

Application Layer Software Fault Tolerance for Distributed Object-Oriented Systems*

Hyun C. Kim and V.S.S. Nair

Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275, USA

Abstract

A technique for software fault-tolerance on a distributed object-oriented system is presented. It provides software fault tolerance as well as resilience to the object failures. The technique is an extension of the active object replication schemes. Using the property of encapsulation, a group of objects with multiple implementations sharing a common interface can be treated as if they are a group of replicated objects. The resilience from hardware failures is achieved by the replication of computation. The software fault tolerance is attained by the design diversity. The application layer implementation of the technique is discussed.

1 Introduction

Object-oriented programming(OOP), using its properties of abstraction, encapsulation, inheritance, and polymorphism, offers many advantages over a conventional system [7]. The computational model where encapsulated objects pass messages to accomplish tasks, maps very well with distributed systems. With the emergence of distributed object-oriented technology, rapid development of software by connecting distributed objects is becoming routine [14].

In most of the previous research on distributed object-oriented systems, researchers considered message passing objects that are distributed over different computing nodes connected by local area network (LAN) [4, 13]. In these studies, key issues such as object location and activation, inter-object communications, and synchronization have been addressed: Object location and activation problem involve finding a collection of object using a generic name and instantiating the specified objects from permanent storage. Inter-object communication addresses the method invocations between objects in different nodes so that it appears as if all objects reside in the same node. Most systems use remote procedure call (RPC) for this purpose. Synchronization is addressed by ensuring executions of methods within the object in correct order so that the objects are in consistent state.

Atomic transactions and nested atomic transactions are used in many object-oriented systems to guarantee serializability, failure atomicity, and permanence of effects thus, providing fault tolerance by means of roll-back recovery [13]. Some researchers considered forward error recovery technique using active replication of objects [5, 9]. In active replication, multiple copies of an object are concurrently executed in different nodes. Thus, using the active replication strategy, systems can tolerate up to $k-1$ node failures given it has k replicated objects. The strategy requires ability to treat the replicated objects as if they are single entity. They are addressed using a single logical name; they are in identical states; and objects that become inconsistent are screened out. Again, many of these issues have been addressed previously by using the concept of object groups and reliable multi-casting [3]. Although replicated objects do provide tolerance against hardware faults, they do not provide protection from software design faults.

Simple replication of software components does not provide software tolerance since design defects will also be replicated. Use of multiple implementation for software components is the choice of the software fault tolerance techniques that are used in N-Version programming system (NVPS) [2] and Recovery Block system (RBS) [12]. For distributed systems, distributed recovery block technique(DRB) uses redundant computation resources to provide effective protection against both hardware and software failures [8]. In DRB, a pair of self checking programs concurrently execute two different alternates to provide software fault tolerance as well as hardware fault tolerance. The fault tolerance is provided at the procedural level. Due to the function oriented design, application of DRB in object-oriented systems is limited at best.

In this paper, we present a technique that provides both software and hardware fault tolerance in a distributed object-oriented environment. It uses encapsulation property of objects to extend the active replication technique by providing more than one version of any given object. The encapsulation property enables an object to be described in terms of its external behavior (interface) without revealing its internal structure (implementation). Two objects that share a

*Supported in part by a grant from Bell Northern Research, Inc.

common interface but different implementations appear to be identical (replicated) objects since their external behavior would be the same. Multi-version objects can be treated as if they are replicated objects thus techniques for replicated objects can be used with minor modifications. In addition to the fault tolerance provided by replication, the multi-version objects can add protection from software design faults using its design diversity. We describe an implementation at application level without hardware or system level support.

This paper is organized as follows: In the second section, we introduce the theoretical model of the distributed object-oriented system. Then we describe the computational environment where the technique can be implemented. In section three, we explain the basic concepts of multi-version object and how it differs from the traditional software fault tolerance techniques as well as the object replication strategies. Then we discuss the requirements for implementing multi-version objects and the details of how they were addressed. In section four, we describe example applications where the technique is applied. Finally, the paper is concluded.

