A nested invocation suppression mechanism for active replication fault-tolerant CORBA

Deron Liang[1] **and Chen-Liang Fang**[2]

[1]Institute of Information Science, Academia Sinica, Taipei, Taiwan, 11529, R.O.C.

[2]Department of Information Management, Jin-Wen Institute of Technology, Taipei, Taiwan, R.O.C.

Keywords: fault-tolerance, CORBA, distributed computing environment, active replication

Abstract

Active replication is a common approach to building highly available and reliable distributed software applications. The redundant nested invocation (RNI) problem arises when servers in a replicated group issues nested invocations to other server groups in response to a client invocation. Automatic suppression of RNI is always a desirable solution, yet it is usually a difficult design issue. If the system has multi-threading (MT) *support, the difficulties of implementation increase dramatically. One can design a deterministic thread execution control mechanism, but there is a drawback of this. Commonly, modern operating systems implement thread scheduler on kernel level for execution fairness. Unfortunately, in this case, modification on the thread scheduler implies modifying the operating system kernel. This approach loses system portability which is one of the important requirements of CORBA and other middleware. In this work, we propose a mechanism to perform auto-suppression of redundant nested invocation in an active replication fault-tolerant* (FT) *CORBA system. Besides the mechanism design, we discuss the design correctness semantic and the correctness proof of our design.*

1 Introduction

With the advance of computer and communications technology, distributed computing systems have become increasingly popular in recent years. Many of these distributed systems are designed to perform critical tasks in a hazardous environment [1]. Active replication techniques are commonly used to build critical software systems to ensure their reliability and availability [12]. Modern large-scale distributed applications are usually built on distributed middleware to cope with design issues such as heterogeneity, scalability, and portability [4]. One of the popular middlewares is CORBA, proposed by the Object Management Group (OMG)[14]. OMG recently announced a specification called *Fault-tolerant CORBA* [17] that recognizes the importance of fault tolerance. One of the open issues that OMG pointed out in its RFP of *Fault-tolerant CORBA* is the *redundant nested invocations problem* (RNIP) [18].

A nested invocation refers to an invocation on a server *B,* from another server *A,* upon an invocation on *A*. The RNI problem arises when a group is serving a client invocation and replicas in this active replication group all make the same (redundant) nested invocations to another server. An example is used in Figure 1 to illustrate the RNI problem in more detail. Figure 1 shows an active replication group *A* that is configured with two active replicas A_1 and A_2. Suppose that an invocation *A->do*() arrives at group *A*. This invocation later triggers two nested invocations, namely, *V->addV*(2) and *U->addU*(1) shown in Figure 1, one on each server *U* and *V*. Server *V* will receive two identical invocations, one each from A_1 and A_2. Because they are identical in *A,* it is clear that these two nested invocations are redundant to server *V*. Similarly, server *U* will face the same problem. Redundant invocations could cause inconsistent states, particularly if such requests lead to state changes. OMG's RFP of Fault-tolerant CORBA [18] advocated

the installation of a *suppression mechanism for redundant nested invocation* (**SM)** on active replication groups, as shown in the dotted box in Figure 2. The purpose of SM is to ensure that only one of the redundant nested invocations is allowed to be forwarded to the server. In other words, this mechanism identifies all redundant nested invocations first, and then suppresses all of them but one. Figure 2 depicts an effective SM where *V->addV*(2) from A_2 is suppressed, since its equivalent RNI from A_1 has been sent to server *V* earlier. By the same token, the nested invocation *V->addV*(1) from A_1 is suppressed. Furthermore, this mechanism should deliver invocation results to every member in the replicated group.

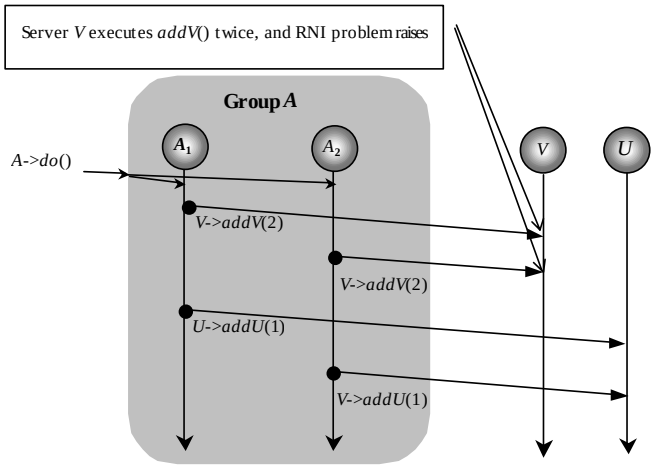

Figure 1. The RNI problem.

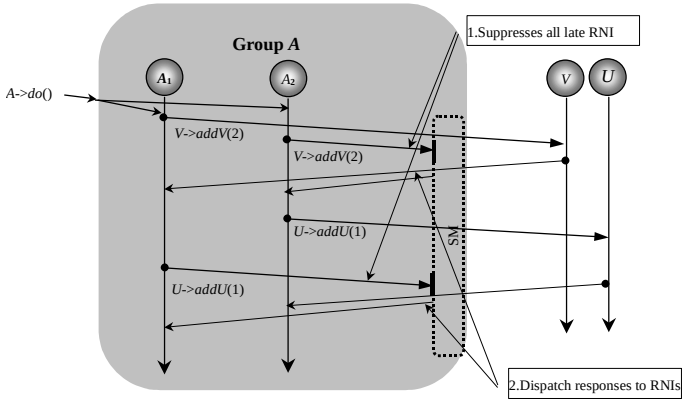

Figure 2. The desirable features of SM.

Suppose that a server serves one invocation at a time (known as the per-object invocation model in ORB). All servers are implemented as single thread, and the execution of each replicated server is deterministic. It is readily seen that all nested invocation sequences from replicated servers are identical given a client invocation to the group. Suppose the SM can incrementally assign a number, shown in the small box in Figure 3, to a nested invocation from a server in the group. For instance, the first nested invocation *V->addV*(2) from server A_1 is assigned the value 1 as its nested invocation ID. Invocation IDs are shown in the box in Figure 3. The SM can detect that a nested invocation is redundant if its ID has appeared before. For example, the nested invocation *V->addV*(2) from A_2 is blocked at the SM because it's equivalent from A_1 has been sent to server *V*. Similarly, the nested invocation *U->addU*(1) from A_1 is blocked at the SM. We can argue that the SM design solves the RNI problem given the assumptions. Mission-critical systems that deploy an active replication mechanism tend to be more performance sensitive in terms of response time, system throughput, real-time constraints, etc. Based on our experience [10], multi-threading (MT) implementation at both the middleware and application levels is an effective approach for addressing performance issues.

