RTRE^{X}: Trace and Replay Engine for Debugging Java RMI-Based Applications

Aditya Akella Ashwin Bharambe Vahe Poladian

{aditya, ashu, poladian}@cs.cmu.edu
Computer Science Department,
School of Computer Science,
Carnegie Mellon University.

Abstract

Debugging distributed programs is an arduous task because the asynchrony and unpredictability of interconnecting network results in system state which is hard to reproduce. These problems make traditional methods of debugging ineffective for distributed applications. In this paper, we describe the design and implementation of RTRE^{X}, a trace and replay based distributed debugger for the Java RMI system. In the replay phase, one component is re-executed while the rest of the system is simulated using the traces recorded in the record phase. We envision the system to be used as a debugging aid to help removal of data race conditions like deadlocks; as an automated test harness for testing RMI components; as a simulator for remote components in case they are unavailable, etc. We measured time overheads ranging from 50% to 150% during the recording phase, but believe that such overheads are acceptable.

1 Introduction

To err is human. This is what makes bugs or errors in any program, no matter how simple it is, a near certainty. Thus debugging has been, and is going to be an integral part of any software development process.

A natural way of debugging any application is to retrace its execution and examine the state of the computation while it is executing. However, writing such a tool for debugging becomes a challenging task when it comes to distributed applications. The asynchrony and unpredictability of the underlying network result in system state which is hard to reproduce. Thus, the output produced by a program is not just a function of the inputs, but also depends upon the ordering of events across the system. What this means is that the corresponding errors are also extremely hard to reproduce. Given these reasons, traditional solutions like using **printf** statements, or using tools like **gdb** are ineffective. This paper is a modest attempt at exploring some ideas for implementing an easy-to-use distributed debugging aid which can help capture a reasonable range of errors.

When building a replay tool, the fundamental question we need to answer is: *Upto what abstraction is the replay equivalent to the original execution ?* For example, one might just choose to reprint the final output produced by the program. Clearly, this does not convey any useful information which could aid debugging! On the other hand, one might reproduce the execution to the granularity of a processor instruction preserving *all* timing intervals between *all* events. This however is exceedingly (and perhaps, impossibly) hard to implement since during the replay phase, a replay 'control' process has to essentially share resources with the 'replayed' process which is not the case in the original execution.

The challenge, then, is to identify a level of abstraction which reveals enough *useful* information about the state of a distributed computation, yet is implementable. In this project, we define our abstraction to be

the *ordering* of Remote Procedure Calls calls in a distributed system. Thus, only the *ordering* of RPC calls is guaranteed to be the same across the original execution and the replay. No guarantees are made about the time intervals between the events. We use the Java Remote Method Invocation (RMI) implementation of RPCs as the base for our prototype.

We also envision that a distributed system would typically be debugged in a manner which is a natural extension of single process debugging, viz. by concentrating on and instrumenting *one* machine or thread and observing results.

The tool we describe, called $\mathsf{RTRE}^{\mathsf{X}}$, captures the trace of all RMI calls across the system in the recording phase. The replay phase employs a novel idea: the programmer first specifies a machine which is going to be actively debugged. The system then executes the selected component; but the rest of the distributed system is 'simulated' by presenting *identical asynchronous events*[1] to the selected component.

The modifications are all performed in the Java Runtime Environment which means that the programmer is freed from any manual instrumentation. Also, by balancing 'simulation' with 're-execution', $\mathsf{RTRE}^{\mathsf{X}}$ functions as an automated and controlled test harness.

The paper is organized as follows. In Section 2 we discuss work related to ours and show the similarities and differences of our design with past work. Section 3 discusses the design of the $\mathsf{RTRE}^{\mathsf{X}}$ in detail. In Section 4 we present our evaluation methodology and experimental validation results. Finally, Section 5 summarizes the paper.

2 Related Work

Bates [1] describes an event based behavioral abstraction (EBBA) model in which debugging is treated as a process of creating models of expected program behaviors and comparing those to the actual behaviors exhibited by the program. While providing for a high level abstraction for defining models of behavior, it leaves on the programmer the burden of defining the accurate behavior, which might be difficult for the programmer to formalise *a priori*. The EBBA system, also does not provide any capability for replay of events, and hence, our project can be viewed to provide a complimentary functionality.