2 Model of Computation

The theoretical model used in the paper is based on an object-oriented computation model proposed for ABCL/1 [15]. In this model, software can be constructed by specifying set of objects that interact among themselves by sending and receiving messages. Each object is assumed to have its own computational resources as well as its own local memory. The content of the local memory represents a state of an object at a given time. The state of an object changes when the object receives a message. Whenever an object receives a message, it executes a sequence of computations in response, some of which may in turn send messages to other objects. An object may, after sending messages to other objects, suspend its own computation and wait for the response (synchronous mode) or may continue its own computation without waiting for a response (asynchronous mode). There are two types of messages; one that modifies the state and one that probes an object for current state. The collection of objects can be partitioned into clusters of objects. Each cluster is responsible for implementing a part of the software features.

Based on the theoretical model, we present the following computational model: The distributed environment consists of networked nodes of heterogeneous workstations. The objects may or may not reside in the same node. However, all objects have different threads of computation even if they are in the same node. All nodes are assumed to have stable storage area where persistent versions of the objects reside. Some objects are composed of several smaller objects. However, in this case, all sub-objects share the same thread of computation. These objects encapsulate several sub-objects to form an abstract entity. Distributed objects can play the role of either a client object or a server object. The client object requests services from one or more server objects by sending

messages. The server object processes computation in response to the requests from clients. Some server objects may in turn request services from other server objects thus themselves becoming client objects.

Following assumptions are made about faults in the system; Network communications are assumed to be reliable, and a node failure does not lead to a network partition. Nodes exhibit fail-silent property; that is, either nodes work correctly or they crash without affecting neighboring nodes. Recovery of crashed nodes is possible using log files saved in stable store. The faults that do not cause crash of a node can be detected using acceptance test built into objects. The acceptance test will take advantage of application specific properties on each type of object if it can, if not, it relies on generic checking techniques. The faulty objects are removed from service immediately when fault is detected.

3 Multi-version Objects

Active object replications to tolerate node failure have been implemented in many distributed object-oriented systems [9, 5]. The active replication is used in object-oriented system since objects encapsulate the data and code. In [9], the authors noted that “problems of managing replicated objects really amounts to that of managing replicated computations”. The replication of computations can be naturally extended into multi-version computation by applying multi-version strategies from NVPS or DRBS.

The key issue in concurrently replicated computations, as in NVP or DRB, is to ensure that all alternates are processing same input data at any given moment. For this purpose, these techniques provide synchronization mechanisms. The mechanism is easily implementable since all alternate modules share a common input/output pattern. Similarly, we need some common mechanism to synchronize concurrently executing objects. However, unlike functions, objects do not have the concept of input and output. Instead, they use messages to communicate. We can take advantage of encapsulation in object-oriented systems for this purpose. Using encapsulation property to separate implementation details from interfaces is a common object-oriented programming technique used to improve reusability and maintainability of software. For our purpose, the separation of interface from implementation provides a way to address a group of concurrently executing alternate objects as a single entity.

The multi-version objects (*MVO*) can be defined as follows: A collection of active objects that have a common interface but different implementations. By interface, we mean a list of messages that an object receives and processes. By implementation, we mean a list of data, used as building blocks, and a list of methods, each consisting of sequences of instructions that will be executed in response to one of the messages the object receives. The common interface presents a uniform behavior to the outside world. That is, a group of *MVO* can be addressed as if they are a single entity. The separation of implementation provides protection from common software design defects.

Acceptance test (AT) is used to detect faults that do not cause crash of the object. When a fault is detected, AT notifies the object's group that an error has occurred. The group updates membership information to remove the faulty object. AT detects occurrence of error(s) by recognizing an object is in inconsistent state. Using assertions to specify preconditions and postconditions on software to improve its reliabilities has been proposed long ago [6] and extension of this idea to object-oriented programming, using class invariants to detect occurrence of inconsistent states, has been proposed [10]. Each member of *MVO* may or may not have different AT, depending on whether implementation specific or domain specific properties are used to detect inconsistent states. AT is applied to a member of *MVO* whenever a method that changes internal state of the object, is completed. In many applications, coming up with a good AT is difficult. For these cases, general assumptions are made about the types of faults that can occur and based on them, a generic algorithm is developed. We could also use voting strategy to detect software faults.

