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

Deron Liang¹, Chen-Liang Fang², JiChiang Tsai¹, Chyouhwa Chen²

¹Institute of Information Science, Academia Sinica, Taipei, Taiwan, 11529, R.O.C.
²Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C.

Keywords: fault-tolerance, CORBA, distributed computing environment, object-oriented programming, active replication, replica determinism

Abstract
Active replication is a common approach to build highly available and reliable distributed software applications. Redundant nested invocation (RNI) problem arises when servers in a replicated group issue nested invocations to another server group in response to a client invocation. Automatic suppression of RNI is always a desirable solution, yet it is usually a difficult design issue. A new solution is often required for a new problem due to the changes in system assumptions such as non-deterministic sources. As a result, the correctness of this new solution needs to be verified, usually in an ad hoc manner. In this research, we attempt to propose a new deterministic reference model based on which the verification process of such a solution can be accomplished in a more systematic manner. We wish to demonstrate that it is sufficient to claim the correctness of a solution if that solution satisfies certain properties within the deterministic reference model provided the source of non-determinism and underlying system assumptions are known in prior. The proposed deterministic reference model consists of four levels, namely, they are ideal determinism, isomorphic determinism, similar determinism, and non-determinism. We consider a class of multi-threading CORBA environments in order to demonstrate the power of the proposed deterministic reference model. Given the system assumptions that are commonly set in this field, we are able to prove that any auto-suppression solution toward the RNI problem with multi-threading implementation is correct as long as the nested invocation sequence from the replica server is either ideal deterministic or isomorphic deterministic. Furthermore, we can prove that a solution is incorrect if the nested invocation sequence is either similar deterministic or non-deterministic.

1 Introduction
With the advance of computer and communication technology, distributed computing systems have become increasingly popular in recent years. Many of these distributed systems are
designed to perform critical tasks in hazard environment [1]. Active replication techniques are commonly used to build critical software systems in order to ensure their reliability and availability [13]. Modern large-scale distributed applications are usually built on distributed middleware in order to cope with design issues such as heterogeneity, scalability, and portability [4]. One of the popular middleware is CORBA proposed by the Object Management Group (OMG) [17]. Recently, OMG announces a recommendation named *Fault-tolerant CORBA* [16] to recognize the importance of the fault tolerance issue. One of the open issues pointed out in this recommendation is the *redundant nested invocations problem* (or RNI problem).

A nested invocation refers to the invocation on another server $B$ from a 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. We shall use an example 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 $A \rightarrow \text{do()}$ arrives at group $A$ and this invocation later triggers two nested invocations, i.e., $V \rightarrow \text{addV}(2)$ and $U \rightarrow \text{addU}(1)$ in Figure 1, one on each server $U$ and $V$. Sever $V$ will receive two identical invocations, one from each $A_1$ and $A_2$, since 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 in particular if such requests lead to state changes. OMG’s recommendation [16] has advocated the installation of a *suppression mechanism for redundant nested invocation* (or SM) on active replication groups, the dotted box as shown in Figure 3. The purpose of SM is to ensure that only one of the redundant nested invocations is allowed to forward to the server. In other words, this mechanism needs to identify all redundant nested invocations first, and then suppress them all but one. Furthermore, this mechanism should deliver invocation results to every member in the replicated group. Suppose that a server serves one invocation at a time (or known as 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 sequences of nested invocations from replicated servers are identical given a client invocation to the group. Suppose the SM can incrementally assign a number, small box in Figure 3, to a nested invocation from a server in the group. For instance, the first nested invocation $V \rightarrow \text{addV}(2)$ from server $A_1$ is assigned value 1 as its nested invocation ID. Invocation IDs are shown in small box in Figure 3. SM can detect a nested invocation is redundant if its ID has appeared in prior. For example, the nested invocation $V \rightarrow \text{addV}(2)$ from $A_2$ is blocked at SM since it’s equivalent from $A_1$ has been sent to server $V$. By the same token, the nested invocation $U \rightarrow \text{addU}(1)$ from $A_1$ is blocked at SM. We can argue that the design of SM solves the RNI problem given the assumptions.
Mission-critical systems that deploy active replication mechanism tend to be more performance sensitive in terms of response time, system throughput, real-time constraints, etc. Based on our experience [11], multi-thread implementation at both middleware and application levels is an effective approach to address the performance issues. The multi-thread implementation, however, introduces another dimension of randomness, which makes the design and implementation of SM more complicate [13]. Suppose that both \( A_1 \) and \( A_2 \) are implemented in multi-threads. Their nested invocation sequences may not be identical even if their implementations are the same. The Figure 5 gives one such example where \( A_1 \) and \( A_2 \) are each running with two independent threads. It is obvious that the solution offered in Figure 1 doesn’t work in this case, as shown in Figure 5.