BugNet system, described by Wittie et al [2] is a distributed trace and replay system, which is based on global synchronization of clocks and uses periodic checkpointing for replaying the system as accurately as possible. Unlike this work, our project is based on Lamport's happened-before relationship[3] and event ordering, and does not attempt to preserve the time intervals between the events. Also, Bugnet work traces events such as IPC (inter-process communication) and file I/O, whereas our system focuses on RMI events.

Mittal et al [5] describe a solution to the *predicate control problem* which takes a generic distributed computation and a *safety property* specified on the computation, and outputs a controlled computation which maintains the property. The design underlying our system can be viewed as a specialised case of the above model, where the safety property is the ordering of RPC calls across the system.

Instant Replay [4] is an algorithm developed by LeBlanc and Mellor-Crummey to replay shared-memory parallel programs. Instant Replay assumes that all accesses to shared variables are guarded. The replay is guaranteed by recording the order in which variables are accessed, rather than recording the data that is accessed. Several adaptations of Instant Replay were proposed to debug other types of non-deterministic constructs.

Our system is novel in the sense that it uses a unique combination of 're-execution' and 'simulation' to provide a replay. We are not aware of any previous work which concentrates on a single machine in the replay phase. While simplifying issues related to total ordering of events, this also maps well to real-world debugging needs.

Note however, that $\mathsf{RTRE}^{\mathsf{X}}$ critically depends on thread level synchronization outcomes if the clients are multi-threaded. If we were to make assumptions made in systems like Instant Replay, this dependency would be easily satisfied and a fully correct replay would be guaranteed.

[1] *identical* is defined by the level of abstraction

As noted by Ronsse et al [6], replay systems face several challenges, including non-determinism at various levels. Our system, by the virtue of replaying events in the same order that they occured, eliminates non-determinism in the ordering of the events. However, it is important to note that we do not provide any mechanisms for dealing with non-determinism that may occur inside a system during the replay phase. For example, if the system being debugged makes decision based on system calls such as rand(), or getpid(), then the replay may not exactly follow the actual execution.

In the Java RMI implementation, a simple call-logging facility for debugging is supported which simply logs the headers of RMI-server side method calls along with timestamps. But there are no current plans [8] to support a full-featured, interactive, remote debugger. By integrating our replay engine into a traditional java debugger, we can easily support this facility.

Our system can be used as a testing tool to dig out various race conditions in a distributed system by *explicitly forcing* specific orderings of RMI calls. In this manner, our system extends the Eraser dynamic data race detection [7] work to a distributed environment.

3 Design of RTRE^{X}

Our prototype, RTRE^{X} is a tracing and replay facility for Java RMI. Java RMI is a remote procedure call infrastructure which allows building and deploying distributed applications using a standard set of development and runtime tools provided as part of the standard Java Runtime Environment.

The RTRE^{X} debug facility is composed of two main phases at the very high level:

- **Recording:** Execution of the instrumented application which captures relevant trace information, and
- **Replay:** Forcing a specific component of the system through the ordering defined by recorded RMI events.

As an example, consider the following scenario which describes a typical but simplified debugging setting. The application developer has three processes: process A, process B, and process S. Process S hosts the RMI server object, and A and B act as RMI client objects. Some interactions or inter-leaving of processes A and B cause S to get into undesired states, exhibiting such behavior as data race conditions. We enable the developer to perform the following debugging actions:

- trace the execution of the entire system up to a certain point in time,
- bring down all the components,[2]
- bring up a specific component, e.g. Process S in replay mode,
- replay the events that were generated by the clients A and B in a controlled manner. Process S would typically be brought up in debug mode, and the developer would have the ability to step through the server method executions, as many times as he wishes, in response to simulated client calls.

