Journal of Computer Science 5 (1): 11-22, 2009
ISSN 1549-3636
© 2009 Science Publications

Providing Reliability in Replicated Middleware Applications

[1]R. Saravanan and [2]N. Ramaraj
[1]Department of Computer Science and Engineering,
Pacheri Sri Nallathangal Amman College of Engineering and Technology, Dindigul, India
[2]G Katha Muthu Engineering College, Chennai, India

Abstract: Problem statement: Data inconsistency is raised in actively replicated environment due to non-determinism in the applications that defeats the purpose of replication as a fault-tolerance strategy. **Approach:** We proposed an efficient framework RTC which ensured determinism among the replicas in fault tolerance middleware applications. This method exploits the technique of statically analyzing the application source code of client and identifies the variables and system calls which lead to non-deterministic state in the replicas. The source code consists of non-deterministic variables and system calls which are identified and set the flag field. The client request consist of flag field and the service request, which is sent to all the servers through time stamp based replication protocol (TSP) that facilitate the multiple clients and the request is sent to the servers. The distributed coordination method was initiated if necessary; otherwise send any one response of the servers to the client by duplicate removal. Distributed coordination which involves, the selection of a primary replica based on the time stamp value. It is responsible for taking all non-deterministic decisions. The state of the primary replica was updated to all other replica connected asynchronously to maintain consistency. **Results:** We evaluated our technique by increasing the contamination percentage of non-determinism and increasing number of replicas. **Conclusion:** The method suggested by us reduces the communication and synchronization overhead which was proved through implementation. We evaluate our technique for the active replication of servers using micro benchmarks that contain various sources of non-determinism. Multi-threading, system call, shared I/O and random ( ).

Key words: Non-determinism, fault-tolerance system, distributed coordination, active replication, time stamp based replication

INTRODUCTION

Replication of components is a common technique for providing fault tolerance in distributed systems. The concept of replication is the creation and distribution of multiple identical copies (Replicas) of a component across a system so that the failure of a replica can be masked by the availability of other replicas. There are essentially four kinds of replication styles[8]-active replication, semi-active (leader-follower) replication, passive replication and coordinator-cohort replication. In active replication (state-machine approach[18]), each server replica processes every client invocation and returns the response to the client. With active replication the availability of system is more when comparing to any other replication technique. Care must be taken to ensure that only one of these duplicate responses is actually delivered to the client. The failure of a single active replica is masked by the presence of the other active replicas that also perform the operation and generate the desired result. Semi-active (or leader-follower) replication is a hybrid replica organization technique which accommodate non-deterministic replicas with an availability nearly as high as in active replication. As in active replication, all replicas receive a request; however, one replica (the leader) plays a special role. Whenever the leader makes a non-deterministic decision, it notifies the other replicas (its followers) of its choice. The followers are then forced to take the same decision. This guarantees that the state evolution in all replicas is the same. In semi-active replication, only the leader replica replies to clients.

With passive replication, only one of the server replicas designated the primary, processes the client's invocations and returns response to the client. With warm passive replication, the remaining passive replicas, known as backups, are preloaded into memory and synchronized periodically with the primary replica

Corresponding Author: R. Saravanan, [1]Department of Computer Science and Engineering,
Pacheri Sri Nallathangal Amman College of Engineering and Technology, Dindigul, India

so that one of them can take over if the primary replica fails. With cold passive replication, however, the backup replicas are "cold," i.e., not even running, as long as the primary replica is operational. To allow for recovery, the state of the primary replica is periodically check pointed and stored in a log. If the existing primary replica fails, a backup replica is launched, with its state initialized from the log, to take over as the new primary. Both active and passive replication styles require mechanisms to support state transfer. For passive replication, the transfer of state occurs periodically from the primary to the backups, from the existing primary to a log, or from the log to a new primary; for active replication, the transfer of state occurs when a new active replica is launched and needs its state synchronized with the operational active replicas. Also note that passive replication cannot be used to mask byzantine failures as there is only one single replica executing, the backups serve only as warm stand-bys.

Coordinator-cohort replication is another hybrid replica organization, very similar to semi-active replication. It has been developed in the context of the Isis toolkit[4]. From the point of view of the communication pattern, it is very similar to passive replication, the only difference being that all replicas receive a request. This makes it possible to mask even failures of the primary replica; the client does not have to re-send a request. However, only the coordinator handles the request and updates the cohort replicas by means of checkpoints. The result is therefore determined by the execution on the coordinator, which may be non-deterministic. If the coordinator fails, one of the cohorts becomes the new coordinator and proceeds with execution from the last checkpoint. Checkpoints therefore must be coordinated with respect to output

Determinism[13] is an important property that requires the replication to work consistently. A component is said to be deterministic if it contains no characteristics that could cause replicas to become inconsistent with each other. A Component is said to be deterministic, when started from the same initial state and supplied the same ordered sequence of input messages, should reach the same final state and produce the same output. But in real time application, while executing some system calls and variables, replicas enter in to non-deterministic state.

One simplistic approach to avoid non-determinism that forbids the use of multithreading, shared memory, local I/O, system calls, random numbers and timers. In fact this approach is adopted by the industrial standards such as Fault-Tolerant CORBA[15]. In real world application we wish to use all these non-deterministic functions. Application state can be in any one of the three mutually exclusive categories: pure non-determinism, contaminated non-determinism and pure determinism.

In a Pure non-determinism, any functions are the originating source of non-determinism and affect the server's state. Examples include system calls such as gettimeofday or random, change the server's state non-deterministically. For example the variable det is nondeterministic:

```
for (int j = 0; j < 100; j++)
det [j] = random ( )
```

Shared state among threads also falls within this category. However, we treat shared state in a special way each access of shared state by a thread is considered to be a separate source of non-determinism. For example, consider a single shared variable between two threads; if each thread accesses this variable four times, then, there exist eight separate instances of pure non-determinism. It is immaterial that these eight instances happen to involve the same variable. The Contaminated non-determinism covers the state that has any dependency, direct or indirect, on an instance of pure non-determinism. Contaminated state captures the effect of pure non-determinism when it is executed and it is propagated to the rest of the application. An example is the contaminated variable bar that depends on the purely nondeterministic variable det:

```
for (int j = 0; j < 100; j++)
{det [j] = random ( ); bar [j+100] = det [j] ;}
```

Pure determinism indicates the state that has no dependency or whatsoever on the identified pure non-determinism. This category of state will always be consistent across all server replicas. An example is:

```
int x = random ( ); b = 5; return b
```

Here the variable x is nondeterministic, but its value does not affect the server state.