Narasimhan, et al. have proposed an SM solution to address this RNI problem in their Eternal fault-tolerant system [14]. That SM involves the installation of a deterministic thread scheduler in the kernel of ORB. This implementation ensures that all replicas in an active replication group produce identical sequence of nested invocations. As a results, the SM can distinguish redundant nested invocations by assigning sequence numbers to the nested invocations from each replica. Figure 7 depicts such scenario.

Liang, et al. proposed another SM implementation for the same problem [10]. As shown in Figure 9, we notice that the two sequences of nested invocations from the replicas are not identical due to the fact that the application servers are multi-threading. This SM uses a formatted invocation ID that consists of the following invocation information; the target server, method ID, and the arguments. The SM assigns identical ID \( V \langle add\rangle 2 \) for the nested invocations \( V \langle add\rangle 2 \) from replica \( A_1 \) and \( A_2 \), and suppresses \( V \langle add\rangle 2 \) from \( A_1 \) since the same ID has been found in prio with distinct nested invocation from \( A_2 \).

We notice that the two SM designs in Figure 7 and Figure 9 are both correct. We also notice that the arguments of the correctness from [13] and [10] are somewhat ad hoc. Furthermore, we find that the SM shown in Figure 1 becomes ineffective when MT is introduced in the AP implementation. By the same token, the SM proposed in Figure 7 and Figure 9 may become ineffective if the ORB is multi-threading. This observation suggests that a new SM is required for a new RNI problem. As a result, we find it necessary to search for a new framework so that the verification of a “correct SM” for a RNI problem can be conducted in a more systematic manner. We shall explain the framework in more detail later in this article. We notice that the complexity of the problem comes from the non-deterministic nature of the NI sequences generated from each replica in the active group. Therefore, we believe that a new reference model of determinism is needed in order to build such a “correctness verification” framework. With the reference model, we are able to verify the correctness of an SM designed for a new RNI problem in a more systematic manner. We can provide further recommendations to improve an SM if we find that SM is ill-designed based
on this reference model. Finally, we provide scientific insights to the correctness of the SM designs in [10] and [14] using the same reference model.

We propose a deterministic reference model that consists of four levels; they are *ideal determinism*, *isomorphic determinism*, *similar determinism*, and *non-determinism*. We denote these four levels of determinism by L1, L2, L3 and ND, respectively. The formal definitions of determinism will be given in Section 3. We design an algorithm to assist the correctness verification of the SM design based on the proposed reference model. Figure 11 presents this algorithm. This algorithm is based on a few fundamental assumptions on the underlying distributed computing environment. We assume that the arrival requests delivered to an active replication group are in total order. We also assume that both ORB and application servers might be running in multi-threading mode. Furthermore, we assume that the implementation of replicated object servers are identical, and the thread execution within each replicated object server is deterministic, if they are implemented with multi-threading.

As shown in Figure 11, the algorithm takes two input parameters: the RNI problem assumptions (such as the multithreading model), and the corresponding SM design for that problem. Given the information, a software engineer can examine all possible NI sequences to determine its level of determinism. For example, the NI sequences from the example in Figure 7 is an ideal deterministic (or L1) whereas the one in Figure 9 is isomorphic deterministic (or L2). We will prove later in Appendix that a solution is correct if the NI sequences is either L1 or L2. We further prove that a solution is incorrect if the NI sequences is either similar deterministic (or L3) or non-deterministic (ND).

These proofs suggest that a modification to the SM design will make it “correct” if the level of determinism is improved from L3 (or ND) to either L1 or L2. We shall show the advantage of our framework in example of section 4. Other alternatives shall be applied if no design of SM can be found toward an RNI problem. Possible alternatives are user-aware approach and let AP programmer add extra information for redundancy detection and suppression. We will discuss this topic in more detail in Section 4.

The remainder of this paper is structured as follows. Section 2 discuss the importances of multi-threading non-determinism. The new determinism definitions are given in section 3. Finally, conclusions are summarized in section 5.

2 The problem statement
In this section, we will present the problem statement for this work. This includes the formal the formal abstract model of the replicated group, notations used throughout this paper, the assumptions of the non-deterministic sources, and the definitions of “redundant nested invocations”, and the definitions of “correct suppression mechanism” for a given RNI problem.
Figure 13 depicts the single object model for an object implementation. An object server $A_i$ serves a sequence of arrival requests, $R=\{r_1, r_2, \ldots\}$, that triggers a sequence of nested invocations $N_i=\{n_{i1}, n_{i2}, \ldots\}$ to other target servers. Figure 15 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 produces a sequence of nested invocations, i.e., $N_{i1}, \ldots, N_{im}$. We can assume that there are “redundant” nested invocations among these sequences since the object servers $A=\{A_1, \ldots, A_m\}$ are replicated to each other. We assume there is a suppression mechanism (SM) in place to detect the redundant nested invocations as shown in Figure 15. This SM shall form a **group sequence** of nested invocations $N=\{n_1, n_2, \ldots\}$ to corresponding target servers.

