introduction

The integration of Java with the OMG's CORBA [1][2] has opened the way for a wide variety of Internet applications. However, such applications will frequently come across communication and node failures which can affect both the performance and consistency. For commercial services, such failures can result in loss of revenue and credibility. Therefore, there is a need for techniques which will allow applications to tolerate such failures.

A common technique for fault-tolerance is through the use of atomic transactions, which have the well known ACID properties, operating on persistent (long-lived) objects. Transactions ensure that only consistent state changes take place despite concurrent access and failures. However, they may be insufficient to ensure that an application makes forward progress. It is possible to improve the probability of forward progress by increasing the availability of an application's objects (resources) by replicating them on several nodes (machines), and managing them through appropriate replica-consistency protocols. The failure of a subset of these replicas may then be masked, allowing the application to continue.

In this paper we shall describe a toolkit which provides support for the construction of fault-tolerant Java applications which can possess both consistency (through transactions) and forward progress guarantees (through replication). Importantly, we shall show how we have accomplished this in a manner which requires no changes to the middleware on which applications run, thus improving portability. Because of its wide acceptance, the middleware we chose was CORBA, where the Object Request Broker (ORB) enables distributed objects to interact with each other and the Object Transaction Service (OTS). Although described within this context, the techniques should be applicable to any other distributed computing platform. In fact an earlier version of this scheme was implemented on top of a set of Unix machines without using CORBA [3].

2. Managing replicated persistent objects

In order to understand the design of the toolkit, it is first necessary to review the issues involved in replicating persistent objects. First we state our basic failure assumptions. It will be assumed that the hardware components of the system are computers (nodes), connected by a communication system. 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 assume that the communication system can suffer *partition failures*, preventing functioning computers from communicating with each other; a partition eventually heals, permitting disrupted communication to be resumed. In this paper we will only address the case of *strong consistency* which requires that the states of all replicas that are regarded as available be kept mutually consistent.

2.1 Binding information on non-replicated persistent objects

An object has state and methods; the methods are executed by a server and the state of the object has a permanent home on some persistent object store. We assume that there is at least one machine (say α) that is capable of running a server for that object (i.e., it has access to code for the object's methods), and there is a machine (β) whose object store contains the persistent state of the object. The server and the persistent object store need not reside on the same machine (α and β could be same or different).

To be able to use an object in a distributed environment, an application (a CORBA client) must have a *reference* to that object. As shown in figure 1, the client requests this information from a naming service by presenting it the name of the object (RPC1). Assuming that the name to reference mapping exists within the naming service, the naming service returns the reference (name of α); the client then directs its invocations to α (RPC2). It is up to α to forward the invocation to the server for the object within α . If such a server within α does not exist, α may have to *activate* the object by loading the methods of the object at an existing server or by creating a new server.

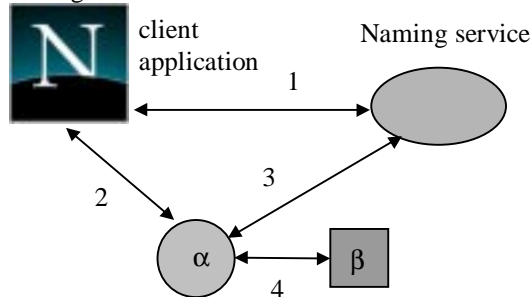


Figure 1: Object binding.

The server at α may have to contact the naming service to obtain a reference to β (RPC3) in order to be able to load/store the state of the object from/to the store at β (RPC4). A complimentary action to object activation is *passivation*: if no more bindings to an activated object exists, α may decide to unload the methods of the object from the server, store the state at the object store and even delete the server.

In a non-replicated system, a common activation scheme for persistent objects activates an object at a fixed server node which also maintains the persistent state. Therefore, a name server that maps an object name to the server location is all that is required.

2.2 Binding information on replicated persistent objects

In the scheme shown in figure 1, an object can potentially become unavailable to the client if any of the four communication paths get disrupted due to partitions or any of the nodes hosting α , β , or the naming service crash. We will assume that we do not have direct control over communication resources (possibly because we simply using the 'Internet communication cloud'), 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. For the time being we will assume that the naming service remains available and accessible; subsequently we will remove this assumption.