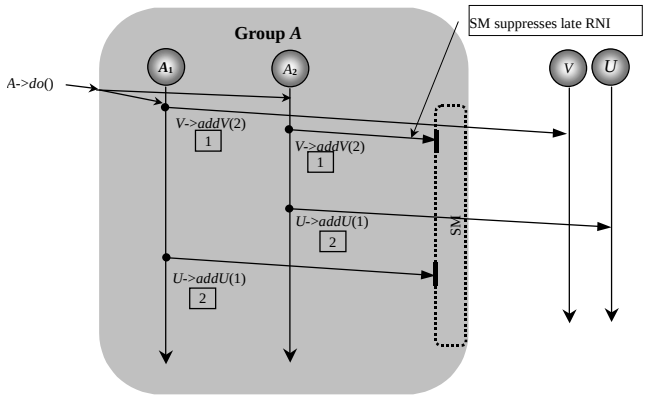

Figure 3. Simple solution for single threaded application RNIP.

However, MT implementation introduces randomness, adimension which makes the design and implementation of the SM more complicated [13]. Suppose that both A_1 and A_2 are implemented in MT. Their nested invocation sequences may not be identical even if their implementations are the same. Figure 4 gives one such example where each replica forks two distinct threads to interact with distinct targets *U* and *V*. The thread execution in A_1 may be different from the one in A_2. As a result, the nested invocation sequence of A_1, namely {*U->addU*(),*V->addV*()}, differs from that of A_2. If we use the same SM in this case, the SM assigns a different ID number to *V->addV*() from A_1 than to the one from A_2. The two RNI *V->addV*() are forwarded to server *V*. It is obvious that the solution offered in Figure 3 does not work in this case, as shown in Figure 4.

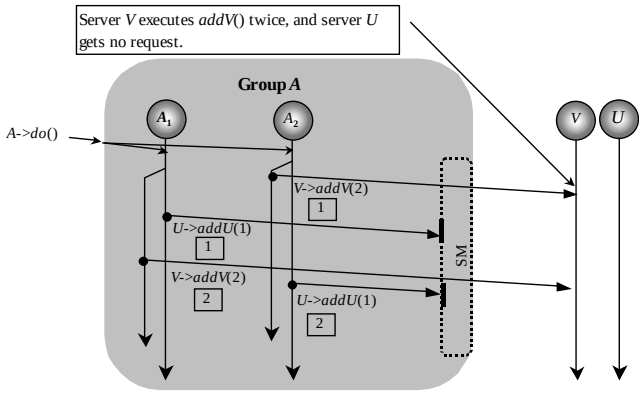

Figure 4. Sequential number ID fails to identify RNIs.

Narasimhan, et al. proposed an SM solution to address this RNI problem in their *Eternal* fault-tolerant system [13]. This SM involves the installation of a deterministic thread scheduler in the ORB kernel. This implemention ensures that all replicas in an active replication group produce an identical nested invocation sequence. As a result, the SM can distinguish redundant nested invocations by assigning sequence numbers to the nested invocations from each replica. Figure 5 depicts such a scenario. Some modern operating systems implement a thread scheduler on the kernel level for execution fairness.

In this case, modification on thread control implies modifying a operating system kernel. This approach loses system portability which is one of the important requirements of CORBA and other middleware.

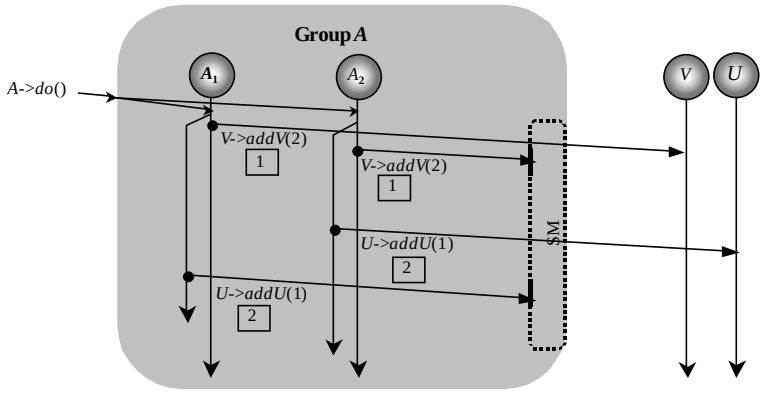

Figure 5. A deterministic thread scheduler solution for the redundant nested invocations problem.

Based on the above discussion, an application server with single-threaded implementation is unlikely to satisfy the high reliability performance requirement. Furthermore, Eternal system is not a promising solution for a portable SM in that portablility is one of the most desireable features for the CORBA community. In this paper, we propose a portable SM design for this multi-threading RNI problem. Our SM intercepts all nested invocations, and appends a header that contains all necessary information needed for RNI identification. We show in section 4 that this SM is able to detect all redundant nested invocations successfuly and efficiently. Our SM takes advantage of standard ORB functions, such as portable intercepter and portable object adaptor (POA), and a standard multi-threading library, *setspecific*()/*getspecific*() in the Open Software Foundation's Distributed Environment thread package. As such, we can ensure the portability of our SM.

The remainder of this paper is structured as follows. Section 2 uses an abstract group

model to define the RNI problem and the auto-suppression machanism is discussed. In section 3, we will discuss our SM prototype design in detail. The prototype implementation is given in section 4. Two performance experiments are designed to measure the overhead of the SM in section 5.. Conclusions and future works are discussed in section 6.

2 The auto-suppression mechanism of redundant nested invocation

In this section, we first introduce an abstract model that defines the notations used throughout this paper. We present the assumptions about this abstract model. We then give the formal problem definition based on the abstract model and the assumptions.

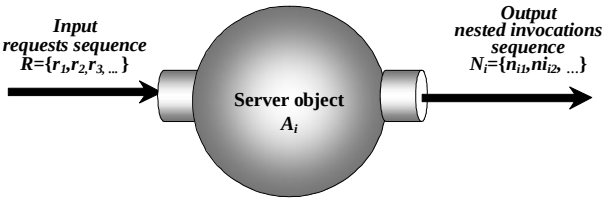

Figure 6. A single object server model.