The most important feature of SM is to identify redundant nested invocations. The software engineers need to design a correct identification system to identify redundant nested invocations. Basically, to support redundancy detection we may need: parent invocation information if the active group can serve multiple requests at a time; the thread information if application is implemented with MT; the content of the nested invocation; and any extra information to deal any non-determinism source.

The global view of the SM architecture is shown in Figure 17. The SM is divided into two devices: the **invocation ID (IID) assignment device** and **redundancy auto-suppress device (RAD)**. The IID assignment device is implemented in each replica. This device collects all necessary information to composite IID. The redundancy auto-suppression device, second part of the SM, is to identify all redundant nested invocations, forward exact one of them to target server and dispatch responses to associated nested invocations. This device identifies redundancy and dispatching responses by simply maintaining the received IID. The correctness of SM depends on the correct assigned IID.

The design of SM will be straightforward under two conditions; the implementation of all replicated object servers are identical and the operating environment is deterministic. This is because that $N_i = N_j$, $\forall \ i, j$ under such condition. However, it is more complicated in reality. Figure 7 and Figure 9 present two such examples. Poledna has described numerous non-determinism sources that could affect the deterministic behavior of an active replication group; they are inconsistent inputs, inconsistent order, inconsistent membership information, non-deterministic program constructs, local information, timeouts, dynamic scheduling decision, and consistent comparison problem [18]. We have described that multi-thread (MT) implementation in applications is a non-determinism source for active replication system. The AP can be implemented in MT to deal multiple distributed servers for performance concerns [12] if the development tool supports MT. This is so called AP level MT. ORB can be configured to serve two or more requests at a time if they have perfect concurrency control mechanism. This is called ORB level MT. This implies that we have several combinations of
MT system implementation; i.e., single threading in both ORB and AP; MT in ORB; MT in AP; and MT in both ORB and AP.

We are now ready to give a more formal definition of redundant nested invocations, or RNI. A nested invocation can be characterized by the following information, called invocation information: (1) its parent invocation that triggers this nested invocation, (2) the thread that produces this invocation, (3) the order (or the sequence number) of this invocation generated in that thread, and (4) the content of this invocation that contains the information of its target, operation, and arguments. In semantic, redundant nested invocations is stated as following:

“Two nested invocations \( n_{ik} \) and \( n_{jl} \), from two distinct replica \( A_i \) and \( A_j \), respectively, are considered redundant if their invocation information are identical.”

When an active replication group is affected by one or many non-determinism sources, the replicas’ nested invocation sequences \( N_1, \ldots, N_m \) could be non-deterministic. That implies we could not identify RNI by their sequential number in nested invocation sequences. For instance, Figure 5 shows that MT implementation causes \( N_h \) and \( N_2 \) becoming non-deterministic. We define RNI problem as following: An active replication group is injected one or many non-determinism sources and cause non-deterministic replica nested invocation sequences. Software engineers will need a careful design of SM in order to solve a given RNI problem.

In order to give formal definition of redundant nested invocation, we call the collection of information for redundancy detection as redundancy detection information (RDI). Figure 19 depicts the structure of an IID containing RDI. The IID contains header and RDI as body part. The header could be any kind of extra information what software engineers want to fill in. The RDI body contains parent invocation, thread, and nested invocation information. The thread information is important supports when the MT is one of non-determinism source. Most MT libraries offer thread-information management functions to system designer. For example, a system designer could use \( \text{setspecific()} \) and \( \text{getspecific()} \) in DCE thread library to manage the threads in the active group.

In order to complete the correctness proof, we formally define redundant nested invocations:

**Definition 1:** \( Tseq(n_{ik}) \) denotes the order (or the sequence number) of invocation \( n_{ik} \) generated in that thread.

**Definition 2:** Let \( RDI_{ik} \) denote the RDI of \( n_{ik} \). If \( RDI_{ik}=RDI_{jl} \) and \( Tseq(n_{ik})=Tseq(n_{jl}) \), then these two nested invocation are redundant to each other and denoted as \( n_{ik} \equiv n_{jl} \).

Let’s use an example to explain the definitions. As shown in Figure 21, suppose that replicas serves one invocation at a time and the operations are multithreaded. All replicas serve an arrival request \( r_j \) and fork two threads, \( t_0 \) and \( t_1 \), to make nested invocations to two
distinct targets object/group. The thread $t_0$ makes two consequential $addU(1)$ to $U$ and the thread $t_1$ invokes one nested invocation $addV(2)$ to $V$. Suppose the system uses an IID design as shown in Figure 19. Let $IID_k$ denote the IID of $n_k$. If we fill null value as IID header, then $IID_{11}=IID_{21}=\begin{bmatrix}1|0|1|addU|1\end{bmatrix}$, $RDI_{11}=RDI_{21}=\begin{bmatrix}1|0|1|addU|1\end{bmatrix}$ and $Tseq(n_{11})=Tseq(n_{21})=1$. By definition, we find $n_{11} \equiv n_{21}$. Furthermore, $IID_{11}=IID_{22}=\begin{bmatrix}1|0|1|addU|1\end{bmatrix}$, $RDI_{11}=RDI_{22}=\begin{bmatrix}1|0|1|addU|1\end{bmatrix}$ and $Tseq(n_{11})=1 \neq Tseq(n_{22})=2$. The nested invocations $n_{11}$ and $n_{22}$ are definitely different in semantics.