Given the above description, it can be seen that the naming service must maintain two sets of data [4]:

- (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 .

It is now possible to activate an object at more than one machine and obtain its state from more than one object store. We must avoid the situation where different clients of A are bound to different subsets of Sv_A and interacting with different subsets of St_A . Clearly, object binding has become complicated, requiring 'atomic' interactions between object servers, object stores and the naming service; this is discussed at length in the paper.

2.3 Replica management for persistent objects

Figure 2 depicts a general replication scheme: single server and object store are replaced respectively by a server group and a state group.

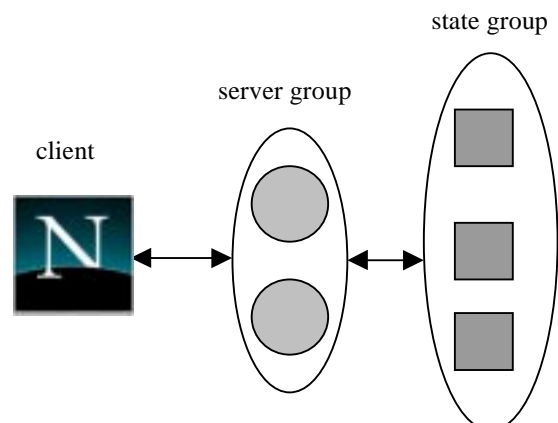


Figure 2: Server and state replication

Process groups with ordered reliable group communications and membership services, (process group for short) provide the necessary functionality for group management [e.g., 5, 6, 7, 8]. There are two main

techniques for group management: a group could be an *active* group [9] or a *primary-backup* group [10]. In the former, every functioning group member performs processing; so active replication requires that functioning group members receive identical invocations in an identical order, and that the computation performed by each member is deterministic. 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.

The replication scheme provided by a primary-backup group is also known as *passive replication*; here only a single copy (the primary) performs processing; the primary regularly checkpoints its state to other 'backup' members of the group (also called the secondaries). Passive replication does not require the restriction of deterministic processing; however, its performance in the presence of primary failures can be poorer than under no failures, because the time taken to switch over to a secondary is non-negligible.

We now consider the specific case of object replication within the context of transactions. First of all we note that there is no explicit requirement for a process group for managing object states; rather *replicated data management* techniques that go hand in hand with transaction commit processing [11] provide an integrated solution to object state management. Three types of server replication schemes are then possible:

(i) *Single copy passive replication*: an object, say A , is activated at a single node $\in Sv_A$ only. If the client is unable to invoke operations on the server (server has failed or partitioned from the client), the client aborts the transaction. Forward progress is possible by activating the object at some other node $\in Sv_A$. This scheme has the advantage of simplicity in that no process groups are required; however, a failure of the type indicated above cannot be masked from the client transaction. This is the default replication scheme supported by our toolkit.

(ii) *Primary-backup server group*: an object, say A , is activated at $2K+1$ nodes $\in Sv_A$ ($K \geq 1$), and this set of nodes is managed as a primary-backup server group. 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, else they have to abort their transactions.

(iii) *Active replication*: same as in (ii), except the group is managed as an active group.

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.

The name 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 (commit their transactions). Name server replication schemes are discussed separately later. In the remaining part of the paper we present specific design details of our toolkit, beginning with the transaction component.

3. The transaction component

For the transaction component of the toolkit to be widely applicable, it must conform to the standards. The most widely accepted transaction standard for distributed objects is the Object Transaction Service (OTS) from the OMG [14]. Furthermore, a Java version of the OTS is at the core of the Sun's Enterprise Java Beans specification [15].

The OTS provides interfaces that allow multiple distributed objects to cooperate in a transaction such that all objects commit or abort their changes together. The transaction service distinguishes between *recoverable* and *transactional objects*. Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or aborts to ensure the consistency of the state changes. In contrast, a transactional object need not be a recoverable object if its state is actually implemented using other recoverable objects. The major difference is that a transactional object need not take part in the transaction commit protocol since it does not maintain any state itself, having delegated that responsibility to other recoverable objects.

The OTS is simply a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transaction properties. Therefore transactional objects require other co-operating services, including persistence (to support the atomicity and durability properties) and concurrency control (to support the isolation properties).