The abstract model consists of two parts: the single object server object model, and the group object model. Figure 6 depicts the single object model for an object implementation. We assume a server object A_i has a main thread t_i that forks a distinct child thread to serve each arrival request $r_k \in R=\{ r_1, r_2, r_3, \ldots\}$ where R denotes the sequence of arrival requests. This child thread may fork other threads to interact with other servers if needed. This is when nested invocations can occur. We call this model the MT application (MT-AP) implementation. We let $N_i=\{n_{i1}, n_{i2}, \ldots\}$ represent the nested invocation sequence triggered by the arrival request sequence R.

For the group object model, Figure 7 depicts an active group of replicated object servers $A=\{A_1, \ldots, A_m\}$. Suppose these replicas serve the identical sequence of requests R, and each replicated server object produces it's a sequence of nested invocations, $N_1, \ldots, N_m$. We can assume that there are redundant nested invocations (RNI) among these sequences since the object servers $A=\{A_1, \ldots, A_m\}$ are replicated of each other. As shown in Figure 7, we assume there is an SM in place to detect RNIs, , and then form a *group sequence* of nested invocations $N=\{n_1, n_2, \ldots\}$ out of individual sequences N_1, N_2, …, *and* N_m by blocking all RNIs. SM then delivers the set N to corresponding target servers.

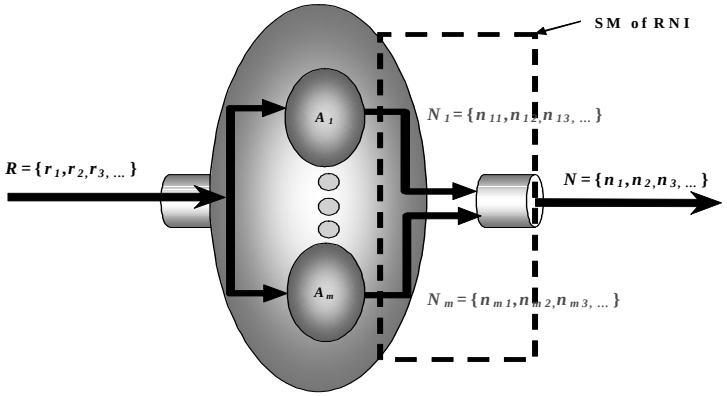

Figure 7. An active FT group model.

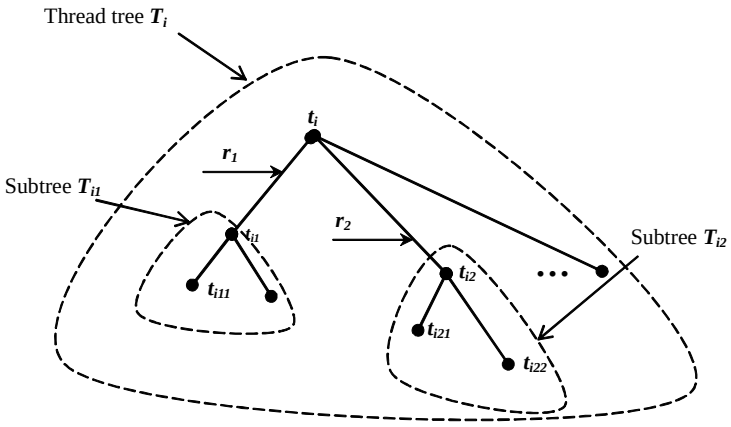

Figure 8. The thread tree of server object A_i.

A thread may fork child threads during execution. Each thread is forked from only one parent thread, i.e. each thread has only one parent thread. Naturally, all the threads of A_i (subject to R) form a thread tree T_i. Consider the example in Figure 8, all descending threads of thread t_i form a thread tree denoted by T_i. We let child thread t_{ik} of t_i represent the thread serving the incoming request $r_k \in R$, since we assume t_i forks a new thread to serve each incoming request. Thus the sub tree T_{ik} represent the thread tree generated to serve request r_k. Suppose thread $t_{iri_1i_2\ldots i_{n-1}}$ represents a thread in the thread tree T_{ir}. Then, the thread $t_{iri_1i_2\ldots i_n}$ denotes the i_n^{th} child thread forked from thread $t_{iri_1i_2\ldots i_{n-1}}$ during its execution; and this notation implies $i_a \in N$, $\forall a \geq 1$. For example, the sub tree T_{i1} and T_{i2} in Figure 8 serve the request r_1 and r_2 respectively. The sub tree T_{i1} has a root thread t_{i1} and two child threads t_{i11} and t_{i12}.

Our SM design is based on the above abstract group model and the following assumptions for an active fault-tolerant system.

Assumption 1: All replicated object servers $A_1, \ldots, A_m$ serve the same sequence of arrival requests R.

Assumption 2: The threads t_{ik} and t_{jk} are identical $\forall i, j$, denoted as $t_{ik}=t_{jk}$, since they all serve the same request r_k.

Assumption 3 identical thread behavior: If two threads $t_{iri_1i_2\ldots i_n}$ and $t_{jrj_1j_2\ldots j_m}$ are identical, then their child threads are piece-wise identical, i.e., $t_{iri_1i_2\ldots i_ni_{n+1}} = t_{jrj_1j_2\ldots j_mj_{m+1}}$ if $i_{n+1}=j_{m+1}$.

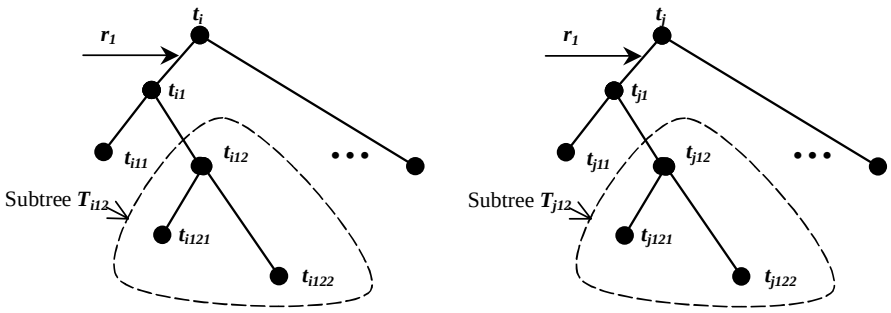

Figure 9. Identical thread behavior.