To solve a given RNI problem, software engineers need to design a suppression mechanism of RNI. Base on the above discussion, a correct suppression mechanism can be stated in the following:

“Any nested invocation from a replica can find exact one redundant nested invocation from each of rest replicas, and only one of them can be forwarded to the target. If the response of the nested invocation is sent back to the group, then all replicas should get the response.”

Obviously, a correct auto-suppression mechanism should have the following three properties:

**Property 1:** If the RNI suppression mechanism of a active replication group $A$ is correct, then all $IID_k$ in IID sequence $IID_i$ of nested invocations from arbitrary replica $A_i$, $IID_i = \{IID_{i1}, IID_{i2}, \ldots \}$, are distinct.

**Property 2:** If the RNI suppression mechanism of an active replication group $A$ is correct, then $\forall IID_k \in IID_i, \exists IID_{j_l} \in IID_j, i \neq j, IID_k = IID_{j_l}$.

**Property 3:** If the RNI suppression mechanism of an active replication group $A$ is correct, then $IID_k = IID_{j_l}$ implies $n_k \equiv n_{j_l} \forall i, j$.

The Property 1 ensures that the IID sequence is able to distinguish all invocations in a replica. This property is a basic requirement for a correct SM of RNI. The IID sets from all replicas are all equivalent which is stated in Property 2. The Property 3 describes what a correct IID assignment to be.

In this research, we attempt to propose a framework based on which the correctness of an SM for an RNI problem can be verified in a systematic manner. We will show later in this paper that this framework consists of a new deterministic reference model and an algorithm that implements such a verification process.

**3 The determinism definitions**

The new determinism definitions are used as a reference model in our proposed framework. The determinism is defined in four-level and which are ideal determinism, isomorphic
determinism, similar determinism, and non-determinism. The relationship of the four determinism levels is also given in the end of this section.

We suppose a replicated group $A$ is configured with replicas $A_1, \ldots, A_m$. The group $A$ is serving arbitrary request sequence $R=\{r_1, r_2, \ldots\}$.

**Definition 3:** Two sequences $X$ and $Y$ which elements are all distinct are said to be identical, $X\equiv Y$, iff $|X|=|Y|$ and $\forall x_i \in X$, $\exists$ only one $y_j \in Y$, $x_i \equiv y_j$.

**Definition 4:** Let $IID_i$ and $IID_j$ be the IID sequence of replicated server $A_i$ and $A_j$ respectively when they serve the same arbitrary request sequence $R$. The two replicas $A_i, A_j$ are called ideal deterministic, iff $IID_i \equiv IID_j$.

**Definition 5:** A replicated group $A$ is said to be ideal group, if $\forall A_i, A_j \in A$, $A_i$ and $A_j$ are ideal deterministic. It is denoted as $L1$.

Figure 23 shows an ideal deterministic example. There is only one request in input request sequence $R=\{r_1\}$. The request $r_1$ triggers two nested invocations $U->addU(1)$ to server $U$ and one nested invocation $V->addV(2)$ to server $V$. The system produces two IID sequences $IID_1=\{1|1|0|U|addU|1, 2|1|3|V|addV|2, 3|1|0|U|addU|1\}$ and $IID_2=\{1|1|0|U|addU|1, 2|1|1|V|addV|3, 3|1|0|U|addU|1\}$. The two IID sequences $IID_1$ and $IID_2$ are ideal deterministic.

**Definition 6:** Two sequences $X$ and $Y$ which elements are all distinct are said to be isomorphic, $X\cong Y$, iff $|X|=|Y|$ and $\forall x_i \in X$, there exists only one $y_j \in Y$, $x_i \equiv y_j$, and vice versa.

**Definition 7:** Let $IID_i$ and $IID_j$ be the IID sequence of replicated server $A_i$ and $A_j$ respectively when they serve the same arbitrary request sequence $R$. The two replicas $A_i, A_j$ are called isomorphic, iff $IID_i \cong IID_j$.

**Definition 8:** A replicated group $A$ is an isomorphic group, if $\forall A_i, A_j \in A$, all pairs $A_i$ and $A_j$ are isomorphic. It is denoted as $L2$.

Figure 25 shows an isomorphic deterministic example. The active group is the same as the group in Figure 23. The system uses the $Tseq()$ in IID header and produces two IID sequences $IID_1=\{1|1|0|U|addU|1, 2|1|3|V|addV|2, 3|1|0|U|addU|1\}$ and $IID_2=\{1|1|0|U|addU|1, 2|1|1|V|addV|3, 3|1|0|U|addU|1\}$. The two IID sequences $IID_1$ and $IID_2$ are isomorphic.