The objective of the research is to permit the programmers to continue and create distributed applications and even they can use functions and variables that cause non-deterministic state across the replicas. To provide fault tolerance among the replicas, we are using CORBA fault tolerance middleware. With the active replication of servers, existing method[14] involves delay, synchronization over head of replicas (Use of Group Communication Protocol) and communication overhead while transferring the state information (transfer-ckpt, transfer-contam) of any one of the server to client and again from the client it is communicated to all the servers actively connected in the network. In this study we have proposed a framework RTC (Real Time Compensation). To avoid synchronization over head of replicas, we are using the time stamp based replication (TSP approach[17]) instead of group communication protocol and reduce the communication overhead (Delay and Congestion) by sending the state of the primary replica directly to other secondary replicas (Leader-follower) actively connected and thus avoiding the state transfer to the client and maintain determinism in all the replicas.

MATERIALS AND METHODS

An overview: The following assumptions are made about the system. RTC relies on having complete access to the application's source code, along with the ability to modify it prior to deployment. Specifically, we assume that we are allowed to modify the source codes of the client, the server and the IDL interfaces of all objects. Both the client and server source code must be available for analysis, although only the server is replicated. We also assume that all of the application state can be determined statically. The replicas are replicated in several sites and are communicate each other to reliable FIFO channels.

Our approach involves the static analysis of the source code in the client and set the flag field if non-deterministic variables and system calls are found and it is sent to server together with client request. The program analysis also tracks all live variables and their dependencies that lead to non-deterministic state in the replica.

The server replicas are actively replicated. The client request is passed to all the server replicas through the time stamp based replication together with the flag field. It ensures that all the replicas execute the client's request in the same order. Server replicas check the flag field, if it is true, the client request consists of non-deterministic variables and execution this clearly leads to non-deterministic state in the replicas. The following activities are performed while handling non-determinism in the replicas:

- Server replicas receive the client request and check the flag field. If it is true, initiate the distributed coordination method
- Distributed coordination method involves selection of primary replica based on the time stamp value, one replica is selected as primary and others are called secondary replicas. The flag is true; the request is processed by the primary replica
- Then the state of the primary is propagated to other replicas connected actively and maintains consistent state in all the replicas. The replicas send responses to the client. Using duplicate removal, only one response is allowed to client
- It is not necessary to connect all the replicas in lock step synchronizing state. All the server replicas allowed to be connected in asynchronous mode
- Communication overhead is reduced, because there is no state transfer between the client and server
- If the flag is false, allow the replicas to execute the client request. The server replicas send responses to the client. The same response received from different replicas leads to duplication of results. Our frame work (RTC) ensures to send only one response to the client

The client-server architecture is implemented using CORBA (JacORB)[20] and it will act as a vehicle between client and server. The replicas form a group called Replica Service Group and it is identified through the logical address G. Servers (Replicas) are replicated in several sites and each replica site consist of a frame work RTC and the server. The server provides all the service to the client. The RTC is residing between the client and the server. RTC is responsible for ensuring the consistency of the replicas.

A client sends a service request to the RTC. RTC verifies the flag field of the request and it is forwarded to their corresponding replica and also the other RTC in the different replica site. Replica executes the client request and the response is sent back to RTC. RTC is responsible to send the server response to the client. Verification of flag field may rise to two cases. Case 1-Flag field is true, case 2-Flag field is false. We handle the non-determinism according to the flag status.

Case 1: Flag field is true; it means the client request consisting of variables and system calls which may leads to non deterministic state, if they are executed in the server. It is necessary to initiate the distributed coordination method.

Case 2: Flag field is false; it means the client request does not having any variables and system calls which may leads to non-deterministic state in the server and not necessary to invoke the distributed coordination method. The client request is executed by the server and the response in sent to the client through RTC.

Source program analysis framework: To perform program analysis, the application source code is statically analyzed[19] and finds the variables and system calls that will lead to non-deterministic state in the replicas if executed. For static program analysis, we have used the CC-RIDER[6] the free open source software.

CC-RIDER: CC-RIDER, is a unique and powerful code visualization tool, promotes efficiency and productivity. This enables to understand source code quickly. CC-RIDER is not merely a class browser-it provides complete information on functions, variables, enum values and macros. It is uniquely designed to work with the tools already using and helps to easily penetrate the complexity of the source code. Figure 1 shows the two main components of the CC-RIDER package, the Analyzer and the Visualizer and how they interact with the project's source code to facilitate editing and documenting the code.

A database is created to store all the details of the source files like header files, functions, variables, dependency of the variables and system calls. The analyzer then processes the source modules and header files to the database, which contains detailed interrelationships between all symbols in the source code. Once a database is built, the Visualizer provides several ways to explore edit and document the code.