3.1 JTSArjuna

The interfaces defined by the OMG for building transactional applications are too low-level for most programmers. For example, the programmer is required to manage persistence and concurrency control on behalf of every transactional object, and register such objects with each transaction they participate within. Therefore, as well as supporting the raw OTS interfaces, the transaction component of our system, *JTSArjuna*, also provides a higher-level API to hide these details, and simplify the construction of transactional applications [16].

The *JTSArjuna* model for building transactional applications is based on the original Arjuna design [17] and exploits object-oriented techniques to present

programmers with a hierarchy of classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control (see figure 3):

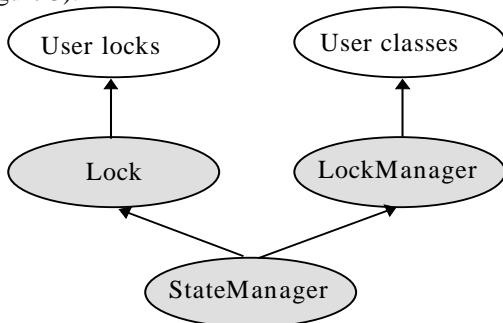


Figure 3: JTSArjuna class hierarchy.

StateManager is the class responsible for naming, persistence and recovery control of persistent objects. When not in use, persistent objects are stored in an object store. All *JTSArjuna* object states are assigned *unique identifiers (UIDs)* for naming and locating them within object stores. To satisfy the durability requirement, StateManager makes use of a persistence service when loading and storing persistent objects.

LockManager is responsible for two-phase locking, and uses instances of the Lock class, in conjunction with a suitable concurrency control service. The programmer uses the `setlock` method to acquire appropriate locks; the transaction system is responsible for releasing these locks.

Classes which inherit from LockManager are automatically transactional, with LockManager and StateManager being responsible for guaranteeing the ACID properties. Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the programmer has no other responsibilities: *JTSArjuna* guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and invokes recovery mechanisms automatically in the event of failures.

In order to support different persistence and concurrency implementations without requiring changes to *JTSArjuna*, LockManager and StateManager access these services through interface objects (as shown in figure 4). At runtime each interface object obtains information from the underlying system about which implementation to use, and binds to that implementation, delegating all requests to it [18]. As we shall show later, this mechanism enables us to incorporate replication transparently.

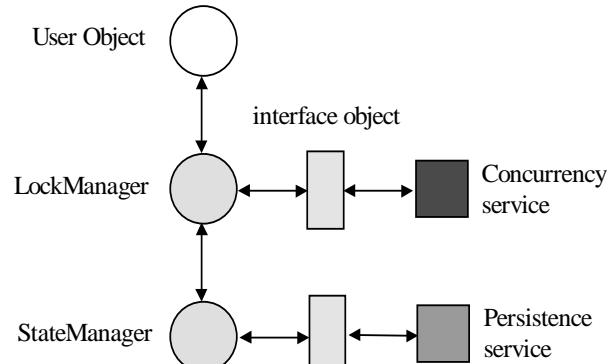


Figure 4: Transaction services.

4. The replication component

The replication component, *Jydra*, supports the types of replication schemes discussed earlier. Its main function is to maintain a transactional name server, (implemented using *JTSArjuna*) called the Group View Database (*GVD*) that accurately represents the binding and replication related information. This database is composed of two distinct transactional databases: an *Object Server* database for maintaining object name, say A , to Sv_A mappings and an *Object State* database for maintaining UID_A to St_A mappings. The reason for implementing this database is that it must be transactional, a property not currently supported by the CORBA naming service standard.

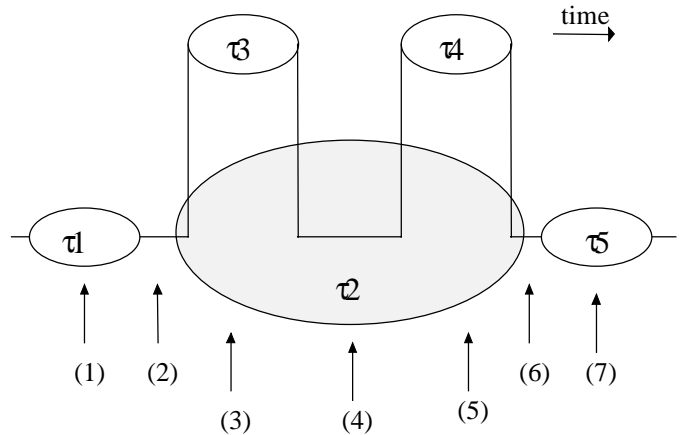


Figure 5: Using the GVD.