**Definition 9:** A set $S$ is called redundant set iff $\forall s_i \in S$ are all identical.

**Definition 10:** A sequence $S$ is categorized based on specific semantic into a finite number of distinct non-empty redundant sets $C_i, \ldots, C_n$, and $\sum_i C_i = |S|$. The set of category subsets $C=\{C_i, \ldots, C_n\}$ are called category set of sequence $S$.

For example, a sequence $S=\{a, b, c, b, a, d\}$ and its category subsets are $\{a, a\}, \{b, b\}, \{c\}$, and $\{d\}$. The category set of sequence $S$ will be $\{\{a, a\}, \{b, b\}, \{c\}, \{d\}\}$.
**Definition 11:** Let $X'$ and $Y'$ be the category set of the sequences $X$ and $Y$ respectively. The sequences $X$ and $Y$ are said to be similar, $X \approx Y$, iff the category sets $X'$ and $Y'$ are equivalent.

**Definition 12:** Let $IID_i$ and $IID_j$ be the IID sequences of replicated server $A_i$ and $A_j$ respectively when they serve the same arbitrary request sequence $R$. The replica $A_i$ and $A_j$ are similar, iff $IID_i \approx IID_j$.

**Definition 13** A replicated group $A$ is said to be a similar group, if $\forall A_i, A_j \in A$, $A_i$ and $A_j$ are similar. It is denoted as $L3$. Figure 27 shows a similar deterministic example. This active group is the same active group as in Figure 23. The system uses null value as IID header and produces two IID sequences $IID_1 = \{ r1 t0 UaddU1, r1 t0 UaddU1, r1 t1 VaddV2 \}$ and $IID_2 = \{ r1 t0 UaddU1, r1 t0 UaddU1, r1 t1 VaddV2 \}$. The category set of $IID_1$ is $\{ \{ r1 t0 UaddU1 \}, \{ r1 t0 UaddU1 \}, \{ r1 t1 VaddV2 \} \}$ and it is identical to the category set of $IID_2$. Thus, these two IID sequences $IID_1$ and $IID_2$ are similar.

**Definition 14:** Let $N_i$ and $N_j$ be the nested invocation sequence of replicated server $A_i$ and $A_j$ respectively when they serve the same arbitrary request sequence $R$. The two replicas $A_i, A_j$ are called non-deterministic, iff $N_i$ and $N_j$ are not similar.

**Definition 15:** A replicated group $A$ is **non-deterministic group**, iff $\exists A_i, A_j \in A$, $A_i$ and $A_j$ are non-deterministic.

Figure 29 shows a non-deterministic example. The two-replica FT-system uses sequential number in IID header and produces two IID sequences $IID_1 = \{ r1 t0 UaddU1, r1 t0 UaddU1, r1 t1 VaddV2 \}$ and $IID_2 = \{ r1 t0 UaddU1, r1 t1 VaddV2, r1 t1 UaddU1 \}$. The second and third IID in $IID_1$ are absent in $IID_2$. Thus, these two IID sequences are non-deterministic.

According to the definitions of determinism, we have the following two properties. The relation of the determinisms is shown in Figure 31. The proof can be done by definitions of determinism.

**Property 4:** If a replicated group is said ideal deterministic, then the group must be isomorphic deterministic; furthermore, an isomorphic group is also a similar group. That is, $L1 \subseteq L2 \subseteq L3$.

**Property 5:** If a replicated group is said not isomorphic or loose isomorphic deterministic, then the group must be non-deterministic. That is, $ND^c = L3$.

### 4 The framework

We shall use an example to demonstrate how to use our framework in fault-tolerant system design. The system design process is briefly divided into three stages. The first stage is to define the RNI problem assumptions. For example, the software engineers decide what kind
of revise it threading model in the new developing system. Based on the problem assumptions, the software engineers need to prove the correctness of SM by referring to the four-level determinism model for the verification procedure in stage 2. That is, software engineers need to prove that the SM design is correct when the worst determinism is verified as L1 or L2 and the SM design is incorrect when design is verified as L3 or ND. After the correctness proving, the software engineers perform the verification procedures, as shown in Figure 11. In this section, we demonstrate the 3-stage example to show how design work is done.

The software engineers define problem assumptions based on their desired system designs. For instance, some basic active replication system implementations have been done to have the following properties and the software engineers may define the problem assumptions as:

1. All arrival requests are totally ordered.
2. The execution behavior of all distinct AP threads is deterministic.
3. All active replicas are built on identical thread ID management mechanism. That is, the execution thread IDs in all active replicas are identical.
4. The IID header should be in monotonically increasing or monotonically decreasing.