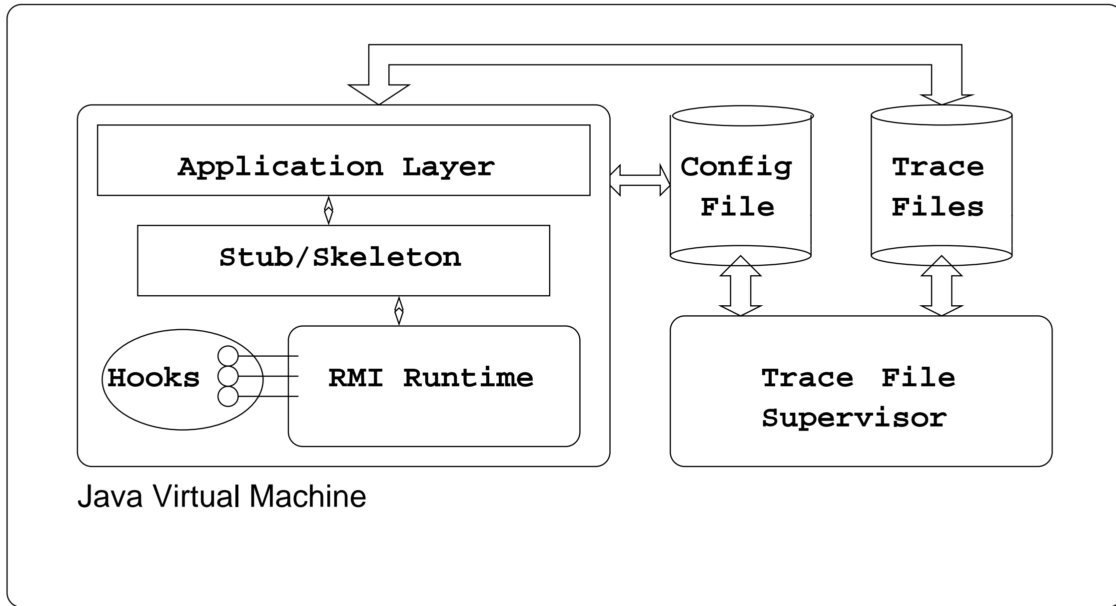
A replay can be *defined* to be an execution E' which is similar to the reference execution E up to a certain level of abstraction. The abstraction that we present in this system is the ordering and the contents of RMI remote procedure calls. Thus, we guarantee that the RMI calls will be executed in the same environment and their ordering will be the same as in the initial execution. We use event causality as defined by Lamport's happens-before relation[3] to order events. An alternative approach to ordering events is to use physical timestamps. Physical timestamps gurantee an ordering of events only if certain synchrony assumptions are made about the interconnecting network. Also, in the single system replay model, since we are only concerned with reproducing the *order* of events, a physical timestamp does not provide more information than a simple logical clock.

[2] relevant machine processes in the distributed system

3.1 The Gory Details

In what follows, we describe our design, lay out the design issues and the choices we made.

3.1.1 RTRE^{X} Module Decomposition



Recording Phase: Relevant Components

Figure 1: Hooks in the RMI Runtime capture RMI events (client and server) and call state. Also, they insert logical timestamps into the events. Traces are flushed into files in a specific directory. The Trace File Supervisor serves these files to the Replay Engine during the replay phase.

The RTRE^{X} system consists of the following functional modules:

- **Hooks:** These are method interceptors which capture *i.* RMI client calls, *ii.* RMI server up-calls and *iii.* a small set of RMI administrative calls. These are the basic means by which RMI event environment and logical timestamps are recorded.
- **Trace File Supervisor:** This is a separate daemon process running on each machine in the system, which provides a socket based service for accessing and manipulating trace files.
- **Global Control Module:** The core of this is a discrete event simulator which consumes events from a processed list of events obtained from the Trace File Supervisor.

Fig. 1 and Fig. 2 depict these components and their interactions.

In our system, we store the remote reference name, method and args (which together constitute the environment) plus the logical timestamp, thread name and a few other bits. We flush all this information into a file, one per each virtual machine.

We implemented the replay engine as a discrete event simulator. In our implementation, we bootstrap the original program, let it initialize via RMI normal administrative calls. After the initialization, it returns the control back to the discrete event simulator. Then, the discrete event simulator consumes events from the trace database, and makes direct calls on the correct objects. This effectively simulates the RMI runtime.