In the following sections we will describe each database separately and the components which use it. To help in this we shall use the example transactional bank account application shown below, and refer to figure 5, which considers the progress of the application at various stages in time indicated by the vertical arrows.

```

{
  /* bind to the bank account */
  bankaccount myAccount =
    bankaccountHelper.bind(myDetails);

  current.begin(); /* start of transaction */

  myAccount.balance();
  . . .
  myAccount.credit(...); /* invocations */
  . . .
  current.commit(); /* transaction commits */
}

```

The sequence of activities involved in the example are: (i) creation of bindings to the bank account object; (ii) start of the application transaction; (iii) method invocations; as a part of a given invocation the object will be locked in read or write mode; and (iv) commit/abort of the transaction.

Here is the broad overview of how the replication scheme works. In figure 5, transactions τ_1, τ_3, τ_4 and τ_5 are transactions that use the GVD and transaction τ_2 is the application level bank account transaction. Let object name A refer to the bank account object, `myAccount` and assume that A is passive. Transaction τ_1 activates A at one or more nodes in Sv_A (depending the object replication policy); to obtain the persistent state of A , the servers get the list St_A of available (i.e., mutually consistent) object states from the GVD using τ_3 ; at the commit time of τ_2 , the GVD is updated again using τ_4 and finally, the application unbinds from A using τ_5 . Transactions τ_3 and τ_4 are executed as top level nested transactions to ensure that the GVD does not remain locked for the entire duration of τ_2 .

4.1 The Object Server database

This database maintains two lists per object: (i) for an object A , its *available server list*, giving references to the servers $\in Sv_A$; (ii) for each server $\in Sv_A$, a *use list*, which maintains information about the identity of its clients. An object A is in a passive state if, for each server $\in Sv_A$, the corresponding *use list* is empty. The object is unavailable if the set Sv_A is empty.

The interface to the Object Server database is defined using IDL (the relevant parts of which are shown below); it is derived from the OTS `TransactionalObject` interface which indicates that it must be invoked within the scope of a client transaction:

```

module Jydra
{
  interface ObjectServerDatabase :
    CosTransactions::TransactionalObject
  {
    sequence<string> getServer (in string
      objectName, out boolean alreadyActivated);

    void increment (in string objectName,
      in UID client,
      in sequence<string> activatedHosts);
    void decrement (in string objectName,

```

```

      in UID client);

    void exclude (in string objectName,
      in sequence<string> failedHosts);
  };
};

```

The operations are described below:

- *getServer(objectname, alreadyActivated)*: this returns the available server list for the object, and indicates whether the object is already activated. If it has not been activated then the client is free to choose one or more of the server nodes for activation from the list, otherwise the list is the currently activated replicas which the client must also use.
- *increment(objectname, client, activatedHosts)*: the specified replicas are marked as being activated (in use) by the client, i.e., the client is added to the object's use list.
- *decrement(objectname, client)*: a complementary operation to the previous one, which removes the client from the use list. If the list becomes empty, then the object is no longer activated, so a candidate for passivation.
- *exclude (objectname, failedHosts)*: removes the named nodes from the object's *available list*.

4.1.1 Using the Object Server database

At the start of the application, the client must bind to the bank account object (stage 1 in figure 5). The client (more accurately, *client proxy*, see later) begins a top-level transaction (τ_1), invokes the *getServer* method of the Server database, and binds to the specified number of servers from the list (just one for single copy passive replication, or $2K+1$ otherwise). If during binding any servers are suspected to have failed, their names are removed using the *exclude* operation, preventing subsequent clients from contacting them. Finally, the *increment* operation is invoked, passing the names of servers where the binding succeeded. If a server group has been created, it is then managed by the associated process group service.

During execution of the application transaction τ_2 (stage 4), the client invokes operations on the proxy, which in turn invokes operations on the object (using the process group multicast if a group has been created). Any server failures detected by the client proxy are noted for later exclusion (during τ_5). If the group membership falls below the majority, then the proxy will force transaction abort.