Methods implementing persistence of the object are used to recover from node crashes. There are two different types of methods. One type is used to save and restore an object as a whole. Since each member of the *MVO* may have different implementation and have different structure, the methods must be customized for each object. They can be used to activate objects from secondary storages and to save the objects to the secondary storages when objects are no longer needed. For this study, we assume that all objects have been activated before the program starts. Thus, issues of object activation and de-activation are not considered. The second type of method used to log all message invocations to *MVO*. The log is used to recover crashed objects.

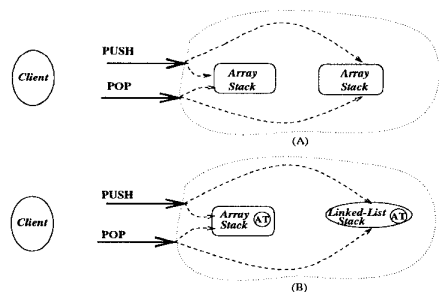


Figure 1: The Replicated Object Group and Multi-version Object

In figure 1, similarity between a replicated object system and *MVO* system is illustrated. From a client's point of view, both replicated objects and *MVO* objects are a single entity. The client sends messages to interact with a logical view of the server object. Within each group, all members process the same message. Thus, a client does not need to know that the

stack object it uses is in fact a group of replicated objects or for that matter, it is a group of objects that have different implementations.

3.1 Requirements

Distributed object-oriented systems that support replicated object should meet the following requirements;

- The replicated objects should behave as a single entity.
- Failed members of the group should be removed from the group.
- A method invocation on a group by multiple clients should be processed in uniform order.

Addressing the requirements for replicated objects has been studied by various researchers. [9, 5]. The first requirement is addressed by using the object group concept. The object group can be treated as a single entity by keeping a global look-up table where a group name and membership information are kept. In order to maintain the appearance of single entity, all members of the group should be in identical states. This implies that the message passing scheme should have atomic property; either all members receive messages or none do. The second requirement can be achieved by comparing a list of *MVO* members to which a message was sent and a list of the *MVO* members from which the response for the sent message was received [9]. For the third requirement, lock/unlock on common resources can be used to synchronize message delivery. Arjuna, for example, uses this approach [9].

In addition to the above requirements, *MVO* should address issues of design diversity, object recovery and design fault detection.

Providing enough design diversity among the implementations is similar to problem in NVS where sufficiently diverse implementation must be provided from the same specification [2] or in RBS where alternate algorithms must be provided for the same problem [12]. We must use application specific knowledge to come up with good alternate implementations.

In replicated object systems, restoring a crashed member is straight forward; information on one of the members is copied on to the object to be restored. In *MVO*, since all members of the group may have different internal state due to different implementations, we cannot use simple copying scheme. Instead, we need another way to restore the state of an object. One way is by keeping a log of all method invocations done for each *MVO*. An object to be restored invokes messages in the log until its external state is in synch with the rest of the group. In order to improve the performance, we could use checkpointing technique. In this case, the execution of the log can start from the last stable state of the object. An alternate way is by using a special function that can extract necessary information from an active object. However, for this to work, each member must have a mean to access the internal information. Providing access function for the internal states is not always desirable and in many cases, it is impossible.

In replicated object system, if there is a software design fault that does not cause node failure, all members of the group would have the same error. Thus, there is no reason to provide error detection mechanism for faults other than for detecting crashed nodes. In *MVO*, we could recover from software design faults since *MVO* provides design diversity. Thus, an on-line error detection mechanism must be provided. The mechanism can make use of the object's application specific property to develop an acceptance testing. If no suitable property exists for use as acceptance testing, a generic error detection scheme or voting strategy can be used.