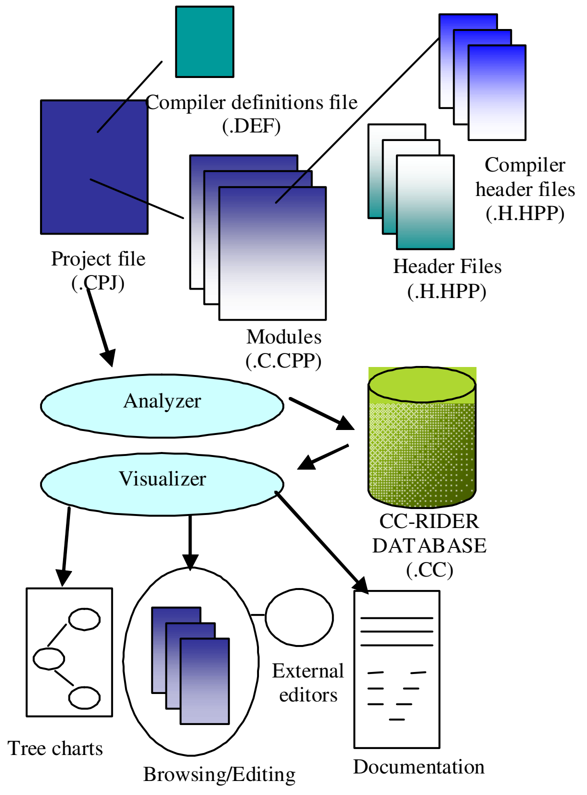

Fig. 1: Components of CC-RIDER

CC-RIDER reveals detailed information about the symbols, where and how they're used along with complex member inheritance relationships, macro expansions and template instantiations. The Class Hierarchy view is a graphical representation of the class inheritance structure of the program. Function calls and data references are represented as differently shaped nodes in the tree. These trees are extremely useful for examining the structure of C applications. The Project Statistics window shows statistics about the analyzed application, for example: the number of source code lines, number of comments, number of classes, macros, functions and enums. Using the statistics information, we have found the source code consist of system calls and functions which lead to non deterministic state in the server. A flag field is introduced and set to true if nondeterministic variables are found in the source code. Otherwise set the flag field to zero. This flag field is send to the server together with the client request.

Design details of time stamp based replication protocol (TSP): To ensure that the states on the replicas are consistent, it is required that (a) the code running on the replicas be deterministic and, (b) the clients' requests must be sent to the replicas in the same order. Since multiple clients might send requests to replicas simultaneously, a total order is needed when multicasting clients' requests to the replicas. Total order means all requests are delivered to the replicas in the same order even if the senders of the requests are different. Group communication primitives[4] can be used to ensure the ordering of the clients' requests in active replication. However, replication schemes using group communication primitives suffer from high overhead due to the high synchronization cost amongst the replicas[21]. To reduce the synchronization overhead, we are using time stamp based replication protocol to ensure total ordering of client request.

The TS Protocol, the system is based on the state machine approach[18]. The TS protocol allows the replicas to reach an agreement on the order in which the clients' requests are processed. In TSP, each client's service request is given a unique timestamp. The replicas carry out the execution of the clients' requests in their timestamps' order. That is, a request with a smaller timestamp will be executed before a request with a larger timestamp. TSP assumes that it is rare that different clients send service requests to the replicas simultaneously.

The TS protocol is running a part of RTC Thus, when a client's request is received by a RTC; the RTC sends the request to the RTC's replica for immediate execution. Meanwhile, the RTC exchanges information with the other RTCs to determine whether the request has been executed in the correct order.

Each replica keeps a logical clock defined in[11]. The logical clock is an integer counter which increases monotonically. It is initialized to 0. A timestamp is a pair (l_clock, ip) where ip is the IP address of a replica and l_clock is the logical clock value of the replica. The ">" relation between two timestamps, (l_clock_1, ip_1) and (l_clock_2, ip_2), is defined:

$$(l\_clock_1, ip_1) > (l\_clock_2, ip_2) \Leftrightarrow (l\_clock_1 > l\_clock_2) \vee ((l\_clock_1 = l\_clock_2) \wedge (ip_1 > ip_2))$$

The logical clock, l_clock, of a replica is updated according to the statements S1 and S2 described below:

- S1 when a replica, say r, receives a service request from a client, the logical clock is updated as below:

  $l\_clock \leftarrow l\_clock+1$

- S2 when a replica, say r, receives a multicast request, say m, from another replica:

```
 let (l_clock, ip) be the timestamp of m, $l\_clock_r$
 be the logical clock of r and, $ip_r$ be IP address of r
     if $(l\_clock, ip) > (l\_clock_r, ip_r)$
     then $l\_clock_r \leftarrow l\_clock+1$
     else $l\_clock_r \leftarrow l\_clock_r+1$
```

A MsgList on a replica holds received clients' service requests before the requests are processed by the replica. A ProcessedList is used to record the requests that have been executed by the replica. op(m) denotes the operation that is invoked by the client's service request m. l_clock is the logical clock value of the RTC and ip is the IP address of the RTC. ip is also used as the ID of the RTC.

S3 when receive a request, m, from a client:

- m.init_receiver ← ip
- update l_clock according to S1
- m.timestamp ← (l_clock, ip)
- multicast m to all the replicas (including itself) in the service group

When a RTC receives a client's request, say m, the RTC sets the init_receiver attribute of m to indicate that the RTC will be responsible for returning the response to m to the client (line 1 of S3). m is given a timestamp (line 3 of S3). Since l_clock increases monotonically (S1 and S2), m's timestamp is larger than the timestamps of any other requests on the RTC. Then, m is multicast to all the other RTCs. So that it can be executed on all the replicas.

S4 when receive a multicast request m:

- update l_clock according to S2
- generate an acknowledgment, ack and, ack.timestamp ← (l_clock, ip)
- send ack to m.init_receiver
- let wrong_set = {msg | (msg.timestamp > m.timestamp) $\wedge$ (msg is in ProcessedList)}
- for each msg such that msg $\in$ wrong_set do
- (1) undo op(msg)
- (2) remove msg from ProcessedList and add msg to MsgList
- end-for
- add m to MsgList and sort MsgList into ascending order according to the timestamps of the messages in MsgList

When a RTC, say r, receives a multicast request from another RTC, r generates an acknowledgment message, ack and assigns a timestamp to ack (line 2 of S4). According to line 1 of S4 and S2, the timestamp assigned to ack is greater than the timestamps of all the requests previously received by r. m's sender is m.init_receiver (line 4 of S3). ack is sent back to m's sender (line 3 of S4). ack helps m's sender to decide whether m has been executed in the correct order. Since all replicas should execute the requests in the order determined by the timestamps of the requests (i.e., the requests with smaller timestamps should be executed before the requests with larger timestamps), r needs to check whether any requests have been executed in a wrong order. Set wrong_set contains all the requests that have been executed in a wrong order (i.e., the requests whose timestamps are greater than m's timestamp and have been executed before m is received). For all the requests that have been executed in a wrong order, the operations triggered by these requests are undone (line 6 of S4) and these requests are added to MsgList for re-execution (line 7 of S4).After changes are made to MsgList, the requests in the list are re-sorted to ensure that they will be delivered to the replica in ascending timestamp order (line 9 of S4).