This assumption states that the child threads of two threads are piece-wise identical if the two threads are identical. An identical thread example is shown in Figure 9. If two distinct threads t_{i12} and t_{j12} on distinct server object A_i and A_j are identical, then the child threads t_{i121} and t_{i122} are identical to t_{1121} and t_{j122}, respectively. Based on assumptions 1~3, the thread tree of server objects have following property.

Property 1: Given $t_{ir}=t_{jr}$, $t_{iri_1i_2\ldots i_n} = t_{jrj_1j_2\ldots j_n}$ *if* $i_k=j_k$ $\forall k$=1, ..., *n*.

Proof**:** Proved by induction:

Basic step**:** Given Assumption 2 and **Assumption 3**, we have $t_{iri_1} = t_{jrj_1}$ *if* $i_1=j_1$.

Inductive step**:** Suppose $t_{iri_1i_2\ldots i_{l-1}} = t_{jrj_1j_2\ldots j_{l-1}}$ *if* $i_k=j_k$ $\forall k$=1, …, *l*-1. We have to show that $t_{iri_1i_2\ldots i} = t_{jrj_1j_2\ldots j_l}$ *if* $i_k=j_k$ $\forall k$=1, …,*l*. By **Assumption 3**, we have $t_{iri_1i_2\ldots i_l} = t_{jrj_1j_2\ldots j_l}$ *if* $i_l=j_l$ since $t_{iri_1i_2\ldots i_{l-1}} = t_{jrj_1j_2\ldots j_{l-1}}$.

We conclude that $t_{iri_1i_2\ldots i_n} = t_{jrj_1j_2\ldots j_n}$ *if* $i_k=j_k$ $\forall k$=1,..,*n*. ■

It is readily seen that the two thread trees from distinct object servers are homogeneous if they are rooted at identical thread, i.e., $t_{iri_1i_2\ldots i_n} = t_{jrj_1j_2\ldots j_n}$. We next state

Assumption 4 for nested invocations triggered by identical threads.

Assumption 4: Two threads in two distinct server objects trigger identical nested invocation sequences if these threads are identical.

Before we define RNI, we characterize RNI using the following information, called *invocation information*: (1) the thread that produces this invocation; (2) the order (or sequence number) of this invocation generated in that thread; and, (3) the invocation context that contains the information from its target, operation, and arguments. The formal definition of redundant invocation is defined as:

Definition 1 Redundant Nested Invocations): Any two nested invocations from distinct replicas are called redundant if their invocation information is identical.

In this paper, we are interested in the design of an SM that has the following functions: the SM is able to: 1) automatically detect and suppress RNI from a group of replicated servers with MT implementation; 2) forward the earliest RNI and suppress the rest; and, 3) return the response to interested object servers once the response is available.

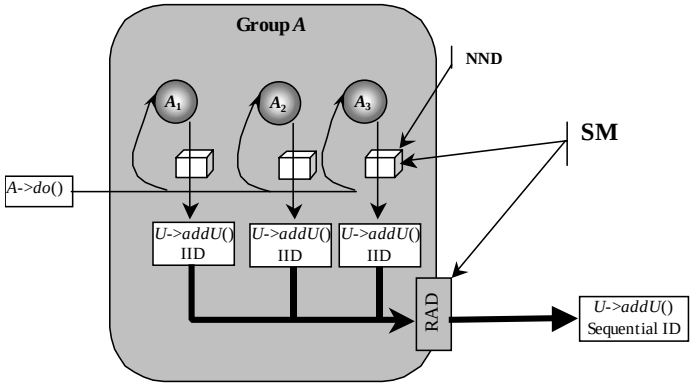

Figure 10. The architecture of suppression mechanism (SM) for RNI.

3 The suppression mechanism design

In this section, we present the proposed SM design as shown in Figure 10. The architecture of the proposed SM is divided into two major components: the *NI notification device* (NND), and *redundancy auto-suppression device* (RAD). NND is responsible for the collection of invocation information of each NI in order to construct a data structure called *invocation identifier* (IID). It attaches the IID as a header for the nested invocation. RAD identifies redundant nested invocations if their IIDs are identical. When the RAD receives nested invocations, it inspects their IID for for redundancy, forwards exactly one RNI to the target server, and returns responses to those nested invocations. We now present our detailed design of NND and RAD. We also use the example discussed in Figure 4 to explain the interaction scenario in SM.

The design goal of NND is to design an IID for RAD to identify RNI based on Definition 1. Therefore, we use invocation information as the IID, as shown in Figure 11. Based on Property 1, we use partial thread index "$k.i_1.i_2..i_n$" of thread $t_{iki_1i_2...i_n}$ as the thread's name. Each thread in the object server's thread tree has its own unique name. We expect that a thread from a replica can identify its equivalent in another replica using the same thread name. The NND keeps track of all the nested invocations from each replica's thread. The sequence number indicates the order that this nested invocation is triggered in that thread. As well as the thread name and sequence number, we use invocation context in IID to double check for redundancy of nested invocations.

NND acquires thread information, and nested invocation information when it intercepts (receives) a nested invocation. Then it constructs the IID for this nested invocation. Therefore, we design NND as an interceptor in each replica to intercept all

out-bound nested invocations. NND appends the header containing the IID information and forwards it to RAD.

We reuse the example in Figure 4 to illustrate the NND design, shown in Figure 12. The replica A_1 and A_2 forks thread t_{11} and t_{12}, respectively, to serve the request *A->do*(). Thread t_{11} and t_{12} forks thread t_{111} and t_{121}, respectively, to interact with target server *V*. Each thread triggers one nested invocation to target server *U* or *V*, shown in the Figure 12. These nested invocations are intercepted and processed by NND. We assign thread t_{111} and t_{211} the same thread key, i.e., t_{111}= t_{211}="1.1". Both thread t_{111} and t_{211} trigger the first nested invocation *V->addV*(2), denoted as n_{12}. and n_{21}. Thus, the sequence number on the thread of the nested invocation n_{12} is 1. As a result, the NND assigns IID [1.1|1|V|addV|2] to n_{12}.

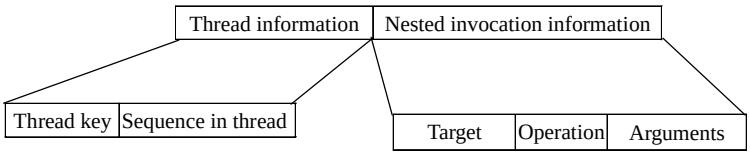

Figure 11. The IID data structure.