When the client has finished with the object it breaks its bindings (stage 7). The proxy starts another transaction (τ_5). Any server failures detected during τ_2 can be removed, and the *decrement* operation of the Server database is called to remove this client.

The failure of the client node after the binding has been completed (stage 2), or prior to breaking the bindings (stage 6) will not automatically undo any side-effects at the GVD created by the proxy; these are undone during the recovery of the node (described later).

4.2 Client proxy

The CORBA model for remote object invocations is based upon a client using a proxy for the remote object, and this proxy communicating with a single server counterpart, which then invokes operations on the real object. There is no CORBA standard to enable a client to invoke an operation simultaneously on multiple objects. However, most ORBs provide a capability to override the client proxy implementations, and *Jydra* uses this to provide transparent replication. Our current implementation runs on Visibroker [19] and ORBacus [20].

The *Jydra* proxy can turn a client invocation into multiple invocations on a series of CORBA objects, performed using the standard middleware invocation mechanism (shown in figure 6, where dotted boxes are used to indicate process boundaries). This technique is used in the default implementation of our toolkit for invoking replicated GVD. The proxy can also be programmed to interface to a process group service for invoking object group.

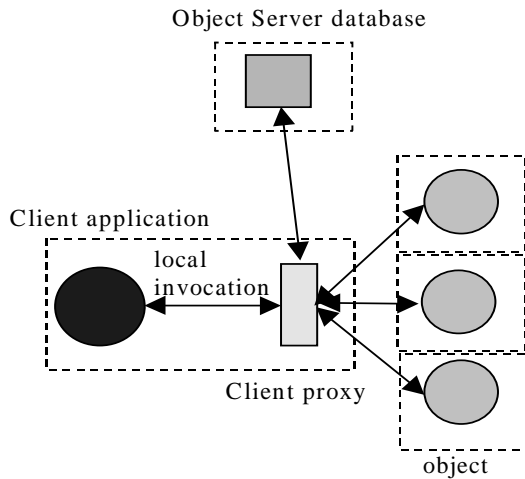


Figure 6: *Jydra* replica client proxy

4.3 The Object State database

This database also maintains two lists per object: (i) for an object A , its *available state list* containing the references of the nodes $\in St_A$; (ii) for each node $\in St_A$, a *use list*, which maintains information about the identity of the servers currently bound to it. An object A is not in use if, for each node $\in St_A$, the corresponding *use list* is empty, and likewise it is unavailable if St_A is empty:

```

module Jydra
{
  interface ObjectStateDatabase :
    CosTransactions::TransactionalObject
  {
    sequence<string> getState (in UID stateName,
                              in UID server);
    void exclude (in UID stateName,
                  in sequence<string> failedHosts,
                  in UID server);
  };
}

```

The operations are:

- *getState(stateName, server)*: returns the *available list* for *stateName*. As a side effect of this call, the use list of *stateName* is updated to include the server.
- *exclude(stateName, failedHosts, server)*: removes the specified hosts from the *available list* of *stateName*, i.e., marks them as out-of-date. The specified server is also removed from the *use list*.

4.4 Replicated object stores

Recall that transactional objects in *JTSArjuna* are derived from the *StateManager* class, which uses a persistence service implementation through an interface object. This indirection allows implementations to change without requiring changes to *JTSArjuna* or its applications. Therefore, to add state replication, *Jydra* provides a replication specific persistence service (shown in figure 7). This implementation works in a similar manner to the client proxies described above, invoking each persistence store serially, given the St_A information.

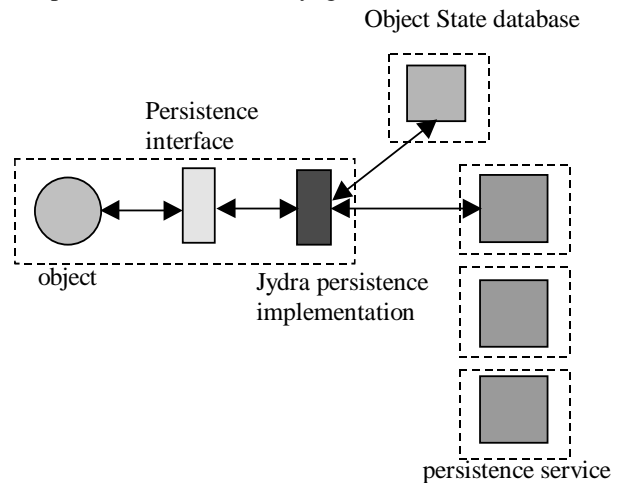


Figure 7: *Jydra* replicated persistence implementation.