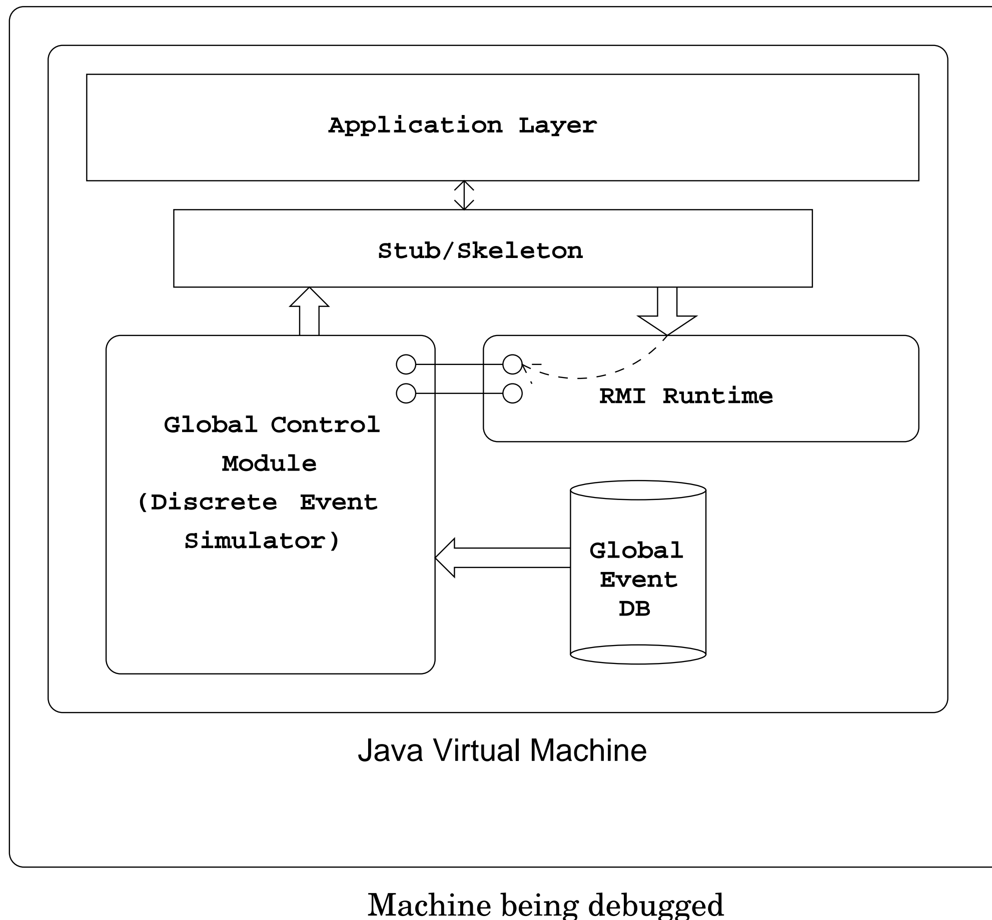


Figure 2: The Global Control Module classes are loaded into the Virtual Machine being debugged. The GCM contacts the Trace File Supervisors on all machines and collects the trace information which is then processed and put into the Global Event Database. GCM acts as a Discrete Event Simulator enforcing the orderings as specified by the Event Database.

For correct simulation, we spawn a worker thread for each incoming RMI request – just like the RMI system does. It is worth mentioning that the RMI system does not do thread re-use, which somewhat simplyfies our job. To preserve event ordering, we control when the thread goes into a method and exits from it, at the RMI-level. This implies that the thread is not under our control after this point. Within that thread, we handle all the work that was relevant to that request.

Here, we would like to emphasize one novel feature of RTRE^{X} that is an upshot of our design. Since our modifications are made into the RMI runtime system, and the user-level API is completely unchanged, we are able to provide a debugging experience for the user which does not require separate instrumentation or compilation. The user only needs to provide run-time flags to the Java Runtime.

3.1.2 Java Packages: Choices

For the sake of completion, we would like to mention that we used the following packages in the construction of RTRE^{X}:

- Java Development Kit version 1.2.2_08 from Sun Microsystems, binary version

- Source Code for the same JDK

We decided to standardize on 1.2.2 because this is a fairly stable version ensuring minimal risk to our project. Furthermore, this was the first version for which a stable Linux implementation was available. Lastly, the source code for this version is available under Sun Community Source License. The relevant packages are: `java.rmi.*` and `sun.rmi.*`.