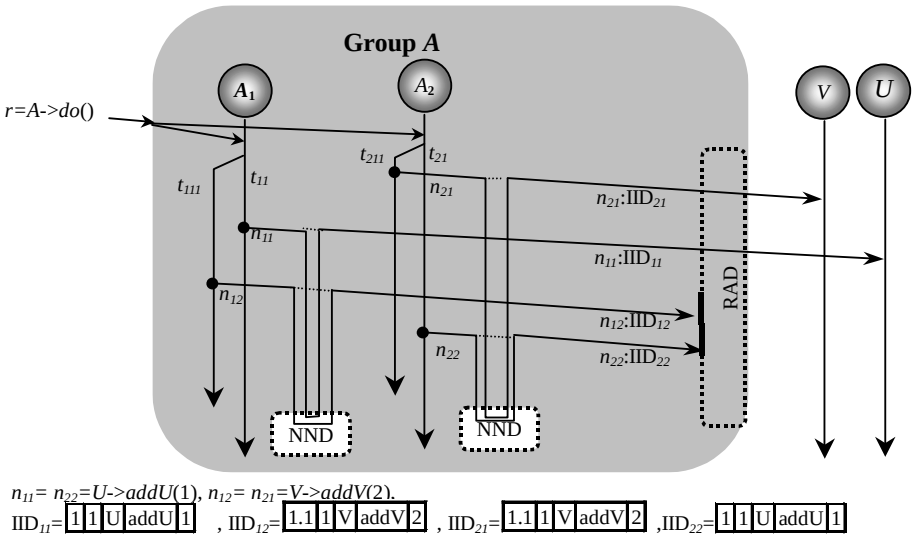

Figure 12. The auto-suppression mechanism of redundant nested invocations.

Each replica group is assigned one RAD that identifies RNIs. The RAD acts as an invocation proxy to send invocations and to accept responses for all replicas. The RAD sends out the earliest RNI by inspecting their IIDs. The RAD forwards a nested invocation, if its IID appears for the first time. In contrast, the RAD blocks a nested invocation if an identical IID appeared before. Furthermore, the RAD dispatches the response to all suspended nested invocations associated with the same IID when it receives a response from the target. Figure 12 depicts a successful auto-suppression example. RAD forwards n_{21} and suppresses n_{12} because $IID_{21} = IID_{12}$ and n_{21} arrives earlier than n_{12}. Similarly, the RAD forwards n_{11} and suppresses n_{22}. As a result, the RAD successfully auto-suppresses all RNIs when using our IID design.

4 Implementation

We present the implementation details of both NND and RAD in this section. One of our design goals is to maintain the SM's portability across all CORBA platforms among different vendors. Later we show in this section that NND is implemented as CORBA portable interceptor [16], and the implementation of RAD adopts the *dynamic skeleton invocation* (DSI) [15]. Both NND and RAD use the standard interfaces in the CORBA 2.4 specification (or up). Therefore we assure the portability of SM. We will illustrate the interaction among the replicas, the SM components, and underlying ORBs via a complete invocation scenario.

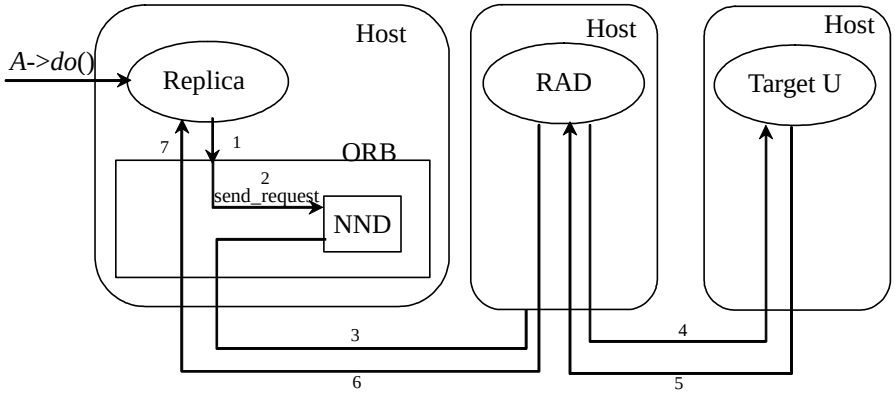

Figure 13. The system architecture implementation

In order to perform thread name management, AP programmers have to use our *FTThread* instead of standard *Thread* class for their MT implemenation. The *FTThread* class inherits standard *Thread* class. Such that our system can set thread key of each child thread in *FTThread*'s constructor. We use *Thread.setName*() to set thread key (or name). Based on **Assumption 1**, each arrival request is assigned a sequence number and is used as root thread key. The sample code is shown in Figure 14.

```
class FTThread extends Thread
{
    public int child=1;                                    //The number of children of this thread
    public FTThread()
    {
        threadKey=this.currentThread().getName()+"."+((FTThread)this.currentThread()).child++;
        this.setName(threadKey);
    }
}
```

Figure 14. Sample code of FTThread implementation

The major function of NND is to intercept NI, create its IID, and forward it to RAD. Figure 15 shows that our NND is implemented as a CORBA portable interceptor to gain transparency and portability by implementing the interceptor callback interface *ClientRequestInterceptor* [16]. In order to acquire the thread information of nested invocation, we implement NND in each replica. The original invocation path outbound

nested invocation is shown as dotted line. Such that our NND automatically intercepts all outbound nested invocation by invoking *ClientRequestInterceptor*::*send_request*() from ORB. NND retrieves thread key by calling Thread.getName(). We use *HashTable* to maintains the number of nested invocations triggered from each thread. It retrieves target operations, and arguments from this NI's ClientRequestIfno [16]. It constructs IID from these retrievable data. NND then appends the IID as a header to the NI and redirect this NI to RAD by raising the standard CORBA exception *ForwardRequest* [16]. A sample code of the NND implementation is shown Figure 16.

Figure 17 depicts the sample code of RAD implementation. A dynamic skeleton interface is implemented in the RAD, the RAD can accept any forwarded nested invocation [15]. We use a Java *Hashtable* as a tracking table to keep track the IID of all forwarded nested invocations. RAD calls *Hashtable.put*() to add the new IID of this nested invocation if it fails to find identical IID in *Hashtable* by calling *Hashtable.get*(), line 11 in Figure 17. RAD forwards or suspend nested invocations by calling a private synchronized function *NIforward*(). We use Java synchronized code segment, line 23 to 32 in Figure 17, to suspend late RNIs until the response is available.