S5 when receive an acknowledgment for message m:

- m.ack ← m.ack+1
- if (m.ack = total) and (m is in ProcessedList)
- send the result of op(m) to the client that sends m
- end

m.ack (line 1 of S5) records the number of acknowledgments received for a multicast request m. total (line 2 of S5) represents the number of replicas in a service group. Assume that (a) a RTC, say p_1, multicasts a request m to a RTC, say p_2 and, (b) p_2 sends multicast messages $m'_1, \ldots, m'_n$ to p_1 before sending the acknowledgment for m to p_1. Since the communication channels between the replicas have the FIFO property, when p_1 receives m's acknowledge from p_2, p_1 must have received $m'_1, \ldots$, m'n sent by p_2. According to line 4-8 of S4, when p_1 receives $m'_1, \ldots, m'_n$, p_1 has carried out the operations to ensure that m and $m'_1, \ldots, m'_n$ are executed in the correct order on p_1. In other words, if m'i.timestamp < m.timestamp where $1 \leq i \leq n$, p_1 would have scheduled m'_i to be processed before m. According to S1 and S2, it can be seen that if p_2 multicasts a request msg after sending the acknowledgment for m, then msg's timestamp must be greater than m's acknowledgement's timestamp.

Thus, if p_1 has received the acknowledgments for m from all the replicas, p_1 knows that it has scheduled to execute all the requests whose timestamps are less than m.timestamp before m. Hence, p_1 knows that m's execution order is correct. This is because clients' requests are executed in their timestamps' order. As a result, if the replica has completed the execution of m, p_1 can return the result of the execution to the client (line 2-4 of S5).

S6 when receive the result of op(m) from the replica:

- cache the result of op(m) and, add m to the end of ProcessedList
- if ((m.ack = total) and (m.init_receiver = ip)) send the result of op(m) to the client
  end-if
- if MsgList is not empty
- let fm be the first request in MsgList
- remove fm from MsgList and send fm to the replica
- end-if

When the replica completes the processing of a client's request, the RTC stores the result of the processing to cope with possible failure of the RTC that receives the client's request. If the RTC is responsible for sending the result back to the client (i.e., m.init_receiver = ip) and the RTC has received the acknowledgments for the request from all the replicas, as explained for S5, the result of the processing can be sent to the client (line 2 of S6). Then, the next message in MsgList is sent to the replica for processing (line 3-6 of S6). After being processed by the replica, the request messages are added to the ProcessedList.

Thus, the list might grow infinitely. To avoid this problem, the RTCs periodically broadcast a list of messages that have been acknowledged by all the RTCs. For a message m, if m and all its predecessors in ProcessedList have been acknowledged, m and its predecessors can be removed from ProcessedList.

Design details of distributed coordination method: The client request together with the flag field is passed to the RTC of the one replica. The RTC receiving the client request is responsible to multicast the request to all the server replicas actively in the group. Let us consider the client request r_1, together with the flag field. (Client _ req_1+Flag+Receiving time of the request in the RTC, IP_c) Where IP_c is the IP address of the client. The time stamp[17] of the incoming request is calculated using the value of the Time scheduler of the respective RTC. The Time Scheduler preserve the incoming time of the last client request (L_r).

For example, the time stamp value of RTC_i is T_i. T_i = Incoming time of the last request (L_r) ~ Incoming time of the recent client request. This is the way the time stamp value is calculated in each RTC and compared with the time stamp values of other RTCs. The time stamp value of client $request_1$ for the different RTCs

Ti < T_{i+1} < T_{i+2} < T_{i+3} ….. then Ti is selected. The time stamp value of the RTC is least, it will act as a primary RTC. The client request is executed by the primary RTC-server replica and the value is updated to the other replicas actively connected.

As shown in Fig. 2a and b, the replicas update the value according to primary, thus consistency is maintained in all the replicas. This method avoids the time delay raised by sending the state information from client to server.

The implementation of the system: Requests sent by clients are included with the flag field. Each client request has a unique ID. The ID is used for detecting possible duplicated clients' requests in the event of replica failure. Each RTC consists of two modules, i.e., a Message Handler (MH) and a Failure Detector (FD) as shown in Fig. 3. A client application program interacts with an MH to exchange requests and responses. The MH is responsible for (a) handling the request messages received from clients, (b) recording the IDs of the messages received from clients, (c) holding copies of the responses to clients' requests and sending responses back to the clients (if necessary), (d) running the TS protocol to interact with the MHs of other RTCs and, (e) handling failure of the replicas.