In the second stage, the software engineers need to prove the correctness of their verification reference model based on the problem assumptions. Basically, the reference model is consist of the following theorems:

**Theorem 1:** If an active replication group $A$ is classified as ideal deterministic (or L1) then the auto-suppression of redundant nested invocation is correct.

**Theorem 2:** If an active replication group $A$ is classified as isomorphic deterministic (or L2) then the auto-suppression of redundant nested invocation is correct.

**Theorem 3:** If an active replication group $A$ is classified as similar deterministic (or L3) then the system may cause incorrect suppression of RNI

**Theorem 4:** If an active replication group is classified as non-deterministic (or ND) then the system may cause incorrect suppression of RNI

The software engineers need to prove correctness of reference model every time when they change the problem assumptions. The software engineers need to prove that the SM design is correct when the worst determinism is verified as L1 or L2 and the SM design is incorrect when design is verified as L3 or ND. The proofs of these theorems is shown in Appendix for interested readers.

The final stage of the demonstration is to perform the verification procedures against to the SM design. The software engineers need to enumerate all possible cases nested invocation sequences. Extra information can be added as RDI header. The extra information could be a
sequential number of NI in a thread or an operation with other information like parent invocation ID, invocation arguments or other information you can support. Then the software engineers have to assign the IID for each nested invocation based on the IID design and classify the determinism of all possible IID sequences by referring to our determinism model. If the worst determinism level of all IID sequences is L1 or L2, then we are done. Otherwise, we improve SM design and run the verification procedure again. We should change the problem assumptions and run the 3-stage process again, if we could not find improvement on SM design based on current assumption.

We shall use a previous example, as shown in Figure 21, to demonstrate the verification procedure. Obviously, successful SM design should forward two consecutive \( U \rightarrow \text{addU}(1) \) and \( V \rightarrow \text{addV}(2) \). The nested invocation sequences from two replicas \( A_1 \) and \( A_2 \) may vary due to non-deterministic thread scheduling. We use null value as IID header information for first try of SM design. As shown in Figure 33, we get IID sequence \( \text{IID}_1=\{ [1][0][U\text{addU}] 1, [1][1][V\text{addV}] 2 \} \) from \( A_1 \) and \( \text{IID}_2=\{ [1][0][U\text{addU}] 1, [2][0][U\text{addU}] 1 \} \) from \( A_2 \). These two IID sequences are verified as worst determinism level. \( \text{IID}_1 \) has two IIDs, the first and the second one, are identical. These two IID sequences are similar determinism by Definition 12. As shown in the figure, the SM only forward one \( U \rightarrow \text{addU}(1) \) and one \( V \rightarrow \text{addV}(2) \). This SM design is not a feasible solution.

We improve the SM design by assigning a sequential number in IID header and verify it again. After enumerating all possible RNI sequences, the worst IID sequences are shown in Figure 35. We get the worst IID sequence \( \text{IID}_1=\{ [1][0][U\text{addU}] 1, [2][1][V\text{addV}] 2 \} \) from \( A_1 \) and \( \text{IID}_2=\{ [1][0][U\text{addU}] 1, [1][0][U\text{addU}] 1 \} \) from \( A_2 \). For the second IID within \( \text{IID}_1 \), we could not find identical IID within \( \text{IID}_2 \). These two IID sequences are non-deterministic by Definition 14. Unfortunately, the SM fails to suppress \( A_1 \)'s RNI \( V \rightarrow \text{addV}(2) \) and \( A_2 \)'s RNI \( U \rightarrow \text{addU}(1) \), as shown in dotted arrow. This is not a feasible solution either.

The SM design is improved by assigning \( Tseq \) in IID header for the third try. After enumerating all possible cases, we find the worst case and the two isomorphic deterministic IID sequence \( \text{IID}_1=\{ [1][0][U\text{addU}] 1, [2][1][V\text{addV}] 2 \} \), \( \text{IID}_2=\{ [1][0][U\text{addU}] 1, [1][1][V\text{addV}] 2 \} \). As shown in the Figure 37, the system successfully auto-suppress the redundant nested invocations. We are done in the verification procedure and the SM design is a feasible design based on current problem assumptions.

5 Conclusions
In this paper, we define RNI problem for an active replication fault tolerant system. We propose a deterministic reference model that consists of four levels; they are ideal determinism, isomorphic determinism, similar determinism, and non-determinism. We have
shown how to prove the correctness of verification model. We also design an algorithm to assist the correctness verification of the SM design based on the proposed reference model. A modification to the SM design might become “correct” if the level of determinism is improved from L3 or ND to either L1 or L2.

We have to remind readers that a design is classified similar or non-deterministic determinism does not imply this system has to be wrong. For example, a design called active parallel replication [5] is classified as non-deterministic but the special system design is correct. Fortunately, we have shown that a L1 or L2 system has to be correct in auto-suppression.