3.2 Design and Implementation

We describe a design scheme that can be realized at the application level without any system level support. The design described here should be implementable on networked unix workstations. In addition, any object-oriented distributed system that already supports replicated objects [9, 5], *MVO* can be implemented with minimal modifications.

We use RPC to implement message passing between objects. We modify client stub and server stubs for this purpose. The client stub is responsible for locating server objects via global naming service, sending messages to the found objects by concurrently initiating RPC for each server object in the list, receiving the responses from server objects via server stubs, updating membership of *MVO* by removing detected faulty members. The server stub is responsible for detecting faults in the server object. We use a global look-up table to keep track of *MVO* membership information. For now, let us assume that the client of the object group is a single object not a group of objects. When the client invokes a method on a *MVO* using a logical group name, a client stub is created. It first acquires the member list of the group from the look-up table, then it sends a polling message to each node in the list. When a computational node receives the message, a server stub is used to send back either a confirmation after locating a server object or a failure indication if the server object could not be located. The client stub waits until either it receives replies from all member list or until a preset timer expires. When the timer expires, the client stub treats any members whose replies were not received as if they have crashed. The failed nodes are excluded from the member list. The client stub then sends actual messages to the remaining members.

Each server stub invokes appropriate method on the server object located at its node. The server object either returns the result of the invocation or if AT detects fault, an error indication to the server stub. Each server stub forwards the response back to the client stub. The client stub again waits until either it receives responses from all members or a preset timer expires. Based on received responses, the client stub again updates *MVO* member information in the look-up table. Then it selects one of successful result and returns it back to the client object.

Note that two different fault detection schemes are used. The first is by checking for failure to respond to

the client stub in predefined time. The server stubs must respond to the initial locate request message from the client stub within a predefined time. Then they must respond to the client stub within a predefined time after receiving actual message. Any object that could not meet the deadline is removed from the group. It means either the object has crashed or it could not meet the real time constraint. The first scheme is used mainly for detecting hardware faults such as node failures or communication failures. The second way to detect the failure is by using acceptance test. Each object has built in AT which will be executed at the end of method invocation. If an error has occurred while processing received message, the AT may detect the failure and an indication will be sent. Otherwise, the object will send the result back to the client stub. In addition, occurrences of unexpected conditions can be raised by executable assertions planted through out the code. Using the second scheme, some of the software design faults can be handled.

If more than one client is trying to invoke a method on an object group, the consistency of the object group may be violated. Some members of a *MVO* may process one client's request and others may process the other client's request. In order to prevent this, we use lock/unlock on *MVO* membership information. Before a client stub can acquire a member list of a *MVO*, it must set a lock on the *MVO* entry in the look-up table. Other clients cannot access the member list of that *MVO* until the lock is released. Thus, only one client at a time can send messages to a *MVO*. When the client stub receives the result from the object group, it updates the member list and releases the lock. Thus, the lock on *MVO* is held for the duration of a method invocation. To prevent occurrences of dead lock, a client stub is restricted to hold only one lock at a time. This restriction is not unrealistic if the client object is a sequential process that executes one method invocation at a time and since we use RPC to implement message passing, all message invocations are synchronous. To avoid situation where a client that has crashed from holding on to a lock thus preventing other clients from accessing a *MVO* indefinitely, the lock will be released by the look-up table if it does not receive any indications from the owner within a predefined time.

Many of the mechanisms described depend on the global look-up table. To prevent the look-up table from becoming a single point of failure, we must replicate it. The replication scheme is similar to the regular situation. However, unlike normal replication, the membership list and location of each replicated look-up table are pre-defined. Thus, all client stubs can directly invoke messages to all members of the look-up table group. The message invocation to the look-up table group must be guaranteed of atomicity by the client. The client takes the similar steps that were taken by a client stub when a message was passed to a regular *MVO*.