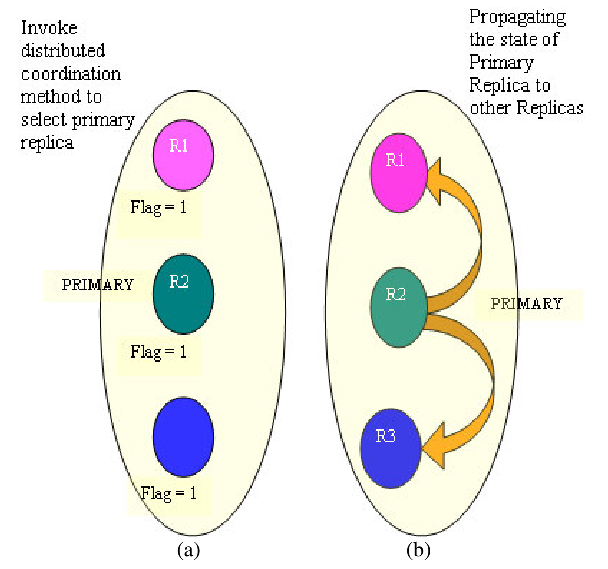
Invoke distributed coordination method to select primary replica
R1
Flag = 1
PRIMARY
R2
Flag = 1
Flag = 1
(a)
Propagating the state of Primary Replica to other Replicas
R1
R2
PRIMARY
R3
(b)

Fig. 2: (a): Nodes R1, R2 and R3 form a coordination group; (b): The state of primary replica is propagated to other replicas

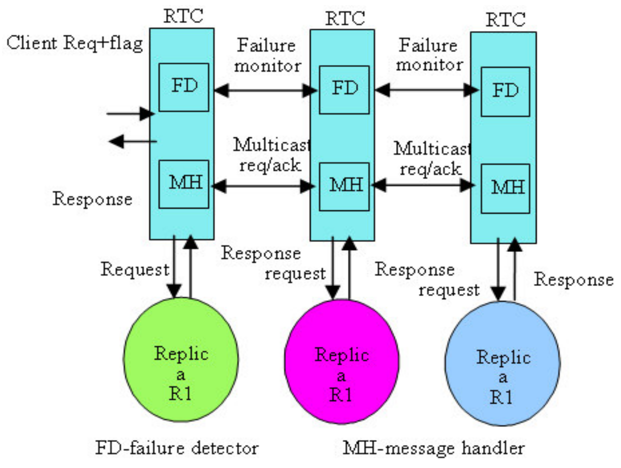

Fig. 3: Middleware for replicated client server application

The FD is responsible for monitoring the failure of the other replicas. The FD is implemented as a class ◊ S failure detector[7]. The MH puts the received clients' requests in the MsgList. The MH sends the request in the MsgList to its corresponding replica one at a time. That is, a request is not sent to the replica until the response to the previous request in the list is received from the replica. This ensures that the requests are executed by the replica in the order determined by the TS protocol. Only the service requests sent to the same operation need to be ordered. Requests sent to different operations do not need to be ordered. Thus, for each service operation offered by the replicas, the MHs run a TS protocol thread to multicast and order the requests sent to the operation. Each operation has its own MsgList set up on each of the MHs. Thus, requests for different operations can be sent to the replicas simultaneously as long as these requests are ordered in their respective MsgLists.

When an MH receives a response from its replica, it stores the response in its buffer in order to handle possible failure. An MH receives a client's request either (a) directly from the client or (b) from another MH. Case (b) occurs if the client sends the request to a different MH in the service group; and, consequently, the request is multicast to all other MHs in the service group. As a result, each MH also receives requests that are not directly sent to it by clients. When a response to a client's request is received, the MH that receives the client's request directly is responsible for returning the response to the client. After delivering the response to the client, the MH asks the other RTCs to delete the response from their buffers. Clients need to handle the failure of a replica in the sense that the clients need to connect to another replica in the service group. If a replica fails before a client sends its request, the failure is discovered when the client attempts to connect to the replica's RTC and fails in its attempt. In this case, the client will send its request to another replica's RTC. If the RTC fails, the client loses the connection to the RTC. In this case, the client attempts to establish a connection with another replica's RTC in the service group.

After a client connects to another RTC, the client resends its request. Re-sending the request is necessary. This is because the failed RTC might fail before it multicasts the client's request to other RTCs. However, the resending of the request might result in duplicated request since the failed RTC might have multicast the client's request to other RTCs before it fails. To cope with message duplication problem, when an MH receives a request, it uses the ID of the request to check whether the same request has been received previously. If the request has been received previously and been processed in the failure recovery phase, the MH does not multicast the request to the other replicas. In this case, the response to the request will be sent to the client when the response is available. The FDs monitor whether a replica fails by exchanging messages with each other. When a replica fails (i.e., the replica's FD does not respond to other FDs' messages), the RTCs enter the recovery phase. During the recovery phase, the replicas do not accept any client request. In the recovery phase, the MHs run the consensus protocol in[4] to agree on the execution order of the clients' requests that have been received by the MHs. These requests are marked as having been sorted. When the responses to these requests become available, the replicas that receive the requests from the clients directly can send the responses to the clients immediately. The recovery phase ends after the MHs reach agreement on the execution order of the clients' requests.

Re-execute contaminated non-determinism: Another technique to maintain consistency among the replicas by executing all the possibilities of compensation snippets. The divergence state of the replica is nullified by executing the compensation snippets. We insert prepared portions of code that can be executed to re-generate the contaminated non-determinism, if provided the pure non-determinism (i.e., the origin of the contamination) as an input. Each of the replicas are requested to perform compensation, before processing the next request, by first setting the pure nondeterministic part of its state to the received nondeterministic struct and then re-executing the inserted code-snippets to regenerate the corresponding contaminated non-determinism. At the end of this compensation, each replica is consistent and is ready to process the current request.

Compared to transfer-contam (the state transfer between client to server), the reexec-contam technique should incur lower communication overheads due to the reduced amount of nondeterministic state being piggybacked back and forth; however, the tradeoff is that run-time latency is increased by the reexection of the compensation snippets at the server side. Also, reexec-contam requires more compile-time analysis and source-code modification to the server-side than transfer-contam. This is because additional control-flow passes are needed to isolate the code that encapsulates the contaminated nondeterministic state. The client-side code is the same as in transfer-contam. Obviously, reexection is justified when the compensation overhead is out-weighed by the communication overhead of the transfer techniques.

RESULTS

Communication overhead is reduced in our method because; there is no state transfer between server replica and the client. The state transfer over head is directly proportional to the amount of actual non-determinism that exists within the application, e.g., if only 5% of the application is actually nondeterministic, our compensation overheads should be incurred only for that portion of the application. After the compensation is performed in the primary, its state is propagated to all the actively connected replicas. The total delay is the combination of actual delay incurred during the execution of compensation snippets and the delay involved while propagating the primary state to all other server replicas.