References


Appendix

Based on the example problem assumptions, we demonstrate the correctness proof in the followings to show how the stage 2 is done:

**Theorem 1:** If an active replication group \( A \) is classified as ideal deterministic then the auto-suppression of redundant nested invocation is correct.

**Prove:** Suppose the group \( A \) is configured with \( m \) active replicas and it is classified as ideal-deterministic group. 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. By definition of ideal determinism, all elements are distinct and the IID sets from all replicas are equivalent. We only have to show that \( IID_A = IID_B \) implies \( n_i = n_j \) for \( 1 \leq i, j \leq m \) by induction.

By definition, ideal determinism implies \( IID_B = IID_A \) for \( 1 \leq i, j \leq m \) and \( 1 \leq k \). Since \( IID_B = IID_A \) and RDI is part of IID, we have \( RDI_B = RDI_A \). Let \( T_{ik} \) and \( T_{jk} \) be the thread key in \( RDI_B \) and \( RDI_A \) respectively. Since \( RDI_B = RDI_A \) implies \( T_{ik} = T_{jk} \). Basic step: \( RDI_i = RDI_j \) implies \( n_{i1} = n_{j1} \) and \( n_{jt} \) are triggered by identical threads and both are first nested invocation in the identical threads \( T_{ik} \) and \( T_{jk} \). \( Tseq(n_{ik}) = Tseq(n_{jk}) = 1 \). Therefore, \( RDI_i = RDI_j \) implies \( n_{jt} = n_{jt} \).

Inductive step: Suppose that \( n_{ik} \) and \( n_{jk} \), \( 1 \leq k \), are identical nested invocations, we have to show \( n_{ik+1} = n_{jk+1} \) too. We suppose that \( n_{ik+1} \) and \( n_{jk+1} \) are triggered in thread \( T_{ik+1} \) and \( T_{jk+1} \) respectively. If none of \( \{n_{i1}, \ldots, n_{ik}\} \cup \{n_{j1}, \ldots, n_{jk}\} \) is triggered in thread identical to \( T_{ik+1} \) and \( T_{jk+1} \), then both \( n_{ik+1} \) and \( n_{jk+1} \) are the first nested invocation in thread \( Tseq(n_{ik+1}) = Tseq(n_{jk+1}) = 1 \), and \( n_{ik+1} = n_{jk+1} \). Otherwise, there must exist \( \{1 \leq s \leq k\} \) pairs of \( \{n_{is}, n_{js}\} \) are triggered in threads identical to \( T_{ik+1} \) and \( T_{jk+1} \). Hence, \( Tseq(n_{ik+1}) = Tseq(n_{jk+1}) = l + 1 \). Therefore, \( n_{ik+1} = n_{jk+1} \).

Since \( RDI_i, RDI_j, \ldots \) assigned by any replica \( A_i \) are distinct and \( RDI_i = RDI_j \) implies \( n_i = n_j \) for \( 1 \leq i, j \leq m \), we have proved that the auto-suppression of redundant nested invocations is correct.

**Theorem 2:** If an active replication group \( A \) is classified as isomorphic deterministic then the auto-suppression of redundant nested invocation is correct.

**Prove:** Proved by induction.

Suppose the group \( A \) is configured \( m \) active replicas and it is classified as isomorphic-deterministic group. The system assigns IID headers in 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. By definition of isomorphic determinism, all elements are distinct and the IID
sets from all replicas are equivalent. We only have to show that \(\forall n_i \in N_i, n_j \in N_j, IID_{n_i} = IID_{n_j}\) implies \(n_i \equiv n_j, 1 \leq i, j \leq m\) by induction.

Basic step: We have to show that \(IID_i = IID_j\) implies \(n_i \equiv n_j\).

We prove it by contradiction. Suppose that \(IID_i = IID_j\) and \(n_i \neq n_j\). \(IID_i = IID_j\) implies \(RDI_i = RDI_j\). Let \(H_i = \text{header of } IID_i\). \(n_i \neq n_j\) \(Tseq(n_i) = 1\) implies \(Tseq(n_j) > 1\). Hence, there exists an \(n_j \in N_j\) triggered by the same thread with \(n_j\). This leads to \(H_j < H_i\). By Definition 7, there must exist an \(n_e \in N_i\) such that \(1 < e\) and \(IID_e = IID_j\). Obviously, \(RDI_e = RDI_j\). Since \(RDI_i = RDI_j\), \(RDI_e = RDI_j\), and \(H_e < H_i\), that implies \(H_e = H_i = H_j\). This leads contradiction to the IID header monoticity assumption. Therefore, \(Tseq(n_i) = Tseq(n_j) = 1\) and \(n_i \equiv n_j\).