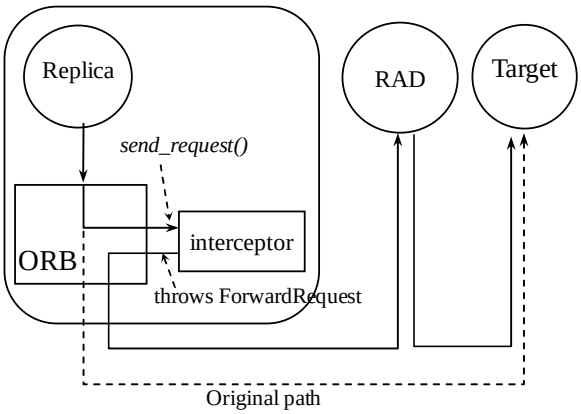

Figure 15. The implementation of the proposed NND

```
1: public class NND extends org.jacorb.orb.LocalityConstrainedObject implements ClientRequestInterceptor
2: {
3:     ...
4:     public void send_request(ClientRequestInfo ri)       throws ForwardRequest
5:     {
6:        ...
7:        throw new ForwardRequest(RAD);
8:        ...
9:     }
10:    ...
}
```

Figure 16 Sample code of the NND implementation

```
1: public class RAD extends org.omg.PortableServer.DynamicImplementation
2: {
3:     private Hashtable RNI= new Hashtable();
4:     ...
5:     public void invoke(org.omg.CORBA.ServerRequest request)
6:     {
7:              ..
8:     //retrieve IID and lookup the IID;
9:       synchronized(RNI){
10:       CRNI =(ControlRNI)RNI.get(iid);
11:       if(CRNI==null){                          //fails to find exist IID
12:            CRNI=new ControlRNI(orb,iid);   //add new IID in Hashtable
13:             RNI.put(iid,CRNI);
14:        }
15:       }
16:     CRNI.NIforward(sr);                  //Forward or suspend this nested invocation
17:      }
18:           ...
19: }
20: class ControlRNI {                       //all identical nested invocations share one distinct object
21:     private Any res=null;
22:     String iid=null;
23:     synchronized void NIforward(org.omg.CORBA.ServerRequest request){
24:       String op = request.operation();
25:     if(LateNI){
26:          request.set_result( res );      // DSI returns the result
27:         return;
28:     }
29:     try{
30:         // prepare and forward the nested invocation inv
31:          res=inv.invoke();
32:     }
33:     ...
34:     }
35:     ...
36: }
```

Figure 17 Sample code of the RAD implementation

Figure 13 depicts a complete scenario that explains the SM's implementation in 7 steps. We notice that in Figure 13 each replica is configured with an NND in ORB and an RAD is assigned to this active group. The replica sends a nested invocation to target *U*

when it serves a request *A->do*(). We now explain each step in Figure 13:

1. Suppose the replica triggers a nested invocation to target *U* when it serves a request *A->do*().
2. The replica's ORB triggers the NND by calling *ClientRequestInterceptor*::*send_request*(), based on portable interceptor specifications [16]. The NND assigns the proposed IID when it intercepts the outbound nested invocation.
3. The NND raises a *ForwardRequest* exception [16] to notify ORB to redirect the nested invocation to the pre-configured RAD.
4. The RAD receives all redirected nested invocations and inspects its IID. It maintains a forwarded IID table and detects redundant nested invocations by looking up this table. It makes a new request to the NI target if the IID is absent in the table. After forwarding the request to target *U*, it puts the forwarded IID in the table. If the IID is in the table it implies an earlier RNI has appeared before. In such a case, the RAD suspends the redirected nested invocation.
5. The RAD receives the response from target *U*.
6. Based on the DSI specification [15], the RAD individually dispatches the response to all suspended RNIs by calling *ServerRequest*::*set_result*().
7. The replica receives the nested invocation response.

5 Performance evaluation

In this section, we examine the overhead contributed to SM components in terms of

response time (or round-trip delay) of a client's invocation. We notice that an (nested) invocation takes an additional 5 steps when SM components are installed in a replicated group as opposed to the case without an SM. (See the discussion of Figure 13 in the previous section). These 5 steps of execution are the major source of overhead. The overhead can be contributed to two SM components, RAD and NND. Therefore, we design two experiments to measure the overhead of RAD and NND respectively. The NND is implemented in each active replica. These NNDs act like a parallel system to process redundant nested invocations. That is, the NND overhead will not rise as the number of active replicas increase. Later we will show that the overhead due to NND is insignificant. The RAD identifies RNI from each replica and dispatches a response to each of them. Thus the RAD overhead depends on the number of active replicas. Our experimental results confirm our observation.

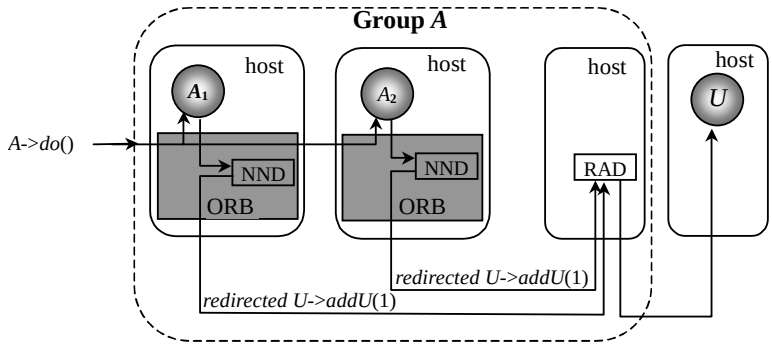

Figure 18. The performance experience environment

These two experiments are built on the same networking environment. We implemented our SM prototype on the Linux platform by using a free Java ORB JacORB 1.3.30 with JDK1.3.1. Group *A* is implemented as an active replication fault-tolerant group. The server replicas are installed on four hosts on the same local area network (LAN). As shown in Figure 18, the client object, RAD, and target object *U* are installed

on distinct hosts on the same LAN. The seven hosts are all Pentium III 866 PCs each with 512MB RAM. The scenarios of both experiments are straightforward. The client issues an invocation *A->do*() to replica group *A* which in turns trigger nested invocation, *U->add*(1). The purpose of the experiments is to measure the response time delay of client invocation *A->do*() due to SM components RAD and NND. Note that all results reported in this section are obtained with 95% of confidence interval with interval half-widths of less than 3% of average response time.