We conducted our experiments in the distributed environment, with homogeneous test-bed nodes. Each node run the Linux operating system on a 2.8 GHz-64 bit AMD processor, 256 KB cache and 512MB RAM over a 100Mbps LAN. In our experiments, we do not load the nodes with any other running programs other than RTC, our micro-benchmarks and the native OS utilities that typically run on each node. Each replica runs on separate node. We evaluate a number of metrics (communication overhead, compensation overhead, server-side processing time and round-trip time) under fault-free conditions.

Methodology: In our experiments, we vary the following low-level parameters:

- Replication style:
  - Active
  - Semi-active replication
- Replication degree: 1, 2, 3 or 4 server replicas
- Number of clients: Single client
- Percentage of contamination. (10, 20, 30 and 40%)

Tested for the following bench marks:

- Lotus (Base line bench mark)
- Compensation technique

(a) State propagation (Maintain the determinism among the replicas, the state of the primary replica is propagated to all the replicas actively connected.)
(b) Reexection of compensation snippets

Request arrival rate: The clients insert a pause of 0, 0.5, 2, 8 or 32 ms. The lack of a pause (0 ms) represents bursty client activity.

Micro-benchmarks: We have developed two micro-benchmarks to compare our various compensation techniques. The two micro-benchmarks are identical in many way, they both constitute a two-tier application, i.e., with a single client and a single replicated server. Both micro-benchmarks use multi-threading with homogeneous threads, identical code at each of the server replicas (except for the fact that each replica stores a unique, hard-coded server_id SID) and identical initial state to start out with. Each micro-benchmark contains an array of 10,000 longs that represents its state. Pure non-determinism involves generating a random number and assigning it to one of the elements in the array. Contaminated state is subsequently created by performing arithmetic on the random number and assigning the result to another element in the array.

The server state is changed in different ways: Varying the pure non-determinism (contamination) to 10, 20, 30 and 40%. For each value of pure non-determinism, we vary the amount of contaminated non-determinism to 10, 20, 30 and 40%. For each of the above state combinations, we evaluate each of our compensation techniques i.e., execution of propagation snippets and reexec-contamination and comparing with the existing technique. This is clearly depicted in the graphs. Note that we can compare all of the techniques for a given x% of non-determinism. The Lotus case simply serves as a baseline for performance comparison. We also vary other parameters, such as the number of replicas (1-4), amount of multithreading (2-6 threads) and amount of state (100, 1000 and 10,000 longs).

Empirical observations: We observe the effects on the round-trip time when increasing the amount of contaminated non-determinism and increase the number of replicas within the micro-benchmark. The amount of contamination is gradually increased by 10, 20, 30 and 40% and tested for all micro benchmarks. Second the non-determinism for these results is fixed at 30% and the number of replica is gradually increased. Note that the algorithm has a significant amount of processing time. This is readily visible when comparing these results with propagation technique and re-execution of compensation technique.

Varying amount of contamination: Figure 4 shows the effect on the roundtrip time of increasing the amount of contaminated non-determinism within the micro-benchmark. The amount of pure non-determinism for these results is varied based on the percentage of contamination and 3 replicas are used. Because pure nondeterministic state is handled identically across all of our various techniques, the graph demonstrates how each technique handles an increase in contaminated state.

The processing time increases slightly across all techniques because additional work is done due to the increased amount of contaminated state. However, in our approach the processing time is relatively small compared to the communication overhead of passing the entire state back and forth of client to server. We eliminate the communication overhead by avoiding the state transfer between client and servers and allow the servers to communicate with each others through distributed coordination method.

The most interesting observation here is due to the fact that communication overhead does not dominate processing time. For instance, with the following percentage of ex. 10, 20 and 30% contamination, our approach shows the lower overhead comparing to transfer-ckpt by transferring the state of any one replica to other replicas actively connected by invoking distributed coordination method. Transfer-ckpt appears to have higher overheads because of transferring the state between client and servers. Reexec-contam comes under next level of overheads. This is because the increased processing time outweighs the communication overhead for lower amounts of contaminated states.

Varying degree of replication: As shown in the Fig. 5, the amount of pure and contaminated non-determinism is constant, but the number of replicas is varied. For every additional replica, the communication load increases because all of the replicas send their nondeterministic state, along with their responses, to the client in case of transfer-ckpt. But in the method we suggested, the communication over head is only due to the propagation of the state of one primary replica to other replicas.