Inductive step: Suppose \(IID_a = IID_{d1}\) implies \(n_i \equiv n_{ja}, 1 \leq c, d\). We have to show there exists a nested invocation \(n_{ja}\) such that \(IID_{a+c} = IID_{df}\) implies \(n_{ic} = n_{ja}\). \(IID_{a+c} = IID_{df}\) implies \(RDI_{ic} = RDI_{df}\). Since \(RDI_{ic} = RDI_{df}\), both \(n_{ic} \neq n_{ja}\) are in the identical threads \(T_{ic+1} = T_{df}\). We have to prove that in two cases:

Case 1: If none of identical nested invocation pairs \((n_i, n_{ja})\), \((n_c, n_{ja})\) are in threads identical to \((T_{ic+1}, T_{df})\), then \(Tseq(n_i) = Tseq(n_{ic}) = 1\). Therefore, \(IID_{a+c} = IID_{df}\) implies \(n_{ic} = n_{ja}\).

Case 2: Otherwise, there must exists \(l \leq c, d\) pairs of \((n_i, n_{ja})\) are in threads identical to \((T_{ic+1}, T_{df})\). In similar to the proof in basic step, we can show that \(Tseq(n_i) = Tseq(n_{ic}) = l + 1\). Therefore, \(IID_{a+c} = IID_{df}\) implies \(n_{ic} = n_{ja}\).

We can conclude that \(IID_{ic+1} = IID_{df}\) implies \(n_{ic} = n_{ja}\). Since the IIDs from arbitrary replica are distinct and identical IID implies identical nested invocations, the auto-suppression of redundant nested invocations is correct.

**Theorem 3:** If an active replication group \(A\) is classified as similar deterministic then the system may cause incorrect suppression of RNI

**Proof:** Suppose the group \(A\) is configured \(m\) active replicas. Let \(IID_1, ..., IID_m\) be the IID sequences from active replicas \(A_1, ..., A_m\) respectively. By Definition 7, there may exist redundant sets and their size could be greater than 1. That is, IIDs in the IID sequence may be redundant. This violates **Property 1**. Therefore, the system fails to auto-suppression redundant nested invocations.

**Theorem 4:** If an active replication group is classified as non-deterministic then the system may cause incorrect suppression of RNI

**Proof:** Suppose the group \(A\) is configured \(m\) active replicas. Let \(N_1, ..., N_m\) be the nested invocation sequences from active replicas \(A_1, ..., A_m\) respectively. By definition, there exists at least one nested invocation in a nested invocation sequence that could not be found in one or more other nested invocation sequences. This conflicts with the definition of correct auto-suppression of redundant nested invocations.
Server $V$ executes $addV()$ twice, and RNI problem raises

**Figure 1** The RNI problem

SM suppresses late RNI

**Figure 3** The desire features of suppression mechanism for RNI
Figure 5 Sequential number ID fails to identify RNIs

Figure 7 A deterministic thread scheduler solution for redundant nested invocations problem
Figure 9 Using formatted invocation ID to solve the same problem

Figure 11 The global view of the framework

Figure 13 The single server object model
Figure 15 The group model

Figure 17 The architecture of suppression mechanism (SM) for RNI

Figure 19 The invocation identifier embedding RDI
How to identify redundant invocation

Figure 21

Ideal determinism example

Figure 23
IID₁ = { [r₁][t₀][U][addU]₁, [r₁][t₀][U][addU]₁, [r₁][t₁][V][addV]₂ }  
IID₂ = { [r₁][t₀][U][addU]₁, [r₁][t₁][V][addV]₂, [r₁][t₀][U][addU]₁ }  

Figure 25 An isomorphic example

IID₁ = { [r₁][t₀][U][addU]₁, [r₁][t₀][U][addU]₁, [r₁][t₁][V][addV]₂ }  
IID₂ = { [r₁][t₀][U][addU]₁, [r₁][t₀][U][addU]₁, [r₁][t₁][V][addV]₂ }  

Figure 27 A similar determinism example
\[
I_{ID_1} = \{ \{ r1, t0, U_{addU}1 \}, \{ r1, t0, U_{addU}1 \}, \{ r1, t1, V_{addV}2 \} \}
\]
\[
I_{ID_2} = \{ \{ r1, t0, U_{addU}1 \}, \{ r1, t1, V_{addV}2 \}, \{ r1, t0, U_{addU}1 \} \}
\]

Figure 29 A non-deterministic example

Figure 31 The relationship of the determinism
Figure 33 The first try of design is verified as similar determinism

$$I_{ID_1} = \{ [1], [1], [2], [3] \}$$

$$I_{ID_2} = \{ [1], [1], [2], [3] \}$$

Figure 35 The second try of design is verified as non-deterministic

$$I_{ID_1} = \{ [1], [2], [3] \}$$

$$I_{ID_2} = \{ [1], [2], [3] \}$$
\[ IHD_1 = \{ 3r1t0U \text{add}U1, 3r1t0U \text{add}U1, 3r1t1V \text{add}V2 \} \]
\[ IHD_2 = \{ 3r1t0U \text{add}U1, 3r1t1V \text{add}V2, 3r1t0U \text{add}U1 \} \]

Figure 37 The final design is isomorphic determinism