Figure 2 illustrates the overall design of a *MVO* system. In the system, there are three replicated global look-up tables, a *MVO* consists of two objects each

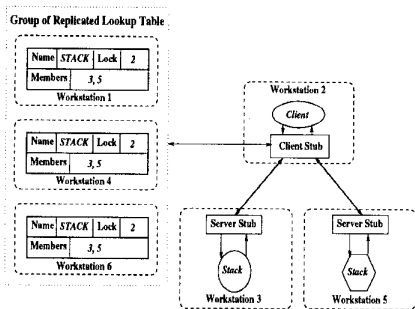


Figure 2: Multi - Version Object Based System

with different implementation, and a client object that uses the *MVO*. The client stub asks the look-up table group for membership information of *MVO* named *stack*. When the client stub finds that the entry for *stack* is available, it locks the *stack*'s entry then gets the member list and their locations from the entry of one of the replicated look-up tables. In the figure, the locations of two server objects are workstation 3 and 5. The client stub sends the message to the server object via each server stubs. Each server object returns the result of computation or if an AT detects a software fault, notification is returned to the client stub. Accordingly, the client stub updates the *MVO*'s member list in the look-up tables and unlocks the *MVO* entry.

Once a member is removed from *MVO*, it tries to recover from the fault by accessing a log file. The log contains the history of all message invocations done on the *MVO*. The recovering member may start from an initial state, processing all messages that are recorded in the file if the node has completely failed. If check-pointing was done within the node, then recovery can start from the last saved state. The file may be replicated in more than one stable storage located through out the network. It is the responsibility of a client stub to update the log whenever it successfully process message invocation (i.e., new checkpoint has been established). Techniques to implement atomic transactions such as discussed in [13] can be applied to keep the consistency among distributed message logs. Although we could recycle the history log, for example, by using time stamps on the record, it is not important since the life spans of *MVO* objects are relatively short. If a crashed node execute s all method invocations in the log then it is now considered as recovered. When an object is recovered, it sends the look-up table a message to notify that now it can rejoin the *MVO* membership.

In situations where the client object itself may be a logical group, *MVO* can be supported by converting one of the members of the client group to a client proxy. The rest of the group will receive the response from the client proxy after the proxy successfully makes message invocation to the *MVO*. If the look-up table detects the crash of the proxy client, instead of simply unlocking the entry in the look-up table and terminating the computation, another mem-

ber of the client group assumes the role of the proxy and computation is attempted again. Note that there cannot be a *MVO* client since members of *MVO* have different implementations and thus, each member may not use the same servers.

4 Examples

In this section, examples of *MVO* are presented.

4.1 Universal Pattern Analyzer

When a list of unformatted text data is received from a client, Universal Pattern Analyzer (UPA) formats the text based on pre-configured information, sorts the processed data and builds distribution of the input, and computes average, the maximum and minimum values. For many applications, the UPA has to meet a strict dead-line or if it cannot, at least it should provide the best approximated values as possible. The UPA can be used as a component in any distributed application software where a large amounts of unformatted and unpredictable data must be processed in a short time.

For example, UPA could be used to build a financial software for the stock market. The software will constantly monitor the changes in the prices of the stocks and displays the interesting patterns (big jumps or drops in prices) to stock brokers. For this application, UPA has to perform reliably even if the input data may be unpredictable. We address this problem by noting that depend on input patterns, a different sort scheme [1] is more suitable than others. For example, if the range of the values in the input is small, *bucketsort* can quickly sort large amount of data without storage overhead. However, if the range of values is big or all or most of values are uniquely distributed, then *quicksort* has less memory overhead and generally more efficient. In the figure, UPA is composed of two different objects. One uses *bucketsort* as its implementation and the other uses *quicksort*. When a sequence of data whose range is very large is sent to UPA, the object that uses *bucketsort* may not process sufficient amount of data. However the client may be able to receive a better answer from the object using *quicksort*.

For AT, we can use a cutoff values to determine the failure of the individual objects. The members of UPA are considered as failed object when its computed value does not meet the predetermined requirement. The cutoff value can be a percentage of the data that has been sorted or an application specific value that UPA produces. For example, for the financial software package, the user configurable value for the smallest acceptable price changes can be used as the cutoff values. If the maximum price change value computed by an object is smaller than this value, the object is considered to have failed the AT.