RAD overhead

Because the RAD overhead depends on the number of replicas, we measure the variation of response time by increasing the number of replicas. The average response time result is shown in Figure 19. The experiment shows that the response time rises from 120 to 128 ms when the number of replicas is increased from 2 to 4. The response of *A->do*() is returned to the client object after replicated servers complete the request. Therefore, the increased response time is the processing time for each replica in RAD. Our experiments show that the RAD takes 4ms to process an NI for each replica.

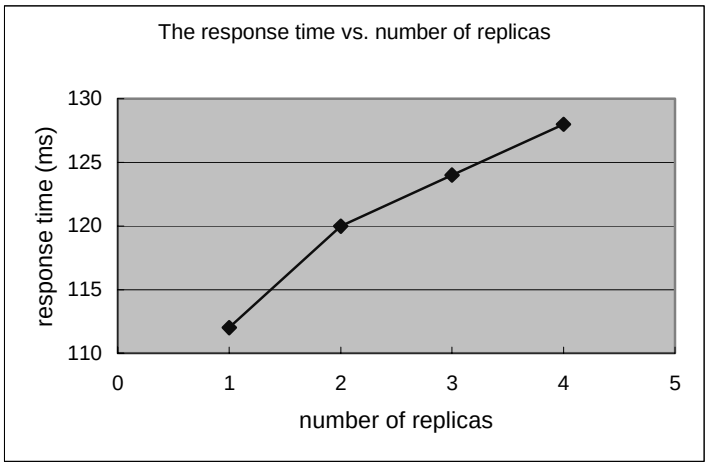

Figure 19. The RAD overhead is proportional to number of replica

We implemented an alternative user-aware active replication group to perform identical functions as our SM. That is, the replicated server AP sends nested invocations to a user designed proxy server that performs identical functions as our RAD. In order to observe the NND overhead, the *A->do*() triggers exact one nested invocation *U->add*(1) to target *U*. As shown in Figure 20, the overhead is calculated as $(T_a\text{-}T_b)/T_b$, where T_a is the response time with NND and T_b is the response time without NND. The response time comparison is shown in bar. The result shows that NND overhead is insignificant, less than 0.8% or 1ms, when the fault-tolerant system has at least two replicas.

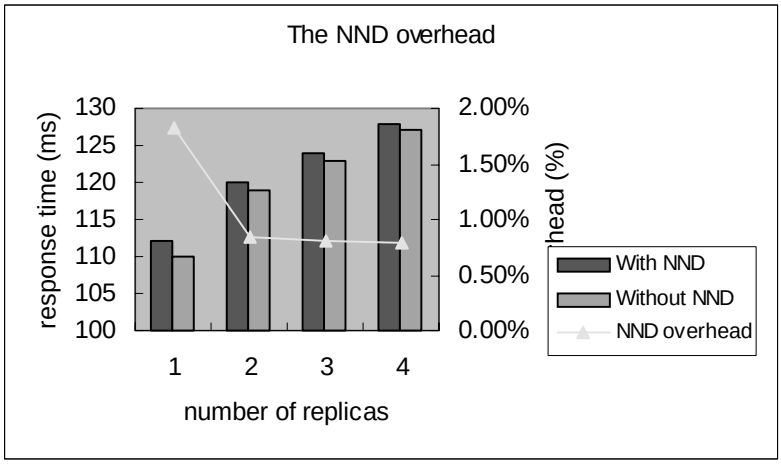

Figure 20. The NND overhead

6 Conclusion

In this paper, we proposed an auto-suppression mechanism of redundant nested invocations to solve the RNI problem in active replication FT CORBA. We also show the system correctness of our suppression mechanism. Actually, the correctness of Narasimhan's SM implementation can be proved in a similar way as our correctness proof. We demonstrated the way to prove the correctness of a system design. Therefore, the system designer can verify their design in advance. We have built a prototype of the

proposed SM to demonstrate how it correctly auto-supress RNI and it is built without modification on OS. The NND can be implemented in ORB kernel as a system facility. That is, the NND is fault-tolerant. The RAD implementation can be built fault-tolerant by configuring multiple copies of RAD. Thus, the fault-tolerance of the SM is ensured.

Appendix

The correctness of the suppression mechanism

We shall discuss the correctness of the SM and prove the correctness of our SM design by referring to the group model in Figure 7. As shown in this figure, the replicas in the group take identical arrival request sequence $R=\{r_1, r_2, \ldots\}$. The output nested invocation sequence from A_m is denoted as $N_m=\{n_{m1}, n_{m2,\ldots}\}$. The nested invocation sequences $N_1,\ldots,$ N_m from $A_1, \ldots, A_m$ are managed by our SM. The correctness of the SM design depends on the correct IID assignment.

In order to prove the correctness of the proposed SM, the redundant nested invocations **Definition 1** is redefined as:

Definition 2: Let $Tseq(n_{ik})$ denote the invocation sequence number in the thread trigger n_{ik}, and IID_{ik} denote the IID of n_{ik}. If $IID_{ik}=IID_{jl}$, and $Tseq(n_{ik})=Tseq(n_{jl})$, then these two nested invocation are redundant to each other and denoted as $n_{ik} \equiv n_{jl}$.

Based on **Definition 1** and the assumptions, a correct SM should have the following properties:

Property 2: Using the sequential number of nested invocation in a thread as IID header, all the IIDs from a replica are distinct.

Property 3**:** The redundancy is detected by inspecting the IIDs of outgoing nested invocations. The system design's correctness can be proved by showing that: (a)the IIDs of nested invocations from arbitrary replica A_i, {IID_{i1}, IID_{i2}, …}, are distinct; (b) $IID_{ik}=IID_{jl}$ implies $n_{ik} \equiv n_{jl}$ for arbitrary replica pair A_i and A_j.

We can prove the system correctness as follows:

Theorem: If an SM is built under the above assumptions and uses 

as IID, then the auto-suppression of redundant nested invocation is correct.

Prove: Proved by induction:

Suppose the group A is configured m active replicas and the IID headers are assigned monotonic increasing. Let $N_1=\{n_{11}, n_{12}, \ldots\}, \ldots, N_m=\{n_{m1}, n_{m2}, \ldots\}$ be nested invocation sequences from replica $A_1, \ldots, A_m$ respectively. The IID_{i1}, IID_{i2}, **…** assigned by any replica A_i are distinct by **Property 2**. We only have to show that $\forall n_{ic} \in N_i \; n_{jd} \in N_j$,$IID_{ic}=IID_{jd}$ implies $n_{ic} \equiv n_{jd}$ $1 \le i, j \le m$ by induction.