The latter are the Sun implementation classes for the RMI system. Our original estimation was that we may need to modify as many as a dozen or more classes. However, we were able to streamline our design such that only the following three classes were modified: `java.rmi.Naming`, `sun.rmi.server.UnicastRef` and `sun.rmi.server.UnicastServerRef`.

3.1.3 Design Choices and Decisions Made

During our implementation, several obstacles arose, which we have handled reasonably successfully. It is worth mentioning a few in order to give an idea of some systems issues involved.

- We considered three choices for inserting code that produces traces (i.e. hooks). The first choice was to modify the automatically generated stub and skeleton bytecodes using a bytecode engineering library. The second choice was to modify the RMI compiler, `rmic`, so that it can produce stubs and skeletons that included tracing hooks. Although both these choices were feasible, we decided to insert the hooks into the RMI runtime classes, thereby streamlining all the modification to the two of the three aforementioned classes. This, we think, was the cleanest and extensible mechanism.

- RMI protocol did not provide for a way to pass a logical timestamp from client to server upon request, and from server to client upon response. We therefore had to explore several options for passing the counter in such a way so as not to interfere with the normal protocol. We resolved this by modifying the protocol so that the receiving party (server or client) is expecting a logical timestamp during the recording phases before any communication, and the sending party correctly sends this counter.

- Default Java Serialization mechanism has some optimizations, which at first created problems for us in storing traces. Consider a situation when the same object is communicated between the client and the server for multiple RMI calls, and therefore needs to be traced at every call. When using Java Serialization with default settings, we had problems with the optimizations that it did – basically, upon successive serialization of the same object, the serialization mechanism would replace the original copy with the modified copy, effectively desctroying all the previous traces involving that object. We resolved this by requiring that serialization not modify objects already stored. This of course means larger traces – see discussion on memory usage and size of traces below.

- In order to allow our system to function, we had to modify a handful of classes that are part of the JDK. Also, we had to put some additional classes, those of our system, into the standard JDK. If this is not done carefully, it is possible to get runtime exceptions. The reason is the Classloader for classes that come from the Java language standard classes, including RMI, is different than the Classloader for User level classes. Hence, in order to make our system functional, we had to put all the classes of our system into the JDK, except for the bootstrap class, which had to stay outside of JDK, as that latter class need to invoke user level class – the RMI application.

- In order to correctly simulate the RMI runtime during replay, we had to ensure that replay engine itself does not deadlock. This system needs to ensure that only one event at a time is consumed, and that the correct thread consumes that event. Hence, we utilized a system of locks to ensure that (1) system itself does not deadlock, and (2) it provides for as much concurrency as possible.

One detail that we don't handle yet are user-level exceptions that occur during the recording phase. Basically, those are exceptions that the server throws to the client, stating that client did something wrong. We believe that these are errors that can be fixed somewhat easily. Also, with more time, we could incorporate

this into our system, allowing for this exceptions to be stored, and then successfully replayed, i.e. re-thrown during the recording phase.

It is worth mentioning that tracing is greatly helped by the standard Serialization feature of the Java language. Since in the recording phase, all the objects that were sent between two processes, were required to be Serializable by the RMI interface specification, it then follows that we can serialize objects for our own tracing as well.

4 Performance Evaluation

We considered the following criteria for evaluating our system -

- *Correctness of implementation:* This is critically important since our system should ensure that any bugs in the debugged program are not an artifact of our implementation, i.e. we do not *introduce* any new bugs in the system.
- *Functionality and Usability:* The system should be able to help the programmer uncover as many bugs as possible within its limitations. The system should have an intuitive and easily usable interface, so that it can be actually deployed in a real-world scenario. Also, programmer intervention should be minimal.
- *Overhead:* The memory and run-time overhead imposed by our system should not be prohibitively large.