4.4.1 Using the State database

Within the application's transaction (τ_2) each replica must acquire its state (stage 3 in figure 5). *StateManager* uses the *Jydra* persistence implementation, which starts a new top-level transaction (τ_3) and from within this invokes the State database operation *getState* to obtain the

St_A information. The replica then accesses the object store of any of the specified nodes. If failures of any of these nodes are detected, their identities are remembered by the replica (complete failure forces τ_2 to rollback).

Commit processing of τ_2 involves: (i) sending a 'prepare to commit' invocation to all objects; (ii) a *modified* replica then copies its state to all of the relevant object stores; assuming some object stores are still available, the replica prepares an *exclude list* of nodes where the state could not be updated (including nodes which could not be contacted during τ_3), and launches a transaction τ_4 (stage 5) which invokes the *exclude* operation of the State database to remove these nodes. If τ_4 fails, the replica cannot commit τ_2 and informs the coordinator, which will roll back the transaction. If the application transaction decision is to rollback, or the object state was not changed, then no changes to the State database are required.

4.5 Replicating the GVD and tolerating partitions

Jydra replication protocols require that the GVD is available at all times. This can be realistically met by replicating it as well. To eliminate the requirement of the GVD itself requiring a binding system for replicated objects, two simplifying restrictions were imposed: (i) a node storing the states of GVD objects also runs the servers for them; (ii) every client knows the locations of the GVD replicas. *Jydra* uses a quorum-consensus protocol to cope with partitions [21]: a majority of the GVD replicas must be available to an application in order for it to execute. If a partition occurs, only clients in the majority GVD partition (if one exists) will make forward progress. An alternative would be to use the name server replicas as a process group along the lines already indicated.

4.6 Node recovery

A node recovering from a failure first completes any outstanding transactions whose processing was interrupted; it then contacts the GVD to remove any entries kept for this node in any of the *use lists*: these entries record pre-crash usage information that is now out of date. The node then checks with the Server database to see if it has been excluded from acting as a server for any objects. If so, it may modify this information to resume serving for such objects. A node that has been excluded as an object store can likewise use the State database information (after having updated the states of excluded objects: information on up-to-date states can be obtained from the database). Finally, since network partitions are possible, a node occasionally checks if it has been

excluded in error, and if so, includes itself as discussed above.

5. Related work

Object replication using the process group approach has been studied extensively. [22] discuss system issues in object replication using this approach, discussing in particular how to move away from 'process' centric groups to 'object groups'. The paper describes how a CORBA object group service[12] can be utilised for supporting replication. The Eternal system [23] offers replicated, non-persistent CORBA objects through a similar proxy mechanism to *Jydra*'s. The system supports both active and passive replication and uses a group communication protocol to guarantee delivery and ordering of messages; however, such messages are not delivered by the ORB but by a separate system called Totem. This requires Eternal and Totem to be more closely integrated with a particular ORB implementation. Looking at the system structure shown in figures 1 and 2, these approaches essentially cover replica management issues of a server group, not addressing object activation and binding issues covered here. Indeed, as described here, our system design is capable of exploiting the above approaches for server group management. Filterfresh is a replication system for RMI, Java's own distributed object infrastructure [24]. The system provides active replication for the non-persistent RMI naming service, and relies upon a virtual synchrony group communication mechanism. Replication of application objects is not available.

Transactions and process groups are often seen as rivals; for example a recent paper suggested that transactional systems are a better alternative to process groups [25]. There have been attempts to 'unify' the two models, for example[26] describes how to enrich group communication with some transactional flavour. In a recent paper we have argued that the two approaches can be seen to be complementary [27]. The design described here strengthens that argument. An early implementation of active replication of Arjuna transactional objects is described in [28]; it implemented all of the GVD functionality described here.