Basic step: We have to show that $IID_{i1}=IID_{jb} \rightarrow n_{i1}=n_{jb}$.

We prove it by contradiction. Suppose that $IID_{i1}=IID_{jb}$ and $n_{i1} \neq n_{jb}$. Let H_{ik} be the header of IID_{ik}. $n_{i1} \neq n_{jb}$, $Tseq(n_{i1})=1 \rightarrow Tseq(n_{jb})>1$. Hence, there exists an $n_{ja} \in N_j$ triggered by the same thread with n_{jb}. This leads to $H_{ja}<H_{jb}$. By definition, there must exist an $n_{ie} \in N_i$ such that $1<e$ and $IID_{ie}=IID_{ja}$. Since $IID_{i1}=IID_{jb}$, $IID_{ie}=IID_{ja}$, and $H_{ja}<H_{jb}$, it implies $H_{ie}=H_{ja}<H_{jb}=H_{i1}$. This leads contradiction to the IID header monotonicity assumption. Therefore, $Tseq(n_{i1})=Tseq(n_{jb})=1$ and $n_{i1} \equiv n_{ja}$.

Inductive step: Suppose $IID_{ic}=IID_{jd} \rightarrow n_{ic} \equiv n_{jd}$, 1≤c,d. We have to show there exists a

nested invocation n_{jf}, such that $IID_{ic+1}=IID_{jf} \rightarrow n_{ic+1} \equiv n_{jf}$. Since $IID_{ic+1}=IID_{jf}$, both n_{ic+1} and n_{jf} are in the identical threads $t_{ic+1}=t_{jf}$. We have to prove that in two cases:

Case 1: If none of identical nested invocation pairs (n_{i1}, n_{ja}), **…,** (n_{ic}, n_{jd}) are in threads identical to (t_{ic+1},t_{jf}), then Tseq(n_{i1})= Tseq(n_{j1})=1. Therefore, $IID_{ic+1}=IID_{jf} \rightarrow n_{ic+1} \equiv n_{jf}$.

Case 2: Otherwise, there must exist l, $1 \leq l \leq c$, pairs of $(n_{is}, n_{jt})|_{s \leq c}$ are in threads identical to (t_{ic+1},t_{jf}). Similar to the proof in the basic step, we can show that $Tseq(n_{i1})= Tseq(n_{j1})=l+1$. Therefore, $IID_{ic+1}=IID_{jf} \rightarrow n_{ic+1} \equiv n_{jf}$.

We can conclude that $IID_{ic+1}=IID_{jf} \rightarrow n_{ic+1} \equiv n_{jf}$. Since the IIDs from an arbitrary replica are all distinct and identical IIDs implies identical nested invocations, the auto-suppression of redundant nested invocations is correct. ■

References

1. APM, ANSAware Version 4.1 Manual Set, Architecture Projects Management Ltd., Castle Park, Cambridge UK, *March 1993*.
2. Birman, K., Integrating runtime consistency models for distributed computing, Tech. Rep. 91-1240, Dept. of Computer Science, Cornell University, July 1993.
3. Booch, G., *Object-Oriented Design with Applications*, The Benjamin Cummings Publishing Company, Inc., 1991.
4. Brown, K. and Kindel, C., *Distributed Component Object Model Protocol – DCOM/1.0,1996*
   http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-01.txt.
5. Cherif, A., and Katayama, T. “Replica Management for Fault-tolerant Systems”,

IEEE Micro., vol.18, No.5, pp 54-65, 1998.

6. Cristian, F, Understanding fault-tolerant distributed systems. *Comm. Of ACM***,** 34(2)**,** pp 57-78, 1991.
7. Felber, P., Garbinato, B., Guerraoui, R., and Schiper, A., The implementation of a CORBA object group service. *Theory and Partice of Object System, 4(2), pp 93-105, 1998*.
8. Huang, Y., and Kintala, C., Software Implemented Fault Tolerance, *in Proc. of 22nd Fault-tolerance Computing Symposium*, 2-10, 1993.
9. IONA and Isis, *An Introduction to Orbix+Isis*. IONA Technologies Ltd., and Isis Distributed Systems, Inc., 1994.
10. Liang, D., Fang, C.L., and Yuan, S.M., A Fault-Tolerant Object Service on CORBA, Journal of Systems and Software, Vol. 48, pp. 197-211, 1999.
11. Landis, S. and Maffeis, S., Building Reliable Distributed Systems with CORBA, in *Theory and Practice of Object Systems*, R. Soley, ed., John Wiley, New York, 1997.
12. Moser, L.E., Melliar-Smith, P.M., Agarwal, D.A., Budhia, R.K., and Lingley-Papadopoulos, C.A., Totem: A Fault-Tolerant Multicast Group Communication System, Communications of The ACM, Vol. 39, No. 4, 54-63, 1996
13. Narasimhan, P., Moser, L.E., Melliar-Smith, P.M., Enforcing Determinism for the Consistent Replication of Multithreaded CORBA Applications, Proceedings of the IEEE Symposium for Reliable Distributed Systems, 263-273,1999.
14. Object Management Group, The Common Object Request Broker (CORBA): Architecture and Specification, v 2.6, OMG Technical Committee Document orbos/01-12-01, 2001.
15. Object Management Group, Dynamic Skeleton Interface specification, *OMG*

Technical Committee Document orbos/01-12-46, 2001.

16. Object Management Group, Portable Interceptors specification, *OMG Technical Committee Document orbos/01-12-59*, 2001.
17. Object Management Group, Fault Tolerance CORBA specification, *OMG Technical Committee Document orbos/01-12-63*, 2001.
18. Object Management Group, Fault tolerant CORBA using entity redundancy: Request for proposal. OMG Technical Committee Document orbos/98-04-01, April 1998.
19. Poledna, Stefan "Replica determinism in distributed real-time systems: A brief survey". Real-Time Systems, No. 6, 289-315 (1994).
20. Powell, D., Delta-4: A generic architecture for dependable computing. Springer Verlag, 1991.
21. Slye, J. H. and Elnozahy. Supporting nondeterministic execution in fault-tolerant system. *In Proceedings of IEEE 26th International Symposium on Fault-Tolerant Computing, pp 250-259, Sendai, Japan, June 1996.*