4.1 Correctness

To demonstrate correctness, we tested a few simplistic distributed applications with our debugging tool. The framework for these experiments consisted of the following:

a. An RMI server implementing a handful of methods. Each method call resulted in printing out of the method name, the arguments received and the result.

b. A set of clients invoking some or all of these methods through RMI calls.

The test applications were:

- A single client making repeated random calls to functions implemented by the server. Correctness was established upon checking the outputs generated by the replayed system against the original execution. This validates that ordering of events from a single machine is preserved.
- Two clients making function calls to the server with random interleavings. This is a more complicated version of the previous test and validates that ordering of events across different machines is preserved.
- *Deadlock replay:* In this test, two client applications acquire two locks in different order, i.e. client *a* requests `lock x;` first and `lock y;` next, whereas client *b* does the reverse. Depending on the orderings of these events at the server side, the system may or may not deadlock. RTRE^{X} faithfully reproduced the effects in both cases.

4.2 Functionality and Usability

RTRE^{X} faithfully reproduces transient conditions in a distributed system allowing the application writer to discover bugs like data race conditions, etc. It does so cleanly by reducing the distributed debugging problem to local debugging! The system is completely transparent to application code and hence requires minimal programmer intervention.

RTRE^{X} can also act as a very useful automated test harness in the following ways:

- A programmer can force a component through specific or exhaustive orderings of events across the distributed system.
- While a component is being upgraded, the trace of execution of the older version can be fed through $\mathsf{RTRE}^{\mathsf{X}}$ to the new version to check if the new version is backward compatible.
- $\mathsf{RTRE}^{\mathsf{X}}$ can also act as a remote component simulator by caching results of a few executions. Thus testing and development can continue despite the unavailability of some remote components.

4.3 Overheads

In this section, we report timing, disk and memory overheads measured during the *recording* phase. Note that *replay* overheads such as time to start up the replay, etc. are also important but we did not measure them. Also, since a *replay* would be used in conjunction with a traditional debugger, we believe slowdown effects of our system would not affect the response times perceived the programmer.

4.3.1 Timing overheads

Here, we want to measure the extra time we add to the normal execution time. The total overhead, T_O, consists of two components:

(i) T_{dump} which is the overhead added due to the dumping of events, and

(ii) T_{send} which is the overhead added due to sending logical timestamps.

In general, our tests to measure overhead consisted of a single client making repeated and identical (fixed size requests) to the server. We measure the average time taken for normal execution without the recording happening in the background, T_{clean}. Then, we employ the recording phase and measure the total time, T_{record}, which is composed of T_{clean} and T_O. To study the composition of T_O, we measure the total time spent in writing the trace during the record phase, T_{dump}. We them compute $T_{send} = T_O - T_{dump}$. In general, we noticed that T_{send} was negligible and hence $T_O \approx T_{dump}$.

Also, in order to test whether the amount of overhead incurred is a function of the request size, we ran our tests with requests of various sizes. For example, in one set of tests which we label as Short, the client gets the server to repeatedly echo a fixed length string of size 100 bytes. In another set of tests, which we label Long, the server echoes strings of length 4000 bytes.

We also wanted to examine the effect of latency of network transfer (network propagation delay). To this end, we ran two different tests; one where the server and client ran on the same machine, which we call Local and another in which the server and client were run on different machines which we call Remote. In what follows we present results for the four cases (Local, Short), (Local, Long), (Remote, Short) and (Remote, Long).

The cases (Local, Short) and (Local, Long) We ran both the server and the client on the same machine `gs97.sp.cs.cmu.edu` running Windows NT for either case. For the (Local, Short) case, we had the client request short 100 bytes strings to be echoed by the server at regular intervals for a total of 10,000 times with and without $\mathsf{RTRE}^{\mathsf{X}}$ recording running in the background. We measure the average overhead per 10 calls, average over the 10,000 calls. Similarly for the (Local, Long) case, we have the client get 4000 byte strings echoed at regular intervals from the server for 1000 times and measure the average overhead per 10 calls.

For the Short case, we noticed that without $\mathsf{RTRE}^{\mathsf{X}}$ recording T_{clean} was nearly 5.0 ms per 10 calls. We saw that when recording was plugged on, we found that T_{record} was nearly 13.0 ms per 10 calls so that $T_O = 8.0$ ms. Of this overhead, though most was due to T_{dump}, the contribution of T_{send} was about 2 ms. The total overhead was about 150%.