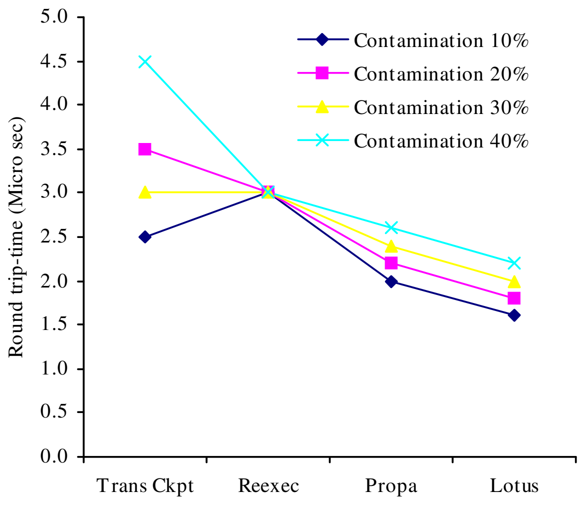

Fig. 4: Benchmark of RTT for increasing percentage of contamination

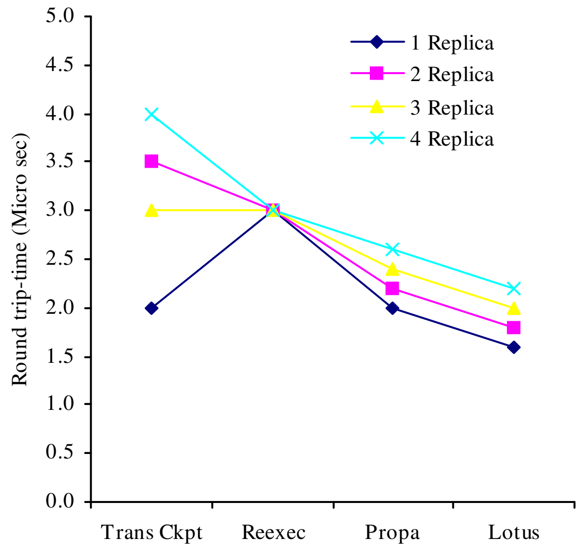

Fig. 5: Benchmark of RTT for increasing number of Replicas

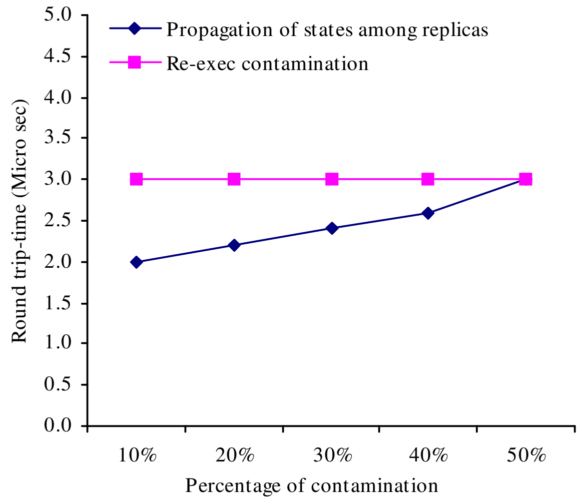

Fig. 6: Cross-over between the propagation of state and the reexec-contam technique for increasing contamination percentage

We can observe from the Fig. 6, the Cross-over performance between the propagation of states to all the replicas and re-execution of snippets in each replica. Our technique propagation of state is dominated in all aspects when comparing to re-execution of snippets.

DISCUSSION

Existing approach for handling non-determinism is mentioned as follows. Joseph Slember and Priya Narasimhan[14] perform the static analyzing of source code and list the variables and system calls (MEAD[12] approach) which lead to nondeterministic state in the server replicas. These variables and system calls are sent to the servers as a client request. By executing the client request, the server replica goes to non-deterministic state. The state of any one of the server replica is piggybacked to client and it is send to all the actively connected replicas through group communication protocol. The replicas execute the dynamic snippets in order to reduce the divergence raised with the received replica state. After the execution of snippets the state of all replicas are identical and consistent (Deterministic state). In this method the snapshot (State information) of one replica is taken and it is spread to all the replicas. Based on the state information of one replica, all the replicas adjust their state.

The delay which the snapshot has taken in one server and it is piggybacked to the client, (transfer ckpt, transfer contam) from the client it is sent to all the servers. The delay is more when the percentage of contamination is more, because it will take more time to transfer the contamination state from server to client and from client to all the servers. Reexection of dynamic snippets are also used when the transfer of checkpoint, transfer of contamination dominate more communication delay. Gaifman[9] targets non-determinism that arises in concurrent programs due to environmental interaction. This technique involves backup replicas lagging behind the primary to ensure consistency. The technique is transparent to the user, but the application is actually modified by transformations that handle multithreading.

The Multithreaded Deterministic Scheduling Algorithm[10] aims to handle multithreading transparently by providing internal and external queues that together enforce consistency. The external queue contains a sequence of ordered messages received via multicast, while each internal queue focuses on thread dispatching, with an internal queue for each process that spawns threads. Basile[2] addresses multithreading using a preemptive deterministic scheduler for active replication. The approach uses mutexes between threads and the execution is split into several rounds. Because the mutexes are known at each round, a deterministic schedule can be created. This approach does not require any communication between replicas. Hypervisor-based fault tolerance[5] involves a virtual machine that ensures that all non deterministic data is consistent across the replicas.

Delta-4 XPA's semi-active replication[1] addresses non-determinism through a hybrid replication style that employs primary-backup replication for all non-deterministic operations and active replication for all other operations. In SCEPTRE 2[3], non-determinism arises from preemptive scheduling. Semi-active replication is used, with deterministic behavior enforced through the transmission of messages from a coordination entity to backup replicas for every non-deterministic decision of the primaries. Similarly, Wolf's piecewise deterministic approach handle non-determinism by having a primary replica that actually executes all nondeterministic events, with the results being propagated to the backups at an observable, deterministic event.

TCP tapping[16] captures and forwards non-deterministic execution information from a primary to other replicas. The backup replicas gain information from the primary after it has done the work. The approach is transparent, but involves setting up routing tables to snoop on the client-to-server TCP stream, with the aim of extracting the primary's non-deterministic output. The solution involves the interception of I/O streams of replicas and the appropriate handling of input and output streams. In this study a new attempt is proposed to reduce the communication delay and improve the quality of service in replicated middleware applications.

CONCLUSION

We present RTC; a new approach, handling non-determinism in distributed, replicated applications using distributed coordination method by exploiting static program analysis on the application's source code and identifies the sources of non-determinism within the application. We describe two different techniques; one that involves the state of the primary replica is propagated to all other server replicas. Another that involves reexection of contaminated non-deterministic code. We can support even the active replication of non-deterministic applications in this manner. Our empirical evaluation involves various performance-sensitive techniques by varying amount of contamination and increasing number of replicas for distributed middle-ware micro-benchmarks that contain various sources (multi-threading, system calls and contamination) of non-determinism. We note that our current implementations of the propagation of state, multi-tier applications and nested end-to-end requests introduce increased complexity in handling non-determinism, especially with actively replicated tiers.

The propagation of non-deterministic state is no longer contained at the client or at any one tier. We need to handle any non-deterministic state or execution that propagates to other tiers. This is especially evident when a failure occurs during an end-to-end request, resulting in some of the replicas at every tier becoming inconsistent. Multiple clients are complicate in back-and-forth compensation technique. But the method described in this study has no complication because there is no transfer of back-and-forth compensation of non-determinism and we would then require coordination across clients or some alternative way of ensuring consistency across multiple clients. Both multi-tier and multi-client fault-tolerant architectures are part of our ongoing research on the scalable compensation of non-determinism, but remain outside the scope of this study.