4.2 Matrix

MVO representing N-dimensional matrix can be built as follow. The *MVO* consists of two objects with different implementations. As shown in the Figure, the *column* object stores the elements of the matrix in column major. The *row* object on another hand, stores

the elements in row major. The MVO supports as internal methods, an operation to access each element by its indices and encoding functions. These are private methods that are not part of external interfaces and are used to help implementing external interfaces. The MVO support as external interfaces, the computational operations *add*, *subtract* and *multiply*. These operations take as their argument, a reference to another MVO. For object *column*, the operations *add* and *subtract* are performed column wise. For object *row*, they are performed row wise. Note that we use the access operator to get values from the argument MVO while computing each individual elements. For operation *multiply*, the *column* object uses the regular matrix multiplication whereas the *row* object uses the Strassen's algorithm [1]. Using this MVO, we achieve both structural and algorithmic design diversity.

For this application, we apply techniques from algorithm based fault tolerance (ABFT) for matrix computations [11] to come up with AT that can be used for addition, subtraction and multiplication operations. The technique consists of adding one additional row or column for a matrix. The added information encodes a property of the whole matrix that is transitive to the aforementioned operations. Thus after the computations, the encoded property is checked to detect occurrence of faults. For example, given two matrix to be computed, we use the sum of elements in each column to form an additional row for the first matrix. For the second matrix, we use the sum for each row to generate an encoded column. After any matrix computations, we verify whether the new result for the encoded row and column still represents the sum of the calculated elements.

5 Conclusions

We presented a new software fault tolerance scheme for distributed object oriented environments. The technique is based on *MVO*, a logical entity that consists of a collection of objects sharing a common interface but each having different implementations. The common interface guarantees consistency among the members of the same group since the common interface enforces the common external behavior of the object. Thus, It provides resilience in the case of complete node failures much like a replicated system would do. However, *MVO* provides distinct advantages over a group of replicated objects since its multiple implementation provides the design diversity which enables *MVO* to tolerate the software design faults. *MVO* can be implemented at application level with minimal system level supports. Also, systems that already support active object replication can implement *MVO* with minor modifications.

References

- [1] A. Aho, J Hopcroft, and J Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, third edition, 1976.
- [2] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):458–464, December 1985.
- [3] Kenneth. P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.
- [4] P. Dasgupta. The design and implementation of the clouds distributed operating system. *Computing Systems*, 23(1), March 1989.
- [5] O. Hagsand, H. Herzog, K. Birman, and R. Cooper. Object-oriented reliable distributed programming. In *Proc. of the Second International Workshop on Object Orientation in Operating Systems*, pages 180–188, 1992.
- [6] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.
- [7] Ivar Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering:: A Use Case Driven Approach*. Addison-Wesley, 1992.
- [8] K.H. Kim and H.O. Welch. Distributed execution of recovery blocks: an approach to uniform treatment of hardware and software faults in real-time applications. *IEEE Transactions on Computers*, pages 626–636, May 1989.
- [9] M. Little and S. K. Shrivastava. Replicated k-resilient objects in arjuna. In *Proc. of IEEE Workshop on the Management of Replicated Data*, pages 53–58, Houston, Texas, November 1990.
- [10] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, pages 40–51, October 1992.
- [11] V.S.S. Nair and Jacob A. Abraham. Real-number codes for fault-tolerant matrix operations on processor arrays. *IEEE Transactions on Computers*, 39(4):426–435, April 1990.
- [12] B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering*, pages 220–232, June 1975.
- [13] Santosh Shrivastava, Graeme Dixon, and Graham Parrington. An overview of the arjuna distributed programming system. *IEEE Software*, 10(3):257–267, January 1984.
- [14] J. Stikeleather. Why distributed object computing is inevitable. *Object Magazine*, 4(1):34–37, March 1994.
- [15] A. Yonezaw, E. Shibayama, T. Takada, and Y. Honda. Modelling and programming in an object-oriented concurrent language abcl/1. In Akinori Yonezawa and Mario Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 55–89. The MIT Press, 1986.