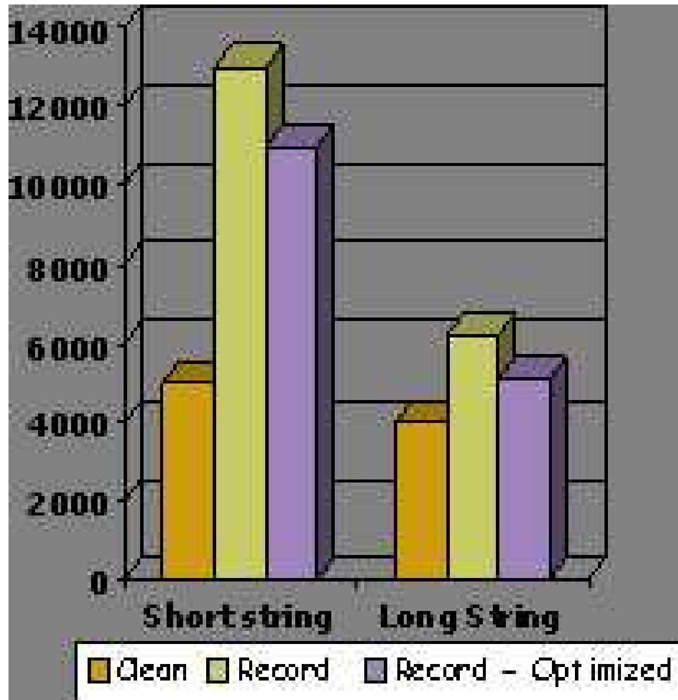


Figure 3: Recording Time Overheads (Local case)

Similarly, for the Long case, we had, $T_{clean} = 4.0$ ms and $T_{record} = 6.2$ ms so that $T_O = 2.2$ ms. Almost the whole of this overhead consisted of T_{dump}. So, the contribution of T_{send} was negligible. The total overhead was about 50%.

The overhead due to logical timestamps, T_{send} is higher in the former case for a very simple reason: the messages are about 100 bytes long to which sending logical timestamps adds another overhead of about 25 bytes.[3]

The cases (Remote, Short) and (Remote, Long) We ran the server on the same machine `gs97.sp.cs.cmu.edu` running Windows NT and the client on `gs276.sp.cs.cmu.edu` which ran Linux. The remaining parameters remain the same as above.

For the Short case, we observed numbers of $T_{clean} = 18$ ms, $T_{record} = 50$ ms, so that the overhead is more than 200%. Of the overhead of about 32 ms, we had $T_{dump} = 23$ ms and $T_{send} = 9$ ms. Again, the overhead of dumping vastly dominates, though the overhead of the transport of logical timestamps also increases almost proportionally.

For the Long case, we had seen $T_{clean} = 20$ ms, $T_{record} = 34$ms, so that the overhead is about 80-90%. While T_{dump}'s contribution to the overhead was about 13.5 ms, that of T_{send} was very negligible. The reasons for this observed behavior are identical to those pointed out above.

One point is worth mentioning here though. We expected the latencies for dumping to be symmetrical on the server and the client. However, we noticed that only when both the server and client are running identical OSes are the overheads similar. Besides, the machine running Linux always contributed an order of magnitude to the dumping overhead than that contributed by Windows (typical numbers were 11.5 ms for Linux and 2 ms for Windows). The reason for this, we realised after some analysis of our traces, was the following: the recording routines *flush* the dumped events to the trace file. Linux obediently flushes

[3] `sizeof(int)` + Java Serialization information overhead

whenever asked to do so while Windows buffers the flushes and does them at one go at regular intervals resulting in an amortization of the dump costs.

We optimized the dumping operation by buffering the events and flushing them out periodically by separate threads. This resulted in reductions of upto 50% in the overhead times. Note that we ran the optimized version only in the Local case with Windows NT as the operating system. The reduction is only 50% because Windows NT already buffers the writes into the filesystem before syncing to the disk.