6. Conclusions

With the advent of Java the range of Internet applications has grown, and with it their requirements for fault-tolerance. Ideally a fault-tolerant system should provide both consistency and forward progress in the presence of concurrent users and failures. We have described a toolkit which integrates standards compliant transactions and middleware solutions with object replication. Rather than provide a single replication

protocol, *Jydra* has been designed to support both active and passive replication. The default implementation of *Jydra* supports single copy passive replication and does not require any support of a group management system. More sophisticated forms of replication requiring server groups will need support from process/object group services; *Jydra* has been designed to work with them.

Acknowledgements

The work reported here has been supported in part by a grant from UK Engineering and Physical Sciences Research Council (grant no. GR/L 73708) and ESPRIT working group BROADCAST (project No. 22455).

References

- [1] Java Development Kit 1.2, Sun Microsystems, November 1998.
- [2] "CORBA services: Common Object Services Specification", OMG, 1997.
- [3] 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>)
- [4] M. C. Little, D. McCue and S. K. Shrivastava, "Maintaining information about persistent replicated objects in a distributed system", ICDCS-13, Pittsburgh, May 1993, pp. 491-498.
- [5] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication", CACM, 39 (4), April 1996, pp. 64-70.
- [6] K. Birman, "The process group approach to reliable computing", CACM, 36, 12, pp. 37-53, December 1993.
- [7] L. E. Moser, P. M. Melliar-Smith et al, "Totem: a Fault-tolerant multicast group communication system", CACM, 39 (4), pp. 54-63, April 1996.
- [8] P. D. Ezhilchelvan, R. Macedo and S. K. Shrivastava, "Newtop: a fault-tolerant group communication protocol", 15th IEEE International Conference on Distributed Computing Systems, Vancouver, pp. 296-306, May 1995.
- [9] F. B. Schneider, "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial", ACM Computing Surveys, 22(4), December 1990, pp. 299-319.
- [10] P. A. Alsberg and J. D. Day, "A Principle for Resilient Sharing of Distributed Resources", Proceedings of the Second International Conference on Software Engineering, 1976.
- [11] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
- [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] "The Common Object Request Broker Architecture and Specification", OMG, February 1998.
- [15] V. Matena and M. Harper, "Enterprise JavaBeans", Sun Microsystems, November 1998.
- [16] M. C. Little and S. K. Shrivastava, "Java Transactions for the Internet", Proceedings of the 4th Conference on Object-Oriented Technologies and Systems, April 1998, pp. 89-100.
- [17] G.D. Parrington, S.K. Shrivastava, S. M. Wheeler, and M.C. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), Summer 1995, pp. 255-308.
- [18] M. C. Little and S. M. Wheeler, "Building Configurable Applications in Java", Proceedings of the 4th IEEE International Conference on Configurable Distributed Systems, May 1998, pp. 172-179.
- [19] "VisiBroker for Java Programmer's Guide Version 3.2", Visigenic Software Inc., February 1998.
- [20] "ORBacus 3.1 Programmer's Manual", Object-Oriented Concepts Limited, 1998.
- [21] D. K. Gifford, "Weighted Voting for Replicated Data", 7th Symposium on Operating System Principles, December 1979.
- [22] 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.
- [23] P. Narasimhan et al, "Replica consistency of CORBA objects in partitionable distributed systems", Distributed Systems Engineering Journal, No. 4, 1997, pp. 139-150.
- [24] A. Baratloo et al, "Filterfresh: Hot Replication of Java RMI Server Objects", Proceedings of the 4th Conference on Object-Oriented Technologies and Systems, April 1998, pp. 65-78.
- [25] D. R. Cheriton and D. Skeen, "Understanding the limitations of causally and totally ordered communication", Proceedings of 14th ACM Symposium on Operating Systems Principles, Operating Systems Review, 27 (5), pp. 44-57, December 1993.
- [26] A. Schiper and M. Raynal, "From group communication to transactions in distributed systems", CACM, 39(4), April 1996.
- [27] M. C. Little and S. K. Shrivastava, "Understanding the Role of Atomic Transactions and Group Communication 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.
- [28] M.C. Little and S. K. Shrivastava, "Replicated K-resilient objects in Arjuna", in Proceedings of the 1st IEEE Workshop on the Management of Replicated Data, Houston, November 1990, pp. 53-58.