REFERENCES

1. Barrett, P., P. Bond, A. Hilborne, L. Rodrigues, D. Seaton, N. Speirs and P. Verissimo, 1990. The delta-4 Extra Performance Architecture (XPA). Proceeding of the Symposium on Fault Tolerant Computing, June 26-28, IEEE Xplore, Newcastle, UK., pp: 481-488. DOI: 10.1109/FTCS.1990.89386
2. Basile, C., Z. Kalbarczyk and R. Iyer, 2003. A preemptive deterministic scheduling algorithm for multithreaded replicas. Proceeding of the International Conference on Dependable Systems and Networks, June 22-25, IEEE Xplore, San Francisco, CA., pp: 149-158. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1209926
3. Bestaoui, S., 1995. One solution for the non-determinism problem in the SCEPTRE2 fault tolerance technique. Proceeding of the 7th Euro Micro Workshop on Real-Time Systems, June 14-16, IEEE Xplore, Odense, Denmark, pp: 352-358. DOI: 10.1109/EMWRTS.1995.514332
4. Birman, K.P., 1996. Building Secure and Reliable Network Applications. 1st Edn., Manning Publications Co., USA, ISBN: 10:884777295, pp: 591.
5. Bressoud, T.C. and F.B. Schneider, 1996. Hyper visor-based fault-tolerance. ACM Trans. Comput. Syst., 14: 80-107. http://www.cs.cornell.edu/fbs/publications/HyperFTol.pdf
6. Westernwares Product Version 6.2. Introduction to CC-Rider. the source code analyzing tool kit. http://www.westernwares.com/info/info.htm
7. Chandra, T.D. and S. Toueg, 1996. Unreliable failure detectors for reliable distributed systems. J. ACM., 43: 225-267. http://portal.acm.org/citation.cfm?id=226647

8. Frolund, S. and R. Guerraoui, 2000. X-ability: A theory of replication. Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing, June 16-19, ACM Portland, Oregon, United States, pp: 229-237. http://portal.acm.org/citation.cfm?id=343623
9. Gaifman, H., M.J. Maher and E. Shapiro, 1991. Replay, recovery, replication and snapshots of nondeterministic concurrent programs. Proceedings of the 10th annual ACM symposium on Principles of Distributed Computing, Aug. 19-21, ACM Montreal, Canada, pp: 241-255. http://portal.acm.org/citation.cfm?id=112600.112621
10. Jimenez-Peris, R. , M. Patino-Martinez and S. Arevalo, 2000. Deterministic scheduling for transactional multithreaded replicas. Proceeding of the 19th IEEE Symposium on Reliable Distributed Systems, Oct. 16-18, IEEE Xplore, Nurnberg, Germany, pp: 164-173. DOI: 10.1109/RELDI.2000. 885404
11. Lamport, L., 1978. Time, clocks and the ordering of events in a distributed system. Commun. ACM, 21: 558-565. http://portal.acm.org/citation.cfm?id=359563
12. Narasimhan, P., T.A. Dumitras, S.M. Pertet, C.F. Reverte, J.G. Slember and D. Srivastava, 2005. MEAD: Support for real-time fault-tolerant CORBA. Concurrenc. Comput. Pract. Exp., 17: 1527-1545. http://portal.acm.org/citation.cfm?id=1085001
13. Narasimhan, P., L.E. Moser and P.M. Melliar-Smith, 1999. Enforcing determinism for the consistent replication of multithreaded CORBA applications. Proceeding of the 18th Symposium on Reliable Distributed Systems, Oct. 19-22, IEEE Xplore, Lausanne, Switzerland, pp: 263-273. DOI: 10.1109/RELDIS.1999.805102
14. Slember, J.G. and P. Narasimhan, 2006. Living with Non-determinism in replicated Middleware systems. Proceeding of the ACM/IFIP Conference on Middleware, Nov. 27-Dec. 1, Melbourne, Australia, pp: 81-100. http://www.pdl.cmu.edu/PDL-FTP/stray/slember-middle06.pdf
15. Object Management Group, 2000. Fault Tolerant CORBA Specification, V1.0. ftp://ftp.omg.org/pub/docs/ptc/00-04-04.pdf
16. Orgiyan, M. and C. Fetzer, 2001. Tapping TCP streams. Proceeding of the IEEE International Symposium on Network Computing and Applications, Oct. 8-10, IEEE Xplore, Cambridge, MA., USA., pp: 278-289. DOI: 10.1109/NCA.2001.962544
17. Xinfeng, Y., 2007. Providing reliable web services through active replication. Proceeding of the 6th IEEE/ACIS International Conference on Computer and Information Science, July 11-13, IEEE Computer Society, Washington DC., USA., pp: 1111-1116. DOI: 10.1109/ICIS.2007.151
18. Schneider, F.B., 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Survey, 22: 299-319. http://DOI.acm.org/10.1145/98163.98167
19. Slember, J.G. and P. Narasimhan, 2004. Exploiting program analysis to identify and sanitize non-determinism in fault-tolerant, replicated systems. Proceeding of the Symposium on Reliable Distributed Systems, Oct. 18-20, Florianopolis, Brazil, pp: 251-263. DOI: 10.1109/RELDIS.2004.1353026
20. Slember, J.G. and P. Narasimhan, 2006. Non-determinism in ORB: The perception and the reality. Proceeding of the 17th International Conference on Database and Expert Systems Applications, Sep. 4-8, Krakow, pp: 379-384. http://portal.acm.org/citation.cfm?id=1155785
21. Ye, X. and Y. Shen, 2005. A middleware for replicated Web services. Proceedings of the IEEE International Conference on Web Services, July 11-15, IEEE Computer Society, Washington DC., USA., pp: 631-638. http://portal.acm.org/citation.cfm?id=1090954.1092072&coll=&dl=