The results show that recording approximately doubles the execution time. While expensive, we think the overhead is reasonable if we note that the main aim of the system is as a debugging aid rather than a performance critical production system.

4.3.2 Storage overheads

It is important to notice that the trace overhead in terms of size of the trace database will be quite large, as for each RMI event, we are taking a complete snapshot of the environment of the event. Hence, if the same objects were involved during multiple RMI interactions, for each interactions, we take a snapshot of the environment, effectively storing multiple objects several times.

We noticed that for every event recorded the Java system adds an overhead of about 500 bytes which in part is due to the expensive object serialization. This is quite significant for small messages.

4.3.3 Memory overheads

We performed some qualitative tests to understand the overhead that our system imposes on the virtual memory of the system. We ran a set of tests involving the RMI server and the client, wherein the client makes random calls on the server. Here is the scenario we ran:

- The server is started (*startup phase*),
- The client runs a large number of repetitive tests,
- The client stops and waits for several seconds,
- We repeat steps 2 and 3 for about a dozen times (*repetition phase*).
- When finished with the last test, we let the server running for a long time (*cooldown phase*).

While these tests ran, we observed the amount of memory used by the server process. We compare the results of clean and recorded execution.

As the tests are run, the virtual memory used by the server Java VM increases. The increases become smaller as we re-run the test several times. Eventually, after the cooldown phase, the memory used by the server process reduces to about the same level as it was before the test repetitions started.

During recording phase, the amount of virtual memory used by the server process was generally higher as compared to the clean case. However, we made two important observations: (1) the overhead memory was small, and (2) after the cool-down period, the virtual memory usage level was down to the level of the startup phase.

The results are consistent with the implementation of the recording phase. During recording, some memory is used to generate the traces. However, as traces are written into a file, and the references to these trace objects are abandoned, the java virtual machine garbage collector eventually cleans all the memory previously held by those objects.

Note that the measurements here are only approximate because the Java Virtual Machine does its own memory allocation. Taking precise memory usage measurements in Java requires special handling, including steps such as calling the garbage collector before and after critical calls to estimate the memory usage. We believe that while this could be a positive addition to our project, the qualitative results indicate the absence of serious, prohibitive overhead.

5 Conclusions and Future Work

Debugging distributed applications is difficult because reproducing transient conditions in a distributed system is hard. Our goal was to design and implement a replay based distributed debugger. We defined the ordering of RPC events as the level of abstraction for the replay, and we believe it is a reasonable compromise between simplicity and functionality. We reduced the distributed debugging problem to that of a local debugging situation by focusing on a single machine during the replay.

We implemented our tool RTRE^{X}, as a part of the Java RMI Runtime Environment. The resulting system is clean and completely transparent to user level code and requires minimal assistance from the developer. It also serves as a useful automated test harness.

While being a successful prototype, the system is in no way complete. Some of the ways in which it can be extended are:

- Handle user level RMI exceptions - this would make our implementation more complete and usable.
- Incorporate other paradigms/systems, e.g. Java Messaging Service, Enterprise JavaBeans, DEC RPC, etc.
- Integrate with replayers guranteeing faithful synchronization outcomes in the rare cases where our system cannot guarantee faithful replay.
- Integrate with other debugging aids such as data race detectors, deadlock detectors, etc.

References

[1] Peter C. Bates. Debugging heterogeneous distributed systems using event-based models of behaviour. *ACM Transactions on Computer Systems*, 13(1):1–31, 1995.

[2] R. Curtis and L. Wittie. Bugnet: A debugging system for paraller programming environments. In *Proceedings of the Third International Conference on Distributed Computing*, pages 394–399, 1982.

[3] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[4] T.J. LeBlanc and J.M. Mellor-Crummey. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[5] Neeraj Mittal and Vijay K. Garg. Debugging distributed programs using controlled re-execution. *ACM Principles of Distributed Computing*, 2000.

[6] Michiel Ronsse, Koen De Bosschere, and Hacques Chassin de Kergommeaux. Execution replay and debugging. In *Proceedings of AADEBUG*, 2000.

[7] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.

[8] RMI FAQ: `http://java.sun.com/j2se/1.3/docs/guide/rmi/faq.html`.